QUAESTIONES INFORMATICAE

Vol. 1 No. 2

September, 1979

Quaestiones Informaticae

An official publication of the Computer Society of South Africa
'n Amptelike tydskrif van die Rekenaarvereniging van Suid-Afrika

Editors: Dr. D. S. Henderson,
Vice Chancellor, Rhodes University, Grahamstown, 6140, South Africa.
Prof. M. H. Williams,
Department of Computer Science and Applied Maths,
Rhodes University, Grahamstown, 6140, South Africa.

Editorial Advisory Board

Subscriptions

Annual subscriptions are as follows:

	SA	US	UK
Individuals	R2	$3	£1.50
Institutions	R4	$6	£3.00

Quaestiones Informaticae is prepared for publication by SYSTEMS PUBLISHERS (PTY) LTD

for the Computer Society of South Africa.

The Management of Operating System State Data

Trevor Turton

IBM South Africa,
PO Box 1419, Johannesburg, South Africa

Abstract

A new approach is proposed for the management of operating system state data. Relational database concepts are suggested for the analysis and organization of this data, and a common central service routine is described which could manage all shared state data. This central routine is based on the techniques currently employed by database management systems.

1. State Data in Operating Systems

An operating system can be characterized as supporting a population of users who generate transactions requesting service. The operating system controls a set of resources such as processors and storage which it uses to ensure that all the transactions that it accepts are completed in a "reasonable" amount of time. This is achieved by a family of sequential processes which execute concurrently to provide the various services which the operating system offers. It is possible that two processes may access and modify the same data concurrently, leading to erroneous results. This must be prevented by the systematic application of a design philosophy in all operating system processes. The evaluation and selection of appropriate design philosophies has been a topic of great interest for many years, as shown in the literature [3,8,9,12,15,18].

In this paper we investigate the applicability of database management systems techniques to managing operating system state data. We restrict our attention to "message oriented" operating systems, as defined in [18]. In this type of operating system, the transactions accepted by the system are represented within the system as messages, which are passed from one process to the next, where they are queued while waiting for attention. Operating systems also contain tables or "control blocks" to describe the resources which they have at their disposal, such as processors, storage and I/O devices, and their status. These two classes of data together define the state of the operating system. Examples are:

(1) The users who are authorized to submit transactions.
(2) The transactions which have been accepted.
(3) The available computer resources, eg memory, CPU(s), I/O devices, timers.
(4) Queues of service request messages for each type of computer resource, generated by transactions in the course of their execution.

We will refer to both classes of data as control blocks (CBs). As well as describing items, CBs participate in relationships between themselves. These relationships are usually implemented by direct pointers imbedded within the CBs, in the same way that database record relationships may be implemented in network type database systems [2,4,6]. For example, an I/O request will usually point to the CB defining the job which requested the I/O operation, and to the CB defining the required I/O device.

An inspection of operating system code will reveal that most of the instructions exist to do nothing but create, delete, scan and update these CBs, and perform the consequent maintenance of the pointers which link them.

2. Control Block Management System (CBMS)

Control blocks in operating systems are usually fixed length fixed format records. They are susceptible to the same data analysis procedures that have been developed for application program data. In particular, Relational database analysis techniques can be applied to produce a rationalized set of CBs in Third Normal Form [5,7]. This procedure has proven very beneficial for the analysis and design of commercial DP systems, and there is every reason to expect similar benefits if the same approach is used in the construction of operating systems.

Since the CBs are in principle records with relationships it is possible to use conventional database management system (DBMS) concepts to manage them. An operating system architecture follows where all creation, deletion, searching and updating of CBs, and the relationships between them, is handled by one central service routine called the CB Management System, or **CBMS**

Certain concepts must be defined to explain how such a CBMS would work.

2.1 Control Block Descriptors (CBD)

For each distinct type and format of CB to be managed by the CBMS a formalized descriptor must be made available to it, which must describe the following:

(1) The name of the CB type (and hence CBD).
(2) The format of the CB.
(3) Optionally, range values for fields in the CB.
(4) The relationships the CB participates in, with the type of each CB related to indicated by its CBD name.
(5) The organization and sequencing to be used for CBs of this type.

2.2 CB Set Descriptor (CBSD)

When a new set of a particular CB type is needed (for example when a job is initiated, a set of CBs is needed to describe the programs that it will load), the CBMS must be requested to create

a CBSD to describe the set. It would contain:
(1) A pointer to the CBD from which it was generated.
(2) A definition of where this CB set is located in store, and the maximum number of CBs that this set may contain.
(3) The relationships that CBs in this set participate in, with pointers to the CBSDs describing the specific sets related to.
(4) Caller authorization required for CB access.

Every procedure executing in or under the operating system will have an authorization key associated with it by the operating system. When the CBMS is entered, this key will be available to it, and it will compare it to the one or more authorization codes in the CBSD. If the key matches a code, then the CBMS will allow the caller to do only those types of operations associated with that code in the CBSD (eg. create, delete, retrieve). The operations allowed may be limited to subsets of the CBs by Implicit Search Predicates in the CBSD (see Section 2.5.1).

2.3 Sequencing Strategies

The CBMS should offer various sequencing alternatives to match those commonly employed in current operating systems [16]:
(1) FIFO Queues, where new CBs are appended to the end of the set, and consumed from the beginning.
(2) Circular buffer queues, which is a variant of the preceding item, with a maximum set on the number of CBs in the set.
(3) LIFO Stacks, where new CBs are appended to the same end of the set that they are consumed from.
(4) Ordered Sets, where the CBs are maintained in ascending sequence of keys imbedded within them. For example, I/O requests queued on a particular disk may be ordered in the sequence of the cylinders that they require, to minimize access arm movement.
(5) Dequeues, a variant of the above item, where the CB key is binary and indicates either the start or end of the set.

The sequencing alternative required would be specified in the CBD and implemented by the CBMS. The callers of the CBMS need not be aware of the CB sequence chosen.

2.4 Organization Strategies

CBs are stored in a variety of ways. A common approach is to allocate them in arbitrary locations and then link them together with direct pointers.
Another is to provide space for a vector of CBs and to allocate slots in the vector on demand.

To meet the needs of operating systems and other potential users, a CBMS must support the above techniques, and preferably others described below. The technique most appropriate to each CB type can then be selected and specified in the CBD. The CBMS will then use the selected strategy to manage the CBs. Callers of the CBMS should not be aware of which technique is used, so that the technique may be changed with no impact on calling programs if changing CB usage patterns merit this.

2.4.1 Balanced Tree Strategy

When a particular CB type has a large number of occurrences and is very volatile, search time and storage management becomes expensive. A Balanced Tree technique [17] such as is implemented by IBM's VSAM access method [14] could be offered by CBMS for these cases. All CBs of this type would be kept in ascending key sequence in blocks allocated by the CBMS, which would also maintain an index to these blocks to speed access to them. New CBs would be slotted into the appropriate block in key sequence. Block overflows would be handled by spilling half of the contents of the overfull block into a newly allocated block, and updating the index.

This strategy would result in CBs moving around. Traditionally, a CB is identified by its storage address. With mobile CBs each CB would have to be assigned a unique identifier in place of a storage address. This identifier would be included in the CB and would provide the key of the CB in the index maintained by the CBMS. Alternatively, a unique field or subset of fields in the CB could provide the key.

2.4.2 Secondary Indexing of CBS

CBs are sometimes accessed by different field content by different callers. For example, Job Descriptor records in a Job Queue need to be accessed by job name, by originator, or by scheduling priority sequence at different times. Rather than scan potentially large numbers of CBs in no particular sequence, the CBMS could provide additional paths directly into the CBs by building and maintaining Secondary or Alternate Indexes to them. The instruction to the CBMS to build and maintain the Secondary Indexes would be contained in the CBD. Once again the existence of such Secondary Indexes, and the CBMS's use of them, would be invisible to all calling programs.

2.4.3 Use of DASD for CB Storage

CBs are often very numerous. Volumes frequently force the use of secondary storage for CBs. The CBMS should give some support to this mode of operation. One approach would be to let the operating system procedures move the blocks of CBs to and from secondary storage, and invoke the CBMS to handle blocks of CBs in main store only. A better approach would be to store all the CBs in virtual storage. The CBMS could then process all the CBs, since it would be able to access any of interest through the normal page fault mechanism. This approach would be viable for large volumes of CBs if a Balanced Tree organization were used.

2.5 Control Block Manipulation Language (CBML)

A well-defined interface to the CBMS must be provided for its users. This is the CBML. The user must pass the following information in the specified CBML format –
(1) The set of CBs to be operated on, by referencing its CBSD.
(2) The operation to be performed, eg:
  (a) insert a new CB in a set,
  (b) search for and retrieve selected CBs in a set,
  (c) replace an old CB in a set with an updated version,
  (d) consume a CB, ie retrieve and remove it from its set.
(3) For CB create or update, a copy of the new/changed CB.
(4) For CB search, a Search Predicate to specify the required CB.
(5) An indication of whether the caller wishes the CBMS to delay its execution if the request cannot be immediately met, and if so, how long a delay is acceptable.
(6) The authorization of the caller (this is passed implicitly by the supervisory software in control of the machine).

When the CBMS is invoked it must validate the parameters passed to it, check the caller's authorization to do what it requests, validate any CB supplied, then perform the required action. Any new or changed CBs will be copied by the CBMS into the storage area designated for that CB set in its CBSD. Any CBs retrieved for update will be copied into the caller's storage area. Links from the user supplied CB to other CBs will be validated, by ensuring

that the CBs linked to do exist and are of the correct type. If any CB inserts or deletes take place, the CBMS must reorganize the remaining CBs in the way which suits their organization (eg if the organization is linked list, by relinking the CB's siblings).
On some occasions it may not be possible to satisfy a caller's request immediately. For example, a process may request a CB from a set which is temporarily empty. The caller must specify in a CBML parameter either that it wishes to be delayed until a suitable CB becomes available (and if so, how long it is prepared to wait), or else that it wishes the CBMS to return control to it immediately. In the latter case, the caller must indicate whether it wants its request to remain outstanding, or to be cancelled immediately. This facility would support for example the requirements of reader/writer processes sharing a circular buffer set, as described in [3,12].

2.5.1 CBML Search Predicates and Implicit Predicates

The caller specifies to the CBMS which CBs it wants returned in a Search Predicate. This is a list of one or more Search Arguments, linked with boolean operators. Each Search Argument specifies a field in the CB and a value to compare it with, plus an indication of what comparison is to be used – greater than, less than, equal to etc. Conceptually the CBMS scans each CB of the set selected, and compares the selected fields with the supplied values, performs the boolean operations specified, and returns only those CBs which meet the Search Predicate. In practice, the CBMS may use its knowledge of how the CBs are organized to limit the number of CBs it must scan. For example, if the CBs are maintained in key sequence by the CBMS and the caller's Search Predicate specifies a key value, then the CBMS will use its index to access the CB directly. Search Predicates are common to several database management systems [1,13].
In addition to the Search Predicate passed by the caller, CBMS may impose further limits on the searches it performs for the caller. A given caller may be authorized to see only a subset of a particular CB set. This would be indicated in the CBSD. Where the authorization for CB search is defined, it would include an Implicit Search Predicate. The CBMS performs this Search on the CBs in parallel with the others specified by the caller, and will return to it only those CBs satisfying both criteria. The Implicit Search Predicate may specify field comparison values either explicitly, or by reference to a field in a CB associated with the caller. For example, a user asking for a list of the jobs submitted to the operating system may be restricted to seeing only those that it submitted.

2.6 Ensuring Control Block Integrity

With the proposed approach all system state data which may be accessed by multiple concurrent processes would be controlled by the CBMS. Hence only the CBMS would need integrity mechanisms to ensure that updates to state data were serialized, and these could be implemented using semaphores as described in [8]. Semaphores could be introduced at different levels for each CB type, depending on the amount of contention anticipated. The CBSD should contain a code instructing the CBMS which level to apply for each set of CBs.

(1) For low contention CBs, a single global semaphore for each CB type would suffice.
(2) For moderate contention CBs, a semaphore could be located in each CBSD to control access to each set of CBs independently.
(3) For high contention CBs it may be necessary to associate a semaphore with each individual CB or, with a balanced tree organization, each block of CBs.

The integrity mechanism should allow multiple concurrent accesses providing they are read-only. Once a CB has been accessed for the purpose of update or delete, it must remain inaccessible to all other processes until that update or delete completes.

2.7 Use of the CBMS by other Systems

Most operating systems support important subsystems which also manipulate CBs extensively, for example subsystems which manage databases and teleprocessing networks. If the operating system had a CBMS for its own use, it would also be of great value to these subsystems. If the CBMS were implemented efficiently in hardware, performance improvements could be realized as well. Furthermore, in some database systems the database records on DASD are stored as "flat files", which means as an array of fixed format fixed length records. In this case the CBMS should be implemented so that it can manage the database records as well as the CBs.

2.8 CBMS Performance Considerations

As mentioned before, the majority of instructions executed by an operating system pertain to CB management. If all CB manipulation were to be done by a centralized CBMS, it would be very heavily exercised. If this CBMS were implemented in software, a net performance degradation would be inevitable, since overheads would be introduced just in calling it. Furthermore, the CBMS would have to interpret the CBML parameters passed to it, and the associated CBSD and CBD parameters, before it could carry out the required CB manipulations. When compared to the purpose written direct code which operating systems currently contain to do the job, the CBMS approach would seem to entail an unacceptable amount of overhead.
However the CBMS could be implemented in microcode. The complexity of the proposed CBMS (and hence of the microprogram required to support it) is considerably less than that of the microprogram which was written to implement an APL compiler and interpreter for the IBM System/370 Model 145 [10,11]. Performance tests on this system have shown that the microcoded system performs the interpretation of its parameters about ten times faster than an equivalent software interpretive system, and generally performs the subsequent instruction execution faster. This microcoded interpretive APL system has been found to execute programs at about the same speed as their equivalents in direct machine code, and to outperform the machine code equivalents when iterative operations over large vectors of operands are involved. CB search operations often have to scan large numbers of candidates.
While the CBMS functions described in this paper would not be extraordinarily difficult to microprogram, CB field validation could in principle be arbitrarily complex, so a break-out from microcode into machine language field validation subroutines would be a prudent provision. The implementation in [10,11] has this facility.

2.9 Automatic Backout of CB Updates by Failing Programs

Given the above CBMS approach, it would be possible to apply another Data Base Management concept, namely that of

automatic backout of updates applied by failing processes. Any real-world operating system must change with time, and will have errors introduced. These systems must have some mechanism for dealing with the resulting failures. For example, when IBM rewrote OS/360 MVT as MVS, a major investment was made in having each functional module leave "footprints" so that it could attempt functional recovery when failures occurred. This recovery technique requires an application-specific trap routine for each module.

An interesting comparison can be drawn with the way some DBMS's handle the same problem [1,13]. Application programs may fail after applying only some of their intended data base updates, leaving the overall data base in an invalid state. To correct this, the DBMS keeps track of all the updates applied by the program, and automatically backs out all those applied if the program does not complete successfully. If the program is processing one simple transaction then the number of pending updates which the DBMS must save is not large. For longer running programs, the total number of pending updates would rapidly become too large to store. This problem is avoided by having the application programs issue synchronization calls whenever they reach a "point of commitment" (that is to say the data base integrity is preserved if all their pending updates are applied at that point).

Exactly the same principle could be adopted by the CBMS. The supervisory routines using the CBMS could indicate whenever they reach a "point of commitment", for example after allocating all the resources needed by a job prior to starting its execution. Then should any supervisor routine fail, the CBMS could reverse out all the updates it had applied to the supervisor state data (control blocks) since its previous point of commitment. This would leave the overall system state in an acceptable condition, and clearly result in a robust operating system. Functional recovery routines could be coded to operate at a very much higher level.

The same principle could be used to resolve deadlock or "Deadly Embrace" [8]. In the data base management systems [1,13] when a deadlock situation arises, one of the deadlocked programs is terminated, its pending updates reversed out, and the records that it is holding are released. This process is repeated as needed till the deadlock is resolved. Freeing the operating system architect of the fear of deadlock would make the design process a lot easier than it now is.

3. Benefits of the CBMS System

(1) CB integrity and hence system reliability could be greatly improved.

(2) Since most operating system (and much subsystem) code exists only to manipulate CBs, a CBMS facility would greatly reduce the amount of this code, and hence the system maintenance burden.

(3) The areas of any operating system which are most complex, error prone and difficult to debug are those which deal with process synchronization. All these complexities could be contained in the CBMS, which would be relatively small and could be exhaustively tested independently of the operating system procedures which it must support. This approach conforms well with Dijkstra's technique of constructing operating systems in hierarchical layers of function [9].

(4) The use of parallel processors to accelerate database searches has been extensively investigated [20,19]. Vehicles such as head-per-track disks and charge coupled devices have been explored for parallel data searches. An operating system using a CBMS based on DBMS principles could obtain the advantages offered by this parallelism, without any explicit coding.

References

[1] ASTRAHAN, M.M. et al. (1976). System R: Relational Approach to Database Management, *ACM Transactions on Database Systems,* **1,** 2.

[2] BACHMAN, C.W. and WILLIAMS, S.B. (1964). A General Purpose Programming System for Random Access Memories, *Proc. FJCC,* AFIPS Press.

[3] BRINCH HANSEN, P. (1973). *Operating System Principles,* Prentice Hall, Englewood Cliffs, N.J.

[4] CINCOM SYSTEMS INC. (1973) *TOTAL Users' Manual,* Order No AB65.

[5] CODD, E.F. (1970). A Relational Model of Data for Large Shared Data Banks, *Comm. ACM,* **13,** 6.

[6] Data Base Task Group of CODASYL Programming Language Committee. (1971). Report.

[7] DATE, C.J. (1977). *An Introduction to Database Systems,* Addison-Wesley, Reading, Mass.

[8] DIJKSTRA, E.W. (1965). *Cooperating Sequential Processes,* Technological University, Eindhoven, The Netherlands.

[9] DIJKSTRA, E.W. (1968). The Structure of the THE Multiprogramming System, *Comm. ACM,* **11,** 5.

[10] HASSIT, A. and LYON, L.E. (1976). An APL Emulator on System/370, *IBM Systems Journal,* **15,** 4.

[11] HASSIT, A. and LYON, L.E. (1975). The APL Assist, *IBM Technical Report No ZZ20-6428.*

[12] HOARE, C.A.R. (1974). Monitors: an Operating System Structuring Concept, *Comm, ACM,* **17,** 10.

[13] IBM Corp. (1975). *IMS/VS General Information Manual,* Form No GH20-1260.

[14] IBM CORP. (1973). *Virtual Storage Access Method (VSAM) Programmer's Guide,* Form No GC26-3838.

[15] KEEDY, J.L. (1978). On Structuring Operating Systems with Monitors, *Australian Computer Journal,* **10,** 1, reprinted in *Operating Systems Review,* **13,** 2 (1979).

[16] KNUTH, D.E. (1968). *The Art of Computer Programming,* **1,** Addison-Wesley, Reading, Mass., p234.

[17] KNUTH, D.E. (1973). *The Art of Computer Programming,* **3,** Addison-Wesley, Reading, Mass., p473.

[18] LAUER, H.C. and NEEDHAM, R.M. (1978). On the Duality of Operating System Structures, *Proc. Second International Symposium on Operating Systems,* IRIA, reprinted in *Operating Systems Review,* **13,** 2. (1979).

[19] LIN, C.S., et al. (1976). The Design of a Rotating Associative Memory for Relational Database Applications, *ACM Transactions on Database Systems,* **1,** 1.

[20] SU, S.Y.W. and LIPOVSKY, G.J. (1975). CASSM: A Cellular System for Very Large Databases, *Proc. Int. Conf. on Very Large Databases.* Framingham, Mass. pp456-472.

Notes for Contributors

The purpose of this Journal will be to publish original papers in any field of computing. Papers submitted may be research articles, review articles, exploratory articles or articles of general interest to readers of the Journal. The preferred languages of the Journal will be the congress languages of IFIP although papers in other languages will not be precluded.

Manuscripts should be in double-spaced typing on one side only of A 4 paper and submitted to Dr. D. S. Henderson or Prof. M. H. Williams at

Rhodes University
Grahamstown 6140
South Africa

Form of manuscript

Manuscripts should be in double-space typing on one side only of sheets of A4 size with wide margins. The original ribbon copy of the typed manuscript should be submitted. Authors should write concisely.

The first page should include the article title (which should be brief), the author's name, and the affiliation and address. Each paper must be accompanied by a summary of less than 200 words which will be printed immediately below the title at the beginning of the paper, together with an appropriate key word list and a list of relevant Computing Review categories.

Tables and figures

Illustrations and tables should not be included in the text, although the author should indicate the desired location of each in the printed text. Tables should be typed on separate sheets and should be numbered consecutively and titled.

Illustrations should also be supplied on separate sheets, and each should be clearly identified on the back in pencil with the Author's name and figure number. Original line drawings (not photoprints) should be submitted and should include all relevant details. Drawings, etc., should be about twice the final size required and lettering must be clear and "open" and sufficiently large to permit the necessary reduction of size in block-making.

Where photographs are submitted, glossy bromide prints are required. If words or numbers are to appear on a photograph, two prints should be sent, the lettering being clearly indicated on one print only. Computer programs or output should be given on clear original printouts and preferably not on lined paper so that they can be reproduced photographically.

Figure legends should be typed on a separate sheet and placed at the end of the manuscript.

Symbols

Mathematical and other symbols may be either handwritten or typewritten. Greek letters and unusual symbols should be identified in the margin. Distinction should be made between capital and lower case letters between the letter O and zero; between the letter l, the number one and prime; between K and kappa.

References

References should be listed at the end of the manuscript in alphabetical order of author's name, and cited in the text by number in square brackets. Journal references should be arranged thus:

1. ASHCROFT, E. and MANNA, Z. (1972). The Translation of 'GOTO' Programs to 'WHILE' Programs, in *Proceedings of IFIP Congress 71,* North-Holland, Amsterdam, 250-255.
2. BÖHM, C. and JACOPINI, G. (1966). Flow Diagrams, Turing Machines and Languages with only Two Formation Rules, *Comm. ACM,* **9,** 366-371.
3. GINSBURG, S. (1966). *Mathematical Theory of Context-free Languages,* McGraw Hill, New York.

Proofs and reprints

Galley proofs will be sent to the author to ensure that the papers have been correctly set up in type and not for the addition of new material or amendment of texts. Excessive alterations may have to be disallowed or the cost charged against the author. Corrected galley proofs, together with the original typescript, must be returned to the editor within three days to minimize the risk of the author's contribution having to be held over to a later issue.

Fifty reprints of each article will be supplied free of charge. Additional copies may be purchased on a reprint order form which will accompany the proofs.

Only original papers will be accepted, and copyright in published papers will be vested in the publisher.

Letters

A section of "Letters to the Editor" (each limited to about 500 words) will provide a forum for discussion of recent problems.

Hierdie notas is ook in Afrikaans verkrygbaar.

Questiones Informaticae

Partial proceedings of the first South African Computer Symposium on Research in Theory, Software, Hardware, organised by The Research Symposium Organising Committee of The Computer Society of South Africa. 4 & 5 September 1979, Pretoria.

Contents/Inhoud