Optimal dynamic use of memory for PL/1 object programs in a real memory environment

K. Dos and H. Otto

Systemprogramme, SIEMENS AG, Munich, Germany

Some basic considerations on dynamic memory handling are made and a method is described which will improve the efficiency of dynamic memory allocations of PL/1 compiled object programs on 360-type machines. Its main objective is the special treatment of storage items with proper runtime stack behaviour by a runtime stack memory retailing subsystem using multiple, not necessarily adjacent memory extents with an almost as high average allocation speed as when using a single extent. This is achieved by the preference of a 'current stack extent' which requires overhead only upon over- or underflow. The other storage items are handled by a non-runtime stack memory retailing subsystem with the main goal of keeping memory scattering as low as possible. A memory wholesaling subsystem interfaces the operating system and provides memory for both the retailing subsystems.

(Received March 1971)

1. Introduction

PL/1 compiled object programs can have a rather complex memory allocation structure:

There exist language facilities to prescribe explicitly—to a certain extent—in which way and especially in which interval at runtime—a variable must occupy memory space. It will occupy it permanently throughout the program if its storage-class is declared STATIC; its space will be allocated at block prologue and freed at block epilogue if its storage-class is AUTOMATIC, and its allocation state will be completely controlled by program statements if its storage-class is CONTROLLED or BASED. Moreover, there are further storage items not having an explicit counterpart in the source program text, which will for some interval of runtime require storage. Library routines require workspace, which unfortunately is needed in multiple generations as recursive use is possible via ON-units. Therefore, the simplest way would be to allocate storage dynamically on invocation of any routine including library routines and to free it on return. There are also items used by library routines which survive a single activation, e.g. file control blocks or buffers that are controlled by OPEN/CLOSE statements. In many cases, temporary workspace has to be allocated for storing intermediate results of a computation. All these storage-occupying items must be provided with memory space with the restriction that active items must not share parts of memory at any time, unless this is explicitly intended. A large part of this work can be done by the storage allocation routines of a compiler, but in a few more complicated cases this must be left to runtime routines.

The following discussion will mainly deal with that part of the problem that concerns runtime routines, especially from the viewpoint of runtime and memory efficiency. In Section 2, the problem will be presented by introducing some environmental restrictions and basic considerations and in Section 3 a solution will be described in detail.

2. Environmental restrictions and basic considerations

The design goals are largely dependent on the nature and restrictions of the environment, in which the object programs are running: In our case this will be a SIEMENS SYSTEM 4004 with non-pageable medium-size memory, with multiprogramming facilities including optionally shareable code. (The SIEMENS SYSTEM 4004, is as far as the internal structure is concerned, comparable with the IBM 360, the RCA SPECTRA 70 or ICL SYSTEM 4.)

In this environment, memory must be handled economically and excessive pending of any task must be avoided, because during the delay, memory is held inactive that could be used for some other task. Moreover, the system could become paralysed, if a loaded task were waiting for additional system-resources not actually available (e.g. main memory), which are held by some other task, that in turn waits for resources held by the first task. This deadlock can be prevented if waiting for system resources is prohibited. Nevertheless, a facility for dynamically providing additional memory will be available and useful both for increasing efficiency and for emergency cases. If, however, a task is not able to continue without further memory and this memory is not available, this task must leave the system (e.g. by checkpointing or abnormal termination).

Next, we will discuss the consequences of keeping parts or all of the used memory contiguous:

1. Keeping contiguous all of the used memory, i.e. using only one extent, results in most simple and time efficient memory housekeeping since the allocation state is completely described by a single boundary location between used and free memory.
2. On the other hand, using multiple extents yields most economic use of memory but requires more complicated housekeeping.

We must strive to get an optimal compromise between these conflicting points by allowing for use of multiple extents in principle while holding the housekeeping overhead as low as possible. One approach is to select groups of storage items at compile time to become contiguous allocation units at runtime. These units should not be larger than necessary for data access with a single base covering all items contained. For instance, simple data of a block can be assembled at compile time and addressed via a common base. One may also assemble at compile time the simple data of a procedure together with those of all its nested BEGIN-blocks, thus simplifying the runtime-access to global data within any nested BEGIN-block. Those assembled groups of simple data, be they for an entire procedure or for single BEGIN-blocks, will form independent allocation units as, e.g. single arrays will do. By this type of grouping it is possible to avoid too large contiguous units without increasing the data access overhead.

Another basic problem is that increased scattering of used and free extents is inevitable if freeing does not always occur in the reverse order of allocation. In fact, some features in PL/1 make such scattering possible. But fortunately a large part of PL/1 storage items is freed only in the reverse order of allocation. As this is the property of a 'last in first out' stack, this part which is based on the block activation structure is conveniently

Downloaded from https://academic.oup.com/comjnl/article/15/1/18/418338 by guest on 19 April 2024

handled in a so-called runtime stack. In order to preserve the contiguity of such runtime stack memory, it is very useful to separate it from the remaining memory structure as strictly as possible. Typical storage items not fitting into the runtime stack are CONTROLLED and BASED variables, I/O supporting items (as buffers or file control blocks) that are controlled by OPEN/CLOSE statements and results of value-returning procedures if the lengths of the results are not known in the calling block. But even if runtime stack management is treated separately, the runtime stack may have a multi-extent structure.

If multitasking is used, there is no longer any guarantee that the freeing of items allocated in a random order by several tasks will occur in reverse order. However, it may be expected that each task will itself have its own runtime stack behaviour pattern. It is therefore profitable to separate the runtime stacks of different tasks from each other to reduce memory scattering.

In addition there are of course other individual stacks, one belonging to each of the CONTROLLED variables, which should not be confused with the runtime stack described above.

3. A suggested solution of a memory management system

As mentioned, we expect a special treatment of runtime stack storage items to be most efficient. Let us subsequently refer to runtime stack memory as *type-1* memory while referring to the remainder as *type-2* memory. We need to introduce a few further definitions: An *extent* will be a contiguous area of memory. In particular, there will be frequent references to three types of extents: A *free* extent is one that is recorded to be available for allocations. A *used* extent is any extent previously allocated and not yet freed. A *stack* extent is one limited by a bottom- and a top-location with used space from the bottom up to an allocation level and free space from this allocation level up to the top.

The solution presented subsequently will have two levels of memory management: An upper level *wholesaling* subsystem that interfaces the operating system and provides memory for any of the two *retailing* subsystems on the lower level: The one for handling type-1 memory and the other for handling type-2 memory. We will first discuss these two retailing subsystems separately and later discuss the functions of the wholesaling subsystem.

The list structure used by the following memory management subsystems is shown in **Fig. 1.**

The type-1 memory retailing subsystem

We have to realise the following functions:

1. Allocation of contiguous memory for an allocation unit of specified size out of a free memory pool and returning its starting location.
2. Freeing any specified amount of not necessarily contiguous memory starting from the top of the runtime stack in reverse order of allocations. As more than one allocation unit may be freed on a single function call, this facilitates block epilogue actions as well as non-local GOTO-processing.

These functions would be simplest in the case of a single large stack extent as defined before. But this scheme would reduce flexibility of memory management because of its rigid demand for contiguous memory.

Both time, efficiency and flexibility however can be combined by the following method which is based on a *current stack extent* the state of which is maintained in a pointer triple consisting of bottom, top and allocation-level.

Beside this current stack extent, further stack extents not necessarily adjacent to the current one may exist. Most changes of the allocation state may be performed by simply updating the current allocation-level, thus obtaining a very quick allocation process on the average. Only if a change of the allocation state violates the current stack extent-bounds will additional actions become invoked: On allocation, the remaining free space within the current stack extent may not be

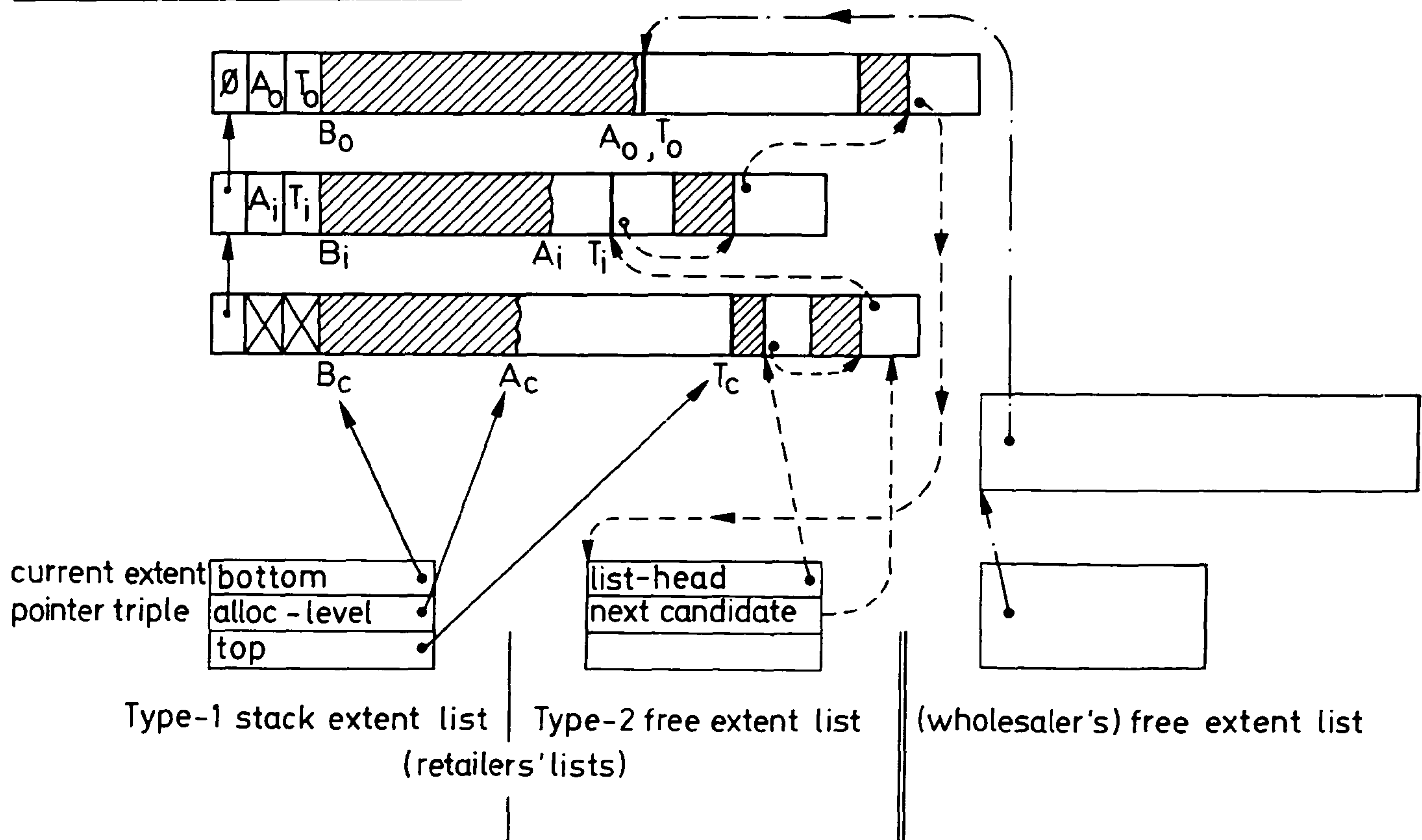

Fig. 1. The list structure used by the memory management system

Downloaded from https://academic.oup.com/comjnl/article/15/1/18/418338 by guest on 19 April 2024

large enough to satisfy the request resulting in an overflow. In this situation, another free extent must be found by the memory wholesaler, as will be described later, and it will be made the new current stack extent which contains a link back to the old current stack extent. Conversely, on freeing memory, a request to free more used space than is present in the current stack extent may occur. Now, this old current stack extent will be given back to the memory wholesaler and the previous stack extent to which the old current stack extent was linked is made the new current stack extent for additional freeing of used space within the runtime stack.

This type of runtime stack memory handling will on average need only a few machine instructions per function call. We consider this feature to be very important, since type-1 allocations occur very frequently at runtime and optimisation facilities are rather limited or lead to clumsy structures (e.g. when using rigidly distributed common workspace for library routines with nested calling structure).

The type-2 memory retailing subsystem

Here, scattering of memory to some extent is inevitable. One could indeed consider to ignore 'holes' of free memory generated on 'non-top'-freeing, but this could be wasteful in memory if there would be no compacting facility. Compacting as well as garbage collection, however, is only possible, if all pointers to active storage can be retrieved. This would cause much housekeeping overhead (e.g. when PL/1 list processing facilities are used). Thus, the concept should be discarded.

If free memory holes are to be used for allocations, the most natural way of recording free memory is to maintain a linked list of all the holes as free extents. There exist a number of methods using such lists of available memory which are more or less sophisticated. They are discussed in detail by Knuth (1968). An important consideration for any method chosen is that the memory structure formed should not be too scattered. Especially those methods are dangerous, which permanently try to allocate by searching the free extents in the same sequence. This leads to a free memory list structure with many small free extents near the head of the list. According to Knuth, a favourable statistical distribution of extents is achieved, when searching is always started from that free extent, which follows the one previously used for allocation, with turn around supposing the list to form a ring. In addition, holes too small should not be added to the free extent list.

The memory wholesaling subsystem

We still have left open the problem of how memory is obtained from the operating system by the wholesaling subsystem and in which manner it is supplied to both the different retailing subsystems described above. In Section 2 we mentioned the environmental restriction, that programs must be able to run with the memory resources obtained at load time. Although memory requests at runtime are possible, the program will not be allowed to wait for subsequent delivery if any request is rejected owing to actual shortage. Thus, the only admissible reasons for dynamically requesting additional memory are to increase efficiency (e.g. by getting workspace for additional I/O buffers) or in an emergency try to continue. In general it is assumed that initially there exists only a single extent of free memory. Nevertheless, all memory management routines should assume that several extents of available memory can be present, since that is the only way to make emergency solutions possible.

The initial state will be as follows: In the non-multitasking case, the whole free memory space obtained at load time is made the current stack extent for type-1 (i.e. runtime stack) housekeeping. The initial free memory list for type-2 allocations will be empty. In the multitasking case, for each task, at attaching, a current stack extent of standard size (e.g. 2048 Bytes) will be allocated for runtime stack housekeeping and an empty type-2 list is obtained. Available free memory is recorded by the wholesaler in a linked list similar but not identical to that list used for type-2 allocations. This list is empty after initialisation of non-multitasking programs but will be filled during the program. If any stack extent for type-1 allocations is exhausted and the list of available memory is able to satisfy the actual storage request, a new type-1 stack extent is delivered by the wholesaler with either standard size or requested size or remaining size depending on the amount of storage requested and on the size of available memory. Otherwise, an emergency routine of the wholesaler will try to obtain additional memory from the operating system, or even from the space actually available in the code area in case of code segmentation. Unused parts of the old stack extents can be added to the free memory list, if they have reasonable size.

There is some freedom of choice how type-2 memory can be obtained. The best way is to cut the requested memory from the top of the current stack extent if the list of available type-2 memory is not able to satisfy the request (which is the normal case after program initialisation, when the list is empty).

4. Conclusion

The solution discussed here has been suggested by the methods used in two compilers: The SIEMENS SYSTEM 4004 DOS/TDOS ALGOL compiler and the IBM OS/360 F-level PL/1 compiler.

As ALGOL does only deal with data of AUTOMATIC type, the SIEMENS SYSTEM 4004 ALGOL compiler is mainly concerned with runtime stack requests, although there exist some type-2 storage requests for I/O supporting items which, however, can be handled including hole usage in quite an easy way owing to the restricted number of simultaneously active data sets. The SIEMENS DOS and TDOS operating systems have no facility for dynamic request of memory resources. Thus, the rather simple and time efficient method of handling only one memory extent was chosen with runtime stack memory growing from the bottom upwards and type-2 memory growing from the top downwards within this single extent obtained at load time. It only remains to prevent these two parts from overlapping at any allocation. Storage overflow leads to abnormal termination.

On the other hand, the PL/1 compiler seems to make extensive use of the OS/360 GETMAIN/FREEMAIN supervisor calls:

BASED- and CONTROLLED-storage is handled via these supervisor calls with the exception of AREA allocations, while AUTOMATIC storage is also delivered by the operating system but in blocks of reasonably large size. Thus, AUTOMATIC memory retailing is achieved in most cases without issuing of a supervisor call. The pointer structure used for this purpose, however, requires updating of many pointers on each allocation and freeing, thus creating considerable overhead for these actions. This overhead can be reduced by introducing the method of the *current stack extent* which increases allocation speed to almost that of the 4004 ALGOL compiler but moreover is able to use memory in the most efficient way as does the IBM PL/1 compiler. In addition to that, the use of supervisor calls for each type-2 allocation event is undesirable. This type-2 memory should be obtained from an own type-2 free extent list. The PL/1 list processing facility, which represents a very useful new language facility compared to conventional programming languages, ought not to be made too slow.

Acknowledgement

The authors would like to thank Mr. Howard Webb for giving some assistance in preparing the manuscript.

Reference

KNUTH, D. E. (1968). *The Art of Computer Programming, Vol.* 1: *Fundamental Algorithms*, Addison-Wesley: Reading, Mass., pp. 435-455

Downloaded from https://academic.oup.com/comjnl/article/15/1/18/418338 by guest on 19 April 2024