Which Kinds of OS Mechanisms Should Be Provided for Database Management ?

P. Christmann, Th. Härder, K. Meyer-Wegener, A. Sikeler
University of Kaiserslautern, West Germany

Abstract

The performance of database management systems (DBMS) critically depends on the availability of effective and efficient services offered by the underlying operating system (OS). The DBMS needs for OS support are identified, and appropriate interfaces for the cooperation of OS and DBMS are discussed. Several OS functions including file handling, process management, communication mechanisms, and transaction management are examined with a view toward their suitability for database management support. For these services, important properties and features are derived; their availability at the OS interface greatly improves OS-DBMS cooperation.

1. Introduction

Operating systems usually provide functions to create and manipulate files on secondary storage. The files are structured in blocks or records, and they may be organized in a sequential, relative (i.e. entries addressed by their number), or index-sequential manner. There are operations to read and write single entries, but any link between different files has to be established by the application programs. Relationships between files expressed by record fields with the same meaning, e.g. article number, or same record sets in different sort orders are not known to the operating system and thus cannot be maintained by it.

This is only one of the reasons why database management systems have been developed. These systems usually have not been built on top of the OS file system but instead use the most primitive mode of reading and writing physical blocks. Moreover, there are other areas (buffering in main storage, atomicity, parallel processing) where DBMS do not use existing OS functions but implement it themselves [St81]. This suggests that the services of most OS seem to miss the needs of DBMS. The same observations can be made with distributed DBMS; instead of using the communication mechanism provided by the OS (or a separate communication system) they employ just the basic message exchange and implement their own protocols on top of it (e.g. System R* [Li83]).

The purpose of this article therefore is to recall the needs of DBMS, and then to contrast them with the offerings of standard, advanced, and experimental OS. The result of our investigations concerning DBMS support should be a list of recommendations to improve current OS and to influence the design of future OS.

2. A Summary of DBMS Needs

2.1 Transactions

It is widely recognized that the concept of a transaction as an atomic, consistent, and isolated sequence of actions (operations) with durable results is a fundamental issue in DBMS [HR83a, Gr81b], equivalent to data abstraction and data independence. Its purpose is to provide data integrity and consistency despite failures and concurrent execution of user requests. In particular, it facilitates the design of DB application programs by isolating them from all aspects of parallelism and failure. Similar objectives have been considered by OS research which has recognized the importance of atomic actions [SS83]. However, atomic actions supported by OS do not reach the power of DBMS transactions. As a consequence, the four above-mentioned characteristics are usually implemented on top of existing OS mechanisms:

- *Atomicity*:
  Implementation of atomicity is a DBMS task, but it is based on smaller atomic actions, e.g. a single-block write to secondary storage that must be provided by the OS.

- *Consistency:*
  The ultimate goal is enhanced semantic integrity control by the DBMS. However, the OS can only control the consistency of the objects that it knows, e.g. files as a collection of blocks. This includes the maintenance of extent tables. If the OS offers records, then maintenance of access paths and tree-structured indices will also be its task. Consistency from the OS's point of view means that these physical structures are correct.

- *Isolation:*
  The unit of isolation for the OS is not the transaction, but the process. Locks on files, or sometimes on blocks or records, are acquired and released by processes. The DBMS can rely on this mechanism only if it decides to assign a single transaction to a process at a time.

- *Durability (persistence):*
  The DBMS provides durability of all committed data despite failures. For this purpose, the OS must guarantee the persistence of some objects, especially of blocks handed to it with a write request. If the OS maintains a file cache or buffer, these blocks will not be persistent before they are flushed to disk. Instead a DBMS needs a so-called force-write that immediately writes data to non-volatile storage.

The discussion so far assumed the DBMS to be just a single instance, i.e. a program and a process, and for reasons to be discussed it can be split into several processes running in the same or even in different systems. A transaction may span the activity of more than one process which leads to the

need for coordinated commit of all subtransactions in the processes engaged (e.g. two-phase COMMIT [Gr78]). The OS may indeed support this coordination in that it keeps track of the processes participating in the transaction and generates PREPARE-TO-COMMIT, COMMIT, and ABORT messages when requested by the coordinator process [RN84].

2.2 Processing

The last paragraph already mentioned the fact that a number of processes can be used for DBMS processing. The needs of DBMS concerning the process structure are characterized by three principles:

- *Protection:*
  Internal data structures of DBMS (whether they are in main storage or on disk) must not be accessed by application programs (APs). This is hard to guarantee, if program and DBMS run as a single process with a single address space. The OS then treats them both as one entity and, as a consequence, will execute read or write requests to DB files that are issued by the application program. Solutions have to assign DBMS and AP to different address spaces or to structure the address space into different protection domains.

- *Communication:*
  The transfer of requests and responses between the AP and the DBMS should be easy and quick. There seems to be a trade-off with the before-mentioned protection. If they are both put in the same process, the transfer is reduced to a subroutine call with parameters, whereas communication between different processes is far more expensive (in terms of machine instructions), even if shared memory segments can be used. Communication becomes even more crucial if the DBMS itself is distributed over several processes.

- *Potential of parallelism:*
  It should be possible for the DBMS to proceed with another request while one is waiting for the completion of I/O. In general, a database is spread over a number of disk devices each of which can perform an I/O operation in parallel with the others. This should be utilized by DBMS to increase throughput [St81]. This can be done with or without OS support, where either the DBMS runs a number of processes and the OS switches to the next while one waits for I/O or the DBMS performs its own multitasking inside a single process.

Quite a number of mechanisms offered by OS have to be investigated to see how these needs can be met: process management, inter-process communication, shared memory segments, protection domains, messages in general, etc. Faced with a number of new proposals and experimental systems it should be remembered that "the cheap process and the cheap message are the two myths of computer science" (Bruce Lindsay).

2.3 Secondary Storage

Before going into the details of which objects the OS should offer to the DBMS (blocks on disks, pages in a buffer, or records), some more general requirements concerning the storage should be remembered:

- it must be non-volatile, i.e. its contents must survive a power failure
- it must offer a notion of physical contiguity that helps to optimize a sequence of I/Os
- I/O should be as fast as possible
- there must be some support for the persistence of blocks (i.e. force-write).

It is to be noted that again there is a trade-off between some of these goals. While the need for fast I/O encourages some sort of caching, the persistence required to implement the transaction concept (e.g. in writing log data) does not allow the use of it all the time.

The next section introduces the internal structure of a DBMS in terms of an abstraction hierarchy. This structure allows us to show how far the concepts of various OS support DBMS work. The discussion will refine the abstract view of this section.

3. The Mapping Hierarchy of a DBMS

Thus far, we have described the DBMS needs at a rather abstract level. In order to approach the question of which primitive OS functions can be used for the implementation of a DBMS, we outline the mapping hierarchy of a DBMS. Such a hierarchy transforms step by step the stored representation of the DB on non-volatile storage (a huge collection of bits on disks) to the logical view of data as referred to by the application program or user, i.e. to the objects and operations of the data model used.

For this purpose, a multi-level hierarchic model for DBMS implementation was introduced elsewhere [HR83b]; here we only describe its essential features. Fig. 1 illustrates the mapping layers by their most important tasks and some typical auxiliary mapping data; the kind of interfaces between them is sketched by some typical objects and operations. The architectural model gives only a static description of the mapping process; to derive a user object, the bit representation of data on disk is dynamically transformed into a hierarchy of more and more abstract objects. At each level a set of suitable operations is provided for the corresponding objects in order to construct the objects and operations available at the next higher level. Each layer implements the objects and operations offered at the interface to the next higher layer. They are used as primitives at this interface to accomplish the tasks of the next layer.

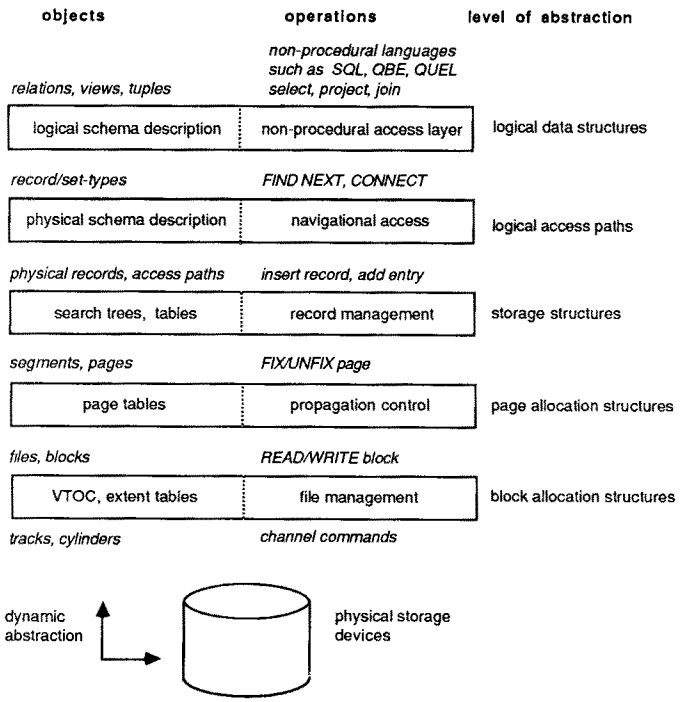


Fig. 1: Description of the DBMS mapping hierarchy

3.1 A Multi-Level DBMS Model

Let us quickly describe the major functions of each layer in order to improve the understanding of our architectural model:

- *File management:* The bottom layer copes with the physical characteristics of external storage media, abstracts these characteristics into fixed-length blocks, and offers an elementary file interface to the next higher layer. Such an interface allows simple read and write operations of fixed-length blocks identified by a (relative) block number and a few file control operations (e.g. open/close, create/drop).

- *Propagation control:* Based on the file management interface this layer establishes a further abstraction consisting of segments with visible page boundaries. These may be ideally used as 'infinite' linear address spaces by the next layer. A page is a fixed-length partition of a linear address space and is mapped onto a block by the propagation control layer. Therefore, a page can be stored in different blocks during its lifetime in the database. Hence, the conceptual separation of pages and blocks allows the introduction of mapping redundancy which may be used for fault-tolerance and failure recovery. For example, shadow page algorithm [Lo77] or differential file method [SL76] could be taken to implement mapping redundancy by the

propagation control. Update-in-place algorithms on the other hand do not use such (expensive) redundancy when propagating pages to blocks on disk.
A second function of this layer is to maintain a DB buffer for the purpose of interfacing main memory and disk. The buffer (volatile, typically several MBytes and more) consists of page frames of uniform size which contain pages to be processed. The record management is aware of the page boundaries and uses the DBMS catalog, index structures, address translation tables, etc. to find the page numbers of the pages it has to access on behalf of a transaction (user). A page request is issued by a FIX operator which involves a page-lookup in the buffer and potentially page replacement (including propagation of modified pages). A fixed page can be directly referenced by the requestor; it can execute machine instructions addressing data objects within that page. The propagation control layer guarantees addressability until the page is explicitly released (UNFIX).

- *Record management:* This layer implements and maintains all physical object representations in the database (records, fields, etc.). A variety of access path structures such as pointer chains, hash tables, search trees, etc. has to be provided for efficient access to DB objects. Updated records have to be reflected in all related access path structures to guarantee consistency of physical storage structures.
  For performance reasons, the page structure of segments is still visible at this level. All implementation techniques are explicitly designed and optimized for page structures, e.g. B*-trees or page-oriented hash tables. Since a DBMS should offer a great variety of access path and representation structures, this layer implements mapping functions much more complicated than those performed by subordinate layers.

- *Navigational access layer.* The record management layer provides primitive operations on physical objects. They are used by the navigational access layer to implement objects and operations that are typical for a procedural data manipulation language (DML). Some abstraction is gained by implementing logical access paths (which hide the characteristics of the referenced physical structures). Hence, the user navigates along access paths, hierarchies, or networks with operations like FIND NEXT. Such a 'one record at a time' interface is comparable to navigational DBMS interfaces like CODASYL [CODA78] or IMS [IBM].

- *Non-procedural access layer:* The top-most layer has to provide logical data structures and a non-procedural language. This implies that the user does not see any access paths and has to refer to abstract objects such as relations or views. With each operation the user can handle sets of tuples rather than single records. The layer has to translate and optimize the set-oriented user queries into sequences of record operations along access paths. Since the user specifies only 'what but not how', the DBMS is solely responsible for performance. The typical example for the abstraction gained by the top layer is the relational model with a high-level query language such as SQL [As76] or QUEL [St76].

The architectural model in Fig. 1 assumes 'clean' interfaces with strict observation of the information hiding principle. Compared to existing systems, our mapping hierarchy may be somewhat idealized, that is, often some information available at higher layers is bypassed to lower layers for performance optimization, e.g. prefetching of pages in the buffer. Furthermore, real systems usually do not exhibit such a detailed explicit mapping hierarchy. To save runtime overhead some consecutive layers are 'glued' together in a single component. For example, System R [As76] is divided into only three explicit layers: the storage system comprises the two bottom-most layers, the access system is comparable to the middle layer, and the data system contains the two top-most layers.

However, we prefer our architectural model for a number of important reasons. For didactic purposes, it gives a comprehensive view of implementation concepts. The separation of functions and clarity of concepts yield a deeper understanding of interdependencies among functions and components and of data independence. Finally, the same architecture and implementational principles can also be found in distributed DBMS. Each node of a DDBMS must provide all functions of a centralized DBMS.

Traditional approaches to DDBMS do not rely on services of distributed OS which could (partly) solve various problems associated with the distributed nature of the system - data as well as processes. A particular layer of the DBMS at each node is aware of the local distribution and implements a 'global view' to all layers above (and, especially, the application program), i.e. it provides most aspects of distribution transparency. Typically, the top-most layer of our DBMS mapping hierarchy performs this task without using special OS mechanisms (besides communication primitives). In particular, query optimization and distribution of work (subqueries) are performed at this level taking into account communication costs and overhead of algorithms incorporating various nodes (e.g. joins). Therefore, we do not pay much attention to approaches trying to solve distribution transparency at the OS level. In the contrary, it should be noted that most aspects of transparency in a distributed system - location, replica, fragmentation, concurrency control, and failure transparency - are not desirable for the DDBMS itself, because they would prevent high level optimization and overall distribution decisions of the DBMS work.

3.2 What Is the Appropriate OS-DBMS Interface?

So far, we have described the mapping hierarchy of a DBMS and identified a number of interfaces in a 'used'-hierarchy for a suitable system implementation. Our discussion did not include considerations of whether such an interface should be provided by the OS or implemented by the DBMS. For this purpose, the layered hierarchy of Fig. 1 is a convenient scheme to select an appropriate OS-DBMS interface.

A puristic DBMS approach (I) uses the interface to the external storage media as a starting point for system implementation, i.e. it does not exploit OS services at all. Some early DBMS implementations were developed according to this 'do all by yourself' philosophy [Si77]. All major reasons for such a decision were performance-related, i.e. extreme performance requirements dictated tailored solutions and ways to abandon (expensive) OS services. However, this approach has severe drawbacks since portability and independence of the DBMS may not be achieved. For example, change of storage technology or communication protocols affect the DBMS code.

A second approach (II) is based on the file management interface. The OS offers a simple file concept for the DBMS implementation. Typically, neither concurrency control and recovery nor a transaction concept is supported by such an interface. However, the use of this (and every higher) interface guarantees isolation of the DBMS code from the external world (external storage, communication). This approach may be denoted as the 'classical' approach, since most DBMS implementations rely on OS file management.

The next possible OS-DBMS interface (III) is the buffer interface. Ideally, the OS could provide a potentially unlimited linear address space for the DBMS, e.g. a virtual memory architecture. Requests to data objects could be made directly in terms of virtual byte addresses thereby referencing variable-length byte strings. Such a solution implies an enormous mapping overhead if the storage objects may dynamically vary their sizes. According to [Tr82], it appears to be infeasible mainly for performance reasons.

However, even if the OS offers a less refined interface with visible page boundaries to the DBMS (often called the OS file cache) a number of severe disadvantages has to be taken into account [St84]:

- Access overhead is too high. Just to move a block of data across the DBMS/OS boundary may cost as much as 5000 instructions.
- The replacement algorithm in the file cache may not be optimal. It cannot be adapted to DBMS-specific access characteristics since it is designed to serve all OS file users. To be noted is that a database-oriented prefetch policy is not implementable because sequential access in the database does not always mean access to neighbouring pages of an OS file.
- Selected force-out is not possible. The file cache manager writes pages back to disk according to the cache's replacement algorithm. Unfortunately, most DBMS recovery systems are based on the possibility of being able to force specific pages out of the buffer at certain points in time in a specific sequence (typically during the COMMIT phase at end-of-transaction). Since the correct force-out sequence is only known to the DBMS, the OS file cache with its replacement algorithm would interfere with the log- and recovery-manager of the DBMS, and recovery after a system failure would be impossible.

A fourth possible interface (IV) for OS-DBMS cooperation is the internal record interface. The OS implements storage objects for record types as well as a rich variety of access path structures (sophisticated logical access methods). Every single record operation is a call to the OS (SVC). The interface may be suitable characterized as a 'physical record at a time' interface with single scan property. Only external (logical) records and views consisting of joins (record type crossing operations) are derived by the navigational access layer. The DBMS processing is in this way considerably simplified. Moreover, the transaction concept as well as logging/recovery and concurrency control functions are integrated into the OS. An example of this approach is the so-called disk process of Tandem's OS Guardian [TSR85].

The choice of an even higher OS-DBMS interface means that the entire DBMS is integrated into the OS (database operating system [Gr78]). The implications of such approaches are not investigated, since they are rarely implemented in practical systems.

Our discussion has revealed that there are two interesting interfaces for OS-DBMS cooperation. Hence, we refine our considerations mainly for approaches (II) and (IV).

4. File Systems

In a first attempt, OS file systems may support both approaches for OS-DBMS cooperation depending on whether they are block-oriented or record-oriented. On closer inspection most record-oriented file systems (except Tandem's disk process [TSR85]) turn out to be unsuitable in exchange for the three lower layers of our DBMS mapping hierarchy (Fig. 1). The reasons for that are manifold: The functionality of both is not comparable, sophisticated access path structures as well as a transaction concept are not supported, the underlying file cache management is not appropriate (see. 3.2), etc. Therefore, replacing the DBMS file management by a block-oriented OS file system is the better way for OS-DBMS cooperation at the file level. However, there are some requirements placed on such a block-oriented OS file system, most directly derived from requirements imposed on the propagation control layer [Si87].

4.1 Requirements of DBMS

In conventional DBMS (e.g. System R [As76]) a database at the propagation control layer commonly consists of a number of segments each divided into pages of equal size. Since pages are separately asked for and freed, the unit of data exchange between disk storage and main memory is typically a single page, i.e. block. As a consequence, the size of objects manipulated by the record management layer is limited by the corresponding page size. Thus modeling of application objects to be stored in the database must consider the page size. In commercial applications objects are

simple, they can be described by a single record of limited size (approximately less than 2000 bytes). The objects of so-called non-standard applications (such as office automation, geographical data processing, or CAD/CAM), however, are generally more complex, and often composed of other simple or complex objects. Even a simple object can be described by only a few bytes or by several MBytes. Hence, the record management layer of so-called non-standard DBMS handles records spanning two or even more pages and it clusters records describing one complex object into one or more pages [DPS86, HMMS87]. As a consequence, the propagation control layer also has to be extended in regard to objects as well as operations in order to improve performance:

Dynamic and temporary segments

The amount of data stored in a database may vary strongly over time. Therefore, neither the number of segments nor the size of a segment should be static, i.e. segments should be created, expanded, shrinked, and deleted dynamically. Furthermore, the higher DBMS layers need temporary segments in order to store intermediate results during the execution of a complex query. Such a segment, however, should be deleted automatically when it is closed, at (database) system shutdown, or in a transaction-oriented environment at the end of the corresponding transaction.

Different page sizes

Since the record size of different record types may be very different, it seems useful to offer an appropriate number of different page sizes. Then the page size may be defined in order to approximate the record size or to provide record clustering at the page level. The page size, however, should be a parameter of the corresponding segment, i.e. all pages of a segment are of equal size which is kept fixed during the lifetime of the segment.

Set-oriented operations on pages

Nevertheless, even different page sizes do not meet all requirements of non-standard applications. The restriction to a certain page size, say 8 Kbytes, is too stringent, especially regarding arbitrary length objects such as complex objects or strings. Hence, it would be helpful if the propagation control layer could support set-oriented operations on pages. As an example, PRIMA (prototype implementation of the molecule-atom data model [HMMS87]) distinguishes between three kinds of set orientation. A page sequence as a predefined set of pages treats an arbitrary number of pages as a whole. Since page sequences are used to store arbitrary length objects which may also vary in size, they have to be dynamic, i.e. a page sequence may grow and shrink regarding the number of assigned pages. A page set serves to reduce the number of calls at the interface of the propagation control layer since a set of pages and/or page sequences can be fixed and unfixed by a single call. Both page sequence and page set may be used to optimize disk access (chained I/O, optimized channel programs [WNP87]). The third set-oriented operation of PRIMA, the page contest, supports the access to replicated pages. A page contest delegates the decision, which page or page sequence from an arbitrary set is provided in the database buffer, to the propagation control layer. Selection criteria consider data exchange and synchronization needs.

For performance reasons most of these objects and operations of the propagation control layer should be mapped directly onto corresponding objects and operations of the file management layer or of an appropriate OS file system. That is, such a OS file system should support

- dynamic and temporary files,
- different block sizes, and
- set-oriented operations on blocks ("block sequence" and "block set").

The main task of the two bottom DBMS layers is to minimize the number of disk I/Os in order to achieve good performance. The number of disk I/Os, however, depends on the access pattern at the interface of the propagation control layer which is strongly influenced by the behaviour of the record management layer. This layer manages records stored in a page or a page sequence allowing for direct and sequential access to them as well as additional access path structures such as hash tables or B*-trees. Therefore, the access pattern is a combination of

- random access to a page (page sequence) containing records with a very low ($\approx$0) probability of rereferencing
- random access to a page belonging to a hash table with a high probability of rereferencing
- sequential access along the logical page numbers (scan) with a very low ($\approx$0) probability of re-referencing for the single pages
- sequential access along a tree structure with a high probability of rereferencing, especially for the root page

according to the different storage structures [St81]. However, this is one of the reasons why DBMS are not built on top of a virtual memory management or a file cache. Rather they implement their own database buffer using more appropriate replacement algorithms. In that way, the number of blocks requested from the file management is kept small. Nevertheless, requesting a block may cause several disk I/Os depending on how files and blocks are mapped to disk and whether a "file cache" is used. The block to slot mapping determines in a high degree the flexibility and performance of the overall DBMS [HR83b] since auxiliary information (such as volume table of contents, extent tables, etc.) have to be maintained in order to read or write a block. Hence, this mapping function has to be designed carefully, especially with respect to the file size (DBMS files may become very large) and the block sequences introduced above. Additionally, a "file cache" may be used in order to store current auxiliary information in main memory. Data blocks, however, should not be stored in this "file cache" since the propagation control layer already maintains a corresponding buffer. As a consequence, write operations directly force out data blocks on disk. In order to achieve reliable write operations modified auxiliary information has also to be written to disk.

Summarizing these aspects an OS file system has to support

- efficient block to slot mapping in order to minimize the number of disk I/Os for reading or writing a block
- force write operations
- block sequences and block sets in an efficient way (e.g. by chained I/O).

Although locking and logging/recovery are often integrated into the file management layer for simplicity of implementation, an in-depth discussion of the transaction concept (chapter 7) will show a number of serious disadvantages of this solution (e.g. page granules for locking and logging are very expensive [HR83a]). Therefore, transaction management should be part of the record management (or even a higher DBMS) layer. However, to that the file management layer has to support a controlled force-out of pages in order to enable reliable recovery. Hence, the file management layer, or a corresponding OS file system as well, should support force-write of blocks.

So far, we will conclude our considerations concerning the requirements of (non-standard) DBMS. In the following, some well-known as well as some experimental OS file systems are investigated with respect to how they may satisfy these requirements.

4.2 OS File Systems

Each OS provides its own file system supporting either a block-oriented or a record-oriented interface (or both, e.g. PAM, SAM, ISAM of BS2000 [Ko87]). However, all these file systems are more or less different from each other, not only with respect to the objects and operations offered at the interface but also regarding the block to slot mapping and the management of a file cache. The file systems investigated in the following make this clear. Nevertheless, most of the statements will prove applicable to many other file systems as well.

In order to achieve a more general overview, four different file systems are incorporated in our investigation. Apart from the UNIX file system [McK84, QSP85] and the BS2000 PAM file management [Ko87] which both support a wide range of applications, two experimental systems are considered. The ALPINE network file system [BKT85] is of some interest since its primary purpose is to store files that represent databases. The DISTOS file system [Fr87], however, is partially influenced by the requirements outlined above. Let us now examine the interface of each system, i.e. the objects and operations offered by them, as well as their block to slot mapping and their file cache management (the transaction concept will be treated in section 7). The properties of the various file systems are summarized in Table 1:

Dynamic and temporary files

Whereas dynamic files are commonly supported, temporary files are treated by each system in a different way. In BS2000 temporary files are combined with the process which has created the file, i.e. the file is automatically deleted at the end of the process provided that no other process has locked the file. DISTOS, however, does not support temporary files in any way. Hence, the user himself has to implement them. In contrast to these, UNIX may be generated in such a way that all files belonging to the directory /temp are automatically deleted, for example, at a certain time or at system startup.

file system	dynamic files	temporary files	different block sizes	set-oriented operations on blocks	block to slot mapping	general cluster mechanism	file cache	force write
UNIX file system	+	directory /temp	-	read/write a byte sequence of arbitrary length ↳ arbitrary number of logical consecutive blocks no chained IO	tree structure	-	data blocks and auxiliary information	-
BS2000 PAM file management	+	combined with process	-	up to 16 consecutive blocks (on disk and in main memory) using chained IO	extent-based	-	data blocks and auxiliary information	+
ALPINE file system	+	?	-	page runs ↳ arbitrary number of consecutive blocks (on disk and in main memory) using chained IO (?)	extent-based	-	data blocks and auxiliary informatiom	transaction concept
DISTOS file system	+	-	0.5, 1, 2, 4, and 8 kbytes	block sequences block sets	tree structure	-	auxiliary information	force write

Table 1: Properties of various OS file systems

Different block sizes

The DISTOS file system is the only system that supports five different block sizes (1/2, 1, 2, 4, 8 Kbytes) whereas all others are restricted to a single block size (e.g. 1/2 or 2 Kbytes). In UNIX, however, files are treated as a sequence of bytes and the page structure has to be implemented by the propagation control layer. Therefore, different page sizes are possible but the block size, i.e. the unit of data exchange, is fixed.

Set-oriented operations on blocks

Each of the file systems supports set orientation in some way. In UNIX each read or write operation allows for reading or writing of a byte sequence of arbitrary length. In addition to that, a certain number of logically consecutive blocks have to be read or written. However, depending on the underlying block to slot mapping (see later) this cannot be done using chained I/O. The read/write operations of ALPINE refer to so-called page runs. A page run is an arbitrary number of consecutive blocks which may be handled by chained I/O. On the other hand, a read/write operation in BS2000 is restricted by a maximum of 16 consecutive blocks. Both, however, assume the blocks to be consecutive on disk as well as in main memory. The most common set orientation is supported by DISTOS. It is possible to read or write an arbitrary number of single blocks, a sequence of blocks, and the whole file, with the addition that blocks may be clustered. All blocks of a cluster are stored on disk in such a way that an efficient access on the whole cluster is feasible, e.g. by chained I/O.

Block to slot mapping

In principle, two main techniques to map a file consisting of n fixed-length blocks to disk may be distinguished [HR83b]:

- extent-based allocation with dynamic growth using fixed- or variable-length extents (BS2000, ALPINE) and
- dynamic allocation based on a tree structure (UNIX, DISTOS).

However, the dynamic allocation has some disadvantages in the context of DBMS. Records are often accessed specifying the desired value of a certain attribute. Therefore, DBMS maintain additional access path structures such as a B*-tree in order to speed up this kind of access. Typically, a B*-tree has a height of 3, i.e. accessing a single record requires 4 page requests. However, using a tree-based allocation strategy may dramatically increase the number of corresponding disk I/Os, since each page request may cause several disk I/Os in order to read the proper data block (see Fig. 2 for UNIX/DISTOS). Additionally, 4 pages have to be replaced and possibly forced out to disk which also cause several disk I/Os. Hence, an extent-based allocation seems more appropriate [HR83b]. This structure, however, contradicts a general cluster mechanism that allows to cluster arbitrary blocks (e.g. a cluster containing blocks 127, 12, 15, 31 in the given order).

File cache

UNIX, BS2000, and ALPINE maintain their own file cache in order to store data blocks as well as auxiliary information. Additionally, ALPINE supports a transaction concept thus implementing atomic

file update while BS2000 only supports force-write. In UNIX, however, data may be lost since a write operation primary effects the file cache, dirty blocks are forced out periodically. The file cache of DISTOS, on the other hand, is only used to store auxiliary information.

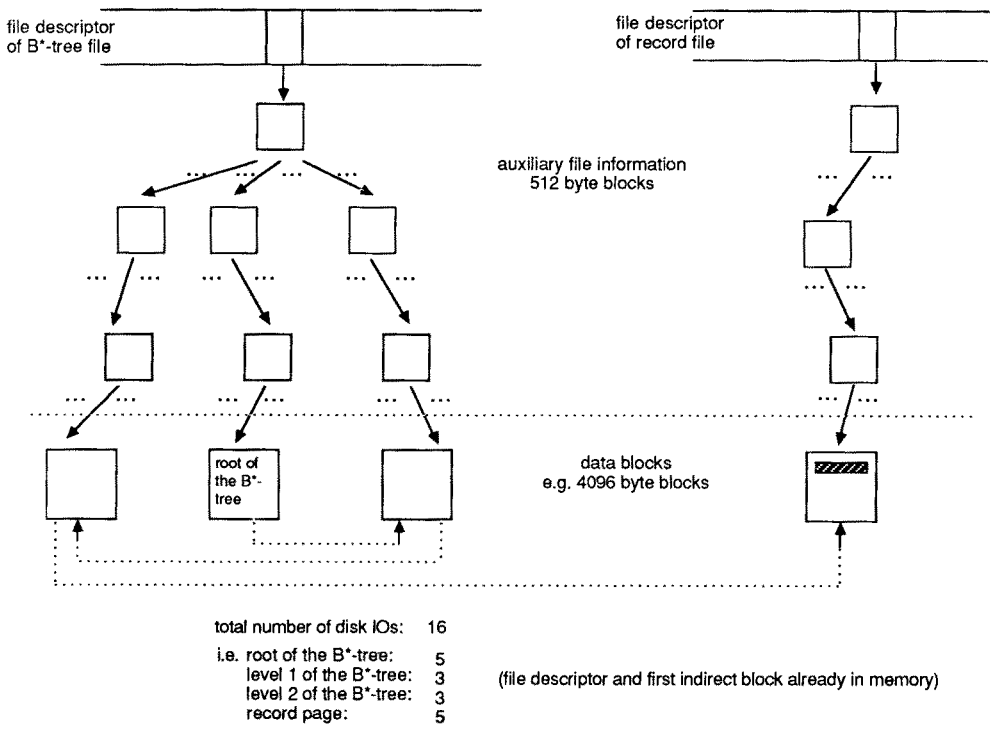


Fig. 2: Number of disk I/Os in UNIX/DISTOS in order to access a single record

4.3 What Should OS File Systems Support?

Summarizing the results of the above investigations two main requirements for an OS file system may be obtained. The first concerns the objects and operations which a file system should offer. Above all set-oriented operations on blocks (as well as different block sizes) are of prime interest. Regarding set orientation a flexible concept is mandatory, arbitrary block sets as well as predefined block sets ("block sequences") should be supported, the later one by chained I/O, for example. Therefore, a general cluster mechanism allowing for consecutive storage of arbitrary blocks is needed. However, this cluster mechanism may not conflict with the second important requirement, i.e. an appropriate block to slot mapping that supports both random and sequential access in an efficient way (even for large files). This mapping should be designed carefully, since it strongly influences the performance

of the overall DBMS. The major objective should be a single disk I/O for random access to a block, a "block sequence", and perhaps an arbitrary block set. A file cache may be useful in order to minimize the number of disk I/Os with respect to read operations. Write operations, however, should be reliable.

Furthermore, an OS file system should support asynchronous operations on files and blocks. This requirement was not yet mentioned, but is however of some interest regarding the process structuring treated in the following.

5. Processes and Parallelism

The mapping hierarchy introduced in chapter 3 does not reflect the fact that a database is accessed by a number of users and programs concurrently. It also does not indicate the potential to execute parts of a single DB operation in parallel. This chapter adds a discussion of the dynamic aspects, i.e. running the DBMS in a number of processes. The primary purpose is to serve the application programs (APs), and this can be done in a number of ways as shown by the classification tree of Fig. 3. A secondary purpose is to do this as efficiently as possible which may lead to a functional distribution of the DBMS over several processes. For instance, while one process extracts the data according to the mapping hierarchy of section 3, the other may already request the locks, write the log information, and check the user's right to access those data. This kind of parallelism is not included in Fig. 3.

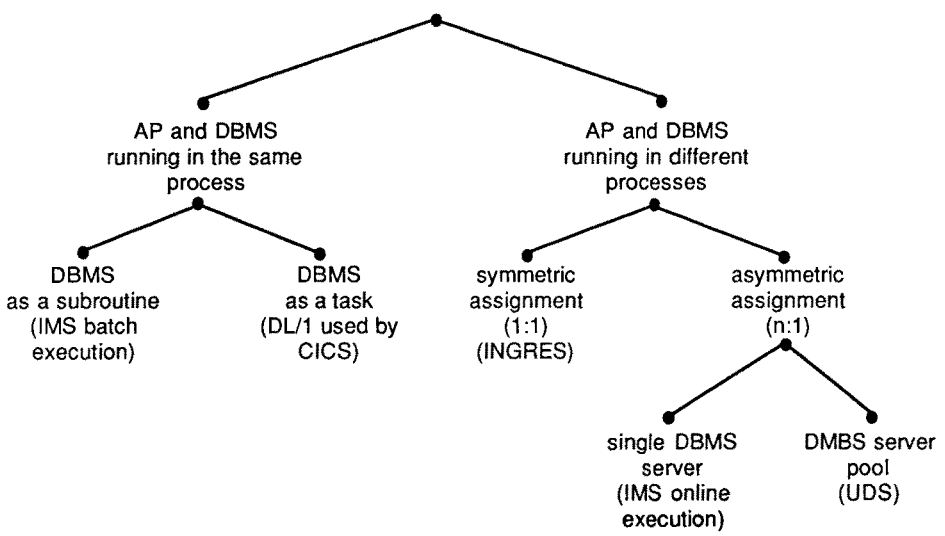


Fig. 3: Classification of AP and DBMS process configurations

5.1 Requirements of DBMS

Apart from the three general requirements already mentioned in the introduction (protection, fast communication, and parallelism) the needs of DBMS strongly depend on the chosen process configuration. If DBMS and AP run in the same process, communication is no problem at all, but protection becomes a critical issue. There is a need for different protection domains within a process (or an address space). If there are several instances of a DBMS in a number of processes, a notion of sharing is important for

- files (the database, but also log files, queues etc.);
- main storage (DB buffer, lock table, again queues).

Therefore, it is appropriate to describe the configurations of Fig. 3 in more detail. The discussion follows the work of [Hä79] and [St81]. Today's OS usually provide protection only, if AP and DBMS are running as separate processes. The transfer of requests and responses should then be facilitated through overlapping portions of the address spaces to avoid huge message traffic. The OS mechanism needed for this is usually called 'common memory' or 'shared segment', and it is limited to rather large portions of virtual memory. Therefore, a single shared segment is used by all the application processes. Still the database buffer (DBB) and the global system tables (GST) are protected from access by the application programs. When either communication partner (DBMS or AP) has put some information in the data exchange area, it triggers the partner through the OS synchronization primitives (e.g. SIGNAL).

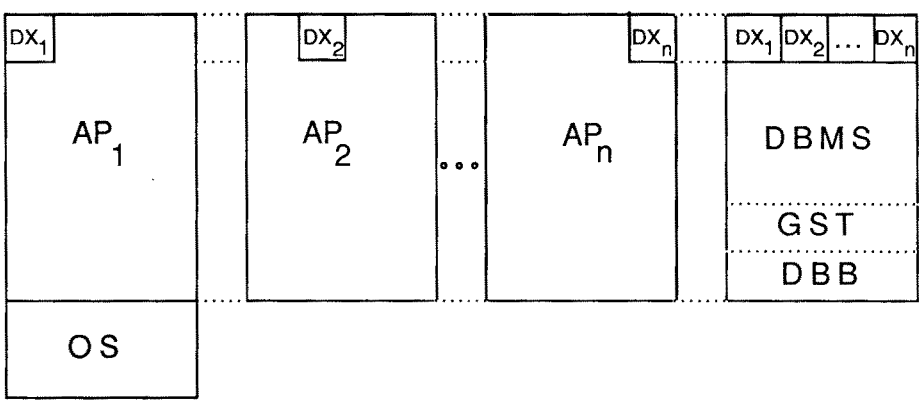


Fig. 4: Process configuration with single-server DBMS

Fig. 4 depicts the case of a single DBMS server. It has the advantage that no communication and no process switches are required while processing a DBMS operation. The DBMS process, however, has to perform asynchronous I/O and, as a consequence, multi-tasking in order to avoid the formation of a bottleneck. The OS has to assign high scheduling priority to it, but in the case of a page fault the whole DB processing stops anyway. To summarize, the OS does not help the DBMS in processing

multiple requests (scheduling, synchronization) in any way.

The symmetric assignment of DBMS processes to application processes is shown in Fig. 5. It eliminates the need for multi-tasking in the DBMS processes. In addition to Fig. 5, even the DBMS code can be put into a shared segment (assumed it is reentrant). Then only the local data have to be allocated for another DBMS process. The OS still needs a sizeable amount of process management data, e.g. status registers, control blocks, segment tables, etc. This all makes the configuration rather expensive. There is also a need for synchronization when accessing the GST and DBB, and the OS must allow for shared access to files by a number of processes. As only one process in a pair of AP and DBMS is active at a time, a waste of address spaces can be stated.

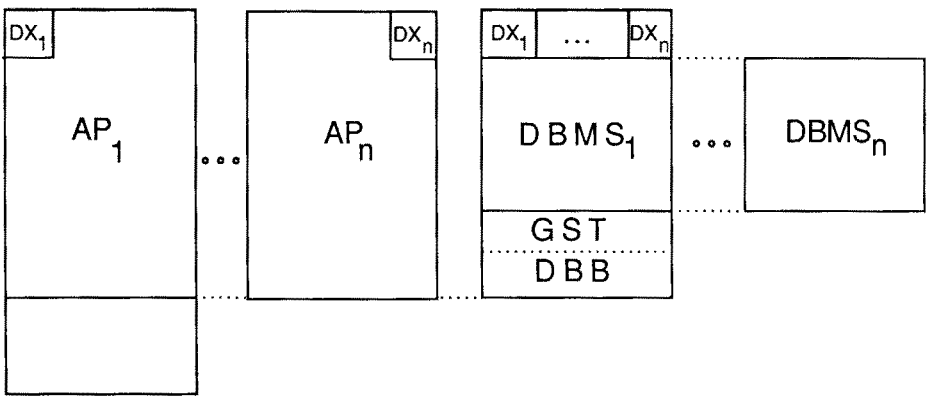


Fig. 5: Process configuration with symmetric assignment of DBMS processes

The case of a multiple server DBMS then seems to yield a compromise. The picture looks exactly like Fig. 5, only that we have m DBMS processes instead of n. As a rule of thumb, usually 2 to 5 APs are served by one DBMS process. In the contrary to the symmetric case, assignment is now done dynamically on an per-operation base. Synchronization overhead among DBMS processes increases, since access to data exchange areas must be coordinated, too.

Any case of multi-process DBMS is susceptible to the convoy phenomen [BGMP79]. There are proposals as to how it can be avoided [PR83]. The idea is that processes may declare themselves uninterruptable for a short period of time, e.g. a time slice, during which they are not calling for OS services (in particular, no I/O). This is just the time while they are holding short-term locks (semaphores, latches).

On the other hand, a multi-process DBMS is a very easy way to utilize multi-processor hardware. One can also combine it with multi-tasking within each process in order to avoid the expensive process switch. This of course makes the DBMS implementation and code as complex as in the single-server case.

Finally, we can state that the separation of AP and DBMS raises the following requirements on the OS:

- shared memory segments (at best of small size, e.g. one page);
- synchronization primitives (as cheap as possible);
- sharing of files among processes;
- some help to avoid convoys, e.g. non-preemptive scheduling (on request).

If the OS is developed to an extent that it can manage different protection domains inside a single address space, then there is no need to separate AP and DBMS. The OS involvement in the communication between AP and DBMS is minimized, and so are the costs. The somewhat 'ideal' concept is depicted in Fig. 6. Advantages and disadvantages are quite similar to those of the symmetric assignment.

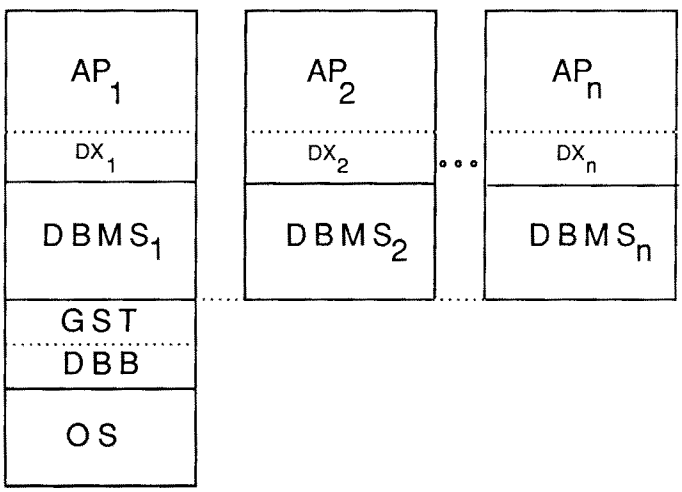


Fig. 6: Process configuration with linked-in DBMS

From the OS point of view, each DBMS instance now stands for exactly one user (one AP). Therefore, the OS can distinguish requests from different users and can (to some extent) handle conflicting access. Of course, it does not know about access to the DBB in shared memory. Thus the DBMS must still maintain its own lock table. However, the OS knows about all disk writes of a process, and therefore it could do some transaction management for the processes. This question will be raised again in section 7.

5.2 OS Process Concepts

There is a wide variety of OS with completely different process concepts and, to make things worse, with similar names for them. On a higher level of abstraction, a process is always the unit of

scheduling, of resource allocation, and of protection. The old commercial OS have been designed to manage a small number of processes, and during years of practical experience they have been enriched by numerous features that all tend to make a process switch expensive and slow (e.g. sophisticated priority and scheduling strategies). It is known that such a process switch usually consumes from 5000 to 20000 machine instructions. This becomes painful especially when processes are not just working on their own, but instead have to cooperate.

Among the features that have been added to the old OS are shared data segments (e.g. the Common Service Area in MVS [MVS80]) and process communication. The latter is often very expensive, because it has been designed in a general way and does no optimization for the special case. Again several thousand machine instructions are consumed by the management of queues, bourses, etc. Some systems offer only one-to-one communication (e.g. the pipe in Unix [QPS85]), while others also support the many-to-one case that many DBMS configurations need.

In order to manage a large number of small processes (required in online transaction processing) some OS have very early been extended by the concept of a task inside a process. This has recently gained some attraction under the new name of 'lightweight processes' [Svo85]. The so-called multi-tasking inside a process reduces the amount of expensive process switches. It can be performed by the application programs themselves (often a TP monitor) or by the OS. The latter has some advantages:

- scheduling and dispatching on two levels is done by the OS only
- when a task is interrupted by a page fault, the OS can switch to another task of the same process instead of activating another process
- implementation of the AP performing multi-tasking (e.g. a DBMS) is made easier.

A single disadvantage is that this OS subtasking is again rather expensive. Therefore several systems were built that do their own multi-tasking.

Usually there is only one address space per process, and it is divided into two protection domains, one for the OS, the other for the user. Among the commercial OS, there is only Multics [Org72] offering a ring protection scheme with more domains in an address space.

The problems of embedding a DBMS in such OS have led to further enhancements that all add complexity to the OS. One example are the Cross Memory Services of MVS [MVS80]. They are based on some hardware extensions, i.e. new control registers and machine instructions. The principal idea is to call a subroutine in another address space without doing a process switch, executing that subroutine with the priviledges of the other address space and then returning control to the caller thereby reestablishing its access rights. This allows processes to work together without performing a process switch and fits very well to the client-server relationship between AP and DBMS. However, the mechanism is rather complex and not easy to use.

In contrast to the old OS there are new concepts in OS development. UNIX is probably the most prominent example [QPS85]; Tandem's Guardian is also well established by now [Bo81, TSR85]. The idea is to make processes more flexible, easier to create and manage. If they become less expensive, there can be more of them. Instead of providing functions as OS calls to processes, they are executed in another process. Process switches are performed with 500 to 1000 machine instructions. Inter-process communication is a central issue of the whole OS as well as application design and therefore optimized. Unfortunately, some of the features known from the old OS are no longer available. There are no shared data segments except for reentrant code, and they must not be modified, thus cannot be used for the transfer of large amounts of data. Furthermore, the UNIX pipe only supports one-to-one communication [QSP85].

Tandem's Guardian is based on the concept of the remote procedure call, i.e. a message pair of request and response. By the way, it works in the same way when used for long-distance communication between systems, which in most other OS is done with a different set of functions (see section 6). To support the one-to-many and many-to-many type of communication Tandem has introduced the notion of an application server class. This is a group of processes with the same name, so that every message sent to the "process" with that name can be received by any process within the class. This fits very well to the asymmetric assignment of DBMS servers to application processes.

An expermimental system, with which we have gained some experience, is DISTOS in connection with the programming language LADY that must be used for implementing the applications running on DISTOS [Ne85, Ne87]. LADY provides mechanisms for the definition of processes and monitors. Monitors are used for process synchronization and communication. Processes and monitors can be grouped in teams. A team is the unit of distribution, i.e. each team can be assigned to a different processor, while the processes and monitors of the same team always run on the same processor. Communication among teams is established through typed ports. A LADY system finally consists of a collection of teams. The important characteristics of DISTOS and LADY are:

- the number of processes is fixed; processes cannot be created dynamically
- code can be shared, but only the latest version provides shared data segments (areas)
- DISTOS implements neither virtual memory nor swapping; all processes must be loaded into real storage (this limits the number of processes)
- as a consequence, a process switch is very fast.

The general idea behind all the new OS designs is not to pack complete applications into a single process, but instead to structure them into a number of communicating processes. This not only reduces software complexity, but also bares a potential for parallelism. In that case, one should remember that beyond the regular and planned communication among those processes there is also a need to handle exceptions and to propagate them to other processes. A typical example in the database context is the abort or rollback of a running transaction. There are numerous concepts to

implement some sort of "software interrupts" and asynchronous routines to handle them in a process. Because DISTOS does not provide such a mechanism, we have learned how important it really is. To simulate it, every exchange of messages must be checked for the signalling of exceptions. This makes programming very cumbersome.

5.3 Which OS Mechanism Should Be Provided?

The decisive point in DBMS process configuration design is whether the OS supports different protection domains within an address space or not. If it does not support them, the asymmetric assignment (Fig. 5, but with m DBMS processes, $m < n$) will be the configuration of choice. This implies the need for shared data segments as a means of fast communication between AP processes and DBMS processes. Unfortunately, the large size of the segments often limits their number. Defining them in units of 2K pages instead of 64K segments seems more appropriate for the exchange of requests and responses. Furthermore, the DBMS should know about the client-server relationship and about the group of processes acting as servers. This information can help to improve scheduling compared to a simple priority scheme.

If the OS does support different protection domains within an address space, the linked-in DBMS configuration will be the winner. Compared to the asymmetric assignment it allocates only n processes instead of n+m. However, it should be noted that these processes have very large address spaces, which may cause problems in some OS environments. Even if the (reentrant) code of the DBMS can be shared, the data structures maintained separately for each process remain voluminous and slow down the process switches. As the DBMS itself, without the application programs, is anything else but small, this problem arises with the asymmetric assignment as well.

There are other characteristics that both configurations have in common. Whenever the DBMS runs as a number of instances in several processes, its performance critically depends on the availability of shared memory. If the database buffer and the global system tables cannot be placed in a shared data segment, the algorithms needed to implement the transaction concept (in particular the isolation) will be as complex and as expensive as in the case of distributed DBMS. Otherwise, there will be just a need for synchronization instead of a huge amount of data transfer.

A DBMS process has to acquire a (short-term) lock before it accesses the data structures in the shared memory. If the OS does not know about these locks, it cannot favor processes holding them. Other processes will, when activated, soon end up waiting for one of the locks (convoy phenomen [BGMP79]). It is awkward to tell the OS about all those locks. Instead a proposed solution [PR83] is based on a mechanism that allows a process to tell the OS that it is currently holding a lock and thus should not be interrupted, i.e. should get an additional time slice. In that the process "promises" to release the lock after a short period of time.

It has already been pointed out that both process configurations lead to a number of processes with rather large address spaces. The only way to reduce them in size is to distribute the functionality of the DBMS over more than one process. This would of course increase the total number of processes significantly. Not every function can be moved to a separate address space. Assigning the lock manager or the access path manager to its own process would create a tremendous overhead of process switches for every single DB operation. Even in an OS environment with an optimized process management the functional distribution of DBMS code seems to be far too expensive.

6. Communication

Communication among processes is required when they want to (or have to) cooperate. It serves for synchronization as well as exchange of information. As already mentioned in the previous chapter, it can be actually done either by the use of shared memory or by explicit transfer of messages [We84].

6.1 Requirements of DBMS

DBMS put some minimum requirements on the operations provided by the OS communication system for the exchange of messages between processes. These requirements define the basis on which elaborate protocols are implemented by the DBMSs themselves. According to Liskov [Li79] and Rothermel [Ro84] the operations "no-wait-send" and "asynchronous receive" are appropriate. They have to fulfill the following requirements:

- *Correctness*
  If a message is delivered, it must be complete and not be modified [Li83, Ro84].
- *Preservation of message sequence*
  The messages sent from one process to another are delivered in the same temporal order. This is important to realize other principles.
- *Flexible message format*
  Size and structure of the messages should not be predefined, as they vary considerably (e.g. locking messages, read and write requests to the file system including the responses, component calls)

Further requirements deal with addresses and establishing connections between processes:

- no separation of local and remote message exchange
- logical addresses that do not depend on the actual location of a process (allowing for a migration of processes to mask failure and to balance global system load)
- dynamic setup of connections (at runtime).

The latter three requirements provide the basis for tuning mechanisms, e.g. the replication of processes (Tandem's Guardian [Bo81]) and the extensibility of the system concerning the integration of new components (Guardian, R* [Li83]). In order to establish another subtransaction of a distributed transaction on a remote system, R* creates a connection (a "session") and initiates a process (an agent) in the remote system. In Guardian new processors extending a system can be utilized by relocating processes to them or by creating additional replicas of client and server processes. Logical addresses also allow for a switch from a primary to a backup server process (in case of processor failure) without the client noticing.

Further requirements on the communication systems are:

- *Support of multicast and broadcast*
  Higher-level protocols, e.g. the two-phase commit, use multicast. Several receiving processes must be asked to change the state of a specific transaction into 'prepared', 'aborted', or 'committed'. This should be done in a single call to the communication system.
- *Addressing groups of processes (service classes)*
  A sending process does no longer send its message to one other process. Instead it uses the address as a name for some service request, irrespective of which particular process answers it (server classes in Guardian, functional port classes in [Ro84]).

Apart from the requirements listed so far (they are important for distributed applications in general, not only for DBMS), there are some other requirements that supervene on the pure exchange of messages:

Secure communication to implement protection
Before it is checked, whether an operation may be executed, authentification is required for the one that wants to execute it. For this purpose, the communication partner must be identified to prevent that requests are accepted from someone who is not permitted. Furthermore, a disruption of a connection must be communicated to both partners, so that authentification is repeated when the connection is reestablished (R* [Li83]).

Support of client-server relationship
If a subtask of the job that one process has to do is performed by another process, this leads to a client-server relationship. In order to put the request and to deliver the response the two processes must communicate. The concept of remote procedure call (RPC) provides an abstraction from the organization of the relationships and the sending of messages. The communication system can support this [TR86].

When the client issues a request to the server, it makes an assumption about the semantics of the execution [Ho86]:

- 'Maybe' semantics:
  The request (the operation) is executed at most once (or not at all).
- 'At-least-once' semantics:
  The execution is always guaranteed, but no effort is made to prevent repeated execution.
- 'Only-once' semantics:
  The operation is always executed, and it is executed exactly once.

The communication system follows one of these three semantics. The application must know about it and, if it does not suffice, make additional effort to reach the semantics it needs.

The communication system can only make sure, that the request is delivered to the server according to the semantics of the RPC. It cannot check other conditions that the execution of an operation by the server has to fulfill, e.g. atomicity, consistency, and isolation. The application is responsible for them.

The communication system can perform only the message part of the RPC and handle data transmission failures. Hence, it provides just a message exchange pattern for the RPC.

Support of some primitives for transaction management [Ro84]
The communication system can provide operations for transaction management (start transaction, end transaction etc.), but it can only control the flow of messages and keep track of the relations among hierarchically nested transactions. For instance, it can generate messages to all processes participating in a distributed transaction, asking them to put the transaction into a safe state ('prepare to commit'). However, the communication system cannot guarantee or check correctness and completeness of the actions performed by the processes upon such a request [Sa84].

The communication system therefore can only offer primitives to start and end a transaction (CREATE_TA, CREATE_SUB_TA, ABORT, COMMIT) and to control the client-server relationship between transactions (WORK, WORK & COMMIT). The management of relations among the transactions must then be managed by the communication system. The resulting information and message must not get lost or be falsified by a failure of the communication system or on one of the sites.

All the other parts of transaction management that go beyond the organization of messages and the management of relations between transactions remain the application's task (done in the processes).

[Sa84] lists examples that support the 'end-to-end' argument. There are features that could be implemented by the application as well as by the communication system. The 'end-to-end' argument says that only the application has sufficient knowledge to implement them correctly and completely.

Application-oriented test and acknowledgements provide a high level of security without burdening the lower layers of the communication system.

Cooperation using shared data

Cooperation of processes using shared data is another form of process interaction [We84]. It has been discussed in section 5 how DBMS can use it. Shared data segments can be regarded as a result of non-disjoint or overlapping address spaces. Processes must be synchronized before accessing those segments. This is done by the applications themselves using the OS synchronization primitives (e.g. semaphores). Of course, protection is required to prevent unauthorized processes from accessing the data segment, i.e. connecting its address space to it.

Shared data can be used to speed up message exchange between processes on the same site. It cannot be used for communication between different sites. Nevertheless, it is our opinion that the user, i.e. the programmer of a complex distributed application, wants to choose among shared memory and message passing, because the philosophy of both concepts is different. And there are uses for both.

6.2 Communication Concepts of Current OS

As in the previous chapters we concentrate on the OS that we are familiar with. There is one representative for each generation of OS: BS2000 as an 'old' commercial OS, UNIX and GUARDIAN as relatively 'modern' OS, and DISTOS as our local experimental system.

DISTOS

It has been mentioned in section 5 that in DISTOS teams are the unit of distribution. Communication between processes inside a team use other mechanisms than the communication between teams: Processes of a team communicate by using monitors and semaphores. The latest version also supports shared data in a so-called area. Processes of different teams can communicate with typed messages. The types must be defined statically. The operations to send and receive these messages are executed synchronously. Teams are interconnected by logical busses. Broadcast and multicast are supported.

UNIX

UNIX now provides a flexible concept for message exchange between processes, the so-called sockets. There is no difference between communication on a system and across system borders. Connections can be established (stream sockets), but there is also the possibility for connectionless communication (datagram sockets). A particular implementation of UNIX supports the remote procedure call as well (SUN-RPC). Cooperation using shared data is offered in some implementations (System V, SUN), but is then of course limited to processes on the same machine.

BS2000

BS2000 uses different concepts for local and remote communication. The local 'inter-task communication' (ITC) is equipped with 'no-wait send' as well as an asynchronous receive. In addition the operations can define asynchronous procedures ('contingency routines') that are executed when specific events occur, e.g. the receipt of a message. The remote communication must use the 'data communications access method' (DCAM). A DCAM application serves as a logical network address and can be shared by a number of processes (server class).

Cooperation using shared data is supported (Common Memory Pool). Protection is not very strong. Access can only be restricted to the processes with the same user identification. If this is not done, everyone that knows the name of the pool can access it.

GUARDIAN (Tandem)

Communication is solely based on messages. There is no cooperation using shared data. Addressing in fact uses logical names for processes independent of their location. Connections are established dynamically. There is no difference between local and remote communication. The same primitives are used for process communication and for access to external devices ('everything is a process').

6.3 Which OS Mechanisms Should Be Provided?

Most of the points named above are already covered by one OS or another. Of course, one would like to have an OS that covers all of them. Despite all the comfortable, application-oriented features one should make sure that more primitive operations are available to the applications as well. This is necessary to implement higher-level protocols efficiently according to the 'end-to-end' arguments. The OS should also support the exchange of data through shared memory. In particular, this is valid for advanced hardware and system architectures that enable overlapping address spaces even across system boundaries.

7. Transaction Concept

The ACID principle sketched in section 2 was characterized as a fundamental issue of DBMS. Its implementation is primarily concerned with isolation of user activities and handling of failures. In the case of DBMS, it is accepted wisdom that atomic transactions are the correct unit of synchronization and recovery. It is claimed that this is also true for distributed systems [PWP85]; hence, in particular (distributed) OS should support them.

7.1 Requirements of DBMS

Let us discuss the DBMS needs in some more detail.

Synchronization

Concurrent activities executed by the DBMS on behalf of its users must be synchronized in order to guarantee serializable schedules for transactions [Gr78]. The concurrency control (CC) component is in charge of this task. While a large number of CC algorithms have been proposed in the literature, only little knowledge and practical experience is available on their performance (except for locking). It is (currently) safe to say that the method of choice for CC is locking [Pe86]. However, the following important aspects should be considered carefully:

- granularity of locks
- use of lock hierarchy
- handling of hot spot data elements.

Page locking often used for simplicity of implementation and lock management may be sufficient for most kinds of data; however, it does not suitably support concurrent access to objects with moderate traffic frequencies such as catalogs, addressing tables, index structures (B^*-trees). Hence, selected use of smaller granularities (e.g. record or entry locks) on such objects may greatly improve concurrent activities.

Locking of disjoint partitions of a given size is insufficient in most applications for performance reasons. Apart from concurrency among transactions, locking overhead of a transaction is strongly affected by the choice of lockable units (space for lock control blocks, time to request and release locks). As a consequence, there exists an implementation tradeoff of increased concurrency using fine lockable units and higher cost for lock management. Assume, for example, a sequential scan of a file with 10^5 records distributed over 5000 blocks. A file lock would cause one lock request, whereas block or record locks would require 5000 or 10^5 lock requests, respectively. To adjust the lock granules to a transaction's need, an appropriate hierarchical locking scheme was proposed by Gray [Gr78]. It may be used for either locking a few items using a fine lockable unit or locking larger sets of items with larger lock granules.

Lock requests and releases must obey a strict two-phase locking protocol [EGLT76], that is, the 'growing phase' is spread over the entire transaction, whereas the shrinking phase is concentrated to phase 2 of the COMMIT protocol. The CC component has to enforce such a strict protocol for read as well as for write requests to guarantee 'repeatability of reads' (level 3 consistency) and to avoid situations like recursive backout in case of transaction aborts or system crashes.

To optimize access to very active data items sometimes called 'hot spot' data elements, tailored

mechanisms are required. Since transactions refer to such data with high frequencies, use of a two-phase locking protocol would serialize transaction processing at such points of contention. Hence, semantic knowledge of transaction operations is necessary to provide efficient solutions without compromising serializability (e.g. escrow mechanism for commutable operations on aggregate field data [ON86]).

Logging and recovery

Dealing with DB recovery requires a clear understanding of

- the type of failure the DB has to cope with, and
- the notion of consistency that is assumed as a criterion for describing the state to be reestablished.

The traditional DB failure model includes transaction failure, system failure, and media failure as well as site failure in the distributed case [Gr81]. For more detailed disussion see [HR83a]. The state to be reestablished after successful recovery, e.g. transaction recovery or system restart, is clearly implied by the ACID principle or the 'all-or-nothing' nature of a transaction. A database is consistent if and only if it contains the results of successful transactions (such a state is called transaction consistent or logically consistent). Hence, a transaction failure implies rollback of all its effects. A system or site crash requires that all effects of incomplete transactions must be removed from the DB. On the other hand, modifications of all successful transactions must survive any failure. Thus, the target state of a successful recovery is the most recent transaction-consistent state.

To enable reliable (and fast) recovery a number of mechanisms must be provided. To achieve atomicity of a transaction, the so-called two-phase COMMIT protocol [Gr78] must be supported. It requires the synchronous output of enough REDO information (logging) for the correspdoning transaction to a safe place, e.g. disk. Such a mechanism is sometimes called a force-write. Reliability concerns often lead to duplex logging. UNDO information must be force-written to a safe place before a dirty data page (with uncommitted information) is replaced in the buffer, when update-in-place is used on disk (write ahead log (WAL) principle). Furthermore, a checkpoint scheme should be supported, that is, to guarantee that modified data pages are forced to disk in a controlled manner to limit the costs for partial REDO during crash recovery [HR83a].

The collection of log information burdens the normal system operation; nevertheless, it must be sufficient to survive all types of failures mentioned. Since system operation benefits from minimization of I/O, small log granules should be chosen; entry or record logging allows for buffering of log information. On the other hand, page logging (often used for simplicity reasons) produces an enormous amount of log data and I/O. Moreover, page logging implies at least page locking [HR83a], that is, the lock granule must cover the log granule. Our discussion of the synchronization issue, however, has identified the need for small lockable units.

Nested control structure

So far, the transaction has been introduced as the only unit of control in a DBMS as far as synchronization and recovery is concerned. When executing more complex transactions, it turns out that single level transactions do not obtain optimal flexibility and performance. Especially in distributed systems, it is highly desirable to have more general control structures supporting reliable and distributed computing more effectively. More decomposable and finer grained control of concurrency and recovery would support intra-transaction parallelism and intra-transaction recovery control. An explicit control structure within a transaction facilitates system modularity, distribution of system implementation as well as flexible use of implementation techniques.

The concept of nested transactions [Mo81] provides the ability to invoke transactions from within transactions. These subtransactions are atomic and isolated (but they need not be consistent and durable). Consistency may be preserved or controlled by some ancestor transaction in the composition hierarchy. Persistence can only be guaranteed by the top-level transaction since the results of subtransactions are removed whenever the enclosing transaction is undone.

With the ability to nest transactions, distributed system design, exploitation of parallelism, use of small recovery granules, etc. are simplified. Programmers are free to compose existing transaction modules just as procedures and functions are composed in programming languages [MMP83].

7.2. Transaction Support of OS

As already mentioned, transactions need not necessarily be linked to the framework of DBMS, but could be useful for other types of applications, too [SS83, We86]. Hence, a common transaction management facility within the OS would be desirable as generally available service. However, conventional OS such as MVS or BS2000 neither have the concept of transaction nor do they provide tailored mechanisms for mapping transactions to the available OS facilities. Such deficiencies are considered as 'natural' since these systems were usually developed before the notion of transaction was formalized [EGLT76].

We have identified two interfaces in our DBMS mapping hierarchy ((II) and (IV)) which could serve for OS-DBMS cooperation. Let us investigate interface (II) first and its consequences for transaction support. OS have some (rudimentary) concurrency control at the file/block level. Its use for controling concurrent DBMS activities would be disastrous mainly due to large granularities and due to lacking adjustment to DB objects. Let us just quickly mention some more problems:

- Even if a locking hierarchy (file-block) would be supported, it would not be sufficiently refined.
- Handling of hot spot data or use of semantic knowledge would be impossible.

- OS lockable units must have a unique name upon which all processes agree, DBMS objects may have multiple names (synonyms) or may be referred to by predicates.
- Mapping of predicates or key intervals to lockable units would probably result in very coarse approximations and ponderous procedures.
- Every object reference requires an OS kernel call (even if the object is already locked for the requesting transaction), if synchronization is solely built upon OS locking.

This (non-exhaustive) list of drawbacks may convince the reader that an efficient locking service cannot be implemented by the OS below interface (II). Similar arguments apply for logging and recovery services. Since objects within a page are not known at interface (II), an OS logging service would have to use page logging for UNDO and REDO information. Although a number of low-level optimizations (chained I/O, central log service for all DBMS processes, etc. [We86]) may be utilized, such an approach may not be feasible for performance reasons.

Let us summarize our arguments concerning OS transaction management: Because of the block orientation of the discussed OS-DBMS interface, implementation of the most essential transaction services would imply

- block level synchronization for all shared data types
- block level logging for all recoverable data types and, as a consequence, block level recovery for all types of failures.

Such features incorporate low-performance solutions in a DBMS context. Therefore, transaction services or even integrated OS transaction management are not recommended at such a low level.

The record-level interface (approach (IV)) is much more appropriate for using an integrated OS transaction management. The arguments raised so far do not apply anymore since locking as well as logging/recovery could be based on entries or records which are proven to lead to efficient solutions [Gr81, Hä87]. Since performance arguments play the dominant role in all design considerations, we can state the following observation (or commandment): If full transaction management (including CC and recovery services) should be integrated into an OS, then the record-level interface (or even a higher one) must be chosen as the OS-DBMS interface!

However, even with such an interface not all desirable transaction management properties could be satisfied:

- The hot spot problem still needs special mechanisms.
- CC based on application semantics (known in higher DBMS layers) requires (complicated) DBMS-OS interaction.
- Logical logging (operator logging) allocated to DBMS layer 4 would require OS-DBMS coordination, e.g. for synchronizing the built-in mapping redundancy (shadow-page mechanism, checkpoints) with the forced log writes.

Note, there are a lot of interdependencies among CC, logging/recovery, and propagation control [HR83a]. Therefore, it is not advisable to arbitrarily distribute transaction management functions across OS and DBMS components. To say the least, such implementations tend to become very complex. Hence, when transaction management is supported by the OS, it either can not offer all desirable options or features or it is divided into clean functions and responsibilities guaranteeing reliable and efficient cooperation.

Tandem's OS GUARDIAN provides a generic disk process (DP2) which is used for the database manager of a DB partition [TSR85]. DP2 could be considered as an example for a record-level OS-DBMS interface. Locking is done by DP2 for a DB partition. Other transaction services (logging, two-phase COMMIT, deadlock detection, etc.) are implemented by other process types belonging to the ENCOMPASS system (AUDIT, BACKOUT, TCP, etc. [Bo81]). It should, however, be mentioned that DP2 has been adjusted to DBMS needs (multi-tasking as a consequence to the lack of shared memory).

Nested transaction structures are not supported at all by OS mechanisms. As shown above, it is even hard to cope with 'flat' transactions. Typically, the DBMS code is mapped to a number of OS processes (1-n servers) as discussed in section 5. A direct OS transaction support would imply that each active transaction had to be identified by some unique criterion, that is, it would be natural from the OS point of view to use process ID's for this purpose. As a necessary consequence, 2n processes would have to be sacrificed for this type of mapping (a very expensive solution). From a performance point of view, the allocation of a server pool of m DBMS processes would guarantee satisfactory results [HP84], but would create problems when transaction support is tied to processes [We86]. Another process structure (called the 'ideal concept' in section 5) seems to be much more advantageous, for performance reasons as well as for using direct OS transaction support.

It should be mentioned that OS transaction management is a first-class objective in many ambitious research projects in the OS/DBMS area including ARGUS [Li84], LOCUS [Po81] and TABS [Sp85]. Their emphasis is on extending the OS for effectively organizing and maintaining distributed programs where DBMS programs are only a special case. Hence, mechanisms for supporting nested transaction structures have been made available. To our knowledge, there does not exist broad system experiences for demanding practical applications which would make a thorough treatment mandatory. On the other hand, a detailed discussion of all (important) design and implementation attempts would blow up the framework of our considerations.

In the following, we investigate some transaction-related issues as candidates for OS integration which seem important from our background and DBMS experiences.

7.3 What OS Mechanisms Should Be Provided?

As indicated in the previous sections, appropriate OS transaction support depends on the interface chosen for the applications. Block-oriented interfaces would probably obtain satisfactory performance only for low-concurrency applications with little logging demand. For DBMS, record-oriented OS interfaces are a prerequisite in the case that OS services should be used efficiently for CC and logging/recovery. However, such a solution does not seem to be mandatory for DBMS; it has been only chosen by very few practical systems. High-performance solutions require more degrees of freedom; hence, a larger potential for optimization may be utilized when the DBMS has complete and efficient control over the critical functions. (In a multi-server DBMS, shared memory must be available for global system data - see section 5.) In particular, CC enriched by semantic knowledge [SS84] or based on special concepts (escrow mechanism [Re82, ON86]) allows for more powerful and effective implementations. As far as logging is concerned, the DBMS may collect log information tailored to the special recovery needs.

On the OS side, we prefer improved mechanisms for transaction coordination and structuring when transactions are running in multiple servers or even in a distributed system. For example, log information could be written as variable-length byte strings to a shared buffer. The OS could provide a fast sequential I/O operation (e.g. chained I/O) and could force filled buffers to the log file on disk. Of course, it must be guaranteed that a transaction cannot commit before its REDO information has reached a safe place. Such a mechanism could be generalized to a so-called group commit. The OS defers committing transactions (processes) until the block or sequence of blocks containing their log data are filled with log information and forced. (Let us assume that such a delay takes place in intervals of 100 ms and does not affect response time.)

If storage redundancy such as mirrored disks is made available, it should be controlled by the OS which can efficiently utilize read optimization, etc. However, we do not advocate a block-oriented stable storage mechanism obtained by two consecutive (synchronous) I/Os. We are of the opinion that such a feature is too expensive; log-based solutions are much cheaper.

Furthermore, support for transaction nesting seems to be important. Subtransactions may be executed in multiple processes at the same or at different sites. From our point of view, the OS should be responsible for reliable data transfer, location transparency and request/answer matching (bookkeeping) as well as detection of transaction/system failures. Essential OS tasks could be the coordination of transaction abort or commit [Ro85]. For example, if a (sub-)transaction fails, it must be rolled back which implies the following:

- an 'ABORTED' message has to be sent to the parent transaction
- 'ABORT' messages have to be propagated to all descendents.

An OS bookkeeping component should keep a list of transactions/sites and guarantee delivery of ABORT messages despite failures. A receiving OS component will create new ABORT messages according to its list and forward them to the next lower subtransaction until finally the leaves of the transaction tree are reached. Since subtransactions to be aborted may have already committed, a special DBMS or OS component having access to the log information has to take care of the requested rollback.

A distributed transaction commit protocol for nested transactions is a very complex operation; with a coordinator (TL-transaction) and n subordinate transactions, 4n messages and 2(n+1) log writes are necessary to complete the protocol (in the unoptimized case) [ML83]. The TL-transaction (outermost sphere of control) can only commit if all descendents agree to commit. A two-phase COMMIT protocol (2P) is initiated by the TL-transaction after the user has decided to commit. In a first phase, PREPARE messages are sent to the subordinates. After having received the votes from all subordinates, it initiates the second phase of the protocol. If all the votes are YES votes, it sends COMMIT messages to all the subordinates which respond with an acknowledgement (ACK message). The log writes mentioned are necessary to make the states of the 2P protocol fault-tolerant.

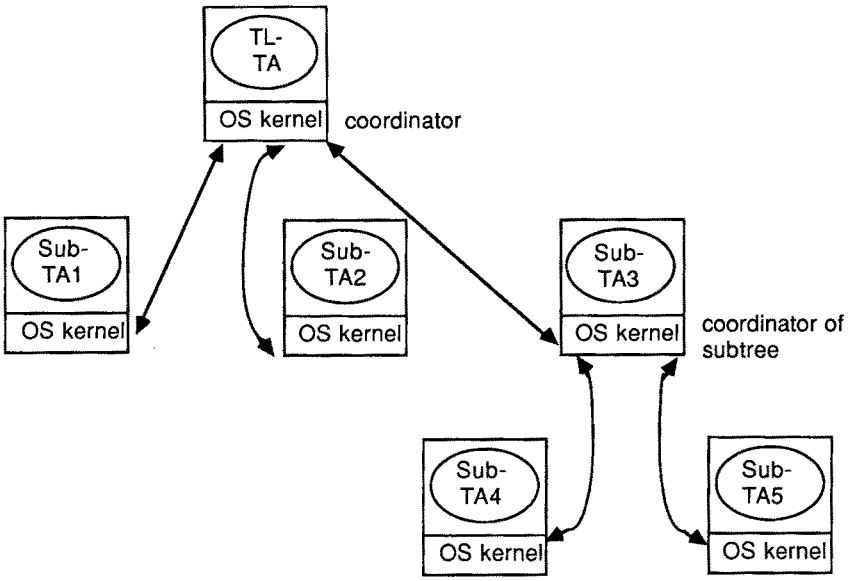
The complexities of this distributed protocol could be hidden from the DBMS by providing an OS mechanism for the exeuction of such a hierarchical protocol [Ro85]. As illustrated in Fig. 7a, OS kernel functions represent the coordinators (also for subtrees) in the hierarchical 2P protocol. They accept the COMMIT/PREPARE request and ask the participants known to them to prepare. Furthermore, they collect the votes, pass a subtree vote to the parent node, and make a decision at the root of the tree. In phase 2 they propagate the decision. Fig. 7b shows the interactions between kernel coordinator and participants; it becomes clear that the protocol interface is very simple:

- for the TL-transaction: COMMIT(TID) and COMPLETE/BACKOUT
- for a subordinate: PREPARE, VOTE YES/NO and COMPLETE/BACKOUT.

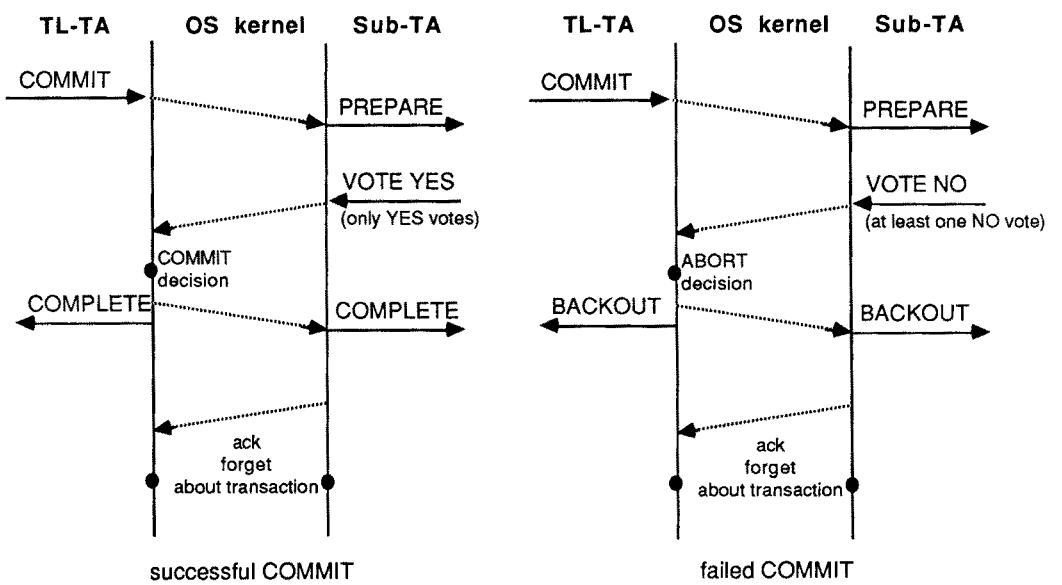
Of course, optimizations such as PRESUMED ABORT protocol or PRESUMED COMMIT protocol [ML83] could be implemented as the primitive COMMIT mechanism in the OS.

8. Conclusions

We have presented an investigation of OS support for DBMS where we identified file management, process structures, communication mechanisms and transaction management as main areas of interest. The focus of our paper has primarily been on discussing appropriate OS-DBMS interfaces thereby referring to a multi-level model which describes the mapping hierarchy of a DBMS architecture. Issues of distributed database management did not play a particular role in our subject because current distributed DBMS (e.g. R*) typically use in addition to communication primitives only OS services like centralized DBMS. In such systems, the 'global view of the database' is established



a.) Nested structure of an example transaction



b.) Interaction between kernel coordinator and participants

Fig. 7: Hierarchical two-phase COMMIT protocol as an OS kernel mechanism

by the top-most layers (see Fig. 1), that is, distribution of requests (query distribution, function shipping) is performed at the logical level aiming at large data or operation granules. A consequence of such an approach is the need to accomplish all desirable properties of distributed systems by the DBMS and not by the OS, e.g. location transparency, failure transparency, etc.

Let us reconsider our most important findings in all four functional areas. File management should provide a file concept guaranteeing cheap maintenance and very fast access. Extreme flexibility may be sacrificed for moderate growth flexibility and cost (extent table mechanism and predeclaration of extent to be allocated). Moreover, a general cluster mechanism is needed in order to support clustering of arbitrary blocks (chained I/O). To that, the conventional file interface should be extended with respect to set-oriented operations on blocks ("block set" and "block sequence") as well as different block sizes.

Process management should be flexible enough to avoid a second level of scheduling. This leads to an asymmetric assignment of multiple server processes or, if separate protection domains inside an address space are available, to a linked-in DBMS. For performance reasons, the later is reagarded as the ideal concept. Both configurations rely on the efficient management of rather large processes, i.e. a fast process switch. Due to the nature of cooperation among DBMS processes, shared memory is indispensible for the DB buffer and the global system tables. Furthermore, the OS should allow a process holding a short-term lock to go on and to release that lock before a preemption takes place.

Communication can be implemented on the basis of no-wait send and asynchronous receive. According to the end-to-end arguments anything else needed to obtain a safe communication can be done by the DBMS itself more efficiently. This is also valid for the remote procedure call. Broadcast and multicast are needed to simplify transaction commit protocols.

Essential transaction management services such as locking and logging/recovery cannot be supported efficiently by the OS at the level of block-oriented objects. A record-oriented OS-DBMS interface would satisfy many performance requirements of highly concurrenct DBMS applications. However, since often semantic application knowledge should be exploited, implementation of those services within the DBMS is preferable. Nevertheless, a number of important mechanisms could be provided by the OS: fast logging support, group commit, two-phase commit protocol.

As already mentioned, distributed DBMS were not typically built on top of distributed OS (with the partial exception of Tandem/ENCOMPASS). Thus, all mechanisms dealing with the distributed nature of the data and processors were implemented by the DBMS. This situation may change in the near future, since ongoing research attacks the problem of designing and implementing a distributed database operating system. However, it is hard to believe that a DBMS can derive optimal execution plans when location transparency is achieved by the OS. To name only a single project, GENESIS [PWP85] uses LOCUS as a basis which already provides atomic commit, automatic updates to

distributed replicated files, a network transparent name server, remote tasking, and inter-process communications. GENESIS improved the LOCUS transaction mechanisms, tailored to support distributed transaction management and implemented a flexible record-level locking facility. Although a lot of conceptual work has been done on nested transactions and distributed transaction management, we believe that still much effort is needed to investigate efficient implementations for these concepts and functions - the most critical task of future OS-DBMS research.

Literature

As76 Astrahan, M.M., et al.: SYSTEM R: A Relational Approach to Database Management, in: ACM TODS, Vol. 1, No. 2, June 1976, pp. 97-103.

BGMP79 Blasgen, M., Gray, J., Mitoma, M., Price, T.: The Convoy Phenomen, in: ACM Operating Systems Review, Vol. 13, No. 2, April 1979, pp. 20-25.

BKT85 Brown, M.R., Kolling, K.N., Taft, E.A.: The Alpine File System, in: ACM TOCS, Vol. 3, No. 4, November 1985, pp. 261-293.

Bo81 Borr, A.: Transaction Monitoring in ENCOMPASS, in: Proc. 7th Int. Conf. on VLDB, Cannes, 1981, pp. 155-165.

CODA78 Report of the CODASYL Data Description Language Committee, in: Information Systems, Vol. 3, No. 4, 1978, pp. 247-320.

DPS86 Deppisch, U., Paul, H.-B., Schek, H.-J.: A Storage System for Complex Objects, in: Proc. Int. Workshop on Object Oriented Database Systems, Asilomar, 1986, ed. by K. Dittrich, U. Dayal, pp. 183-195.

EGLT76 Eswaran, K.P., Gray, J.N., Lorie, R.A., Traiger, I.L.: The Notions of Consistency and Predicate Locks in a Database System, in: Comm. ACM, Vol. 19, No. 2, 1976, pp. 624-633.

Gr78 Gray, J.: Notes on Database Operating Systems, in: Operating Systems - An Advanced Course, Lecture Notes in Computer Science 60, ed. by Bayer, R., Graham, R.M., Seegmueller, G., Springer 1978, pp. 393-481.

Gr81a Gray, J., et al.: The Recovery Manager of the System R Database Manager, in: ACM Computing Surveys, Vol. 13, No. 2, 1981, pp. 223-242.

Gr81b Gray, J.: The Transaction Concept - Virtues and Limitations, in: Proc. 7th Int. Conf. on VLDB, Cannes, Sept. 1981, pp. 144-154.

Fr87 Franz, B.: Konzeption und Implementierung eines Dateisystems für das DISTOS-Betriebssystem, Master's Thesis, University of Kaiserslautern, 1987.

HMMS87 Härder, T., et al.: PRIMA - a DBMS Prototype Supporting Engineering Applications, in: Proc. 13th Int. Conf. on VLDB, Brighton, 1987, pp. 433-442.

Hä79 Härder, T.: Die Einbettung eines Datenbanksystems in eine Betriebssystemumgebung, in: Datenbanktechnologie, Proc. II/1979 German Chapter of the ACM, ed. by

Niedereichholz, J., Teubner 1979, pp. 9-24.

Hä87 Härder, T.: Some Selective Performance Problems of Database Systems, in: Proc. Int. Conf. on Measurement, Modelling and Evaluation of Computer Systems, Erlangen, Sept. 1987, IFB 154, pp. 294-312, (invited lecture).

Ho86 Hofmann, F.: Remote Procedure Call, das aktuelle Schlagwort, in: Informatik-Spektrum, Vol. 9, No. 4, 1986, p. 308.

HP84 Härder, T., Peinl, P.: Evaluating Multiple Server DBMS in General Purpose Operating System Environments, in: Proc. 10th Int. Conf. on VLDB, Singapore, 1984, pp. 129-140.

HR83a Härder, T., Reuter, A.: Principles of Transaction-Oriented Database Recovery, in: ACM Computing Surveys, Vol. 15, No. 4, Dec. 1983, pp. 287-318.

HR83b Härder, T., Reuter, A.: Concepts of Implementing a Centralized Database Management System, in: Proc. Int. Computing Symp. 1983 on Application Systems Development, German Chapter of the ACM Report No. 13, Teubner 1983, pp. 28-59.

IBM Information Management System, General Information Manual, IBM Publications No. GH 20-1260, IBM Corp. White Plains, New York.

Ko87 Koch, R.: Datenverwaltungssystem BS2000, Siemens AG, Munich1987.

Li79 Liskov, B.: Primitives for Distributed Computing, in: Proc. 7th Symp. on Operating Systems Principles, Dec. 1979, pp. 33-42.

Li83 Lindsay, B., et al.: Computation and Communication in R*: A Distributed Database Manager, IBM Res. Rep. RJ3740, San Jose, June 1983.

Li84 Liskov, B.: The ARGUS Language and System, in: Distributed Systems: Methods and Tools for Specification, An Advanced Course, Lecture Notes in Computer Science, Springer 1984.

Lo77 Lorie, R.A.: Physical Integrity in a Large Segmented Databases, in: ACM TODS, Vol. 2, No. 1, March 1977, pp. 91-104.

McK84 McKusick, M.K., et al.: A Fast File System for UNIX, in: ACM TOCS, Vol. 2, No. 3, August 1984, pp. 181-197.

ML83 Mohan, C., Lindsay, B.: Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions, in: Proc. 2nd ACM SIGACT/SIGOPS Symp. on Principles of Distributed Computing, Montreal, Canada, Aug. 1983.

MMP83 Mueller, E., Moore, J., Popek, G.J.: A Nested Transaction Mechanism for LOCUS, Proc. 9th Symp. on Operating Systems Principles, Bretton Woods, NH, Oct. 1983.

Mo81 Moss, J.E.B.: Nested Transactions: An Approach To Reliable Computing, Ph.D. Thesis, M.I.T. Report MIT-LCS-TR260, Laboratory of Computer Science, 1981.

MVS80 OS/VS2 MVS Overview, IBM Corp., Poughkeepsie, 2nd ed. (May 1980), Order No. GC28-0984-1.

Ne85 Nehmer, J., et al.: The Multicomputer Project INCAS - Objectives and Basic Concepts, University of Kaiserslautern, SFB124, Report No. 11/85, 1985.

Ne87 Nehmer, J., et al.: Key Concepts of the INCAS Multicomputer Project, in: IEEE Transactions on Software Engineering, Vol. SE-13, No. 8, 1987, pp. 913-923.

ON86 O'Neil, P.E.: The Escrow Transactional Method, in: ACM TODS, Vol. 11, No. 4, Dec. 1986, pp. 405-430.

Org72 Organick, E.I.: The Multics System, MIT Press, Boston 1972.

Pe86 Peinl, P.: Synchronisation in zentralisierten Datenbanksystemen - Algorithmen, Realisierungsmöglichkeiten und quantitative Bewertung, Ph.D. Thesis, University of Kaiserslautern, 1986.

Po81 Popek, G.J., et al.: LOCUS: A Network Transparent, High Reliability Distributed System, Proc. 8th Symp. on Operating Systems Principles, Pacific Grove, CA., Dec. 1981.

PR83 Peinl, P., Reuter, A.: Synchronizing Multiple Database Processes in a Tightly Coupled Multiprocessor Environment, in: ACM Operating Systems Review, Vol. 17, No. 1, Jan. 1983, pp. 30-36.

PWP85 Page, T.W., Weinstein, M.J., Popek, G.J.: Genesis: A Distributed Database Operating System, Proc. ACM SIGMOD'85 Conf., pp. 374-387.

QSP85 Quarterman, J.S., Silberschatz, A., Peterson, J.L.: 4.2BSD and 4.3BSD as Examples of the UNIX System, in: ACM Computing Surveys, Vol. 17, No. 4, December 1985, pp. 379-418.

Re82 Reuter, A.: Concurrency on High-Traffic Data Elements, in: Proc. Conf. on Principles of Database Systems, Los Angeles, CA, 1982, pp. 83-93.

RN84 Rothermel, K., Neuhold, E.J.: Mechanisms Supporting Application Layer Protocols for Distributed Database Systems, in: Final Technical Report, European Research Office of the U.S. Army, London, 1984.

Ro84 Rothermel, K.: A Communication Model for Transaction Oriented Applications in Distributed Systems, in: Proc. 17th Annual Hawaii Int. Conf. on System Sciences, 1984, pp. 88-95.

Ro85 Rothermel, K.: Kommunikationskonzepte für verteilte transaktionsorientierte Systeme, Ph.D. Thesis, Institut für Informatik, Universität Stuttgart, Nov. 1985.

Sa84 Saltzer, J.H., et al.: End-to-End Arguments in System Design, in: ACM TOCS, Vol. 2, No. 4, Nov. 1984, pp. 277-288.

Si77 Siwiec, J.E.: A high-performance DB/DC system, in: IBM Systems Journal, Vol. 16, No. 2, pp. 169-195.

Si87 Sikeler, A.: Buffer Management in a Non-Standard Database System, SFB124 Research Report, University of Kaiserslautern, 1987, in preparation.

SL76 Severance, D.G., Lohman, G.M.: Differential Files: Their Application to the Maintenance of Large Databases, in: ACM TODS, Vol. 1, No. 3, Sept. 1976, pp. 256-267.

Sp85 Spector, A.Z.: The TABS Project, in: Database Engineering, Vol. 8, No. 2, June 1985.

SS83 Spector, A.Z., Schwarz, P.M.: Transactions: A Construct for Reliable Distributed Computing, in: ACM Operating Systems Review, Vol. 17, No. 2, 1983.

SS84 Schwarz, P.M., Spector, A.Z.: Synchronizing Shared Abstract Types, in: ACM TOCS, Vol. 2, No. 3, 1984, pp. 223-250.

St76 Stonebraker, M., et al.: The Design and Implementation of INGRES, in: ACM TODS, Vol. 1, No. 3, Sept. 1976, pp. 189-222.

St81 Stonebraker, M.: Operating System Support for Database Management, in: Comm. ACM, Vol. 24, No.7, July 1981, pp. 412-418.

St84 Stonebraker, M.: Virtual Memory Transaction Management, in: ACM Operating Systems Review, Vol. 18, No. 2, April 1984, pp. 8-16.

Svo85 Svobodova, L.: Summary of the 9th SIGOPS Workshop: Operating Systems in Computer Networks, in: ACM Operating Systems Review, Vol. 19, No. 2, June 1985.

Tr82 Traiger, I.L.: Virtual Memory Management for Database Systems, in: ACM Operating Systems Review, Vol. 16, No. 4, October 1982, pp. 26-34.

TR86 Tanenbaum, A.S., Renesse, R.V.: Distributed Operating Systems, in: ACM Computing Surveys, Vol. 17, No. 4, Dec. 1986, pp. 419-470.

TSR85 Tandem System Review: Selected Papers, Vol. 1, No. 2, June 1985.

We84 Wettstein, H.: Architektur von Betriebssystemen, Hanser, Munich 1984.

We86 Weikum, G.: Pros and Cons of Operating System Transactions for Data Base Systems, Proc. ACM/IEEE Fall Joint Computer Conf., Dallas, Nov. 1986.

WNP87 Weikum, G., Neumann, P., Paul, H.-B.: Konzeption und Realisierung einer mengenorientierten Seitenschnittstelle zum effizienten Zugriff auf komplexe Objekte, in: Proc. GI Conf. on Database Systems in Office, Engineering, and Science Environments, Darmstadt, ed. by H.-J. Schek, G. Schlageter, Informatik-Fachberichte Bd. 136, Springer 1987, pp. 212-230.