An algorithm for dynamic storage allocation of variable-sized programs and records is described. The algorithm is designed for real-time systems in which core is assigned for data and programs in a completely unscheduled manner as, for example, in reservation systems.

The objective is to make efficient re-use of available core with minimal movement of data or programs after entry.

The procedure given depends on the frequency distributions of program usage and of data block sizes. However, the distributions need not be specified since the system will adapt to these distributions and, equally important, to any changes in them that may occur.

The algorithm has yet to be simulated or tested within an operating system.

Dynamic storage allocation for a real-time system

by B. I. Witt

nature of the problem

The environment for which the algorithm was devised is typified by the real-time airline reservation problem. A large number of agent-operated terminals are assumed, each connected to a central computer. Requests from the terminals (for information, reservations, etc.) arrive at the computer at random intervals. Each request must be processed by use of several programs retrieved from a very large library of programs and with several references to massive files containing flight information, billing accounts, etc. Since neither all programs nor all data files can be contained within core, most of the response time is spent in retrieving information from an external backup store. Retrieval time for one request is overlapped with CPU processing of other requests. Typically, there may be some 30 requests in various stages of processing at any one time. Lifetime within the system of a typical message is in the order of several seconds.

The need for dynamic storage allocation in this environment arises from the random nature of the requests from the terminals. Programs (especially if compiler generated) and data come in a variety of sizes and must be entered in core only when they are needed. In the process, older programs and data blocks are overwritten. After several minutes or hours of operation, unless care is taken, all of core may be fragmented, with many free core locations, but not enough contiguous ones for the current need.

However, data and programs do have certain "natural" char-

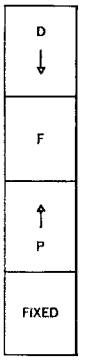
acteristics which can be used to advantage by the algorithm. Data block sizes will be determined by certain characteristics of the file organization and of the storage device used. Thus, block sizes can be expected to be one of several discrete sizes. Also, for periods of time (measured in minutes) the number of required blocks may be expected to be somewhat stable. Furthermore, retrieved data has a very short life of usefulness in core (several seconds). Programs, on the other hand, will tend to be of random sizes, but during relatively long periods of time (several hours) their pattern of usage will be somewhat fixed. That is, certain programs can be identified as being used fairly frequently. Although these characteristics are not mandatory the degree to which they are present decreases the CPU time necessary to execute the algorithm.

The objective of the algorithm is to make efficient re-use of available core without frequent moving of data or programs once they have been entered.

General organization of memory

Memory organization is depicted in Figure 1. Low-order memory[1] contains certain privileged programs and files which permanently reside in core (e.g., the monitor, program directory, etc.). This sector does not enter into the dynamic storage allocation scheme and will not be mentioned again. The next area, called the *P* sector, houses all other programs. Programs are called from external storage as they are needed and stored in contiguous locations in core starting from the fixed lower boundary of the *P* sector. The high-order sector, called the *D* sector, houses all data blocks. Data records (or requested work-storage blocks) are also placed in core in contiguous locations as they are needed, but starting from high-order memory. The free area between the *P* and *D* sectors, which has variable boundaries, is called the *F* sector. The general scheme will be to attempt to make efficient re-use of space within *P* and *D*, and to encroach on *F* only when needed. (If *F* becomes critically small, the system is considered overloaded, and new requests from the terminal are either refused or logged for later processing.)

Figure 1

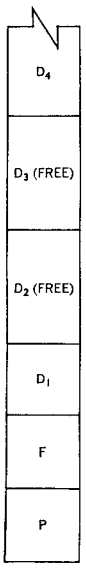


Procedure under normal load

management of the D sector

In addition to the basic area for data, each data block cut from the *F* sector contains control information which links it to other data blocks of corresponding size. Thus, if there are 15 distinct block sizes, then 15 separate lists, or chains[2] of data blocks, would be maintained. Each block points to the block of equal size next lower in memory. Whenever a data block is released, with one exception noted below, the block is flagged. Whenever a new data block is needed, the chain of blocks of corresponding size is referred to, and the flagged block first encountered (i.e., highest in

Figure 2



memory) is selected for re-use. Only if no flagged blocks exist is a new block cut from the F sector. With this rule, freed sectors in upper memory tend to be quickly re-used, those in lower memory, less so. The exception mentioned above is that: whenever a block is freed which touches the F sector (as then defined), the block is incorporated into the F sector instead of being flagged, and furthermore, all contiguous flagged blocks immediately above such a block are incorporated in the F sector.

Thus, in Figure 2, D_2 and D_3 remain free because the most recent assignments for blocks of corresponding sizes were made from blocks higher in memory within D. If D_1 now becomes free, D_1, D_2, and D_3 are all incorporated in the F sector.

Soon thereafter, some short-term stability of data block assignments may be expected. The distribution of data block sizes actually created will correspond to the distribution actually needed. Data blocks of a certain size will be freed at approximately the same rate as needed. Many short-term aberrations in the distribution will be handled with no additional overhead since the D–F border may recede in the direction of D. It is this expected stability that makes the algorithm economical.

However, because of changes in the external environment (for example, the end of a work shift), the distribution may undergo severe changes. Initially the system reacts by cutting into the F sector. That is, more blocks of the new size now needed are created. However, many blocks of a size no longer "popular" may be released (i.e., flagged in their chains) and, in a sense, be trapped behind the lines. That is, since the new-type data blocks are in active use near the F border, they prevent the old-type blocks from being assimilated into F. Thus, there may be large unused (and unusable) patches of core within the D sector.

Clearly, if the F sector is small, which may indicate the system is under a heavy load, this is a matter of concern. However, a period during which the system is under a heavy load may be the worst time to take additional CPU time to correct the situation—since corrective action (called *trimming*) would slow down response-time and contribute to an even heavier load. In order to minimize corrective action taken during a heavy load, the criterion adopted for trimming is made dependent on the relative sizes of the unused patches and the F sector. Specifically, the criterion is to trim the D sector whenever the total area of the free blocks contained therein is greater than the area of the F sector. Thus it is sufficient to test for a trim whenever the F sector is reduced in size by an encroachment from either P or D; and whenever an interior data block is released. Note then that trimming procedure is preventive rather than corrective.

Next, the question arises as to the specific form of the trimming action. In other circumstances it might have been to move all used blocks to contiguous positions in upper memory, with all the resulting problems for the programs operating on the data. For this application, however, with each individual data block

having a relatively short life, it seems best to allow all old data blocks to "die on the vine," a procedure described next.

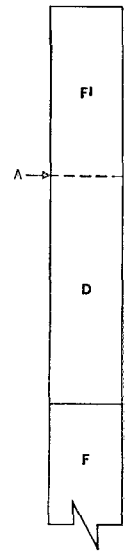
trimming the D sector

The procedure for trimming the D sector is as follows:

Step 1. Set a pointer A (see Figure 3) marking the higher boundary of the highest occupied data block in the D sector. Call the area above A (which may be null) the F' sector. Remove all blocks in F' from the D sector chains (D chains, for short).

Step 2. Move the pointer down and continue to modify the D chains whenever the highest data block within the D sector becomes free.

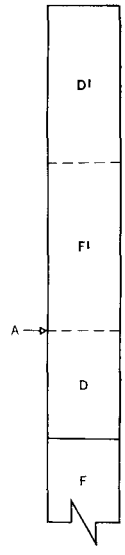
Figure 3



Step 3. When a new data block of any size is needed, proceed as follows. If the data block can be contained in F', place it in the highest portion of F'. Call the assigned area D'. (See Figure 4.) Maintain a new set of chains for D', following precisely the same rules for D' and F' as was previously followed for D and F. For example, data blocks of D' which border F' are released to F' when they are freed. Otherwise, they are flagged in a set of chains, one for each unique block size (D' chains). As subsequent requirements for data blocks arise, the rule described herein (Step 3) is modified to read: Search the appropriate D' chain, and assign the first available block (that is, the upper-most block). If there are no empty (flagged) blocks on the chain, but if the new block can be contained in F', place it in the highest portion of F' and update the appropriate D' chain.

Step 4. If the assignment cannot be made by the procedure in Step 3 (i.e., if there are no empty blocks in the appropriate D' chain and if F' is not large enough), then refer to the original D chains. However, in order to foster the movement downward, the chains are now referred to in *reverse* order. That is, the assignment is made from the lowest available block in D, rather than the highest. (Hence, a two-way linkage is required in the chains.)

Figure 4



Step 5. If the appropriate D chain indicates that there are no available blocks in the D sector, then a new block is cut out of the F sector and added to the D chain, just as in the normal procedure.

The presumed effect of this procedure is that the A pointer will continue to move downward. The further it moves, the larger D' and F' become. The larger they become, the less often reference will need to be made to either the D or the F sector, and this in turn will facilitate the downward movement of A. In short, it is expected that this procedure will soon result in the complete elimination of the D sector. Blocks on its lower border will be absorbed by F, blocks on its upper border will be absorbed by F'. When the last block of D is released, that is, when the F and F' sectors touch, all special controls are removed and the system proceeds to handle the F' and D' sectors as if they were normal F and D sectors. Since only current blocks are contained in D',

the desired redistribution has taken place with no movement of data blocks and no interruption of service.

management of the P sector

The general procedure for programs is the same as for data: programs are initially placed in contiguous positions (starting from lower memory), and by some criteria may eventually be declared to be *releasable*. To satisfy new requirements, releasable programs are examined from lower to upper core. The first releasable program encountered which is equal or larger in size than the new program is over-written. Only if no such releasable program is found is a new block of the required size cut out of the F sector. Releasable programs which border the F sector are incorporated into F. And finally, when the combined area of all releasable programs and the scraps (i.e., the unused portions of reassigned program blocks) exceeds the area of the F sector, the P sector is trimmed. (Trimming the P sector means actually moving or reloading all non-releasable programs into contiguous locations in lower core, while maintaining their current sequence.)

Despite the apparent similarity of the procedure to be used with programs and data, there are some fundamental differences, all revolving about the criteria used to declare programs releasable.

Programs, unlike data, never become completely useless. It is not known when they will be needed next. If they are already in core when next needed, the lifetime of a message in the system will be shorter. But since it is assumed that there is not enough room to store all programs, some compromise is in order. The compromise in general is this: programs which are in use will always be allowed to remain in core.[3] In addition, if the system is stable, certain unused programs will be allowed to remain in core. But as the system load increases, or if a phase change is entered, we will place an increasingly severe age criterion on the unused programs in order to permit them to remain in core. Programs failing to meet the criterion will be declared *releasable*.

The method chosen to detect increases in system load or a change in the pattern of requests being presented (i.e., a phase change) is simply to keep track of the size of the F sector. When F contracts by 10%, an *inspection* is made for programs which have not been used for a certain length of time. As a result, if the system is undergoing rapid change, inspections will follow one another at a rapid rate; if the system is undergoing a very slow change, inspections will be infrequent. Furthermore, since the length of inactivity allowed a program is based on how fast the system is undergoing change, the inspections themselves can be used—rather than, say, a clock—to determine if a program has been unused for a sufficiently long time. Specifically, a program is declared releasable if it remains unused for two successive inspections.

Before proceeding with the step-by-step procedure of P sector inspections and trims, one final general note is in order regarding P sector trims. Because the sequence of programs is maintained during the relocating or reloading or programs during a trim, the

programs tend to arrange themselves after several trims in a sequence approximating their relative activity (i.e., their probability of use). A program that remains non-releasable will never be moved upward, always downward. Hence, after several trims, if the probability-of-use distribution of the programs remains approximately the same, the P sector stabilizes. All programs with a short useful life-time appear near the F sector, where they are either absorbed by F or where their area is used by other programs of equally short life. In short, the P sector will have automatically organized itself into something resembling two distinct areas, one containing in compact form the "permanent" programs, the other containing in a somewhat loose-knit manner the "temporary" programs. However, the algorithm permits the continual change of the content in each area as change is dictated by external events. Furthermore, it may be expected that although many trims will be needed at the beginning of every phase, fewer and fewer will be required later; and with each progressive trim, fewer and fewer programs need to be moved, for the most active programs are already compacted.[4]

inspecting and trimming the P sector

The following procedure is used for P sector inspection and trimming:

Step 1. A base value F_0 is established against which changes in the size of the F sector are measured. F_0 is always the size of the entire F sector as recorded at certain specified points of time. Initially it is all of available core since P and D are presumably empty. Redefinitions of F_0 occur at the following times:

- At the completion of any inspection (which should not be confused with the 10% test which leads to an inspection).
- At the completion of a P sector trim, which necessarily results in a larger F sector than the one determined at the completion of the pre-requisite inspection.
- At the release of a data block bordering the F sector, provided that the size of the expanded F sector exceeds the current value of F_0.

Note that between inspections F_0 can only be made larger, and hence the system is permitted to respond more quickly to a resumption of activities after a lull. During D sector trims, the size of the F sector is taken to be the sum of the F and F' sectors.

Step 2. Whenever the F sector is impinged upon by either the P or D sectors, the new area of F is compared against the current value of F_0. If it is 10% less than F_0, a new inspection is made as described in *Step 3.*

Step 3. It is assumed that either all program blocks currently assigned, or all corresponding entries in the program directory, are chained in descending core sequence. Thus, the first link in the chain is the program bordering the F sector. At each inspection, this chain is examined. Every unused program not already flagged, is flagged. Every unused program which has a flag is declared

releasable, and with one exception, will be placed in a new chain, with link addresses pointing to the previous program encountered (i.e., a chain of releasable programs is formed which is in ascending core sequence). The one exception is that releasable programs which border the F sector are not placed in the chain, but instead are incorporated into the F sector. A cumulative sum is made of all releasable programs so defined. In addition, the new area of the F sector (i.e., F_0) is calculated.

Step 4. The sum of all releasable areas and all scraps is compared against the area of the F sector. If the former is less than the latter, the chain of releasable programs becomes the tool for locating the lowest releasable space large enough to accommodate a new program. If the combination of releasable and scrap area is greater than the F sector area, the P sector is trimmed, and the chain of releasable programs is initialized (i.e., made empty).

Procedure under heavy load

It was said earlier that trimming was performed as a preventive action in order to avoid overloading the machine when it was already under a heavy load. It follows that trimming and related procedures should actually be suppressed or altered during periods when the machine is under heavy load.

measuring the load

First, some simple but unambiguous means is needed for detecting that the system is entering a period of heavy load. A suitable measure appears to be the total amount of core occupied by in-use programs and data. An equivalent and more convenient measure is the total free core in the system, i.e., the sum of the areas of the F sector, the unused data blocks, releasable programs and program scraps. These areas are, of course, the very same areas used for trimming tests. The evaluation of the load is merely a comparison of the amount of free core (as defined above) with some pre-defined number s to serve as a "stress" criterion. If the amount of free core is less than s, we shall say that the system is in a state of *stress*, and by implication, in danger of overloading. The actual value of s will depend on the problem application and, of course, the total size of core. However, it seems easy enough to keep track of the free core over a period of time, and to eventually establish s as an amount of free core which is available, say, 85% of the time. A lower limit for s should represent enough core to satisfy the clean-up operations discussed later. The initial value for s may be guessed. To prevent rapid oscillation between the stress and normal states, we might demand that to return to normal, the free core available must not only be greater than s, but significantly greater. But for present purposes this damping scheme will be ignored.

procedure under stress

Whenever a new data or program block assignment is made, whether it be a re-use of an older block or an encroachment on the F sector, the new free core amount is compared against s. If it is less than s, the following actions are taken:

Step 1. A special inspection of the P sector is made, without regard to the 10% change normally required of the F sector. Now *all* unused programs are added to a chain of "releasables," without regard to how long they have been unused (i.e., without regard to the special flags). The area of all releasable programs and all scraps are summed, as before.

Step 2. As before, if the calculated area is greater than the area of the F sector, a trim is made; otherwise the new chain of releasables is used for future program assignments.

Step 3. As soon as a program becomes unused, it is added to the chain of releasables. There is no age requirement in periods of stress.

Step 4. No further inspections or trims are made of P or D until the system leaves the state of stress.

Note that during a normal period (i.e., non-stress) the D sector is tested for trim whenever anything happens which may affect its trim status. Hence it enters stress already in trim.[5] After stress is detected a special test is made of the P sector to insure that it also is in trim. Subsequently, of course, the system may become further overloaded. However, if all goes well, the system eventually returns to a normal mode of operation. The fourth condition above is nothing more than an effort to reduce the housekeeping overload.

overload

No storage allocation algorithm can guarantee that demands for core can always be satisfied and the question arises as to appropriate action if such demands cannot be satisfied. (It is not enough to say that the installation should have had a larger memory.) Although the answer to this question is beyond the intended scope of this paper, we have some brief comments.

The first defense against inundation is to stop the message flow. As soon as the system enters stress, the polling of communications lines for new messages can be discontinued, or all new messages can be logged on backup store. Thus, all core requests represent requirements for the processing of messages already in the system. If the stress criterion was properly chosen, there should be sufficient free core remaining in the system to meet such demands.

If this proves not to be the case, other procedures which may be considered include:

- Granting core to only high priority messages, even if it means that CPU time will not be efficiently used.
- Utilizing for high priority message processing the space occupied by programs supporting only low priority tasks—even though those programs may be in use.
- Writing out on backup store all low priority messages and data blocks, and using that space for high priority message processing.

Summary

The storage allocation algorithm considered brings programs and data into core as they are needed, directing the former to contiguous positions in lower core and the latter into contiguous positions in upper core.

All data blocks so defined are placed into separate chains according to their size. When a new requirement is placed for a data block of a certain size, the corresponding data chain is searched. If a previously assigned block on the chain has been freed, that block is assigned for the current requirement. If no previously-assigned blocks of the required size are available, a new block is generated, contiguous to the data block lowest in core. As long as the distribution of data block sizes remains relatively stable, the system operates with little overhead. As the distribution of data block sizes changes, the system adapts itself to the change. No moving of data blocks is anticipated.

Programs are generally allowed to remain in core, even though they are no longer in active use. However, when the system requires it, programs are released, according to the amount of time in which they have been inactive. Programs are over-written only by new programs of equal or smaller size. Occasionally, programs in use must be moved, but with each such move programs which are heavily used tend to compact themselves and, hence, tend to resist additional moves.

The algorithm uses the inherent frequency distribution of programs and data block sizes to reduce overhead costs without requiring any prior knowledge of the distributions.

Since the algorithm has not been tested within an operational system, its value is subject to some speculation. Presumably, its behavior within a system could be simulated with modest effort. However, the accuracy of results obtained would depend on the accuracy of the frequency distributions of program usage and data block sizes (together with pattern of change in these distributions) assumed to describe the system considered.

Since information of this type tends to be incomplete, simulation of the algorithm would probably tend to give indicative rather than definitive results.

ACKNOWLEDGEMENT

The author acknowledges the valuable assistance of his colleagues —T. Kallner, Lucille C. Lee and Janette T. Wood.

FOOTNOTES

1. Throughout this paper, memory with smaller core addresses is referred to as *low-order* or *lower* memory; memory with larger core addresses is referred to as *high-order* or *upper* memory.
2. *Chain* is used in the conventional sense with the following implications:
   - The actual location of each of the data blocks is immaterial.
   - There exists for each chain a set of control words in a fixed location which contains the location of, say, the first word of one of the blocks. Hence, the location of the usable portion of the block may be inferred.

- The first word of the block referred to contains the location of the first word of another block. The first word of that block refers to the first word of yet another block, and so on. Each of the blocks is thus a *link* in the chain.

3. The term *in use* need not be defined rigorously for present purposes. Suffice it to say that many programs may be in use simultaneously in this real-time system. Also, any one program may be simultaneously in use in servicing the requests from a number of different terminals. Furthermore, a program operating upon a particular set of data may call a subprogram with the expectation that return will be made to the program. Under certain circumstances such a program might be declared in use, and other circumstances not in use.
4. Although re-locating or re-loading a program which is still in use is treated in a cavalier manner in the body of the paper, the problem is by no means trivial. Two alternatives present themselves. The first is that the programmer specifies particular relocation points within his program at which the program can be reloaded. This implies that he makes no modification of a program across a re-location point, that he stores no absolute addresses for use in the program segment following the relocation point, etc. With this alternative, program re-location takes place only when all programs have reached relocation points and have relinquished control to the monitor. Programs so written have, by definition, been called *disciplined* programs. The second alternative requires that programs be so written that they are re-locatable at any point. The implication is that the monitor must be informed of the way in which index registers are being used (and hence be able to modify them if the program is re-located), the program cannot store any data relative to its present location except perhaps through the monitor, etc. Programs so written have, by definition, been called *well-disciplined.*
5. Actually, as described, D is in trim only if the block assignment which led to stress was from an empty block previously formed. If it was a new encroachment on F, D may be knocked out of trim at the same time that the system is put in stress. Furthermore, that same encroachment may be enough to satisfy the 10% comparison which leads to P sector inspections. Here then is a problem of priorities. Any encroachment on F as described in the paper calls for three things: a test for D sector trim, a test for a P sector inspection, and a test for stress. But the sequence of the tests will affect the resultant structure of core. For example, if D were tested and trimmed, the larger resulting size of F might indicate that P need not be inspected. If, however, P had been looked at first, it might have resulted in P being trimmed and, with a larger F, no need to trim D. Similarly if P were trimmed before the stress test, newly declared releasable programs might leave the system in a normal state, whereas if the sequence were reversed, the system might have been declared in stress. There is a set of trade-offs here that is probably not too significant but which will nevertheless need to be examined more carefully.

BIBLIOGRAPHY

- "Papers presented at the ACM Storage Allocation Symposium, June 23-24, 1963", *Communications of the ACM*, **4**, no. 10, 416-464, October 1961.
- John W. Weil, "A Heuristic for Page Turning in a Multiprogrammed Computer," *Communications of the ACM*, **5**, no. 9, 480–481, September 1962.