1967

Program relocation in a multiprogramming environment.

Stewart, James Jeremiah.

Monterey, California. U.S. Naval Postgraduate School

PROGRAM RELOCATION IN A MULTIPROGRAMMING ENVIRONMENT

JAMES JEREMIAH STEWART

PROGRAM RELOCATION

IN A MULTIPROGRAMMING ENVIRONMENT

by

James Jeremiah Stewart
Captain, United States Marine Corps
B.S., United States Military Academy, 1960

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ENGINEERING ELECTRONICS

from the

NAVAL POSTGRADUATE SCHOOL
June 1967

ABSTRACT

Various methods are studied for the relocation, or movement, including address mapping, of programs within a multiprogrammed digital computer. The aim of doing so is to determine the best method for use in the limited time-shared computing system proposed for development in the Digital Control Laboratory of the Naval Postgraduate School. In this light, the concepts of time-sharing and multiprogramming are discussed, as is the implementation of relocation in a very large computer obtained for the School's main computer facility. The features and requirements of the D.C.L. are then established and evaluated. It is found for the Laboratory that complete job swapping will be a fully satisfactory method of relocation. The time taken will not be excessive, and this method will be the easiest to incorporate in the time-sharing system. Details of a possible implementation are given in an appendix to the thesis.

TABLE OF CONTENTS

TABLE OF CONTENTS
(continued)

LIST OF ILLUSTRATIONS

LIST OF ILLUSTRATIONS
(continued)

TABLE OF SYMBOLS

F_a	-	Fractional increase in program execution time due to address mapping
F_e	-	Fractional portion of system overhead time due to program relocation
I	-	The time required for initialization or set-up of an input/output operation
K	-	1,024 computer words or bytes, as specified
L	-	The length, in words or bytes, of a program or program part being relocated
M	-	Computer main memory size in words or bytes
N	-	The length, in words or bytes, of the relocation program
T_a	-	Address mapping time
T_e	-	Program exchange time
T_{em}	-	The time, within program exchange time, required for actual input or output
T_r	-	Program relocation time
W	-	Gross transfer rate, in words or bytes per second, between main memory and some other storage device
f	-	Fractional portion of program instructions containing a memory address
k	-	The number of program pages
m	-	The number of memory cycles required to execute the average instruction in the computer under consideration
n	-	The number of non-overlapped memory cycles per input/output of one word or byte
q	-	Quantum; that amount of processing time assigned to one user during one turn of a time-shared computing system
s	-	The time within a quantum during which the user's program is actually executed

TABLE OF SYMBOLS
(continued)

t_a	-	The time required to map one program address into a physical location.
t_m	-	Computer memory cycle time
$( )_{BP}$	-	Subscript referring to the blocks and pages method of program relocation
$( )_{mm}$	-	Subscript referring to main memory
$( )_{NR}$	-	Subscript referring to the "no relocation" method of program relocation
$( )_{p}$	-	Subscript referring to a program page
$( )_{RR}$	-	Subscript referring to the relocating register method of program relocation
$( )_{2s}$	-	Subscript referring to secondary storage; $( )_{3s}$ refers to tertiary storage, etc.
$(\bar{})$	-	Overbar implying the average value of the indicated parameter

TABLE OF ABBREVIATIONS

CDC	-	Control Data Corporation, Minneapolis, Minnesota
CODAP	-	Control Data Assembly Program
D.C.L.	-	Digital Control Laboratory, a facility of the Department of Electrical Engineering, Naval Postgraduate School
FWA	-	First Word Address
GE	-	General Electric Company, Information Systems Division, Phoenix, Arizona
IBM	-	International Business Machines Corporation, Data Processing Division, White Plains, New York
NPGS	-	Naval Postgraduate School
PDP	-	Programmed Data Processor, a name given to computers manufactured by the Digital Equipment Corporation, Maynard, Massachusetts
PST	-	Program Status Table
SDS	-	Scientific Data Systems, Inc., Santa Monica, California
CRT	-	Cathode ray tube

1. Introduction.

This thesis studies the problem of the relocation of programs within a multiprogrammed digital computer. The objective of doing so is to determine the most suitable method of relocation to be applied in the limited time-sharing system proposed for development in the Digital Control Laboratory of the Naval Postgraduate School.

The plan of the thesis is to progress from general consideration of background matter to specific investigation of the D.C.L. system and its requirements. The following subjects are treated:

a) a brief survey of the meaning and potentialities of the end application, time-shared computing;

b) consideration of the general multiprogramming environment and the need for program relocation;

c) study of a number of techniques of relocation;

d) investigation of the relocation method implemented in a very large time-sharing computer, the IBM[1] System/360, Model 67;

e) study of the Digital Control Laboratory's requirements and features of its new SDS 930-centered computing system; and

f) recommendation and conclusion.

The primary investigative tool used is comparative analysis, as, for example, of different techniques of program relocation. Expository discussion is interleaved with consideration of advantages and disadvantages.

The following section introduces time-shared computing.

[1]International Business Machines Corporation. The meaning of all abbreviations used in this thesis is given in the table on page 9.

2. Time-Shared Computing.

A major impediment to the full use of digital computers has been expressed as the "speed-cost mismatch"[2] between man and machine. Computers are very fast, but expensive. Men, relatively, are slow, but their time is cheap. One result of this mismatch has been a tendency to hand the machines over to the group of computer professionals - programmers, operators, and managers - who know best how to keep them busy. The real users of computer power - the professors, executives, colonels, and generals - are, in the main, isolated from direct contact with the machines. No one would suggest that the professors and colonels become full-time programmers or operators; but there are many problems where time and meaning are critical, where the isolation of the real users is a distinct disadvantage.

One answer to the speed-cost mismatch, and to the matter of letting the real user have direct contact with the machine, is time-shared, multiple-access, on-line[3] computing. Here a computing system is designed so that a number of different, possibly distant, users have concurrent, real-time access to it. The speed of the computer is put to good use in moving between each user's tasks, solving them at a rate which, in a well-designed system, approaches that of human reaction. The expense of operating the system may now be spread over its many concurrent users. Because there are relatively natural programming languages available, and because the means of access to the computer can be an easily-employed device (teletypewriters and CRT displays with typewriter-like keyboards

[2]Licklider, J.C.R., Man-Computer Symbiosis (IRE Trans. on Human Factors in Electronics, March 1960), p. 7.

[3]For the sake of brevity, all these adjectives will be implied when, henceforth, only "time-shared" is written.

are most common), direct contact of users is facilitated.

Time-sharing offers several other interesting possibilities. One of these is in the area of formulative, or trial-and-error, problem-solving. Since computers follow only the steps for which they have been programmed, they have been most useful heretofore in solving completely pre-formulated problems, using pre-determined procedures. Now, with time-shared computing, the user with a less-well-understood problem has the opportunity to sit at his access terminal and to interact with the machine on an almost conversational basis. If there is a solution to his problem, he may be able to "feel" his way to it.

There is, with time-sharing, an advantage in the management of an information base. Only one, central file need be maintained, as its contents may be made accessible to all authorized users at their terminals; further, once a user enters data into the system, it is immediately and identically available to other intended subscribers.

Time-sharing can extend the power of a large computer. This is its major superiority over a proliferation of independent small machines. For the same computing power and number of users, a time-sharing system with reasonable communications costs appears to be less expensive than a system of independent machines.

Perhaps the most interesting possibility of time-shared computing is the concept of a "computer utility". That is, like water or electricity, computing power would be furnished from a generating element (here, the central computer) to locations where it can be used, there to be "turned on" (employed) when needed and "turned off" when finished.

The number and vitality of current applications of time-sharing demonstrate that this means of computing is quite practical. One

compilation lists 40 installations.[4] Educational institutions with time-sharing systems include Stanford, California at Berkeley, and the Massachusetts Institute of Technology. The Naval Postgraduate School will soon join this group, not only with its D.C.L. system but also through new equipment being installed in the central Computer Facility. Commercial systems are becoming quite numerous. Many of these are available anywhere there are telephones, for they employ the telephone lines to link remote terminals to the computer. Shown in Figure 1 is the equipment used in one such commercial time-sharing system.

It is not difficult, finally, to imagine military applications. For example, the possibility of formulative problem-solving might be as valuable to a military research organization as to a similar civilian enterprise. In a large supply center or personnel directorate, time-sharing's advantage in management of a centralized information base could be useful. Because the remote user's terminal may typically be a lightweight teletypewriter, linked by radio or wire to the computer, time-sharing may be feasible even on the battlefield. A tactical system could serve as a message processor, handling battle reports, logistics status, etc., and also as a means for rapid computation at diverse locations of such time-consuming problems as aircraft schedules and embarkation tables. An added advantage here would be that when a using unit was not in action, and its computing requirement was therefore small, an expensive computer would not be idled; only a terminal would not be in use.

[4]Time-Sharing System Scorecard, No. 4 (Computer Research Corporation, Fall 1966).

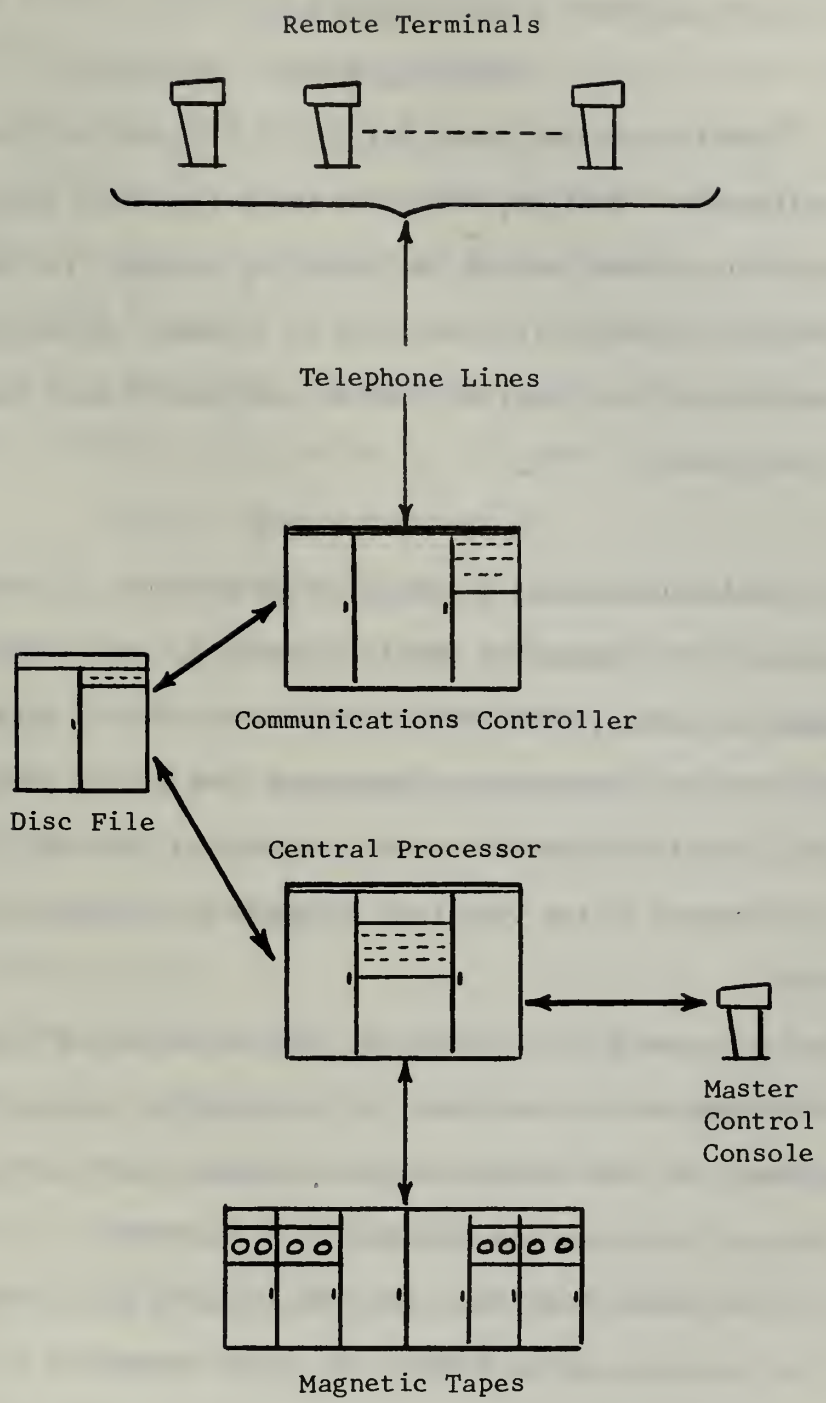


Figure 1. Configuration of a commercial time-sharing system; adapted from that of the General Electric Company

3. Multiprogramming and Program Relocation.

Multiprogramming - Definition

The time-shared computing just introduced implies a multiprogramming environment. That is, more than one active user program will simultaneously be present within the computing system. The processor must be operated to permit the execution of a number of programs in such a way that none of the programs need be completed before another is started or continued.

Goals

Multiprogramming a computer, for whatever application, may be done with any of a number of possible system goals in mind. One of these might be termed, "improvement in user service". Possible sub-goals to this include reduction in turnaround time and an increase in the number of allowable concurrent users. In general, pursuit of this goal reflects an awareness of the view that a computer is properly the servant of its users.

A second goal, conversely, aims at realizing the greatest possible efficiency in the employment of the physical components of the computing system. It thus recognizes that computers are very expensive machines. This goal stresses the achievement of economy.

The third, and final, multiprogramming goal to be considered here is a variation of the first. It can be expressed as "improvement in service to all users, but with special emphasis on the needs of some". That is, certain users or user classes would be favored, with the system responding preferentially to their requests. Presumably, these special users would be either those whose needs so required or those whose equipment permitted them to take advantage of their favored position. An

example of the former might be a hybrid system simulator, who typically requires a definite amount of digital computing service at fixed time intervals. He must have this service at the prescribed times if his simulation is to function at all. The latter could include persons using an on-line display console to interface with the computer. The relative problem-solving power of a good display, properly backed with necessary software, compared to many other input/output media, is so great as to probably warrant favored consideration to its users.

Problems of Multiprogramming

The designer of a multiprogrammed computer must solve a number of rather special problems. In general, these problems are either not found, or are experienced in much less severe degree, in other forms of computing. Their solutions seem particularly critical in the time-sharing application, where the human user, at his terminal, is immediately awaiting answers from his programs.

These problems include:

a) scheduling - the order in which the different programs actively present in the system will be served must be determined;

b) input/output communications - messages to and from system users, programs and answers, must be handled;

c) memory allocation - programs must be dynamically assigned to and within the different levels of system storage;

d) security - necessary isolation must be provided between different users' programs, and between a user and a system program not his to use;

e) system monitoring - the complexity of multiprogrammed operation often warrants special consideration for the task of monitoring

system functioning and accounting for the charges to each user; and

f) program relocation, which is the subject of this thesis.

Program Relocation

In general, in the on-line multiprogramming implied here, it is not possible to process a program through to completion in one "turn" of the system. That is, the requirement to provide a sufficiently brief response time to each user - to receive his program or to provide some answers - necessitates the interruption, before completion, of all but the briefest processes. Further, it is disadvantageous to try to allow interrupted programs to remain, unaltered, in main memory. Such practice is probably impossible, in view of the unforeseeable requirements of subsequent users' programs, and to attempt it would certainly result in a severe limitation upon the number of allowable concurrent system users.

Thus there is a need for the movement of programs about the computing system as their status changes. It is this movement which is called, in general, program relocation. Some examples follow. A program which at one instant of time resides in main memory for purposes of active computation may, a few moments later, be placed for temporary storage on a drum or disc file. Conversely, a routine stored on magnetic tape may, at some time, be called into core for processing. Or, a data area may be required by a running program when, as the result of prior relocation, the two are in different parts of main memory.

Another way to consider program relocation is to realize that to function properly, the computing system must be able to access any program in the system at any time. No program can ever become "lost" to the central processor and its operating system. Thus program "movement" requires a consideration of "access" methods. In studying relocation,

this thesis must then investigate the addressing requirements of multi-programmed computations. Addressing methods, in fact, are at the center of the topic of relocation, for they have a direct and important effect upon the speed and efficiency of the computer.

The following section begins this study by considering a number of relocation techniques.

4. Techniques of Relocation.

First Considerations

System Goal and End Application. Some possible goals of multiprogramming were previously given. The goal which is chosen for a system must be kept in mind when weighing the relative merits of different relocation techniques. The end application of the computing system may also influence the choice of relocation method. As indicated before, the end application important to this thesis is time-shared computing.

Size of Main Memory. Usually, in the multiple-access, on-line computing implied here, main memory will be insufficient in size to hold all active computations simultaneously. This is in contrast to the special purpose Naval and Marine tactical data systems; there, the total quantity of stored program is known, and because of the real-time requirements of the systems, main memory holds it all. Here, the system works under a varying program load. Further, there will be a need to store some programs in an inactive status. Thus considerations of overall economy suggest a main memory limited in size, so that it cannot be expected, in general, to hold all processes at once.

Single-Level Store. With the consequent use of several different storage media, there has developed the concept of the "single-level store".[5] Since for a user, it would be difficult, if not impossible, to keep track of where his program resides at any moment, he is not expected to do so. Instead, the operating system records and uses this information, while the user codes as if his programs were always in main memory. To him, the system does not appear to have its actual hierarchy

[5]Kilburn, T., et al., One-Level Storage System (IRE Trans. on Electronic Computers, April 1962), p. 223.

of different storage media, such as core, disc, and tape; he sees it as possessing one "single-level store".

Address/Location Map. It may be desirable, for greater flexibility, that the system not be required to place a program in the same region of main memory each time it is called up. To so require would incur extra overhead upon program exchange and would complicate the queueing of waiting processes, although it may be justified for other reasons. Thus a means is needed to relate the addresses used by a computation to the physical locations in main memory actually employed for storage. This means may be called, after Dennis[6], the "address/location map". It may be considered to effect a translation from an address, or "name", space to a location space. This thesis will often speak of program relocation in terms of methods of creating and maintaining this "map".

Memory Protection. As suggested previously, proper isolation between different users' programs, or "memory protection", must be a part of any multiprogrammed computing system. It is not difficult to think of pertinent reasons. In a commercial system, one business user must not be permitted access to another firm's secrets stored in the computer. In a military application, classified information must be protected. Memory protection is needed in any situation because new programs, which often contain errors, must be prevented from interfering with other processes in the system. Because the form of memory protection provided is often affected by the choice of relocation technique, memory protection will be treated as a secondary subject in the remainder of this thesis.

[6]Dennis, J.B., Segmentation and the Design of Multiprogrammed Computer Systems (Journal of the ACM, October 1965), p. 590.

System Evolution. One lesson learned by designers in recent years is that provision must be made for evolution in the design of any computer system. Changes in technology and applications occur too rapidly for this not to be so. There are numerous ways to provide for system evolution; whichever seem appropriate, any method of handling the relocation of multiprogrammed computations must be judged partly upon its ability to evolve.

Quantitative Aspects

There are two very useful parameters which may be measured in the evaluation of a relocation technique. These parameters are:

a) the physical size of the relocation program;

b) the time required to effect relocation.

The size of relocation coding is important because it represents a demand upon a key system resource, storage. When a new computer system is being planned, consideration of the probable size of the relocation program may affect the amount of storage specified. In an existing system, the otherwise available amount of system storage is reduced by the size of the relocation code.

It is reasonable to measure the size of the relocation program in terms of main memory computer words or bytes, whichever is appropriate to the particular computer under consideration, since it is in main memory that the code will reside while being executed. The number of words or bytes required is called here, N. In many cases, not all of the relocation program need be in main memory at all times. For reasons of greater economy and space-saving, lesser-used relocation routines are often placed in other, cheaper storage. Of course, the disadvantage of doing so is the greater access times to such routines. In general, the

size of the relocation program may be expressed as

$$N = N_{mm} + N_{2s} + N_{3s} + \ldots \tag{4.1}$$

where N_{mm}, N_{2s}, N_{3s}, ... refer to the main memory, secondary, tertiary, ... storage used, these amounts being specified in terms of main memory words or bytes.

The value of N depends upon the features of the instruction set of the computer under consideration and upon the relocation technique used. For a smaller N, efficient table-search and powerful input/output instructions are necessary, as these are found to be the major tasks of relocation. Also, as to be expected, the more complex the relocation technique, the longer the implementing code. In general, considering the normal size of such necessary components as a loader, many relocation programs occupy one thousand or more computer words.

Especially critical in a time-sharing application, where human users are waiting for answers at their terminals, is the time taken for program relocation. Even though a useful function is being performed, relocation time is all overhead, non-productive in terms of actual execution of user programs. This time may be considered in two ways:

a) the absolute amount required;

b) the relative effect, first, in terms of increased program execution time due to address translations, and second, as a contribution to total system overhead.

In the absolute measurement, relocation time will be denoted here as T_r. It will often be convenient to measure T_r over one user's assigned time-slice, or quantum, q. There are two major contributions to relocation time. These are the time required for address mapping, T_a, and the

time taken for program exchange (the input or output of code), T_e. Thus

$$T_r = T_a + T_e \tag{4.2}$$

Further, it may be seen that

$$T_a = t_a L f \tag{4.3}$$

where t_a is the time required to translate one address, L is the length of program under consideration, and f is the fraction of program instructions containing a memory address. T_e will be discussed elsewhere in this section of the thesis.

For relative measurements, it is possible to express the fractional increase in program execution time due to address mapping, called here F_a, as

$$F_a = \frac{t_a}{mt_m} f \tag{4.4}$$

where m is the number of memory cycles required to execute the average instruction in the computer under consideration, t_m is the computer's memory cycle time, and t_a and f are as defined above. Obviously, mt_m may be replaced by the average instruction time of the computer, if that is known directly.

If s is the time within q during which the user's program is actually executed, then $(q - s + T_a)$ is the total overhead time within a quantum. F_e is defined to be the ratio of relocation overhead to this total overhead. Then

$$F_e = \frac{T_a + T_e}{q - s + T_a} \tag{4.5}$$

With some relocation techniques, T_a is zero or is small enough to be negligible. In this situation, the above expression becomes

$$F_e = \frac{T_e}{q - s} \qquad (4.6)$$

Any or all of these expressions (4.1)-(4.6) may be usefully evaluated when comparing different methods of program relocation. Some will be discussed further and used in examples in the remainder of this section.

Consequences of "No Relocation"

Suppose that the address/location map consists of a one-for-one translation of addresses into physical locations; that is, the addresses are always the same as the locations, and a "no relocation" (within main memory) situation exists. Each program is assumed to have full use, outside the resident portion of the operating system, of the possible addresses in the computer. In such a case, dumping of information from main memory is often required when, during processing, one program is interrupted and another started or resumed. This is so because:

a) the new program may require more locations than are left free by program(s) now in main memory; or

b) even if the required quantity of locations is available, there may be duplication in the addresses (= locations here) used. (See Figure 2.)

In special cases, where the total naming requirements of all processes are less than the number of addresses available, this frequent dumping of information may be avoided. Then, the different programs may be allocated to separate portions of memory, and relocation is never necessary. This is the situation with the previously mentioned Naval and Marine tactical data systems. This is not generally the case, however, due to changing total demand, in the time-shared computing discussed here.

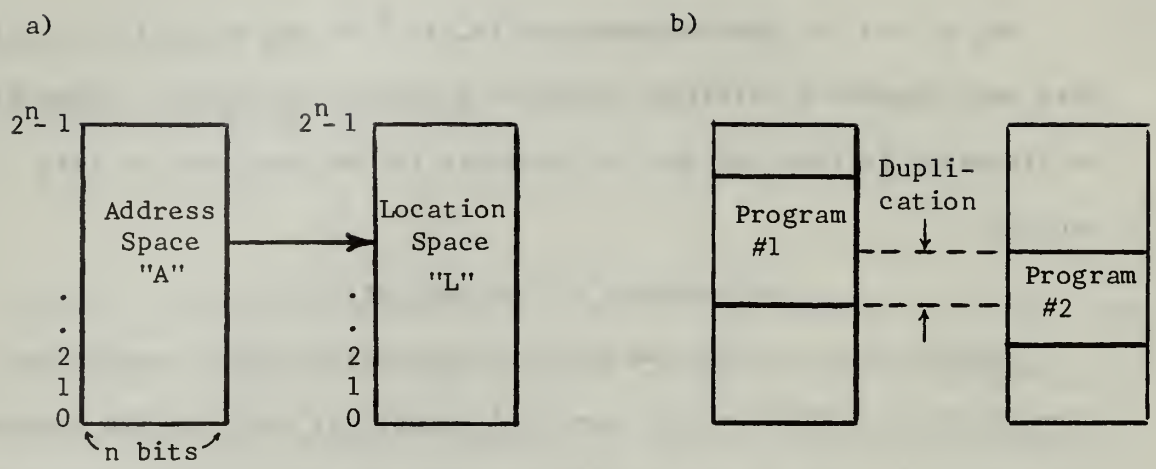


Figure 2. a) No relocation within main memory

b) Duplication of addresses

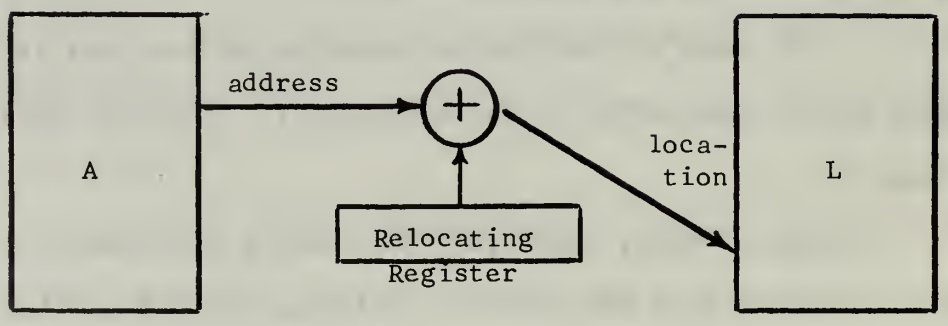


Figure 3. Use of a relocating register

"No relocation" is sometimes known as "job swapping", although such terminology is rather loose. Often only that portion of previous job(s) necessary to provide sufficient space for the incoming program is swapped.

The new program, when the exchange is complete, is normally "run from zero", i.e. started at some fixed location outside the resident portion of the operating system. In such event, the only main memory protection required is to ensure that the program does not access above its upper bound. Perhaps the fastest way to implement this would be by adding an additional hardware register. During program exchange, this register would be loaded with the new process' upper bound. Its wiring would be such that during the running of a user program, each memory access would produce an automatic "compare" with the register's contents; the occurrence of an access violation would result in a "no operation" on the access and a trap to a designated location. The isolation provided here between programs is complete, with neither reading nor writing outside one's own process permitted.

To establish the timing of "no relocation", it is first noted that since there is no address mapping,

$$t_a = T_a = 0$$

Thus

$$T_r = T_e \tag{4.7}$$

It is useful to divide T_e into two parts. The first, called I, is defined as the initialization or set-up time required before the movement out of, or into, main memory of a user program is actually initiated. This time is used to perform tasks such as searching memory tables for a user program's length, storage location, first word address, and other

quantities necessary to define the input/output operation. The second part of T_e is the time taken for the actual input/output transfer, called here T_{em}. Thus, in general,

$$T_e = I + T_{em}$$

For a "complete" relocation, i.e. one input and output movement, as during a quantum,

$$T_e = 2(I + T_{em}) \tag{4.8}$$

The initialization time varies, of course, with such factors as the computer's instruction features and speed, and the number of system users. It is also dependent upon the exact "no relocation" technique employed. It will be smallest for the simplest application, complete job swapping.

The actual input/output time, T_{em}, depends upon the characteristics of the computer under consideration. It is most useful to consider as T_{em} only that input/output time not overlapped with other processor functions. Then

$$T_{em} = 0$$

if the transfer is made on a path completely independent of processor action,

$$T_{em} = nt_mL$$

where n is the number of non-overlapped memory cycles per word or byte transferred, and L is the number of words or bytes being transferred, if the transfer occurs on a cycle-stealing channel, and

$$T_{em} = \frac{L}{W}$$

where W is the gross transfer rate of the secondary storage device used, if the transfer halts all processor action while it is taking place.

In relative measurements with "no relocation",

$$F_a = 0$$

because t_a is zero, and

$$F_e = \frac{2(I + T_{em})}{q - s}$$

per quantum.

Despite the large amount of program movement into and out of main memory, this method has proven to be quite usable in practical systems. It is the technique employed by the General Electric Company's commercial time-sharing system. Further, it was used for several years at Project MAC of the Massachusetts Institute of Technology.[7]

Use of a Relocating Register

An improvement in a technical sense over "no relocation" is use of a relocating register, which permits translation of a contiguous set of addresses in name space to any contiguous set of physical memory locations. During the exchange of programs into and out of main memory, the new program is stored in any convenient set of locations. Thus less dumping is required than with "no relocation", where the new program is always loaded starting at the same location. The mapping is effected during program execution by adding the proper constant, stored in the relocating register, to each accessed program address. Thus occupation of a different part of location space during processing is permitted with

[7]Saltzer, J.H., Compatible Time-Sharing System Notes (Project MAC, Massachusetts Institute of Technology, 1965), pp. 31-36.

no changes in the addresses actually contained in a program. Again, the fastest implementation would be to provide the register and addition in hardware. Some foresight is needed in the design of the computer word, however, for a means must be provided so that the register does not operate on program instructions which do not reference a memory address. (Figure 3.)

This method can be considered to be an extension of the relocating loader in non-multiprogrammed batch-processing. There, address translation occurs only on loading; there is no provision for relocation during execution. By contrast, a relocating register effects repeated translations during execution.

Memory protection may be provided here by using "bounds registers". Such registers would contain, for the running program at any moment, the current upper and lower physical memory locations. Any attempt by the program to access outside these bounds, or to alter the bounds registers, should result in a protection trap. Again, the isolation between different programs will be complete.

When a relocating register is used, in contrast to "no relocation", $t_a \neq 0$. Therefore, in general, $F_a \neq 0$ and $T_a \neq 0$. That is, a finite address mapping time is required, and program execution time is thereby increased. A reasonable range to be expected for a hardware-implemented t_a is from 20 to 200 nanoseconds. This is the time required to sense that relocation is needed and to add the contents of the relocating register to the program-contained memory address. The effect upon program execution time is, by (4.4), directly proportional to t_a and inversely so to t_m. Thus the effect of address mapping time is greater in a faster computer. For an example, choose, as reasonable values,

$$t_a = 100 \text{ nanoseconds}$$

$$m = 3$$

$$f = \frac{2}{3}$$

Then, if t_m is three microseconds,

$$F_a = \frac{t_a}{mt_m} f \tag{4.4}$$

$$= \frac{0.1}{3(3)} \frac{2}{3}$$

$$= 0.0074 \text{ or about } 0.7\%$$

But if t_m is one microsecond,

$$F_a = 0.022 \text{ or } 2.2\%$$

which is, of course, three times as great. Plotted as Figure 4 are the variations of F_a with t_a and with t_m, as given by (4.4). The values of m and f are chosen as above. It can be seen that for the ranges shown of t_a and t_m, the maximum F_a is about 0.05 or 5%. Such an increase in execution time may or may not be of consequence in a particular computer system.

If some assumptions are made, it is possible to make a direct comparison between the "no relocation" and relocating register methods in terms of the average time required. The independent variable will be L, the average length of program being relocated. For "no relocation", it is recalled that (with subscripts now added, for clarity)

$$(T_r)_{NR} = (T_e)_{NR} \tag{4.9}$$

in general, and

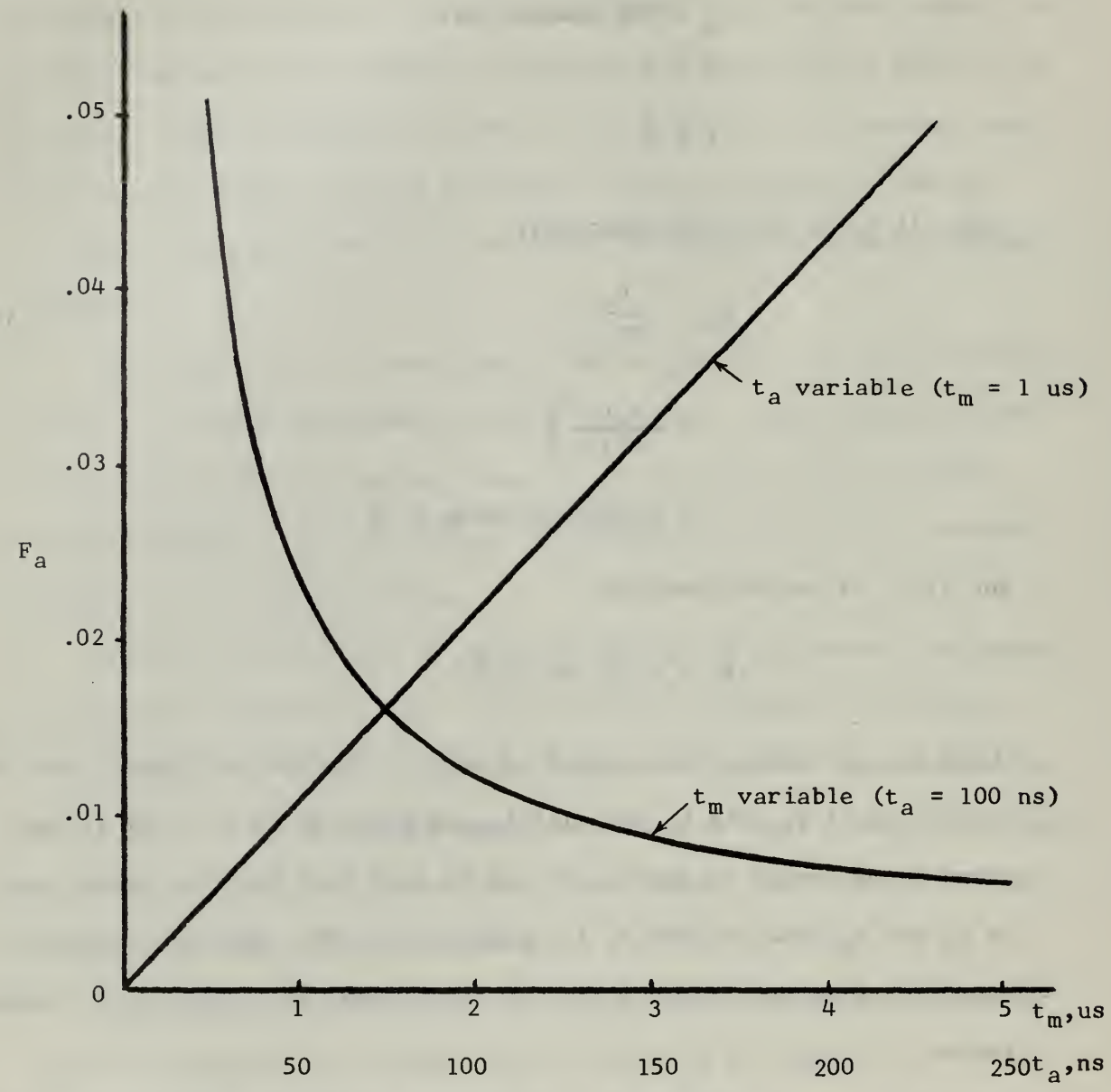


Figure 4. Increase in program execution time due to address mapping

$$(T_r)_{NR} = 2\left[I_{NR} + (T_{em})_{NR}\right] \tag{4.10}$$

per quantum. It is assumed now that the initialization required with use of a relocating register will take three times as long as that with "no relocation". That is,

$$I_{RR} = 3I_{NR} \tag{4.11}$$

This relation seems reasonable, considering the greatly expanded amount of main memory management which is required when a relocating register is used. It is also assumed that

$$(T_{em})_{NR} = nt_m L \tag{4.12}$$

That is, the secondary storage device to be used in relocation is connected to a cycle-stealing input/output channel which permits partial overlap of program exchange time with other processing. This particular assumption is not critical; a non-overlapped channel could just as well have been used. $(T_{em})_{RR}$ will normally be less than $(T_{em})_{NR}$, because use of the relocating register allows more than one complete program to be in main memory at one time. In fact,

$$(\overline{T}_{em})_{RR} = \frac{\overline{L}}{M} (\overline{T}_{em})_{NR} \tag{4.13}$$

where $\frac{\overline{L}}{M}$ is the fractional portion of main memory occupied by the average user program. The meaning of this expression is that if, for example, the average user requires one-eighth of main memory, then program movement time with use of a relocating register will be, in sum, one-eighth that taken with "no relocation". This is so because, on the average, eight user programs can reside in main memory at one time when a relocating register is employed, and when control transfers from one of these

programs to another, $T_{em} = 0$. Now, by (4.2)

$$(T_r)_{RR} = (T_a)_{RR} + (T_e)_{RR}$$

and, per quantum, using (4.8)

$$(T_r)_{RR} = (T_a)_{RR} + 2\left[I_{RR} + (T_{em})_{RR}\right] \tag{4.14}$$

Substituting (4.3), (4.11), (4.12), and (4.13), and using average values, (4.14) becomes

$$(\bar{T}_r)_{RR} = t_a\overline{Lf} + 6\bar{I}_{NR} + 2\frac{nt_m\bar{L}^2}{M} \tag{4.15}$$

Substituting (4.12) into (4.10), and again using average values,

$$(\bar{T}_r)_{NR} = 2\bar{I}_{NR} + 2nt_m\bar{L} \tag{4.16}$$

These expressions, (4.15) and (4.16), represent in comparable terms the average times for program relocation with a relocating register and with "no relocation". To demonstrate their different variation with $\bar{L}$, average program length, the following values are chosen:

t_a = 100 nanoseconds

$\bar{f} = \frac{2}{3}$

$\bar{I}_{NR}$ = 500 microseconds

n = 2

t_m = 2.5 microseconds

M = 8000 words

These values are reasonable for a smaller, medium speed system. The results are:

$$(\bar{T}_r)_{RR} = 6.7(10^{-5})\bar{L} + 3 + 1.25(10^{-6})\bar{L}^2$$

$$(T_r)_{NR} = 1 + 10^{-2}\,\bar{L} \quad \text{(milliseconds)}$$

These two expressions are plotted as Figure 5, for $0 < \bar{L} \leq 8000$. The superiority of the relocating register method is clearly shown for all $\bar{L}$ except:

a) $\bar{L} \leq 200$, where the greater initialization required with the relocating register is the dominant effect;

b) $\bar{L} \geq 7800$, i.e. $\bar{L}$ close to M, where, even with the use of a relocating register, an input/output transfer is necessary upon almost all program exchanges.

The disadvantage of the relocating register method of program relocation results from the fact that a contiguous set of addresses is always mapped into a contiguous set of physical locations. This tends to create overhead during program exchange when suspended programs in main memory must be moved up or down solely to provide a sufficiently long set of free locations for an incoming process.

Nevertheless, the relocating register has also proven to be a feasible technique. It has been successfully tested at Project MAC.[8] Its relative simplicity of implementation is an attractive feature. Where program exchange is a frequent occurrence, however, this method appears to contribute a significant amount of system overhead.

Blocks and Pages

A method of avoiding the contiguity problem is to divide main memory into increments, called "blocks", and to divide programs into "pages". All blocks and pages are of the same fixed length. Pages may also be considered to be divided further into "lines", a line being actually a word or a byte. When translating an address, the address/location map

[8]Corbato, F.J., System Requirements for Multiple-Access, Time-Shared Computers (Project MAC, Massachusetts Institute of Technology, 1964), pp. 6-7.

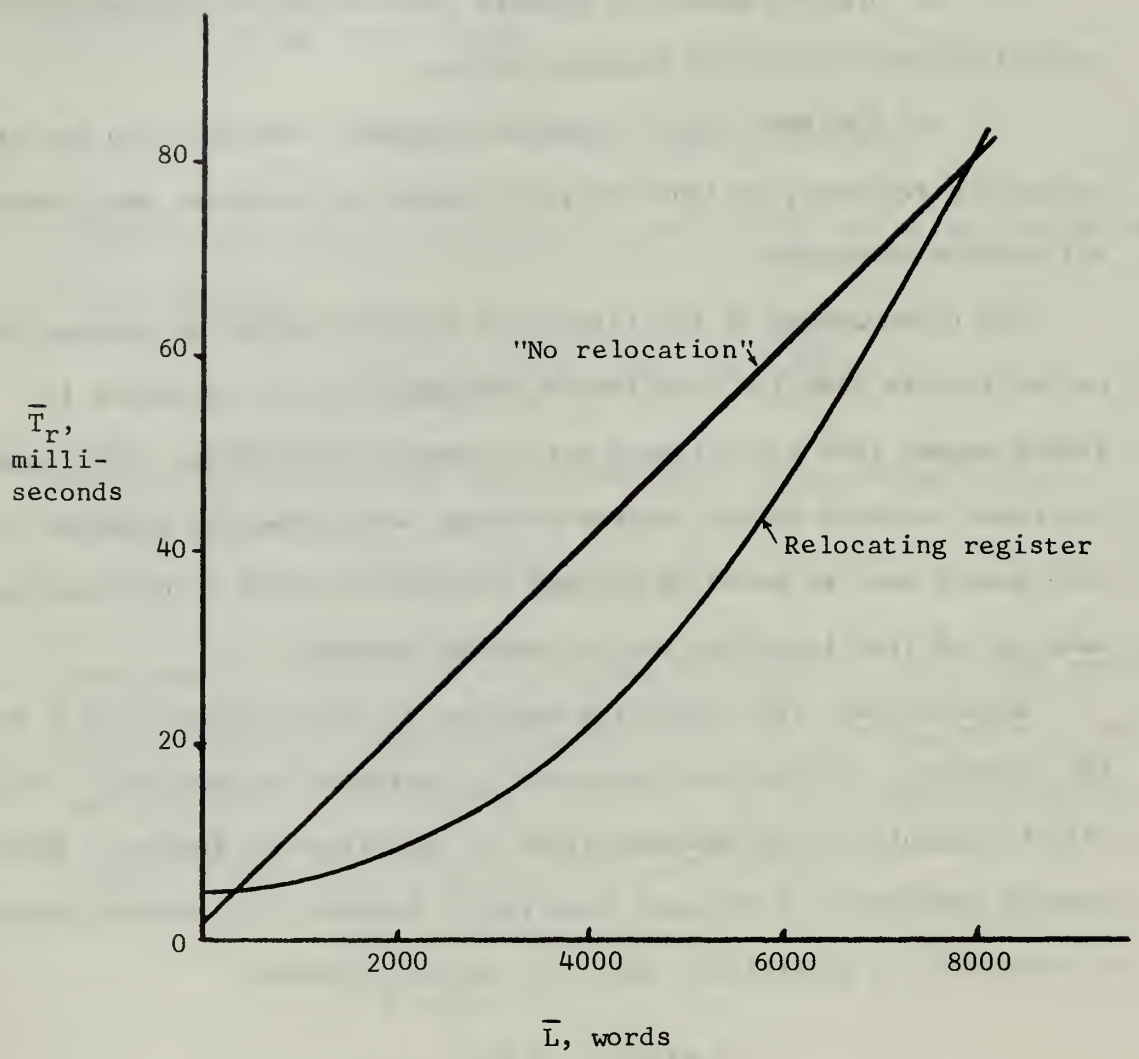


Figure 5. Relocation time vs. average program length, with "no relocation" and relocating register methods

must relate the referenced page to its current physical block in memory. There is no requirement for contiguous pages in a program to be located in contiguous memory blocks. Also, there is no need for address space to be of the same size as location space in main memory; the former may well be larger, as long as the operating system knows which pages are in main memory blocks, and which are in other storage, at any time.

An added advantage of using blocks and pages is that it now becomes convenient to call into main memory only those parts - pages - of a program which are currently active. Of course, an algorithm is needed to judge activity and to decide which pages to bring in during program exchange. Thus relocation overhead will be further reduced, beyond the reduction offered by the removal of the contiguity requirement. It may also be observed that the paging of a process is not a matter of concern to the applications programmer. Pages are fixed-length subdivisions which may occur at arbitrary points in a program; they are not subroutines. Paging is effected in the operating system and is invisible to the general system user.

One straight-forward implementation of blocks and pages creates a table for each active program in the operating system. This table associates each page of a user program with its current physical block or other location in memory. The look-up is made on the page number, obtained from the referenced address in the program, with the result being the location. There is no translation of the line, which is assumed to occupy the same relative position in both page and block. (See Figures 6 and 7.)

Memory protection may be provided by the association of one or more bits with each block in the block-page table; these bits indicate the

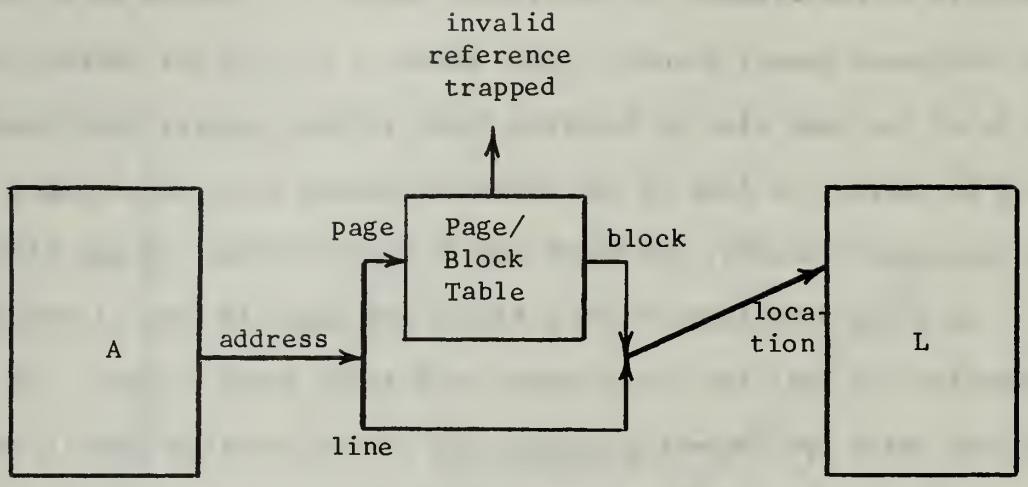


Figure 6. Address translation using blocks and pages

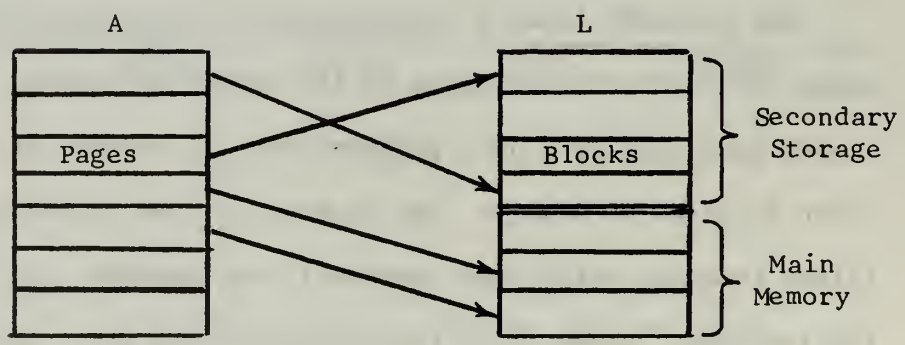


Figure 7. Non-contiguous storage of program pages in memory blocks

type of access which the program is to have to this block. If more than one bit is allocated, several forms of protection, such as "read only", "write only", and "no access", can be identified.

As implemented above, the blocks and pages method of relocation imposes a time penalty for its use. Extra memory cycles are required during processing to make the table look-ups for the address translations. This is a serious disadvantage, and to reduce it, current computers employing blocks and pages perform the mapping with hardware. The SDS 940, for example, provides two extra registers which contain block locations applicable to the program in execution. The wiring is such as to replace the upper, or page-indicating, bits of a program-referenced address with the appropriate block location before the fetch, store, or branch specified takes place. Two larger computers, the IBM System/360, Model 67 and the GE 645, incorporate an associative memory element within the central processor. In this element are stored, for the running program, the page-block combinations of a number of high-use pages (perhaps these would be the most recently referenced ones). When an address is referenced, a fast parallel search of the element is made; if the page is present, its block is then immediately known.

As with the relocating register method, $t_a \neq 0$ when blocks and pages are used. However, the possible values of t_a now vary over a wider range. If the address mapping is performed in hardware, t_a will be of the same magnitudes as for the relocating register, and Figure 4. applies. However, if programming is used to translate addresses, t_a may be many times t_m. This, of course, would lead to much higher values of T_a and F_a. This matter will be pursued further in Section 5 of this thesis, where an example implementation of program relocation using combined hardware-software address mapping is presented.

Generally speaking, the blocks and pages method has been incorporated in newer computers, in which there are usually available input/output channels which operate independently, during actual transfer, of processor functioning. In this case

$$T_{em} = 0$$

provided only that there exist other tasks for the processor to perform while the transfer is taking place. The only contribution to program exchange time, then, is the required initialization time. Per program page used, this time may be called I_p. If k is the number of pages needed during the measurement interval, then

$$(T_e)_{BP} = kI_p \tag{4.17}$$

Recalling (4.2) and (4.3), the total relocation time becomes

$$(T_r)_{BP} = t_a Lf + kI_p \tag{4.18}$$

This quantity may be compared to the times required with use of "no relocation" (4.8) and with use of a relocating register (4.14) over the same measurement interval. L, in (4.18), is to be interpreted as the length of program executed over the interval. While, in general, it will be related to k, the number of pages used, these two quantities may not be strictly proportional. L may not increase as rapidly as k, because the use of more pages in the measurement time suggests that fewer instructions are being executed from each page. Figure 8 plots (4.18) against k for the following relations between L and k:

$$L \sim k$$

$$L \sim k^{2/3}$$

$$L \sim k^{1/3}$$

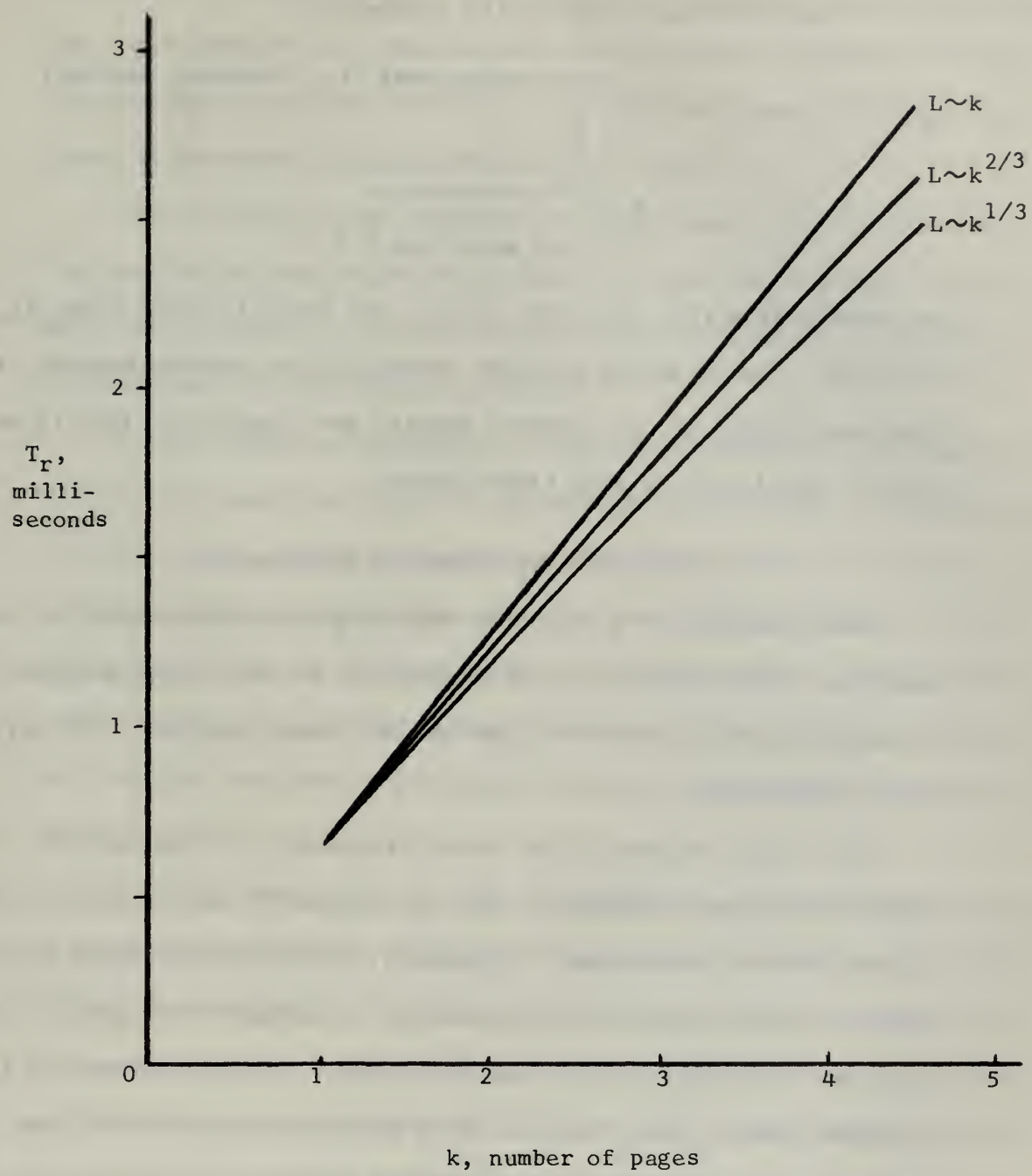


Figure 8. Relocation time vs. number of pages used, with blocks and pages method

and for the following values of the parameters:

$$t_a = 100 \text{ nanoseconds (i.e. hardware mapping)}$$

$$f = \frac{2}{3}$$

$$I_p = 500 \text{ microseconds}$$

$$L = 2000 \text{ words for } k = 1$$

The results show that for these values, the initialization time, kI_p, dominates. The situation would be reversed, with address mapping time being more important, if software mapping were employed. This is because t_a would then be many times larger.

A More Realistic Computing Environment

Three features of a realistic computing environment have not yet been given proper emphasis. These features are very large programs, variable-size data structures, and use of common routines. All affect program relocation.

Very large programs occur fairly frequently in some systems. In non-multiprogrammed computers, they are handled by well-developed overlay and chaining techniques. In general, if the address space of the system is large enough, unique names may be assigned throughout a computation. Re-naming is then never necessary. Otherwise, some of the addresses used in later portions of a process will have to be made the same as some used earlier. The operating system must keep a record of any such correspondence. In a multiprogrammed computer using relocation, this re-naming represents an extra, time-consuming translation beyond that required by the normal address/location map.

Examples of variable-size data structures in programs include arrays, lists, and pushdown stacks. These occur frequently, and it is difficult to know their eventual size, particularly in the on-line

environment discussed here. This leads to a dilemma. If, in processing, too many addresses are reserved for variable-size structures, an inefficient use of name space results. On the other hand, if too little space is reserved, naming conflicts will arise.

For both very large programs and variable-size data structures, the conclusion from the point of view of naming requirements is that it would be desirable to have an address space sufficiently large that, in practice, it would never be filled.

A third important feature of a realistic computing environment is the use of common routines. It has been suggested that in a time-sharing system, "common routines" means more than just a library collection. An important facet of interactive, on-line programming appears to be the frequent exchange of information between system users. In some cases, this exchange has occurred between two users active at their terminals, one entering some matter and then transmitting it, through the system, to the other.[9]

It is desirable, for the sake of efficiency, to code as many common routines as possible in "re-entrant" form. Such routines may, by definition, be entered by a second program before a first has finished its use. Ideally, then, only one copy of a re-entrant routine need be present within the system, no matter how many users might call it. How is reference to be made by programs to this single copy?

First, a separate address/location map may be assigned to the common matter, just as if it were an independent user program. This method has the advantage of permitting the common routine use of the full address

[9]Fano, R.M., and F.J. Corbato, Time-Sharing on Computers (Scientific American, September 1966), p. 140.

space of the computer. However, it requires a change of map whenever the routine is called. More importantly, the transmittal of arguments through two maps would be rather complex and possibly time-consuming. This disadvantage would compound when a number of common routines are referenced.

Second, the common routine may be assigned a portion of the address space of the calling program. Then no change of map is needed when the routine is called. If address space is sufficiently large, the assignment does not impose a significant restriction on the calling program. However, this method will work with a single copy of the common matter only if 1) it is arbitrarily relocatable, or 2) it always occupies the same portion of address space in any using program. Arbitrary relocatability would impose an additional constraint on the coding, beyond that of re-entrancy; it also implies an extra, time-consuming movement. By requiring the movement, it really begs the question of whether a single copy is being used. By contrast, use always of the same portion of address space appears to be a serious restriction. Yet if address space is large enough, this technique has the virtue of simplicity; a particular common routine would always be found at the same program addresses.

Segmentation

Up to this point, certain desirable features in the addressing structure of a multiprogrammed computer have been noted:

a) address space should be large enough that unique addresses may be assigned throughout any practical computation;

b) data structures should be expandable without necessitating a reallocation of addresses; and

c) information common to several programs should have the same addresses for all programs that reference it.

It should be stressed that the total address space of a system need not be physically implemented in main memory storage. In fact, considering the large addressing capability of some new computers, the cost of such implementation would be economically very prohibitive. The GE 645, an extreme example, provides 36-bit addressing, or the capability of specifying over 68 billion words.

Given a sufficiently large naming capability, the segmentation of program addresses provides a suitable way to structure the addressing scheme. Physically, the resulting program segments are an ordered collection of computer words with an associated segment name. The number of bits used for the segment name is chosen to permit as many segments as may be needed to distinguish different common routines, parts of programs, etc. The number of bits then left for word addresses within the segment should allow for the largest collection of information that is to be addressed as one ordered sequence. In the GE 645, 18 bits are employed for the segment name, leaving the same number for word addresses; $2^{18} = 262,144$. (See Figure 9.)

Segments are used for the allocation of address space, not physical memory. But unlike pages, they are not invisible to the programmer. To him, a segment is any more or less independent subdivision of a program.[10] It may consist entirely of instructions, entirely of data, or it may be a mix of both. Examples of likely segments are main programs, common subroutines, and data arrays.

The segment may serve as the basis of memory protection. An advantage here is that address space, which is unchanging over the life of a

[10]McGee, W.C., On Dynamic Program Relocation (IBM Systems Journal, No. 3, 1965), p. 188.

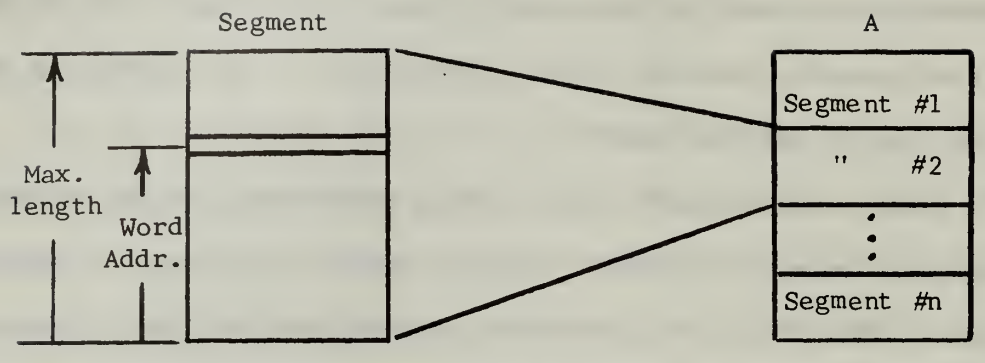


Figure 9. A segment, and the segmented address space

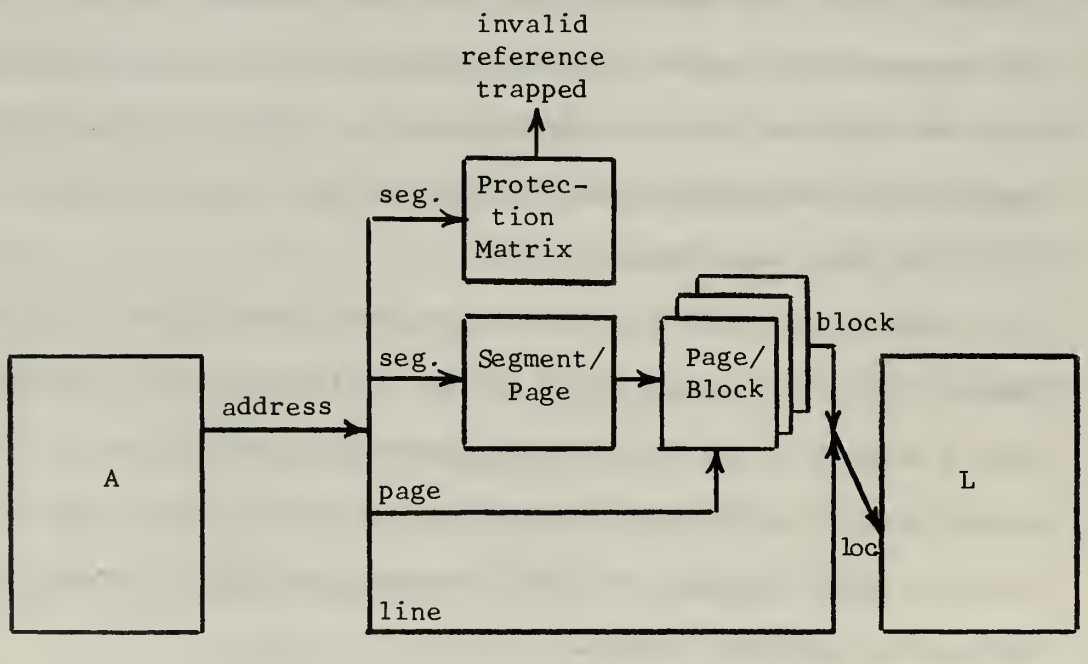


Figure 10. An address translation using segments, blocks, and pages

computation, is used. With, for example, m programs employing a total of n segments, an m x n matrix might be formed, its elements indicating the type of access which each program is permitted to make to each segment.

During execution, a correspondence is drawn between a program address identified by segment name and word number, and a physical memory location. This might be done directly, but a more flexible technique - in that it limits exchange overhead while not penalizing long segments - calls for dividing the segment into pages, and main memory into blocks. Thus segmentation may be considered to add a second level of translation to the blocks and pages method of program relocation.

Because of the presence of this second level, mapping times with segmentation will normally be higher than those which occur with use of other relocation methods. Also, because of the greater complexity of the tables to be searched, initialization times will tend to be higher. In general, then, segmentation will not be the fastest method of relocation on a single-transfer basis. Justification for its use is based, instead, upon its overall efficiency in address space allocation, which should, in fact, reduce the total relocation requirement.

The general nature of an address/location translation using paged segments is shown in Figure 10. An example implementation of program relocation, employing segmentation, is discussed in the next section.

5. Implementation - A Very Large Time-Sharing Computer.

Why Considered

This section presents and evaluates the method of program relocation implemented in a particular very large, time-sharing computer. The purpose of doing so is to obtain ideas which may be applicable to the Digital Control Laboratory system. Although the D.C.L. computer is not "very large", it should prove worthwhile to consider such a system where the problems of relocation are fully met. Possibly, some of its solutions may then be scaled to fit the D.C.L.'s requirements.

The machine chosen for this investigation is the IBM System/360, Model 67. The reasons for selecting this particular computer are the following:

a) It is further developed than the only other computer of like size and purpose, the GE 645;

b) the pre-eminence of IBM as a manufacturer of digital computers is based in part upon technical excellence, and the Model 67's relocation technique may reflect this fact; and

c) a Model 67 is being installed in the central computer facility of the Naval Postgraduate School, which causes a natural increase in interest in its design.

A note of caution is in order. The first System/360, Model 67 was delivered in January, 1967. Completion of the operating system for time-sharing, however, will not occur until 1968.[11] Thus, although the relocation hardware has been delivered, the systems programming necessary to employ it in a functioning system has neither been fully developed

[11]As announced in January, 1967. This is a slippage from an earlier stated date of August, 1967.

nor, more importantly, user-tested. The discussion which follows must, therefore, be tentative.[12]

Address Translation

A Model 67 equipped for standard, 24-bit addressing offers an address space, or "virtual memory" in IBM's terms, of 16,777,216 eight-bit bytes. (All specifications by IBM of addressing or storage capabilities are made in terms of these bytes.) The four high-order bits of a program-contained, or "logical", address are interpreted as a segment number; the next eight form the page number, and the final twelve give the line, or byte. (Figure 11.) Thus address space is divided into

16 segments, each of which contains up to

256 pages of

4,096 bytes each.

It is this 4,096-byte page which is the fundamental quantity moved or translated during program relocation.

32-bit addressing is available as an option. This provides a virtual memory of over four billion bytes. The logical address is broken into three sections of twelve, eight, and twelve bits specifying the segment, page, and line, respectively. Thus there are 4,096 segments addressable with this option, while the number of pages in each, and the length of a page are the same as in 24-bit addressing.

It is intended to use strict "demand paging" in the Model 67. That is, when a program is to begin or to continue executing, only its current page is necessarily brought into main memory. Further pages not already

[12]Available technical information on this computer is limited. The principal reference is System/360 Model 67 Time Sharing System Preliminary Technical Summary (IBM Form C20-1647-0, 1966).

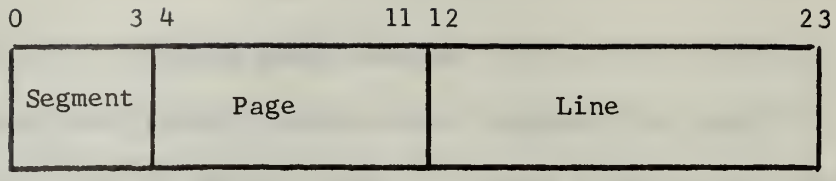


Figure 11. Model 67 logical address

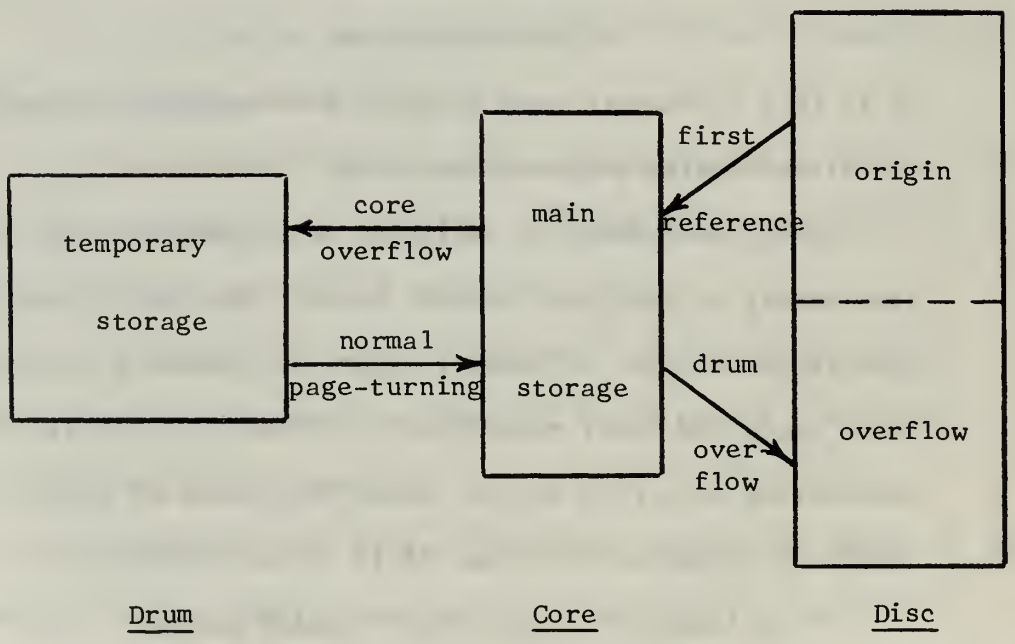


Figure 12. Movement of program pages in the Model 67

in main memory will be brought in only when "demanded", i.e. referenced. This technique will, it is expected, eliminate overhead due to unnecessary page movement. Storage for program pages, until they are first needed, will be on a disc; thereafter, they will be either in core memory, or as required to free core space, on a "swapping" drum. Any overflow from the drum will be held again on the disc. (Figure 12.) All transfers between these different levels of storage will be completely transparent to the user.

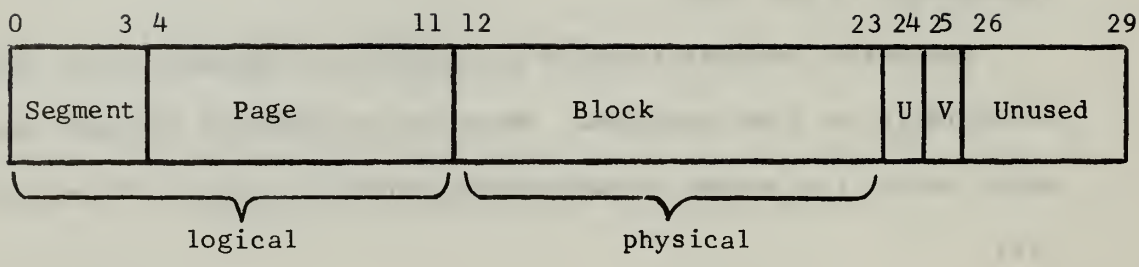
The entire address/location translation is implemented in hardware to minimize the time required. There are two special hardware features which assist the system in maintaining execution speed. These features are:

a) an associative memory element;

b) storage of the instruction counter in relocated form.

The associative memory element, first mentioned in the preceding section of this thesis, consists of eight registers. Each time a new page is referenced by a program, its segment and page values, and current physical block location are loaded into one of these registers. On subsequent program references to virtual memory, a high-speed parallel search of the registers is made. If the desired segment and page number are found, the physical location information is routed to replace that which otherwise would have been supplied by the segment and page tables. With eight registers, the relocation of the addresses of a program containing up to 32,768 bytes can be performed entirely in the associative memory.

The structure of each associative register is shown in Figure 13. Besides the logical segment and page numbers, and physical block location,



U - use

V - validity

Figure 13. Contents of Model 67 associative register

there are included a "use" bit, a "validity" bit, and four presently unused bits. These last provide a very desirable, built-in capability for system evolution. The validity bit indicates whether the page named in that register is in core. If zero, the page is not, and any associative comparison being made is aborted. In effect, this bit appears to offer the operating system a safety check, particularly valuable just after program exchange. Upon an exchange, all validity bits are automatically set to zero; this allows the operating system to load only the associative register initially needed for the new program, without being concerned that a program reference to a new page might result, through combinations still left in other associative registers, in inadvertent access to another user's program. Each use bit, also set to zero during program exchange, becomes one the first time its register is used during relocation. When the eighth use bit is set to one, all of these bits are returned to zero, and the cycle repeats. The operating system should attempt to find a register with a zero use bit when determining where to load the page/block information for a newly referenced program page.

Storage of the instruction counter in relocated form adds to program execution speed by obviating the need for instruction address translation until a branch occurs or a page boundary is crossed. The storage is accomplished in an extra register used solely for this purpose.

How is a complete 24-bit address translation performed? (Figure 14.) First, relocation must be properly specified in the Program Status Word, the 64 bits of control information associated with each active program in the system. Bits four and five of the Word indicate the relocation mode as follows:

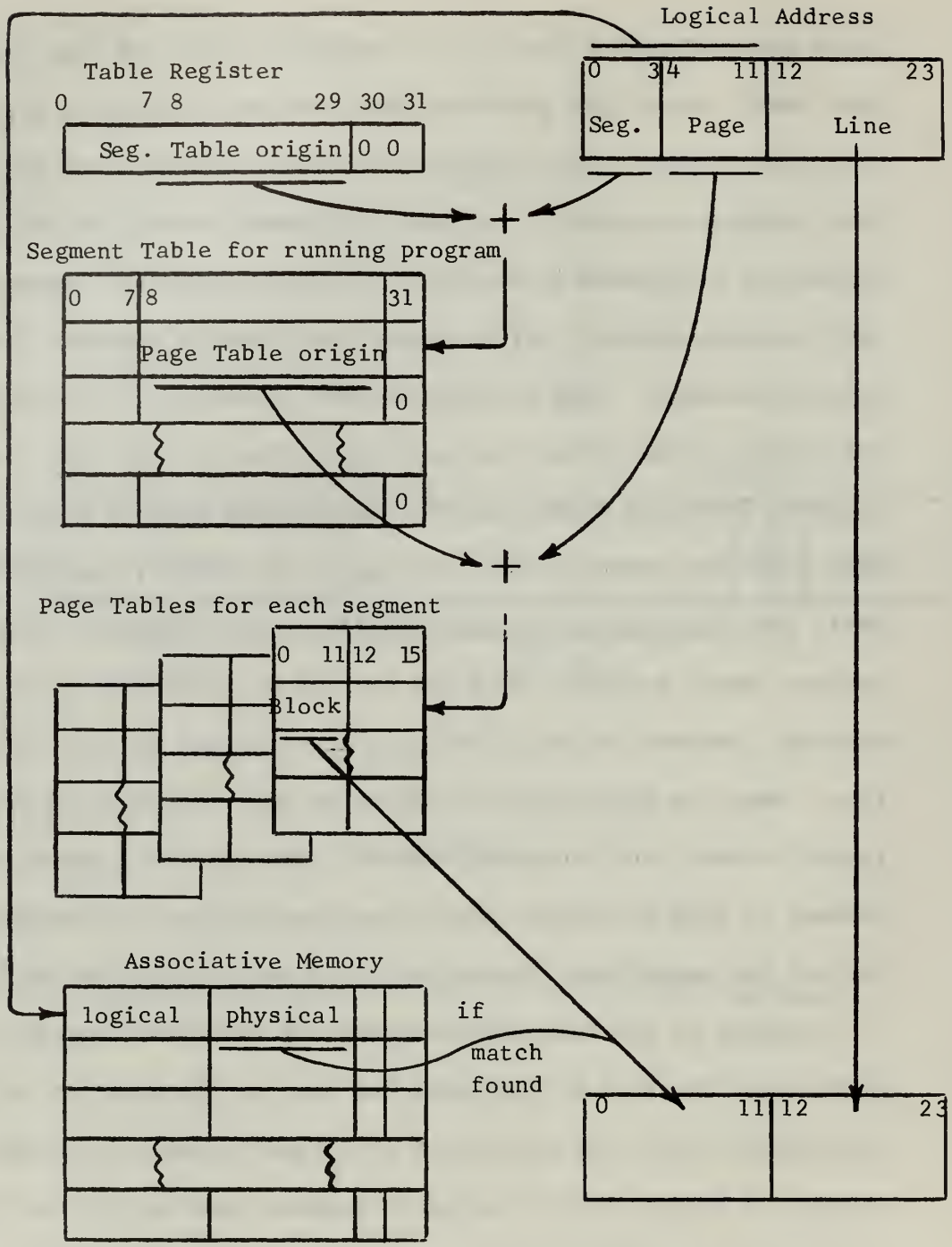


Figure 14. 24-bit address translation in the Model 67

Bit 4	Bit 5	Mode
0	0	No relocation, 24-bit addressing
0	1	Relocation, 24-bit addressing
1	0	Invalid combination resulting in a trap
1	1	Relocation, 32-bit addressing (invalid combination if 32-bit addressing not installed)

Then, when a program references a logical address, a match is first attempted between bits 0-11 of the logical address (the segment and page numbers) and bits 0-11 of each associative register having bit 25 (the validity bit) set to one. If a match is found, bits 12-23 of the associative register become bits 0-11 of the actual core storage address. Bit 24 (the use bit) is set to one if not already at that value.

If, however, there is no match, the segment and page tables stored in core memory must be used. All additions described in the look-ups on these tables are permanently wired for speed, reducing the reference time for each table to one memory cycle. There is first a Table Register, whose bits 8-31 contain the origin of the segment table for the running program. To this origin are added bits 0-3 (the segment number) of the logical address. For this addition, these bits are aligned with bits 26-29 of the segment table origin since the entry being found is four bytes long. This obtains, held in bits 8-31 of the result, the origin of the page table for the indicated segment. Added then to this origin are bits 4-11 (the page) of the logical address, aligned with bits 23-30. This finds a two-byte entry in the page table consisting of a physical block location portion (bits 0-11) and control bits (12-15). Bit 12 is zero if the referenced page is actually in core; if so, bits 0-11 are used as the same bits of the physical address. If bit 12 is one, the

operating system may be called in order to set up an input operation to bring the desired page into core. This last action would presumably continue independently through an input-output processor while another program takes over the central processor. The other control bits (13-15) are reserved for future use. The translation is completed with bits 12-23 of the logical address forming, unchanged, the same bits of the physical address.

Address translation with 32-bit addressing is different only in that the segment table for each program may be much longer, containing as many as 4,096, instead of 16, entries.

Relocation Timing

Enough is known about the Model 67 to quantify the address mapping portion of program relocation time. When relocation is operative, and a memory reference occurs, the Model 67's clock is stopped for 150 nanoseconds during the associative compare. If a match is found, that time is the delay imposed by use of address translation. That is, t_a = 150 nanoseconds. If, however, the segment and page tables must be used, the clock remains blocked while two accesses to the tables are made and while the page entry found is loaded into one of the associative registers. This action takes three memory cycles, or about 2.1 microseconds. Now, t_a = 2.25 microseconds. Obviously, system performance is greatly degraded when the segment and page tables are used, even if all pages referenced are already in core memory. How often will use of these tables be necessary during execution of a typical program? An IBM simulation indicates that a figure of 5% will be realistic.[13] That is,

[13]System/360 Model 67 Time Sharing System Preliminary Technical Summary (IBM Form C20-1647-0, 1966), p. 56.

the tables will be required on 5% of the memory references made by an average program; for the other 95%, a match will be found in the associative memory. Certainly this figure will vary with different programs and in different computing environments. Using it, however, the effective, or weighted mean, mapping time can be computed to be

$$t_a = 0.05(2250) + 0.95(150) = 255 \text{ nanoseconds}$$

This value is beyond the range specified (20-200 nanoseconds) for single-level address translation schemes. Further, it is recalled that

$$F_a = \frac{t_a}{mt_m} f \qquad (4.4)$$

For the Model 67, $t_m = 750$ nanoseconds. Choosing, as suitable values,

$$m = 3$$

$$f = \frac{2}{3}$$

then

$$F_a = \frac{255}{3(750)} \frac{2}{3}$$

$$= 0.0756 \text{ or nearly } 7.6\%$$

With an F_a of this magnitude, increases in program execution times due to address translation overhead will be noticeable. From another point of view, consider a hypothetical program requiring 100 storage references, whose run time on the Model 67 in unrelocated mode is 200 microseconds. This might be a typical short scientific calculation, quite active in memory as it carries out the programmed algorithm. It is assumed that the program is entirely in core memory. With relocation, and 5% of the memory references requiring use of the segment and page tables, the run time will become

$$200 + 100\left[0.15 + 0.05(2.1)\right] = 225.5 \text{ microseconds}$$

This is an increase, due to address translation alone, of nearly 13%. Even if the table activity were zero, execution speed would be degraded in this example by 7.5%.

Program Sectioning

System users must be provided with programming aids which enable them to take advantage of the program segmentation implemented in the hardware. They must be able to conveniently section their programs into logical units. There must also be a means of linking these sections, including their joining to those, such as common routines, written by others. In a multiprogrammed computer where it is intended that only one copy of a re-entrant common process be called by all users, special consideration must be given to this linkage.

Compiler languages generally provide already a means of sectioning processes. In FORTRAN and in ALGOL, for example, program subdivisions are formed by use of Subroutine and Block statements, respectively. Many assemblers provide a similar capability through ORiGin directives and through machine instructions which branch. Examples of such instructions include the Return Jump of CDC 1604 CODAP and the SDS 900 Series computers' Mark Place and Branch. The assembly language for the System/360, Model 67 includes similar features; additionally, the programmer's general control of sectioning has been expanded, and a means of linking programs to a re-entrant common process has been provided.

Three assembler directives implement the new sectioning power on the Model 67. These directives are:

CSECT - Control SECTion

COM - COMmon Control Section

PSECT - Prototype Control SECTion

A "control section" is a block of coding whose virtual memory assignments can be adjusted, independently of other coding (save for linkages), at linkage or load time without impairing the operation of the program. Thus a control section is a logical unit, or in the sense of Section 4 of this thesis, a program segment. The CSECT directive identifies the beginning of a control section. A tag may be placed in the label field (to the left on the coding sheet) of a CSECT, thus naming the section. All statements following a CSECT are assembled as part of that control section until a new CSECT directive is encountered. The object code for each CSECT starts on a page boundary, and a page table (without physical location assignments, of course) is produced as the section is assembled.

The COM directive identifies common coding blocks which may be referred to by more than one independent assembly when the assemblies and the common block have been linked and loaded as one overall program. "Blank" common sections may contain only data placed there during program execution. Named common sections, however, may contain instructions, constants, or data, in any combination.

It is the PSECT directive which provides for the linkage of calling programs to re-entrant common routines. The chief problem here is the handling during execution of temporary, or "working", storage required by the routine for each program which is concurrently using it. In the Model 67, this matter is resolved by the setting up of an individual working area for each calling program within that program's own virtual memory. Re-entrant routines in this computer appear to have different address space assignments to different programs, although their actual physical locations remain unchanged. Thus when control is transferred to such a routine, the calling program must specify an "address constant",

a quantity which reflects its virtual memory assignments, in order that the routine may obtain a working area therein for this caller. The prototype control section is defined for re-entrant process use to handle these address constants and working storage assignments.

Within a re-entrant routine all working storage and address constant requirements are placed within a prototype control section. This section forms a special subdivision of the re-entrant process. When the routine is called, a copy of the contents of the prototype control section is made and assigned to virtual memory locations within the calling program. Thus a working storage area and proper address transfers are established in and for the calling program. All of this is transparent to the user; he need not know any of the internal requirements of the re-entrant routine which he is employing.

Lastly, one or more operands may, quite usefully, be included with a CSECT, COM, or PSECT directive in order to specify certain attributes of the control section. These operands include:

PUBLIC - indicates that the control section contains matter to be accessible to any program

REENTRANT - indicates that the section's coding may be re-executed from any point after interruption

VARIABLE - denotes that the section's length may vary during program execution

READONLY - indicates that the section contains instructions or data which are never modified

Evaluation

Consideration of the extensive relocation hardware incorporated in the System/360, Model 67 leaves no doubt that its designers are attempting to make thorough provision for the program movement and addressing requirements of time-shared, multiprogrammed computations. However,

without a completed operating system and user tests, it is difficult to assess the relocation performance of this computer. The importance of the operating system as it uses, or fails to use, the features of the hardware to produce an efficient total relocation method cannot be over-emphasized.

The address space of at least 16 million bytes (four million 32-bit nominal words) appears sufficiently large to allow the Model 67 to effectively use segmentation. That is, virtual memory can readily hold very large programs together with a full library of common routines, while having a further allowance for variable-size data structures. The small number, 16, of segments provided in the translation of standard 24-bit addresses suggests, however, that these segments, while useful in the hardware translation itself, will not serve as the immediate logical subdivisions of programs. For the latter, groups of pages will be more appropriate in available number and length, as required during the assembly of control sections.

The size of the individual program page, 4,096 bytes, is adequate to hold many shorter routines or data areas. Yet for computing environments where longer programs are the rule, this short length may lead to considerable page-turning in and out of core. This is a critical subject, for how much of page movement overhead may be really submerged by input/output independent of processing? May not the central processor, in this complex multi-level store system, have to do a significant amount of initialization and set-up before turning over the operation to a channel? If the use of strict demand paging results in excess overhead time, it will be necessary to make some modification to the page-turning algorithm. It might be desirable to ensure that core contains several

pages, rather than just one, of a program about to execute, or even to limit a user to programs occupying some reasonable subset of total core and then automatically to bring in his entire program before he executes. The SDS 940 time-sharing system, for example, does the latter; there, programs are normally limited to 16K of a 64K word maximum core.[14] A further reason to limit users to a subset of core is to minimize the length of the segment and page tables that must be handled by the system. With enough concurrent users, the core space occupied by these tables may become significant; if, in such case, some of the tables are swapped in and out during program exchange, a further addition is made to system overhead.

A definite liability in the Model 67 is that part of relocation overhead due to address translation. Extra time is always required, even when the associative memory alone is used. Some smaller time-sharing systems with single-level mapping (no segments, only pages) such as, again, the SDS 940,[15] are able to perform address translation with no increase in execution time.

Finally, from the point of view of the Digital Control Laboratory's requirements, the major idea obtained from study of the IBM System/360, Model 67 is a realization of the complexity of segmentation. This concept, whose complexity is apparent in both hardware and supporting systems programming, is far more difficult to implement than to describe.

[14] SDS 940 Computer Reference Manual (SDS Publication 900640A, August 1966), p. 8.

[15] Ibid.

6. Implementation - Digital Control Laboratory.

The Computing Environment

The Digital Control Laboratory, a facility of the Department of Electrical Engineering, serves as a tool of research and instruction for faculty and students of the Naval Postgraduate School. Many projects, most of which are associated with coursework or theses, are accomplished here during the academic terms. For example, all students in the beginning course in digital computers offered by the Department of Electrical Engineering currently perform at least one-half their laboratory work in the D.C.L. While there are some extensive projects, most are quite small, being measured, in terms of digital computer program lengths, in tens and hundreds of instructions.

A particular competence has been developed in the use of cathode-ray tube displays and in hybrid computation. In fact, the Laboratory presently contains the only display and hybrid equipment available at the Naval Postgraduate School. Much advanced course and thesis work has been performed with the aid of this equipment, in applications such as tactical warfare simulation and sampled-data control systems. Considering the value of this work to the Department of the Navy, its continuance is important and even necessary. Participating officer students gain experience which may be invaluable to them in future assignments.

The new computer system which is currently being obtained for the D.C.L. will significantly expand and enhance its capabilities. The principal item ordered is a Scientific Data Systems Model 930 digital computer, which is a 1.75-microsecond memory cycle, 24-bit word machine. Two keyboard cathode-ray tube displays, each capable of operation in

character, vector, and point modes, are also included. These displays do not contain any internal memory; because of this, all information being presented on them will have to be stored in the memory of the SDS 930. There will be a new analog computer and the necessary analog-digital converters for connection to the digital computer. For the first time in the D.C.L., a nearly full range of standard digital computer peripheral equipments will be available; these are a card reader, line printer, paper tape reader and punch, and two magnetic tapes.

Development of a limited, internal time-sharing system is envisioned as the best means to make full use of all this equipment. The concurrent operations thus provided should reduce problem-solution time and, it is hoped, allow a closer interface between user and machine. Because of their greater potential in these respects, the two displays will receive preference over the standard peripherals in service received from the SDS 930. Small-scale batch-processing using the card reader, line printer, and paper tape system in the background is planned, however. In fact, two priorities are envisioned for this background computing. The higher would be assigned to normal, short programs; the other would be for the infrequent long program. It is hoped to later include hybrid computation within the capabilities of the time-sharing system. When this is done, however, the highest scheduling priority in the SDS 930 will probably have to be accorded to its hybrid program, because of the latter's relatively rigid requirements for execution at fixed time intervals.

The overall goal of the time-sharing system proposed for the Digital Control Laboratory may be stated as follows: improvement in service to all, but with preference to display and hybrid computing.

Pertinent Features of the D.C.L. Digital Computer

The SDS 930 computer being delivered to the Digital Control Laboratory[16] has many features which will influence the choice of relocation method to be used in the proposed time-sharing system. In general, this computer may be characterized as a later second-generation machine, not designed for multiprogramming or time-sharing.

There will be 16,384 words of core memory. This amount at first seems most adequate, considering the probable small size of most user programs. However, in the time-sharing system, all of this memory will not be available for user programs. Space must be reserved for the resident portion of the operating system and for a buffer for each of the displays. The relocation method used will probably affect the size of the operating system, including the core resident.

Secondary storage is provided as a 131,072-word rotating disc.[17] This disc is unusual in that a read/write head is included for each track, thus eliminating head positioning time when access is made. At 1710 revolutions per minute, the average rotational latency time is 17.5 milliseconds, while the actual transfer rate is 117,000 words per second. This last is the figure when more than one disc sector (a sector holds 64 words) is accessed during a transmission; it is somewhat lower than the single-sector rate because of intersector gaps. At this speed, the entire 16K core memory can be copied onto the disc in 0.175 seconds, which includes the maximum latency time of 35 milliseconds. On this disc the sector address is automatically incremented during a multiple-sector

[16]Requisition N62271-67-C-0013, 13 October 1966, from Supply & Fiscal Officer, Naval Postgraduate School, to Navy Purchasing Office, Washington, D.C.

[17]SDS 940 Computer Reference Manual, pp. 75-78.

transfer. Manually-controlled prevention of writing operations is provided for each of the four 32,768-word blocks; this feature will prove useful in preserving permanent files, such as disc-resident portions of the operating system, against inadvertent destruction.

The two displays and the analog equipment will be connected to the SDS 930 through separate, direct accesses to core memory. Except for initialization, these devices may operate independently of the central processor, under the following condition. Core memory is divided into two 8K blocks; when a display or analog input/output transfer involves the block other than the one which the central processor is currently accessing, the two actions are independent, and the processor is not held up at all. However, when the transfer operation and processor simultaneously use the same memory bank, the transfer will take precedence, and the processor will be delayed one memory cycle time.[18] This fact suggests that due to display refresh requirements, as far as possible user programs and the display buffers should occupy different memory banks.

In constrast, all the other peripheral devices, including the disc, will be joined to the digital computer through what is termed a time-multiplexed communication channel. This channel shares use of an internal register with the central processor, and input/output operations on it always involve cycle-stealing.[19] This is not to say that input/output cannot take place concurrently with computing, for it can, but computing time will increase by the number of memory cycles used for the input/output operation, at a rate of two cycles per word transferred.

[18]SDS 930 Computer Reference Manual (SDS Publication 900064D, February, 1966), p. 28.

[19]SDS 930 Computer Reference Manual, p. 25 and p. 28.

The SDS 930 possesses no hardware aids to the address translation part of program relocation. In particular, the fourteen-bit address field of instruction words provides an address space or virtual memory exactly equivalent to the physically-implemented 16K of core. There is no provision for translation on a segment, page, or program basis. Finally, there will be no core memory protection feature. Such feature is a very desirable corollary to any address translation method. An SDS option which provides write lock-out protection in 512-word blocks of core is available,[20] but it was not ordered.

The software or systems programming to be furnished with the D.C.L. SDS 930 is, with one exception, designed solely for non-multiprogrammed, non-interactive batch-processing. It includes MONARCH, a magnetic tape-oriented operating system.[21,22] A disc-resident version of MONARCH is now in preparation. There is a second operating system, Real-Time MONITOR,[23] now being written. It too is disc-resident.

MONARCH provides batched assemblies, compilations, and executions in any combination for any number of programs. Its language processors are SYMBOL, META-SYMBOL, FORTRAN II, and Real-Time FORTRAN II.[24] Input/output devices which it will handle are card reader and punch, line

[20] Ibid., p. 4.

[21] SDS MONARCH Reference Manual, 900 Series/9300 Computers (SDS Publication 900566B, August 1965).

[22] SDS MONARCH Technical Manual, 900 Series/9300 Computers (SDS Publication 900616B, October 1965).

[23] SDS Real-Time MONITOR Reference Manual (SDS Publication 901108A, February 1966).

[24] ALGOL is available upon request.

printer, magnetic tape, paper tape reader and punch, typewriter, and (for the disc version only) disc. MONARCH also includes the library of SDS programmed operators.[25]

Real-Time MONITOR is intended to add a generalized interrupt-handling feature, not interactive time-sharing, to a standard batch-processing executive. Its processors include FORTRAN IV, SYMBOL, and META-SYMBOL. It will handle the same peripherals as Disc MONARCH and also has the programmed operator library.

Both operating systems include relocating loaders. These routines are intended strictly for use in non-multiprogrammed batch-processing. They make no provision for program movement or address translation after execution has once started.

A special display program is the only software item being furnished which immediately provides a capability for more than batch-processing. It offers a very basic facility for on-line utilization of the digital computer from the display consoles. It allows source-language program creation, including editing, at a display and provides for transmission of the prepared program to such storage as magnetic tape or disc. When a program is ready for assembly or compilation, the display may then function as the system control medium to bring in MONARCH to perform the desired processing. There is no further interaction with the program until MONARCH has finished and, if requested, the program has executed. The MONARCH system and the display program will not reside in core memory or operate at the same time.

[25]SDS 930 Computer Reference Manual, p. A-17.

The Key Factors

Of all the mentioned features of the Digital Control Laboratory and its new computing system, the following are considered to be the most important in terms of their effect upon the choice of method for program relocation:

a) most programs will be small, containing hundreds, rather than thousands, of instructions;[26]

b) there will be only two interactive users, at the displays;

c) a disc with a high transfer rate and low latency time is to be available; and

d) the computer has no hardware or programming aids designed to facilitate any method of relocation.

These are the key factors to be remembered in the analysis below.

Relocation Analysis

The use of segmentation, of any two-level address translation scheme, appears to be neither warranted nor feasible in the Digital Control Laboratory. The limited address space of the SDS 930 would, in itself, prevent the gaining of the advantages of segmentation. In addition, the complexity of the necessary hardware and programming would be, relatively, immense. The System/360, Model 67 is good proof of this last point.

Employment of a single-level, blocks and pages method of program relocation would theoretically allow the most efficient allocation of core memory. Further, address translation may be accomplished in hardware at small cost in terms of execution speed. Recalling that

[26]This point might be questioned in view of the expanded capabilities of the D.C.L. However, with the System/360, Model 67, the NPGS Computer Facility has received a sizeable increase in its computing power and should continue to attract most large programs.

$$F_a = \frac{t_a}{mt_m} f \qquad (4.4)$$

it is calculated for the SDS 930, where t_m is 1.75 microseconds, with

$$m = 3$$

$$f = \frac{2}{3}$$

$$50 \leq t_a \leq 200 \text{ nanoseconds}$$

that $0.005 \leq F_a \leq 0.025$

That is, program execution time would be increased, at most, by about 2.5%. A possible paged address translation scheme for the D.C.L. is shown in Figure 15. It is modelled upon the mapping performed in the SDS 940,[27] which also employs 2,048-word pages. The advantage of using such a page size in the D.C.L. is that this makes possible the use of two mapping registers, as in the 940. This similarity would reduce the original design effort required for the implementation of blocks and pages in the D.C.L. Otherwide, shorter program pages, perhaps 1,024 words, would probably be advisable in the Laboratory in view of the many small user programs. Unlike the 940's, the translation scheme shown does not provide for a physical core memory of 64K words; instead, provision is made only for a core of 32K, which seems a reasonable limit to potential D.C.L. expansion in view of the small number of concurrent users. The extra bit thus made available is to be used for core memory protection, which, now with two bits per block, could include four forms. The SDS 940, which reserves one bit per block for such protection, thus provides only two forms.

[27] SDS 940 Computer Reference Manual, pp. 8-9.

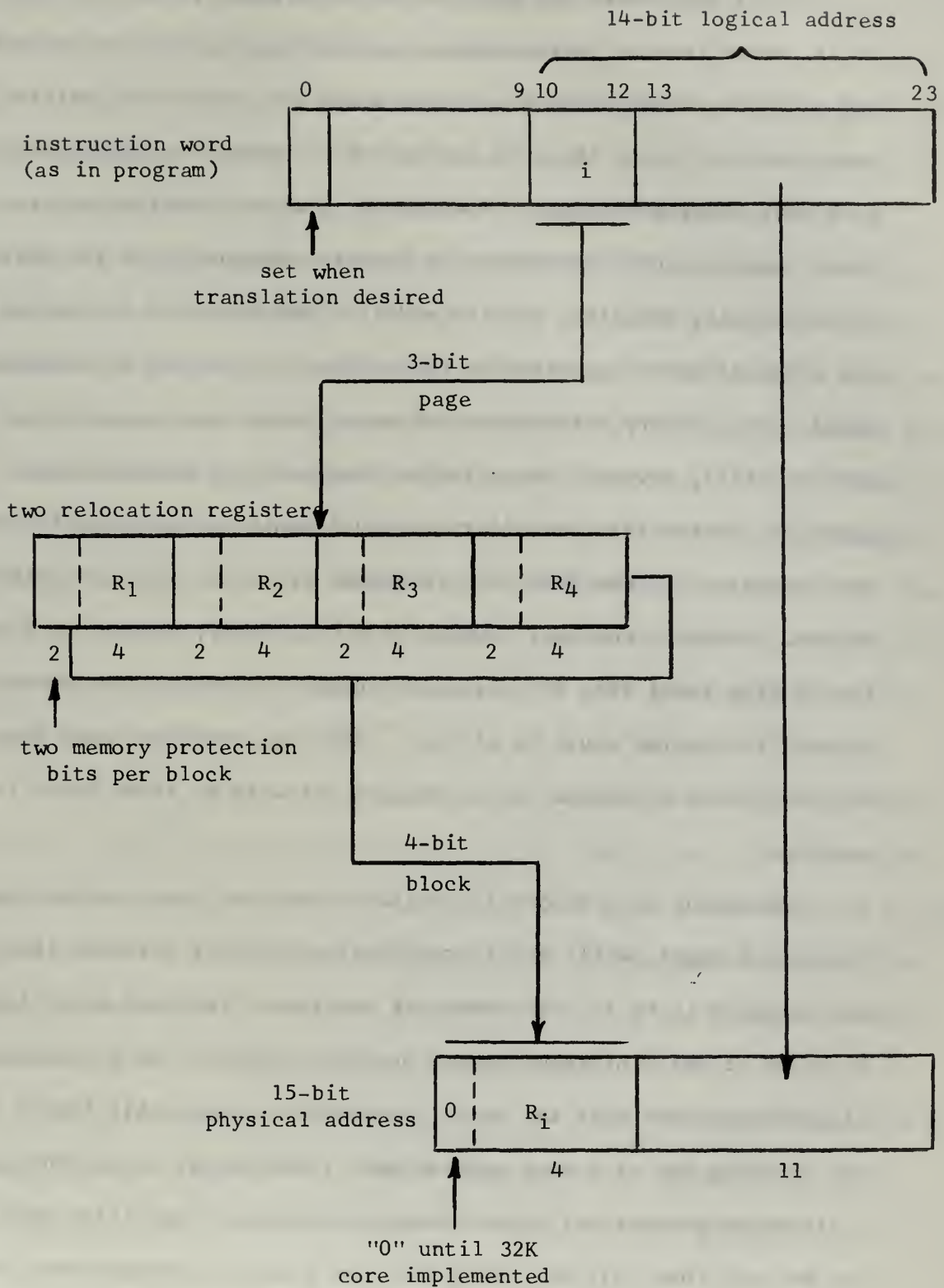


Figure 15. Possible page/block address translation scheme for the D.C.L.

Use of the blocks and pages method of program relocation in the D.C.L. would require extra expense, considering the relocation registers and additional logic needed. Coding would also have to be written to dynamically allocate the core memory and to update the relocation registers upon program exchange. Presumably, however, programs written for these purposes under Department of Defense sponsorship at the University of California, Berkeley, for its modified SDS 930 would be available, so the original effort required in this respect at the D.C.L. would be reduced. The primary disadvantage of using blocks and pages in the Laboratory is still, however, the relative complexity of implementation. It cannot be denied that the effort required would be considerably more than that necessary if the relocation register or "no relocation" methods were chosen. Further, the small number of D.C.L. users, coupled with the fact that a high-speed disc is available, suggests that the time advantages gained from paging would be minimal. That is, very fast swap times can be attained on a program basis, unpaged, as will be shown later in this section.

Employment of a relocating register would be less complex than blocks and pages, while still providing potentially for more than one user program to be in core memory at one time. Programs would have to be moved as one contiguous block, but this might not be a significant disadvantage when most are small. Considering again this factor of size, the swapping out of a user program when interrupted, to provide space for an incoming process, will not always be required. Recalling (4.13), it can be seen that this will keep down the program exchange time. Of course, since $t_a \neq 0$ with the relocating register, there will be some increase in program execution times due to the address mapping time. The

amount would be small, however, of about the same magnitude range as calculated above for the blocks and pages method.

This relocation method does necessitate the design and construction of the mapping register and associated logic. The latter involves, in particular, the screening of instruction codes during execution to select those for which address translation will be performed. Programming will also be needed to keep track of available space within core memory and to provide shifting of user programs within core in order to free space for an incoming process. The relocating register method also requires the incorporation of memory protection, usually in the form of two bounds registers which restrict the range of access of the program being executed. If this method of relocation were to be chosen for the D.C.L., some assistance in its implementation could probably be obtained from The RAND Corporation, which employs it on a PDP-6 computer in the JOSS time-sharing system.[28,29]

The final method to be considered is "no relocation", or the swapping of jobs upon program exchange with no address mapping within core memory. The program to be executed next is always loaded starting at the same address. In the simplest application, which is that which is considered here, the active user program is the only user program in core memory; this is called complete job swapping.

The disadvantage of this method is the large amount of program movement into and out of core. Except when a program is ended, an "out" movement is required upon program exchange, and an "in" movement is always necessary.

[28]Interview with R.L. Clark, The RAND Corporation, Santa Monica, California, 9 February 1967.

[29]Bryan, G.E., JOSS: User Scheduling and Resource Allocation (The RAND Corporation, Memorandum RM-5216-PR, January 1967), pp. 2-4; 17-18; 39-47.

On the other hand, "no relocation" would certainly be the simplest method to add to a fundamentally non-time-sharing computer such as the SDS 930. Its allocation of core memory to one user program at a time emulates that of the non-multiprogrammed, batch-processing software being furnished with the machine. Thus, if it were to be the method chosen, more of this extensive programming might be usable in the D.C.L. Since t_a is zero, there is no increase in program execution time due to address mapping. Also, the core memory protection required will be minimal. The relative simplicity of this method can be counted upon to produce the smallest size, N, of relocation program.

The time taken for program exchange with "no relocation" may be tolerable in the Laboratory for two reasons: there are only two interactive users, for whom response time is most critical, and there is a high-speed disc. The latter provides for very fast program swap times, such as the following:

Operation	Time
Swap out and in a 1K program	0.09 seconds
Swap out a 1K program; swap in a 5K program	0.12 "
Swap out and in a 5K program	0.16 "

All of these times include the maximum latency time for both the "out" and "in" transfers. In addition, they are calculated for completely non-overlapped input/output. Thus they are absolute maxima, and yet they are short in terms of human reaction times.

These times may be directly compared to those required if a relocating register method were implemented. The comparison will be based upon

the technique developed in Section 4 of this thesis. The variable is $\bar{L}$, the average program length. It is again assumed that

$$I_{RR} = 3I_{NR} \tag{4.11}$$

and that $$(T_{em})_{NR} = nt_m L \tag{4.12}$$

Thus the following expressions apply:

$$(\bar{T}_r)_{RR} = t_a \overline{Lf} + 6\bar{I}_{NR} + 2\frac{nt_m \bar{L}^2}{M} \tag{4.15}$$

$$(\bar{T}_r)_{NR} = 2\bar{I}_{NR} + 2nt_m \bar{L} \tag{4.16}$$

Suitable values are chosen for the D.C.L. 930:

t_a = 100 nanoseconds

$\bar{f} = \frac{2}{3}$

$\bar{I}_{NR}$ = 500 microseconds

$n = 2$

t_m = 1.75 microseconds

M = 8000 words (i.e. one-half the SDS 930"s core memory is available for user programs)

The results are:

$$(T_r)_{RR} = 6.7(10^{-5})\bar{L} + 3 + 8.75(10^{-7})\bar{L}^2$$

$$(T_r)_{NR} = 1 + 7(10^{-3})\bar{L} \quad \text{(milliseconds)}$$

These expressions are plotted in Figure 16 for $0 < \bar{L} \leq 1000$ words, i.e. for the short program lengths expected in the D.C.L. The plot shows that at these lengths, the relocating register method has little time advantage over the simpler "no relocation". The maximum advantage of the relocating

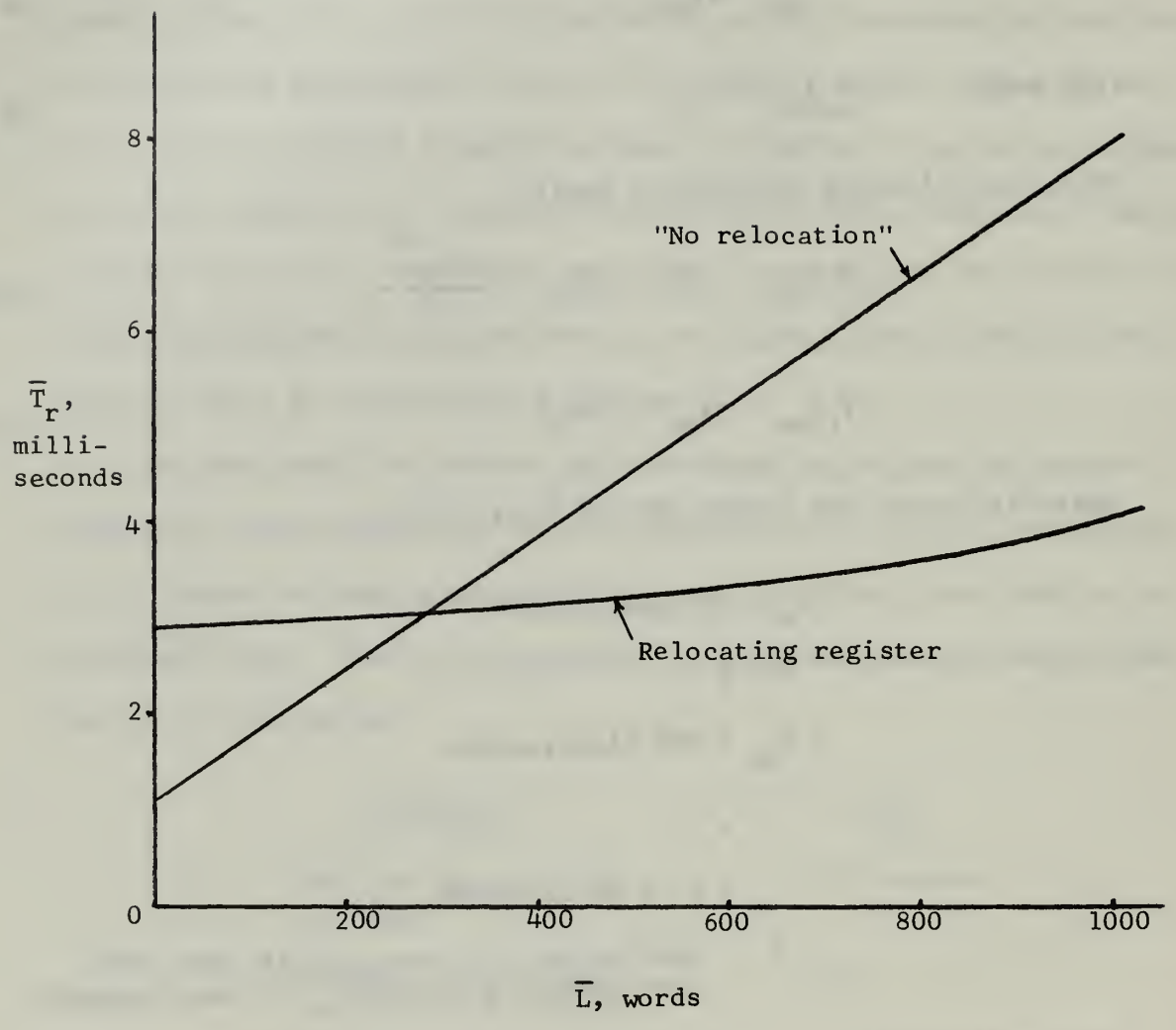


Figure 16. Relocation time in the D.C.L.; use of relocating register compared to employment of "no relocation"

register is only 4.05 milliseconds, at $\bar{L}$ = 1000. In fact, until $\bar{L}$ = 300, "no relocation" is actually faster, due to the larger initialization time associated with the relocating register method.

Finally, "no relocation" in the sense of complete job swapping has been demonstrated through a number of successful applications to be a practical method for use under conditions of full-scale time-sharing. It is the relocation method now employed in the General Electric Company's commercial time-sharing system, which serves up to 40 concurrent users. It is also employed at the System Development Corporation, for the 31 users of its AN/FSQ-32 system.[30]

Recommendation

Considering the requirements of the Digital Control Laboratory, there is no need, in program relocation, for more than complete job swapping. The time taken for swapping would not be excessive, and the relative simplicity of implementation is most attractive. It is, then, the recommended method. When the programs are to be exchanged, a transfer out of the entire old user program would occur, and the new user program would be loaded beginning at one fixed address.

Only if some large programs are found to be a common occurrence in the new system, may one refinement be found worthwhile. This would be to swap out, upon program exchange, only so much of the old user program as is required to free sufficient core space for the new user program. When the old user is large, and the new user is small, relatively, a significant amount of time may be saved by thus avoiding the unnecessary relocation of the entire old user program. At some point, the time saved

[30]Interview with E. Myer, System Development Corporation, Santa Monica, California, 14 February 1967.

should become greater than that required to execute the extra coding needed to keep track of how much of each user is in core at any moment.

For details of a possible implementation of complete job swapping in the Digital Control Laboratory, see Appendix II of this thesis.

7. Conclusion.

In order to be able to make a recommendation for the new computing system in the Digital Control Laboratory, this thesis investigated the general subject of program relocation in a multiprogramming environment. Four methods of program relocation were identified and analyzed:

a) "No relocation", now used, for example, on the General Electric Company's commercial time-sharing system;

b) Relocating register, successfully employed at The RAND Corporation;

c) Blocks and pages, featured in the SDS 940; and

d) Segmentation, implemented in the IBM System/360, Model 67.

Basic upon the D.C.L.'s specific requirements, a recommendation for use of "no relocation" was made. Thus the announced aim of the thesis was achieved.

One general conclusion other than the recommendation was reached during the writing of this thesis. As research progressed, it became evident that the technical elegance, in itself, of a relocation technique is a very poor criterion upon which to base a choice of method for a particular system, such as the D.C.L. First, the more elegant the method, the more complex its implementation. Second, the relocation methods studied all differ in elegance, yet each has found practical application. The reason for this is that far more important factors than elegance are found in the environment and features of the target system. What was learned in the writing of this thesis is that these factors must be identified, for they, properly, will affect most the choice of relocation method. Technical elegance is a secondary consideration.

BIBLIOGRAPHY

1. Bryan, G.E. JOSS: User Scheduling and Resource Allocation. The RAND Corporation. Memorandum RM-5216-PR. January, 1967.

2. Corbato, F.J. System Requirements for Multiple Access, Time-Shared Computers. Project MAC, Massachusetts Institute of Technology. Report MAC-TR-3. 1964.

3. Dennis, J.B. Segmentation and the Design of Multiprogrammed Computer Systems. Journal of the Association for Computing Machinery, v. 12, no. 4, October, 1965: 589-602.

4. Fano, R.M., and F.J. Corbato. Time-Sharing on Computers. Scientific American, v. 215, no. 3, September, 1966: 128-140.

5. General Electric Time-Sharing System Manual. The General Electric Company. CPB-1182A. February, 1966.

6. Gibson, C.T. Time-Sharing on the IBM System/360:Model 67. Proceedings of the Spring Joint Computer Conference, 1966. Spartan, 1966.

7. Glaser, E.L., J.F. Couleur, and G.A. Oliver. System Design of a Computer for Time Sharing Applications. Proceedings of the Fall Joint Computer Conference, 1965. Spartan, 1965.

8. Hatch, R.R. An Investigation of Program Exchange Methods for a Multi-programming Environment. Naval Postgraduate School M.S. Thesis. 1964.

9. Kilburn, T., D.B.G. Edwards, M.J. Lanigan, and F.H. Summer. One-Level Storage System. Institute of Radio Engineers Transactions on Electronic Computers, v. EC-11, no. 2, April, 1962: 223-235.

10. Licklider, J.C.R. Man-Computer Symbiosis. Institute of Radio Engineers Transactions on Human Factors in Electronics, v. HFE-1, no. 1, March, 1960: 4-11.

11. Lindgren, N. Human Factors in Engineering. Part II - Advanced Man-Machine Systems and Concepts. IEEE Spectrum, v. 3, no. 4, April, 1966: 62-72.

12. McGee, W.C. On Dynamic Program Relocation. IBM Systems Journal, v. 4, no. 3, 1965: 184-199.

13. SDS MONARCH Reference Manual, 900 Series/9300 Computers. Scientific Data Systems. Publication 900566B. August, 1965.

14. SDS MONARCH Technical Manual, 900 Series/9300 Computers. Scientific Data Systems. Publication 900616B. October, 1965.

15. SDS Real-Time MONITOR Reference Manual. Scientific Data Systems. Publication 901108A. February, 1966.

16. SDS SYMBOL and META-SYMBOL Reference Manual. Scientific Data Systems. Publication 900506E. October, 1966.

17. SDS 930 Computer Reference Manual. Scientific Data Systems. Publication 900064D. February, 1966.

18. SDS 940 Computer Reference Manual. Scientific Data Systems. Publication 900640A. August, 1966.

19. System/360 Model 67 Time Sharing System Preliminary Summary. International Business Machines Corporation. C20-1647-0. 1966.

20. Time-Sharing System Scorecard, No. 4. Computer Research Corporation. Fall, 1966.

APPENDIX I

SUMMARY OF MAJOR DEFINITIONS

Batch-processing	-	that operation of a computing system in which programs are collected into groups, or batches, and are then processed from start to finish without programmer intervention.
Demand paging	-	a page-turning algorithm in which program pages beyond the current one are brought from secondary storage into main memory only when referenced.
Logical address	-	a memory address as contained within a program; when relocation is used, that which is translated into a current physical storage location.
Multiprogramming	-	that operation of a computer which permits the execution of a number of programs in such a way that none of the programs need be completed before another is started or continued.
Program relocation	-	within a computing system, the physical movement of programs and translation of program-contained memory addresses into actual storage locations.
Project MAC	-	the on-line, multiple-access, time-sharing computing system of the Massachusetts Institute of Technology.
Real-time computing	-	program execution to satisfy a particular operational response time, which ranges in different applications from microseconds to minutes.
Re-entrancy	-	a characteristic of a program which can be executed for more than one user concurrently; meaning, there is no internal data storage or address modification which will affect results if a second user enters the program before a first has finished.

Time-sharing	-	that operation of a computing system which permits a number of users to employ it simultaneously in such a way that each is or can be completely unaware of the activity of the others.
Virtual memory	-	or address space; a term for the maximum addressing capability of a computer, not all of which is necessarily implemented in physical storage.

APPENDIX II

ON RELOCATION

IN THE DIGITAL CONTROL LABORATORY

The purpose of this appendix is to discuss further the implementation of program relocation on the new computer in the Digital Control Laboratory. Taken as a point of departure is the recommendation made in Section 6 that "no relocation", or job swapping, be the method used. Certain assumptions relative to a time-sharing operating system for the D.C.L. are described, and a Program Status Table to be used during relocation is defined. Timing considerations and memory protection requirements are discussed. Finally, charts showing the possible flow of relocation are presented.

A fundamental premise is that the operating system will be designed initially, within the goal of providing time-sharing between the two displays and other peripherals, to use as many portions as possible of Disc MONARCH or Real-Time MONITOR. This assumption was certainly a consideration leading to the decision to recommend "no relocation", and it seems quite reasonable in view of the limited amount of time which is available among D.C.L. users for writing a new operating system. It does, however, place restrictions upon the philosophy of operation. In particular, it implies that each user program while executing will have the computer to itself, as far as core memory is concerned, save the portions reserved for the operating system's resident and for special functions such as the display buffers. This will provide the closest emulation to standard, non-multiprogrammed use of MONARCH/MONITOR. Further, programs are to be formed - subroutines linked, and a copy of all common routines attached - before execution. No attempt is to be made to

refer to a single copy of common matter. With these restrictions placed, it becomes possible to hope to employ the MONARCH/MONITOR language processors, such as FORTRAN and META-SYMBOL, and input/output drivers with relatively few modifications.

The scheduling program (called here, SKED[31]) is to be the dominant routine within the new operating system. This is a reasonable assumption consistent with its responsibility for controlling the flow of jobs through the computer. It is further assumed that SKED will be the first system routine entered when execution of one user program is interrupted, and the last routine employed before control is transferred to the next user program. Thus SKED will be in a position to oversee the housekeeping and other services performed between user program quanta. Figure 17 is a representation of how processing might flow in the computer. Further, SKED must be responsible for storing the old user's machine conditions - in the D.C.L. SDS 930, this means the contents of the A,B,X, and P registers, and the status of the overflow indicator - and for setting the conditions for the new user. These actions will be the first and last tasks, respectively, performed during the service period between user program quanta.

The disc will be used for storage of both temporary and permanent files. The relocation program, named RELOC, controls the temporary section, employed to hold the core images of programs which have been interrupted prior to their completion. The permanent portion contains the non-resident parts of the operating system, including the language processors. If space permits, this portion may also hold certain user

[31]Program names used in this appendix are chosen only for their brevity, consistent with some amount of meaningfulness.

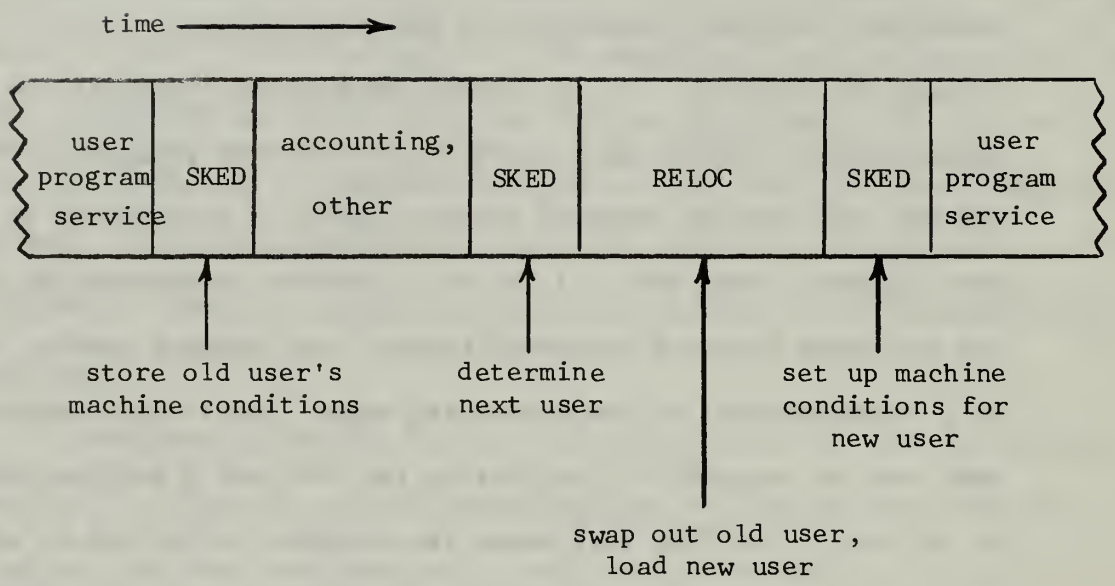


Figure 17. Flow of processing within D.C.L. SDS 930

programs in an inactive status, such as between working sessions on a course or thesis project. Figure 18 shows symbolically the organization of the disc.

To the operating system, each user will be, in fact, a program. Thus a person using the computer from, say, a display console is known by the name of his program,[32] not by some other means such as his own name or console number. If he is assembling or compiling before executing, his program name will first identify his copy of the language processor which he is using. Later, it will refer to his object program in execution.

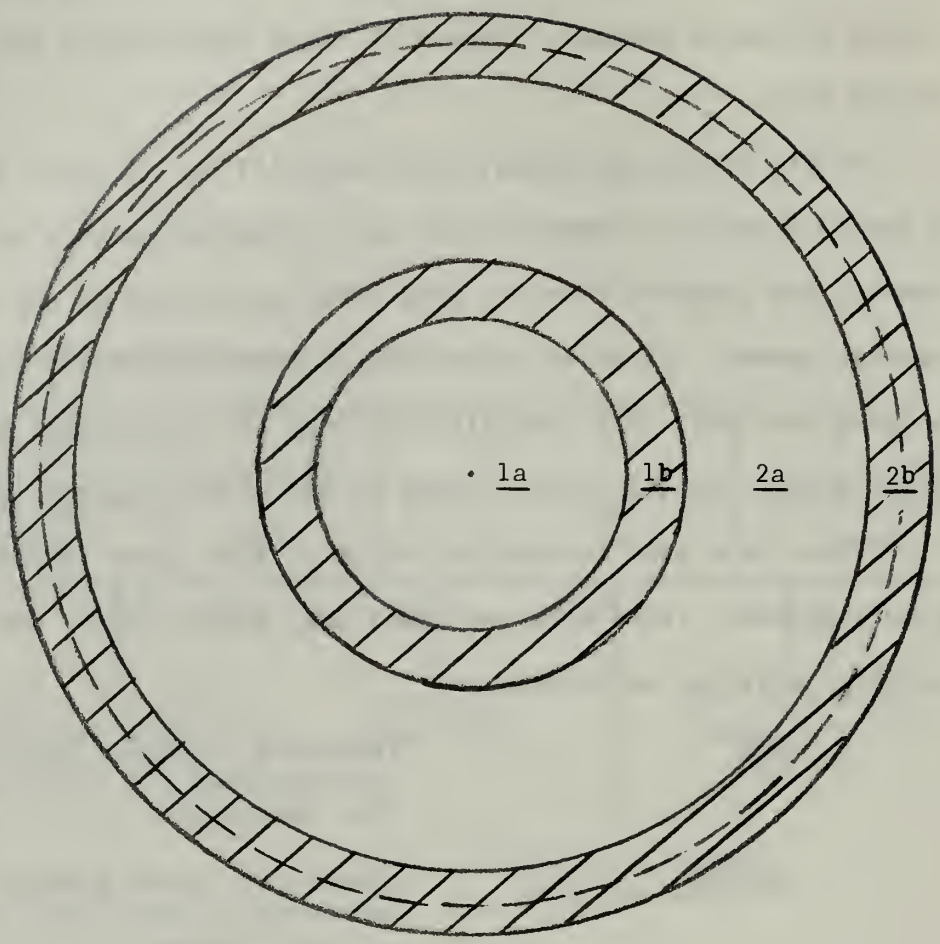
There is a need for definition of at least four different user program statuses. These might be named NEW, ACTIVE, DEAD, and SAVE. Their meanings would be as follows:

NEW	-	indicates a program which has not yet received its first quantum for execution
ACTIVE	-	refers to a program which has executed at least once, but which is not finished
DEAD	-	this program has finished, and its core image may be discarded
SAVE	-	this program has also terminated, but it is desired to store its core image for future use

The different operating system programs will use these status indicators to determine which of the possible alternatives open to them will be followed as they perform their functions.

Affected by the above will be entries in a Program Status Table established and used jointly by SKED and RELOC. Each user program will

[32]Or, equivalently, by a number assigned to his program by the system. Using a number might save space in the Program Status Table (q.v.), but it also implies some name-to-number and number-to-name translation.



1 - Temporary section

1a - temporary files
(core images of interrupted programs)

2 - Permanent section

2a - permanent files

2b - catalog (two parts, as indicated by dotted line, one for system programs and one for saved user programs)

Figure 18. Organization of the disc

be included in this table from the time when it is NEW until it becomes either DEAD or SAVE. Permanent entries will be present for operating system language processors. The Program Status Table will be a part of the core resident portion of the operating system. A complete entry for a program will contain the following items:

a) program name

b) size of core image

c) first word address when loaded[33]

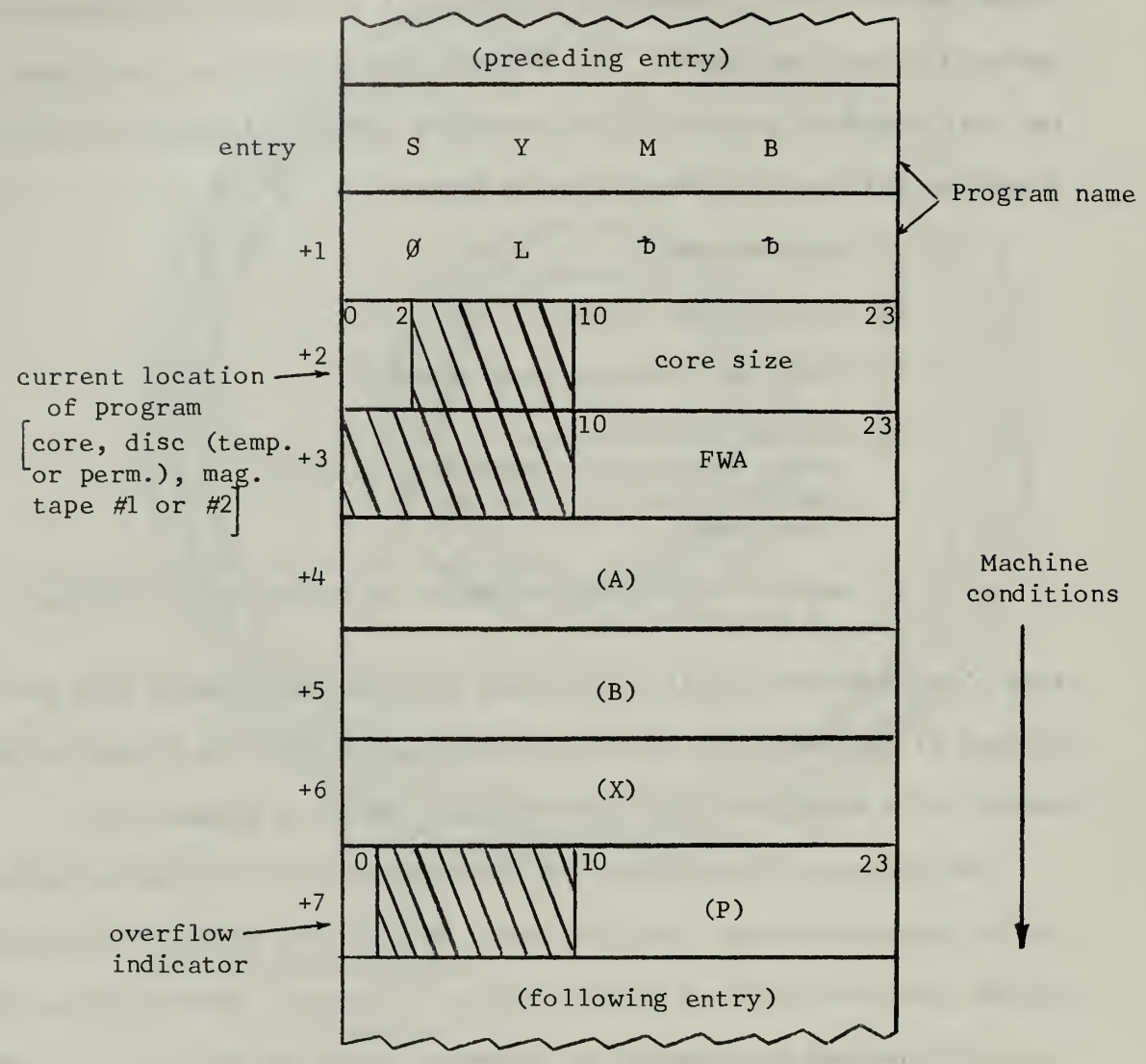
d) current location, i.e.
disc, temporary or permanent section
magnetic tape no. 1 or no. 2
core memory

e) machine conditions to be set up before next execution of program.

Item e) will be principally maintained by SKED, while RELOC will use a) through d) in performing program relocation. A possible format in core memory for a Program Status Table entry is shown in Figure 19.

The principal disadvantage of job swapping as a relocation method is the program exchange time involved. With so few users in the D.C.L. system, the time required here should be tolerable. Nevertheless, it is certainly desired to minimize the overhead caused by relocation. Most of this overhead will be due to transfers between disc and core. In turn, the time taken for these transfers depends upon two factors, the transfer rate and the rotational latency of the disc. The former is a fixed quantity, and the time required for actual transfer cannot be submerged since the disc is attached to the cycle-stealing time-multiplexed communication channel. The effect of the rotational latency, however, can be reduced.

[33]Normally, the FWA will be fixed, and thus could be eliminated, for all user programs. However, it may vary for the language processors and is therefore included.



Shaded areas show bits reserved for table expansion/evolution

Figure 19. Format of Program Status Table entry

This is possible because as the disc rotates, the current sector address can be read into the computer at any time.[34] Thus RELOC may set up a transfer but not initiate it until the proper sector is under the read/write heads, provided that other useful functions are available to be performed in the meantime. Or, RELOC may alter the order of transfer of the words within the block being moved to take advantage of the current sector location. This means, perhaps, that a transfer may be initiated with the middle of the block, rather than its beginning.[35] The added coding complexity should be worth it in either case, considering that the average rotational latency of 17.5 milliseconds is as long as the actual transfer time for a program of 2,048 words. An attempt has been made in the relocation routines presented in this appendix to follow the set-up of a transfer operation with another function which might be accomplished while waiting for the disc to rotate to the proper sector. However, since latency times are measured in milliseconds, the provision of enough functions to submerge a major part of the expected time in this way would certainly involve use of other operating system programs not discussed within this thesis. It is difficult to say more about timing until details are known for the disc input/output handler to be furnished with Disc MONARCH and Real-Time MONITOR. This routine, it is assumed, will be investigated for possible use in the D.C.L. operating system, and any employment of it may well affect time considerations.

A modest form of core memory protection would be very desirable, even in the D.C.L. system where only one user program is to be in core

[34] SDS 940 Computer Reference Manual, p. 77.

[35] Ibid. An example of the use of this technique is given. As 547 microseconds are required for one sector to move by the read/write heads, there is considerable time available for block manipulation.

at any one moment. This is so that user programs do not inadvertently destroy parts of the operating system resident, thus preventing continuous operation of the time-sharing system. The experience of the System Development Corporation in this respect is informative; with no memory protection, the AN/FSQ-32 time-sharing system never ran for longer than ten minutes before a user destroyed a portion of the resident executive. Bounds registers are now installed, limiting the range of access of each user program.[36] One way to provide protection of specified areas of core memory in the D.C.L. system would be to purchase the SDS 930 memory write lock-out feature. This option allows program- or manually-controlled prevention of writing into any or all 512-word blocks in core; when an attempted violation occurs, a "no operation" and trap to a fixed location result.[37] Addition of this feature would, of course, involve extra expense. It would also be possible to design an implementation of memory protection at the Naval Postgraduate School. One method would involve a single bounds register. This method would be feasible if all that is to be protected - the operating system resident and any other reserved areas - is located either above or below, in core memory, the user program area. The bounds register, functioning for specified operation codes when user programs are executing, would cause a trap whenever the instruction address specified a location within the protected area. Figure 20 shows an allocation of core memory in which the resident and its tables, forming the protected area, are placed at the uppermost addresses. This method of protection is quite restricted in flexibility, but its relative simplicity is in keeping with the goals of the D.C.L. system.

[36]Interview with E. Myer, System Development Corporation, Santa Monica, California, 14 February 1967.

[37]SDS 930 Computer Reference Manual, p. 4.

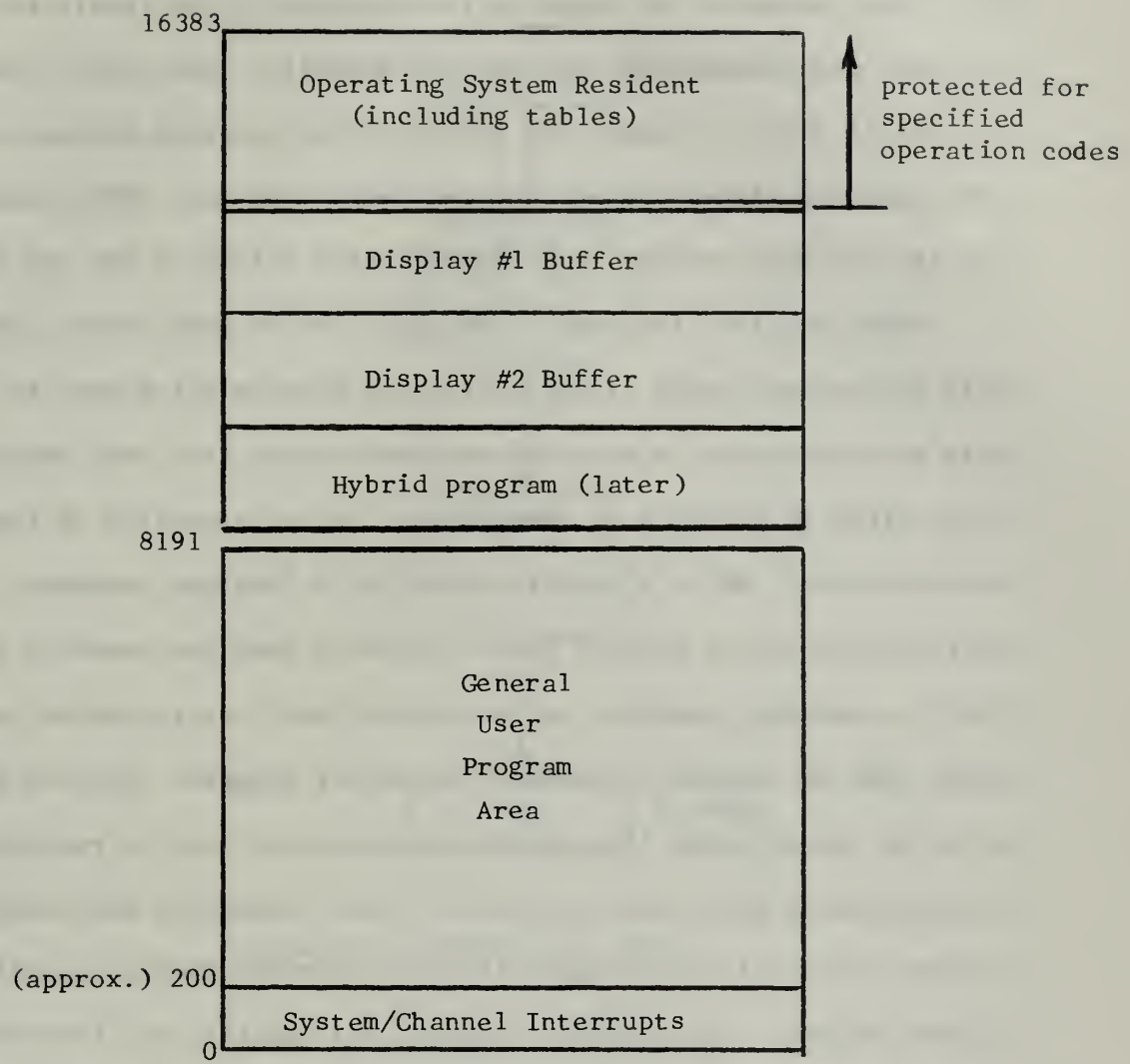
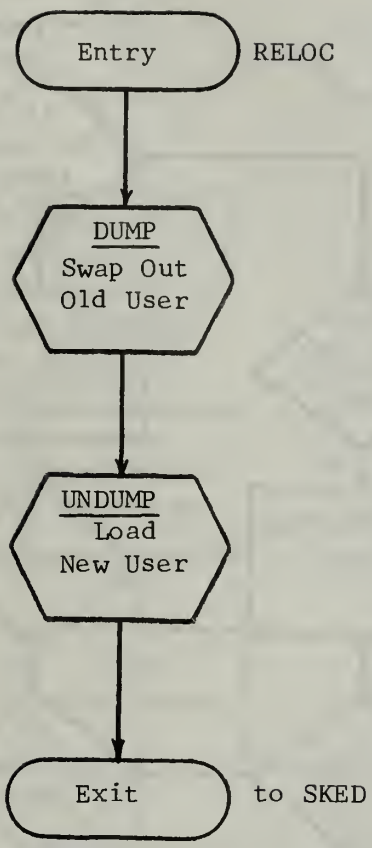


Figure 20. Possible allocation of core memory in D.C.L. SDS 930

The routines suggested for RELOC are charted in Figures 21 through 25. They implement job swapping in keeping with the thesis recommendation and with the assumptions made in this appendix. The overall relocation process is shown in Figure 21, including the entering arguments from SKED. The remaining figures depict the two major routines, DUMP, which swaps out the old user program, and UNDUMP, which brings in the new one.

RELOC routines frequently reference the Program Status Table. In this connection, there is one particular problem which must be solved. This problem is how to make the required change to a user program PST entry after an assembly or compilation, before execution of the object code produced. While a user is employing a language processor, the PST entry refers to his copy of that processor; when the assembly or compilation is complete, however, he is finished with the processor, and his PST entry must be changed to reflect the object program. How and when this is to be accomplished is a system problem which must be resolved. One solution would be to add at the end of each assembler and compiler a short routine which will investigate its binary output medium, on which is the object program. The program's length, and starting and transfer addresses could be located there, and with these known, the required PST entry could be created.

One final matter to be determined at the system level is design of the loading programs. Each of the operating systems being furnished with the D.C.L. SDS 930 already includes one or more of these. Tape MONARCH, for example, incorporates two loaders. One loads binary object programs, including the output of the SYMBOL and META-SYMBOL assemblers; the other handles previously compiled FORTRAN programs. Both are designed to accept



Arguments to RELOC from SKED:

Old User status

New User name

Old User's SAVE location)
) "0" if not applicable
New User's system program)
(when NEW)

Figure 21. D.C.L. relocation overall

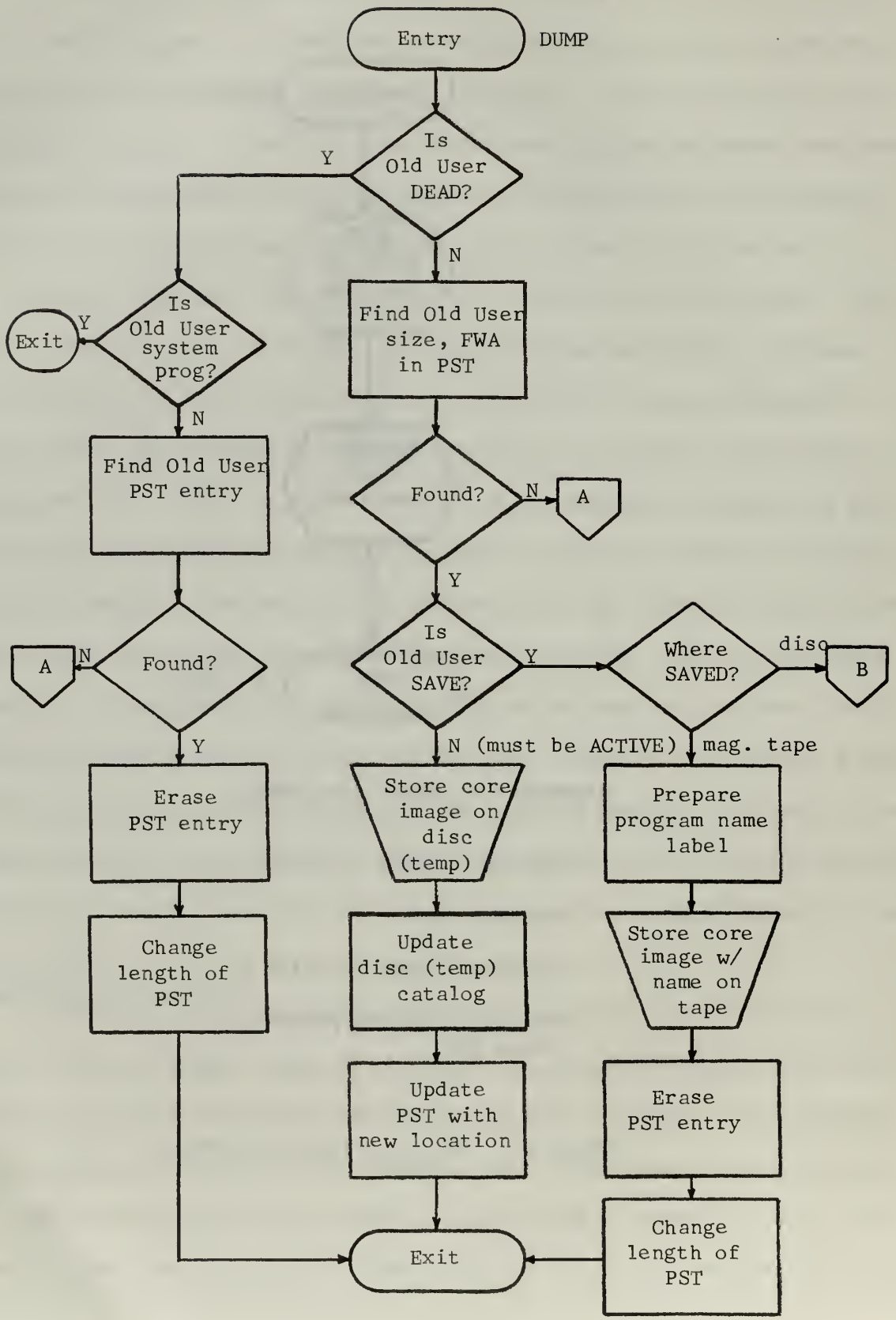


Figure 22. Swap out of old user

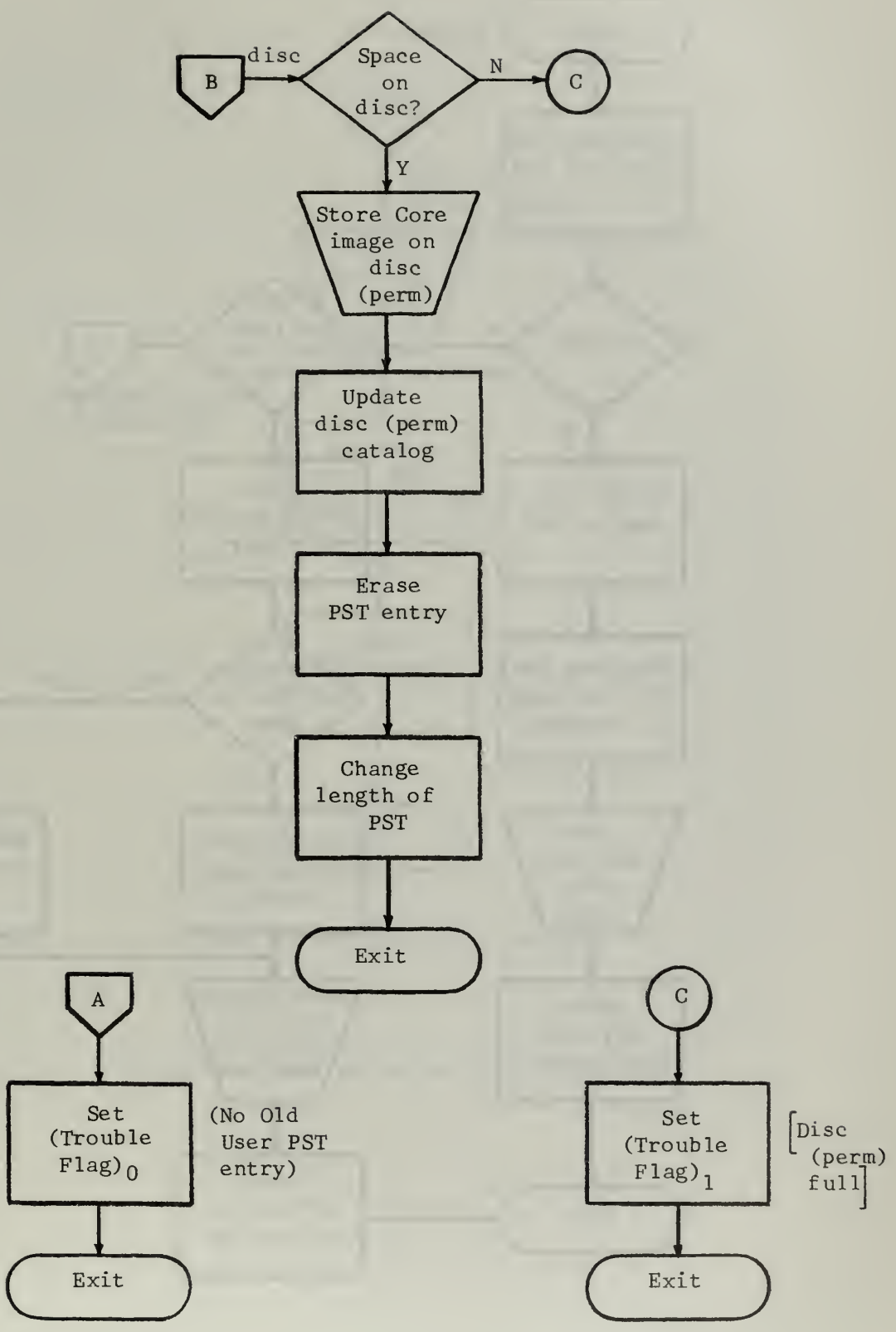


Figure 23. Swap out of old user (continued)

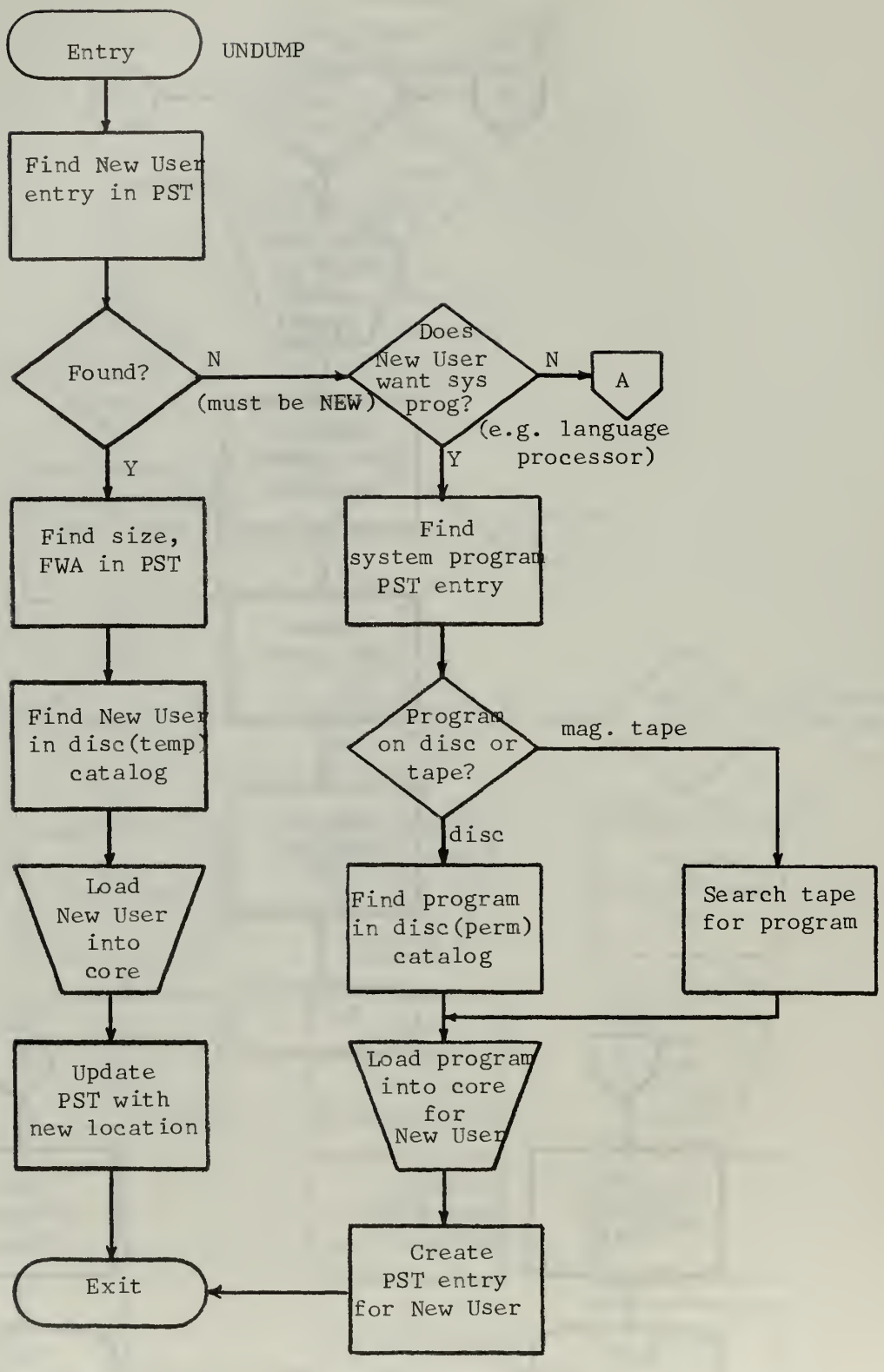
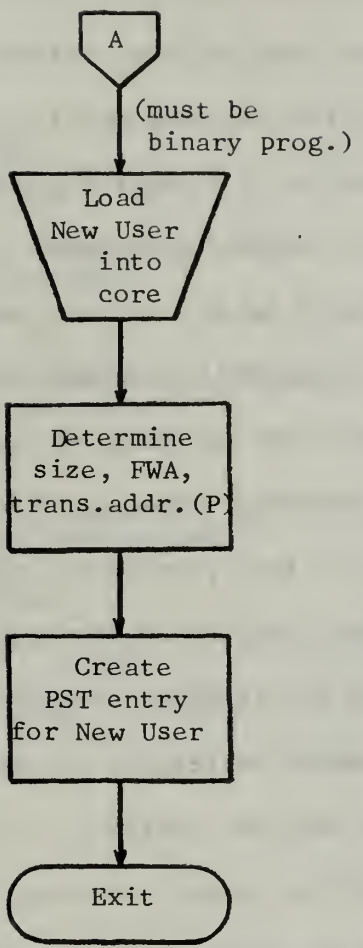


Figure 24. Loading of new user



(P) - Contents of P register

Figure 25. Loading of new user (continued)

files written in the relocatable standard binary language of SDS.[38,39] References to programmed operator and FORTRAN library routines are satisfied by attachment of copies at load time. The Disc MONARCH/Real-Time MONITOR loaders, when written, will doubtless have similar features. In the D.C.L., all the features mentioned will be necessary, especially for the initial loading of a program previously compiled or assembled by one of the standard language processors. After that, there are no further external references to be handled, and in the proposed system, no requirement for relocatability within core memory. A simple, absolute loader will handle the movement of core images in and out of main memory during program exchange subsequent to the first one. It will probably not be feasible to hold permanently resident in core memory a powerful relocating loader, because of its size; the binary object program loader of Tape MONARCH, for example, occupies about 1480_{10} words.[40] Thus the D.C.L. system should anticipate the use of a short, resident, absolute loading program whenever possible, calling upon a non-resident relocating loader only when its special features are required.

There are many problems to be solved before the Digital Control Laboratory will have a functioning time-shared computing system. If, however, job swapping is chosen as the method of program relocation, it is believed that a reasonable start has been provided for its implementation in this system.

[38]SDS MONARCH Reference Manual, 900 Series/9300 Computers, p. 59.

[39]SDS SYMBOL and META-SYMBOL Reference Manual (SDS Publication 900506E, October, 1966), p. 66.

[40]SDS MONARCH Technical Manual, 900 Series/9300 Computers, p. 38.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Documentation Center Cameron Station Alexandria, Virginia 22314	20
2.	Library Naval Postgraduate School Monterey, California 93940	2
3.	Commandant of the Marine Corps (Code A03C) Headquarters, U.S. Marine Corps Washington, D.C. 22214	1
4.	Commandant of the Marine Corps (Code AP) Headquarters, U.S. Marine Corps Washington, D.C. 22214	1
5.	Professor Mitchell L. Cotton Department of Electrical Engineering Naval Postgraduate School Monterey, California 93940	3
6.	Captain James J. Stewart, USMC 1209 Durham Road Madison, Connecticut 06443	1

DOCUMENT CONTROL DATA - R&D

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

1. ORIGINATING ACTIVITY *(Corporate author)*
Naval Postgraduate School
Monterey, California 93940

2a. REPORT SECURITY CLASSIFICATION
UNCLASSIFIED

2b. GROUP

3 REPORT TITLE
PROGRAM RELOCATION IN A MULTIPROGRAMMING ENVIRONMENT

4 DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
Thesis, M.S., June 1967

5 AUTHOR(S) *(Last name, first name, initial)*
STEWART, James J.

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
June 1967	102	20

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
b. PROJECT NO	
c.	9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*
d	

10. AVAILABILITY/LIMITATION NOTICES
[redacted]

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY

13. ABSTRACT

Various methods are studied for the relocation, or movement, including address mapping, of programs within a multiprogrammed digital computer. The aim of doing so is to determine the best method for use in the limited time-shared computing system proposed for development in the Digital Control Laboratory of the Naval Postgraduate School. In this light, the concepts of time-sharing and multiprogramming are discussed, as is the implementation of relocation in a very large computer obtained for the School's main computer facility. The features and requirements of the D.C.L. are then established and evaluated. It is found for the Laboratory that complete job swapping will be a fully satisfactory method of relocation. The time taken will not be excessive, and this method will be the easiest to incorporate in the time-sharing system. Details of a possible implementation are given in an appendix to the thesis.

DD FORM 1 JAN 64 1473

14 KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Computers, digital						
Time-sharing						
Multiprogramming						
Program relocation						
Addressing methods						