PROJECT STRETCH

FILE MEMORANDUM

SUBJECT: An Automatic Protection and Relocation Scheme

DATE: December 17, 1957

NOTE: This memo stems from discussions between the authors on December 3, 1957

---

1. Objectives

Two important considerations in deciding whether the Stretch computer system can be multiprogrammed in a general and practical way are automatic protection and relocation.

1.1 Need for Automatic Protection

We shall assume that two or more programs are sharing memory and their execution is to be interleaved and overlapped. Further, we shall assume that these programs were written either without the knowledge that they were to be operated together or not making use of this knowledge. These assumptions represent the normal circumstances in which Stretch users will be multiprogramming, if they multiprogram at all.

Consider now the environment in which any one of these programs is operating: this environment can be thought of as a system consisting of the machine, the supervisory program, and the other programs with which it is sharing memory space and CPU time. The reliability of this system (defined as the probability of running without error for a given period of time) is the product of the reliabilities of its components; that is, the reliability of the machine itself is multiplied by that of the supervisory program and by that of each program being time-shared with the given program. It is felt that a supervisory program can be developed with a degree of reliability that can for all intents and purposes be taken as unity, and hence ignored.

What, then, can be said of the reliability of the problem programs themselves? The chances that one of these will clobber another or clobber the supervisory program cannot be ignored, even though we may assume that:

1) each problem program has been "debugged"

2) during assembly a check is made on address parts to determine that there is no obvious reference outside of the area allotted in a relative way to the program.

The reason for this is that debugging is seldom, if ever, 100 percent complete. This is particularly true of problems involving computed references to memory, or indexed references for which the limit or count depends on data. It is just these kinds of references which are likely to go outside of the region allotted to the program as a whole, and it is just these kinds of references which are not detectable at assembly time.

In addition to programming blunders there is always the possibility that a machine malfunction may cause a program P to clobber a program Q. The likelihood of such malfunction may be small, but it is greater the longer a program sits in the machine — and it should be observed that individual programs sit longer when multiprogramming is the order of the day.

Taking both programming blunders and machine malfunction into consideration, we may conclude that, for the basic computer as now defined, a program is operating in a significantly less reliable environment when it is time-shared with other programs than when it is executed by itself.

Let us turn our attention now to fault location. Suppose program P clobbers program Q either in Q's instruction area or data area. It is very likely that P will not discover what it has done if the machine is not equipped with protection facilities. The execution of P could, in fact, be completed successfully if it so happened that Q did not change any of its locations which P had written into. As far as the program Q is concerned, the effect of being clobbered by P is somewhat akin to being stored in an unreliable memory which has no automatic error detection and no customer engineer. It would be satisfactory if there were a reliable, general and efficient programming technique for Q to determine that it had been clobbered. Such a technique does not appear to exist. (The method of forming a check sum for Q whenever it loses the CPU and sum checking Q whenever it regains the CPU is reliable but adds a heavy burden to multiprogramming).

A customer's confidence in using Stretch on a multiprogramming basis will depend on the machine being as alert to clobbering as it is to a simple malfunctioning of memory.

1.2 Need for Automatic Relocation

The primary aim of multiprogramming is to obtain better utilization of the equipment, particularly the CPU. In many installations the amount of memory available will be a limiting factor on the number of programs which can be operated together and therefore on the advantage to be gained by multiprogramming. This is the first reason for requiring that it be possible to pack programs in memory.

The second reason for packing programs is that idle memory locations represent a significant dollar loss in value extracted from a given installation.

How is the packing of programs related to relocation? Suppose we have a memory packed to the hilt with programs and, in a broad sense, start off their execution together. It is most unlikely that these programs will be completed together. As various programs are completed, space in memory becomes available first in one part of memory, then in another. After a while, there will be several scattered areas in memory into which a new problem program would fit, if only these areas were contiguous. In order to collect all the available space into a single area, it is necessary to relocate some or all of the problem programs which are still being executed.

Relocation of programs which are still being executed needs careful consideration from two standpoints:

1) is it feasible?

2) if feasible, is it worthwhile?

With regard to feasibility, it is quite obvious that a program cannot, in general, be relocated at an arbitrary stage in its execution in the Stretch machine as now defined. An address standing in the accumulator, index registers or temporary storage would be overlooked in a programmed relocation procedure unless a relocation bit-map (1 means adjust, 0 means do not) were kept absolutely up-to-date for every use of every location. Absolute up-to-dateness is not quite achievable.

It is natural to suggest that, if programmed relocation is not feasible at an arbitrary stage in the execution of a given program, it should be possible at specified points. In practice, however, such points either do not occur or occur very infrequently unless a programmer takes special pains (and extra program steps) to create them. The number of such points he creates can hardly be legislated and there is little or no incentive for him to create any at all.

With the currently defined machine, we are, therefore, reduced to some such approach as the following: a new program will not be brought into the memory until enough programs have been completed to make available a single area of memory sufficiently large to accommodate the new program. In other words, excluding initial allocation, relocation will be avoided altogether. It is clear that consistent use of this approach results in poorer utilization of the equipment than could be achieved with a machine which provided for relocation in a viable and efficient form.

Another aspect of the need for automatic relocation is the use of overlays: that is, overwriting one portion of a program by another portion of the same program, the new portion being brought in from auxiliary storage. In a multiprogrammed Stretch there would be benefits to be gained by using overlays in some cases where there would be none in a singly programmed machine. Low-duty portions of one program should yield memory space to high-duty portions of another. Relocating programs with overlays during execution is a sufficiently complex procedure in a machine which does not have automatic relocation for the problem programmer to abandon the use of overlays or for the supervisory programmer to avoid relocation.

1.3 Summary of Objectives

The objectives of this proposal may be summarized as follows:

1) to provide a protection mechanism which will confine all memory references made by a given program to a specified area together with those locations whose addresses are less than 64;

2) to provide a selective means of adjusting addresses so that relocation during execution becomes feasible and efficient.

2. Proposal

e, E denote effective addresses respectively before and after superindexing

b, B denote the lower and upper bounds of a given problem program

As presently defined, the machine compares an effective address e with the bounds b, B. The exact nature of this comparison — and of the interrupt which may result from it — have not yet been agreed upon.

In this proposal the effective address e is either superindexed by b or compared with b according as its high order bit is 1 or 0 respectively. The detailed steps are outlined in the flow chart. Superindexing is made conditional because:

1) it will be necessary to relocate control words by traditional (brute force) methods

2) it is desirable to be able to use control words as index words without a prior transformation (specifically, unrelocation).

3. Comments

If it turns out that the superindexing step materially affects the computing speed of Sigma, then it should be suppressed in this system but retained in the Basic system.

Attachment
EFC/jcv

John Batchelder
J. Batchelder

E. F. Codd

W. P. Heising

B. Moncreiff

DISTRIBUTION LIST

Messrs.	J. C. Batchelder
	G. A. Blaauw
	L. Briner
	W. Buchholz
	E. W. Coffin
	J. C. Gibson
	R. Goldfinger
	P. S. Herwitz
	F. E. Johnston
	H. G. Jones
	H. G. Kolsky
Miss	E. McDonough
Messrs.	R. R. Nern
	D. W. Pendery
	C. Scalzi
	D. W. Sweeney
	W. Wolensky