MINIX for a Distributed Database System

M D Meumann and M H Rennhackkamp
Department of Computer Science, University of Stellenbosch, Stellenbosch 7600

Abstract

A user of a distributed database management system must be able to access data which is stored on a number of different sites, connected by a network, without being aware of the physical data distribution. The NRDNIX distributed database management system consists of four major components, namely the presentation manager, communication kernel, database manager and network manager.

The development of a distributed system requires the addition of communication capabilities to the supporting operating system. The MINIX operating system could be considered as a possible implementation environment. MINIX is based on the client-server message passing model. Architecturally it consists of a user layer, a server layer and two kernel layers. A distributed database implementation using MINIX can be configured as a database server process, an extension to the existing file system, a spesialized database file system, a network operating system process or a distributed operating system process.

MINIX has a number of drawbacks, namely deadlock, alarm signals, deviations from the message passing model and memory limitations. The severity of the fundamentals design flaws in MINIX render it unusable for the implementation of a distributed database management system.

Keywords: *Distributed database management system, Operating system, MINIX.*
Computing Review Categories: *D.4.1, D.4.3, D.4.4, H.2.4.1*

Received September 1989, Accepted May 1990

1. Introduction

The goal of the NRDNIX system under development is the implementation of a distributed database management system. A distributed database is a logically interconnected system of databases, which are physically dispersed on geographically remote, but interconnected computing facilities. A user must be able to access data which is stored on a number of different sites, connected by a network, without being aware of the data dispersion or the distributed access. An aim of the NRDNIX system is to run on small, relatively powerful micro-computer systems, in order to combine the capabilities of a number of machines to overcome the limitations of a single machine.

Traditionally, database systems were implemented and run as user processes. These make use of facilities offered by the supporting operating systems, including the file system. Seeing they run at user process priority, they have to function through an additional layer of software. The database system also has to compete with other user processes and the operating system processes for processing resources, usually controlled by an unintelligent scheduler. This could account for the poor response times usually experienced by users of such database systems. A more efficient approach would be to modify the operating system to include database features. This would result in the processes performing database requests being scheduled at the same priority as operating system processes and not at the lower priority of the user processes. Thus, the requirements for the implementation include an operating system which allows a number of processes to execute concurrently.

The implementation of a distributed system also requires the addition of communication features as part of the operating system. The operating system should offer communication facilities through an interface similar to that of other devices.

One of the more popular operating systems currently available on the target machines is UNIX® or its look-alike Xenix®. Another operating system which had the functional capabilities of UNIX v7 is MINIX. The major advantages of MINIX are its well defined layered structure and the availability of its source code. It was on these grounds that it satisfied the basic requirements for the development of the distributed database system.

This paper addresses MINIX as an implementation environment for a distributed database management system, by considering aspects fun-

damental to the implementation. The design of the MINIX operating is given, as well as changes necessary to accommodate the database management system. Certain shortcomings and errors of the MINIX system have been detected. The influence of the solutions to these problems are also discussed.

2. Distributed database management system

The NRDNIX distributed database management system under development must offer users access to database files, presented as data structures of the entity-relationship model [4]. User requests are broken down by the query decomposition algorithm in the presentation manager and the resulting primitive operations are passed via the communication kernel to the database manager module. The database manager is then responsible for the management of the data and the handling of low level operations. The communication kernel also controls access to the network manager and routes primitives access requests to remote sites if necessary. The interactions between these components are illustrated by figure 1.

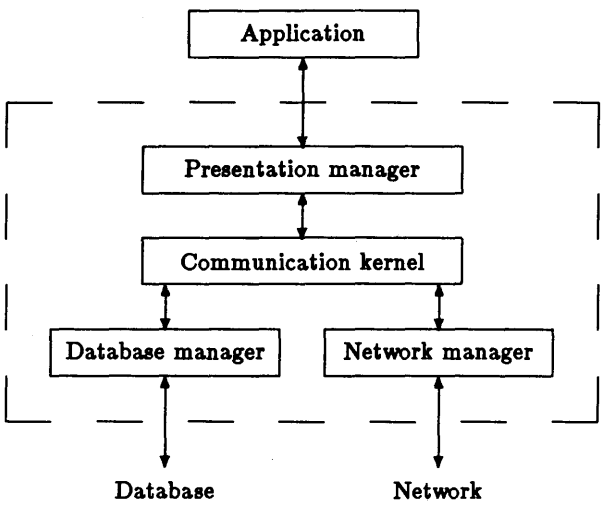

Figure 1: Distributed database components

The major functions of these components are the following:

- Presentation manager: User transactions are decomposed into primitive relational algebra operations which are then passed to the communication kernel. It also interfaces with the users of the system, including formatting the output of the database queries.
- Communication kernel: The distribution and execution of transactions are managed, including concurrency control. It interfaces to the network manager for access to remote data and to the local database manager for access to local data.
- Database manager: All accesses to the local database files are controlled. It manages the actual reading and writing of the data, making use of B-tree indexes.
- Network manager: Error free transmission of subtransactions and data are controlled.

Of these components, the communication kernel forms the cornerstone of the architecture. The main task of the communication kernel is to act as transaction manager. Fragments of global transactions, in the form of primitive relational algebra operations, are passed to the kernel. It controls the correct execution of these as part of a global transaction.

The atomicity requirements of a transaction must be adhered to and it must be ensured that transactions are executed in a serializable manner. This includes handling the concurrency and integrity control for the data in the database. Management of a transaction entails the decision and control to pass a subtransaction to the local database manager for local processing, or to pass it to a remote site via the network for remote processing. Other tasks of the communication kernel include passing requested data to the network manager for transmission to remote sites, as well as accepting and controlling transaction operations in the form of messages from remote sites.

3. MINIX structure

The design of the MINIX operating system is based on the client-server (master-slave) message passing model. MINIX is structured in four layers. Each layer is independent of the others and they communicate through a fixed interface via messages using the message passing primitives *send()* and *receive()*.

The kernel comprises the two lowest levels of the operating system and contains the only device dependant code. Level one is the process management task, which handles all process scheduling. It is also responsible for handling all interprocess communications. Level two contains a number of tasks which make up the drivers for all the I/O devices handled by MINIX.

Two servers, namely the memory manager and the file system, make up the third level of the system. The file system is described as a network file server which is resident on the same site as the rest of the system.

The fourth level, for user processes, initially only consists of the *INIT* process. This is the

root process of all the user processes. At startup, *INIT* ascertains the numbers of terminals which are connected to the system. It then executes a *fork()* and an *exec()* system call to create a shell for each terminal. The process layout is shown in figure 2.

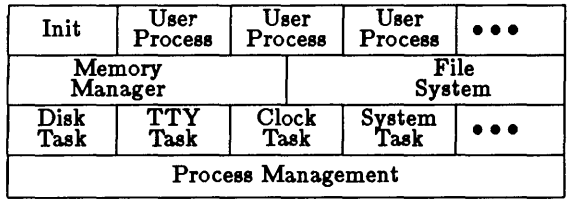

Figure 2: Process structure in MINIX

Process management and scheduling

Process creation is dynamic in MINIX; a child process is created when a process invokes the *fork()* system call. The *exec()* system call then allows the process to execute a specific program. The scheduler uses a three-level queueing system, corresponding to the three upper layers of the MINIX system. When selecting a process to run, the scheduler firsts checks to see whether a process in level one is ready. This level contains all the tasks. If one or more are ready, then the one closest to the head of the queue is run. If no task is runnable, a server (either the memory manager, or the file system) is selected if possible. Otherwise a user process is run. A user process is the only process which can be preempted. This may happen when it has been running for 100 msec.

Inter-process communications

Inter process communication takes place via the message passing primitives *send()* and *receive()*. This enables processes to send fixed size messages to other processes. On the Intel based micro computers this message size is 24 bytes. MINIX uses the rendezvous method for message passing. A sending process is blocked until the destination process executes a *receive()*. A receiving process will block until the source process indicated executes a *send()*. It is claimed to be adequate for this system. The layers in MINIX are strictly enforced when applied to message passing. An operating system process may only communicate with a process in the same layer or the layer immediately below or above it. Processes in layer three may only send messages to processes in layers two, three and four. A user process may only send messages to processes in the level directly below it; the user processes in layer four may only communicate with processes in layer three. A user process may not, for example, send a message to an I/O task. User tasks are limited in that they are only permitted to execute the *sendrec()* call. This is a *send()* followed immediately by a *receive()*. The outgoing message is overwritten by the incoming message.

Device drivers

All device drivers are implemented as tasks in the kernel. These drivers contain all the device dependant code. All other processes gain access to the devices via the message interface offered by the drivers. In order to add a new device to the system, a new driver must be incorporated into the kernel. The necessary tables must be updated to indicate to the file system and the memory manager which task is to handle the I/O. A task is a special type of process which runs to completion once called. Completion implies that the task cannot be preempted. Once a task is executing, another process (possibly another task) can only commence executing when the said task is labelled as not runnable, for example it is blocked on either a *send()* or a *receive()*.

File system

The file system, along with the memory manager, form the second level of the scheduler's queues and the third level of the operating system. All access to the file devices is done via the file system. A user process accesses devices, such as a disk, by referring to files and not physical devices. Even the terminal is seen as a character special file. The file system offers a hierarchical file structure to a user. MINIX maintains a block cache to increase the performance of the file system. A background process is also used which executes a *sync()* system call every thirty seconds, which is used to flush the cache to disk.

More than one disk can be used by the system. A floppy disk can be mounted onto a subdirectory of any active file system. Mounting a disk device onto the */user* directory, for example, would imply that any reference to the */user* directory would in fact be an access to the root directory of the mounted disk device.

The file system executes an infinite loop which entails waiting for a message to arrive, acting on the contents of the message and sending a reply if required. The main system calls entail communicating with device drivers to handle reading from and writing to physical devices on behalf of user processes. A table in the file system indicates which procedures handle which device accesses. When a process requests a disk block, the cache is first examined to determine whether the block is available. If not, a message is sent to the de-

vice driver requesting the block, which is then also placed in the cache.

Memory manager

The memory manager is responsible for allocating and deallocating memory for processes. No form of virtual memory, paging or swapping is implemented. The memory manager maintains a list of holes in memory address order. When allocation of memory is required, for either a *fork()* or an *exec()* system call, the list is searched using a first fit method. Once a process has been placed in memory, it stays there until it terminates. Allocated memory is never enlarged and does not shrink either. The explanation given for this limited implementation is that the design of the system was aimed at small microcomputers, typically the IBM PC and compatibles, which implies that the number of running processes would be small and the available memory should be sufficient. The implementation of any virtual memory would involve the use of secondary storage, which would greatly reduce the efficiency of the system. The inclusion of code to facilitate this feature would also cause the memory manager and file system to be far more complex. Portability is another excuse for a simple implementation; the simpler the system and the fewer the assumptions it makes about the hardware is runs on, the more systems it can be ported to.

The memory manager is a server which runs in user space. It is therefore not part of the kernel, although the actual setting of memory maps for processes is done in the kernel. The two main data structures in the memory manager are the process table and the hole table. Entries in the process table of the file system, the memory manager and the kernel all refer to the same processes; slot k of the file system's table refers to the same process as slot k of the memory manager. The most important element of the table is an array containing the virtual addresses, physical addresses and the lengths of the text, data and stack.

As all other processes in MINIX, the memory manager is message driven. After completing its initialization procedure it enters its main loop, which entails waiting for a message, carrying out the request and sending a reply.

4. Network

The network hardware supplied for the implementation consists of ArcNet communication cards, which operate over a co-axial cable at a specified rate of 2.5 Mbs. The cards offer access to a token passing based network and includes a cyclic redundancy checksum (CRC-16) which is responsible for error detection. Communication control is based on a modified token passing protocol. All token passes are acknowledged by the site receiving the token. This protocol management is handled by the COM 9026's internal microcoded sequencer, present on the network card. The processor only has to load a packet, with the destination ID, into a RAM buffer on the card and issue a command to transmit. When next the COM 9026 receives the token, it issues a *free buffer inquiry* message. If the receiving node sends an acknowledge *ACK* message, the message is transmitted. If a negative acknowledge *NAK* is received indicating the receiver is not available, the transmitter passes the token. If the message is received correctly, an acknowledge message is sent. This enables the receiving site to set a status bit indicating successful transmission. If a faulty packet arrives at the destination site it is not acknowledged. Broadcast messages are also not acknowledged and no *free buffer inquiry* message is sent.

The inclusion of network capabilities for MINIX entails linking a new device driver into the kernel. Due to the fact that only processes in the third level can access this new device, either the file system and memory manager must be modified to handle user calls to the device, or another process must be added to this level. The tables must be updated to show which driver is handling the new device.

The implementation of a network driver poses problems similar to some of those experienced by the terminal driver. Like a process requesting a character from a keyboard, a network driver cannot guarantee a maximum waiting period for the arrival of a packet from a remote site. This means that the requesting process must be suspended to enable the involved server to service other processes.

5. MINIX implementations

The NRDNIX distributed database management system could be implemented with the MINIX operating system in one of the following ways.

Third layer process

A database server could be added to the third level of the operating system, which would then receive requests from user processes in the same way as the file system and the memory manager. It must then communicate with the file system in order to gain access to disk facilities. This would, however, result in slower response times.

Another variation of this type would be for the system to have special files which can only be accessed via the database server. This would imply that the database server would then be solely responsible for the manipulation of data in the special files, resulting in the duplication of certain routines presently offered by the file system. Communication with the file system would then only be required to allocate and deallocate disk space.

File system extension

Specialized routines could be incorporated in the file system to handle database operations. This would avoid the increased difficulties of file management of the above solution. It would also reduce the duplication of code, where the operating system and database system perform similar tasks. The disadvantage of this approach is that the file system will become more of a bottleneck than it already is. As the file system cannot be preempted, the data manipulation operations for a database request would have to be completed one at a time. This would be very advantageous for providing transaction atomicity, but it would cause a serious reduction in the degree of concurrency.

Specialized database file system

The file system could be replaced with a similar database system, which would be specially tailored to handle database transaction operations. This implies that the system would become a dedicated database system and could not be used for normal development. The user interface would be a limited interface consisting of a few select operations, including editing a query file and running a transaction. A more simplified file structure could be offered for the management of user files. This would then limit the system capabilities to only database implementations.

Network operating system process

There are a number of configurations which could be classified as network operating systems. As a disk device can be mounted onto a system such as MINIX, so would it be possible to mount a remote site so that access to a specific directory would in fact imply access to the disk space of a remote site. Concurrency control could then be handled at a lower level by the operating system, depending on the level of concurrency control needed by the operating system itself. If the database system were to implement its own concurrency control, assuming that the control offered by the operating system is inadequate, then the database system would only require timestamps and file names from remote sites in order to handle ordered concurrency control. The actual reading and writing of the remote or local disk blocks would then be functions supplied by the local operating system. The network driver could be implemented in a way similar to that of the disk drivers. Instead of suspending the requesting user process, the file system could actually wait for the remote zone to be transferred. This would decrease the efficiency of the system, but could avoid the deadlock problem described in the subsequent section.

As mentioned, the file system is a network file server which happens to be on a the same site as the rest of the system. This means that the location of the disk being accessed does not concern the other active processes. The main difference between a network file system and a distributed system is that the user would be aware of the remote location of files. One method suggested could be the inclusion of a super root directory. The reference to the file */../siteX/AAA* would be a reference to a file *AAA* in the root directory of *siteX*. The command *cd /../siteX* would make the working directory the root directory of *siteX*.

The file system would then keep track of the site of the working directory. Any message sent to the local file system for access to a remote disk would then be sent via the network to the file system of the relevant site. At the remote site, the message would be processed as if it originated at that site, except that the data for a read request would be sent to the network driver for transmission to the requesting site. For a write request, the data would be written and the message for the completion of the command sent to the remote site.

The problem of maintaining a cache in a network environment is a complex problem. The easiest solution would be to only allow sectors from the local disk to be kept in the cache. Access to a remote disk will then always involve communication, with the possibility of the sector being in the cache of the remote site.

Distributed operating system process

The ideal environment for the development of a distributed database management system is a distributed operating system. The database system could see the networked system as a single centralized system. Depending on the level of tranparancy offered by the operating system, information about the distribution of the data could be used in the optimization of query execution. The concurrency control method used by the database system would be influenced by the the distribution of data, but the operating system

should offer sufficient features to facilitate the development of the concurrency control mechanism for the data base system.

6. MINIX problems

The problems encountered in MINIX are deadlock, alarm signals, deviations from the message passing model and memory limitations. Of these deadlock is the most critical.

Deadlock

MINIX handles a possible deadlock situation in the same way as UNIX does; it ignores the problem. The master-slave model on which MINIX is based, fails under certain circumstances, with respect to the requirement that a slave process must only send a message at the explicit request of a master.

Consider the use of an I/O device where the response is not guaranteed to occur within a specified time. For example, if a process requests a character from a keyboard, then a request must be made via the file system which in turn communicates with the specific device driver. If a character is not available, the device driver indicates to the file system that the user process requesting the service should be suspended until the character is available. The master-slave model proves inadequate in this situation. When a character is received from the device, a message must be sent to the requesting process. This, however, reverses the previous roles of the file system and the device driver. If, at this stage, another process has requested a character and the file system is preparing a message for the device driver, then a master-master conflict situation could occur, with the file system and the device driver both blocking while attempting to send to the other.

The presence of this conflict was recognized in the interaction between the file system and the memory manager. It has been avoided by not admitting the file system to send requests to the memory manager but to only send replies when reacting to requests from the memory manager. The only exception is during startup, when the memory manager explicitly waits for a message from the file system.

Another possible solution to the deadlock problem is the implementation of channels. When a process wishes to communicate with another process it can open a channel. Messages sent can then be directed via a channel and not directly to a process. For a process to send a message, it simply sends to a channel. The next process which does a *receive()* along that channel will then be the recipient of the message. If a process blocks on a *send()* or a *receive()*, it then only blocks on the specific channel. To implement this system, the servers must be multiprogrammed. For each instance of the file system, for example, an activation record must be created. The memory for this must be available in a heap accessible to the file system. An instance of a process must be created for each channel on which a process blocks, so that there is always an instance of the process to continue and avoid the deadlock problem. The process creation and scheduling for such an implementation becomes cumbersome, if it is at all feasible in the limited space available.

Alarm signals

Only one alarm is kept by the system. The standard MINIX system runs a background process which is activated by an alarm which is triggered every thirty seconds. This process executes a system call which flushes the disk cache kept by the file system. If another background process is activated which resets the alarm to trigger every five seconds, then all processes waiting for an alarm signal are activated every five seconds. If this second process is terminated using a *kill* system call, then a message is also sent to deactivate the alarm. The result is that the update process never receives another alarm signal and the buffers are no longer flushed at regular intervals.

Deviations from the model

The implementation of the MINIX system deviates in a few instances from the strict theoretical model on which it is based. The idea that the kernel consists of separate processes making up the different device drivers and the process management implies that each process must have its own data area which cannot be accessed by other processes. As all these processes are linked into one executable space, the separation of data can only be simulated by ensuring that no processes may access variables which may be accessed by any other process. There are, however, a number of variables which are global to all processes in the kernel. Code is also shared amongst all processes.

One violation of the model is in the passing of information between the device drivers and user processes. If a character is requested from a keyboard and one is not available, the user process is suspended. When such a character becomes available, the character is copied to the requesting user's data space and not to the file system. A message is sent to the file system in order to awaken the user process. The file system is thus informed that the user has already received its

character. The only reason that disk blocks are sent via the file system is that the file system uses the data to update its cache.

Memory limitations

The maximum amount of primary memory visible to MINIX is 640K. The memory manager cannot make use of the extended memory capabilities of a larger machine such as a PC/AT. No process may be larger than 64K. The compiler supplied with the system produces code which does not support separate data and code spaces. The system, however, can execute code compiled using another compiler which does. All pointers are 16 bits, which enforce the process size limit of 64K and prevents the implementation of such utilities such as a debugger. The complexity of recovery features may well find the memory limitations a problem which will require at least the implementation of paging or swapping.

7. Conclusion

The layered structure and simplicity of implementation make MINIX an ideal environment for the development of a system which relies on changes being made to the operating system. The role of each part of the MINIX system is clearly defined with each module having a set location. The structure of the system requires that any subsequent additions fall within this structured frame. The functions of each level clearly inidicate the location for new modules.

The MINIX system originally showed great potential for the development of the distributed database management system project. The different possible approaches briefly described in this report indicate the investigations which the availablility of this operating system has spawned.

The flaws discovered in MINIX during the investigations are sufficient to pose extensive problems in the development of the proposed distributed database management system. The above mentioned development possibilities rely on the availability of the source code of the operating system. If an alternative operating system were chosen, the source code would probably not be available. An operating system such as Xenix or UNIX does however offer the capabilities of adding a device driver into the kernel. With the addition of network features, a distributed database system could be developed which would run as a user process. Despite the scheduling drawbacks of this approach, it is one taken by most commercially developed and utilized database management systems. It appears, therefore that a change in the course of the development of this project would be necessary and the adoption of either UNIX or Xenix as the supporting operating system would be advisable.

References

[1] P A Bernstein and N Goodman, [1981], Concurrency Concurrency Control in Distributed Databases, *Computing Surveys*, **13**(2).

[2] S Ceri and G Pelagatti, [1984], *Distributed Databases - Principles and Systems*, McGraw Hill, New York.

[3] M H Rennhackkamp, [1986], *Comparison Of Concurrency Control Methods in Distributed Databases*, M.Sc thesis, University of Stellenbosch.

[4] M H Rennhackkamp, [1990], The NRDNIX Distributed Database System, *The South African Computer Journal*, **1**(1).

[5] A S Tanenbaum, [1981], *Computer Networks*, Prentice-Hall, New Jersey.

[6] A S Tanenbaum and R Van Renesse, Distributed Operating Systems, *Computing Surveys*, **17**(4).

[7] A S Tanenbaum, [1987], *Operating Systems Design and Implementation*, Prentice-Hall International, Inc.