MODULES
PROGRAM STRUCTURES
and the
STRUCTURING OF OPERATING SYSTEMS

C. Bron
Department of Electrical Engineering
Twente University of Technology
P.O. Box 217, Enschede, Netherlands

Abstract

In this paper some views are presented on the way in which complex systems, such as Operating Systems and the programs to be interfaced with them can be constructed, and how such systems may become heavily library oriented. Although such systems have a dynamic nature, all interfacing within and among modules can be checked statically. It will be shown that the concepts presented are equally valid for single user systems, multi-programming systems and even distributed systems. The ideas have been spurred by the implementation of a modular version of Pascal and a supporting Operating System, currently nearing completion at Twente University of Technology, The Netherlands.

1. Co-operating Modules

1.1 Modules

The basic programming tool we consider in this paper is a module. This concept is incorporated in several recent languages (ADA[1] (where it is called "package"), Concurrent Pascal[6] (where it is called "class"), Pascal Plus[17], LIS[13], MESA[15], Modula[19], Modula-2[20] and many others). Although details may differ from one language to another, the following description should suffice for the sake of this paper:

A module is a set of related (type-)definitions, data declarations, operation declarations (viz. procedures and/or functions) and a section describing the initialization (sometimes also the finalization, as in Pascal Plus) of the module's local data.

In order that programs (or systems) may be composed from co-operating modules some of the declarations within modules may have to be made accessible outside these modules. We will say that these declarations are exported from these modules. Conversely, the use of items declared in other modules will be called "importing". If a program is composed from several modules, then the rules according to which modules may be interconnected determine the accessibility and scope of the objects within the program. It needs no arguing that the visibility structure that can be obtained in this way may well differ from the scope rules one encounters in classical, block structured languages (of which Pascal may be considered an example). A judicious use of the interconnection rules between modules may lead to a simple, but, nevertheless very powerful means of structuring systems, and in particular: Operating Systems. This we hope to show in the sequel.

1.2 Interdependency of Modules

In the following we postulate that each module specifies from which other modules it wishes to import items. Thus a program (composed from modules) may be modelled by a directed graph, where the modules are the vertices and the relation "imports from" determines the (directed) arcs of the graph. (Classical block-structure would limit the structure of such graphs to trees.)

If knowledge of the total set of modules is only used when the program is composed from a set of object modules, as is the case in traditional systems with "independent compilation", it can hardly be checked that the arguments supplied for a procedure

called from a certain module, are type-correct with regard to the definition of such a procedure in another module. Such organizations discard most of the advantages that are generally recognized as to be obtained from full type-checking. In this respect it should be noted that the same insecurity was originally present in Pascal with regard to the arguments of formal procedures. (Fortunately, this flaw has been mended in the forthcoming Pascal standard[2].)

On the other hand, if during compilation of a module, the source texts of all modules from which it imports are available, the full power of type-checking can be maintained. Obviously we must be able to guarantee that the vital part of the exporting modules is not changed later, to invalidate the type checking performed previously. We will return to this consistency issue in 1.4.

Often, it may not be necessary to have available the full source text of an exporting module. E.g. in order to check an actual parameter list for type consistency only the heading of the called procedure (even without formal identifiers) is necessary. In several of the languages mentioned in 1.1. we find such excerpts from modules as language entities. (E.g. definition modules in Modula 2, Mesa.) The best term for such entities seems to be "interface modules".

1.3 Further ordering imposed on co-operating modules

If interface modules are present it seems well possible to compile importing modules, when the implementation of the modules which do the corresponding exporting is not yet given. When the latter is given it only needs to be checked that it complies with its own interface module. Furthermore the same consistency restriction will have to be observed as mentioned in 1.2. Although a scheme with interface modules seems to offer the greatest flexibility, there are arguments in favour of not separating interface modules from their implementation.

Note that the scheme without separately defined interface modules enforces a partial ordering on the compilation of modules, i.e. the directed graph must be acyclic. The most important advantage is the observation that acyclicity of a program's graph guarantees the existence of an instantiation order of modules, such that during initialization of a module it has at its disposal all items imported from other modules, for these modules can be forced to be instantiated first.

As a drawback of this scheme it should be noted that mutual recursion between procedures from different modules is impossible unless at least one of them has been passed as a procedure parameter. In the latter way a procedure defined in an importing module can be made available in an exporting module.

1.4 Partial recompilation, Time-stamping

In this section we discuss some aspects of our current effort in building modular systems. More details are given in [7, 8].

Interface modules as described in the previous chapter are not part of the language, but are produced as a by-product of the compilation of an (exporting) module. They are called specification files. Compilation of a module requires the existence of the specification files of all imported modules. Obviously these files will be the most up-to-date versions and therefore consistency at the time of compilation is guaranteed. However, it must be checked, at the time modules are instantiated, that no exporting module has changed its "outward face" after compilation of correspondingly importing modules. This could be achieved by time-stamping all object modules and checking these time-stamps when the modules of a program are about to be instantiated.

In the case of compiler produced specification files, a considerable relaxation is possible: Instead of attaching time-stamps to object modules, we attach time-stamps to specification files. If - as a by-product of recompilation of a module - the new specification file is identical to the old one, the old one, including its time-stamp, is maintained. We do not give details of the conditions under which specification files remain unaffected, but the major benefit is derived from the fact that neither procedure bodies, nor the initialisation part influence the specification file.

It will be readily seen that in most cases small changes to large systems affect one module only and can be brought about by the recompilation of a single module with the conservation of the benefits of type-checking. We stress - again - that when the modules of a program are instantiated the partial ordering of the modules must be observed by the ordering of the time-stamps of these modules. In other words: for any pair of modules A and B, where B imports A, the time-stamp of the specification file of A must be older than the time-stamp of the object file of B.

(As an aside - at this point - we mention the value of partial compilation for small mini- and micro-based computer systems where the size of the addressing space may create an obstacle to the compilation of large programs and systems as a whole.)

1.5 A model for sequential program execution

In a program (composed from a number of modules) one particular module, the export of which - if at all present - is not used by any other module, can always be identified

as the "main program" (or: "main module"). It may be considered as the root of the directed graph. One might say that the external effects of the program are the side effects of that particular module's initialisation. Now let us look at the way such a program's execution might proceed. As will be made more explicit in 2.1, we want the lifetimes of modules to be strictly nested, so this nesting also holds for the data of these modules. Therefore, a stackwise allocation scheme for module data must be implemented. This will be accomplished in a handsome manner by a set of nested procedure activations, as will be described now.

First consider a linear ordering of the vertices of the graph in such a way that the partial ordering is obeyed. The main module of the program is at the top, and some module that does not import from any other module will be at the bottom. Given this ordering, and considering each module as a procedure, the environment (usually called the "Operating System") invokes the bottom module, and each module in turn invokes as its last action (i.e. after it has performed its initialisation) the next module in the sequence. It would carry too far - at this point - to describe how modules invoke other modules that are "unknown" to them. After all, the direction of invocation is the direction of exporting, whereas visibility is always in the direction of importing. We leave it at the remark that this instantiation scheme can be accomplished by passing to each module its successor module as a procedure-parameter. Obviously the root module will eventually be invoked with a successor having an empty body. In chapter 2 we will show that the "Operating System" itself may have been instantiated in a manner fully analogous to the instantiation mechanism just described.

Note that this scheme is equally applicable for the "envelopes" of Pascal Plus[17], where each module consists of an initialisation part and a finalization part. At the borderline of the two, the successor module may be invoked.

The reader will have noticed that by instantiating programs in this way, the local data of modules are allocated in a stackwise fashion, and there is therefore no need to treat the data segments of modules in a way that differs from procedural data frames.

1.6 Addressing structure and context switching

In the previous chapter we discussed a trivial scheme for the allocation and initialisation of module data. In this chapter we discuss the addressing of objects across module boundaries. Not only for the sake of brevity but also because this is by far the most interesting aspect, we limit the discussion to the invocation of procedures in other modules: "external call".

We associate with each module (and allocate in its local data space) a table containing one entry for each imported module: the "environment display". Each entry contains the "base-address" of the local data of the corresponding module (note the similarity with the display concept to administer statically nested blocks.)

Because the instantiation order obeys the partial ordering of the vertices of the program-graph, all the addresses to be filled in in the environment display are available at the moment a new module is to be instantiated.

External call may be compared to the mechanism for calling a formal procedure which also has the property that the calling context and the called context may have nothing in common. External call is even simpler, since the procedure to be activated can only be declared at the outer block level of a module and therefore the addressing environment that has to be created is extremely simple.

It will be evident that an external procedure can be activated by accessing the display element corresponding to the module in which the procedure is declared and providing the code-location of the procedure relative to its code-segment-base. The calculation of the actual code-address implies one simple addition of the code-segment-base which may be stored in a fixed position of the new module's data frame.

In addition to the return information that has to be stored for any procedure call, the address-base of the module being left must be saved (and restored upon return). (For the PDP11 the full call/return mechanism takes approximately 8 instructions.) It may be superfluous to remark that the communication of parameters and function results may proceed in a normal, stackwise fashion.

It may seem unnecessary to spend so many words on such a simple context switching mechanism, but even today there is evidence that procedure calls are burdened by the implementation to such an extent that programmers tend to shy from procedures like the plague, and compilers do their utmost to substitute in-line code for procedure calls where ever this is defendable[18].

Having presented the mechanism for external call in its most simple form, it seems worthwhile to remark that the concept of module switching can be used in a variety of ways. For instance, in a system with an addressing space smaller than the memory space, the point of module switching may be used to adapt the address map such that the new module appears in the code addressing space (a single map register would do, provided each code segment is located in contiguous memory locations). In our case we exploit the external call/return mechanism slightly further by adding a test on presence of the invoked module's code segment. If not present, it will be loaded

from backing store. Notice that, in essence, this provides a low overhead virtual memory mechanism for program code. Suitable hardware or microcode to perform the above simple call/return operations would reduce the overhead to become negligible.

2. Operating Systems composed from Modules

2.1 A basic operating system structure

Having discussed at length how a program may be composed out of modules, we postulate that an operating system may display exactly the same structure. It may contain modules that provide service to the system itself (c.q. terminal i/o and filing services) as well as to any programs to be run "on top" of the operating system.

The only difference being that the operating system "main program" does not terminate and that - in order to maintain the analogy with the instantiation of program modules - some form of bootstrap (at least for the bottom module(s) of the system) must be devised. We propose that the main task of the operating system is the administration of modules. The modules in the system may be divided in two groups: the active modules which - currently - are participating in the system, and the passive modules which reside in the file system. We now consider the "running" of a program as the shift of those modules constituting the program and not already active (!!) from the passive state to the active state, by instantiating each of these modules in an allowed order. Modules are therefore implicitly shared by different programs. In particular the O.S. calls on its own services in exactly the same way as a "user" program does. In fact, there is no distinction between Operating System and application. At any instant in time, the system as a whole may be viewed as a set of co-operating modules, sometimes expanding on account of RUN('A program'), at other times shrinking, on account of the termination of 'A program'.

In order to prepare the chapters that are to follow, we describe the concept of RUN in more detail. The argument of RUN is the (unique) name of a module. (Possibly a file-name.) Central in our description is a structure we will call the "load-table", containing relevant data of all currently active modules, in their instantiation order. Let, for the basic system, the structure of the load-table be given by fig. 1, and consider the activation of

```
P(importing: D, E, A), where
    E(importing: D)
    D(importing: "nothing")
```

(In the diagram, the import relations are given by downward arrows.)

The action RUN('P') proceeds in two phases. First the load-table is extended, to become (see fig. 2):

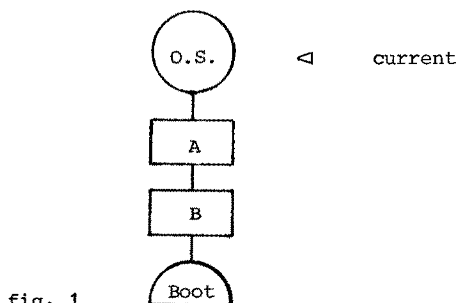

fig. 1

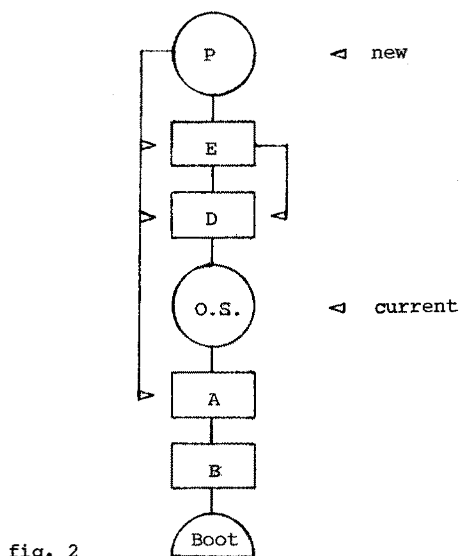

fig. 2

Next "current" is moved up, instantiating each module it passes on its way until current equals "new". This process has been described in a different terminology in 1.5.

If the procedure RUN is exported from its defining module, which is our explicit intention(!), there is nothing that prevents module P from instantiating another program on top of itself. We believe that in this way the sharing of program modules can be carried to its extremes. E.g. all programs within such a system may share the same routines for binary/decimal conversion. Different versions of a compiler may share the majority of their modules. Although this seems a natural approach, it has not been accomplished in several, otherwise attractive operating systems: (Burroughs MCP, DEC TOPS 10).

The above will also make clear, how the system bootstrap can be accomplished. The initial system-structure is given by fig. 3. Now the second phase of RUN can be started. This scheme makes clear that only one module needs to be loaded in a non-standard way, viz. 'Boot', and that the load-table, its initial contents, and the procedure RUN must be implemented in that module. A very modest implementation of the module O.S. might look like the following:

```
program OS(importing: A, B, Boot);
var name: filename;
begin loop readmodulename(name);
            RUN(name)
      end
end.
```

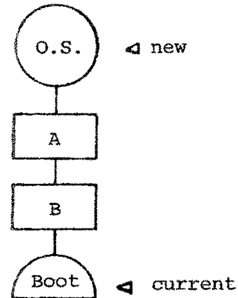

fig. 3

2.2 Structure of the load-table, linking of modules

The load-table is the central structure in the modular system. It contains one entry for each module that is active or about to be activated. Each entry consists of the unique name of the module, its data address base (only for those modules up to and including "current"), the backing store address of the module's code segment if segments are to be dynamically loaded, or the primary code address if modules have been loaded into memory on account of the first phase of expanding the load-table.

Based on this structure we require the existence of a procedure which yields the data address base of a module in exchange for that module's name. The traditional role of "linkage editing" now shrinks to the following:
Each module builds its environment-display (see 1.6) by calling the above procedure once for every module it imports. Note that such calls take place only for modules which have already been instantiated, the address base of which is therefore already defined.

As a complementary obligation, each module must define, in its own load-table entry, its data address base before it instantiates the next module.

3. Parallel Processes and Distributed Systems

3.1 Spawning of parallel processes

We now turn our attention to an environment with parallel processes, based on the same structuring principles as discussed before. Very briefly we relate an experience with parallel processing in a purely sequential language environment. We will not dwell on the complications that arise for memory management when separate stacks for parallel processes have to be allocated:

Parallel processes can be implemented with the aid of two "extra-ordinary" routines. One is needed to set up the initial data space for a process (much in the same way as must be done for the initial system bootstrap), which needs detailed knowledge of the mapping of the language on the target machine (register usage, stack-layout etc.). A second routine will be responsible for process switching, i.e. the current status of the target machine must be saved in the data space of the process being switched from, and the new status must be (re)loaded from the data space of the process being switched to. (In some machines this routine can be recognized as a hardware instruction [Burroughs B6700: MOVE STACK].)

On top of the above, process queueing may be organized, implementing logical wait queues and a CPU queue. Within this organization any form of scheduling (run-to-blocked, time-slicing (if a clock is available), priorities) may be implemented [10].

How does parallel processing fit in the system's module structure discussed so far? We propose an analogon to the procedure RUN, say FORK('A module'). Fork may be described as we did for RUN, with regard to the instantiation of new modules. However, the module responsible for FORK remains active as a process itself. In a pictorial representation Fork is indicated by an oblique line in stead of a vertical one. (See fig. 4)

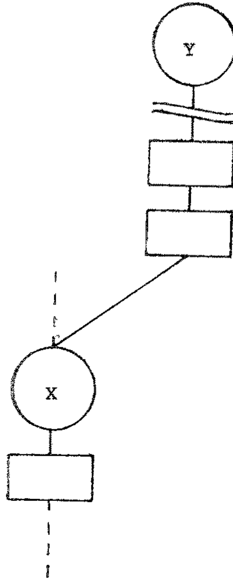

fig. 4

As a result of FORK('Y') in X, Y and X may share all modules up to and including X in the load-table, but whether they actually do so is dependent on the import structure of Y (and X). Let us consider a producer/consumer pair as a concrete example. The role of either one could be played by X or Y in the above example, but for reasons of symmetry, we prefer a second spawning of a process. The spawning module could be the module 'buffer', exporting the operations "get" and "put", whereas the consumer and producer, both importing buffer, would be able to communicate via the shared module 'buffer'. Obviously, any synchronization required for this co-operation would have to be programmed explicitly in the buffer module.

It will be obvious how this scheme can be exploited if - for instance - more than one consumer/producer pair has to be instantiated:

```
FORK('buffer'); FORK('buffer')
```

The presence of the concept of "FORK" forces the structure of the load-table to become a tree, but to each individual process, only the underlying path to the root of the tree is directly or indirectly accessible, that is: as viewed from a module the relevant part of the load-table still behaves as a linear LIFO-list.

(The actual implementation of the load-table as a tree will have to be somewhat more complicated if one wants visibility control to follow the above scheme, but - at the same time - one wants to accomplish the sharing of code segments even between instances in parallel branches of the tree.)

One might argue that implicit sharing of modules is not always desirable and that there should be a way to indicate that additional module instances have to be set up in the same path of the load-table. The counter argument is, that procedure activations are the objects which are implicitly non-shared. Since procedure bodies are in no way restricted in comparison to module bodies, the desired effects of multiple module instances can be obtained by nested procedure activations.

To illustrate this, let us consider the UNIX shell (command interpreter)[16]. Let this shell be a procedure declared in the shell module (which should have no local data in this case). The "shell", being a command interpreter, may RUN or FORK a program. This program in turn can invoke the "shell" as a procedure, provided it imports the shell module. And so we obtain multiple instances in a very natural manner, at the same time sticking to our principle of maximal sharing of modules.

3.2 Synchronization of parallel processes

It is our view that the decisions to be made about the synchronization of parallel system components are not part of the system structure described here, and have to be taken at another level of the system design. We have, also, serious doubts whether uniform (language-enforced) decisions on synchronization structure are desirable. Among all proposals for synchronization made and investigated so far, there is no clear-cut favourite [4, 5, 9, 14].

The viability of a particular mechanism is too often dependent on the characteristics of the application. E.g. when producer/consumer relations have to be implemented (a not infrequent occasion: spoolers, pipes, ...) P and V operations are still at the top, whereas they are rather impractical when complicated logical expressions control the synchronization of processes. Similarly, one might think of the buffer module described in 3.1 as a Concurrent Pascal monitor, but the mutual exclusion thus imposed on "get" and "put" may be much more restrictive than is actually desirable. So we conclude that we should implement only very primitive operations for process synchronization and scheduling, and leave it to the system designer to build other mechanisms on top of the primitive ones, as the situation requires.

It is the implementor's obligation to design modules that are intended to be shared by parallel activities in such a way as to avoid conflicts. The fulfillment of this obligation does not affect the system structure. If we take the file administration as an example, it will be clear that more safe-guards have to be built in in a parallel environment than in a purely sequential environment. Nevertheless, the interface the file administration presents to importing modules may remain the same, and even should remain the same if one wants potential parallelism to be transparent

to importing modules.

3.3 Distributed systems

We now carry our principles for structuring systems one step further and apply them to distributed systems. To this end we postulate a third basic system building operation which we will call FORK_REMOTE.

The arguments, this time, are the (unique) name of a module and the (unique) identification of a node in a distributed system. The effect of this operation is comparable to that of FORK, but for the fact that the branch of the load-table to be spawned will be physically located in the node identified as an argument (fig. 5).

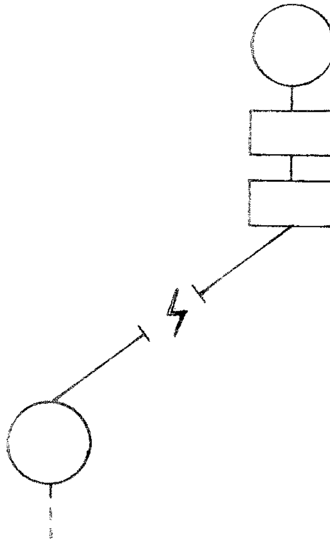

fig. 5

All that will be needed in addition to what we already have for a parallel system is a procedure calling mechanism across the link, possibly restricted with regard to the kinds of arguments that can be passed in such a call. E.g. <u>var</u>-parameters may have to make way for a value/result form of parameter passing.

The picture sketched above is an oversimplification, since, in the spawning node of the network, we need a process that acts as the extension of the spawned branch, and which is willing to accept the requests for procedure calls and to transmit the results back to the calling node. Note that the structure of such an extension process can be very simple since it contains no internal parallelism. (fig. 6)

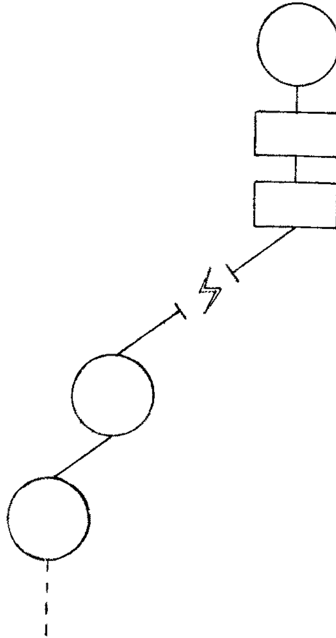

fig. 6

Is the above scheme simply implementable? We believe it is. Of course, it is now insufficient to have in the environment display of a module, address bases only. Each element must also contain a node identification. Given an element of the environment display, the mechanism for external call may now detect easily that some calls have to be directed to foreign nodes, and, in order to get this accomplished, invoke the actual routines that take care of remote procedure invocation. In fact, the scheme is surprisingly similar to that mentioned at the end of 1.6 where the dynamic loading of code segments is delegated to the external call mechanism.

Apart from its conceptual simplicity, we must reflect on some of the potential drawbacks.
The load-table will be distributed over the nodes of the network, which makes the linking of modules somewhat more costly. On the other hand, it may be attractive that - as long as no malfunctioning takes place - each node has up-to-date knowledge of the system structure in those parts of the network it depends on. Failure in a node A which has spawned a process in a node B need not be fatal as long as the actions in B do not perform remote invocations on A. As soon as they do, however, the malfunctioning of A will be detected and it can be coped with in B much in the same way as local malfunctioning is coped with.

It might also be argued that the system is asymmetric in the sense that the node, from which all activities are originally spawned, is too central and too vulnerable within the system. It seems that this need not be the case. We only propose that

the system, when being set up, sprawls out from one node. There may be several nodes in the system able to perform this task. Whether this initial node plays a central role in the system's further activities is highly dependent on the import structure of the modules that are distributed over the nodes.

3.4 Security, protection, access control

In this chapter we briefly discuss some aspects of protection within systems, structured along the lines indicated. The main point to be stressed is that within this form of co-operation between nodes in a network, most of the interface checking can still be delegated to compile-time checks and it seems that very little run-time protocol checking will be required. In contrast, both message oriented systems and capability based systems are to a much higher degree dependent on run-time checks.

With regard to security, we will discuss three aspects:

- security in the use of addresses
- security across node boundaries
- accessibility of modules

The first aspect is the classical one of protecting address spaces. Ideally, the language used for programming the system should be fully fool-proof, or some hardware support should be provided. For Pascal, it is well known that pointers, the heap and record variants are unsafe features. In order to have a secure system at the same time preserving the unsafe features of the language, it needs to be checked that the result of each address calculation falls within the bounds set for the current module. (Such checks are bound to be inefficient in the absence of supporting hardware!) It should be allowed that, at the time of use, an address calculated previously, lies outside the bounds set for the current module. (var-parameters!). External call/return is responsible for switching the address map, including the limits for the addressing space.

The second aspect is more interesting:
Since all communication between nodes in the network takes place by means of remote procedure call and the passing back of results, it will be impossible for one node to break the security in another, provided the procedures are safe with respect to the node they are active in. Therefore, even if locally we can not guarantee perfect operation, malfunctioning can never be caused by non-local events. In other words, the system is safe with respect to the Trojan Horse.

Third, and this seems to be the most interesting aspect, we may wish to exercise access control at the module level. For instance, within the Operating system we may

have some modules that are to be called from other O.S. modules, but not directly by "user programs". This form of access control can be borrowed in a straightforward way from the access control which is generally found in file systems.

Within this scheme a discipline for the use of a particular module can easily be enforced by making externally available only those modules that are known to stick to that discipline, and not the particular module itself. Note that it is not the identity of the ultimate user which determines the access rights, but the identity of the using module.

The exploitation of this form of access control may yield a tremendous increase in the efficiency of intermodule access, since accesses are checked at compile-time, and only once for every inter-module connection (arc in the directed graph representing the system). This strongly contrasts with capability based systems where capabilities are checked at runtime, at the procedure calling level. The latter either hampers efficient implementation [21], or requires additional hardware and storage [11]. It can not be stressed enough that the most vital abstraction mechanism presently available, notably procedures, should be implemented with such efficiency that it induces its frequent use.

Summary

In paragraphs 1.1-3 we have described an addressing structure, based on program modules and visibility control obtained from modular interconnections forming an acyclic directed graph. Out of these modules systems can be composed in a variety of dynamic ways. The classical concept of "running a user program" is viewed as a temporary extension of the system (1.5), where the boundary between Operating System and user program has actually vanished.

A central concept in such systems is a data structure we have called the "load-table", which forms the administration of the modules currently active in the system (2.2). Around this administration, several system structures ranging from dedicated single user systems (2.1) to distributed systems (3.3) may be conceived. All of these systems show a surprising amount of similarity with regard to their addressing structure. They can be considered as members of a single family [12]. A reasonable amount of protection in these systems can be obtained at low overhead. A possible implementation of the addressing structure has been described in 1.6, but many of the other details of the implementation of such a system can be filled in in a variety of ways.

Several classical concepts like linking and loading either have disappeared altogether, or display a very natural form within this system structure (2.2). The systems of this family are particularly suitable for program sharing and the support of software libraries. Procedural parameters - so often neglected in programming languages (ADA) - turn out to be an extremely useful concept, even at the level of so-called hard systems programs.

Acknowledgements

The original concepts of the addressing structure within programs composed from modules are due to Eelco J. Dijkstra, and were implemented by him in a portable compiler for a version of Pascal supporting this module concept. The ideas with regard to Operating System structures were further developed in discussions with P.J. Voda, and recently with Wiek A. Vervoort and Albert L. Schoute in the course of the development of a "high band-width" distributed Operating System.

References

[1] ADA: Preliminary reference manual
ACM SIGPLAN Notices 14 (6, June 1979)
[2] Addyman, A.M., A draft proposal for Pascal
ACM SIGPLAN Notices 15 (4, April 1980), 1 - 66
[3] Andler, S.,
Synchronization primitives and the verification of concurrent programs
2nd International Symposium on Operating Systems
IRIA, Rocquencourt, France (October 1978)
[4] Bloom, T., Evaluating synchronization mechanisms
Proc. 7th Symposium on Operating Systems Principles,
ACM SIGOPS, Asilomar, Ca., U.S.A. (December 1979)
[5] Brinch Hansen, P., A comparison of two synchronizing concepts.
Acta Informatica 1 (1972) 190 - 199
[6] Brinch Hansen, P., Concurrent Pascal
IEEE Transactions on Software Engineering (1975) 199 - 207
[7] Bron, C., Dijkstra, E.J., A Pascal compiler redone
Memorandum 283, Department of Applied Mathematics,
Twente Univerity of Technology, Enschede, Netherlands
[8] Bron, C., Report on the programming language THT Pascal for the PDP11 series
Memorandum #296, Department of Applied Mathematics
Twente University of Technology, Enschede, Netherlands
[9] Bron, C., Trends in the synchronization of parallel processes
Informatie (1974), 646 - 651 (Dutch Computer Society Monthly)
[10] Bron, C., Pascal used for Operating System implementation
Pascal Conference
Chalmers Technical University, Gothenburg, Sweden (June 1980)
[11] Gehringer, E.F., Variable-length capabilities as a solution
to the small object problem.
Proc. 7th Symposium on Operating Systems Principles
ACM SIGOPS, Asilomar, Ca., U.S.A. (December 1979)
[12] Habermann, A.N., Flon, L., Cooprider, L.
Modularization and hierarchy in a family of Operating Systems
Comm. ACM 19 (May 1976) 266 - 272
[13] Ichbiah, J.D., Rissen, J.P., Heliard, J.C., Cousot, P.,
The system implementation language LIS
Technical Report #4549 E1/EN CII, Louveciennes, France (1976)
[14] Lauer, H.C., Needham, R.M.
On the duality of Operating System Structures
2nd International Symposium on Operating Systems
IRIA, Rocquencourt, France (October 1978)
[15] Mitchell, J.G., Maybury, W., Sweet, R.,
MESA Language manual
Xerox, Palo Alto, Ca., U.S.A. (April 1979)
[16] Ritchie, D.M., Thompson, K.,
The UNIX Time-sharing system
Comm. ACM 17 (July 1974) 365 - 375
[17] Welsh, J., Bustard, D.W.,
Pascal Plus - Another language for modular multiprogramming
Software P. & E. 9(1979) 947 - 957
[18] Wichmann, B.A.,
'Ackermann's Function': A study in the efficiency of calling procedures
BIT 16 (1979) 103 - 110
[19] Wirth, N., Modula: a language for modular multiprogramming
Software P. & E. 7 (1977) 3 - 36
[20] Wirth, N., Modula-2
ETH Zuerich, Technical Report #36 (March 1980)
[21] Wulf, W.A., Harbison, S.P.,
Reflections in a pool of processors
Carnegie Mellon University, Technical Report CMU-CS-78-103 (February 1978)