MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Project MAC

Computation Structures Group Memo 63

Management of Names in a Computer System

Jack B. Dennis

November 1971

This material appeared as the fifth module of an outline for an undergraduate subject on operating systems. The outline was prepared by a Task Force of the Committee on Computers in Electrical Engineering (COSINE) of the National Academy of Engineering with support from the National Science Foundation. Copies of the complete outline "An Undergraduate Course on Operating Systems Principles" may be obtained from National Academy of Engineering, Washington, D.C.

Management of Names in Computer Systems

Name management in computer systems concerns meeting the following important system objectives concerning the use of memory to hold the procedures and data structures involved in computations:

1. Long-term storage of information.
2. Controlled sharing of access to data bases and procedures.
3. Creation, deletion, growth and shrinkage of information objects during the course of computations.
4. Program modularity, the ability of users to construct programs by linking together subprograms without knowledge of their internal operation.

These objectives must be met at the system level because they concern use of shared resources (space in main memory, peripheral storage devices and shared procedure or data bases). In each case questions of naming arise: objects of information must be named for reference by computations; decisions to share objects and procedures should not result in conflicts in the meanings of names. In this module, the student should learn how issues of naming objects arise, and he should learn the concepts and schemes through which the system objectives listed above can be achieved. The instructor should emphasize that there is as yet no single, generally accepted solution to the problem of meeting all four system objectives. Thus the instructor must concentrate on developing an appreciation of these issues, and the merits and limitations of known approaches to their resolution.

Basic Concepts

In a program, names are the symbols that specify the objects operated on by the program. In source language programs names are the identifiers of variables, structures, procedures and statements. When a program is compiled, most identifiers are replaced with numerical names (relative addresses) so that efficient accessing of instructions and data is possible: labels become addresses relative to the base of the machine code of the procedure, identifiers of local variables become addresses relative to the base of the procedure's activation record. Identifiers of external objects (e.g. other procedures and files) cannot be replaced by the compiler, and therefore must be retained in essentially unaltered form in the compiled procedure.

A compiled procedure is assigned a position in the address space of a computation (by a loader or a linking routine) so that it may be executed with other procedures during a computation. When this is done, symbolic references between procedures are usually replaced with names in the form of addresses that locate the procedure within the address space of the computation.

Context

By the nature of programmers and machines, the same name often will have two or more valid meanings. Some examples are: The same identifier may be used in distinct FORTRAN subroutines or ALGOL blocks; the address field of an instruction in the machine code for an ALGOL procedure must refer to different instances of variables for distinct activations of the procedure; machine language programs of different users may occupy the same memory locations at different times, the addresses of these locations having different meanings accordingly.

In each of these cases, the different meanings of a name are distinguished by additional information avilable to the compiler, loader, or hardware when the name is interpreted. This additional information is called the context in which the name is used. (Exercise: In each of the examples above, what are the contexts of the names?)

The context of a name need not be known at all stages of a name's use or transformation. For example, the compiler of a procedure cannot act on external names (those referencing other procedures, data structures, or files); it must leave such names in essentially the same form as they appeared in the source program. The context necessary for correct interpretation of these names is often not known until the procedure is assigned to the address space of a computation, or perhaps not even until execution is under way.

Distinct contexts for names used by different users are often provided by physically separating the information belonging to one from that belonging to another. This arrangement makes sharing of information (other than system information) difficult. If each user's information is catalogued in a "directory" but still is physically separated from other information, program modules can be shared only by the tedious and wasteful process of copying them from one directory to another.

A Fundamental Principle

In the discussions of dynamic structures and sharing of procedures to follow, there are illustrations of an important principle concerning the interpretation of names by a computer system:

> The meaning of a name must not change during any interval within which independent procedures may use the name to refer to the same object.

This means, for example, that the positions of objects within the address space of a computation cannot be changed if these objects are referred to by independently specified procedures. The difficulties in using overlay schemes [5, Dennis], [13, Lanzano], [15, Pankhurst] for handling allocation of main memory stem from violation of this principle: Because overlay schemes involve assigning two or more objects to overlapping areas in address space, the names (address) of such areas change in meaning during computation. To avoid chaos, each procedure making a change in the allocation of memory must inform all other procedures of the new arrangement of objects in address space. This requisite communication is, however, inconsistent with the objective that procedures be independently written.

File Systems

Computer systems generally provide for long term storage of information in the form of files. A file is an organized collection of data usually kept in peripheral storage devices such as magnetic drum or disk, or magnetic tape. The file management part of an operating system provides users means for generating and using files and manages the allocation of files to available space on storage units. Each user of the system is provided with a directory, in which his files are indexed or cataloged by names of his own choosing. In a number of systems, the objects indexed in a directory may include other directories as well as files, thus giving each user ability to create a directory tree for organizing his collection of filed procedures and data bases in a hierarchy. A particular object is specified by a sequence of names, a pathname, that selects a path from a root of the directory hierarchy to the desired object. The file system provides also for protection and controlled sharing of files. These general concepts of file system organization are discussed in [3, Daley and Neuman], [6, Dennis and Van Horn] and [1, Clark]. Since the hierarchy of directories defines a mapping from pathnames to objects, the file system may be regarded as defining an address space for files. In most computer systems the address space defined by the file system is logically distinct from the address space for the computational memory. Hence, procedures and portions of files must be copied between the computational address space and the file address space during the course of a computation, an object being directly accessible only if it is in the computational address space.

Files themselves may be structured in several ways:

1. As a string of bits, characters or words.
2. As a sequence of records.
3. As an indexed collection of records, each record having a unique key. The records may be accessed by key or in the sequence defined by a natural ordering of the keys.

Files structured as ordered sets of words are often managed as sequences of blocks of fixed size for convenient allocation to free storage space (note the analogy with the use of paging to implement a linear address space in main memory). The block structure of files is a matter of implementation and is made invisible to user computations. The idea of "record" is an historically important way of delimiting fragments of data, originating in the use of punched cards. Files of records are stored on contiguous areas of storage devices (disk and tape) and are most suitable for sequential processing as is common practice in business applications. The indexed sequential file is important in systems that access data bases in "real time." Hash coding techniques are used to locate the record for an arbitrary key without searching the file [1, Clark], [8, IBM]. In discussing file structure, the instructor should distinguish carefully between structure that is seen by the user (the abstract structure of the file) and structure for implementation purposes that is hidden (or should be hidden) from the user.

Files may be of arbitrary size within wide limits, and may grow or shrink during processing; thus a file system provides facilities for manipulating dynamic structures. Modular programming may be done using program modules that obtain inputs from files and store results in other files. File directories provide long term storage for procedures and data and may include protection and shared access control features. Thus a general purpose file system would seem to achieve all four system objectives stated earlier. Yet, there are serious limitations.

1. A data file (or the portion of it being processed) must be copied into the computational address space to gain the advantage of accessing it through the hardware addressing facilities of the computer. The implied loss of efficiency will be severe for compute-limited computations if files are used as the basic objects.

2. Procedures or program modules retrieved from the file system must be loaded into the computational address space prior to execution. In most systems all procedures must be loaded in advance of execution, wasting address space and making it impossible to activate procedures whose names are not known until retrieved from data bases or typed in from a terminal. If procedures may be loaded dynamically, the copying and linking steps will be time consuming.

3. There are few generally accepted standards for the structure and naming of files and for the primitive file operations implemented by file systems. Existing standards relate to COBOL, a language for data processing applications. Conventional file systems are not a suitable base for efficient implementation of procedures expressed in ALGOL, FORTRAN or PL/I.

4. Clashes of identifiers (file names) appearing in independently written procedures are not avoided. This matter is discussed further in a later paragraph.

Segmented Address Space

These four limitations result at least in part from the distinction between the computational memory and the file memory, and can be relieved by using the virtual memory concept to remove this distinction. This has been achieved by combining use of file directories with a large segmented address space. The address space is divided into a large number of segments, each being potentially large enough to hold any object (procedure or data file) indexed in the file directories. Reference by a computation to information in the address space is made by a pair of values (segment number, word number), segment numbers being assigned to procedures and data files when they first are referenced by a computation. After the first reference, the given procedure or data object is bound to a particular segment of address space and, thereafter the object may be referenced efficiently as a resident of address space rather than by a search of its pathname in the directory hierarchy.

Two methods are in use for implementing a segmented address space. In one [4, Denning], [9, Iliffe], [10, Iliffe and Jodeit], [16, Randell and Kuehner], the segment number is used as an index in a system-managed table of "descriptors" or "codewords." A descriptor (codeword) locates the origin of a segment within the main memory if space in main memory has been allocated for the segment, or in peripheral memory otherwise. The other method divides segments (linear name spaces in their own right) into pages and uses two levels of system tables to map (segment number, word number) pairs into main memory locations [4, Denning], [5, Dennis], [2, Daley and Dennis]. In both schemes the system tables also contain access control and protection tags.

The use of a segmented address space is valuable for providing independently written procedures space for large dynamic structures, and for permitting any object to be shared by computations if desired. These ideas are examined in the following paragraphs.

In current systems, the advantages of segmented address spaces have not compensated for the difficulty and complexity of their efficient implementation. For example, the mechanisms required for linking procedures together in the address space of a computation are intricate [2, Daley and Dennis] and should be considered an advanced topic.

Dynamic Structures

A dynamic data structure is an organized collection of information that changes in extent during a computation. Two types of such data structures are in common use: 1) variable-size tables, such as symbol tables, stacks, or matrices; and 2) linked-list structures [7, Foster]. Both types (or combinations thereof) require some mechanism for managing the address space in which they reside. With respect to the first type, there are two approaches, depending on the size of the address space and the nature of the mapping to memory locations if addresses are virtual. If the address space is sufficiently large, each structure may be assigned to a separate segment of address space large enough so the structure may grow and contract without conflicting with other structures. Since large parts of the address space will be unoccupied, this approach is of interest only when a virtual memory mechanism is present to map the occupied parts of address space into memory (e.g. a segmented address space).

If, on the other hand, a large address space is not available but the structures involved in a computation will fit into memory space, a system of routines may be provided for managing the assignment of parts of structures in the available space. Schemes for dynamically allocating contiguous blocks within a relatively small address space are described in [12, Knuth, pp 435-455], and are similar in function to the routines discussed below for managing linked structures.

A linked list structure is a collection of items, each consisting of a datum and a pointer (or pointers) to other items. The pointers are addresses that locate items within the address space [7, Foster]. A particular datum is accessed by following a chain of pointers from a single item that serves as the root of the structure. The system routines which access the structure on behalf of a program have three functions: 1) Free storage-management, i.e., handling allocation of new items, deletion of old items, and maintaining records of free space; 2) garbage collection, i.e., identifying items which have been deleted but not yet returned to the pool of free items; and 3) compaction, i.e., the relocation of live items in the address space so that the live items occupy a contiguous region [11, Jodeit], [12, Knuth, pp 435-455].

Compaction is used when the structures must be placed into as small a portion of address space as possible. Since the compaction process invalidates the addresses in the items until it is completed, no accesses can be permitted to the structure during compaction.

This is an important illustration of the general principle stated earlier. Compaction changes the names (addresses in this case) by which components of data structures may be referenced by computations. If two computations access the data structures concurrently, both must be halted during compaction and, moreover, any addresses (pointers to data structure items) held in private working storage of either computation must be corrected. This is why compaction is avoided in the design of storage allocation schemes for multiprogram operating systems. Compaction is often used to limit linked structures to a small contiguous portion of virtual address space to improve performance of single process computations on a paged computer.

Linked list structures are the standard representation of data in certain programming languages such as LISP; but they are often useful in providing efficient storage of complex structures for any application. For this reason, facilities for manipulating linked structures are provided in languages such as PL/I and ALGOL 68.

Modularity

The construction of a computer program can be greatly simplified if its major parts are already in the form of program modules that can be easily combined without knowledge of their internal operation. The student should learn the characteristics of computer hardware and software essential to modular programming, and understand how practical systems achieve or fail to achieve these necessary properties.

Two fundamental requirements for successful modular construction of programs are:

1. Program modules to be used together must employ consistent representations for all information exchanged among them.
2. A universal scheme must be established by convention for interfacing program modules with one another.

Modularity can be achieved for a particular group of system users if they adopt uniform conventions among themselves for data representation and inter-module communication. Such standards can be developed and agreed upon for any computer system, but seldom without some compromise between degree of generality and efficiency of program exectuion. Modularity achieved through use of shared files in the manner described earlier is an example. Different user groups, however, are likely to adopt conflicting conventions, and programs which do not honor such conventions are unusable as modules.

This discussion of the limitations of imposing conventions on existing systems should be followed by study of system characteristics that permit any program written for execution by the system to be used as a module in the construction of larger programs.

The requirement for consistent representation of intercommunicated data is met if all modules are expressed in the same source language, processed by the same compiler, and use only the data types provided by the language. Otherwise, this requirement cannot be satisfied without extreme care in the design and implementation of the language processors and execution environment [18, Wegner], [14, McCarthy et al].

The most important form of program module is the procedure. For modular programming, a procedure's author must be free to choose whatever names he desires for objects referenced by the procedure -- instructions, variables, data, structures, and other procedures -- without clashing with independent choices made by the authors of other procedures. To aid in understanding the solution of this and related naming problems, the notions of "argument structure" and "procedure structure" may be introduced.

The information which does not vary from one activation to another of a procedure P is called the <u>procedure structure</u> of P. It consists of

1. The code (machine language) of procedure P.
2. The procedure structures of other procedures that are used by P in the same way in every activation of P.
3. Any data structures (e.g., <u>own</u> data in ALGOL 60, or STATIC data in PL/1) that "belong" to the procedure in the sense that all activations of P refer to the same instances of the structure.

The parts of the procedure structure must be referenced directly by names appearing in the code of the procedure. These names are chosen by the author of the procedure and should be of no concern to the user of the procedure; the context in which these names are interpreted must therefore be distinct for each distinct procedure.

The information which may change from one activation to another consists of

1. The input data.
2. The output data.
3. The activation record (working storage).

The argument structure consists of the input and output data. Conceptually, the components (simple or compound) of the argument structure can be assumed numbered by distinct integers 1,2,3 .... In his coding of the procedure, the author may associate symbolic names $x_1, x_2, x_3, \ldots$ with these numbered components. Similarly, the user of a procedure may associate his own symbolic names $y_1, y_2, y_3, \ldots$ with these numbered components. Thus the ordering and structure of the components is fixed and part of the interface specification, but both author and user are free to choose names for them as they please.

The names used in a procedure to reference its activation record are local and the procedure's author must be free to choose them as he pleases. Since these names refer to different data in different activations, the context in which these names are interpreted must be different for each activation. Furthermore, the working storage must be able to expand to meet the storage requirements of the activation, which may be arbitrarily large.

If one procedure may be called from several independently written procedures, nonlocal references (as in ALGOL 60) have no meaningful interpretation. Also, external references (as in a FORTRAN implementation) only make sense in a global context, where name clashes are possible. Thus modular programming must be done without the aid of side effects -- all input and output data of a program module must be conveyed as components of the argument structure.

Implementations of ALGOL 60 provide distinct working storage areas for nested procedure activations and therefore handle recursive programs, the amount of working storage being specified upon procedure activation. Thus limitation 2 and, to some degree limitation 1, are overcome. In systems that offer ALGOL 68 or PL/1, means for representing and manipulating linked structures are provided, thereby removing limitation 3 for an important class of data structures. Yet clashes between names (of external procedures and files, for example) are still possible and limit the degree to which modular programming is possible. This problem is discussed in [6, Dennis and Van Horn, pp 151-154], [17, Vanderbilt, Chapters 2 and 3].

Clashes could be avoided by providing two contexts for the interpretation of "external" names occurring within procedures. Context for procedures and files that are part of the procedure structure would be provided by a procedure directory associated with the procedure in execution. Context for procedures and files named in the argument structure would be provided by an argument directory selected by the calling procedure. In this way, all names would be interpreted in appropriate contexts, and all possibilities of name clashes avoided. Although this idea is not implemented in any current system, some similar scheme will be necessary in future systems if modular programming is to be achieved.

With these concepts as background, the class can study the extent to which modular programming is permitted by various classes of systems. In a FORTRAN environment, each subprogram has a single, permanently-assigned, fixed activation record. Context for naming the parameters of a call is typically provided by the return address, a list of parameters being in some fixed location with respect to each point of call. Internal references within a subroutine are assigned meaning within the body of the subroutine only. Some limitations of this are:

1. Working storage is not expandable.
2. Recursion is not implemented.
3. There is no means for representing dynamic objects.
4. External references made by subprograms are interpreted within the (global) context of the loader's symbol table; thus a name clash will occur if two subprograms use the same name to refer to distinct objects.

Sharing

To permit sharing of procedures and data among users of a computer system, it is necessary to have a systemwide naming scheme so that one user can reference objects owned by others. One method for doing this uses a systemwide directory tree and allows directories to contain links to other directories [3, Daley and Neumann].

As an advanced topic, the instructor may discuss implementations in which a single copy of a procedure or data object in main memory is used by all computations sharing access to the object. There are three motivations for doing this: 1) conserve main memory, 2) avoid redundant copies of information, and 3) reduce overhead required to move extra copies in and out of main memory.

In most systems sharing of code in main memory is limited to supervisor and library programs; it is implemented by reserving for these shared objects a certain known portion of each user's address space.

If, on the other hand, every procedure in a computer system must be regarded as potentially shared among computations, the implications for addressing mechanisms of the system are far-reaching [5, Dennis]. Not only is it necessary to separate data and procedure information by means of "pure procedures," but a satisfactory means of transferring control between procedures is needed [2, Daley and Dennis].

If control transfer instructions contain absolute addresses in the address space of a computation, a shared procedure must be assigned to the same position in the address space of every computation that uses it. It follows that each address space must be sufficiently large to hold all potentially shareable procedures. Since there is no way for the system to know which procedures users may wish to share, it is attractive to implement a single address space for all computations in the system. No such system has been constructed, and it remains to see whether such an implementation would be practical.

Now, suppose control transfer instructions contain addresses relative to the base address of the procedure in which they appear. A procedure may now be assigned different positions in the address spaces of different computations, but the processor must contain a base register containing the base address of the procedure in the applicable address space; all control transfer instructions are interpreted relative to this base register. Provision must be made for reloading the base register whenever control is passed between procedures, and a scheme for properly implementing external references must be worked out. One complete scheme is described in [2, Daley and Dennis]. Because addresses have different meanings for different computations, this scheme has some serious disadvantages. Communication between computations is difficult, since directory names rather than addresses must be used to identify objects in messages. Also, once a procedure has been assigned a position in the address space of a computation, it is not legitimate for the system to delete it until the user can guarantee that no further attempt to access it by its assigned address will be made. As a result, occupancy of address space will tend to increase as a computation references new objects; thus a program that runs for an extended period, continually accessing new information, will have to manage its own use of address space.

Name Management -- Topic Outline

- Motivation
  - System objectives concerning storage and access to procedures and data bases.
  - Issues concerning treatment of names by system.
  - Objective to appreciate issues and understand merits of known techniques.

- Basic Concepts
  - Forms of names: identifiers, addresses; translation of names by compiler, loader, and system.
  - Context for interpretation of names; examples.
  - A fundamental principle: meaning of name must not change while in use by independent procedures.
  - Example: overlay schemes.

- File Systems
  - Files, directory hierarchies.
  - Structure of files, their representation on storage devices, implementation.
  - Achieving system objectives by use of files to represent data.
  - Limitation of file systems.

Segmented Address Space

- Segments, two-component addresses, binding procedures or data to address space.
- Implementations
  - with base registers
  - with paging

Dynamic Structures

- Two types: arrays of variable size; linked structures.
- Management of address space for dynamic structures: free space management, garbage collection.
- Compaction, a violation of the naming principle.
- Desirability of large segmented address space.

Modularity

- Fundamental requirements
  - consistent data representations
  - interfacing conventions
- Concepts for modular use of procedures
  - procedure structure, argument structure
- Limitation of systems for modular programming
  - FORTRAN
  - ALGOL 60
  - ALGOL 68 or PL/1

Sharing

- Use of links in directories for permitting controlled access.
- Motivations for shared use of information in main memory.
- Implementation alternatives
  - common address space for all computations
  - distinct address spaces with relative addressing; the problem of linking.

References

1. Clark, W. A. The functional structure of OS/360, Part III: Data Management. IBM Systems Journal, Vol. 5, No. 1 (1966), pp 30-51.

2. Daley, R. C., and Dennis, J. B. Vietual memory, processes, and sharing in MULTICS. Comm. of the ACM, Vol. 11, No. 5 (May 1968), pp 306-312.

3. Daley, R. C., and Neumann, P. G. A general-purpose file system for secondary storage. AFIPS Conference Proceedings, Vol. 27 (Fall 1965), pp 213-229.

4. Denning, P. J. Virtual memory. Computing Surveys (ACM), Vol. 2, No. 3 (September 1970), pp 153-189.

5. Dennis, J. B. Segmentation and the design of multiprogrammed computer systems. J. of the ACM, Vol. 12, No. 4 (October 1965), pp 589-602.

6. Dennis, J. B., and Van Horn, E. C. Programming semantics for multiprogrammed computations. Comm. of the ACM, Vol. 9, No. 3 (March 1966), pp 143-155.

7. Foster, J. M. List Processing. Macdonald and Co., London, 1967.

8. IBM. OS/360 Concepts and facilities. Programming Languages and Systems. S. Rosen, Ed., McGraw-Hill 1967, p 598.

9. Iliffe, J. K. Basic Machine Principles. American Elsevier. 1968.

10. Iliffe, J. K., and Jodeit, J. G. A dynamic storage allocation scheme. The Computer Journal (October 1962), pp 200-209.

11. Jodeit, J. G. Storage organization in programming systems. Comm. of the ACM, Vol. 11, No. 11 (November 1968), pp 741-746.

12. Knuth, D. E. The Art of Computer Programming (Vol. 1). Addison-Wesley 1968, Chapter 2.

13. Lanzano, B. C. Loader standardization for overlay programs. Comm. of the ACM, Vol. 12, No. 10 (October 1969), pp 541-550.

14. McCarthy, J., Corbato, F. J. and Daggett, M. M. The linking segment subprogram language and linking loader. Comm. of the ACM, Vol. 6, No. 7 (July 1963), pp 391-395. [Also appears in Programming Languages and Systems, S. Rosen, Ed., McGraw Hill 1967.]

15. Pankhurst, R. J. Program overlay techniques. <u>Comm. of the ACM, Vol</u>. 11, <u>No</u>. 2 (February 1968), pp 119-125.

16. Randell, B., and Kuehner, C. J. Dynamic storage allocation systems. <u>Comm. of the ACM, Vol</u>. 11, <u>No</u>. 5 (May 1968), pp 297-305.

17. Vanderbilt, D. H. <u>Controlled Information Sharing in a Computer Utility</u>. Report MAC-TR-67, Project MAC, M.I.T., Cambridge, Mass., October 1969.

18. Wegner, P. Communication between independently translated blocks. <u>Comm. of the ACM, Vol</u>. 5, <u>No</u>. 7 (July 1962), pp 376-381.