Data Management Facilities of an Operating System Kernel

Hans Diel Gerald Kreissig Norbert Lenz
Michael Scheible Bernd Schoener

IBM EF Laboratory
Schoenaicher Strasse 220
7030 Boeblingen
West Germany

ABSTRACT

The paper describes the part of a general operating system Kernel supporting data management functions. The operating system Kernel can be imbedded into microcode and viewed as an extended hardware interface.
Four Kernel instructions are defined to support data management. They provide a powerful basis for the implementation of different kinds of access methods and file systems, including data base systems. Advanced transaction processing concepts such as concurrency control, support of backout, commit and a variety of share options are included.

OVERVIEW

An operating system Kernel contains the basic functions which are inherent to a class of operating systems. Implemented in microcode it extends the hardware interface to a more powerful function set. The goal of the Kernel concept is to provide the necessary operations of modern operating systems without limiting the applications or setting specific constrains on them.
The Kernel consists of a process management, a storage management and a data management part.

The present paper describes a so-called data management Kernel of an operating system.
With respect to the design of the data management Kernel it has been attempted to achieve generality and flexibility within two directions:

1. Support of different file types

   The functions have to be suitable for the support of different kinds of files and access methods (e.g. sequential, direct, keyed) as well as for the support of data base systems.

2. Support of different operating systems

   The project which led to the definition of the data management Kernel, described below had the general objective that the Kernel should be suitable for the support of multiple operating system types. The data management Kernel described in this paper provides this flexibility. It has been demonstrated that it is suitable for an efficient implementation of the UNIX(*) [UNIX 78] file management as well as for a large subset of the IBM OS/VS2 [MVS 76] access methods. Support of further types of operating systems access methods has been considered without detecting any significant problems.

The desired flexibility has been achieved by leaving out of the data management Kernel any knowledge about the internal structure of files. The data management Kernel does not support (nor does it depend on) any knowledge about the internal structure of files, but it knows only objects being collections of bits or bytes. For most types of access methods this still provides a very powerful basis to build on, for some data management systems (e.g. the UNIX file management system) it is just what is needed. The designers of the data management Kernel came even to the conclusion that the described functions represent already a suitable (substitute for a) data management system (with certain advantages over traditional file management systems) if they are embedded into a higher level language.

As a consequence of this goal to achieve the independence from the internal structure of the objects the Kernel deals with a new term "Persistent Object" which is introduced to denote the objects on which the data management Kernel operates.

A **Persistent Object (PO)** is a named collection of bytes under the control of the data management Kernel. The Kernel ensures that the persistent object stays in existence when the operating system is shut down and it provides services to create, manipulate and delete a persistent object.

In order for a process to access a persistent object, it has to be made accessible by the "Access Persistent Object" instruction. Once a persistent object is accessible by a process (as a whole or partly) it is mapped to its virtual memory - which means it has a starting address and a length associated.

For the communication with the data management Kernel, a persistent object is uniquely identified by a catalog entry. This implies the data management Kernel uses a catalog entry to keep all its knowledge about a persistent object. A reference to a catalog entry is requested in order to specify a reference to a certain persistent object.

Obviously persistent objects are primarily intended to represent things like files and data sets, including data bases. However, also **further kinds of objects** required by an operating system having a need for a longer lifetime can elegantly be represented by persistent objects. One class of such objects would be the objects supporting highly available operating systems, i.e. the information required to restart and to recover an operating system after a crash. Another example for a persistent object are the catalogs, i.e. the collections of catalog entries needed to describe and identify persistent objects.

(*) UNIX is a trademark of Bell Laboratories

In order to support the implementation of data base systems on top (i.e. by use) of the data management Kernel it was felt to be necessary to support already within the data management Kernel certain concepts usually associated with transaction processing [GRAY 80]. Examples for such concepts are sharing and locking, concurrency control, and backout.

Support of these concepts within the data management Kernel not only allows a very efficient realization of these concepts, but in addition it offers the advantage that these concepts apply to all kinds of persistent objects of the operating system (not only to the data base). This is believed to be prerequisite for a reliable operating system Kernel.

There are some relations between the data management Kernel and other areas of the operating system which will be considered next:

As described above, a persistent object is uniquely identified by a catalog entry. Thus, the contents of a catalog entry is defined by the data management Kernel. In order to keep the data management Kernel sufficiently small and compact, it was however decided not to include catalog management functions into the data management Kernel. Thus, the data management Kernel defines how a catalog entry looks like (at least part of it) but it does not define how catalog entries are grouped together to form catalogs, nor does it define the rules for associating names to catalog entries.

The way how persistent objects are stored on non-volatile (i.e. persistent) storage is completely up to the data management Kernel. The services described below to create a persistent object only allow the specification of specific disk volumes as the residence for a specific persistent object.

Persistent objects are accessible by processes after they have been (partly or completely) mapped to virtual memory. For this and for other reasons it turned out that the efficient implementation of the proposed data management Kernel depends on a very close cooperation between the data management Kernel and the page management part of the operating system Kernel. This has no influence on the external functions provided by the data management Kernel, except for certain aspects concerning the granularity for accessing parts of a persistent object (e.g. with respect to locking).

The next part of this section is a brief description of the transaction processing principles and defines the terms used throughout the paper. Section 2 describes the concepts of the data management Kernel in terms of basic concepts, Kernel instructions and objects, access and concurrency control, windowing and deadlock detection.

Transaction Processing Principles

The principles of transaction processing based on the **transaction model** [GRAY 78] are originally dedicated for databases and transaction processing subsystems. A transaction is treated as an atomic unit of work and recovery. It is allowed to **commit** if it transforms persistent objects from one consistent state into another consistent state. If a transaction does not produce a consistent state or if it is aborted the system must guarantee that the state transition is completely **undone**.

In order to manage the concurrent execution of multiple transactions a **locking mechanism** has to be provided. The data to be accessed have to be locked according to the following lock protocol:

- a READ lock is requested for data to be read
- a WRITE lock is requested for data to be written
- a READ lock is granted if the data is not locked or read locked
- a WRITE lock is granted only if the data is not locked

The lock protocol guarantees consistent and legal schedules and prevents concurrency anomalies such as lost updates and dirty or unrepeatable reads.

The consistency lock protocol requires transactions to be **well-formed**, i.e. to lock the data before accessing them, and **two-phase**, i.e. not to lock any data after the first unlock.

Obeying the consistency lock protocol and holding all locks until transaction end results in the highest degree of consistency, degree 3 consistency [GRAY 76], and simplifies recovery mechanisms.

THE DATA MANAGEMENT KERNEL

Basic Concepts

The data management part of the Kernel supports the transaction processing concepts described in the previous section. The data management facilities can be invoked by a set of Kernel instructions operating the data management objects with sharing and locking capabilities. Inclusion of these facilities into the general operating system Kernel provides transaction processing characteristics to all kinds of data management functions, e. g. also for traditional access methods [SPECTOR 83].
This concept centralizes the transaction processing principles at a very low level of the system. It provides its capabilities to all applications, like access methods, data base systems, etc. and avoids its implementation in several systems running on top of the Kernel.

The Kernel instructions support the access and manipulation of persistent objects and guarantee their consistency. For the user of the Kernel instructions, persistent objects or parts of them are available in virtual memory for the duration of his processing. This means that he can access them in terms of normal instructions via address and offset. Possible internal structures of persistent objects, like records or indices are not seen by the Kernel, nor is there a need for functions like buffer management etc. required in traditional data management systems.

This concept can be considered as a realization of a so called **Single Level Storage**. It is implemented by a close cooperation of the data management Kernel and the paging/memory management part of the operating system Kernel.

While the user is working on a persistent object, he has the impression of dealing with a private copy of it. He can commit the private copy at the end of his processing.

Before the user commits his private copy of the persistent object, it is guaranteed that any abnormal termination of his transaction (even due to system crashes) will cause the restoration of the old consistent status of the persistent object. This feature is sometimes referred to as the **Safe File System** or automatic transaction backout of the operating system Kernel.

The safe file system uses the concepts of shadow paging based on the single level storage. Updates on persistent objects are made only in virtual memory. The consistent (old) version of an object remains unchanged. At commit

time the new version from temporary storage replaces the old version and adjusts the catalog entry.
Transaction backout does not require to remove the updates of the transaction on the persistent objects because they have not been propagated.

The recovery strategy of the Kernel can be classified as transaction oriented checkpointing combined with the propagation of updates at checkpoints [HAERDER 79]. The only point during the execution of a transaction at which its updates are propagated is the commit point.

The Kernel Instructions

APO, ACCESS PERSISTENT OBJECT

The APO instruction provides access to persistent object. A parameter specifies the area to be accessed, i.e. it is possible to access the whole object or part(s) of the object. The instruction combines three functions:

- object allocation
- open the object
- map object into virtual memory

The first APO of a transaction is equivalent to a 'begin transaction'. APO provides only access to a persistent object if the request is possible according to the access control protocol.
The Kernel manages the so-called 'accessed persistent object table' describing all presently accessible objects. An APO creates an new entry for that table, its index is returned. A successful invocation of APO returns the virtual memory address and the length of the accessible area.

RELPO, RELEASE PERSISTENT OBJECT

The RELPO function releases the accessible area or object obtained via APO from virtual memory. This instruction does neither an implicit commit nor backout. The released areas may subsequently be reaccessed by a new APO instruction without any influence on the consistency.
This function is required for systems with virtual memory constraints where the size of the persistent object exceeds the size of the virtual memory.

CPO, COMMIT PERSISTENT OBJECT

CPO provides the capability to commit or backout updates on a set of persistent objects. The CPO function commits by updating tables of single level storage and backouts by just releasing the virtual memory parts of the persistent objects. At the end of commit all locks held on the object are released, the virtual memory obtained by the object is freed. CPO uses the two-phase commit protocol [GRAY 78] to ensure its atomicity.

This function provides the commit at end of transaction as it is used in the traditional transaction processing environment. Additionally it enables the user of CPO to commit updates on persistent objects independent from end of transaction. It deviates from traditional definitions of the term transaction. The function can be used by applications with very long running transactions like graphics and text processing.

The extended transaction definition supports a unit of work as the interval between two commits or between APO and CPO. The responsibility and flexibility to issue CPO prior to the end of transaction is left to the application running on top of the Kernel.

LOCKPO, LOCK PERSISTENT OBJECT

The LOCKPO instruction provides the capability to lock individual areas of a persistent object that is accessed via APO. LOCKPO is only valid for one of the share options supported by the Kernel which allows multiple concurrent access to an object. The smallest physical locking granularity in which a LOCKPO is mapped is one page.
A parameter of LOCKPO defines whether it is a READ lock or a WRITE lock. A READ lock will be granted if the area is not locked or READ locked, a WRITE lock may only be granted if the area is not locked.

Locks can only be released during the CPO operation, there is no explicit UNLOCK instruction. Locks will implicitly by released at system restart and abnormal transaction termination.

The Data Management Objects

The data management Kernel instructions operate on several objects. They can be seen as **abstract data types** together with the pertinent operations. This technique ensures the definition of clean interfaces and a proper system structure.

Catalog Entry

A catalog entry for a persistent object contains all necessary information that is required by the data management Kernel instructions and the layer above the Kernel (e.g. access methods). The catalog entry is the only object which is known outside the Kernel.

Disk Layout Table

The disk layout table is a set of entries which relate to disk pages. An entry describes whether the page is free or part of a virtual memory or obtained by a persistent object.

Access Persistent Object Table

The access persistent object table is a list of entries describing the persistent objects which are presently accessed. Among other things it contains a pointer to the catalog entry and the actual share option for the accessed object.

Page Management Table

The page management table is a list of entries describing the pages of a virtual memory.

Persistent Object Page Table

The persistent object page table is a list of entries describing the pages of an opened persistent object. The table is allocated at the time of the first APO to the object. The table is used to support the LOCKPO instruction.

Sharing and Locking, Concurrency Control

OVERVIEW

The Kernel locking facility provides a transaction-oriented locking mechanism dedicated to the concurrent access of multiple transactions to persistent objects.

In order to make transactions obey the consistency lock protocol and thus to achieve the consistency of any legal schedule of those transactions, the layer above the Kernel has to

- Access the persistent objects with appropriate share options via the APO instruction
- Lock the data which will be modified
- Lock the data which is read
- Hold the locks until transaction end

The two-phase lock protocol described previously is automatically obeyed, because APO, LOCKPO, and CPO are the only Kernel instructions that request and release locks, and all unlocking is deferred until CPO, as recommended in [GRAY 80].

Access to persistent objects is controlled by the data management Kernel in a two-way procedure, the **access admission control** and the **concurrency control**.
The access admission control checks the allowed and the requested share option parameters according to the share option protocol.
The second step supervises multiple concurrent access to a persistent object by checking the actual share options and the access types according to the locking protocol.

ACCESS CONTROL, LOCKS

Access to persistent objects is controlled by the Kernel via share options (locks). The allowed share option for an object is specified at creation time and is checked against the used share option specified at APO time. The share option protocol of Figure 1 on page 8 controls whether access is permitted or not.

The data management Kernel provides two locking facilities

1. lock and control access to a complete persistent object
2. lock and control access to parts of a persistent object

The lock option for a complete persistent object (1) will be specified and checked as the actual share option parameter in APO.
The second option to lock a part of a persistent object is provided by the LOCKPO instruction and is applicable with one specific share option only. The system provides 5 different share options:

- **LEX** - lock exclusive for READ and UPDATE
- **LR** - lock for multiple READ only; this lock may not be combined with the access type WRITE
- **LRSU**- lock for multiple READ and single UPDATE
- **LMA** - lock for multiple access multiple READ and multiple WRITE can be done concurrently on one persistent object by multiple transactions. This lock can only be applied in conjunction with the LOCKPO function for portions of an object. The same portion of an object can only be updated by a single transaction.
- **URA** - unrestricted access; multiple READ and WRITE can be made concurrently uncontrolled by the system. (UNIX-like)

Among the five share options described below the LMA (Lock Multiple Access) is primarily intended for support of transaction processing. However, also the share options LEX (Lock Exclusive) and LR (Lock Read) ensure the consistency of data. Share options LRSU (Lock Read Single Update) and URA (Unrestricted Access) are provided for support of traditional non-transaction processing like file systems and may result in non-consistent updates.

In case of the LMA share option, the areas being updated have to be locked by use of the LOCKPO instruction. The LR and LEX share options result in a locking of the complete persistent object object at APO time. The authors' implementation of the Kernel protects the persistent objects from being updated, if no locks are set.

The locking of data to be read is supported by the LEX and LR share options and by the LMA share option together with the LOCKPO instruction. Unlike write locks, read locks are not enforced in the authors' implementation of the data management Kernel.

The holding of locks until end of transaction is ensured by supporting the unlock function as part of the CPO instruction only.

ACCESS ADMISSION CONTROL PROTOCOL

The access admission control step checks whether the allowed share option in the catalog entry and the requested share option of an APO correspond according to the share option protocol and indicates if access could be permitted or not. The access request to an object is checked within the "allocate" function of APO.
The relationship between the allowed share option of a persistent object and the requested share option is illustrated by the following protocol. The protocol defines whether the requested share option agrees with the allowed share option.

allow. share option \ request. share option	LEX	LR	LRSU	LMA	URA
LEX	Y	N	N	N	N
LR	Y	Y	N	N	N
LRSU	Y	Y	Y	N	N
LMA	Y	Y	Y	Y	N
URA	Y	Y	Y	Y	Y
-	Y	Y	Y	Y	Y

Figure 1. Share option protocol: Relationship between allowed and requested share option

CONCURRENCY CONTROL

The concurrency control allows multiple concurrent access to a persistent object without violating data consistency. This control step uses the locking protocol which depends on the actual share options and the access types which are requested by a task.
The requested share option and access type of a task will be checked against the current (actual) share option and access type. The protocol defines whether the access can be permitted or not.

1./2.	LEX	LR	LRSU/r	LRSU/w	LMA/r	LMA/w	URA/r	URA/w
0	Y	Y	Y	Y	Y	Y	Y	Y
LEX	N	N	N	N	N	N	N	N
LR	N	Y	Y	N	Y	N	Y	N
LRSU/r	N	N	Y	Y	Y	Y	Y	Y
LRSU/w	N	N	Y	N	Y	N	Y	N
LMA	N	N	N	N	Y	Y	Y	N
URA	N	N	N	N	N	N	Y	Y

Figure 2. Locking protocol: Relationship between current (actual) share option and access type in the catalog entry (1.) and the requested ones (2.)

Windowing

DESCRIPTION

The windowing concept for persistent objects allows access not only to persistent objects as a whole, but also to subsections of persistent objects. It is offered in connection with the APO, RELPO and CPO Kernel instructions.

An **accessible area** (also called a "window") is a contiguous subsection of a persistent object which resides contiguously in virtual memory. It is said to be **connected** to this virtual memory. Each accessible area is uniquely defined by an offset within the persistent object to which it belongs, a length, and an address within the virtual memory where it resides. Accessible areas can be accessed, i.e. connected to virtual memory via the APO instruction, and can be released, i.e. disconnected from virtual memory via the RELPO instruction. Furthermore, the CPO instruction causes an implicit RELPO for all still accessible areas of the specified persistent object.

Different accessible areas which belong to the same persistent object and which are connected concurrently to the same virtual memory may not overlap in the persistent object (nor may they overlap in virtual memory, but this will automatically be achieved by APO).
However, different accessible areas of the same persistent object connected to different virtual memories concurrently may overlap in the persistent object, if the share option and access type which must be specified for APO do not forbid this. Possible anomalies due to concurrency can only be prevented by use of the appropriate share option and access type in this case.

MOTIVATION

The windowing concept was introduced for several reasons:

- In systems providing only a very limited virtual address space, or in environments where large or many persistent objects have to be processed simultaneously, the user should not be restricted by the size of the virtual address space. The windowing concept allows to process parts of large persistent objects and thus to use the virtual address space economically.

- The windowing concept provides an easy way to extend and shorten a persistent object: extending is achieved by specifying an accessible area which exceeds the current end of a persistent object, shortening is achieved by use of the length parameter of CPO.

- In distributed systems, the accessible area can be used as the unit for accessing persistent objects residing on remote sites rather than accessing whole persistent objects (primarily for performance reasons).

Deadlock Detection

The deadlock detection mechanism of the operating system kernel analyses

- Locking requests due to APO and LOCKPO and

- Other resource enqueuing requests to the Kernel

in order to detect deadlocks.

The system, i.e. the operating system and the applications should request the resources always via those functions which are supported by the deadlock detection mechanism.

Manipulation of Persistent Objects.

As already described in the Overview section, a persistent object is uniquely identified by a catalog entry , however functions for manipulating catalogs are not part of the Kernel. This has been decided to enable support of different catalog types and directory structures by the same data management Kernel. As a consequence of this, some functions for the manipulation of persistent objects have to be implemented (partly) above the data management kernel. These functions are described in the following:

Creation of a Persistent Object

This is performed by creating a catalog entry with PO-length = 0, followed by an APO for this catalog entry and the creation of the persistent object in virtual memory. The subsequent CPO may specify any PO-length which is in accordance with the rules for extending persistent objects (see below).

Deletion of a Persistent Object

The persistent object has to be committed with a PO-length = 0. Afterwards its catalog entry may be discarded. The CPO must obey the rules which hold with respect to shortening of persistent objects (see below).

Renaming of a Persistent Object

This function has to be provided completely above the Kernel. It simply consists of changing the catalog entry.

Extension of a Persistent Object

A persistent object is extended, if the PO-length specified with CPO is greater than the PO-length

so far known in the catalog. Such a CPO is accepted by the Kernel only, if the added part is under the exclusive control of the transaction issuing the CPO.

Shortening of a Persistent Object

A persistent object is shortened, if the PO-length specified with CPO is smaller than the PO-length so far known in the catalog. Such a CPO is accepted by the Kernel only, if the dropped part is not in use by another transaction.

Replacement of a Persistent Object

After a persistent object is brought into virtual memory (by use of APO), it may be overwritten or replaced arbitrarily.

SUMMARY

This paper described the data management part of a general operating system Kernel. Four instructions (APO, CPO, RELPO, and LOCKPO) have been defined for the realization of different kinds of access methods, file systems, and data base systems.

Besides the primary purpose of such a Kernel, to provide the respective functions in a most efficient way and to prevent the replication of these functions in multiple parts of the operating system, this data management Kernel offers the additional advantage that advanced facilities are offered in a generalized way.

Facilities which traditionally are associated only with transaction processing like concurrency control, support of backout and commit, support of a variety of share options and the safe file system mentioned in section "Basic Concepts" on page 4have been integrated into the Kernel and are thus applicable to all types of data management implemented on top of the Kernel.

It can be easily shown that the data management Kernel described in the paper represents an excellent basis for the implementation of file servers in distributed systems. The main advantages would be:

- Provision of an operating system independent, but still high level interface.
- Provision of windowing which allows to vary the amount of data which is communicated between the file server and its clients depending on configuration parameters.

REFERENCES

[DADAM 80] Dadam, P., "Einfuehrung in die Synchronisation in Datenbanksystemen", Elektronische Rechenanlagen, 23. Jahrgang 1980, Heft 1, pp. 4-12

[ESWARAN 76] Eswaran, K.P., J.N. Gray, R.A. Lorie, I.L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System", Communications of the ACM, Vol. 19, No. 11, Nov. 1976, pp. 624-633

[HAERDER 79] Haerder, Th., A. Reuter, "A Systematic Framework for the Description of Transaction-Oriented Logging and Recovery Schemes", Forschungsbericht TH Darmstadt Forschungsgruppe Datenverwaltungssysteme I, 1979, Nr. DVI 79-4

[GRAY 76] Gray, J.N., R.A. Lorie, G.F. Putzolu, I.L. Traiger, "Granularity of Locks and Degrees of Consistency in a

Shared Data Base", Modeling in Data Base Management Systems, G.M. Nijssen editor, North Holland, 1976, pp. 365-394. Also IBM Research Report: RJ 1654, Sept. 1975

[GRAY 78] Gray, J.N., "Notes on Data Base Operating Systems", Operating Systems - An Advanced Course, R. Bayer, R.M. Graham, G. Seegmueller editors, Springer Verlag, 1978, pp. 393-480. Also IBM Research Report: RJ 2188, Feb. 1978

[GRAY 80] Gray, J.N., "A Transaction Model", 1980, IBM Research Report: RJ 2895, Aug. 1980

[SPECTOR 83] Spector, A.Z., Schwarz, P.M., "Transactions: A Construct for Reliable Distributed Computing", 1983, Technical Report Carnegie Mellon University CS-82-143

[UNIX 78] UNIX Time Sharing System, The Bell Syst. Techn. J., 57 (No. 6, 1978), Bell Laboratories, Circulation Group, Whippany Road, Whippany, N.J. 07981

[MVS 76] IBM OS/VS2 MVS Data Management Services Guide, 1976