ON USING PAGING

W. M. McKEEMAN

CEP REPORT, VOLUME 2, No. 5

June 1970

The Computer Evolution Project
Applied Sciences
The University of California at
Santa Cruz, California 95060

ABSTRACT

Much of the utility of paging a computer memory depends upon the ability to execute a program with less than all of its data resident in memory. It is suggested that a considerable improvement in performance, that is, in the amount of memory needed to execute a program for a given quantum of time, can be achieved by using the dynamic sequence of initial program load to keep physically close in memory data that will be accessed close together in time.

Paging.

A *word* is an addressable memory unit; a *page* is a contiguous set of $P = 2^p$ words; a *physical memory* is a contiguous set of $M = m \times P$ words; a *virtual memory* is a set of $V = 2^v$ words; an *effective address* is an integer E, $0 \leq E < M$; a *virtual address* is an integer A, $0 \leq A < V$; the set of high order bits of A, $A_h = A \boxed{\bmod} P$, is called the *page selector*; the set of low order bits of A, $A_\ell = A \boxed{\div} P$, is called the *word selector*. We assume $P << M << V$.

There is a mapping function R with arguments $0 \leq A_h < V/P$ such that either:

(1) $R(A_h)$ is detectably undefined or

(2) $R(A_h) < m$, and

$A_h \neq A'_h$ implies $R(A_h) \neq R(A'_h)$.

For any virtual address A such that $R(A_h)$ is defined, $R(A_h) \times P + A_\ell < M$ and therefore defines a unique association between a location in physical memory and the virtual address.

The paging mechanism provides for all of:

(1) Simplicity in allocating memory.

(2) Absolute protection of programs.

(3) Optimal use of physical memory.

(1) is a result of $M = m \times P$. There are only m slots into which a page may be placed. (2) is provided by relocating programs at initial load time so that different programs have disjoint sets of valid vir-

tual addresses. When R is undefined a mechanism must be invoked to change R so that a physical address can be returned; the argument to R is also checked to lie within the allocated virtual memory space. (3) is realized by letting R be undefined for data areas that are not needed at the moment. If memory space is allocated for n programs, the constraint of physical memory can be expressed in terms of the fraction, f_i , of the virtual memory space, V_i , which is mapped onto physical memory for the program:

$$\sum_{i=1}^{n} f_i V_i \leq M \; ;$$

where, on the other hand:

$$\sum_{i=1}^{n} V_i >> M \quad .$$

<u>The</u> <u>Problem</u>.

We assume that the mechanism to evaluate R is economical and that changing R consumes several orders of magnitude more computer resources than evaluating R . If the sequence of access of the i^{th} program to its pages is random in time then we expect that during its run, $f_i \simeq 1.0$. As we cycle through the programs in a time-shared computer, it is clear that if all f_i need be near 1.0 then R will have to be changed often. Our problem then is to force as high a correlation as possible between the dynamic sequence of access and the pages accessed. The question of how much can be gained is left to the experimenter.

A Suggestion.

It is assumed that time-sharing monitors will be carefully adjusted with respect to the criterion above. We address the problem of allocating storage for compiled high-level language programs. Since program structure will provide us with the means of achieving our result, we will concentrate on highly structured languages such as Algol and PL/I.

Within a program there are identifiable logical segments of data which can be expected to have a high probability of use for all words as soon as one is accessed. We present some examples.

Input and output buffers. A buffer is typically treated as a unit; a whole record is moved at once.

The code body of a procedure. When a procedure is called, the instructions are fetched in sequence. Some time is spent within the procedure during which all program fetches come from the body; after the procedure is left, no accesses to the body are possible.

A declared array. An access to an array happens only where it is explicitly named in a program. Typically this is a few places and therefore we expect the accesses to the array to cluster in time around the execution of these parts in the program.

The set of local variables of a procedure. Variables can be accessed only when execution is within their scope. Thus access to the local variables must cluster between the call and return of a procedure.

The body of a loop. Since it is repeatedly executed, the code body of a loop gives a highly correlated set of accesses.

The run storage stack. If a stack is used to allocate storage, typically only a few parts of it are actually accessible at any given moment. In particular the global and local storage are most active.

The first step for the compiler writer is the identification, based on the language and use to which it will be put, of the logical segments of program structure. The second step is to devise a scheme to get those logical units that themselves have correlated accesses on the same set of pages. It is this second step for which we suggest an approach.

We have already mentioned that the program can be relocated as a whole to achieve protection. We propose that each logical segment be separately relocatable and that the virtual addresses be associated with the code at first access to the logical segment. The effect will be to build the virtual space from the bottom in the order in which the logical segments are first accessed. Our central assumption is that this experimentally discovered correlation is a sufficiently good approximation to the optimum that the fraction f_i can remain small during an entire quantum of execution time.

Implementation.

No special provisions have been made in contemporary hardware to help in the implementation of this scheme. We propose that the compiler allocate storage for each logical segment starting on a page boundary and with a virtual address disjoint from all other segments in this program as well as the final virtual space to be assigned at program initiation time. The page selector then serves as a unique name for the logical segment. When an undefined value of R corresponds to a previously unloaded segment (detectable by the disjoint address) we first must discover if the segment has been loaded but this instruction has not yet been changed to reflect the loading. If the segment has not been loaded, it is added to the virtual space and its actual location recorded in a table. If it has been loaded, the table is consulted and the obvious change is made to the referencing instruction and execution allowed to proceed. Other undefined values of R are handled as normal page faults.

Conclusions.

The proposed scheme will cause R to be undefined at least once for every memory referencing instruction in a program. The overhead in fixing the addresses need not be excessive (and in fact could be done in hardware). Nothing said here should deter the compiler designer from using more static schemes where he "knows" something (such as how his run stack interacts with paging) or even more dynamic schemes where he cannot "know" enough (such as PL/I ALLOCATE where a hole can be left by the corresponding FREE). The scheme does not make sense for programs that need less than a whole page of memory. Overhead can, in some circumstances, be avoided for programs that are run many times by saving the loaded form of the program.

In this proposal, as in all that depend on statistical effects, the gathering of experimental distributions is essential. A compiler can discover the distribution of logical segment lengths (some preliminary data gathered on the B5500 gave a median length of 60 words for Algol programs). The time-sharing monitor can tabulate f_i and ΣV_i due to various compilation methods. To be generally useful, such experiments must be reported in sufficient detail to allow their repetition where compiler, machines or language are changed.

The main effect of the proposal is to attempt to approach the efficiency of memory utilization of the variable length segment machines (Burroughs B5500, B6500, B8500) while retaining much of the simplicity of the paging organization.