To: Distribution

From: Steve Webber

Subject: A New Area Management Mechanism

Date: 8/8/75

Introduction

There have been many complaints about the current area management mechanism over the past years. This MTB proposes replacing the current mechanism with another and gives reasons why I think it would be wise to do so.

Problems with the Current Area Mechanism

The current mechanism has the following flaws:

1. It uses the "buddy system". This has many disadvantages, particularly in a virtual memory systems such as Multics has. The most important problems with the buddy system with respect to the way we use areas are:

   A. The area, once initialized, is difficult, if not impossible, to extend. Hence, an area large enough to handle all reasonable cases must be created. With the buddy system, where the entire area is initialized into regions of increasing powers of two in size, several pages in a large area are touched (modified) during area initialization. This causes unnecessary paging (although it occurs only once per area invocation).

   B. Since areas must be initialized to have a sufficiently large size, and since each Multics process needs a (potentially) large, general purpose area to work with (system_free_N_), each ring in each Multics process is given an area of size 64K. This means that each such area requires a 64K AST entry while the segment containing it is active and being used. In a typical Multics ring, an area of less than 4K is usually sufficient, and hence, we could have gotten away with a 4K AST entry. The buddy system with its required initialization, forces us to use too many large AST entries, a valuable system resource.

C. The buddy system works by allocating the smallest power of two region of storage that will contain the requested amount of storage. This means that if a block of 129 words were needed, a region of 256 words would be returned. This causes a great deal of waste and makes very poor use of the area by utilizing only a relatively small fraction of the available storage.

D. The allocated blocks of storage in a buddy-system-formatted area are not necessarily contiguous. This means that if a particular use of the area is to perform many "allocates" without intervening "frees" that many more pages will be required, first because of large breakage losses in wasted, unused allocated space, and also because the blocks that are allocated are not necessarily contiguous. This also leads to increased paging.

E. One particular use of areas in the system, the backup system, depends on contiguous (except for overhead data) allocation of allocated blocks if no intervening frees be performed. This requires that two area management mechanism be supported by the system.

2. The current mechanism, again because of buddy-system characteristics, is unacceptible for use in directory control and segment control. If a more reasonable area mechanism were available, we could replace the current fs_alloc area mechanism with the system standard, thereby making the system that much more consistent.

3. The current area format, for the reasons mentioned above, would be unacceptible for a new proposal which would change all combined linkage regions and combined linkage segments into areas. This has advantages in that storage in combined linkage regions could then be reclaimed.

The Proposed Scheme

There have been many studies of a theoretical nature (see Knuth Vol. I) trying to determine what is the "best" general purpose allocation/freeing strategy to use. There are, of course, examples that can be fabricated to show that any one strategy is better than another, but the studies show that a "first fit" strategy (one where the first free block found that is large enough is used) with its simplicity in code and minimal overhead data, is quite acceptible for a large class of uses. A common refinement that costs little in execution time and paging, is to partition all free blocks into classes according to size -- merely to minimize the search time for a free block of sufficient size. This is the basic mechanism being proposed.

The new area mechanism would work as follows:

AREA FORMAT

The new area format is defined by the following PL/I declaration:

```
dcl 1 area aligned,
    2 version fixed bin,
    2 last_usable bit (18) aligned,
    2 last_used bit (18) aligned,
    2 flags,
      3 extend bit (1) unal,
      3 system bit (1) unal,
      3 mbz bit (34) unal,
    2 next_area ptr unal,
    2 prev_area ptr unal,
    2 class (17),
      3 fp bit (18) unal,
      3 bp bit (18) unal,
    2 first_trailer bit (36),
    2 storage (M) fixed bin (35);
```

where:

version	is the version number of this area format. It is 0 for current areas being used today, it will be 1 for the newly proposed areas, and it will be > 22 for the old style areas of today.
last_usable	is a relative pointer to the last usable word in the area. To extend the area, all that need be done is to change this value (if nothing exists beyond the area in the segment in which the area resides).
last_used	is a relative pointer to the last word of the area actually allocated. If there is no storage available on the various free lists (area.class (i)) the storage starting at last_used+1 is taken.
flags.extend	is ON if this area can be extended. This flag would be ON for the combined linkage regions managed by the system. The only purpose of the flag is to determine if another area should be defined (by creating another area segment) when an allocation request fails for lack of room in the current area.
flags.system	controls which procedure to call when a new area segment is to be created (because the "extend" bit was ON and not enough room

existed in the current area). If this bit is ON, the procedure hcs_$create_area_segment is called. If this bit is OFF the procedure create_area_segment_ is called. A user, by initiating his own vwersion, can control the name, location, etc. of his area segments.

next_area — is a pointer to the next area in a list of areas. This item is only meaningful if flags.extend is ON.

prev_area — is a pointer to the previous area in a list of areas. This item is only meaningful if flags.extend is ON.

class.fp(i) — is a relative pointer to the first free block whose size is within the range 2**i to 2**(i+1)-1. It therefore defines a list of free blocks which should be searched first when a block of the specified size is wanted. If a block can not be found in this list, the next larger list is searched until all lists are exhausted. When no list contains a free block large enough, virgin storage at the end of all allocated blocks is used and last_used is updated.

class.bp(i) — is used with class.fp(i) to form a circular list to make it easy to thread free blocks into and out of the lists.

first_trailer — is a dummy trailer used to allow the allocation mechanism to assume everything is correctly initialized. (No special casing is needed during normal operation.)

storage — is the actual storage which is usable.

The area consists of 3 types of storage: 1. blocks in use (busy blocks), 2. free blocks, and 3. virgin storage (after blocks of types 1 and 2). Free blocks have a forward/backward thread in the first word of the block. All blocks have a trailer, which is part of the block, that contains the size of the block as well as the size of the following block. The trailers also point back to the area header. The trailers are one or two words long.

ALLOCATION

The allocation strategy is quite simple. Each of the free classes (which contain blocks large enough) are searched, in order, until a large enough block is found. This block is then split into two parts. The first part is returned to the caller; the second is added to the appropriate free list. If no free blocks are

available in any of the free lists, virgin storage is used. If no virgin storage is available in the current area and the "extend" bit is ON for the area, a new area is created and threaded to the current area. The allocation request is then serviced from the new area. Allocation requests can still be performed in the previous area, if there is enough storage.

Note that PL/I offset data does not work with this form of multicomponent area.

FREEING

The freeing of a block is even simpler. A check is made to see if the adjacent blocks are free, and if either or both are, the free blocks are merged into a single block. This block is then threaded into the appropriate free list unless it is the last block before the virgin storage area in which case it is added to the virgin storage area and the header item "last_used" is updated.

System Areas

One extention possible with the new area mechanism and the conversion of combined linkage regions into areas is the ability to automatically create a new area implicitly, when needed. This means that any allocation request in an extensible area can be satisfied as long as the requested block will fit in (a little less than) a 256K segment and there is sufficient quota remaining in the appropriate directory. If this were done for combined linkage regions, the allocation routine, when recognizing that the current area does not have the necessary free storage to satisfy a request, calls upon the system to create a new combined linkage segment, initializes it as an area, threads it to existing areas, and finally uses it to satisfy the current request.

An external variable, system_data_$system_free_ptr_, will be provided to return a pointer into the current system free area. By allocating into the area based on this pointer, a program has access to all of the areas managed by the system. The program get_system_free_area_ will continue to work as today and create system_free_N_. However, system code (and user code as well) should be modified not to use this mechanism (see the alternative proposal below). Note that get_system_free_area_itself is not being changed because this would be an incompatible change with respect to offset variables. Offset variables can not be used in the system-managed areas because the "area" may span several segments and offset data does not include a segment number field. (We would probably want to do some renaming such as adding the name get_user_free_area_to get_system_free_area_ and creating user_free_N_ instead of system_free_N_..)

Even though we propose to retain system_free_N_ for those programs that use it today, it will be more efficient both because it will not require a large AST entry until it is actually needed and also because the area management algorithms are better.

The linker would be changed to allocate (via the PL/I allocate statement) linkage sections and storage assigned via hcs_$assign_linkage. This storage can later be reclaimed if a procedure is made unknown or a program knows that it no longer needs storage it received via hcs_$assign_linkage. With the coming use of seperate static, the static sections will likewise be freed when the associated procedure is made unknown.

Compatibility Issues

There should be no compatibility problems with converting over to the new area mechanism. We have had practice at doing this before. The one unfortunate issue is the problem of existent permanent areas which users surely have. These will continue to be supported indefinitely (via retention of the current area management modules). Hopefully, the day will arrive when this support can be dropped.

PL/I Areas

PL/I would continue to use (nonextensible) areas when a program declares an area. However, it would be more efficient to change the action that PL/I takes when an allocate statement is executed which has no "in" clause. The language spec states that the allocation is done somewhere decided by the operating system, but not necessarily constrained in the same way that allocation must be done in a PL/I area. That is, the allocated data need not be in a PL/I area. This means that we are not bound to allocate such storage in a contiguous region that obeys certain rules; we must merely allocate the storage in some way that it can later be freed. The proposal is to make such allocations in the system managed areas -- the combined linkage regions. The system managed allocation area, consisting of several PL/I areas, can not be used when a PL/I area is called for. However, they work fine for other allocations.

So, it is proposed that Pl/I be changed to allocate such storage in the system-managed areas. This means special case operators, but also means that such allocations are quite efficient. The user is not required to get an area pointer and use a based area. Rather, the operator will make use of the external variable, system_area_ptr_$system_area_ptr_, mentioned earlier.

The system should be converted, as time permits, to use the new scheme. Eventually all references to get_system_free_area_ will

be removed from the system.

Type 6 Links Revisited

One of the mistakes we have made in recent years with the Multics operating system is the concept of type 6 (create-if-not-found) links. The main reason for introducing these into the system was for efficiency reasons in the implementation of PL/I external static variables and FORTRAN common. Today we see how this can be done even more efficiently and without many of the (quite unexpected to the casual user) side effects of type 6 links. The proposal here is to manage such variables without any use of reference names (e.g. stat_ or whatever it may have been renamed to). Instead, a special per-ring data base, managed by the linker (which will also eventually be per-ring), will keep track of external, named variables. This data base will map the names with pointers to the storage for the variable, but not be required to be structured as the definition sections of standard object segments. This means that the resolution of links to such variables can be potentially much faster, particularly if there are many such variables known in a process (ring). It also means that the concept of ".link" segments can be removed from the system as this is the last remaining place where they are used.

The actual storage used by these external variables will be allocated in the system-managed areas instead of a temporary segment initiated with the reference name stat_. This, too, saves another segment (ASTE) in the process directory (per-ring). This also removes the restriction that there can be only 256K worth of storage for external variables. (The restriction that any one variable be only 256K (about) in size or smaller still holds.)

The proposed manner of implementing this change is to define a new link type which is basically of the form:

<*system>|[ext]+exp,m

The compilers will no longer use type 6 links.