Importance of Memory Allocation and Disk Representation in Specific Based Regression

Manoj Kumar[1] Dr. Kalyankar N.V.[2]

[1]Research Scholar, CMJ University, Shillong, Meghalaya

[2]Research Scholar, CMJ University, Shillong, Meghalaya

---♦------------------------------------

Memory allocation is required in all phases of design. It typically consists of only two operations: the allocation of a specified amount of unused memory and the freeing of an allocated block. Some systems reduce this to a single operation that allocates on demand and frees automatically when the object is no longer used. Special **garbage-collection** operations find the unused memory by counting the number of references to each block [Deutsch and Bobrow].

Not only are objects and their attributes allocated, but temporary search lists and many other variable-length fields are carved out of free memory. It is important to deal with the memory properly to make the system run efficiently. Nevertheless, many design systems foolishly ignore the problem and trust the operating system or language environment to provide this service correctly.

One basic fact about memory allocation is that it takes time to do. Most programming environments perform memory allocation by invoking the operating system and some even end up doing disk I/O to complete the request. This means that a continuous stream of allocations and frees will slow the design system considerably. If, for example, memory is allocated to store a pointer to a marked object and then freed when the marker list is no longer needed, then every selection step will make heavy use of a general-purpose memory-allocation system. This approach is not very efficient. A better technique is to retain these allocated blocks in an internal list rather than freeing them when the analysis is done. Requests for new objects will consult this internal free list before resorting to actual memory allocation. Initially, the internal free list is empty, but over time the size of the list will grow to be as large as the largest number of these objects that is ever used.

The reason for not freeing objects is that they can be reallocated much more quickly from an internal free list. The memory allocator in the operating system cannot organize freed blocks of memory by their object nature because it knows nothing about the design program. Therefore it will mix all freed blocks, search all of them when reallocation is done, and coalesce adjacent ones into larger blocks whether or not that is appropriate. All this wastes time. The design system's own free list is much more intelligent about keeping objects of a given class together and can respond appropriately. Thus the speed of the program is improved.

Another time-saving technique is to allocate multiple objects at a time, in larger blocks of memory. For example, when the internal list of free marker objects is empty, the next request for a new object should grab a block of memory that can hold 20 such objects. This memory is then broken down into 49 free objects on the internal list and one that is to be used for the current request. This results in fewer calls to the operating system's memory allocator, especially for objects that are used heavily. In addition, some space will be saved since the operating system typically uses a few words per allocated block of memory as thesiskeeping. The drawback of this block-allocation technique is that the program will typically have allocated memory that is unused, and this can be wasteful.

The biggest problem with memory allocation is that the program will run out of memory. When it does on a virtual-memory computer, the operating system will begin to page the design data onto disk. If the use of memory is too haphazard, the contents of an object will be fragmented throughout the virtual address space. Then the operating system will thrash as it needlessly swaps large amounts of memory to get small numbers of data. To solve this, it is advisable for design systems that run on virtual machines to pay special attention to the paging scheme. For example, all objects related to a particular cell should be kept near each other in memory. One way to achieve this is to allocate the objects at the same time. Unfortunately, as changes are made the design will still fragment. A better solution is to tag pages of memory with the cells that they contain. Memory used for a different cell will be allocated from a different memory page and all cell contents will stay together. By implementing virtual memory as "clusters" of swappable pages, the design activity will remain within a small number of dedicated pages at all times [Stamos].

In addition to aggregating a design by cells, clusters of virtual memory can be used for other special-purpose information such as the attributes of a particular design environment, analysis, or synthesis tool. The resulting organization will provide small working sets of memory pages that swap in when needed, causing the design system to run faster. For example, if all timing-related attributes are placed together, then those memory pages will remain swapped out until needed and then will swap in cleanly without affecting other data.

When the computer does not support a virtual address space, the design system must do its own swapping. This internal paging is typically done by having only one cell in memory at a time. The code becomes more complex because all references to other cells must explicitly request a context change to ensure that the referenced data are in memory. In virtual-memory systems, all data are assumed to be in memory and are implicitly made so. In an internally paged system, the data must be loaded explicitly. Many design operations span an entire hierarchy at once and will run more slowly if paged one cell at a time. For example, it will be harder to support hierarchical display, network tracing, and multiple windows onto different cells. Although the design system could implement its own form of virtual memory, that would add extra overhead to every database reference. It is better to have a virtual computer that implements this with special-purpose hardware. Also, letting the operating system do the disk I/O is always faster than doing it in the application program, because with the former there is less system overhead. Thus the best way to represent a design is to allocate it intelligently from a large, uniform memory arena.

DISK REPRESENTATION

If a design is going to be represented by placing it in virtual memory, then a very large design will consume much virtual memory. To modify such a design, the system will have to read the data from disk to memory, make changes to the memory, and then write the design back to disk. This is very time consuming when the design files are large and the virtual memory is actually on disk. Even nonvirtual-memory systems will waste much time manipulating large design files. What is needed for better speed is an intelligent format for disk representation.

By placing a "table of contents" at the beginning of the design file, the system can avoid the need to read the entire file. Individual parts can be directly accessed so the design can be read only as it is needed. This is fine for efficient perusal of a design but there will still have to be a total rewrite of the file when a change is made. If each cell is stored in a separate disk file, then the operating system's file manager will perform the table-of-contents function. This allows rapid change to single cells but opens up the problem of inconsistent disk files since they can be manipulated outside of the design system.

An ideal disk format would replicate exactly what is in memory. When this is done on a virtual-memory machine with flexible paging facilities, it is not necessary to read or write the entire design file at once. Instead, the file is "attached" to virtual memory by designating the disk blocks of the file to be the paging space for the design program. Initially these memory pages are all "swapped out" but, as design activity references new data, the appropriate disk blocks are paged in. Changes to the design manipulate the disk file in only those places corresponding to the modified memory. Allocation of more memory for a larger design automatically extends the size of the disk file. Of course, writing the design simply involves updating the changed pages. This scheme requires the smallest amount of disk I/O since it accesses only what it needs and does not require a separate paging area on disk.

The only problem with storing the precise contents of memory on disk is that object pointers do not adjust when written to disk and then read back into a different address in the computer. If relocation information is stored with the file, then each page must be scanned after reading and before writing to adjust the pointers. This requires extra time and space in addition to creating the problem of having to identify every pointer field in memory. Relocation could be done with special-purpose hardware that relocates and maintains pointer information in memory and on disk. Unfortunately, no such hardware exists. The only way to avoid these problems and still obtain a single representation for disk and memory is to stop using pointers and to use arrays and indices [Leinwand]. This makes I/O faster but slows down memory access during design.