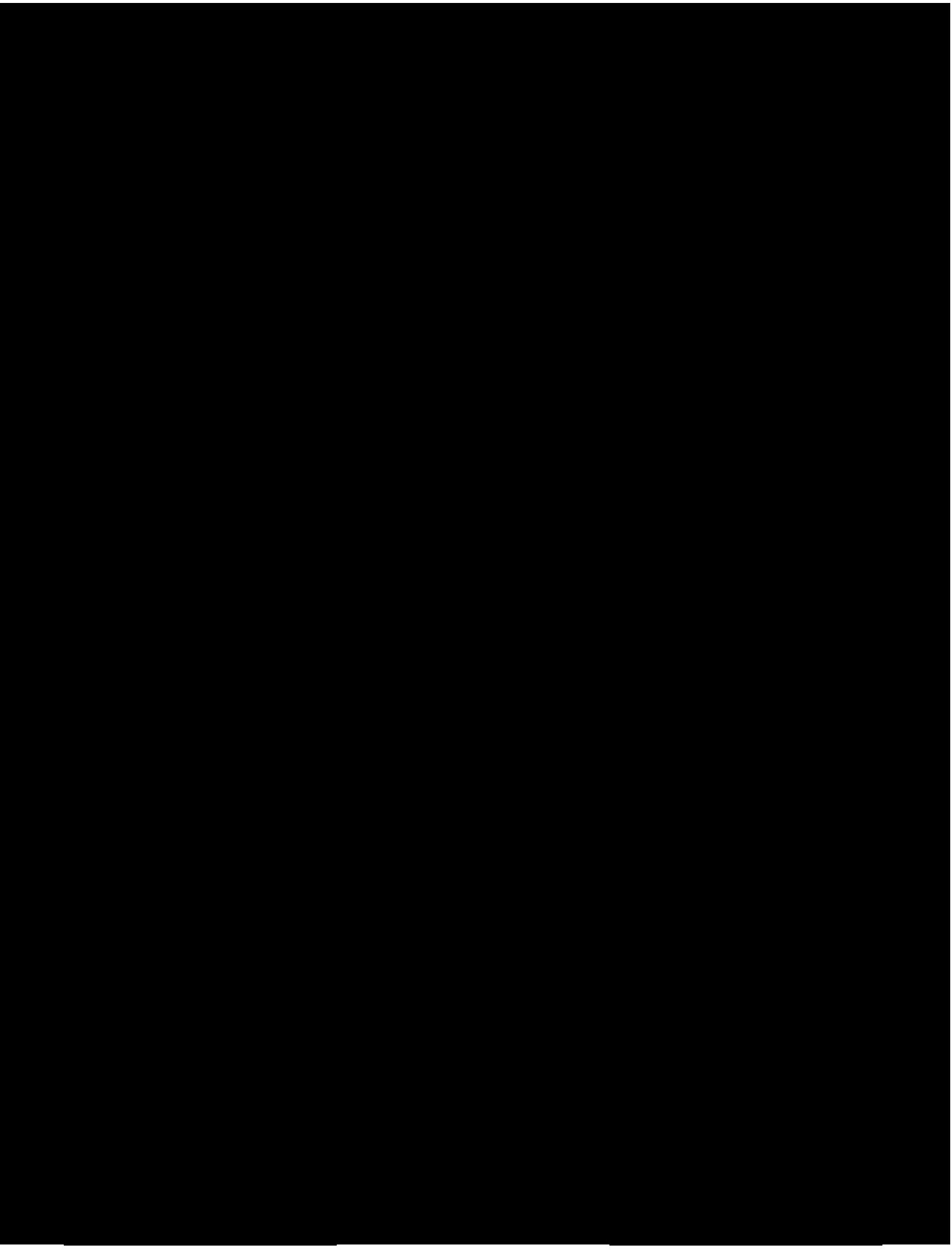

*This blank page was inserted to preserve pagination.*

MAC-TR-3

MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

PROJECT MAC

# SYSTEM REQUIREMENTS FOR MULTIPLE ACCESS, TIME-SHARED COMPUTERS

by

F. J. CORBATÓ

COMPUTATION CENTER

Work reported herein was supported (in part) by Project MAC, an M.I.T. research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Number Nonr-4102(01).
Reproduction in whole or in part is permitted for any purpose of the United States Government.

## Introduction

It is now clear that it is possible to create a general-purpose time-shared multiple access system on most contemporary computers (especially after minor but basic modifications are made). Already the IBM 7094[1], the DEC PDP-1[2], and the Q-32[3] computer have been fully time-shared; somewhat more limited forms of time-sharing are done on the CDC G21[4], the Johniac[5] and IBM 7040[6]; and in the future there will probably be time-sharing on the GE 215[7] and the DEC PDP-6[8].

However, it is equally clear that none of the existent computers are well designed for multiple access systems. The paths of information flow between user and main memory and between main memory and secondary memory, tertiary memory, etc. are not only often difficult to program but in many cases non-existent. Further, for a variety of reasons, it is extremely difficult to multi-program current machines effectively, so that much of the equipment is wasted. At present, good service to a few dozen simultaneous users is considered the state-of-the-art. But the need at most computer installations is for several hundred simultaneous users on a single computer system!

In the early days of computer design, there was an elementary concept of a single program in a single machine computing furiously for large periods of time with almost no interaction with the outside world. Today such a view is obsolete. The following phenomena are typical of contemporary computer installations.

First there are incentives for any organization to have the biggest possible computer that can be afforded. This is because only on the biggest computers are there the sophisticated programming systems, compilers and features (such as main memory space) which make a computer "powerful". This is in part because it is difficult to do a great deal of system

1. MIT Computation Center and Project MAC, IBM (A. Kinslow)
2. Bolt, Beranek and Newman (S. Boilen) and MIT (J. Dennis)
3. System Development Corporation (J. Schwartz)
4. Carnegie Tech (A. Perlis)
5. Rand Corporation (C. Shaw)
6. Belcomm, (A. Speckhard), IBM (J. Morrisey)
7. Dartmouth (T. Kurtz)
8. The Digital Equipment Corporation is the first manufacturer to announce that they will produce a general-purpose multiple console time-sharing system.

assistance when limited by computer speed or size and in part because there is more incentive for all concerned to provide assistance on the larger systems. Moreover, by combining resources in a single computer system, rather than in several, one would expect to get bulk economies and therefore the lowest computing costs. Finally, as a practical matter, floor space, management efficiency and operating personnel provide a strong reason for a single large computer to minimize administrative requirements.

The second phenomenon of a computation center, is that currently there is need for capacity growth due to the increased demand for computers in nearly every aspect of modern life. Multiple access computers promise to accelerate this growth since they lower the barrier of man-machine interaction rate by at least two orders of magnitude. Present indications are that multiple access systems of only a few hundred simultaneous users generate a demand for computation which begins to exceed the speed of the fastest existent single processor computer. Clearly, the only direction left is that of multi-processors since the speed of light and the physical sizes of computer components are an intrinsic limitation on the speed of any single processor. (Of course, multiprocessors allow more reliability and easier steps for growth of capacity, but the speed argument is paramount.)

A third phenomenon that has arisen is that programs interact frequently with secondary storage devices and with the on-line users themselves. This communication traffic produces two major effects: A need for a variety of input-output channels and a need for multiprogramming to avoid wasting main processor time while an input-output request is being completed. The important thing to note though is that the individual computer user is ordinarily incapable of doing an adequate job of multiprogramming since his own program lacks the proper balance and he probably lacks the ingenuity (or patience).

Finally, the experience of a year's operation of the Project MAC system shows that the value lies not only in

providing, in effect, a private computer to a number of people simultaneously, but, above all, in the services that the system places at the finger tips of the users. Since the effectiveness of a system increases as the service facilities are shared, a major goal of future research is a system with multiple access to a vast common structure of data and program procedures; the achievement of multiple access to the computer processors is but a necessary subgoal of this broader objective. Thus the usefulness of a multiple access system depends almost entirely on programs; correspondingly, the memories where programs reside play a central role in the hardware organization.

In summary, then we see how the original view of a single program on a single computer, has been replaced by a large system of many components and a community of users. Thus, we have a multi-user, multi-processor, multi-channel system. Moreover, each user of the system asynchronously initiates jobs of arbitrary and indeterminate duration which subdivide into a sequence of processor and channel tasks. It is out of this seemingly chaotic, random environment that we finally arrive at a public utility-like view of a computation center. For instead of chaos, we can average over all the different user requests to achieve nearly total utilization of all resources. The task of multiprogramming required to do this need only be organized once in the central supervisory program. Each user thus enjoys the benefit of efficiency without the inelegance and pain of trying to average the demands of his own particular program.

The above "exploding popcorn" view of computer use, where tasks dynamically start and stop every few milliseconds and where the memory requirements of tasks similarly grow and shrink, means that one of the major jobs of the supervisory program is the allocation and scheduling of computer resources. The general strategy is clear. Each user's job is subdivided into tasks, (usually as the job proceeds), each task of which is placed in an appropriate queue (i.e. for a processor or a channel). Processors or channels are in turn assigned new tasks as they either complete or are removed from old tasks. All processors should be symmetric and be assigned as needed to the program for tasks; in particular, the supervisor need

not have a special processor. Processors should be able to be added or deleted without any significant change in either the user or system programs. Similarly, the channels should be symmetric and should be logically and physically independent of the processors. Again, as with the processors, one should be able to add or delete a channel according to system load (or reliability) without any reprogramming required.

The above system viewpoint, offers a clear-cut approach to the multi-user, multi-processor computer. However, there are many interrelated problems and requirements which remain to be discussed, namely: clocks, memory protection, program relocation, parallel tasks within a job, common simultaneous use of sub-programs by many users, growth and shrinkage of program segments, and memory allocation.

Clocks

Of course, the most elementary clock required for an operating computer system is a clock for documentation purposes, giving the date-month-year and time-of-day sufficiently accurate so as to be unique. Most contemporary computers have not had such a clock and it must be added on as an expensive accessory. As computer operation becomes round-the-clock and nearly continuous, the need for automatic operation of the date mechanism of the clock becomes a necessity.

A second kind of clock required in any time-shared system, is an interval timer clock which can be enabled and disabled with remembered interrupts. Clearly this clock is needed if the supervisor program is to maintain control over user program loops (intentional or accidental) and successfully schedule the computer time resources (processors and channels) as well as periodically service certain housekeeping functions such as maintaining character flow to and from the user typewriters. Although in many instances interrupt logic can be used to trigger off supervisor program housekeeping, there may be an efficiency trade-off with clock-instigated polling techniques where all users are periodically scanned and given service as needed. The pertinent factors in the trade-off are: the relative probability of user activity in the polling period and whether or

not the frequency of clock interrupts maintains "flicker-free" service.

The resolution of an interval timer need not, of course, be any finer than a basic memory cycle, or for that matter, the time to store a user's program status (i.e., 5 to 500 memory cycles, depending on the machine). However, the timer should certainly allow the supervisor program to interact smoothly with the input characters from a user (i.e., a resolution of at least 100 m.s.). When trying to analyze, monitor, and debug input-output programs, there are often cases where it is convenient to measure I/O times accurately, so that precision in the order of .1 to 1 m.s. is required. Finally, as fully multiprogrammed systems involving multiple simultaneously operating user programs come into being, the only accurate way to account and charge for the use of the various processor and I/O resources will be to maintain microscopically accurate accounting. Thus, timer clocks which resolve down to a few memory cycle times are a reasonable requirement for future computers.

## Memory Protection and Supervisor Mode

The most elementary form of memory protection required is that which prevents a user from writing outside of his own program area. If one makes the assumption that a user's program consists of one solid, contiguous region, then an adequate solution is boundary registers such as those on the IBM Stretch, the IBM 7094's at MIT, and the CDC 3600. The boundary protection registers need only resolve to the size of a physical block of words. Clearly the user program must not be able to issue instructions to modify the protection registers (or many other instructions such as those for input-output) and this is accomplished by always operating the user program in a different mode than that of the supervisor program; thus, any mistake occurring in the user's program causes the processor to be trapped to the supervisor program. The hardware should be such that following a protection trap, the program can be continued in case the supervisor was running only a partially loaded program. When there are multi-processors operating, there must also be provision in the hardware for tie-breaking in the case where two processors attempt to enter the supervisor

simultaneously. In general, each processor must also have separate boundary registers, including channels (unless their command sequences are prechecked for validity).

Although simple system operation requires only write protection of memory, and there are cases where several programs might share a read-only data base, the complete memory protection solution demands that there be the option of read protection as well. Two important cases require it: the first case arises in debugging where a program reading beyond its bound has erratic behavior which in practice is indistinguishable from transient hardware failure. The second case is simply that of user privacy. It should be clear that the general commercial user has no desire to let his competitor browse in his records, the project manager doesn't want his staff to accidentally see each others personnel records and salaries, the clients of a bank consider their bank balances privileged, and a military agency cannot tolerate a high probability of security violation.

## Relocation

As soon as there is more than one user program in the main memory, dynamically starting and stopping, growing and shrinking, there is the need to move programs about in memory, preferably with low overhead, so as to accomodate new programs which are larger in size than the available holes in memory space. It is important to recognize that in general when a program is interrupted during operation it cannot be arbitrarily relocated even if the original program loading relocation information is still available, because the contents of the accumulator, index register, etc. are of unknown relocatibility. A simple solution to the problem is provided by a relocation register, as in the IBM 7094's at MIT. This register (which only resolves to blocks of words) acts like an extra index register for all address references to memory by the user program. Thus the program, the addresses computed and stored in the program, the accumulators and the ordinary index registers always appear as though the program were operating in a fixed location. Of course, whenever the supervisor moves a program, it also must correctly readjust the corresponding contents of the relocation register.

This kind of a relocation scheme has the following fairly obvious limitations:

1) Whenever programs must be moved either to make room for other programs or to grow themselves, there is a minimum of a read and a write operation for each word moved;

2) programs have at most one convenient edge upon which to grow;

3) there is no simple scheme which allows several user programs to use transparently-written system subroutines simultaneously in common.

Common Subroutines

To see more clearly the nature of the system programming problems which can arise without careful hardware design, let us explore the difficulties with the use of common subroutines in a computer with the basic boundary and relocation registers just discussed. The importance of such common subroutines should vastly increase with multiple-user systems since with only one copy of the system library in core memory at all times there will be memory space savings, program maintenance efficiencies, and improved response time.

Several properties are required of common routines in such a system:

1. A common routine should be interruptible at any time on the same basis as the user's program so that the subroutine task may be of arbitrary length.

2. A common routine must not store any quantities within itself but rather in an area designated by each user program as it enters the routine, because the routine must always be invariant for each user. (As a consequence such routines could be in read-only memory.)

3. A common routine must operate in the supervisor mode since it will not be contiguous with the user program area.

4. As a consequence of (3), common routines must be "completely" debugged. This means not only that the routine must perform correctly for correct input parameters, but also that under no circumstances of false input parameters must the routine take any action which violates the user's basic memory protection bounds.

The last requirements (3) and (4), are indeed stringent ones. In particular, all common routines have to guard against the following typical mistakes:

1. The parameter "call" from a user's program certainly will start within the program area but may not be completely contained in it so that false "call" parameters may be implied.

2. All input or output parameters, which are locations of a variable, must be tested to see if the address lies within the program areas.

3. All input or output parameters which are initial array locations (and thus implicitly help define an array) require a test that the end of the array lies in the user program.

While it is true the above tests can be assisted with auxiliary subroutines, or by preliminary supervisor screening upon entry to the common system routine, the logical burden they place on the programmer of common subroutines is immense. Finally a further complication arises in that the instructions of a common routine want to refer to two areas in memory, namely, their own area for the purpose of reading constants, etc., and the user's program area. Since the common program does not move and runs in the supervisor mode without relocation, it can certainly refer to itself for constants. However, all references to the user program must be in a way such that, if at any time an interruption occurs, the user's program can be moved. This means that the common program whenever it computes or stores away any location references to the user's program it must do so using the unrelocated location values; but, of course, to make actual reference to the contents of these locations,

the common routine must somehow relocate these addresses while remaining interruptible. One solution to the seeming dilemna, which works on the IBM 7094, is for the common routines to only make references to the user's program area using a combination of indirect addressing, index registers and conventions.

The conventions are that all common routines may reference a user's program area only by means of three bases. These bases are:

1. The program area origin, for general reference to the program area.

2. The call origin, for picking up parameters.

3. The location of temporary storage for use by the common routine.

To make actual reference, the common routine must load an index register with the desired relative location complement; a second index register is then loaded with the value of the first index register and finally an indirect reference is made to a table in the supervisor which is always kept updated on any movement of the user program. For example, the following sequence would be used to pick up the contents of the first parameter in a call:

```
AXC    1,5     prepare for first parameter
AXC    5,6     prepare to use IR5 entry in table
CAL*   CLBASE,6pickup parameter
```

```
where CLBASE   SYN    *-1
               PZE    (call base),1   Table updated by supervisor
               PZE    (call base),2   as part of user program
                                      status whenever user program
               PZE    (call base),7   moved or a "call" is made.
```

The above example, still makes no attempt to check location validity. This is done, if a further instruction is inserted just before the CAL:

```
XEC*    CLCHK,6
```

where the following table is, like the CLBASE table, updated by the supervisor whenever the program is moved or a call is made:

```
CLCHK   SYN  *-1
        TXL  (error entry),1,-(program length relative to call)
        TXL  (     "     ),2,-(            "                  )
        ...
        TXL  (     "     ),7,-(            "                  )
```

Similar tables must, of course, be maintained for the other bases. Finally the cases of common routines calling common routines and of common routines calling back into the user's program require special care and further conventions.

It should be clear to the reader at this point that the use of common subroutines on the IBM 7094 with only simple boundary registers and a relocation register:

1. is possible

2. is very involved, and

3. requires a large amount of program overhead in the screening of parameter validity and the maintenance of auxiliary tables.

Nevertheless, the example was followed through in some detail for several reasons. Not only is it valuable to see the many aspects which must be considered, but furthermore, it serves as an object lesson in the dangers of trying to "program around" poor hardware-program interfaces.

Clearly, a simpler solution from a programming viewpoint, in the case of common subroutines, is to add to the hardware a second set of boundary registers and relocation register for the common routine area, along with the appropriate reference instructions between areas. There still are non-trivial conventions and supervisor tables needed, but the requirements of completely debugged common routines and parameter validity

checking are removed. The costly overhead of moving programs, and the requirement that a user's program be in one contiguous block are still present, but these are not removed until a rather complete generalization is made.

In particular[1], programs could be composed of not just one or two segments as we have been considering but of many segments some of which are in common and some of which belong to the user. Moreover if one can proceed to find a reasonably efficient mechanism for associating physical blocks of the memory with the logical pages of the program segments, then no storage allocation movement of programs need ever be done and the reduction in multiprogramming overhead would be immense. It is beyond the scope of this report to discuss these ideas further but their importance should be obvious.

## Relocation of Programs on the IBM System/360

It should be enlightening to consider another example of the basic relocation problems discussed in the previous section. Although, other computers could equally well have been selected, the IBM System/360 will be considered, partly because it will undoubtedly be commonly available and partly because the writer has had occasion to become familiar with it. The processor unit contains 15-24-bit base registers, and it is only through one of these that program addressing may occur. By appropriate conventions, which will be outlined, simple relocatability of programs is possible. It is important to realize that unless special conventions are used, programs in the 360 once started cannot be moved and run elsewhere in memory; this follows because not only are the base registers useable interchangeably as general registers (i.e. accumulators) or as index registers, but their contents, which may contain absolute addresses, may be stored by a user within his own program. Thus the situation without conventions is completely analogous to that which exists in the 7094 without a relocation register.

---

1. These program segment ideas stem from those of Anatol W. Holt, (Comm. of ACM, Vol. 4, No. 10, Oct. 1961) and more recently the ideas of Earl Van Horn who is engaged in doctoral research under the direction of Prof. Jack B. Dennis. Preliminary ideas are discussed in Project MAC Technical Report MAC-TR-11. Discussion with Edward L. Glaser has been especially valuable to the writer.

A set of conventions which allow relocatability follow:

1. Base register 15 will always contain the absolute location of the user's program origin. (Base register 0 would perhaps be more logical but register 0 is assymetrically only a general register; base registers 1 and 2 are unfortunately explicitly used in certain instructions.)

2. To simplify the discussion, it is assumed that the basic addressing complications introduced by the base register-displacement addressing of the IBM 360 are avoided by the definition of an appropriate assembly program. This assembly program would allow the user to write conventional programs completely symbolically addressed, without any consciousness of memory addressing limitations. Upon translation, the assembly program would do a flow analysis of the user's program and then insert instructions for necessary loading, saving and restoring of index registers required for the addressing. The base register used will always be 15, as described. This process is no more complicated than, and is analogous to, the index register optimization in an algebraic compiler. Base register 15 is always used for relocation purposes in all instructions.

3. The user program must always declare to the supervisor, upon any changes, which registers are bases with absolute addresses, and which are general registers or index registers with constants or relative addresses. The declarations to the supervisor must, of course, be done by supervisor subroutine calls.

4. The user program must never read out and store away internally an absolute address contained in a base register; instead the user program must only store, within itself, computed addresses, references to itself, etc. that are in terms of relative addresses.

5. As explained above in (2) all program references are by means of indexed-instruction (i.e. double indexed if one includes the base). The passage of call parameters is only by full-address relative location values. The user, of course,

no longer has normal indexing available. For those instructions which are not indexable, a special macro sequence must be used to establish the appropriate absolute address in a base register, (see below).

6. To establish an absolute address in a base register, the program must go through a macro sequence which:

a. Loads a general register, $r_i$, with the relative address.

b. Copies the contents of base register 15 into the desired base $b_i$.

c. Adds together the registers $b_i$ and $r_i$ and stores as the desired base in $b_i$.

Step (2) of course might be preceded by another base register load instruction furnished by the assembler, or by a suitable declaration call to the supervisor regarding register allocation.

7. To do a subroutine transfer, the user program should never just use the "branch and link" instruction since there is no way for the called subroutine, which may make further nested calls, to preserve the return as a relative location. Thus, before making a subroutine transfer, a program must set a general register with either the relative address of the calling location or of the return. Of course, in practice, specific conventions would have to be created.

The use of common subroutines is similar to the 7094 case discussed previously except that the zone memory protect scheme of the 360 may avoid the need for parameter validity checking. In addition, if the present conventions are extended, the relocation of common routines might be included within the same framework. Finally, if the above technique is used, there would be some inefficiency in the resultant programs from:

a) the overhead required for the assembly program index-base register optimization.

b) or the object program overhead in both size and running time if the optimization of (2) is only partial.

c) the declarations of registers to the supervisor and the absolute address creation.

Nevertheless the resultant convenience and ease of use should more than compensate for the programming system complexity required to "finish the hardware". This provides another example of how relocations can be achieved on a specific machine. Similar problems and similar procedures arise for most machines presently available.

## Conclusion

The above relocation examples and solutions have been elaborated in considerable detail to expose the reader to the difficulties encountered with contemporary machines when multiple user multiple-processor systems are considered. The fact that each program may perform unexpectedly, even to the user , demands that running programs be able to be moved as well as to grow and to shrink. As man-machine interaction becomes faster, each program task becomes more intimately connected with secondary storage and with common subprograms; thus effective multiprogramming is essential for efficient use of a multiple access computer system.

**CS-TR Scanning Project**
**Document Control Form**

Date : 12/ 11 /95

**Report #** LCS-TR-3

Each of the following should be identified by a checkmark:
Originating Department:

☐ Artificial Intellegence Laboratory (AI)
☒ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

## Document Information

Number of pages: 16(21-IMAGES)
Not to include DOD forms, printer instructions, etc... original pages only.

Originals are:
☒ Single-sided or
☐ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☒ Typewriter ☐ Offset Press ☐ Laser Print
☐ InkJet Printer ☐ Unknown ☐ Other:_____

Check each if included with document:

☐ DOD Form ☐ Funding Agent Form ☒ Cover Page
☐ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages(by page number):_____

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

| Description : | Page Number: |
|---|---|
| IMAGE MAP! | (1-16) |
| UNNUMBERED TITLE & SUPPORT PAGE, | 1-14 |
| SCANCONTROL, COVER, TRGT'S (3) | (17-21) |

Scanning Agent Signoff:

Date Received: 12/ 11 /95 Date Scanned: 1/ 19 /96 Date Returned: 1/ 26 /96

Scanning Agent Signature: Michael W. Cook

# Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

**Scanned**

**Date:** 1/19/1996

**M.I.T. Libraries**
**Document Services**