Evaluating multiple server DBMS in general purpose operating system environments

Theo Härder
Peter Peinl

Dept. of Computer Sciences, Univ. of Kaiserslautern, West Germany

Abstract

Several concepts and problems in the integration of a database management system (DBMS) into a general purpose operating system are investigated. In particular, isolation and access control, the cooperation between application and DBMS processes as well as the synchronization of multiple DBMS processes are discussed. Basic solutions for the partitioning of DBMS functions to operating system processes are examined and two multiple server DBMS solutions are evaluated by a detailed simulation model. Quantitative results concerning the performance characteristics of those solutions, are presented, in particular for the overall throughput, process switching overhead and the influence of certain high traffic locks within the DBMS.

1. Introduction

Special database operating systems are generally not available [Gr78]. Therefore, database management systems (DBMS) have to be integrated in an environment offered by general-purpose operating systems (OS). DBMS typically run on top of the OS as normal application processes using the standard services available from the OS such as basic file access methods, process management, virtual memory, communication facilities, etc. which are not tailored to the specific needs of a DBMS. Application programs are also executed as normal processes (AP). Some of the problems are discussed in detail in [St81, Hä79, TM82].

Since processes are typically designed to run one application program in full isolation by virtually providing a single user machine (in timesharing mode), they often offer only unsuitable or insufficient means for the case where user processes have to closely cooperate with each other. This property, however, characterizes the situation where a number of application processes has to communicate with the DBMS thereby requesting and exchanging data. On the other hand, the DBMS itself can be embodied for performance reasons by multiple DB processes (servers) which have to manipulate the DB data in a synchronized manner.

For this given situation, a number of necessary requirements and useful properties of embedding a DBMS in an OS environment are discussed. Some process structures for DBMS integration are proposed. These solutions are based on available hardware and OS architectures which are not originally designed for efficiently supporting the cooperation of DBMS and application programs. Two different multiple server solutions are investigated in detail. A simulation model is described and numerical results are presented for a number of characteristic performance criteria.

2. Problems of DBMS integration

Isolation and access control

Usually the basic protection concept realized by OS and hardware primitives provides strict separation of processes often achieved by means of virtual address spaces. Such a spacial separation prevents any kind of interference. From an OS point of view, the process (execution domain) is considered to be the unit of scheduling and the unit of protection. This concept, however, does not permit the straightforward solution to run the DBMS as a subroutine of the AP (inlinked solution) which is the most efficient control structure for cooperation. Since the access rights of a program - controlled by the OS - are associated with the address space (protection unit) in which the program is executed and not with the particular program itself, the AP could act as the DBMS and directly manipulate the database (Trojan horse problem). Furthermore, other ways to circumvent protection mechanisms would be possible, e.g. the AP could forge the access control information of the DBMS and then use normal DB-calls to get the desired information. As long as an OS does not allow to establish different adjusted protection domains with controlled transfer of access rights within one execution domain, the DBMS cannot guarantee any given access control policy. Therefore, strict isolation of AP and DBMS is a prerequisite to enforce a distinct access control policy like least priviledge principle.

Proceedings of the Tenth International Conference on Very Large Data Bases.

Singapore, August, 1984

Cooperation between processes

While usually providing strict isolation of processes, OS generally do not support the direct cooperation of concurrent processes in different execution domains. As a consequence, communication has to be carried out via the OS by primitives like SIGNAL and WAIT. It is a well known fact that interprocess communication (IPC) is very expensive [Hä79]; a process switch typically requries several thousand instructions (~ 5000).

Each DB-call issued by AP synchronously implies at least two process switches. This is particularly bad if the requested DBMS service is short. For example, in CODASYL systems a great share of the executed DBMS services consists of very short operations like FETCH NEXT, IF MEMBER etc. with only a few hundred or less instructions. In these situations more than 90% of the total path length is due to process switching. Hence, process switches should be avoided as far as possible, e.g. by more powerful DB-operations or by suitable structures for AP-DBMS cooperation.

Another aspect is important for efficient process cooperation. If there is no common storage area available for AP and DBMS, parameters and results of each DB-call have to be exchanged via special areas of the OS. Such a MOVE mode requires to copy the transferred data twice in each direction. The use of common areas in main (virtual) memory would greatly facilitate the exchange of data (LOCATE mode).

Ideally, it should be possible for each APi to share a variable length area with the DBMS for private data exchange. Since sharing of variable length items between address spaces is usually not supported by the hardware, shared use of storage has to take place in units of pages (or units of multiple pages). It seems to be sufficient for isolation and fast data exchange to allocate a separate page for each pair APi-DBMS.

The possiblity to allocate shared areas (segments) for families of processes allows for a greater flexibility when the DBMS is embedded in an OS environment. DB-functions can be distributed among processes or several activations (instantiations) of the DBMS code in different processes can be employed to run the given workload. Despite of a conceivable increase in parallelism, we have not considered a judicious distribution of function because of the tremendous IPC overhead expected. The latter idea, however, deserves further attention.

By using shared segments multiple DBMS activations can be installed in different address spaces as independent processes (as seen from the OS). Shared segments are necessary to hold global DBMS data including

- the system buffer (SB)
- the input queue (server queue)
- the log buffer
- free placement information, central administration tables, control blocks for locks and transactions and their related queue structures.

The latter structures are called global system tables (GST). The DBMS code itself can be put into a shared segment and executed in a reentrant manner by each of the DBMS processes saving storage space and enhancing substantially locality of reference which is particularly important in a paging environment.

Synchronization of multiple DBMS processes

Scheduling of processes by the OS is preemptive, since they are considered to be independent. Hence, operations on the global DBMS data can be interrupted at arbitrary times. Therefore, access to these data has to be synchronized to avoid lost updates and inconsistent reads.

How can synchronization of independent processes on common data be achieved? The use of OS services (SVC for lock requests) is ruled out for performance reason. Access to these global resources is characterized by high frequency of synchronization events and relatively short lock duration. Therefore, synchronization has to be performed directly between the participating processes by appropriate protocols and by observing a given discipline. Usually, machine instructions like TS (Test and Set) or CS (Compare and Swap) [IBM83] can be used to implement semaphores and corresponding queue structures [GW79]. Hence, access to a global resource can be protected by a critical section as follows:

- if the resource is free, lock and unlock is cheap and straightforward (~ 10 instr.)
- if a locked resource is found, the requesting process has to wait either by busy wait (spin lock) or by interrupt wait (suspend lock).

In the multiprocessor case busy wait is considered to be the most economic solution [BL79] because probability is high that a process on another processor keeps the lock and releases it soon. On the other hand, increased access to the shared memory which stores the lock may slow down the progress of parallel processes because of high memory contention. In a uniprocessor system it is usually recommended to apply interrupt wait because otherwise the rest of the time slice is consumed by an active IDLE-operation ($> 10^5$ instr.). The waiting process has to be activated by a SIGNAL of the process freeing the resource. Hence, this situation causes two process switches.

Global DBMS tables are charaterized by very high traffic; hence, they are called hot spot data [Bl79]. For example, the lock for the system buffer is typically set by a process once per 1000 instructions and is held for about 50 instructions (high traffic lock). After 1000 instructions it is requested again

with high probability by the same process. When a DBMS process holding a high traffic lock is preempted because of page wait or time slice runout, there is the danger of establishing a convoy in front of the resource. This happens because all subsequent processes will probably run against the high traffic lock, enqueue and then voluntarily go to sleep. When the owner O of the resource finally continues, he will soon release the resource thereby locking it for the first process P in the queue (FIFO). After signalling P and continuing normal processing, O will soon bump against the lock set for P with high probability. According to the given protocol O enqueues (at the end of the queue) and goes to sleep. When the resource is assigned in a FIFO-manner, then a stable quasi-deadlock arises. A proposal to break up the convoy is given in [Bl79].

3. Allocation of DBMS processes

In the previous section we have discussed the main issues of DBMS integration, the basic concepts and problems of which have also been discussed in [St81]. Now we are going to investigate conceivable process structures thereby assuming that the introduced concepts of isolation, cooperation and synchronization between processes are available.

Single server DBMS

Only one process is assigned to the DBMS as illustrated in Fig. 1. The areas DAi are used to directly exchange parameters and data between APi and DBMS. The OS code (system memory) is part of every virtual address space and is protected by hardware primitives. OS functions are exclusively called via SVC's. Communication between APi and DBMS is performed by the resp. primitives of the OS.

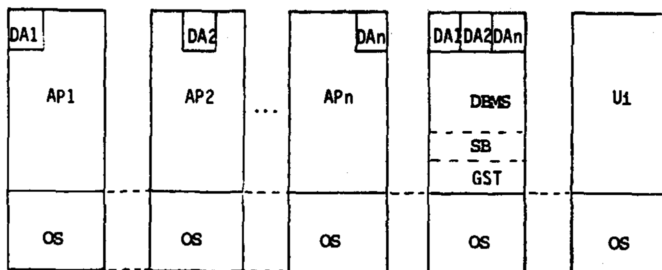

Fig. 1: Allocation of n application processes to a single DBMS server

The single server solution has a number of advantages like simple communication structure, no global DBMS data, no critical sections, etc. However, it also incorporates some inherent complexities and flaws. In order to prevent synchronous I/O of the DBMS process, some kind of multi-threading must be implemented. Whenever the server initiates an I/O-request, the context of the current thread has to be saved and DBMS processing has to resume with another request in a parallel thread. Hence, asynchronous I/O generates more complex DBMS code and introduces two levels of scheduling - in the OS and in the DBMS. Even in this improved solution the entire DB-related activity is often temporarily suspended by page faults or other synchronous wait conditions, since there is only one server process.

Because of the uneven load characteristics of the various processes high priority resource allocation for the server must be mandatory. But even if the server could almost permanently execute user requests, DBMS processing is limited by the capacity of a single CPU. A single server solution cannot take advantage of tightly coupled multiprocessors and is therefore not competitive in such widely used environments.

Multiple server DBMS

As discussed in section 2 multiple processes can be allocated for performing DBMS processing. The distribution of DB-requests to different processes requires shared segments for global DBMS data and the respective access synchronization. A page fault in a server process does not necessarily interrupt DBMS services, when other servers are ready for execution. Although multi-threading is conceivable in each of the servers, the simpler solution with single-threading and synchronous I/O seems to be acceptable. Here, scheduling is entirely left to the OS - its original and natural location - thereby resulting in a substantial simplification of the DBMS code. A server completely executes a DB-request before it fetches the next one from the server queue or goes to sleep in case of an empty queue. Only when a request has to be delayed due to locks on DB objects, the server switches to a subsequent request before finishing the current one, which can be accomplished by a rather simple server protocol. This is necessary because of the long duration which would significantly reduce the availability of server processes to do useful work. Moreover, situations could occur, where all servers were blocked because of long lock waits although there is no transaction deadlock. In any case, different processors in a multiprocessor environment can support DBMS processing simultaneously.

A first, very simple approach could allocate a private server to each AP. With n APs this kind of symmetric process allocation would consume 2n processes. Since there is a synchronous calling structure between each pair APi-DBMS(i) and since processes are expensive objects, this solution does not exploit all resources in an optimal way.

A more realistic solution allocates m servers for n AP processes (n > m). As shown in Fig. 2, a special monitor process is employed to distribute the requested services to the

servers. An APi puts his request into the monitor queue, signals this event to the monitor and goes to sleep. When active, the monitor supplies the servers with incoming user requests in the server queue and activates a sleeping server when necessary. It is expected that the monitor - when activated - can distribute several requests at a time. A server picks up requests from the server queue as long as the queue is not empty. This kind of communication is assumed to reduce the overall process switching costs.

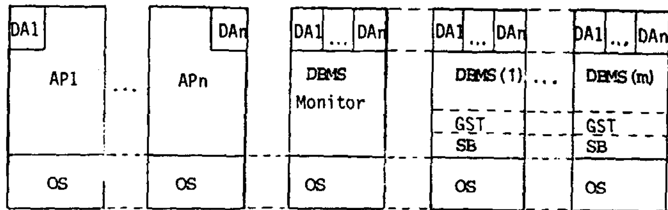

Fig. 2: Multiple DBMS servers with monitor

A symmetric return of a DB-call would imply up to 4 process switches. Fortunately, the return path could be shortened. Since the caller is known to the particular server, it can be signalled by the server and the results can be passed back directly. Hence, each DB-call causes up to 3 process switches in this multiple server/monitor solution.

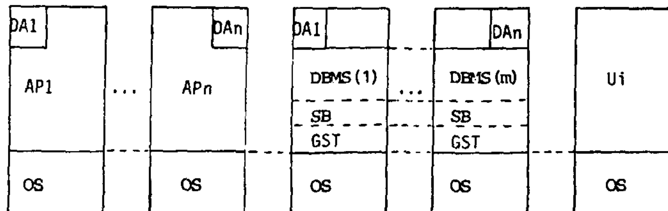

Fig. 3: Multiple DBMS servers without monitor

The following approach promises an improvement of the process switching overhead. The idea of representing the monitor by a separate process is rejected. Instead, its functions are either integrated in the OS requiring modifications of the OS code with all its disadvantages or in each of the AP's using a special connection module linked automatically to the application programs. The resulting multiple server solution is shown in Fig. 3. Obviously, the overhead of each DB-call is limited to 2 process switches.
Both multiple server solutions are further investigated in the next sections by means of a simulation model to determine process switching overhead, response times, convoy problems on high traffic locks, etc. The results of the symmetric 2n solution can be estimated as an upper bound of the n+m-solution.

4. The simulation model

To evaluate both multiple server DBMS solutions and in particular to assess the costs induced by the selected embedding strategy, a comprehensive simulation model has been implemented. A high level programming language was employed in this task, since the available general purpose simulation languages like GPSS did not provide suitable primitives, especially to express the synchronization mechanisms needed to represent DBMS-AP interaction in an easy and adequate manner.
The subsequent explanation of the simulation model follows a conceptual subdivision into three major components. Each of them embodies several important aspects underlying the investigation. Firstly, a relatively coarse abstraction of the DBMS and AP internal processing with special regard to the usage pattern of OS primitives is put into specific AP and DBMS models. Secondly, the way AP's transfer their requests to the DBMS and receive the associated answers, is demonstrated by the communication structure model, and thirdly the basic OS primitives utilized in a DBMS implementation on top of a general purpose OS are incorporated into the OS submodel.

The AP and DBMS models

The structure of the AP and DBMS internal processing model is closely determined by the main purposes of the overall simulation approach. The investigation focusses primarily on the interactions among AP, DBMS, and OS and their related costs, especially in terms of process switches. Additionally, the impact of high traffic locks on the total system's performance is to be analyzed and thus has to be reflected in the DBMS model.
Therefore, only coarse models for the AP and DBMS internal activities were designed, mainly incorporating the sources of OS requests within AP and DBMS and comprising the manipulation of high traffic locks within the DBMS. This led to a solution, where every active component in the system is regarded as a cyclic process which repeatedly executes a sequence of actions until the simulation run is completed.
The AP model, illustrated by Fig. 4a, is very simple. From the OS point of view, each AP performs a certain number of machine instructions, which finally result in a request for DBMS service (execution of a DML-statement). The request is then forwarded via OS primitives to the DBMS described by the communication structure model. In the meantime the AP synchronously waits for the results provided by the DBMS. After reactivation and receipt of the results, the AP continues by analyzing the data returned and starting the next cycle. It is assumed that an AP is never suspended for other reasons by the OS; in particular, an AP does not initiate private

I/O-requests. The only parameter characterizing the AP in this model is the number of machine instructions necessary to execute a DML statement in the underlying application environment. This value certainly depends on several factors like the type of application, the functional capabilities of the DML, the data model, etc. For the sake of simplicity this parameter was set to a fixed quantum in all simulation runs.
One of the two variants of multiple server DBMS analyzed herein employs a so-called monitor process to forward DB requests received from an AP to the servers. Again, the structure of this process is very simple, as can be seen by Fig. 4b. The only action performed by the monitor is the selection of a free DBMS server and the assignment of an DML statement. Every time, the DB monitor is activated, it distributes all the statements found in the monitor queue and then synchronously waits for the next signal. Again, waiting implies a process switch (see discussion of the communication structure model). The DB monitor, similar to the AP model, is characterized by the number of instructions to analyze and transmit a DML-statement.

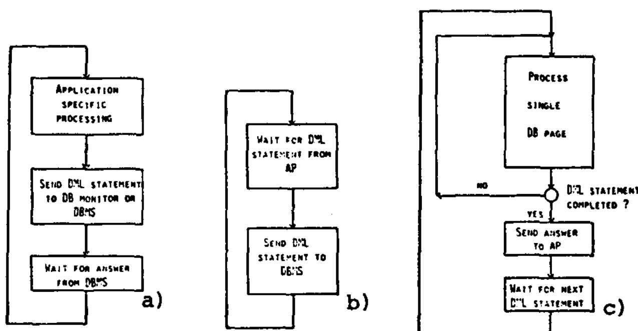

Fig. 4: Models for AP, monitor and server

The most complex model is that of the DBMS server. Therefore it will be explained in two levels of abstraction graphically illustrated by Figs. 4c and 5. Fig. 4c shows the server as a cyclic process which repeatedly answers DML-requests. A DML-statement is executed in a number of steps. Each of them represents the processing of a database page. The DB is assumed to be organized into fixed size pages residing on external memory. Starting with the DML-statement, the server dynamically transforms requests to higher level data objects like tuples into a number of accesses to the pages containing the data which represent those objects on external storage. Depending on the type of statement and the available access paths, one or more pages will be inspected until the requested data have been located. In order to simulate a variety of different requests without explicitly modelling DBMS internal storage structures the number of pages touched per DML-statement was taken from a random distribution as another simulation parameter.
Fig. 5 shows a more detailed specification of the activities involved in page processing. At first, the DBMS server has to ensure that the appropriate locks for the pages to be accessed are acquired. In reality, this is done by examination and perhaps subsequent manipulation of DBMS global data structures. In the simulation model, again, the decision whether or not a page is locked is made according to a random distribution, since the details of lock management are of no interest for this investigation. If a locked page is found, the DML-statement has to be suspended until the lock is released. Since no explicit locking information is maintained, this problem is also solved by taking the suspension period's duration from a random distribution. In the model, the DBMS server is not deactivated in this situation. Instead, it continues with another DML-statement, if possible. Although there are no data structures to represent locking information, their fictive manipulation under the protection of a high traffic lock (lock latch) is included in the server model. Since there are additional latches, the discussion of their management is deferred until all of them have been introduced.
In reality, after the server has acquired the lock, the page itself has to be fixed in main memory to analyze its contents. This implies an I/O operation if the page does not yet reside in the system buffer. Similar to the approach chosen for lock management, no control structures are maintained for the

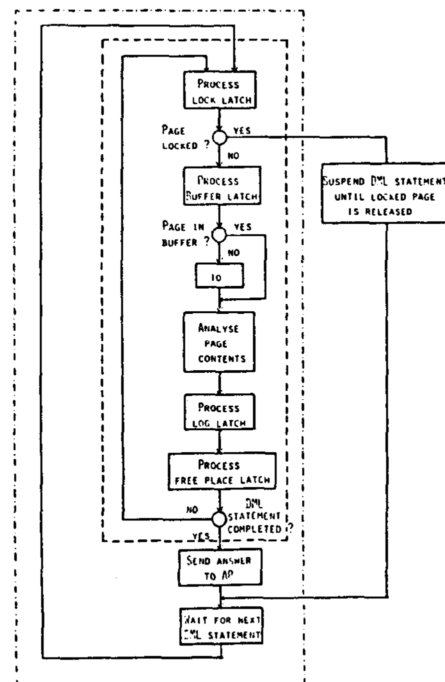

Fig. 5: Detailed structure of a DBMS server

system buffer and the decision, whether or not an I/O is necessary is made due to a random distribution. Again, the high traffic lock (buffer latch) protecting control structures for the system buffer is modelled explicitly. After having fixed the page in main memory, the server can analyze its contents. Only the duration of this step is of interest in the simulation and has been selected as a model parameter. Two more high traffic locks are acquired and released before the processing of the page is completed.
The manipulation of high traffic locks has been modelled in detail and is depicted in Fig. 6. In the simulation model, the state of each latch is maintained and checked whenever a server process wants to enter the respective critical section. In case the latch is free the server acquires it, executes a number of machine instructions to manipulate the global data structures and finally releases the latch. Otherwise, the current server process tries to acquire the latch held by another server process. According to the OS model, this situation can only occur when the time slice of the latter process has run out within the critical section protected by the resp. latch. Therefore, the current process has to wait anyhow until the owner of the latch regains control of the CPU. In the simulation model, the current server process is delayed for a period of time taken from a random distribution.

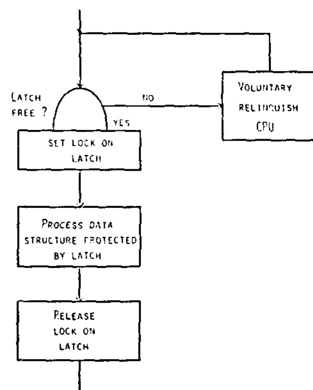

Fig. 6: Control flow on a latch

The communication structure model

In the previous section the process models for DB monitor, DBMS, and AP were introduced and the locations of control transfer and DB requests were identified. Now, the communication structure model describes in detail, how a DML-statement is passed from the AP to the DBMS server and back to the AP and when process switches are mandatory or can be avoided. The first variant examined utilizes a so-called DB monitor to dispatch DML-statement to the DBMS servers and is illustrated by Fig. 7, where n AP´s generate the workload for m DBMS servers. The sequence of events in processing a DML-statement is the following.

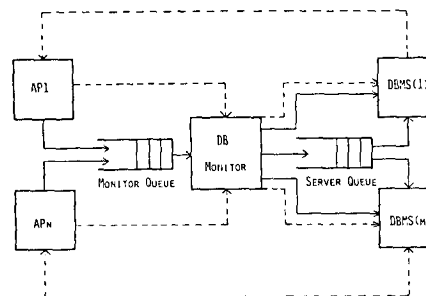

Fig. 7: Communication model for the monitor/server structure

At first, the request is generated in APi and inserted into the monitor queue. Thereafter, APi is suspended until the DML-statement has been completed by the DBMS. Whenever the monitor queue was empty at the time the AP inserted its request, the monitor is signalled, after reactivation by OS scheduling it will eventually dispatch the request. Whereas each single DB-call causes a process switch for the AP, the monitor queue may be emptied ($\leq$ n requests) during a single monitor activation. When removing a statement from the monitor queue, it at first controls the state of the servers. If any of them is waiting for a DB-call, it is signalled by the monitor and assigned the statement. On the other hand, if all servers are currently executing a DML-statement, the monitor inserts the request into the server queue, because whenever a server completes a DML-statement, the server queue is checked for additional work, thus saving unnecessary process deactivations. After signalling the APi, a server immediately starts processing the next DML-statement found in the server queue. Only if the queue is empty deactivation takes place.
The second variant investigated works without monitor. Principally, the functions of this process have been integrated into the AP´s, namely selection of a free server or alternatively the insertion into the server queue, in order to further reduce the number of process switches.

The OS model

The OS model simulates a general purpose time sharing system on a uni-processor, the gross structure of which is shown in Fig. 8. The CPU capacity is distributed based on time slices (TS) in order to grant a fair share of that resource to each process. The processes mentioned so far are the only ones running on the computer system. As can be seen from Fig. 8, the OS keeps track of the processes involved by holding them in one of the four queues. Those ready for execution are gathered in the CPU queue, which is managed in FIFO order. The remaining three queues contain the blocked processes. The I/O queue is entered by servers performing an I/O request to fetch a

DB page into the system buffer. The synchronization queue comprises processes waiting for a signal operation issued by another process, i.e. AP's awaiting the servers' answer, idle servers, etc. Finally, the delay queue is reserved for servers having voluntarily relinquished control of the CPU because they have encountered a locked latch. The server is kept

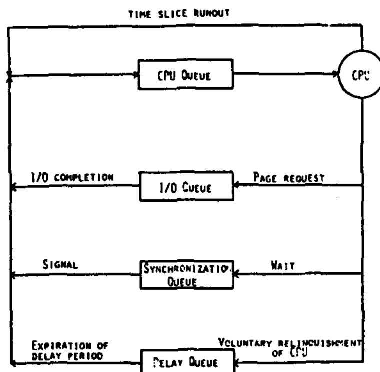

Fig. 8: Structure of the OS model

in this queue until the delay period computed from a random distribution has expired. It is important to note that leaving one of the latter three queues does not automatically imply the acquisition of the CPU, since its queue is processed in FIFO order. The last transition in Fig. 8 to be discussed represents CPU preemption due to TS runout. The running process is inserted at the tail of the CPU queue and gets its TS refreshed. The length of the TS is measured by the number of machine instructions a process is allowed to execute before being preempted from the CPU. This is due to the design decision to chose machine instructions as the unit for the simulation clock. Consequently, real times like I/O duration have to be transformed using the processor speed specified in MIPS.

To gain a better insight into the dynamics of the simulation system the process management, in particular the TS management is outlined in the following. All the processes are given equal priorities and are modelled as a collection of one or more action blocks which account for a certain amount of machine instructions. According to the process model the action blocks pertaining to the running process are executed thereby decrementing the TS and incrementing the simulation clock until either the process is blocked or the TS runs out. In the latter case, the next action block's identification is saved and the process is appended to the CPU queue. The AP model consists of a single action block as well as the monitor model. The server model comprises 6 blocks four of which represent the critical sections. Each time the DBMS is preempted within such a block the resp. latch has to be marked locked. The remaining blocks simulate the page specific computation and the software costs imposed by I/O processing. Two more reasons exist to advance the simulation clock. Contrary to the above case, however, none of the processes is charged with the associated costs.

Firstly, the simulation clock is incremented to bridge processor idle periods, when all the processes are in the blocked state. Secondly, whenever a process leaves the CPU or an idle period is terminated process switching costs occur.

5. Simulation results and interpretation

Model parameters

As can be seen from the discussion in the previous section, the complexity of the simulation model is considerable. Fig. 9 is intended to give a comprehensive overview and a systematic classification of the full spectrum of parameters involved. The values of the parameters kept constant over all simulation runs are listed in parentheses below the parameter name. Besides processor speed and the probabilities the parameter values have been specified in terms of machine instructions. Where necessary, real times have been transformed based on the processor speed of 1 MIPS. Parameters with blank value field have been varied in the simulation. The values for the DB related factors are derived from extensive DBMS measurements [EHRS81]. The DBMS under investigation was of the CODASYL type with a rich variety of storage structures to support rapid data retrieval and manipulation. OS specific factors reflect the situation in the measurement environment and were the standard values provided by the vendor. The number of instructions per DML-statement and per page, to our experience, are typical for a CODASYL interface. Nevertheless, we do not expect principal changes in the basic values, even if these parameters were doubled.

Variation of the number of AP's

The first series of simulation runs was performed with a constant number of three servers and in the average 3.5 I/O's per DML-statement (p_{IO}=0.35). The number of AP's (# AP) ranged from 3 to 11. Figs. 10 to 14 summarize the results obtained. VAR1 is used as an abbreviation for the multiple server solution with DB monitor, VAR2 for the solution without. Fig. 10 gives the total throughput in terms of DML-statements in a run of 1 hour of simulation time. As expected VAR2 performs better than VAR1 in all comparable situations. After drastic increases from 3 to 6 AP's the system approaches a saturation state. The symmetric assignment of AP's to servers (2n solution) leads to the worst results in performance.

Fig. 11 tabulates the average number of servers (# S) already processing a DML-statement whenever an AP produces another request. The closer this value approaches the total number of servers allocated, the more often a request

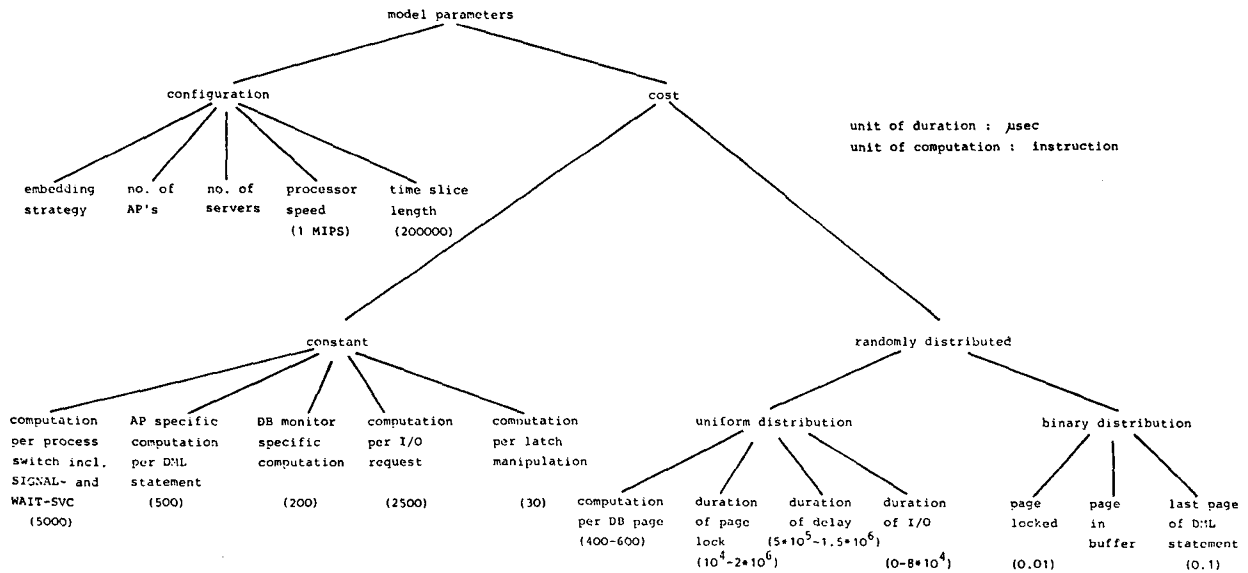

Fig. 9: Overview of the simulation model parameters

#AP	VAR 1	VAR 2
3	38 229	39 309
4	46 469	48 161
5	52 063	54 734
6	57 070	58 870
7	59 359	61 095
8	60 358	61 963
9	60 665	62 363
10	60 950	62 575
11	60 966	62 500

Fig. 10: Throughput

# AP	VAR 1	VAR 2
3	1.20	1.23
4	1.79	1.84
5	2.27	2.35
6	2.61	2.67
7	2.81	2.86
8	2.92	2.94
9	2.97	2.98
10	2.98	2.99
11	2.99	2.99

Fig. 11: Average parallelism of servers

#AP	VAR 1	VAR 2
3	6.512	5.556
4	6.284	5.342
5	5.966	5.061
6	5.699	4.830
7	5.528	4.687
8	5.438	4.619
9	5.397	4.587
10	5.379	4.577
11	5.373	4.573

Fig. 12: Average number of process switches per DML-statement

will be transferred via the server queue, thus avoiding a process switch. The symmetric assignment of servers to AP's cannot benefit at all from that optimization in the communication structure because DML-statements are always sent directly to the server. This is illustrated by Fig. 12, too, which displays the average number of process switches incurred in processing a single DML statement. With the lowest number of AP's the monitor variant almost exactly needs one process switch more. The slight reduction of this difference with a rising number of AP's is due to multiple DB requests being dispatched during a single activation of the monitor.

In Fig. 12 the average number of process switches in a configuration with 3 servers and 11 AP's is about 4.5 for VAR2. Since roughly 3.5 of these are caused by buffer I/O and another process switch is performed after the AP sends its DML statement, the transfer is accomplished in most cases via the server queue.

Fig. 13 tabulates the average length of the time interval between removing a statement from the server queue and signalling the AP its completion. It is obvious that those values are almost constant. However, Fig. 14 indicates that the response times, which include the waiting period for a server, rises with the number of AP's. Moreover, the values for VAR1 are slightly but consistently higher than those for VAR2. This results from the fact, that one additional process consumes its share of CPU resources in VAR1, sometimes deferring the execution of a server which would not have been the case in VAR2. This is consistent with the observation that the throughput virtually remains constant with more than 7 AP's. Looking at the processor utilization reveals that even though the throughput is stagnating, about one third of the simulation the processor idles. This observation gives rise to the supposition that the number of servers does not suffice to effectively service the AP's, because the servers perform too much I/O. In order to corroborate that supposition several simulation runs with 2 and 4 servers and VAR1 were executed. The overall throughput is depicted in Fig. 15. Obviously, the introduction of the fourth server boosted throughput from about 60.000 to 70.000 DML statements for 8 AP's and reduced the idle periods to 10% for more than

# AP	VAR 1	VAR 2
3	0.1646	0.1638
4	0.1702	0.1679
5	0.1730	0.1705
6	0.1750	0.1717
7	0.1760	0.1722
8	0.1765	0.1724
9	0.1771	0.1725
10	0.1768	0.1723
11	0.1769	0.1726

Fig. 13: Average server time

# AP	VAR 1	VAR 2
3	0.2751	0.2676
4	0.3010	0.2906
5	0.3289	0.3191
6	0.3668	0.3560
7	0.4120	0.4008
8	0.4642	0.4527
9	0.5208	0.5072
10	0.5772	0.5629
11	0.6361	0.6211

Fig. 14: Average response time

#AP	#S = 2	#S = 3	#S = 4
2	28 870	-	-
3	35 623	38 229	-
4	39 956	46 469	47 407
5	42 345	52 063	55 803
6	43 294	57 070	62 577
7	43 554	59 359	67 495
8	43 687	60 358	70 585
9	-	60 605	72 183
10	-	60 950	73 195
11	-	60 966	73 242

Fig. 15: Throughput

10 AP's. Now, the elimination of the third server process showed drastically reduced throughput to about 43.000 and yielded idle periods of about half the processor time for more than 6 AP's.

Variation of the number of servers and the I/O frequency

The quantitative results of the previous section showed the number of servers to considerably affect the overall performance. Another lesson taught were the detrimental effects of high I/O frequencies combined with a small number of server processes. Very often all servers waited for I/O completion while the processor stood idle. Therefore, the number of servers was varied from 1 to 8 whereas the number of AP's was fixed at 8. These configurations were simulated with 3.5 and 1.0 I/O operations per DML statement (p_{IO}=0.35 and p_{IO}=0.1).
The results are tabulated in Fig. 16 in terms of throughput.

	VAR 1		VAR 2	
#S	3.5 IO	1 IO	3.5 IO	1 IO
1	23 266	70 593	23 347	71 555
2	43 679	111 921	44 139	118 040
3	60 358	124 834	61 963	135 180
4	70 577	124 745	74 016	135 898
5	73 762	122 466	78 754	133 155
6	73 128	120 588	78 628	131 600
7	72 356	120 652	77 327	130 618
8	72 222	120 769	77 063	131 026

Fig. 16: Throughput

Interestingly for each variant and I/O frequency, the values in the beginning improve drastically, then gradually and finally after a maximum has been reached decrease. In general, the maximum is reached with a greater number of servers for the monitor solution and the greater I/O frequency. Not surprisingly, throughput is clearly improved when the I/O rate is smaller, since every saved I/O eliminates the associated computation costs and a process switch. The performance losses beyond a certain number of servers can be explained as follows. When the number of servers is small in comparison to that of the AP's, each DML statement is transferred via the server queue, but frequently all servers are blocked because of an I/O request. Such idle periods can be exploited by introducing additional servers. However, only a limited number of servers will be able to utilize the entire CPU capacity. Adding further servers only increases the number of process switches at the expense of useful processor capacity, since more and more statements are transferred directly from AP to server (explicit signalling). Now, Fig. 17 displays the average number of process switches per DML-statement. These values include process switches triggered by I/O requests. As can be seen this value steadily rises with the number of servers. At the first glance, this fact might look somewhat puzzling but it is quite consistent with what has been said in connection with Fig. 16. When there is a single server almost all requests are dispatched via the server queue. The more servers the system contains the higher becomes the probability of an additional process switch for assigning the request. But as long as processor capacity is wasted by idle periods anyhow, these process switches do not diminish the total amount of useful work performed. When the CPU is saturated additional process switches reduce the overall throughput. Figs. 18a and 18b graphically

	VAR 1		VAR 2	
#S	3.5 IO	1 IO	3.5 IO	1 IO
1	5.381	2.620	4.536	2.070
2	5.399	2.836	4.569	2.232
3	5.438	3.139	4.619	2.498
4	5.586	3.410	4.770	2.749
5	5.858	3.574	5.039	2.928
6	6.130	3.646	5.334	3.017
7	6.285	3.662	5.524	3.043
8	6.323	3.664	5.561	3.046

Fig. 17: Average number of process switches per DML-statement

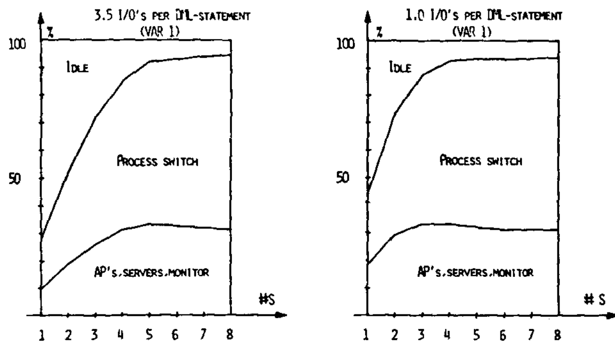

Fig. 18: Distribution of processor time

	Throughput 60		Throughput 30	
#S	VAR1	VAR2	VAR1	VAR2
1	23 238	23 312	23 266	23 347
2	43 397	43 892	43 679	44 139
3	59 441	61 181	60 358	61 963
4	68 981	72 450	70 577	74 016
5	72 201	77 021	73 762	78 754
6	71 713	76 982	73 128	78 623
7	71 051	75 577	72 356	77 327
8	70 842	75 436	72 222	77 063

Fig. 19 Throughput

illustrate the relative share of the CPU time spent for server, monitor and AP processing combined, the idle times, and process switching overhead, which surprisingly can reach 65% of the total capacity.
Though the total number of process switches per DML-statement is about 2.5 less when only 1 I/O is issued per statement, the relative share of process switching overhead does not change significantly. However, throughput increases substantially. Idle periods, which occur when all the servers synchronously wait for I/O completion, are reduced much faster with rising number of servers, when only 1 I/O is required per DML-statement. This is reasonable, since here servers have much longer page processing periods before the next I/O request is performed, thus reducing the probability that all servers wait for I/O completion at the same time.

Influence of latch synchronization

An important aspect of server synchronization is the influence of the frequency of latch requests and the length of critical sections (lock duration) on performance. In particular, the probability of server preemption when owning a latch should be investigated.
Therefore, a number of simulation runs were dedicated to answer these questions. In all simulations the number of servers were varied from 1-8 keeping # AP=8 constant. The probability of physical I/O during page access was $p_{IO}=0.35$.
A throughput test was performed with different lengths of critical sections (lock duration: 30 and 60 instr.). Fig. 19 shows the overall effect with known characteristics. Longer critical sections increase the risk of "blocking" time slice runout which, in turn, enhances the waiting times of servers in front of a latch which directly diminish throughput. VAR2 in superior to VAR1 by up to 7.5%. Using the shorter critical section a gain of up to 2% can be expected.
We are now going to evaluate more thoroughly the effects of latches. Since only the server model is involved with latch requests, both variants of server structures are assumed to produce uniform results concerning the preemption problem. Indeed, all numerical results delivered were very similar. Therefore, we will limit their presentation to VAR1.
The distribution of preemption in the various code sections (latch (critical sections), I/O processing, page processing) is shown in Fig. 20. Of course, the number of all preemptions is strongly dependent on the time slice length. With fixed time slices, the probability of preemption increases with the length of the resp. code section. Having latches with 30 instructions, latch preemption varies between 7% and 8.5%.
The sum of the waiting times in front of latches grows with increasing number of servers. Because latches are used more often, the probability of preemption is augmented. On the other hand, a preempted server has to spent more time in CPU queue applying for a new time slice before it can free the latch. Fig. 21 shows a summary of critical (section) preemptions and waiting servers. The average number of waiting servers, however, is not very critical; in particular, long convoys could not be observed. This is mainly due to the convoy resolution applied according to [B179], that is, as soon as growing queue was detected all waiting servers were signalled (violating the FIFO principle). Nevertheless, the percentage of waiting servers shows that about 20-30% of the servers are blocked in the average.
It can be argued that these results do strongly depend on the activity pattern of the servers. Therefore, the probability of physical I/O was reduced to $p_{IO}=0.1$, that is, a DML-statement requires 1 page fetch in the average. Due to the reduced I/O-activity more DML-statements could be processed increasing latch use and preemption. Fig. 22 and 23 are comparable to Fig. 20 and 21. Latch preemption now ranges from 13% to 14%. Since less servers are waiting for I/O, the CPU queue should be longer, thus enhancing the waiting time for the preempted server. This is indicated by the characteristic values in Fig. 23 for 2-4 servers. Due to the I/O-reduction this range of servers achieved optimum throughput; that is, the servers are most active. For higher server numbers, IDLE-time increases thereby

#S	total number of preemptions	preemptions holding a latch	preemptions processing I/O	preemptions processing a page	% latch
1	1 724	140	993	589	8.12
2	3 259	240	1 915	1 099	7.36
3	4 515	382	2 586	1 539	8.46
4	5 280	418	3 044	1 802	7.91
5	5 522	416	3 292	1 795	7.53
6	5 473	443	3 153	1 847	8.09
7	5 416	416	3 249	1 734	7.68
8	5 406	380	3 192	1 818	7.02

Fig. 20: Distribution of preemptions (p_{IO}=0.35)

#S	Var 1 critical preemptions	waiting servers	% servers waiting	avg. no. waiting servers/latch
1	140	0	0.00	0.00
2	240	52	21.66	0.21
3	382	152	19.89	0.39
4	418	331	26.39	0.79
5	416	494	29.68	1.18
6	443	675	30.47	1.52
7	416	692	27.72	1.66
8	380	644	24.42	1.69

Fig. 21: Critical preemptions (p_{IO}=0.35)

#S	Var 1 total number of preemptions	preemptions holding a latch	% latch
1	3 067	411	13.40
2	4 865	678	13.93
3	5 420	719	13.26
4	5 416	709	13.09
5	5 316	670	12.60
6	5 253	697	13.26
7	5 236	724	13.82
8	5 245	719	13.72

Fig. 22: Number of preemptions (p_{IO}=0.1)

#S	Var 1 critical preemptions	waiting servers	% servers waiting	avg. no. wait. servers/latch
1	411	0	0.00	0.00
2	678	205	30.23	0.30
3	719	491	34.14	0.68
4	709	714	33.56	1.00
5	670	815	30.41	1.21
6	697	901	25.85	1.29
7	724	912	20.99	1.25
8	719	882	17.52	1.22

Fig. 23: Critical preemptions (p_{IO}=0.1)

reducing preemption conflicts. Again, critical preemptions do not impede the overall behavior dramatically.

6. Conclusions

The structure of the various simulation processes closely models real DBMS operation under a general purpose OS. This could be confirmed by extensive DBMS measurement experience [EHRS81]. Results of this project permit at least a partial validation of our simulations. Therefore, the following conclusions can be drawn.
The experimental results have shown the superiority of the multiple server solution without monitor over that with monitor. Depending on quantitative relation between servers and AP's 0.7 to 1.0 process switches were saved per DML-statement. The multiple server DBMS without monitor needed between one and two process switches to process a single DML-statement. The introduction of the server queue to asynchronously transfer requests without OS interaction turned out to be especially helpful, since in the optimum server-AP relation only slightly more than a single process switch was used per DML-statement. The formation of convoys or waiting times due to critical preemption turned out to be no severe problem. This was partially achieved by the chosen resolution strategy [B179].
However, process switching overhead in general consumed a huge share of the CPU resources (about 50%). This underscores the penality paid because of inadequate OS support which necessitates the partitioning of DBMS and AP to separate processes.
Since communication costs remain high even in the optimal configuration, the expressive power of a DBMS interface is also quite important for the communication structure chosen. A non-procedural interface needs generally less DB-calls than a navigational interface. To be specific, to run a given application (parts explosion, bill of material) with a relational and a CODASYL-system, we needed 2.2 times more DB-calls for the CODASYL-system (10356 vs. 22996 DB-calls). To save I/O related process switches multithread solutions within server processes have to be introduced. This kind of saving seems to be much more promising in applications with high I/O frequencies. Of course, another improvement is the availability of large database buffers.
These arguments indicate that better ways of communication support should be found. Ring protection or other novel OS isolation features [PR83] are prime candidates. Their saving potentional is the number of communication-related process switches investigated in this paper.

Acknowledgement

The authors wish to thank the referees for their constructive criticisms which helped to improve the final version of the paper. We are also indebted to Karin Mägel, who did the bulk of programming the simulation model, and to Dr. Andreas Reuter for numerous clarifying discussions on the subject of the paper.

References

Bl79 Blasgen, M. et al.: The Convoy Phenomenon, in: ACM Operating Systems Review, Vol. 13, No. 2, 1979, pp. 20-25

EHRS81 Effelsberg, W., Härder, T., Reuter, A., Schultze-Bohl, J.: Performance Analysis and Prediction of the Database System UDS, Technical Report 41/81, University Kaiserslautern, 1981 (in German)

Gr78 Gray, J.N.: Notes on Data Base Operating Systems, in: Lecture Notes in Computer Science 60, Advanced Course on Operating Systems, Springer-Verlag Berlin, 1978, pp. 393-481

GW79 Gray, J.N., Watson, V.: A Shared Segment and Interprocess Communication Facility for VM/370, IBM Research Report RJ2450, San Jose, 1979.

Hä79 Härder, T.: Embedding a Database System in an Operating System Environment, in: Datenbanktechnologie, Berichte German Chapter of the ACM2, B.G. Teubner, Stuttgart, 1979, pp. 9-24 (in German)

IBM83 IBM Corp.: System/370 Principles of Operation, order no. GA22-7000

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. 30-37

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

TM82 Tanenbaum, A.S., Mullender, S.J.: Operating System Requirements for Distributed Data Base Systems, in: Distributed Data Bases, H.J. Schneider (ed.), North Holland, 1982, pp. 105-114

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.