Compile Time Memory Allocation for Parallel Processes

GREGOR V. BOCHMANN

Abstract—This paper discusses the problem of allocating storage for the activation records of procedure calls within a system of parallel processes. A compile time storage allocation scheme is given, which determines the relative address within the memory segment of a process for the activation records of all procedures called by the process. This facilitates the generation of an efficient run-time code. The allocation scheme applies to systems in which data and procedures can be shared among several processes. However, recursive procedure calls are not supported.

Index Terms—Code optimization, efficient variable access mechanism, memory allocation for activation records, overlays, parallel processes, storage allocation.

I. Introduction

In most computer systems the sharing of data structures and procedures among several processes plays an important role. Several language constructs have been proposed for the design and programming of systems of parallel processes with shared data and procedures. We mention in particular the concept of user-defined and abstract data types [1] and of monitors [2].

In this paper we consider the problem of allocating the necessary storage for the activation records and local variables of procedures called by the different processes. A variety of different storage allocation schemes have been used, depending on the computer architecture and the applications. We mention in particular the following schemes (in order of increasing flexibility of use and complexity of implementation):

1) fixed, disjoint allocation of activation records, such as used for Fortran;

2) overlays, i.e., fixed but nondisjoint allocation;

3) dynamic stack allocation, such as used for Algol 60 and other recursive languages;

4) allocation by segments of different size, with or without garbage collection, such as used for Pascal's pointed variables or Algol 68's *heap.*

Related to the storage allocation is the problem of program and data relocation. For Fortran programs, the problem is usually solved during the object module loading phase. For Algol or Pascal programs, which use a stack allocation scheme, variables are normally accessed via display registers that contain the base addresses of the different activation records; this makes the relocation easier. In systems with independent segments of virtual memory for different processes and data (e.g., Multics), relocation is implicitly performed during run time by the memory accessing mechanism.

In this paper we describe an allocation scheme that uses independent segments of memory for the different processes in the system. The relative locations within the process memory segment of the activation records of the called procedures can be determined during the compilation phase. This is, in fact, an overlay scheme suitable for nonrecursive procedures which may be shared among several processes and which have local variables of fixed size. Because the overlays can be determined during the compilation phase, it is possible to establish more efficient run-time mechanisms for variable accessing and procedure calling.

This allocation scheme may be used for an efficient implementation of programming languages such as Concurrent Pascal [3]. We note, however, that the scheme may be used for any system of parallel processes with shared procedures that satisfy the necessary conditions.

II. An Example

We illustrate the problem of storage allocation for parallel processes by the example of a communication system, which is described in more detail in [4]. The system implements a full duplex synchronous communication protocol for two-way data communication over a point-to-point data link. Each side of the communication link contains four parallel processes.

Fig. 1 shows the system on one side of the link. There is a *source* process which generates messages, a *sink* process which consumes messages that come from the opposite side of the communication link, and the communications control program, which consists of a control monitor, which is called upon by the source and sink processes, and two processes that look after the transmission and reception of frames. The processes execute certain procedures which refer to the control monitor or the transmission lines, as indicated by the arrows. The *sender* and *receiver* processes are synchronized with the speed of the transmission line. The *source* and *sink* processes are synchronized relative to this speed by the control monitor.

The inner structure of the control monitor can be described as follows. It consists of the four procedures: *send*, *receive*, *get*, and *put* called by the processes. Mutual exclusion between the execution of these procedures is ensured by the monitor. These procedures, in turn, call the procedures *enter* and *withdraw* associated with the data structure of a message buffer. For example, the procedures *receive* and *get* call the procedure *withdraw* for obtaining a message from the *input* buffer and *output* buffer, respectively.

Manuscript received June 17, 1976; revised May 5, 1978. This work was supported in part by the National Research Council of Canada and the Quebec Ministère de l'Education.

The author is with the Département d'Informatique et de Recherche Operationnelle, Université de Montreal, Montreal, P.Q., Canada.

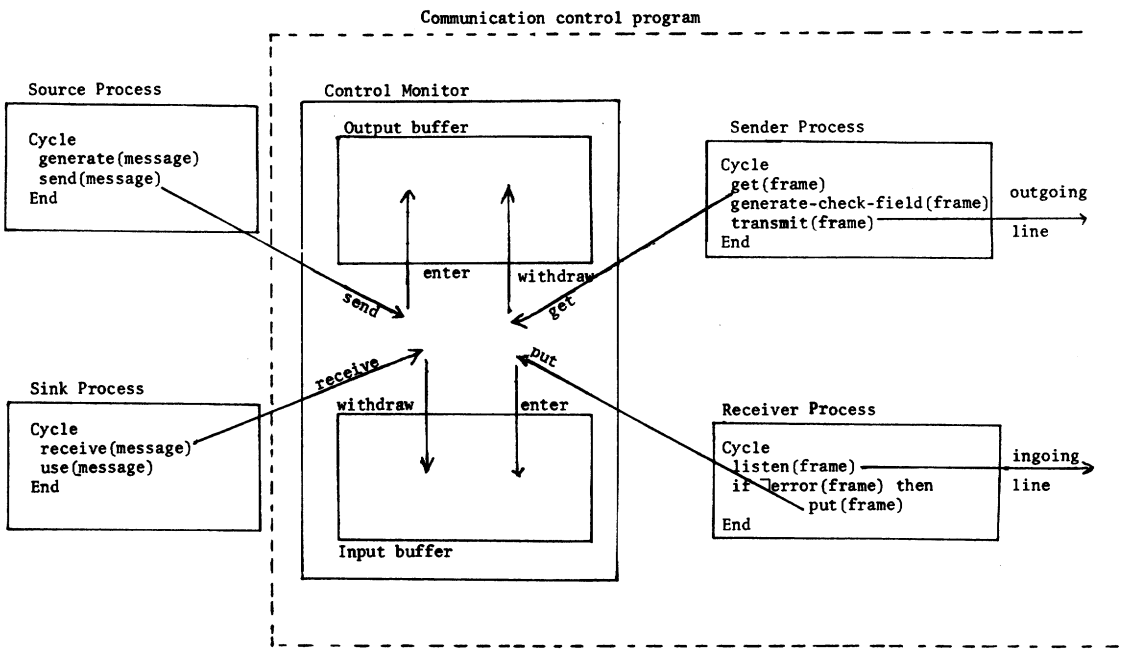


Fig. 1. Structure of a data communication system.

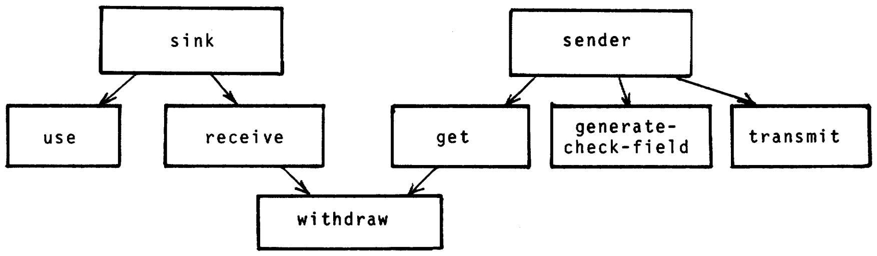


Fig. 2. Calling relation of *sink* and *sender* processes with shared procedure *withdraw*.

The calling relations of the *sink* and *sender* processes, which both use the buffer procedure *withdraw*, is shown in Fig. 2. The usual dynamic stack allocation of the activation records of the called procedures on the stack of the calling process yields the activation record displacements shown in Fig. 3. This allocation scheme may be called "top-down" because the displacements are obtained by traversing the graph of the calling relations from the calling processes down to the called procedures.

It is important to note that for a shared procedure, such as *withdraw*, the displacement of the activation record depends on the calling process (see Fig. 3). In the case of the allocation scheme described below, this displacement is independent of the calling process, as shown in Fig. 4. Following a kind of inverse stack discipline [5], this allocation is obtained by traversing the graph of the calling relations in the opposite order, i.e., from the bottom up.

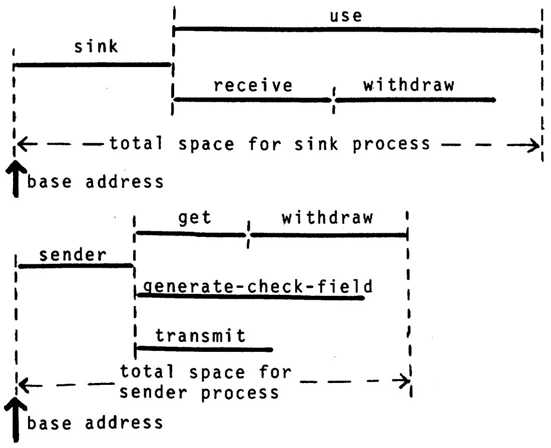


Fig. 3. Memory segments for *sink* and *sender* processes with dynamic stack allocation.

III. Bottom-Up Overlays

A. Assumptions

For the memory allocation scheme described below, we suppose that each process of the system has its own memory segment which contains the variables of the process and the activation records of all procedures called by the process. Data structures shared between several processes may either be allocated within the segment of the process in which they are declared or may have their own memory segment. In all cases, the allocation records of the procedures associated with these data structures are allocated in the segment of the calling process.

A memory segment is identified by a segment number or by

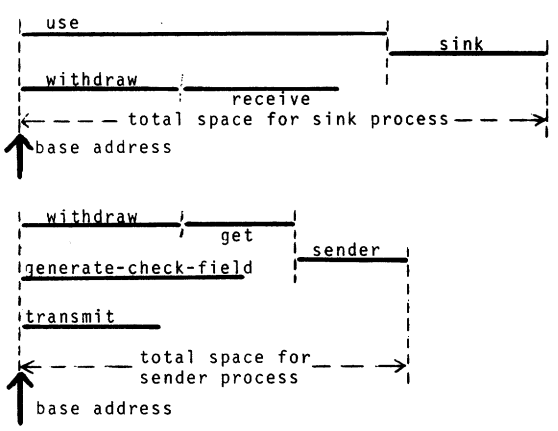


Fig. 4. Memory segments for *sink* and *sender* processes with "bottom-up" overlays [as defined by (1)].

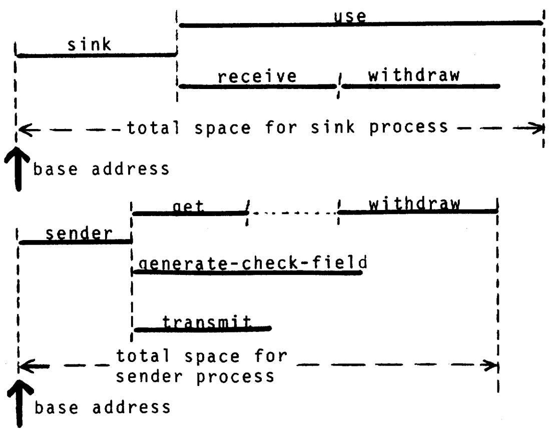


Fig. 5. Memory segments for *sink* and *sender* processes according to the usual overlay scheme [as defined by (2)].

its base address in central memory. During run time, a variable is addressed by identifying the memory segment and giving the relative address of the variable within the memory segment. This addressing scheme can be efficiently implemented by indexed addressing, reserving one base or index register for the base address of the active process.

The number of processes in the system is either fixed (as in Concurrent Pascal) or may vary dynamically. In the latter case the creation of a new process poses no particular problem (as long as sufficient central memory is available), whereas the deletion of a process poses the problem of garbage collection at the level of memory segments for processes.

We suppose that the following restrictions are satisfied:

1) there are no recursive procedure calls;

2) the size of all local variables of procedures can be determined at compile time;

3) procedures are not passed as parameters (this restriction could possibly be suppressed).

B. Definition of the Overlay Scheme

Given a procedure x, we write *size* (x) for the size of its activation record (which contains the necessary call return information, parameters, local variables, and possibly some space for temporary storage).

We write $x \rightarrow y$ if the body of a procedure x contains a call of procedure y, i.e., there is an arrow from x to y in the graph of the calling relation. Because of conditions 1) and 3) above, this graph has no cycles, and the relation generates a partial order on the set of procedures in the system.

For each procedure x we write *low-addr* (x) for the relative address of its activation record within the memory segment of the calling process. (Note that it is independent of the calling process.)

The *bottom-up overlay scheme* is defined by the following equation which holds for all procedures x of the system:

$$\textit{low-addr}(x) = \begin{cases} 0 & \text{if there is no procedure } y \\ & \text{such that } x \rightarrow y \text{, or} \\ \max_{x \rightarrow y}(\textit{low-addr}(y) + \textit{size}(y)) & \\ \quad \text{where the maximum is taken over} & \\ \quad \text{all procedures } y \text{ such that } x \rightarrow y. & \end{cases} \tag{1}$$

We note that the usual overlay scheme used in the case of a single process is defined by a similar equation:

$$\textit{low-addr}(x) = \begin{cases} 0 & \text{if there is no procedure } y \\ & \text{such that } y \rightarrow x \text{, or} \\ \min_{y \rightarrow x}(\textit{low-addr}(y) + \textit{size}(y)). & \end{cases} \tag{2}$$

In the case of procedures shared between several processes, this overlay scheme leads to an allocation of process segments which are not necessarily minimal, as shown in Fig. 5. Therefore, this allocation scheme is not suitable for systems of parallel processes.

C. Allocation Algorithm

An efficient general algorithm for evaluating (1) for all procedures of a given system is as follows.

Step 1: Determine an order for traversing the graph of calling relations such that, when a procedure x is encountered, all procedures y with $x \rightarrow y$ have been encountered previously. Such an order can be obtained by a topological sort [6].

Step 2: Traverse the graph of calling relations in the order obtained in Step 1, and evaluate (1) for each procedure x encountered. This evaluation poses no problem since, for all procedures y considered in the expression, the value of *low-addr* has been determined previously.

In the case of a programming language in which a procedure must be defined before it can be used, as, for example, in Concurrent Pascal [3], the programmer must write the procedures of a system in an order corresponding to the one obtained in Step 1 above. Therefore, a compiler could determine the bottom-up overlays (Step 2 above) in a single scan of the program text from left to right. During the parsing of the declaration part of a procedure x, the size *size* (x) of the activation record is determined. During the parsing of the executable statements of the procedure x, a list of the called procedures is established. At this point, all these called procedures have already been parsed and their *size* and *low-addr* are available. Now the relative address, *low-addr* (x), of the activation record of x is determined according to (1) above.

Note that the relative address of an activation record is only known after the complete procedure has been parsed. This information, therefore, cannot be used during the same pass through the program text for the generation of variable access code. This means that the compiler must perform at least two passes over the program text. If a one-pass compilation is preferred, a relocating link editor may be used for the second pass, while the compiler generates a code assuming a zero relative address for all activation records.

D. Comparison with the Dynamic Stack Allocation Scheme

The following points provide a brief comparison of the bottom-up overlay scheme with the dynamic allocation of activation records "on the stack" [7].

Restrictions: The bottom-up overlay scheme requires the restrictions 1), 2), and 3) given in Section III-A. These restrictions do not apply to the dynamic stack allocation.

Used Memory Space: The memory space allocated for each process is minimal in the case of both allocation schemes. It is easy to see that the size of the memory segment for a process x, where x also denotes the "main" procedure of the process, is equal to the maximum of $\Sigma_{i=1}^{n}$ *size* (z_i) over all paths $x = z_1 \rightarrow z_2 \rightarrow \cdots \rightarrow z_n$ in the graph of calling relations. Each of these paths represents a possible sequence of embedded procedure calls executed by the process x.

Run-Time Efficiency: In the case of the bottom-up overlay scheme, the relative address, with respect to the base address of the calling process, is known at compile time for the activation record of each procedure of the system, being shared or not shared among several processes. Therefore, local variables of called procedures, as well as the variables declared within the process, may be accessed with a fixed displacement with respect to a register pointing to the base address of the calling process' memory segment. This is also true for accesses of nonlocal variables declared in calling procedures. In the case of the dynamic stack allocation, these accesses are usually made through pointer chains or display registers.

Procedure calls can be implemented more efficiently with the bottom-up overlay scheme than with the dynamic stack allocation. For the former, a procedure call involves no updating of addressing information, whereas for the latter, the display or other registers, pointing to the activation records on the stack, must be updated at each procedure call and return.

For example, Brinch-Hansen's implementation [8] of Concurrent Pascal uses the dynamic stack allocation scheme with two base registers ("local base" and "global base") which are both updated during procedure calls and returns. The "local base" is used for accessing the activation record of the active procedure (variables declared in calling procedures are never accessed), whereas the "global base" is used for accessing variables declared in the running process or in the class or monitor components accessed by the process through **entry** procedure calls. If the bottom-up overlay scheme were used, only a "process base" register would be required, which would need no updating during procedure calls. For more efficient access to the variables of class and monitor components, an additional register, similar to the "global base," could be provided which would only be updated during calls and returns of **entry** procedures.

For efficient language implementations, the difference in the efficiency of the procedure call and return primitives between the bottom-up overlay and the dynamic stack allocation schemes could be up to about 50 percent (return address and "local base" updates versus return address alone). For an interpretive implementation such as [8], however, an additional overhead is associated with procedure calls, which reduces the relative gain in efficiency that could be obtained by using the bottom-up overlay scheme.

IV. Conclusions

We have shown that the relative location within the memory segment of a process of the activation records of the called procedures can be determined at compile time according to an allocation scheme of "bottom-up" overlays. In contrast to the well-known overlay scheme for a single process, the scheme can be used for systems in which data and procedures are shared by several parallel processes. Compared with the scheme of allocating the activation records on a dynamic stack, this overlay scheme provides more efficient run-time mechanisms for variable access and procedure calls, but imposes the conditions that procedure calls be nonrecursive and that the memory size of local variables be determined at compile time.

We feel that these conditions represent no serious restriction for many applications, in particular for many special-purpose systems.

The main applications probably lie in the area of small specialized computer systems where efficiency is important, for example, in computer-based communication systems, control applications, and small operating systems.

Acknowledgment

Several interesting discussions with P. Desjardins and J. Vaucher contributed to the development of this paper.

References

[1] B. H. Liskov and S. N. Zilles, "Specification techniques for data abstractions," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 7-18, 1975.

[2] C. A. R. Hoare, "Monitors: An operating system structuring concept," *Commun. Ass. Comput. Mach.*, vol. 17, pp. 549-557, 1974.

[3] P. Brinch-Hansen, "The programming language Concurrent Pascal," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 199-207, 1975.

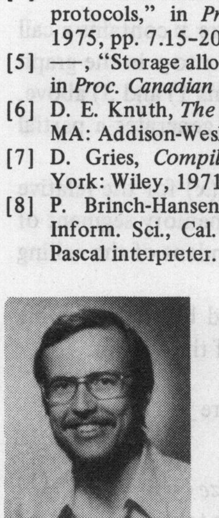
[4] G. V. Bochmann, "Logical verification and implementation of protocols," in *Proc. 4th Data Commun. Symp.*, Quebec City, 1975, pp. 7.15-20.

[5] ——, "Storage allocation for parallel processes in micro-computers," in *Proc. Canadian Comput. Conf.*, CIPS, May 1976.

[6] D. E. Knuth, *The Art of Computer Programming*, vol. I. Reading, MA: Addison-Wesley, 1968, pp. 258-265.

[7] D. Gries, *Compiler Construction for Digital Computers.* New York: Wiley, 1971, sect. 8.9.

[8] P. Brinch-Hansen, "Concurrent Pascal machine," Tech. Rep., Inform. Sci., Cal. Tech, 1975; and the listing of the Concurrent Pascal interpreter.



Gregor V. Bochmann received the Diplom in physics from the University of Munich, Munich, Germany, in 1968, and the Ph.D. degree from McGill University, Montreal, P.Q., Canada, in 1971.

He has worked in the areas of programming languages and compiler design, communication protocols, and software engineering. He is currently Associate Professor in the Département d'Informatique et de Recherche Operationnelle, Université de Montreal. His present work is aimed at design methods for communication protocols and distributed systems. In 1977-1978 he was a Visiting Professor at the Ecole Polytechnique Fédérale at Lausanne, Switzerland.