SEPTEMBER 1986 VOL. 9 NO. 3

a quarterly bulletin of the IEEE computer society technical committee on

Database Engineering

CONTENTS

SPECIAL ISSUE ON OPERATING SYSTEMS SUPPORT FOR DATA MANAGEMENT

Database Engineering Bulletin is a quarterly publication of the IEEE Computer Society Technical Committee on Database Engineering. Its scope of interest includes: data structures and models, access strategies, access control techniques, database architecture, database machines, intelligent front ends, mass storage for very large databases, distributed database systems and techniques, database software design and implementation, database utilities, database security and related areas.

Contribution to the Bulletin is hereby solicited. News items, letters, technical papers, book reviews, meeting previews, summaries, case studies, etc., should be sent to the Editor. All letters to the Editor will be considered for publication *unless accompanied by a request to the contrary.* Technical papers are unrefereed.

Opinions expressed in contributions are those of the individual author rather than the official position of the TC on Database Engineering, the IEEE Computer Society, or organizations with which the author may be affiliated.

Membership in the Database Engineering Technical Committee is open to individuals who demonstrate willingness to actively participate in the various activities of the TC. A member of the IEEE Computer Society may join the TC as a full member. A non-member of the Computer Society may join as a participating member, with approval from at least one officer of the TC. Both full members and participating members of the TC are entitled to receive the quarterly bulletin of the TC free of charge, until further notice.

Dear TC DBE Members and Correspondents:

It is a great honor and a challenge for me to serve as the chairperson of the Technical Committee on Database Engineering. It is a great honor to follow Professor Gio Wiederhold who contributed significantly towards the growth and prestige of our TC during the past few years. On behalf of our TC, I wish to express our sincere appreciation for his valuable services. The chairmanship of the TC is a challenge since we must take rapid action if we are to keep the *Bulletin* going in the presence of a severe financial crisis facing the IEEE Computer Society.

Since these problems require our immediate attention, I have taken several steps on which I need to report to you. Following Gio's advice, I have appointed Professor Arthur M. Keller vice chairperson, Professor Leszek Lilien treasurer, and Dr. Richard L. Shuey secretary to help me tackle these problems. I am also pleased to announce the appointment of Dr. Won Kim as the new editor-in-chief of the *Bulletin*. We are grateful to the outgoing editor, Dr. David Reiner, who has served in this position for several years and wished to step down.

Because of the recent cuts in the Computer Society contribution to the Technical Committees. it has become necessary for us to find new ways to generate funds to meet the publication cost of the *Bulletin*. As Gio pointed out in his letter in the last issue of the *Bulletin*, it costs the Computer Society about $18,000 to publish the *Bulletin*, requiring close to 40% of Computer Society Technical Activities Board's reduced budget, whereas a fair and balanced formula based on a fixed amount per TC plus a fixed amount per TC member who is also a Computer Society member would net us about $1,900. In light of this imbalance, we are considering several different options to set up a viable funding mechanism, but we have agreed to take only two immediate actions.

Beginning with this issue, we are instituting a *voluntary* page charge for the manuscripts accepted for publication in the *Bulletin*. The author's company or institution will be requested to pay a page charge of $50 per printed page, which it may pay in full or in part. I wish to reemphasize that payment of these charges will be strictly voluntary and will not in any way be a prerequisite for publication. On the other hand, if there are funds in an author's grant or organization that have been set aside for this purpose, their use for this is one concrete way to show support for the *Bulletin*.

Moreover, we shall starting with the next issue charge $100 per page for any conference announcements that are included in the *Bulletin*. In the past, several conference organizers have volunteered to pay, but there was no mechanism in place to receive the money.

We are considering other options too, including a subscription fee for the recipients of the *Bulletin*. Please feel free to write to me (arpanet: jajodia@nrl-css uucp: decvax!nrl-css!jajodia) about how you feel about our efforts to deal with the issues facing us.

In addition to all the individuals mentioned above, I want to acknowledge the invaluable help of Professor C. V. Ramamoorthy and Professor Ben Wah in working out the ideas expressed in this letter. Also, we are fortunate to have Mr. John Musa as the Vice President for Technical Activities. I appreciate very much his support of our efforts.

There have been many complaints in the past about the need to update the TC membership list. I am pleased to tell you that the Computer Society has recently hired someone for this purpose and new updates are proceeding rapidly. If you wish to have your name added to the list, you can do so by filling out a TC Application Form. You can obtain this form by writing to either me or the Computer Society.

Finally, I would like to mention the Third Data Engineering Conference, which will be held February 2-6, 1986 in Los Angeles. I along with several other Program Committee members attended the August Program Committee meeting in Chicago. There were over 200 submissions, too many of high quality, forcing the Program Committee to make tough decisions. The end result, I think, will be an excellent conference, and I do hope that all of you will attend.

Sushil Jajodia
September 15, 1986

Changes to the Editorial Staff of Database Engineering

It is a pleasure to return as Editor-in-Chief of Database Engineering. I would like to thank outgoing Chief Editor, Dave Reiner, for the great job he has done during the past two years. I also would like to thank Fred Lochovsky for his services as an Associate Editor during the past three years. I have invited Mike Carey and Sunil Sarin to serve as new Associate Editors. Haran Boral, C. Mohan, and Yannis Vassiliou will stay on as Associate Editors. I look forward to working with these outstanding colleagues on the editorial staff, as well as the new officers of the TC.

Haran Boral has put together the present issue of Database Engineering. The next issue is being put together by Yannis Vassiliou on the European ESPRIT project. These two issues will complete the 1986 edition of Database Engineering. The uncharacteristic delay in the publication of these two issues is the result of uncertainty that existed during 1986 about the financial situation of the Database Engineering TC. It was not clear until late 1986 whether the TC would be able to continue publishing the Database Engineering bulletin. Sushil Jajodia, the new chairperson for the TC, deserves much of the credit for making it possible for the TC to publish these two issues.

I am presently organizing an issue for March 1987 on integrated software engineering systems and/or database requirements for such systems. Mike Carey will edit the June issue on extensible database systems. Sunil Sarin will follow with an issue on federated database systems in September. C. Mohan is tentatively scheduled to do the December issue on bridging database theory and practice.

Won Kim

Editor-in-Chief
January, 1987

Letter from the Editor

This issue of Database Engineering is titled "Operating Systems Support for Data Management". Although not widely discussed in the literature, the impact of the operating system on the DBMS performance is substantial; usually in a negative way as has been originally discussed by Gray in his *Notes on Database Operating Systems* and Stonebraker in his July '81 CACM paper and demonstrated by Hagmann and Ferrari in their March '86 TODS paper.

For this issue I asked several people to contribute papers describing existing implementations/experience as well as current work. We are fortunate to have six papers: three describing running implementations, two describing ongoing research projects, and the last, examining the relationship between the DBMS needs and the services provided by current research operating systems.

The first paper, "An Operating System for a Database Machine" by Chris Nyberg of Britton Lee describes the IDM kernel. The paper discusses the various decisions undertaken in order to achieve high performance. It is an excellent example of specialization. It is also the only paper in the issue describing a commercial system.

Kevin Wilkinson and Ming-Yee Lai of Bellcore contributed "The JASMIN Kernel as a Data Manager Base". JASMIN is an operating system kernel operational at Bellcore. After describing the JASMIN DBMS and the kernel Kevin and Ming evaluate the kernel both from an implementation (i.e., were all the services needed there; could they be used easily) and performance (which of the features hindered overall DBMS performance) points of view.

"Supporting a Database System on Symbolics Lisp Machines" by Hong-Tai Chou, Jorge F. Garza, and Nat Ballou of MCC describes the issues encountered in implementing a DBMS in Lisp on the Symbolics machine. The main issues addressed in the paper are storage of Lisp objects on secondary storage and transforming Lisp objects between disk and in-memory representations.

"The Camelot Project" by Alfred Spector and his students of CMU describes a distributed transaction system to support transactions against a wide variety of object types, including databases. The project emphasizes generality as well as performance. After a description of the Camelot functions its implementation is outlined with emphasis on its relationship to Mach, the operating system kernel Camelot runs on.

Eliot Moss of U.Mass. authored the next paper, "Getting the Operating System Out of the Way". Eliot proposes DBMS's to be built in a new programming language on top of a minimal kernel. The paper outlines some of the desirable language features needed. It also discusses the pros and cons of the approach.

The last paper "Operating System Support for Data Management" by Michael Stonebraker and Akhil Kumar of UC–Berkeley examines many of the features of "next generation" operating systems from the "database guys" point of view. The paper argues that features such as transaction management and location independent files are harmful whereas remote procedure calls and light weight processes are essential.

I hope the DBE readers find this issue as enlightening as I have and join me in thanking the authors for their excellent contributions.

Haran Boral
January 1987

An Operating System for a Database Machine

Chris Nyberg
Britton Lee Inc.
Berkeley, CA

Introduction

The Intelligent Database Machine (IDM) is a backend database machine manufactured by Britton Lee Inc. [Ubell 85, Epstein 80]. The IDM has been observed to have "surprisingly good" performance for multi-user benchmarks [Boral 86, Boral 84]. This paper describes the characteristics of the IDM that allow it to achieve its high level of multi-user performance.

Resident software on the IDM consists of a kernel supporting multiple processes. Each IDM process implements a connection with a host process, executing transactions (multi-statement queries or updates) on the host process' behalf. The special purpose nature of the system allows the kernel and process code to be highly integrated. The kernel does not need to support heterogeneous or potentially hostile processes. This integration results in efficient and coordinated allocation of resources by both the kernel and processes. Resources such as the cpu and process memory are more effectively allocated than in a general purpose operating system. The integration also allows the kernel to provide better database support than a general operating system. Often this support consists of simply letting the processes manage and share their own resources.

The kernel provides a variety of services to the IDM processes. The focus of this paper is on the parts of the kernel that take advantage of the special purpose nature of the machine and on the integration of kernel and process code to provide effective database management support. Features of the kernel such as queue based process management, host communication support and tape i/o will not be discussed.

Section 1 explains how the kernel allows processes to access shared data and describes the allocation method for process memory pages. In section 2 the virtues of the nonpreemptive scheduling policy are given. Section 3 describes how disk management responsibilities are split between the kernel and the processes. Support provided by the kernel for concurrency control is presented in section 4.

1. Memory Management

The IDM is based on a 16-bit microprocessor with four virtual 64K address spaces, kernel program and data, and process program and data. A memory management unit maps the 4 64K address spaces to main memory (1 to 6 megabytes in size). Each virtual address space is divided into 32 2K pages. Pages for process data space are divided into per-process data (private data, stack and heap) and data shared by all processes. The kernel data space contains data private to the kernel and the data shared by all processes. The program images for both kernel and process are permanently resident in main memory, as are the kernel data and shared process data.

The kernel allocates main memory for the per-process part of each process' data image. A simple, efficient memory management scheme is used that does not involve paging or swapping virtual images to disk. Moving process data to and from disk was considered too expensive. In addition paging is incompatible with the nonpreemptive scheduling policy of the IDM (see next section) since a page fault is a preemption.

IDM processes spend most of their time waiting for a query from the host. During this time a process' data image can be taken away and later restored. The kernel uses these facts as the basis of its memory management algorithm.

To begin a transaction, a process must have data memory allocated to it. The process then retains its data memory until the end of the transaction. When the process completes its transaction and waits for another query from the host, it is eligible to have its data memory taken away

by the kernel. This is done only if there is another process in need of the memory. Memory-less processes are always waiting for input from the host and are not inside a transaction. There is nothing special about the state of their data images that cannot be easily recreated by the kernel.

A process without memory is simply a process table entry that contains the current database of the process, the network address of its corresponding host process, and input and output queues for communication with the host process. Such a process will be granted data memory when it receives a query from its host process if memory is available. If not, the process must wait on a queue of processes needing data memory. As memory becomes available, it is allocated to the queued processes on a first-come-first-serve basis.

The limitation of this algorithm is that the number of simultaneous processes is limited by the amount of available main memory. By storing virtual data pages on disk the kernel could support more "simultaneous" processes. However if processes holding locks were swapped to disk, the system could very easily thrash. The processes not holding locks are those not currently executing a transaction. Rather than swapping virtual data for those processes to disk and back, the IDM kernel simply scuttles it and restores it later.

The limitation on simultaneous processes is of practical concern only with small memory configurations. With the largest main memory configuration the kernel can allocate memory for approximately 200 processes. This number of simultaneous transactions is clearly sufficient to find bottlenecks elsewhere in the system.

2. No Preemptive Scheduling

Once the kernel allows a process to run it does not preempt the process. The kernel implements a "soft" quantum by setting a global variable in process data space after a process has held the cpu a fixed amount of time. Processes periodicly check this variable at points where it is convenient to relinquish the cpu. If the variable is set a process will call the kernel to deschedule.

Without preemption, processes can perform critical section code that would normally need to be executed in a privileged address space. Although kernel and process code are differentiated by separate address spaces, the "operating system" of the IDM extends from the kernel and into process code. Indeed many sections of IDM code could be executed either in kernel or process space. Code that involves interrupts (host, disk and tape i/o) or requires numerous memory mapping changes, such as lock management (see section 4), runs in the kernel. Other critical section code can be executed in process space where it can be invoked without suffering the overhead of a system call.

It is possible to allow preemption if processes obtain locks for their critical sections. However this uncoordinated allocation of resources leads to inefficient use of the cpu. The classic example of this is the convoy phenomenon observed in System R [Blasgen 79]. Once a process is preempted while holding a highly utilized short term lock, a queue of processes waiting for the lock can quickly form and persist for a long time. There are steps that can be taken to lessen the degradation of preemptive scheduling on critical sections (such as using finer granularity locks), however it requires careful monitoring of a preemptive system to identify these degradations. It is simpler not to allow preemptive scheduling, thereby allowing processes to easily coordinate use of the cpu with the execution of critical sections.

In addition to critical section execution, nonpreemptive scheduling also allows processes to make better use of noncritical shared resources. For instance a process will not give up the cpu in the middle of referencing a disk cache page. If the process was preempted during this time and the disk page was no longer in the cache when the process was rescheduled, the process would have to arrange for the page to be read into the cache again. Without preemption IDM processes can deschedule when they have completed a short term use of a shared resource.

3. Disk Management

Disk management responsibilities are split between the processes and the kernel. At the low level the kernel schedules disk i/o at the request of processes and supports the optional mirroring

of disk data on separate disk drives. Above this the processes control a cache of disk pages and implement the file system (or access methods).

Disk pages in the IDM are 2048 bytes in size. A pool of up to 950 main memory pages is used to cache disk pages, known simply as "buffers". For each buffer there is a buffer structure in the shared portion of process data space. Buffer structures contain the disk address and status of the buffer and allow the buffers to be hashed by disk address, grouped by relation and ordered by replacement priority. Processes can directly manipulate the buffer structures and rely on the nonpreemptive scheduling policy to execute critical sections.

When a buffer is referenced its replacement priority is updated by the referencing process. The buffer replacement policy implemented is a variation on LRU (Least Recently Used) known as "Group LRU" [Nyberg 84]. The buffers are divided into three groups of decreasing priority: system relation pages, user relation index pages, and user relation data pages. Within each group a LRU policy is used. Buffers can also be made never replaceable and, conversely, immediately replaceable.

A status field in the buffer structure allows buffers to be marked as *scheduled for i/o, dirty,* or *log pinned.* A process that finds a buffer scheduled for i/o usually calls the kernel to deschedule until the i/o has completed. A status of *log pinned* is used in conjunction with *dirty* to indicate that the changed page can not be written to disk before the transaction log is. Log pinning is part of the IDM's write ahead log [Gray 78] implementation.

The kernel provides system calls that allow processes to initiate disk i/o on buffers. Processes can either have the disk i/o done asynchronously or be descheduled until the i/o is complete. The asynchronous i/o is used to prefetch pages or, better yet, read or write sequential sectors.

The kernel supports a system call to flush out all dirty buffers for a particular relation. If the relation is the transaction relation (i.e. the transaction log) the kernel may have to order the writes of the buffers due to dependencies among the pages. This is similar to the *selected force out* described by Stonebreaker [Stonebreaker 81]. A flush of the transaction log will also cause the kernel to clear the *log pinned* status in all buffers, allowing processes to write the these buffers to disk at will.

Five of the virtual pages in process data space are used to directly address buffer contents. Processes can specify which buffers are mapped into the pages by using a specific system call. No data movement is necessary for a process to read tuples from the disk page cache.

In a general purpose operating system, data (presumably a page in size) would have to be copied from the operating system's disk cache to the process' memory. Besides the overhead of the copying, the operating system must allocate a page of main memory to hold the process' copy of the disk page. If virtual memory is used the disk data may get paged to disk, thereby undermining the operating system's caching of the disk page. For these reasons, implementing the file system in process space is more efficient.

4. Concurrency Control

Concurrency control in a database system usually refers to synchronizing data accesses using data locks so that each transaction gets an atomic view of the data. In this section that definition will be extended to include other types of inter-process synchronization supported by the IDM kernel. The additional functionality includes locks for critical sections that involve disk i/o and checkpoint support.

Data locks are supported in two forms: relation and page locks. A table for these locks resides in main memory and can be up to 100K bytes in size. Since numerous memory mapping changes may be needed to address the lock table, the setting and clearing of data locks is done in the kernel. A process will be denied a lock if a conflicting lock is already held. When this happens the requesting process becomes blocked by the process holding the conflicting lock. The kernel checks for deadlock at this time. If deadlock is detected the kernel picks a deadlock victim. The process with the least accumulated cpu time is chosen as the victim and is directed by the

kernel to abort its transaction. When a process completes a transaction, it calls the kernel to clear all locks it holds in the lock table and to wake up all processes blocked by it.

Some critical sections of the IDM process code involve disk i/o. Most of these operations involve the file system: allocating disk pages, creating, destroying, dumping or loading a database. Processes in the IDM are descheduled when waiting for disk i/o. To preserve each of these file system critical sections there are special locks. The processes set and clear these locks themselves in shared memory. The kernel allows processes to wait for one of these locks if the lock is already allocated or wake up other processes waiting for such a lock when the lock is freed.

Checkpoints [Gray 78] on each database are periodicly performed by a dedicated process. The checkpoint procedure consists of writing all dirty buffer pages for the database to disk and writing a checkpoint record to the database's log. To make this operation atomic the kernel allows the checkpoint process to determine what other processes are active updaters in the checkpointed database and to temporarily suspend these processes during the checkpoint.

Conclusions

The IDM takes advantage of its specialized nature to allocate main memory for processes in a manner that does not result in thrashing. Processes share memory and control their own descheduling, allowing them to efficiently allocate and share resources. The kernel and processes are integrated to provide efficient disk management tailored to needs of the database management system. Data locks and other forms of concurrency control are supported by the kernel. The end result of these techniques is a database system that provides remarkably constant throughput in the face of an increasing work load.

Acknowledgements

While the implementation of the IDM kernel was done principally by the author, the design of the memory management algorithm and integration of kernel and process code came from database management architects, notably Bob Epstein, Mike Ubell, Paula Hawthorn and Charles Koester. Many people reviewed this paper and provided useful suggestions: Bob Taylor, Charles Koester, Scott Humphrey, Mike Ubell and Paula Hawthorn.

References

[Blasgen 79] Blasgen, M. et al., "The Convoy Phenomenon," *Operating Systems Review,* Vol. 13, No. 2, Apr. 1979, pp. 20-25.

[Boral 84] Boral, H. and D.J. Dewitt, "A Methodology for Database System Performance Evaluation," *Proceedings of SIGMOD '84,* 1984, pp. 176-185.

[Boral 86] Boral, H., "Design Considerations for 1990 Database Machines," *COMPCON '86 Proceedings,* Mar. 1986, pp. 370-373.

[Epstein 80] Epstein, R. and P. Hawthorn, "Design Decisions for the Intelligent Database Machine," *Proceedings of the 1980 National Computer Conference,* 1980, pp. 237-241.

[Gray 78] Gray, J., "Notes on Data Base Operating Systems," Report RJ3120, IBM Research Lab., San Jose, Calif., Oct. 1978.

[Nyberg 84] Nyberg, C., "Disk Scheduling and Cache Replacement for a Database Machine," Master's Project Report, University of Califorina, Berkeley, August 1984.

[Stonebreaker 81] Stonebreaker, M., "Operating System Support for Database Management," *Communications of the ACM,* Vol. 24, No. 7, July 1981, pp. 412-418.

[Ubell 85] Ubell, M., "The Intelligent Database Machine (IDM)," *Query Processing in Database Systems,* ed. by Won Kim et al., Springer-Verlag, 1985, pp. 237-247.

The JASMIN Kernel as a Data Manager Base

W. Kevin Wilkinson
Ming-Yee Lai

Bell Communications Research
435 South Street
Morristown, New Jersey 07960

ABSTRACT

The JASMIN kernel is an experimental, capability-based operating system kernel that provides a core set of facilities on which to build distributed applications. The JASMIN database system was designed as a research project in distributed processing and database management but also serves as a test of the functionality provided by the JASMIN kernel. This paper provides an overview of the JASMIN database system and the kernel and reports on our experience in using the kernel. We discuss the merits and demerits of the various kernel facilities in building the database management system. We claim that a general-purpose, minimal kernel is an excellent base for a dedicated application such as a database management system.

1. Introduction

The JASMIN *kernel* is a capability-based operating system that provides a minimal set of facilities for distributed applications [LEE84]. It offers three types of services: tasking, memory management and inter-process communication. It includes no database-specific features and even device drivers are not included. A primary goal of the JASMIN *database system* is to support high throughput transaction processing for a wide range of applications. It is designed as a set of cooperating software modules. These modules communicate using the kernel facilities which hide processor boundaries. Thus, modules can be easily moved to multiple processors and modules may be replicated to achieve higher performance.

The next two sections provide an overview of the JASMIN database system and the JASMIN kernel. Following that, we discuss implementing the database system on the kernel and describe the utility of various kernel features and what we found lacking. In a final section, we discuss the impact of the kernel design on system performance.

2. The JASMIN Database System

The JASMIN database system ([FIS84], [LAI84]) was designed to provide high-performance transaction processing over a wide range of applications. The approach taken was to functionally decompose the database manager into modules which communicate using a processor-transparent IPC mechanism. The modules can then be replicated and distributed across multiple processors for improved performance. Our decomposition consists of 3 modules representing increasing levels of abstraction: the Intelligent Store (IS) provides a page interface and transaction management, the Record Manager (RM) provides a record interface and access paths, and the Data Manager (DM) provides a relational model interface. Each module relies on the facilities provided by the lower level. In addition, each module's interface is user-accessible. Applications needing only page

access can access an IS directly and avoid the overhead of higher-level services. Finally, the system is easily reconfigured since the module interfaces are processor transparent.

2.1 Intelligent Store (IS)

The Intelligent Store ([ROO82]) offers a page interface to applications. It models the disk as a collection of *Logical Volumes* where each LV has some number of pages. An LV is simply a number of adjacent disk cylinders. However, an LV differs from a disk extent in that LV page numbers are logical, rather than physical. Thus, page *i* may be several cylinders away from page *i+1**. Further, the page size may vary across LV's allowing an LV containing an index to have larger pages, for example. The IS is transaction oriented and provides an optimistic, versioning concurrency control mechanism. Each transaction is guaranteed the view of the database (LV collection) as of the transaction's start time as if a snapshot had been taken. Conflicting updates to the same page are resolved by giving each transaction its own copy of the page. Note, this is how logically adjacent pages become physically distant. A transaction commits so long as its view of the database is still current at its commit time, i.e. during the lifetime of the transaction, no other transaction commited an update to a page accessed by the transaction. Note that, as in other optimistic schemes, transactions never block. However, our scheme has the additional advantage that read-only transactions always succeed since they always sees a consistent view of the database provided by the versioning scheme of the IS.

The IS provides a priority caching scheme for its buffers. This allows a database designer to specify higher priority for a particular LV (e.g. an LV containing index pages vs. one containing data pages). Also, the IS maintains a set of version counters that are used by applications to detect cache updates. For example, an RM will request the current value of a version counter after reading meta-data† into a local cache. Transactions which update meta-data will increment the corresponding version counter. The RM can detect an out-of-date cache by periodically checking the value of the version counter. Currently, we are working on an implementation to support distributed transactions.

2.2 Record Manager (RM)

The Record Manager ([LIN82]) maps records into IS pages. Records consist of any number of field-value pairs. Variable-length fields, missing values and repeating fields are all supported. Record types are defined by the user. Each record type has a primary (clustering) index and may have multiple secondary indices. The user must also specify the LV to contain the index pages and the LV for the data pages. Thereafter, the RM handles the details of mapping records into LV pages.

An RM request provides associative access to one record type at a time. Currently, B-trees are the access method used. Prefix and exact matching are supported as well as the standard comparison operations. The RM is transaction-oriented and a transaction may access multiple record types during its lifetime. The RM depends on the concurrency control facilities of the IS for database consistency.

2.3 Data Manager (DM)

The Data Manager (DM) provides a full relational interface to the user. It has a QUEL interface ([STO76]) and maps relations into RM records. In addition to the functions of query optimization

---

* Some control over placement is possible using "cells" which are not described here.

† By meta-data, we mean the stored schema that describes the database structure.

and planning, the DM uses the RM to process one or more single-relation queries. Complex queries are processed in a pipeline fashion with intermediate results from separate single-relation RM queries being fed, concurrently, to a network of DM tasks that communicate via shared memory. This avoids the need to create temporary relations.

3. JASMIN Kernel

The JASMIN kernel is a sparse operating system with little of the overhead (and few of the features) found in conventional operating systems. The motivation was to give the designer of a distributed application much control over how resources are allocated and scheduled. It basically provides three types of services: tasking, memory management and inter-process communication. These are discussed in more detail below.

3.1 Tasking

A JASMIN task represents a single thread of control. It consists of a stack, a priority, *private** data space and capabilities (described below). Tasks are created and destroyed dynamically. Task scheduling is by priority and is non-preemptive within a priority class. A task releases the processor only if it waits for a message or a system resource or if a higher priority task becomes runnable.

A JASMIN task executes in the address space of a *module*. A module consists of one or more associated tasks that share text (code) space and data space. A module facilitates concurrent activity within a shared data area, for example, the pipelined tasks within the DM. Modules are indivisible, i.e. all tasks in a module run on the same processor. However, several modules may share a processor. A task cannot tell if a task in a different module is on the same processor. So, programs cannot contain processor dependencies. By simply changing the assignment of modules to processors, the system is easily reconfigured with no change to any programs.

Note that there is no kernel facility to create or destroy a module. Such a facility would imply the existence of a disk server from which to load the module. And since the kernel itself contains no disk service, module creation must be a higher level service built on top of the kernel. A system is booted by downloading a large core image containing one or more modules and the kernel.

3.2 Memory Management

The JASMIN kernel provides a separate address space per task but does not do virtual memory management. Thus, the stack and private space of a task are hardware-protected from other tasks both within and without the same module. Note that since the kernel provides no disk service, it would be hard-pressed to do paging.

As mentioned above, a module consists of text space and data space shared by all tasks in the module. In addition, a module may have a pre-defined section of private data space that is allocated to each task. For example, variables containing the task identifier and task creation time might be in this *static* private space since these values are unique for each task. At task creation time, a task inherits the text space and shared data space of its parent. During execution, a task may create and destroy data segments dynamically. Such segments are normally private except that any child task created thereafter is also given access to the segment. However, there is no facility for true dynamic shared segments (e.g. between siblings).

---

* Like the stack, private data space is hardware-protected and is inaccessible to other tasks.

3.3 Communication

The JASMIN kernel provides a single message mechanism to be used for both synchronization and communication. These messages are small and fixed-length (16 bytes) and are sent along one-way communication channels, called paths. Our paths are similar to links in Roscoe [SOL79] and DEMOS [BAS77]. Paths are actually communication capabilities managed by the kernel. A task may allocate a path and pass it to another task in a message. This grants the receiving task the capability to send a message to the creator. The rights to duplicate or re-use a path are set when the path is created and may be further restricted whenever the path is duplicated.

Additionally, a path is created with a class and tag. There are 7 message classes which represent separate receive queues. They can be used for selective receive and to prioritize messages. A task can elect to wait for a message on any subset of the classes. When a message is delivered, the class and tag (of the path on which the message was sent) are returned to the receiving task. The tag is used to disambiguate messages for paths that use the same class.

The initial rendezvous between tasks is usually accomplished via the *Name Manager* task. A new task is typically created with a path to the Name Manager. Any task wishing to advertise services will register a path and a name with the Name Manager. Tasks requesting service can then get a copy of the server path from the Name Manager.

Large data transfers between tasks are done by attaching a buffer to a path. A separate kernel facility (*iomove*) is used to transfer data between the path buffer and a local buffer.

3.4 Services

The JASMIN kernel provides a few additional functions as aids to application developers. Among these are the local time-of-day clock, cpu-usage clock, primitives to print a string on the system console and to read a string from the console keyboard. There is also a routine to print status information about all tasks in the system.

As mentioned earlier, device drivers are not included in the kernel. Instead, drivers exist as services which advertise in the Name Manager. For example, requests to read and write disk pages are sent as messages to the disk server. The message includes a path with an attached buffer. The disk server then uses the client's buffer for the disk I/O operation. Note that some device drivers require low-level kernel functions unavailable to normal tasks. For example, the kernel must translate device interrupts into messages to the appropriate servers. So, a device driver must tell the kernel its interrupt level and give it a path on which to send an interrupt message. Also, some device drivers need to directly access path buffers, bypassing iomove (e.g. for DMA operations). Thus, there is a routine that returns the physical address for a path buffer.

4. Implementation Experience

In this section, we discuss our experience in using the JASMIN kernel to build the JASMIN database system. We adopt the categorization used in the previous section and, so, discuss the tasking, memory management and communication facilities of the kernel separately.

4.1 Tasking

The tasking facilities of the JASMIN kernel were quite satisfactory for our database application. The ability to multiprogram by using cooperating tasks in a module was exploited by all 3 modules.

The IS and DM modules used a fixed number of tasks while the number of RM tasks was load-dependent. In the DM and IS, a front-end task would receive all requests and then forward the request to a free worker task for processing. By contrast, in the RM the front-end task only received *start-transaction* requests. For each new transaction, it would create a separate task to process all subsequent requests for the transaction. Thus, the kernel provided the flexibility for both approaches*. Another good aspect of the kernel tasking facilities was that tasks (within a module) differed only in their stack and private space. This provides an opportunity for the kernel to optimize intra-module task switch time, since the memory management registers for text and shared data do not change. If only one module runs per processor, task switch time can be significantly reduced. However, the kernel definition does not describe this optimization (although some implementations have used it).

This brings up a minor complaint. The module notion is not formalized in the kernel definition. Formalizing a module might involve adding intra-module facilities (e.g. dynamic shared memory) and changing the semantics of existing facilities. These are things that could be added, of course. The important point is that a module is more than a collection of tasks as implied by the current definition. A module involves shared facilities, cooperating tasks and some overall module control in the form of a coordinator or scheduler. Future designs should recognize this and incorporate appropriate features.

The kernel provided non-preemptive scheduling which we feel is the correct choice for database managers. Besides the obvious advantage of preventing interference over the disk arm, it allowed us to write critical sections with no special synchronization mechanism. For example, the IS uses a single task to process all commit requests. This ensures no concurrent activity and facilitates making the commit atomic. However, in spite of the non-preemptive scheduling, it would have been useful to be able set a limit on processor usage per task to facilitate debugging (detection of infinite loops, etc.). There is a facility to send a message after some specified delay. But this is not sufficient since it requires that the receiving task have some control over the sending task (e.g. to interrupt or destroy it). There is no way to do this in JASMIN. This is another argument for adding module features to the kernel interface.

4.2 Memory Management

The kernel provides a separate address space per task but does not do virtual memory management. This is a good compromise for an experimental database management system. Separate address spaces is good software engineering, especially when several independent programmers are involved in a project. Good performance argues for no virtual memory management since unexpected paging can cause problems for query scheduling and execution. The notion of private space proved very useful. For example, the RM uses private space to store metadata and private buffers for transactions. The IS, which does not have a dedicated task per transaction cannot take advantage of this scheme since multiple requests from a transaction could be processed by different workers. Thus, the IS needs a more complex buffering scheme which is shared among the workers. And a transaction's data is not hardware protected as it is in the RM.

The ability to share data among tasks in a module proved very useful in managing the shared data buffers for the database. This was necessary in the IS, since, several worker tasks share a common data buffer. Buffer management is under the full control of the commit and worker tasks.

---

* The modules were written by separate programmers; thus the different task structures.

As mentioned earlier, a feature missing from the kernel is a primitive to create dynamic shared space. Such a facility could permit more efficient use of memory. Currently, users must preallocate all shared tables at link time so that they exist in static shared space. Thus, there is always enough space for the worst case load. But, this reduces the amount of space available for the buffer cache, etc.

4.3 Communication

The communication facilities of the JASMIN kernel are the most controversial. Some of the best aspects are the logical message addressing through the use of capabilities and the ability to perform a selective receive by specifying a subset of message classes. Logical addressing facilitates easy reconfiguration of the system by eliminating any processor dependencies in the programs. A client need not know the physical location of a server to obtain a connection. The ability to selectively receive a message from any one of multiple message queues was very useful. For example, the front-end task in an RM module may receive messages from either RM clients or RM worker tasks. Under normal load, it is willing to accept a message from either source. But, under conditions of heavy load, it must ignore requests from clients to start new transactions. The selective receive makes this simple. Some other systems decouple the selection from the receive (such as the *select* primitive in BSD 4.2) and, so, require an additional system call. Another nice feature of the JASMIN communication facilities is that it uses a send-receive protocol as opposed to a call/return protocol* (which is used in the V-Kernel [CHE83]). This makes it easier to implement asynchronous processing as is required for two-phase commit.

One major drawback of the JASMIN IPC design is that communication capabilities are many-to-one, i.e. there can be multiple senders on a path but only one task can receive on the path. A consequence of this is that multi-cast messages are impossible to send. But, more importantly, when calling a server, it adds the overhead of an extra message to process a request. Recall that the IS and DM task structures have a front-end task and multiple worker tasks. Requests must first be directed to the front-end which forwards them to a free worker.

A better solution would be to have the worker tasks share a request queue and just remove the next request themselves. But since JASMIN has no shared receive queues (capabilities are owned by only one task) the worker tasks cannot do this. As an aside, note that the RM task structure does not have this problem. The RM has a dedicated task per transaction so requests go directly to the worker. On the other hand, it has the serious limitation that the RM can only process one request per transaction at-a-time.

Another nuisance of the IPC design is that only one capability path can be passed per message. Normally, this is sufficient. But there are situations where two paths are needed. Any request that needs two separate buffers requires an additional message exchange to send the capability for the second buffer. For example, the RM needs to read both the metadata and cache version numbers in one request and these are stored in different buffers. Actually, JASMIN does provide a way to package a collection of paths into one capability called a path-list. To our knowledge, the facility is unique among distributed kernels. However, it is not a good solution for this situation due to the overhead of packing and unpacking the path-list. This situation occurs frequently enough that it merits special attention.

---

* Actually, the mcall and iomcall primitives provide a call/return facility.

A final drawback is that there is no low-cost synchronization facility (such as semaphores or locks). Thus, intra-module synchronization must be done with messages which incurs more overhead than necessary.

5. Performance Implications

Probably the largest impact on performance was the fact that all communication was done via capabilities, i.e. secure communications channels. A seemingly innocent activity such as passing a capability in a message could result in a flurry of activity by the kernel, especially if the sending task, receiving task and the capability owner are all on separate machines. More specifically, the overhead is not so much due to capability maintenance but to the fact that capabilities require a connection-based communications protocol. Passing, duplicating and destroying paths requires creating a channel to the path owner to maintain reference counts, sequence numbers, etc.. And, this (at times, significant) overhead is hidden from the kernel user since one cannot tell who owns the path. A clever implementation can piggy-back the bookkeeping information onto other messages. But this only works in certain situations. Connections work well when the amount of communication exceeds the overhead of setting up and destroying the channel. In our application, much of the communication between modules is one-shot so the overhead of maintaining the connections was often wasted.

And, it is not clear that much was gained by the use of capabilities. They are elegant and in some cases simplified the programming. But in a dedicated application with 'trusted' code such as a database management system, the protection they offer seems not worth the cost.

Another performance drain was the decision to implement device drivers as servers. Although this simplified the implementation somewhat and kept the kernel simple it was at great cost. For example, consider the TTY driver which receives an interrupt per character. Since processing the interrupt involves at least two messages, this adds an overhead of several milliseconds to the interrupt processing time. Similar results hold for the disk driver. In fact, for clients of the disk server, we found the disk latency to be approximately 3/4's of a disk revolution for reading consecutive disk sectors.

Originally, it was hoped and expected that message passing would be fast enough that this overhead would not be a problem. Unfortunately, that was not the case. In our prototype system, performance was not the highest priority and future implementations will do better. Although, it is an open issue if connection-based protocols can ever be competitive with connection-less protocols for this type of application. Similarly, it was hoped that task creation time would be fast. Thus, the RM designer was willing to accept the overhead of creating a new task for each transaction in order to simplify the programming chore. In retrospect, this was not a good choice in our environment since transactions tend to be short.

It is worth commenting on the separate facilities for data transfer (iomove) and communication (sendmsg) in JASMIN. An alternative design would have been a single communication mechanism that supported variable-length messages. Clearly, the iomove facility simplifies the kernel since it never has to buffer large amounts of data and all messages are fixed-length. It is claimed to be a performance winner, as well. With iomove, a task never gets data until it is ready. This saves memory on the receiver side and facilitates DMA between sender and receiver if the hardware is capable. It is not entirely clear if it really is a performance advantage, however. Iomove essentially requires two synchronization points to transfer data; one message from sender to receiver to indicate data is available and a second from receiver to sender indicating the transfer is complete.

Using variable-length messages, there is just one synchronization point. Thus, the tradeoff is between one extra message and memory space (and a more complicated kernel).

In summary, the kernel provided an excellent development environment. It facilitates fast prototyping and the tasking and memory management were just what was needed. The communication facilities are simple and elegant, but, perhaps are too general for high-performance applications.

6. Acknowledgements

Thanks to Haran Boral and Premkumar Uppaluru for reading a draft of this paper and making many helpful suggestions to improve the content and readability. Premkumar and Hikyu Lee ported the JASMIN kernel to our current hardware at Bellcore.

7. References

[BAS77] Baskett, FH, Howard, JH, Montague, JT, Task Communication in DEMOS. Proceedings of the 6th ACM Symposium on Operating System Principles, November, 1977, pp. 23-31.

[CHE83] Cheriton, DR, Zwaenepoel, W, The Distributed V Kernel and Its Performance for Diskless Workstations. Report No. STAN-CS-83-973, Stanford University, July, 1983.

[FIS84] Fishman, DH, Lai, MY, Wilkinson, WK, An Overview of the JASMIN Database Machine. Proceedings of the ACM SIGMOD Conference, Boston, MA, June, 1984, pp.234-239.

[LAI84] Lai, M. Y., Wilkinson, W. K. Distributed Transaction Management in JASMIN, VLDB 84, August 1984.

[LEE84] Lee, H, Premkumar, U, The Architecture and Implementation of Distributed JASMIN Kernel. Bell Communications Research Technical Memo, TM-ARH-000324, October, 1984, Morristown, N.J.

[LIN82] Linderman, J. P. Issues in the Design of a Distributed Record Management System, Bell System Technical Journal 61, 9 (Nov. 1982), Part 2, 2555-2566.

[ROO82] Roome, W. D. A Content-Addressable Intelligent Store, Bell System Technical Journal 61, 9 (Nov. 1982), Part 2, 2567-2596.

[SOL79] Solomon, MH, Finkel, RA, The Roscoe Distributed Operating System. Proceedings of the 7th Symposium on Operating Systems Principles, December, 1979, pp.108-114.

[STO76] Stonebraker, M., Wong, E., Kreps, P., and Held, G., The Design and Implementation of INGRES, ACM TODS 1, 3 (Sept. 1976), 189-222.

Supporting a Database System on Symbolics Lisp Machines

Hong-Tai Chou, Jorge F. Garza, Nat Ballou

Microelectronics and Computer Technology Corporation
3500 West Balcones Center Drive
Austin, Texas 78759

ABSTRACT

We examine a number of important issues related to the implementation of a prototype object-oriented database system on a Symbolics 3600 Lisp machine. Our main focus is on interfacing with the storage system, memory management, and process control of the Symbolics operating system. We discuss various implementation problems that we have encountered and present our solutions.

1. Introduction

ORION is a prototype object-oriented database system under development at MCC to support the data management needs of object-oriented applications from the CAD/CAM, AI, and OIS domains [BANE87]. The intended applications for ORION imposed two types of requirements: advanced functionality and high performance. The ORION architecture was designed to satisfy these requirements. ORION provides a number of advanced features that conventional commercial database systems do not, including version control and change notification [CHOU86], storage and presentation of unstructured multimedia data [WOEL86], and dynamic changes to the database schema [BANE86]. For high performance, ORION supports efficient access paths, and novel and efficient techniques for query processing, buffer management, and concurrency control.

ORION is being implemented in Common Lisp [STEE84] on a Symbolics 3600 machine [SYMB85a]. Supporting an object-oriented database system in a Lisp environment presented a unique challenge. The general problem of supporting a database system in a Lisp environment is twofold. First, using Lisp as an implementation language presents some technical difficulties. Lisp was not originally designed for system programming. Low-level manipulation of primitive data, such as pointers and bytes, requires careful coding. Second, storing Lisp objects efficiently is a challenge to a database system, mainly because Lisp objects are so dynamic in terms of data type and length. We shall focus our discussion on the latter, since the former can be solved more easily by coding techniques. In particular, we will examine implementation issues related to the storage system, memory management, and process control of the Symbolics operating system.

2. Storing Lisp Objects

The Symbolics machine provides three levels of storage interface to users (Figure 1): the *LMFS* (Lisp Machine File System), the *FEP* (Front-End Processor) file system, and the *user-disk* system [SYMB85c]. Internally, the Symbolics machine also implements a *system-disk* component to support paging of its virtual memory.

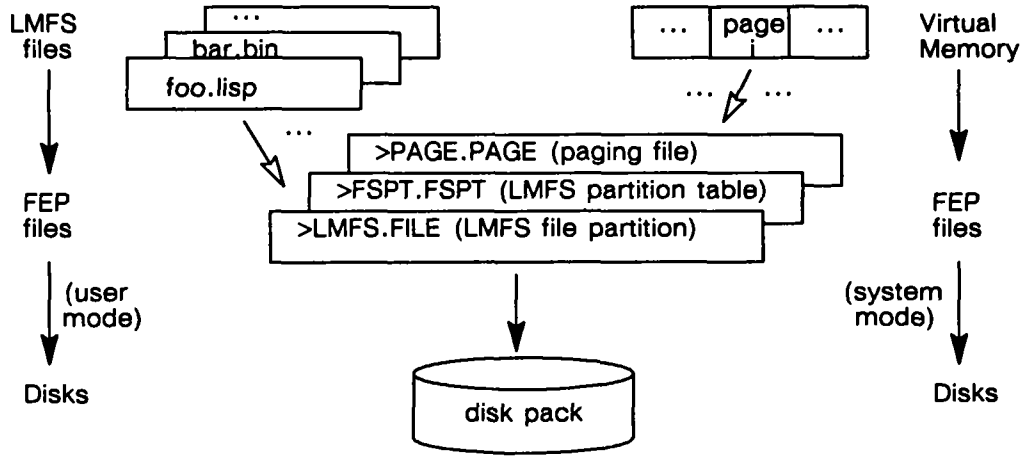

Figure 1. Storage Organization of Symbolics

Lisp Machine File System (LMFS)

LMFS provides a file stream interface through which a Lisp object can be stored in the form of printed text, called its *printed representation*. It is built on top of the FEP file system. Normally, all LMFS files are stored in a big FEP file, "LMFS.FILE", which is the default LMFS file partition (file system space). Alternative or additional file partitions, each implemented as a FEP file, can be specified in the FEP file "FSPT.FSPT". An object stored in a file stream is retrieved as a string of characters and reconstructed into an in-memory Lisp object. As shown in Figure 2, a typical in-memory representation of a

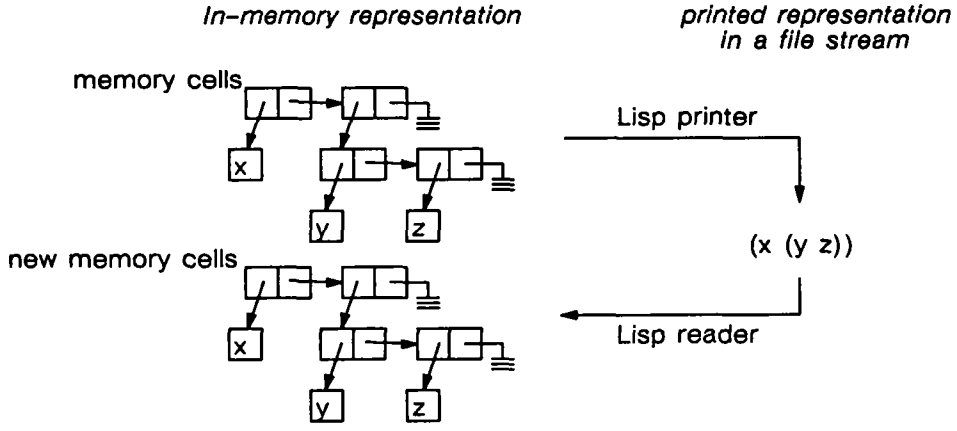

Figure 2. Representation of a Lisp Object (x (y z))

nested list (x(y z)) is a collection of memory cells connected through pointers. The list can be written to a file stream by the *Lisp printer*, and later be read in and reconstructed (in a new collection of memory cells) by the *Lisp reader*. For database applications, there are several functional and performance problems with file streams:

(1) Some objects do not have a printed representation from which the objects can be reconstructed in memory. For example, the printed representation of compiled code can not be read in.

(2) Storing objects in a file stream is expensive as each in-memory object has to be translated into a text string before it can be stored.

(3) Certain Lisp objects have a lengthy printed representation. For example, storing a bit vector as a sequence of one's and zero's in the text form is obviously inefficient.

(4) Although random accesses are possible with a file stream, storing an object at an arbitrary location beyond the end of the stream is not allowed. (In other words, a file stream can not contain "holes" in which no data has been written.) This makes updating an object difficult, because we can not reserve space in anticipation of future growth of the object.

(5) Although logically contiguous, the actual data of a file stream may be scattered on disk. Since Lisp programmers have no control over physical placement of objects, the ability to cluster objects on disk is severely limited. Further, buffering of disk blocks for a file stream is transparent to the programmers. Thus, the ability to prefetch objects and to exploit any intelligent buffering algorithm is lost if file streams are used.

Although binary LMFS files are also provided to help overcome some of these problems, the byte-at-a-time interface (through which one byte is transferred at a time) makes it too inefficient for dealing with large amounts of data. Besides, this binary file facility is not very useful since only integers can be stored.

Symbolics FEP File and Disk Systems

The FEP file system is a low-level storage system that deals primarily with blocks of data. It provides an interface that allows un-buffered reading and writing of disk blocks (i.e. without intermediate buffering). Such an IO operation can be either synchronous (blocking) or asynchronous (non-blocking). Besides providing file space for LMFS, FEP files are used internally by the operating system for storing system data, such as paging files and boot commands. Underneath the FEP file system is the Symbolics Disk System, which supports two modes of disk transfer, *user mode* for normal file storage and *system mode* for virtual memory support. Both the FEP file and user-disk interface require a system resource, *disk array*, for buffering disk blocks. A disk array is an array of fixnums plus some disk related header information, and resides in virtual memory. (A fixnum is an integer which has an efficient and typically fixed-size representation in a Common Lisp implementation.)

Figure 3 shows a typical data flow during retrieval of a Lisp object from an LMFS file. First, LMFS translates the current file position (within a file stream) into a block number (in a FEP file), and calls the FEP file system to retrieve the data block into a disk array. The FEP file system, in turn, looks up the physical disk address of the data block and issues a read request to the disk driver. When the disk-read operation completes, the FEP file system returns control to LMFS, which then extracts the relevant data (in the text form) from the disk array and transforms it into an in-memory representation of the Lisp object. Note that the disk array may have to be paged in through the system disk interface before the data block can be retrieved.

Building a Database Storage Subsystem

It is desirable to integrate the storage subsystem of a database system with that of the Symbolics machine to minimize memory management overhead, such as paging. However, this requires modifica-

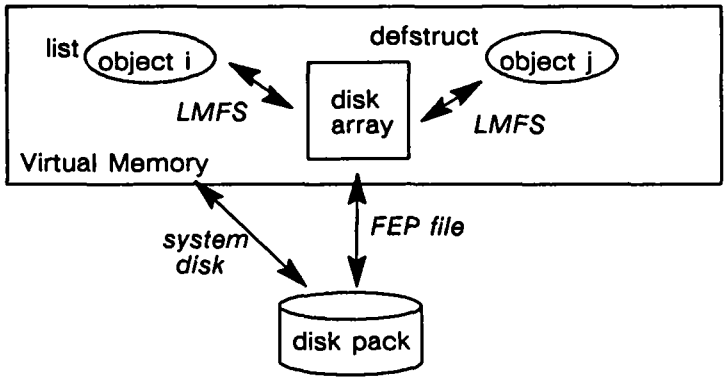

Figure 3. Data Flow in the Symbolics Storage Hierarchy

tion to the Symbolics operating system. To avoid such a major undertaking, we have chosen the FEP file system as our primary "disk" interface. However, creating a FEP file is an expensive operation. Further, performance of the FEP file system may be severely degraded if a disk is fragmented by a large number of small FEP files. Thus, to reduce FEP system overhead and disk fragmentation, we allocate (semi-permanently) a large FEP file as a virtual disk. Similar to LMFS, we implement our database storage subsystem on top of the FEP file system, and take control of page allocation within the virtual disk.

There are two Symbolics utility programs that *dump* and *load* Lisp objects to and from a FEP file in a binary format [SYMB85b]. However, the inability to store a Lisp object at a prescribed position (other than zero) in a file, among other deficiencies, makes them unsuitable for direct use by a database system. Therefore, using these two utilities as the basis, we implemented our own *dumper* and *loader*. The *dumper* encodes a Lisp object into one or more 16-bit packets. The type of the object, and its value or other information about the object, are recorded in the first packet. Additional packets may follow to provide space for storing the object value if it can not fit in the first packet. In general, objects are more compact in the disk format. A small integer, for example, can be encoded in a single packet. With these two utilities, we can place an object anywhere on a disk page according to our clustering scheme. More importantly, it allows us to update part of an object in place without extensive storage reorganization.

3. Managing Objects in Memory

The Symbolics main memory, as presented to the users, is organized as a hierarchy [SYMB85c]. At the top level, the virtual memory is divided into a number of *areas* in which related objects are stored. Each area can have its own paging and garbage collection algorithms, thus enabling knowledgeable users to fine-tune management of main memory. An *area* is further divided into a number of *regions*, each of which contains data of the same representation type. On the basis of its representation type, a Lisp object occupies a number of 36-bit memory words. Each memory word consists of three fields: cdr-code, data type, and pointer or data value. (Cdr-code is a two-bit field for optimizing memory representation of lists.)

Virtual Memory Management

The Symbolics machine allows a user to have certain control over virtual memory management, including garbage collection and resource allocation. A *resource* is basically a data structure definition

plus a collection of instances, which can be allocated to a user upon request. Of particular interest to us is the disk-array resource, which is required as block buffers for the FEP file system. The Symbolics resource manager keeps track of disk-arrays to make them re-usable. Data contained in a disk array, however, is lost once the disk array is returned to the resource manager. Thus, we have implemented our own page buffer manager which keeps track of the buffers allocated and the identities of the pages in the buffers. Initially, page buffers are allocated from the disk-array resource. An allocated buffer is not released even when the page in it is no longer in use, so that the page may be re-used by others. Further, a buffer may be re-assigned to another page by the replacement algorithm. In any event, the buffers are under the control of the page buffer manager until the database system is shut down, at which time they are returned to the disk-array resource.

Physical Memory Management

The Symbolics operating system also allows users to participate to some extent in physical (real) memory management, in particular, the paging algorithm. An object or a section of the virtual memory space can be swapped in or out under user control. In addition, an object or a section of the virtual memory space can be *wired* down (locked) in physical memory. Wiring memory is necessary, for example, to pin down a disk array during an IO operation. A user can also use this feature to "force" important or critical data structures to stay in physical memory to avoid performance degradation due to paging. We use this feature to wire certain important page buffers in physical memory.

4. Process Management and Synchronization

On the Symbolics machine, *stack groups* (primitive coroutines) are used to support multiprogramming. A stack group holds the state information, including a *control stack* and a *binding stack*, for its associated computation. A *control stack* is a stack of function calls. It keeps track of the current running function, its caller, and so on, and the return address of each function on the stack. A *binding stack* (environment stack) contains all the values saved by *Lambda*-binding of special variables [STEE84]. At any time, the Symbolics machine performs a computation that is associated with a stack group. Control over the Symbolics machine can be passed (switched) from the current stack group to another stack group through *resumption*.

Processes on Symbolics are implemented with stack groups. A process can be either *active* (ready to run) or *stopped* (waiting for an event, such as the completion of an IO operation). The active processes are managed by a special stack group, called the *scheduler*, which repeatedly cycles through the list of active processes and determines which process should run next. The *current* process, i.e. the process that is running, continues its computation until it decides to wait, or a system interrupt occurs. In either case, the scheduler is resumed to select another process to run.

All Symbolics processes share the same virtual address space. With the exception of process state information, which is kept in each individual stack group, all the Lisp data and global variables are shared among processes. The ability to share memory address space is important for efficient implementation of common data structures, such as page buffer pool. Direct sharing of in-memory data structures, however, presents some synchronization problems. Mutual exclusion is necessary to protect the consistency of critical system data structures (tables). The Symbolics machine provides a special type of resource called *locks*, which are similar to semaphores. Concurrent database processes follow a simple locking protocol to synchronize with one another. When a database process needs to access a shared data structure, it must first acquire the lock on it. As soon as the process is done with the data structure, it must release the lock so that another process waiting for the same lock can proceed.

5. Conclusions

Building a database system on a Lisp machine is a unique experience. Although the basic architecture and implementation techniques for a database system are mostly unaffected by the Lisp environment, storing Lisp objects on secondary storage is a problem that requires special considerations. The need for transforming a Lisp object between the in-memory representation and disk representation introduces an additional cost factor which must be weighed carefully to avoid potential performance problems. We plan to conduct a performance study to thoroughly evaluate the ORION system with the view to validate our design.

Acknowledgments

We thank Won Kim, our project leader, for his contribution and helpful comments on drafts of this paper. Other members of the project, Jay Banerjee and Darrell Woelk, have also contributed to the work reported here.

References

[BANE86] Banerjee, J., H.J. Kim, W. Kim, and H.F. Korth, "Schema Evolution in Object-Oriented Persistent Databases", in *Proc. 6th Advanced Database Symposium*, Tokyo, Japan, August 1986.

[BANE87] Banerjee, J., N. Ballou, H.T. Chou, J. Garza, W. Kim, and D. Woelk, "Database Support for Object-Oriented Applications", to appear in *ACM Trans. on Office Information Systems*, April 1987.

[CHOU86] Chou, H.T., and W. Kim, "A Unifying Framework for Version Control in a CAD Environment", in *Proc. Int'l Conf. on Very Large Data Bases*, August 1986, Kyoto, Japan.

[STEE84] Steele, Guy L. Jr., Scott E. Fahlman, Richard P. Gabriel, David A. Moon, and Daniel L. Weinreb, "Common Lisp", *Digital Press*, 1984.

[SYMB85a] Symbolics Inc., "User's Guide to Symbolics Computers", *Symbolics Manual #* 996015, March 1985.

[SYMB85b] Symbolics Inc., "Program Development Utilities", *Symbolics Manual #* 996045, February 1985.

[SYMB85c] Symbolics Inc., "Internals, Processes, and Storage Management", *Symbolics Manual #* 996085, March 1985.

[WOEL86] Woelk, D., W. Kim, and W. Luther, "An Object-Oriented Approach to Multimedia Databases", in *Proc. ACM SIGMOD Conf. on the Management of Data*, May 1986, Washington D.C..

The Camelot Project[1]

**Alfred Z. Spector, Joshua J. Bloch, Dean S. Daniels,
Richard P. Draves, Dan Duchamp, Jeffrey L. Eppinger,
Sherri G. Menees, Dean S. Thompson**

Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA

Abstract

Camelot provides flexible and high performance transaction management, disk management, and recovery mechanisms that are useful for implementing a wide class of abstract data types, including large databases. To ensure that Camelot is accessible outside of the Carnegie Mellon environment, Camelot runs on the Unix-compatible Mach operating system and uses the standard Arpanet IP communication protocol. Camelot is being coded on RT PC's, is being frequently tested on MicroVaxes, and it will also run on various shared-memory multiprocessors. This paper describes Camelot's functions and internal structure.

1. Introduction

Distributed transactions are an important technique for simplifying the construction of reliable and available distributed applications. The failure atomicity, permanence, and serializability properties provided by transactions lessen the attention a programmer must pay to concurrency and failures [Gray 80, Spector and Schwarz 83]. Overall, transactions make it easier to maintain the consistency of distributed objects.

Many commercial transaction processing applications already use distributed transactions, for example, on Tandem's TMF [Helland 85]. We believe there are many more algorithms and applications that will benefit from transactions as soon as there is a widespread, general-purpose, and high performance transaction facility to support them. For example, there are a plethora of unimplemented distributed replication techniques that depend upon transactions to maintain invariants on the underlying replicas.

A few projects have developed systems that support distributed transaction processing on abstract objects. Argus, Clouds, and TABS [Liskov and Scheifler 83, Allchin and McKendry 83, Spector et al. 85, Spector 85] are a few examples. These systems permit users to define new objects and to use

---

[1]This work was supported by the Defense Advanced Research Projects Agency, ARPA Order No. 4976, monitored by the Air Force Avionics Laboratory under Contract F33615-84-K-1520, and the IBM Corporation.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of any of the sponsoring agencies or the US government.

them together within transactions. While the interfaces, functions, and implementation techniques of Argus, Clouds, and TABS are quite different, the projects' goals have been the same: to provide a common transactional basis for many abstractions with the ultimate goal of simplifying the construction of reliable distributed applications.

Building on the experience of these and other projects, we have designed and are now implementing an improved distributed transaction facility, called Camelot (Carnegie Mellon Low Overhead Transaction Facility). Camelot provides flexible and efficient support for distributed transactions on a wide variety of user-defined objects such as databases, files, message queues, and I/O objects. Clients of the Camelot system encapsulate objects within *data server* processes, which execute operations in response to remote procedure calls. Other attributes of Camelot include the following:

- **Compatibility with standard operating systems.** Camelot runs on Mach, a Berkeley 4.3 Unix™-compatible operating system [Accetta et al. 86]. Mach's Unix-compatibility makes Camelot easier to use and ensures that good program development tools are available. Mach's support for shared memory, message passing, and multiprocessors makes Camelot more efficient and flexible.

- **Compatibility with Arpanet protocols.** Camelot uses datagrams and Mach messages, both of which are built on the standard Arpanet IP network layer [Postel 82]. This will facilitate large distributed processing experiments.

- **Machine-independent implementation.** Camelot is intended to run on all the uniprocessors and multiprocessors that Mach will support. We develop Camelot on IBM RT PC's, but we frequently test it on DEC MicroVaxes and anticipate running it on multiprocessors such as the Encore and Sequent machines.

- **Powerful functions.** Camelot supports functions that are sufficient for many different abstract types. For example, Camelot supports both blocking and non-blocking commit protocols, nested transactions as in Argus, and a scheme for supporting recoverable objects that are accessed in virtual memory. (Section 2 describes Camelot's functions in more detail.)

- **Efficient implementation.** Camelot is designed to reduce the overhead of executing transactions. For example, shared memory reduces the use of message passing; multiple threads of control increases parallelism; and a common log reduces the number of synchronous stable storage writes. (Section 3 describes Camelot's implementation in more detail.)

- **Careful software engineering and documentation.** Camelot is being coded in C in conformance with careful coding standards [Thompson 86]. This increases Camelot's portability and maintainability and reduces the likelihood of bugs. The internal and external system interfaces are specified in the Camelot Interface Specification [Spector et al 86], which is then processed to generate Camelot code. A user manual based on the specification will be written.

To reduce further the amount of effort required to construct reliable distributed systems, a companion project is developing a set of language facilities, called Avalon, which provide linguistic

support for reliable applications [Herlihy and Wing 86]. Avalon encompasses extensions to C++, Common Lisp, and ADA and automatically generates necessary calls on Camelot. Figure 1-1 shows the relationship of Camelot to Avalon and Mach.

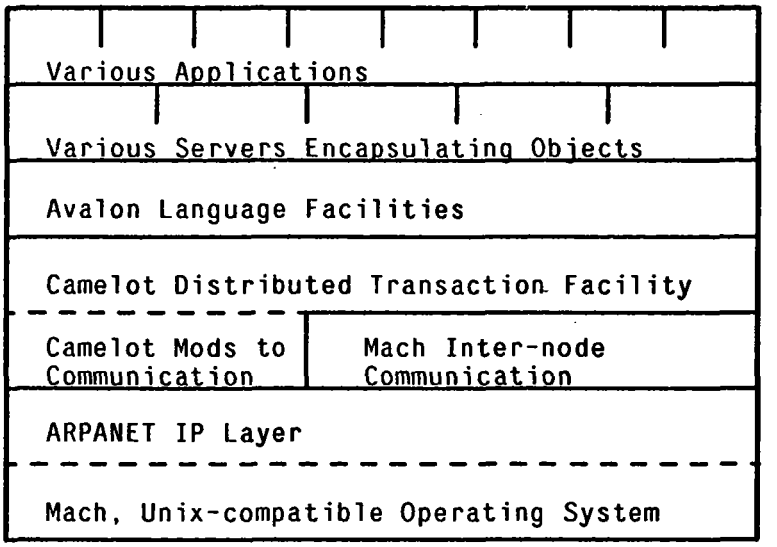

Figure 1-1: Relationship of Camelot to Other System Layers

Mach executes on uniprocessor and multiprocessor hardware. Inter-node communication is logically layered on top of Mach. Camelot provides support for transaction processing, including certain additions to the communication layer. Avalon provides linguistic support for accessing Camelot and Mach. Users define servers encapsulating objects and applications that use those objects. Examples of servers are mail repositories, distributed file system components, and database managers.

One goal of the Camelot Project is certainly the development of Camelot; that is, a system of sufficient quality, performance, and generality to support not only our own, but others' development of reliable distributed applications. In building Camelot, we hope to demonstrate conclusively that general purpose transaction facilities are efficient enough to be useful in many domains. However, we are also developing new algorithms and techniques that may be useful outside of Camelot. These include an enhanced non-blocking commit protocol, a replicated logging service, and a facility for testing distributed applications. We also expect to learn much from evaluating Camelot's performance, particularly with respect to the performance speed-up on multiprocessors.

2. Camelot Functions

The most basic building blocks for reliable distributed applications are provided by Mach, its communication facilities, and the Matchmaker RPC stub generator [Accetta et al. 86, Cooper 86, Jones et al. 85]. These building blocks include processes, threads of control within processes, shared memory between processes, and message passing.

Camelot provides functions for system configuration, recovery, disk management, transaction management, deadlock detection, and reliability/performance evaluation[2]. Most of these functions are specified in the Camelot Interface Specification and are part of Camelot Release 1. Certain more advanced functions will be added to Camelot for Release 2.

2.1. Configuration Management

Camelot supports the dynamic allocation and deallocation of both new data servers and the recoverable storage in which data servers store long-lived objects. Camelot maintains configuration data so that it can restart the appropriate data servers after a crash and reattach them to their recoverable storage. These configuration data are stored in recoverable storage and updated transactionally.

2.2. Disk Management

Camelot provides data servers with up to 2^{48} bytes of recoverable storage. With the cooperation of Mach, Camelot permits data servers to map that storage into their address space, though data servers must call Camelot to remap their address space when they overflow 32-bit addresses. To simplify the allocation of contiguous regions of disk space, Camelot assumes that all allocation and deallocation requests space are coarse (e.g., in megabytes). Data servers are responsible for doing their own microscopic storage management.

So that operations on data in recoverable storage can be undone or redone after failures, Camelot provides data servers with logging services for recording modifications to objects. Camelot automatically coordinates paging of recoverable storage to maintain the write-ahead log invariant [Eppinger and Spector 85].

2.3. Recovery Management

Camelot's recovery functions include transaction abort, and server, node, and media-failure recovery. To support these functions, Camelot Release 1 provides two forms of write-ahead value logging; one form in which only new values are written to the log, and a second form in which both old values and new values are written. New value logging requires less log space, but results in increased paging for long running transactions. This is because pages can not be written back to their home location until a transaction commits. Camelot assumes that the invoker of a top-level transaction knows the approximate length of his transaction and specifies the type of logging accordingly.

Camelot's two logging protocols are based on the old value/new value recovery technique used in TABS [Spector 85] and described by Schwarz [Schwarz 84]. However, they have been extended to

---

[2]Synchronization mechanisms for preserving serializability are distributed among data servers; Camelot supports servers that perform either locking or hybrid atomicity [Herlihy 85]. This synchronization is commonly implemented with the assistance of Avalon's runtime support.

support aborts of nested transactions, new value recovery, and the logging of arbitrary regions of memory.

Camelot writes log data to locally duplexed storage or to storage that is replicated on a collection of dedicated network log servers [Daniels et al. 86]. In some environments, the use of a shared network logging facility could have survivability, operational, performance, and cost advantages. Survivability is likely to be better for a replicated logging facility because it can tolerate the destruction of one or more entire processing nodes. Operational advantages accrue because it is easier to manage high volumes of log data at a small number of logging nodes, rather than at all transaction processing nodes. Performance might be better because shared facilities can have faster hardware than could be afforded for each processing node. Finally providing a shared network logging facility would be less costly than dedicating duplexed disks to each processing node, particularly in workstation environments.

Release 2 of Camelot will support an operation (or transition) logging technique in which type implementors can log non-idempotent undo and redo operations. This type of logging increases the feasible concurrency for some types and reduces the amount of log space that they require.

2.4. Transaction Management

Camelot provides facilities for beginning new top-level and nested transactions and for committing and aborting them. Two options exist for commit: *Blocking* commit may result in data that remains locked until a coordinator is restarted or a network is repaired. *Non-blocking* commit, though more expensive in the normal case, reduces the likelihood that a node's data will remain locked until another node or network partition is repaired. In addition to these standard transaction management functions, Camelot provides an inquiry facility for determining the status of a transaction. Data servers and Avalon need this to support lock inheritance.

2.5. Support for Data Servers

The Camelot library packages all system interfaces and provides a simple locking mechanism. It also contains routines that perform the generic processing required of all data servers. This processing includes participating in two-phase commit, handling undo and redo requests generated after failures, responding to abort exceptions, and the like. The functions of this library are subsumed by Avalon's more ambitious linguistic support.

2.6. Deadlock Detection

Clients of Camelot Release 1 must depend on time-out to detect deadlocks. Release 2 will incorporate a deadlock detector and export interfaces for servers to report their local knowledge of wait-for graphs. We anticipate that implementing deadlock detection for arbitrary abstract types in a large network environment like the Arpanet will be difficult.

2.7. Reliability and Performance Evaluation

Camelot Release 2 will contain a facility for capturing performance data, generating and distributing workloads, and inserting (simulated) faults. These capabilities will help us analyze, tune, and validate Camelot and benefit Camelot's clients as they analyze their distributed algorithms. The information returned by the facility could also be used to provide feedback for applications that dynamically tune themselves. We believe that, when properly designed, a reliability and performance evaluation facility will prove as essential for building large distributed applications as source-level debuggers are essential for traditional programming.

The reliability and performance evaluation facility has three parts. The first captures performance data and permits clients to gauge critical performance metrics, such as the number of messages, page faults, deadlocks, and transactions/second. Certain information is application-independent, but other useful information depends on the nature of the application. Therefore, the performance evaluation facility will be extensible and capture application-specific data from higher level components. Once information is obtained from various nodes on the system, the facility combines and presents it to system implementors or feeds it back to applications for use in dynamic tuning.

The second part of the performance and reliability evaluation facility permits the distribution of applications (or workloads) on the system. When many nodes are involved in a workload, this task can be very difficult unless it is possible to specify the nodes and workloads from a single node. We have built a prototype facility of this type for TABS, and we will extend it for use on Camelot.

The third part permits simulated faults to be inserted according to a pre-specified distribution. This is crucial for understanding the behavior of a system in the presence of faults. For example the low-level communication software may be instructed to lose or reorder datagrams with a pre-specified probability. Or, a pair of nodes could greatly raise network utilization to probe the effects of contention.

2.8. Miscellaneous Functions

Camelot provides both a logical clock [Lamport 78] and a synchronized real-time clock. These clocks are useful, for example, to support hybrid atomicity [Herlihy 85] and replication using optimistic timestamps [Bloch 86]. Camelot also extends the Mach naming service to support multiple servers with the same name. This is useful to support replicated objects.

3. Camelot Implementation

The major functions of Camelot and their logical relationship is illustrated in Figure 3-1. Disk management and recovery management are at the base of Camelot's functions. Both activities are local to a particular node, except that recovery may require communication with the network logging service. Deadlock detection and transaction management are distributed activities that assume underlying disk management and node recovery facilities. Communication protocols and reliability

and performance evaluation are implemented within many levels of the system. The library support for data servers rests on top of these functions.

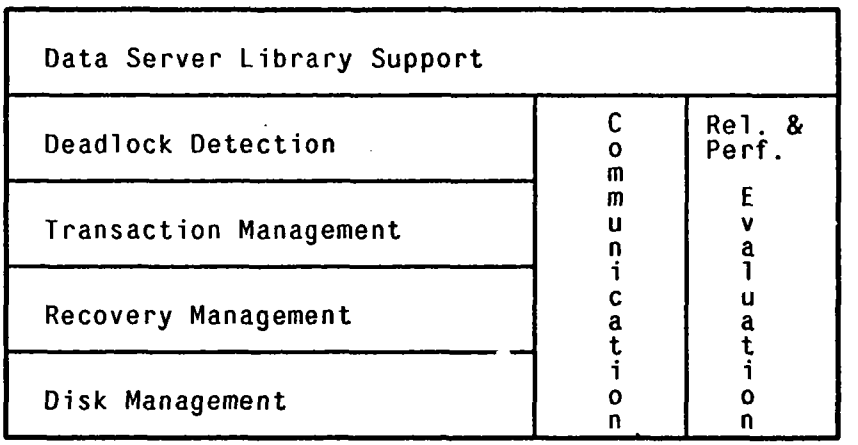

Figure 3-1: Logical Components of Camelot

This figure describes the logical structure of Camelot. Camelot is logically hierarchical, except that communication and reliability and performance evaluation functions span multiple levels.

All of Camelot except the library routines is implemented by a collection of Mach processes, which run on every node. Each of these processes is responsible for supporting a particular collection of functions. Processes use threads of control internally to permit parallelism. Calls to Camelot (e.g., to begin or commit a transaction), must be directed to a particular Camelot process. Some frequently called functions such as log writes are invoked by writing to memory queues that are shared between a data server and a Camelot process. Other functions are invoked using messages that are generated by Matchmaker.

Figure 3-2 shows the seven processes in Release 1 of Camelot[3]: master control, disk manager, communication manager, recovery manager, transaction manager, node server, and node configuration application.

- **Master Control.** This process restarts Camelot after a node failure.
- **Disk Manager.** The disk manager allocates and deallocates recoverable storage, accepts and writes log records locally, and enforces the write-ahead log invariant. For log records that are to be written to the distributed logging service, the disk manager

[3]Camelot Release 2 will use additional processes to support deadlock detection and reliability and performance evaluation.

works with dedicated servers on the network. Additionally, the disk manager writes pages to/from the disk when Mach needs to service page faults on recoverable storage or to clean primary memory. Finally, it performs checkpoints to limit the amount of work during recovery and works closely with the recovery manager when failures are being processed.

- **Communication Manager.** The communication manager forwards inter-node Mach messages, and provides the logical and physical clock services. In addition, it knows the format of messages and keeps a list of all the nodes that are involved in a particular transaction. This information is provided to the transaction manager for use during commit or abort processing. Finally, the communication manager provides a name service that creates communication channels to named servers. (The transaction manager and distributed logging service use IP datagrams, thereby bypassing the Communication Manager.)

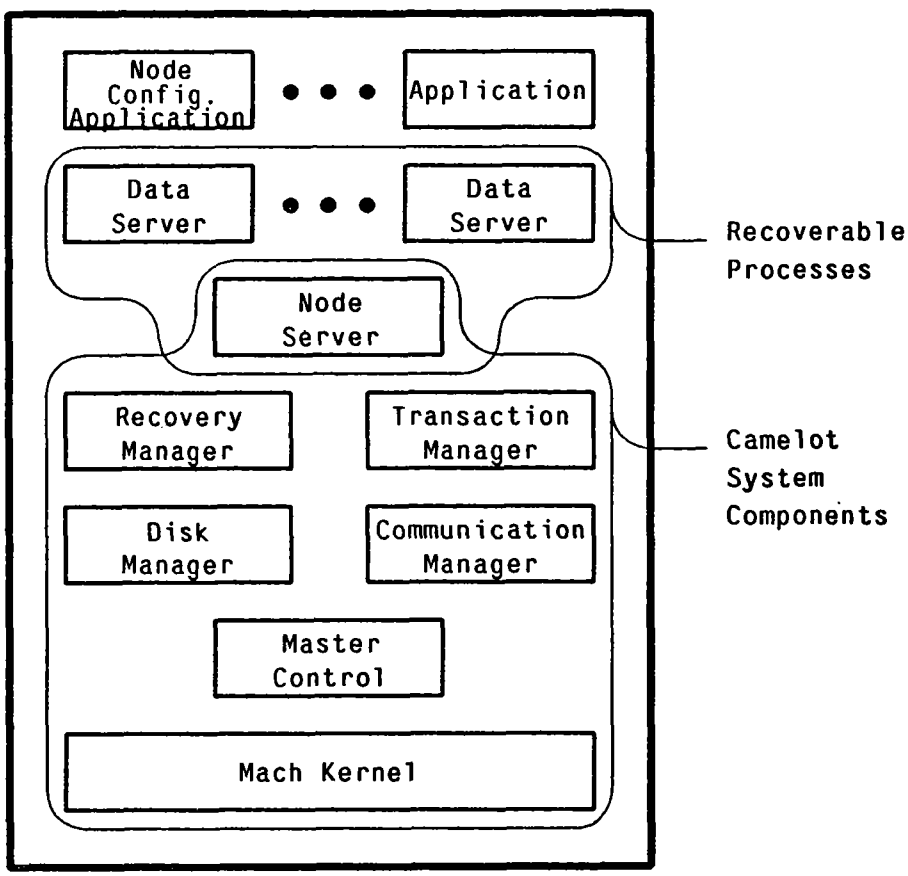

Figure 3-2: Processes in Camelot Release 1

This figure shows the Mach kernel and the processes that are needed to execute distributed transactions. The node server is both a part of Camelot, and a Camelot data server because it is the repository of essential configuration data. Other data servers and applications use the facilities of Camelot and Mach. The node configuration application permits users to exercise control over a node's configuration.

- **Recovery Manager.** The recovery manager is responsible for transaction abort, server recovery, node recovery, and media-failure recovery. Server and node recovery respectively require one and two backward passes over the log.

- **Transaction Manager.** The transaction manager coordinates the initiation, commit, and abort of local and distributed transactions. It fully supports nested transactions.

- **Node Server.** The node server is the repository of configuration data necessary for restarting the node. It stores its data in recoverable storage and is recovered before other servers.

- **Node Configuration Application.** The node configuration application permits Camelot's human users to update data in the node server and to crash and restart servers.

The organization of Camelot is similar to that of TABS and R* [Spector 85, Lindsay et al. 84]. Structurally, Camelot differs from TABS in the use of threads, shared memory interfaces, and the combination of logging and disk management in the same process. Many low-level algorithms and protocols have also been changed to improve performance and provide added functions. Camelot differs from R* in its greater use of message passing and support for common recovery facilities for servers. Of course, the functions of the two systems are quite different; R*'s transactions are intended primarily to support a particular relational database system.

4. Discussion

As of December 1986, Camelot 1 was still being coded though enough (about 20,000 lines of C) was functioning to commit and abort local transactions. Though many pieces were still missing (e.g., support for stable storage and distribution), Avalon developers could begin their implementation work. Before we begin adding to the basic set of Camelot 1 functions, we will encourage others to port abstractions to Camelot, so that we can get feedback on its functionality and performance.

Performance is a very important system goal. Experience with TABS and very preliminary performance numbers make us believe that we will be able to execute roughly 20 non-paging write transactions/second on an RT PC or MicroVax workstation. Perhaps, it is worthwhile to summarize why the Camelot/Mach combination should have performance that even database implementors will like:

- Mach's support for multiple threads of control per process permit efficient server organizations and the use of multiprocessors. Shared memory between processes permits efficient inter-process synchronization.

- Disk I/O should be efficient, because Camelot allocates recoverable storage contiguously on disk, and because Mach permits it to be mapped into a server's memory. Also, servers that know disk I/O patterns, such as database managers, can influence the page replacement algorithms by providing hints for prefetching or prewriting.

Recovery adds little overhead to normal processing because Camelot uses write-ahead logging with a common log. Though Camelot Release 1 has only value-logging, operation-logging will be provided in Release 2.

- Camelot has an efficient, datagram-based, two-phase commit protocol in addition to its non-blocking commit protocol. Even without delaying commits to reduce log forces ("group commit"), transactions require only one log force per node per transaction. Camelot requires just three datagrams per node per transaction in its star-shaped commit protocol, because final acknowledgments are piggy-backed on future communication. Camelot also has the usual optimizations for read-only transactions.

- Camelot does not implement the synchronization needed to preserve serializability. This synchronization is left to servers (and/or Avalon), which can apply semantic knowledge to provide higher concurrency or to reduce locking overhead.

Today, we would guess that Camelot's initial bottlenecks will be low-level disk code and the remaining message passing. For example, though the frequent calls by servers to Camelot are asynchronous and via shared memory, all operations on servers are invoked via message using the RPC stub generator. To further reduce message passing overhead, we might have to substitute a form of protected procedure call. This should not change Camelot very much since all inter-process communication is already expressed with procedure call syntax.

In the course of our implementation and the subsequent performance evaluation, we expect to learn much about large reliable distributed systems. Once Camelot is functioning, we plan to perform extensive experimentation on multiprocessors and distributed systems with a large number of nodes. In particular, we will measure the actual availability and performance of various replication techniques.

Our overall goal remains to demonstrate that transaction facilities can be sufficiently general and efficient to support a wide range of distributed programs. We are getting closer to achieving this goal, but much work remains.

Acknowledgments

Thanks to our colleagues Maurice Herlihy and Jeannette Wing for their advice, particularly as it relates to support for Avalon.

References

[Accetta et al. 86] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, Michael Young. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of Summer Usenix.* July, 1986.

[Allchin and McKendry 83] James E. Allchin, Martin S. McKendry. *Facilities for Supporting Atomicity in Operating Systems.* Technical Report GIT-CS-83/1, Georgia Institute of Technology, January, 1983.

[Bloch 86] Joshua J. Bloch. A Practical, Efficient Approach to Replication of Abstract Data Objects. November, 1986.Carnegie Mellon Thesis Proposal.

[Cooper 86] Eric C. Cooper. C Threads. June, 1986. Carnegie Mellon Internal Memo.

[Daniels et al. 86] Dean S. Daniels, Alfred Z. Spector, Dean Thompson. *Distributed Logging for Transaction Processing.* Technical Report CMU-CS-86-106, Carnegie-Mellon University, June, 1986.

[Eppinger and Spector 85] Jeffrey L. Eppinger, Alfred Z. Spector. *Virtual Memory Management for Recoverable Objects in the TABS Prototype.* Technical Report CMU-CS-85-163, Carnegie-Mellon University, December, 1985.

[Gray 80] James N. Gray. *A Transaction Model.* Technical Report RJ2895, IBM Research Laboratory, San Jose, California, August, 1980.

[Helland 85] Pat Helland. Transaction Monitoring Facility. *Database Engineering* 8(2):9-18, June, 1985.

[Herlihy 85] Maurice P. Herlihy. *Availability vs. atomicity: concurrency control for replicated data.* Technical Report CMU-CS-85-108, Carnegie-Mellon University, February, 1985.

[Herlihy and Wing 86] M. P. Herlihy, J. M. Wing. *Avalon: Language Support for Reliable Distributed Systems.* Technical Report CMU-CS-86-167, Carnegie Mellon University, November, 1986.

[Jones et al. 85] Michael B. Jones, Richard F. Rashid, Mary R. Thompson. Matchmaker: An Interface Specification Language for Distributed Processing. In *Proceedings of the Twelfth Annual Symposium on Principles of Programming Languages*, pages 225-235. ACM, January, 1985.

[Lamport 78] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM* 21(7):558-565, July, 1978.

[Lindsay et al. 84] Bruce G. Lindsay, Laura M. Haas, C. Mohan, Paul F. Wilms, Robert A. Yost. Computation and Communication in R*: A Distributed Database Manager. *ACM Transactions on Computer Systems* 2(1):24-38, February, 1984.

[Liskov and Scheifler 83] Barbara H. Liskov, Robert W. Scheifler. Guardians and Actions: Linguistic Support for Robust, Distributed Programs. *ACM Transactions on Programming Languages and Systems* 5(3):381-404, July, 1983.

[Postel 82] Jonathan B. Postel. Internetwork Protocol Approaches. In Paul E. Green, Jr. (editor), *Computer Network Architectures and Protocols*, chapter 18, pages 511-526.Plenum Press, 1982.

[Schwarz 84] Peter M. Schwarz. *Transactions on Typed Objects*. PhD thesis, Carnegie-Mellon University, December, 1984. Available as Technical Report CMU-CS-84-166, Carnegie-Mellon University.

[Spector 85] Alfred Z. Spector. The TABS Project. *Database Engineering* 8(2):19-25, June, 1985.

[Spector and Schwarz 83] Alfred Z. Spector, Peter M. Schwarz. Transactions: A Construct for Reliable Distributed Computing. *Operating Systems Review* 17(2):18-35, April, 1983. Also available as Technical Report CMU-CS-82-143, Carnegie-Mellon University, January 1983.

[Spector et al 86] Alfred Z. Spector, Dan Duchamp, Jeffrey L. Eppinger, Sherri G. Menees, Dean S. Thompson. The Camelot Interface Specification. September, 1986. Camelot Working Memo 2.

[Spector et al. 85] Alfred Z. Spector, Dean S. Daniels, Daniel J. Duchamp, Jeffrey L. Eppinger, Randy Pausch. Distributed Transactions for Reliable Systems. In *Proceedings of the Tenth Symposium on Operating System Principles*, pages 127-146. ACM, December, 1985. Also available in *Concurrency Control and Reliability in Distributed Systems*, Van Nostrand Reinhold Company, New York, and as Technical Report CMU-CS-85-117, Carnegie-Mellon University, September 1985.

[Thompson 86] Dean Thompson. Coding Standards for Camelot. June, 1986. Camelot Working Memo 1.

Getting the Operating System Out of the Way

J. Eliot B. Moss

Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts 01003

Abstract. In this note I outline what fundamental support database systems need from their operating environment. I argue that general purpose operating systems do not provide the necessary support, or at least not in a very good way, and that they provide a number of things that mainly get in the way of a database manager. In response, I recommend a somewhat radical approach: building the database system in a programming language on top of a minimal kernel consisting of the language run time system. This technique in essence does away with the operating system, replacing it with a programming language. While not feasible for use in systems that must support general purpose computing, this should be an effective way to build efficient, dedicated database management systems.

1 What Databases Need.

First, I will outline the basic needs of database managers, which I categorize as support for input/output, multiprogramming, and memory management. The primary need is configuration and device independence, with interfaces for the devices being simple while allowing very efficient use.

Input/Output. Database systems obviously require support for input/output. They need I/O to be efficient, and they need to be in a position to take advantage of whatever performance enhancing features a device offers. However, it is clearly useful for the database manager to be shielded from a number of device details. While the "topology" of disks (number of cylinders, tracks per cylinder, sectors per track, etc.) may be useful in some circumstances, access to fixed size blocks by sequential block number is probably adequate most of the time. The assumption made is that block numbers are related to physical position: reading sequential blocks should be fast, moving from one block to another will be faster if the blocks' numbers are close, etc. Efficient reading of a number of blocks at a time should be supported (full track read/write), and hints concerning good clustering parameters should be available (these would be derived from disk topology).

A database system needs asynchronous (nonblocking) I/O, directly to and from the memory it manages for its buffers. Low level error handling should be available, to mask most I/O errors, through retry, disk sector mapping, etc., reflecting the error to the database system only if it is unlikely that retry will help matters.

While it seems reasonable, though not trivial, to provide the necessary support for mass storage such as disks and tapes, device independent communications and terminal I/O are of equal importance, and more difficult to achieve. In network communications, database managers seem to need two levels of functionality: essentially packet level I/O,

useful for commit protocols and the like, and virtual circuits (one way at least) for bulk data transfer. Other paradigms might include remote procedure call. While the best high level model might be an issue for debate, the database manager still needs efficient, but simple to use (i.e., messy details are hidden) communications support. As with mass storage, error masking should be provided. The database manager should also be shielded as much as possible from the vagaries of addressing, packet formats, protocol standards, etc.

Communication with interactive terminals pretty much falls into the same category, as far as communcations goes, but raises the additional problems of dealing with visual formatting so that terminal/workstation features are used to best advantage without lots of special case code in the database manager. A number of packages already exist that assist in this endeavor, but there is no easy way around having to support terminals of vastly different capabilities. However, minor differences among similar devices, as well as low level details, should be masked. Database systems have an advantage over more general systems in that they typically do rather stylized interactions (forms, etc.), which, while varying in detail from terminal to terminal, share most of the top level functionality.

See the next section for further remarks concerning I/O.

Multiprogramming. Some support for multiprogramming is clearly necessary, since the sort of databases being discussed are shared. The underlying system must provide means for creating, scheduling, destroying, etc., threads of control. However, these threads are under the management of the database system. As with I/O, what is needed is a clean collection of primitives from which the database manager can fashion whatever it requires.

The appropriate kind of multiprogramming support is lightweight processes, sharing a single address space. A number of process may be managing I/O, some performing actual data manipulation and computation, and others doing housekeeping functions. Processes should be cheap. For example, it should be reasonable to fork one for each incoming query and to let it die when the response has been sent. It is unreasonable to require every logged on terminal to have an associated process, since processes require a stack, etc., which is too much space to devote to every user in a large system. Process switching should be very efficient, too, so that lightweight processes can be used as "interrupt handlers" – that is, they can block waiting I/O completion.

Input/output and other operating system features should be directly incorporated into the database manager. By this I mean that these features are only a procedure call away, not an operating system supervisor call or context switch. If lightweight processes can be used, we can revert to logically synchronous I/O: the lightweight process will block until completion, and another lightweight process will be chosen to run in the meantime.

The database manager needs to have control over the scheduling algorithm and parameters, such as time slices, priorities, and so on. It may be sufficient to offer a priority based scheduler that does round robin scheduling of processes at equal priority. One also

needs to be able to perform critical sections. This can be made independent of database concurrency control to some extent, but it is necessary to prevent preemption during critical sections. The support code should not assume that it knows about all the necessary critical sections. Protecting critical sections must be very efficient (i.e., add little overhead to locking routines that are perhaps less than 100 instructions long).

Memory Management. This breaks down into two areas: management of memory containing code, and management of data space. Management of data space should generally not be demand paged in the sense used with timesharing systems. Most global data should not be eligible for paging, and that which is should probably have priorities associated with it, according to the priorities of processes that have used or expect to use the data. That is, it should be possible to give strong hints about the importance of various pieces of data as a process executes. This will insure that data is paged in with maximum efficiency, and not paged out when it is likely to be needed again. It would be up to the system designer to insure that this scheme would not starve processes. However, high performance systems clearly depend on having enough real memory available to minimize paging.

Relatively automatic management of code, would be helpful. It could be based on hinting, when starting to execute some major piece of code, what the piece of code is and the importance of the activity that will be performed by executing it. When the process is done, it should also note that. Major pieces of code will probably not be eligible for paging, and would be managed by the just described method, which is somewhere between overlays and demand paging in its properties.

It is not clear to what extent code and data paging, as well as I/O buffer space management, should be tied together or separated. If the priority schemes are coherent, it may be best to tie all the memory management together, to gain best utilization of real memory. The point is to provide some minimal functionality, but to allow the database system substantial control over policy, through parameters and hints. It might even be reasonable to let it decide the page replacement policy, by having the page fault handler call a database system supplied subroutine for choosing replacement victims. That is harder to use in some ways, but fully general.

2 What databases do not need.

In this section I deliver my diatribe against general purpose operating systems that attempt to support database systems, too. The main points of the argument are that in trying to support database a general purpose operating system provides features that are too complex, never quite right, rather system specific, and expensive to use. Hence, many database system developers spend much of their time trying to figure out how to game the operating system just right, or how to get it out of their way.

Multiprogramming. In this area operating systems typically provide large, expensive processes, primarily because they are trying to protect the system, as well as other processes, from processes that go haywire, or try to access protected data, etc. If we are prepared to dedicate the computer to the database application, then the protection issue goes away. The database system will almost certainly be of high privilege in a time sharing environment and will itself control user access to the database in ways more sophisticated than the operating system can, so there is not much point in protecting the database from itself. There is an increased risk that bugs in the database system itself will be a problem, but high performance database systems use shared memory already, so I do not think this can be considered a major problem. Getting rid of the protection (and hence context switching) boundaries greatly simplifies things, as well as substantially improving performance. For example, getting at locks, buffers, internal tables, etc., can be done directly rather than having to call across an interface. We can do things in microseconds rather than milliseconds. Using a single address space with no per-process protection makes lightweight processes more practical as well. In sum, to support databases, the operating system should not provide elaborate protection and security features – get the fences out of the way for the database system.

Memory management. Straight demand paged virtual memory does not help a database system very much: it needs control over when data is written to disk, so as to implement recovery protocols. Further, page replacement algorithms suitable for general use are not really appropriate, because the database system knows a lot about how data and code are used. Much better performance can be had by letting the database system have substantial control in this area.

Input/output. Database systems may not be able to take advantage of elaborate access methods that come with an operating system. The problem is they probably will not quite match what the database system wants to do. Besides, the database system will provide more variety and sophistication in this area, so it is probably not worth the effort to provide half hearted measures.

Concurrency control. If locking is an operating system call away, the database system will not use it. The point here is that good locking code is perhaps ten to one thousand instructions long; I have heard that on typical superminicomputers 100 microseconds is an average lock acquisition time (for an available lock). An operating system probably cannot even dispatch the supervisor call that fast, and its data structure probably will not be quite right either. This relates back to multiprogramming: the operating system should provide the basic mechanisms, simple and cheap, and let the database system worry about policy.

Transaction features. While there has been considerable interest of late in the use of transactions to help build operating systems, it would probably be a mistake to build a single transaction mechanism to serve them both. Transactions might indeed help

structure a reliable, distributed operating system, just as they are a powerful and useful concept for databases, but the objects being manipulated are probably rather different. As with concurrency control, the operating system will almost certainly not provide exactly the right thing. Worse, having provided a mechanism, the operating system may then preempt the ability to do something different, leaving the database system designer up the creek.

In sum, operating systems designers should not second guess database system designers, nor preempt their ability to build more sophisticated capabilities.

3 The Language Based Approach.

Being something of a language designer and implementer by trade, I naturally favor designing and then using an appropriate language for building database systems of the kind I have been describing. In fact, one way to achieve some measure of portability is to design the required primitives into the language. To some extent all I have done is turn an operating system kernel implementation problem into a language run time system implementation problem. However, the language approach has the additional power of being able to shove various things onto a compiler and sometimes take care of them at compile time. A typical example of this is checking various properties of arguments in "system calls" (which turn into procedure calls or invocations of builtin primitives).

The sort of language I am describing should not be taken as being at as high a level as a database query language. Rather, we are talking about a systems implementation language. To a certain extent, Ada, Modula, and Mesa have the right flavor. They all provide control over the bits, have lightweight processes, and offer some inter-process synchronization and communication primitives. Since Mesa has been used successfully to build the operating system Cedar, it might even be a reasonable candidate.

Let us consider more specifically some attributes desirable in a database implementation language. Foremost, I would say, is a good facility for data abstraction and code modularization. Data abstraction is one of the best ways to hide hardware dependencies. For example, a data abstraction for page allocation, deallocation, and access can take care of mapping down to disk cylinders and tracks, optimizing placement of related data (when given hints from invokers of the abstraction), etc. It could also help insure that data are written back to the right place, by identifying pages, and, in conjunction with a memory buffer abstraction, associating database pages with main memory buffers.

However, it would appear that one probably needs ways to escape from type checking and similar rules, too. In particular, memory management will require the ability to manipulate addresses, and when data are read from disk, they must be coerced (e.g., via the type cast construct of the C language) into types known to the program. This is an issue that has been raised, and dealt with in various ways, in programming language circles,

but may be less familiar to database cognoscenti. A more ambitious approach would be to extend the language to include persistence, i.e., directly understand and manipulate objects on disk automatically. This is an exciting area of current research, but beyond the scope of the present discussion. While pointer manipulation and memory management require the use of unsafe features, the unsafe code can be quite localized, reducing the likelihood of error. The language should encourage such localization; C, for example, is weak in this respect.

Concerning processes and synchronization, I think the features of Mesa and Modula are better suited than those of Ada. In particular, using locks seems more natural than using rendezvous when expressing mutual exclusion in access to internal tables. In any case, interrupts should turn into synchronization signals (e.g., a V on a semaphore, etc.) to suspended processes. Thus a process might write a buffer out to disk by executing the code, including the device driver, itself, blocking for the completion signal, and then continuing. Components of the system which act as schedulers should be able to access waiting processes as entities and manipulate them easily (e.g., to move them from queue to queue). It is helpful if a process can lock a data structure, file itself in the data structure, and then atomically: block, unlock the data structure, and signal a scheduler. The scheduler can then lock the data structure and get at the suspended process. Similar arguments suggest that individual messages (or streams) be data objects, with no strong, fixed relationship to processes, allowing communications to be handled in a flexible, priority fashion.

Control over paging of code and data is not an issue of the language *per se*. Rather, one aspect is the *ability* to lock and release pages in memory, etc., which can be done via a page data abstraction. The other aspect is describing pieces of code/data, grouping them, etc. This is more an issue in the construction of tools, such as the compiler and linker, and their connection back to the language elements (procedures, variables, etc.). For example, one might group code into segments. It would then be necessary to determine the segment containing a given routine, then the location and size of the segment, in order to lock the routine and its associates into main memory all at once. A complementary approach is to deliver page faults back to the database system. This requires being able to specify some pages that must absolutely never be paged out, etc.

In sum, we can identify from our previous discussion some features desired of a database implementation language. But the features mentioned probably do not form a complete or exactly correct list.

4 Advantages of the Language Approach

The advantages of the language approach are several. First, it should offer greater portability across machines. We get the usual hiding of hardware differences associated with a

high level language. But in addition, rather than adapting to different operating systems, we adapt the machine to us. Thus we push the differences down to low level abstractions, and avoid unnecessary variations (e.g., among different operating systems on the same hardware).

We also gain the many advantages of modern programming languages in support of system development and evolution. In particular, data abstraction and type checking can be offered. At present one must choose between an efficient, unsupportive language (e.g., assembly or C) or a less efficient, less convenient, more supportive language (Ada might be an example). In designing a new language we can likely offer both.

Even better, we gain leverage by providing the *right* features in the language, making it easier to express what the database implementer wants to do. This will ease implementation and maintenance, as well as reduce errors arising from a poor match of language to programming task.

Finally, and most importantly, the language approach offers improved performance by getting the operating system out of the way. We avoid unnecessary process context switches, unnecessary protection context switches, and unnecessary argument checks. We also shed entire mechanisms (protection, access control) that add overhead and must be redone anyway. A more subtle effect is that produced by using less general and more appropriate algorithms. Among these are policies (e.g., for paging or device I/O scheduling) that are more general or even counterproductive compared to database needs. Carried to its extreme, this argument would suggest eliminating unnecessary features from the hardware architecture, and thus increasing speed, reducing cost, or both.

In sum, the main advantages of the language approach include: portability (just implement the run time system on your target architecture – not trivial, but there should be a pretty clear specification of what is required); efficiency (no operating system call overhead, and no protection domains); support for modern software engineering practices (many languages used in implementing databases do not have adequate safeguards, or features for building large systems); and leverage (the appropriate set of features makes programming easier).

5 Objections to My Proposal.

The worst objection is that the approach described is applicable mainly to dedicated database systems, as opposed to general purpose time sharing systems, which might also wish to support database access for their users. If the correct set of features can be worked out, it may be possible to graft a database system of the kind I envision onto a general purpose operating system, provided the operating system offers the necessary hooks, However, there will almost certainly be a significant performance penalty compared with the dedicated system.

A second objection is portability: to port the database system, one must in essence build a new operating system for the target hardware. However, this may not be too bad. First, we are not talking about a full blown time sharing system, with utilities, and so forth. We are talking about an operating system kernel only, and one providing a standard interface. It is conceivable that considerable parts could be written in a reasonable language which is fairly portable and available, such as C, Modula, or Ada. This would cut the porting time down substantially. Having the correct specification and design really will help.

Finally, it is clear that I am describing what must for the time being be considered research or advanced development, by no means a proven concept. If your measure is "Will it sell in the marketplace?", then I have some doubts, at least until I have experience building one of these systems. I am engaged in research to define a language, implement it, and build a distributed object oriented database with it. However, the language will itself be object oriented, and, because the emphasis is a little more on functionality and capability, for the time being we will probably interpret it rather than compile it, and we will not strain for performance at the moment. (Get it right, then make it fast.)

This note owes a lot to discussions I have had with a number of people, and presentations I have seen over the years, as well as bits written down here and there. It expresses opinion and religion more than experimentally demonstrated fact, but I think that a number of people agree with substantial parts of my criticisms and suggestions concerning operating systems. The language part is more radical. Anyway, the words are mine, but the ideas can frequently be attributed to others. However, I will not mention individuals, since it is hard to say to what extent they might agree with what I have said.

OPERATING SYSTEM SUPPORT FOR DATA MANAGEMENT

Michael Stonebraker and Akhil Kumar

EECS Department
University of California
Berkeley, Ca., 94720

Abstract In this paper we discuss four concepts that have been proposed over the last few years as operating system services. These are:

1) transaction management
2) network file systems and location independence
3) remote procedure calls
4) lightweight processes

From the point of view of implementors of a data manager, we indicate our reaction to these services. We conclude that the first service will probably go unused while the second is probably downright harmful. On the other hand, the absence of the last two services in most conventional operating systems requires a data base system implementor to write substantial extra code.

1. INTRODUCTION

An operating system is nominally designed to provide services to client applications, including large scale data managers. In [STON81] one of us commented on the utility of current operating system constructs as services for a data manager. Since that time, there have been numerous proposals for new operating system services, and the purpose of this paper is to comment on several of these from the point of view of perceived utility to the implementor of a data base system.

We focus our attention on four particular services. First, many researchers propose next-generation operating systems with support for transactions. Older proposals include [MITC82, BROW81] while more recent ones can be found in [CHAN86, MUEL83, PU86, SPEC83]. Comments on the viability of operating system transaction managers can be found in [STON84, STON85, TRAI82]. In Section 2 of this paper we present our current thinking on this topic. Then in Section 3 we turn to a discussion of the network file systems that exist in many current operating systems (for example Sun UNIX and Apollo DOMAIN). Current data

This research was sponsored by the National Science Foundation under Grant DMC-8504633 and by the Navy Electronics Systems Command under contract N00039-84-C-0039.

managers which run in this environment (e.g. INGRES and ORACLE) usually do not use this service and we indicate the reasons for this state of affairs. Section 3 then continues to the more general notion of location transparency, such as provided by LOCUS [WALK83] and Clouds [DASG85]. We indicate the reasons why a data manager would be obliged to turn this service off to obtain reasonable performance. Section 4 then discusses remote procedure call mechanisms and the use which a data manager would find for this construct. Although one can code around their absence in current systems, we conclude they would be a most beneficial addition. Lastly, we turn in Section 5 to the need and intended use for lightweight processes. We indicate why they are crucial to data managers and what one would want them to do. The tax for their non-inclusion in most current systems is quite high, and we would encourage more operating system designers to think deeply about providing this service.

2. OPERATING SYSTEM TRANSACTION MANAGEMENT

A **transaction** is a collection of data manipulation commands that must be atomic and serializable [GRAY78]. Current data managers implement the transaction concept by means of sophisticated user space crash recovery and concurrency control software. This software is considered tedious to write, hard to debug, and its reliability is absolutely crucial. All data base implementors that we have met would appreciate somebody else being responsible for this function. In particular, abdicating these functions to the operating system seems the only realistic possibility.

There are two possible ways this abdication could take place, namely:

total abdication
partial abdication

In total abdication, the data manager would accept a ``begin transaction´´ command from a user and pass it directly to the operating system. ``Abort transaction´´ and ``commit transaction´´ instructions would be similarly redirected. Hence, the data manager would be uninvolved in (and unconcerned with) the details of transaction support.

In partial abdication the data manager would still be required to define the ``events´´ which make up the contents of a current data base log as well as provide the required ``undo´´ and ``redo´´ routines. Any special case locking of indexes, system catalogs and other special data structure would also require direct data manager implementation. Partial abdication would move the basic control flow surrounding log processing into the operating system as well as the innards of the lock manager.

In our opinion, partial abdication would simplify data base transaction code only marginally since the easiest part of the code is all that is removed. Defining events and implementing them recoverably is still left to the data manager. This is where a majority of the time and effort in current transaction systems is spent. Hence we are **MUCH** more excited by the prospect of total abdication. A recent IBM Research [CHAN86] proposal for moving transactions into the OS is the most promising one we have seen because it suggests hardware supported locking on small-granularity 128 byte objects and a write-ahead log on the same small granules. Hence, we are investigating the viability of this proposal by performing

an extensive simulation study.

We have constructed simulation models to compare the performance of an OS transaction manager with that of its database counterpart. The models must capture the relative strengths and weaknesses of each approach. The trade-offs involved in the two alternatives have to be highlighted in the models by means of parameters that correspond to performance factors in each situation. For example, our models have to reflect the fact that the OS can set locks at a lower cost than the data manager. This factor is likely to make the OS solution relatively more attractive. On the other hand, the OS transaction manager lacks semantic information about the database. For instance, it cannot distinguish between data objects and indexes while the data manager can. This ability to distinguish between different types of objects enables the data manager to apply short-term locks on hot spots like indexes and system catalogs and, therefore, allow greater concurrent activity. It also means that the data manager need only log updates to data objects since the index may be reconstructed from the data at recovery time. Therefore, the data manager can implement crash recovery mechanisms in a smarter way and at a lower cost. This increased level of semantics tends to make the data manager solution preferable.

Simulation models have previously been used in this area to study the performance of concurrency control algorithms in a conventional data manager (e.g., [RIES79], [CARE84], [AGRA85]). We have expanded these models in several ways. First, we are modelling both concurrency control and crash recovery. Second, we have a more elaborate data base model for treating data and index objects separately. Thirdly, our model includes buffer management which has been omitted in the previous studies where it is assumed that every data object access involves a disk I/O. As a side result from the study we will be able to examine the effect, if any, of buffer size on each alternative. Lasltly, our primary motivation is to study where transactions are supported and not to investigate a range of different user-space algorithms for transaction management.

Our initial studies indicate that an operating system transaction manager provides about 30 percent lower transaction throughput than its data base counterpart under a variety of simulation parameters. This performance loss is primarily the result of a log that is vastly larger than the one used by the data manager, the presence of more deadlocks, and the larger number of locks set by the operating system transaction manager. Moreover, recovery time is about twice as long for the OS transaction manager largely because of the necessity of reading a much larger log.

Hence, our initial experiments suggest that total abdication results in an unacceptable loss of performance. Unless there is some way for the operating system to speed up transaction management, perhaps by providing special hardware appropriate to the task, we are skeptical that the operating system transaction manager can provide the performance required by current data managers. A full paper on this study is in preparation [KUMA86].

3. NETWORK FILE SYSTEMS AND LOCATION TRANSPARENCY

Popular Network File Systems (NFS) allow a user to access a remote file in the same way as he accesses a local file. This level of transparency would seem to

be a good idea; however, it is unused by data managers in the environments in which it runs. According to [LAZO86], the cost of a remote read is about 2-3 times the cost of a local read. Although these measurements were particular to one hardware and operating system environment, they seem to reflect the performance of most current implementations. Consequently, if a data manager is accessing the EMP relation to find all employees who live in Boston and are over 50 or alternatively live in New York and are over 55, then it can read the entire EMP file over an NFS to discover the employees who actually qualify. Alternately, it can run the data manager on the remote node where it will read the entire file as a collection of local reads. Then, only the qualifying records are actually moved between sites. Obviously, if there are few qualifying employees, the latter tactic will be much more efficient. Hence, the common wisdom is ``move the query to the data and not the data to the query´´ Both ORACLE and INGRES operate in this more efficient mode in environments supporting NFS. Hence, a data manager is willing to go to the trouble of executing a remote process and setting up the inter-machine communication to obtain a perceived substantial performance gain.

Beyond NFS is the possible inclusion of location transparency for a file system. In this context, a user need never worry about where a file is, he simply accesses it as if it were local to his site. This generalizes NFS by not requiring a user to ever know where a file actually resides. Moreover, the operating system could decide to move the file if disk traffic or accessing patterns warranted a switch.

Unfortunately, a data manager would insist on turning this service off. If the data manager accesses files as if they were local, then it would simply run the normal one-site query optimizer built into existing data managers. Such a one-site optimizer can decide among various query processing options for complex queries including merge-sort and iterative substitution. However, optimizers that work in a distributed environment are much more complex. They attempt to minimize a different cost function that includes the cost of network communication and utilize additional strategies including semi-joins. Moreover, they worry extensively about the location of intermediate results produced during the query execution process. Often it is a good idea to move two objects to a third site and perform a join at that site.

If a data manager cannot find out the location of objects, then all this optimization cannot be used. Moreover, if the data manager can find out the current location but later the location might be changed by the underlying operating system, then the optimization can be performed but may be disasterously sub-optimal if locations subsequently change. Our intuitive sense is that a system offering location transparency may well be several orders of magnitude slower than a conventional distributed data base system. In this case, a data manager would never utilize a file system offering location transparency.

4. REMOTE PROCEDURE CALL

As noted in the previous section, the application program often must run at a different site than the data base engine. In addition, a distributed data base system is architected by having a local manager which runs at each site containing data relevant to a user command and then a global manager which coordinated the

local managers. Hence, to solve a distributed query involving data at N sites, there will be N local managers and one global manager who must communicate. Lastly, there is considerable recent interest in extendible data base systems, which can be tailored to individual needs by including user-written procedures. See [CARE86, STON86] for two different approaches to such a system. In an extendible environment, one must call procedures written by others, and a robust debugging environment seems essential. Hence, one would like to link the user written procedure into the data manager for high performance and also to execute a call to a protected address space for program isolation during debugging. This last feature requires communication to a remote process running on the same machine.

In all these cases, the run-time system must linearize the parameters of a procedure call, then pass them though the available interprocess communication system, and then repackage the data structure at the other end. All this effort would go away if a remote procedure call (RPC) mechanism were available. Hence, we are enthusiastic about the RPC facilities.

5. LIGHTWEIGHT PROCESSES

A conventional operating system binds each terminal to an application program. Hence, if there are N active terminals, then N application processes will exist. Moreover, the obvious way to architect a data manager is to fork a data base process for each data base user. This will lead to N data base processes and a total of 2 * N active processes. The N data base processes can all share a single copy of the code segment and have private data segments. Moreover, if the operating system provides shared data segments, then the lock table and buffer pool can be placed there.

An alternative approach is to construct a data base server process. Here, there would be only one data base process to which all N application processes would send messages requesting data base services. A server process entails writing a complete multi-tasking special purpose operating system inside the data manager. This duplication of operating system services seems intellectually very undesirable.

As noted in [STON81] a data manager must overcome significant problems when using either architectures. In this section we indicate one additional problem that occurs in a process-per-user architecture. When run in commercial environments on large machines, it is not unusual for there to be several hundred terminals connected to a system. Five hundred is not a gigantic number, and the largest terminal collection we know of exceeds 40,000. Assuming 500 terminals each supporting a data base user and assuming a process-per-user model of computation, the OS must support 1000 active processes. Current operating systems tend to generate excessive overhead when presented with large numbers of propcesses to manage. For example, the common tactic of having the scheduler do a sequential scan of the process table to decide who to run next is not workable on large systems.

In addition, memory management with a large number of processes tends to be a problem because the memory per terminal is often not very large. For example, a 32 mbyte system with 500 terminals yields only 60 kbytes per terminal. Consequently, great care must be taken to avoid thrashing. Lastly, excessive

memory consumption is likely because the implementor of a process-per-user data manager will likely allocate static areas for such things as parse trees and query plans. Consequently, the data segment of a process-per-user data manager tends to be of substantial size. In a server implementation, all such allocation would likely be done dynamically. Hence, in a 500 user environment, the server data segment would be much smaller than 500 times the size of a process-per-user data segment. Consequently, the total amount of main memory consumed by the data manager is higher in a process-per-user environment, and the probability of thrashing is increased.

A solution to the resource consumption and bad performance of a process-per-user model is for an operating system to support lightweight processes. This model of computation would allocate one address space for a collection of tasks. Multiple threads of control would be active in the single address space and thereby be able to share open files and data structures. One would like an operating system to schedule the address space and then as a second decision schedule the particular thread in the address space. The data manager would like easy-to-use capabilities to add and delete threads from an address space. Such operations should be much cheaper than creating or destroying a process.

More generally, the concept of binding an address space to every terminal is also questionable since in a data base environment, many of the users will be executing the same application program. Hence, lightweight processes make sense for application software also. To utilize this construct, the hard binding of a terminal to a process must disappear. The general idea then would be to create N1 application servers and N2 data base servers each supporting multiple threads of control. All the ``plumbing´´ to connect this environment together should be done within the operating system. Moreover, one should be able to add and drop servers easily and have their load be automatically redirected to other servers. CICS in an IBM environment and Pathway in a Tandem environment are system with this flavor. Hopefully, some of these capabilities will migrate into future general purpose operating systems.

REFERENCES

[AGRA85]
Agrawal, R., et. al., "Models for Studying Concurrency Control Performance : Alternatives and Implications," Proc. 1985 ACM-SIGMOD Conference on Management of Data, June 1981.

[BROW81]
Brown, M. et. al., "The Cedar Database Management System," Proc. 1981 ACM-SIGMOD Conference on Management of Data, Ann Arbor, Mich., June 1981.

[CARE84]
Carey, M. and Stonebraker, M., "The Performance of Concurrency Control Algorithms for Database Management Systems," Proc. 1984 VLDB Conference, Singapore, Sept. 1984.

[CARE86]
Carey, M., "The Preliminary Design of Exodus," Proc. 11th Very Large Data Base Conference, Kyoto, Japan, Sept. 1986.

[CHAN86]
Chang, A., (private communication)

[DASG85]
Dasgupta, P. et. al., "The Clouds Project: Design and Implementation of a Fault Tolerant Distributed Operating System," Technical report GIT-ICS 85/29, Georgia Tech., Oct. 1985.

[GRAY78]
Gray, J., "Notes on Data Base Operating Systems," in Operating Systems: An Advanced Course, Springer-Verlag, 1978, pp393-481.

[KUMA86]
Kumar, A., "Performance of Alternate Approaches to Transaction Processing," (in preparation).

[LAZO86]
Lazowska, E. et. al., "File Access Performance of Diskless Workstations," ACM-TOCS, August 1986.

[MITC82]
Mitchell, J. and Dion, J., "A Comparison of Two Network-Based File Servers," CACM, April 1982.

[MUEL83]
Mueller, E. et. al., "A Nested Transaction Mechanism for LOCUS," Proc. 9th Symposium on Operating System Principles, October 1983.

[PU86]
Pu, C. and Noe, J., "Design of Nested Transactions in Eden," Technical Report 85-12-03, Dept. of Computer Science, Univ. of Washington, Seattle, Wash., Feb. 1986.

[RIES79]
Ries, D., "The Effects of Concurrency Control on Data Base Management System Performance," Electronics Research Laboratory, Univ. of California, Memo ERL M79/20, April 1979.

[SPEC83]
Spector, A. and Schwartz, P., "Transactions: A Construct for Reliable Distributed Computing," Operating Systems Review, Vol 17, No 2, April 1983.

[STON81]
Stonebraker, M., "Operating System Support for Data Managers", CACM, April 1981.

[STON84]
Stonebraker, M., "Virtual Memory Transaction Management," Operating System Review, April 1984.

[STON85]
Stonebraker, M., et. al., "Problems in Supporting Data Base Transactions in an Operating System Transaction Manager," Operating System Review,

January, 1985.

[STON86]
Stonebraker, M., Inclusion of New Types in a Data Base System," Proc. 1986 IEEE Data Engineering Conference, Los Angeles, ca., Feb. 1986.

[TRAI82]
Traiger, I., "Virtual Memory Management for Data Base Systems," Operating Systems Review, Vol 16, No 4, October 1982.

[WALK83]
Walker, B. et. al., "The LOCUS Distributed Operating System," Proc. 9th Symposium on Operating System Principles," Oct 1983.

IEEE COMPUTER SOCIETY

TECHNICAL COMMITTEE APPLICATION

IEEE COMPUTER SOCIETY

INSTRUCTIONS: PLEASE PRINT IN INK OR TYPE (ONE CHARACTER PER BOX. INFORMATION WRITTEN OUTSIDE OF BOXES WILL NOT BE RECORDED) BECAUSE OF PACKAGE DELIVERY SERVICES, STREET ADDRESSES ARE PREFERRED RATHER THAN, OR IN ADDITION TO, POST OFFICE BOX NUMBERS. INTERNATIONAL MEMBERS ARE REQUESTED TO MAKE BEST USE OF AVAILABLE SPACE FOR LONG ADDRESSES.

LAST NAME | FIRST NAME | INITIAL | DR/MR/MRS/MS/MISS/PROF, ETC

COMPANY, UNIVERSITY, AGENCY NAME | DEPARTMENT/MAIL STOP/BUILDING/P.O. BOX/APARTMENT/ETC

STREET ADDRESS (OR POST OFFICE BOX) | DATE: MONTH, DAY, YEAR

CHECK ONE
- ☐ NEW APPLICATION
- ☐ INFORMATION UPDATE

CITY | STATE | ZIP CODE

COUNTRY | OFFICE PHONE | HOME PHONE (Optional)

ELECTRONIC MAIL NETWORK | ELECTRONIC MAIL ADDRESS (Mailbox)

TELEX NUMBER (Optional) | IEEE MEMBER/AFFILIATE NO | I am a Computer Society Member ☐ YES ☐ NO

DEFINITIONS:

TC MEMBER—Actively participates in TC activities, receives newsletters and all TC communications. Active participation means that you do or are willing to do something for the TC such as review papers, help organize workshops, conferences, etc., participate in standards development, help with TC operations, etc. Person can be a member in up to 4 TCs.

TC CORRESPONDENT—Does not participate actively in TC activities; receives newsletters and other TC communications. Person can be a correspondent in up to 4 more TCs.

PLEASE INDICATE YOUR TC ACTIVITY INTERESTS

5 4 3 2 1

HIGH INTEREST — NO INTEREST

Technical Interests/Specialties:_____

TECHNICAL COMMITTEE CODES: MEMBER M CORRESPONDENT C	TC Codes
☐ Computational Medicine (01)	01
☐ Computer Architecture (02)	02
☐ Computer Communications (03)	03
☐ Computer Elements (04)	04
☐ Computer Graphics (05)	05
☐ Computer Languages (06)	06
☐ Computer Packaging (07)	07
☐ Computers in Education (08)	08
☐ Computing and the Handicapped (09)	09
☐ Data Base Engineering (10)	10
☐ Design Automation (11)	11
☐ Distributed Processing (12)	12
☐ Fault-Tolerant Computing (13)	13
☐ Mass Storage Systems & Technology (14)	14
☐ Mathematical Foundations of Computing (15)	15
☐ Microprocessors & Microcomputers (16)	16
☐ Microprogramming (17)	17
☐ Multiple-Valued Logic (18)	18
☐ Oceanic Engineering & Technology (19)	19
☐ Office Automation (20)	20
☐ Operating Systems (21)	21
☐ Optical Processing (22)	22
☐ Pattern Analysis & Machine Intelligence (23)	23
☐ Personal Computing (24)	24
☐ Real Time Systems (25)	25
☐ Robotics (26)	26
☐ Security and Privacy (27)	27
☐ Simulation (28)	28
☐ Software Engineering (29)	29
☐ Test Technology (30)	30
☐ VLSI (31)	31
☐ Computer and Display Ergonomics (32)	32
☐ Supercomputing Applications (33)	33

Conference Organizing Committee	Workshop Organizing Committee	Conference/Workshop Session Chairman	Help with Newsletter	Write Survey/Technical Papers	Review Papers	Standards Development	Educational Activities	Help with TC Operations	Other *(Specify)*:

please return this form to the following address: **IEEE COMPUTER SOCIETY**
1730 Massachusetts Ave NW
Washington DC 20036-1903

We look forward to having you with us!

CALL FOR PARTICIPATION

IFIP WG8.4 Workshop on

Office Knowledge: Representation, Management and Utilization

17-19 August 1987
University of Toronto
Toronto, Canada

WORKSHOP CHAIRMAN	PROGRAM CHAIRMAN	ORGANIZING CHAIRMAN
Prof. Dr. Alex A. Verrijn-Stuart	Dr. Winfried Lamersdorf	Prof. Fred H. Lochovsky
University of Leiden	IBM European Networking Center	University of Toronto

This workshop is intended as a forum and focus for research in the representation, management and utilization of knowledge in the office. This research area draws from and extends techniques in the areas of artificial intelligence, data base management systems, programming languages, and communication systems. The workshop program will consist of one day of invited presentations from key researchers in the area plus one and one half days of contributed presentations. Extended abstracts, in English, of 4-8 *double-spaced* pages (1,000-2,000 words) are invited. Each submission will be screened for relevance and potential to stimulate discussion. There will be no formal workshop proceedings. However, accepted submissions will appear *as submitted* in a special issue of the WG8.4 newsletter and will be made available to workshop participants.

How to submit

Four copies of *double-spaced* extended abstracts in English of 1,000-2,000 words (4-8 pages) should be submitted by **15 April 1987** to the Program Chairman:

Dr. Winfried Lamersdorf
IBM European Networking Center
Tiergartenstrasse 15
Postfach 10 30 68
D-6900 Heidelberg
West Germany

Important Dates

Extended abstracts due:	**15 April 1987**
Notification of acceptance for presentation:	**1 June 1987**
Workshop:	**17-19 August 1987**

CALL FOR PAPERS

13th International Conference
on
Very Large Data Bases

Brighton, England, U.K.
1—4 September 1987

THE CONFERENCE

VLDB Conferences are a forum and focus for identifying and encouraging research, development, and the novel applications of database management systems and techniques. The Thirteenth VLDB Conference will bring together researchers and practitioners to exchange ideas and advance the subject. Papers of up to 5000 words in length and of high quality are invited on any aspect of the subject but particularly on the topics listed below. All submitted papers will be read and carefully evaluated by the Programme Committee.

TOPICS

Major topics of interest include, but are not limited to:

Data Models
Design Methods and Tools
Distributed Databases
Query Optimization
Concurrency Control
Database Machines
Performance Issues
Security
Knowledge Base Representation
Multi-media Databases
Implementation Techniques
Object Oriented Models
The role of logics

TO SUBMIT YOUR PAPERS

Five copies of double-spaced manuscript in English up to 5000 words should be submitted by 6 February 1987 to either of the following:

Mr William Kent
Hewlett-Packard
Computer Research Centre
1501 Page Mill Road
Palo Alto CA 94394 USA

Professor P M Stocker
Computer Centre
University of East Anglia
Norwich NR4 7TJ

IMPORTANT DATES

PAPERS DUE:	**6 FEBRUARY, 1987**
NOTIFICATION OF ACCEPTANCE:	**27 APRIL, 1987**
CAMERA READY COPIES DUE:	**29 MAY, 1987**

General Conference Chairman
Professor Peter J H King
Birkbeck College
University of London
England

Programme Co-ordinator
Professor G Bracchi
Politecnico di Milano
Italy

Programme Committee Co-Chairmen
Professor P M Stocker
University of East Anglia, England
Mr William Kent
Hewlett Packard
U.S.A.

Organising Committee Chairman
Dr Keith G Jeffery
Science & Engineering Research Council, England
North America Organising Co-ordinator
Professor Stuart E Madnick
Massachusetts Institute of Technology
U.S.A.

Preliminary Call for Papers

ER APPROACH

6th International Conference on Entity-Relationship Approach

November 9-11, 1987 — New York

Conference Chairmen:	Martin Modell, Merrill Lynch
	Stefano Spaccapietra, Université de Bourgogne
Program Committee Chairman:	Salvatore T. March, University of Minnesota
Steering Committee Chairman:	Peter P. Chen, LSU and MIT

The Entity-Relationship Approach is the basis for many Database Design and System Development Methodologies. ER Conferences bring together practitioners and researchers to share new developments and issues related to the use of the ER Approach. The conference consists of presented papers addressing the theory and practice of the ER Approach as well as invited papers, tutorials, panel sessions, and demonstrations.

Major Themes:

Database Development and Management

* database design
* database management systems
* languages for data description and manipulation
* data models and modelling
* database dynamics
* database constraints

Application Systems

* CAD/CAM and engineering databases
* knowledge-based systems
* object-oriented systems
* multi-media databases
* user interfaces
* business systems

Managing Organizational Information Resources

* data planning
* information architectures
* translating data plans into application systems
* data dictionaries
* information centers
* end user computing

Submission of Papers:

Papers are solicited on the above topics or on any other topic relevant to the ER Approach. Five copies of completed papers must be received by the program committee chairman by April 15, 1987. Papers must be written in English and may not exceed 25 double spaced pages. Selected papers will be published in *Data and Knowledge Engineering* (North-Holland).

Important Dates:

Papers due:	April 15, 1987
Notification of acceptance:	June 20, 1987
Camera ready copies due:	August 15, 1987

All requests and submissions related to the program should be sent to the program committee chairman:

Professor Salvatore T. March
Department of Management Sciences
University of Minnesota
271 19th Avenue South
Minneapolis, MN 55455 USA

Tel: (612) 624-2017
bitnet: march@umnacvx
csnet: march@umn-cs

THE COMPUTER SOCIETY
OF THE IEEE
1730 Massachusetts Avenue, N W
Washington, DC 20036-1903

Prof Philip A.Bernstein
Wang Institute
72 Tyng Road
Tyngsboro, MA 01879
USA