MAKING CANDIDE: A MULTIUSER DBMS IN A LAN ENVIRONMENT

By

HEMA KANNAN

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

UNIVERSITY OF FLORIDA

1994

Dedicated to my
Parents

ACKNOWLEDGMENTS

I would like to place on record my profound gratitude to Dr. Sharma Chakravarthy and Dr. Howard Beck, my thesis advisors, for their support, guidance and inspiration throughout the course of this project. I am grateful to Dr. Herman Lam for agreeing to serve in my committee.

I am indebted to Ms. Ling Li for her friendship and technical assistance in this project. I would like to extend my sincere appreciation to my colleagues at FAIRS and Ms.Sharon Grant at the Database Center for their support and encouragement.

I would like to thank all my friends in Gainesville for their help in making my stay here a *memorable one*. Finally, special thanks are due to my friend Anandhi for her immense help during my graduate studies.

TABLE OF CONTENTS

LIST OF FIGURES

Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

MAKING CANDIDE: A MULTIUSER DBMS IN A LAN ENVIRONMENT

By

Hema Kannan

December 1994

Chairman: Dr. Sharma Chakravarthy
Cochairman: Dr. Howard W. Beck
Major Department: Computer and Information Sciences

It is often necessary for more than one user to share stored data in a database. With shared database comes the problem of concurrency control for preserving data consistency; that is, ensuring that database operations from different users do not interfere with each other. Database operations include both reading and updating stored data. Update involves reading an object from the secondary storage, changing values in the object in main memory, and then writing the object back to the secondary storage. Subsequent access is to the updated object.

Concurrency control has been studied extensively for traditional databases. But research on concurrency control mechanisms for object-oriented databases is still at its infancy.

CANDIDE is an experimental information retrieval system which stores a wide variety of data, such as text, graphics, digitized images, sound, application programs, and computer simulation from different disciplines. Object-oriented database concepts are used to organize these data. Currently, CANDIDE is a single-user system. This thesis involves the design of multiuser CANDIDE DBMS, design of an

efficient concurrency control mechanism, and showing the feasibility of implementing the system in a PC environment.

A client/server architecture is used for this multiuser DBMS. For concurrency control, the hierarchical locking scheme is extended to suit the object-oriented data model requirements. This method exploits the semantics of the data model and provides maximum concurrency control. This scheme is chosen to minimize the number of locks that need to be acquired while accessing a set of objects.

NetBIOS software is recommended for communication between a client and its server, as this is available for implementing the prototype and is compatible with most of the existing LAN including TCP/IP.

CHAPTER 1
INTRODUCTION

Information Retrieval Systems organize a wide variety of data such as text, graphics, digitized images, sound, motion video, application programs, expert systems, and computer simulation from different disciplines. For organizing this information and knowledge new data models, such as object-oriented data model and the closely related semantic data models, offer some advantages over the more widely established relational data model. These include the ability to describe a diversity of different data types, ability to integrate diverse information, and the potential for more powerful searching based on concept recognition and other advanced operations such as machine learning. Some of the salient features of object-oriented databases are

- They support unique identification of objects by system-assigned object identifiers, where an object can represent anything from a simple number to a complex entity.

- Support abstract data types and allow complex objects to be defined in terms of class hierarchies, which captures the IS-A relationship between a class and its subclass (equivalently, a class and its superclass)

- Support inheritance of structural and behavioral properties among object classes in these hierarchies. All subclasses of a class inherit all properties defined for the class and can have additional properties local to them.

The FAIRS (Florida Agricultural Information Retrieval System) project involves developing an experimental information retrieval system called CANDIDE, based on

semantic and object-oriented data modeling concepts. The Extended BNF [Appendix C] of CANDIDE's data definition and manipulation language explicitly supports the abstractions of aggregation, generalization, identification, and classification [Zdo90b]. It differs form its object-oriented counterparts in that

- methods are not part of the objects
- there are no composite objects and
- manipulation of objects are done using classification and subsumption functions [Beck89].

There are two main functions in CANDIDE: i) subsumption and ii) classification [Beck89]. Subsumption relationships determine if one object is a special case of another. Classification is a search technique which correctly places new objects into an existing taxonomy by repeatedly applying the subsumption function. The correct location of the object is immediately below the most specific classes which subsume the new class and immediately above the most general classes subsumed by this new class. Classification involves a combination of depth-first/breadth-first search of the class lattice, beginning at known superclasses of the object to be classified and applying the subsumption function, continuing as long as it succeeds.

Classification as a query processing language: Querying by classification is the process of specifying a query object in the same notation as data objects, and then searching for objects which are structurally related to this query object. Query processing is based on deductive inferencing about object structures rather than a procedural specification of operations.

CANDIDE Single User Secondary Storage Manager: This storage manager stores CANDIDE objects and is implemented in C++. The capabilities of C++ such as inheritance are exploited to represent the objects and operations performed

on them. CANDIDE uses two representations: relocatable objects [Bec93], designed for rapid relocation between various storage devices (especially between disk and main-memory) and instantiated objects [Bec93], which are instances of C++ classes which provide programmers with convenient access to objects in memory. The object storage manager is required to optimize two frequently performed operations. One, sequential access, is retrieving all the objects within a particular class. The other, taxonomic subsumption, is determining whether class A is above class B in the taxonomy. Other operations performed are single-object retrieval and class-induction [Bec94].

CANDIDE secondary storage manager is based on a physical clustering algorithm which attempts to locate objects which are logically clustered as part of the same class structure near the same physical location on disk. The secondary storage manager is based on the concept of virtual memory management used in operating system. Objects are cached into main memory, and when a request is made for an object which is not in main memory, an *object fault* occurs and the disk is accessed to retrieve that object.

The object file contains the data objects. These objects are identified by a unique number called the OID. The object file consists of a set of fixed-sized buckets identified by a bucket number and containing a number of relocatable objects. According to the physical clustering algorithm used, all the objects in a bucket are members of the same class. CANDIDE supports multiple inheritance, that is, an object may belong to more than one class. Multiple identical copies of an object are stored for each class it belongs to. This is done to speed up sequential access, but it slows the updating process to a great extent.

There are two versions of CANDIDE single user system. Version I is a simple version which does not support physical clustering, multiple copies and buffer management, whereas Version II supports all these optimization techniques.

The current storage manager does not have miltiuser capabilities that prevents two users from modifying the database concurrently. The aim of this thesis is to design a multiuser CANDIDE. To achieve this, a concurrency control mechanism is needed. To meet FAIRS user requirements, the multiuser system needs to be implemented in the PC environment.

Concurrency control has been studied extensively for traditional database applications [Bar91]. DBMSs implement concurrency control mechanisms for supporting the concepts of transaction and serializability [Gra93a]. A transaction is an atomic unit that encloses database operations that logically belong together. Users interact with a DBMS by executing transactions. Serializability is a correctness criteria that guarantees non interference among concurrent transactions. A schedule of concurrent transactions is said to be serializable if it is equivalent to a serial schedule; that is, one in which each transaction ends before the next one begins.

Most of the Object-Oriented systems use some form of locking to provide concurrency control. O2's [Ban91] object level locking is handled by Wisconsin Storage Manager, which uses two-phase locking algorithm on pages and files. Open OODB depends on Exodus for concurrency control, which is based on granularity hierarchical locking on pages and objects. ObServer [Zdo90a] uses locks too, but the non serialiazable behavior is controlled through *notify* locks. A notify lock held by a session can make it be aware of other sessions accessing the locked data. All these locking schemes are based on read-write semantics. Here the semantic information of the schema is not considered. An optimistic concurrency control scheme is used in Gemstone [Mai85].

Object-Oriented databases present an opportunity to provide better concurrency control based on the semantics of an object. Here the database system knows more about the operations that are being performed. They are not simply reads or writes. The amount of information used by the storage manager also plays an important role in the implementation of concurrency control mechanisms. For example, ObServer [Zdo90a] and Exodus store very little semantics in the storage manager. The advantages of this approach include simplicity, the ability to support multiple external models and the semantic functions are handled by an interpreter at a higher level. On the other hand, systems, such as Gemstone [Mai85] store additional semantics in the storage manager. This is used when accessing a class and its subclasses. Thus storage management for object-oriented database systems (OODBS) is in its infancy. There is still an enormous design space to explore in terms of clustering, object assembly in main memory, replication and distribution.

ORION [Gar88] and O2 [Car90] have built concurrency control mechanisms based on the semantics of the data model. Both the systems adapt the locking technique to provide concurrency control. ORION extends the granularity hierarchical locking to suit object-oriented system. This minimizes the number of locks requested by a transaction by implicit locking. O2 exploits the parallelism provided by methods of objects. Compatibility of methods are used to lock objects.

The concurrency control mechanism proposed in this thesis for CANDIDE is based on the approach taken in ORION. This approach is modified and extended to exploit the semantics of the CANDIDE data model and provide maximum concurrency. The hierarchical and implicit locking proposed in this thesis minimizes the number of locks used while accessing a set of objects. This method is discussed in Chapter 4 in detail.

The biggest challenge was to demonstrate the feasibility of implementing this system in a PC environment. There are two LAN architectures i) peer-to-peer and ii) client/server. In the peer-to-peer architecture, different users access the server through a logical drive. Physical locking is provided by the SHARE [Art94] command in DOS. There is no concurrency control mechanism provided by the server to maintain the consistency of the database. This architecture is useful while transferring file and for read only operation, for CD-ROM like devices.

In the client/server architecture, the client requests the necessary data and the server after checking the necessary constraints, sends the requested data. Thus by controlling the flow of data, concurrency control can be provided at the server.

LANtastic [Mon91] being a peer-to-peer LAN could not be used for providing the concurrency control mechanisms required in CANDIDE. However the communication software provided by LANtastic is used for transferring data between two PCs. A client/server DBMS is built with NetBIOS [Nan90, Sch85, Sco88] as the communication software. The server maintains all the low-level operations like locking, buffering, and so forth. Thus when the server receives a request from a client it checks the lock table for compatibility, and if compatible, services the request. Otherwise it queues the request. The reason for choosing NetBIOS is that it is compatible with most of the existing LAN protocols and a prototype would be built on LANtastic.

This thesis involves the complete design of multiuser CANDIDE DBMS and conceptual implementation for the Version I (which does not involve buckets and physical clustering). Also it shows the feasibility of implementing this system in a PC environment.

This thesis is structured as follows: Chapter 2 surveys literature useful for multiuser DBMS and its implementation in a PC Environment. Chapter 3 discusses the current CANDIDE single user secondary storage manager, both Version I and Version

II. Chapter 4 gives the design for multiuser CANDIDE for Version I and Version II. Chapter 5 discusses the implementation of Version I of CANDIDE multiuser system. Finally, the summary and future work are discussed under conclusion.

CHAPTER 2
LITERATURE SURVEY

It is often required that more than one user be allowed to access the data in a database. With shared database comes the problem of concurrency control, that requires ensuring that database operations from different users do not interfere with each other. Database operations include both reading and updating stored data. Update involves reading an object from the secondary storage, changing values in the object in main memory, and then writing the object back to the secondary storege. Subsequent access is to the updated object.

Interference can occur, for example, if a second user reads the object for update after the first user has read it, but before the first user has written it back to secondary storage after completing his work. Whichever of the users writes first, that update will be lost (it will be overwritten by the other update).

This chapter discusses the requirements and issues involved in developing a multi-user database. The first section describes the client/server architecture and its suitability for multiuser DBMS. Then the issue of communication protocols' suitable for multiuser system in a PC LAN environment is discussed. Finally, the various concurrency control mechanisms are described.

2.1 Client/Server Architecture

Client-Server computing is the latest trend in the development of multiuser database systems and the local area network (LAN) technology. In the client-server computing paradigm, one or more clients and one or more servers, along with the underlying operating system and interprocess communication systems, form a composite system

allowing distributed computation, analysis, and presentation. Typically, the client is a process which interacts with the user and a server is a process, or a set of processes all of which must exist on one machine which provides a service to one or more clients. Thus the processing is split between the client and server, which is the primary advantage of client/server architecture.

The following are the basic database capabilities needed today.

1. *data staging:* The volume of operational databases keeps growing, and several applications have already passed well beyond the effective range of query optimization and processing. Therefore, data caching mechanisms are necessary for extracting chunks from these *massive* databases and using them in a user-tailored operational region.

2. *database interoperability:* Multiple already deployed databases must be made accessible through a cooperative environment for exchanging data, bindings, and control on a continuous basis. This will allow invaluable data correlation and smooth update propagation.

3. *multi-user access:* More than one user should be able to simultaneously access the database. The consistency of data and serializability of transactions should still be maintained.

All the above requirements can be met by the client-server architecture. Thus, the client/server architecture is being adapted for multiuser databases.

General Client Characteristics

- It presents the User Interface [UI]. This interface is the sole means of garnering user queries for data retrieval, analysis, as well as presentation. Different UI's can exist on each client.

- It converts one or more queries or commands in a pre-defined language for presentation to the server.
- It communicates with the server via a given interprocess communication methodology.
- It performs analysis on the query or command results from the server before presenting it to the user.

General Server Characteristics

- It provides service to the client.
- A server merely responds to the queries or commands from the clients. (Does not initiate a conversation with any client.)
- Ideally a server should hide the entire composite client-server system from the client and the user. (Hardware, platform, and network transparency.)

Roussopoulos and Delis [Rou91] present three client-server DBMS architectures, viz, Standard Client/Server Architecture(CS), the RAD-Unify Type of DBMS Architecture (RU), and Enhanced Workstation-Server Architecture(EWS). In the CS architecture, the applications are run on the clients and the database operations are performed in the server. Though, this architecture distributes the applications, it still retains most of the main resource bottlenecks of a centralized DBMS. In the RU architecture, the server performs the low-level operations such as locking and page read/writes and the client performs query processing. Though, this architecture splits the processing between the client and the server, it is still dependent on the server's I/O bandwidth and the load on the data manager. In the third architecture, EWS, each client maintains a full-fledged DBMS by downloading and caching query results. This increases the autonomy of the workstations and reduces the network traffic. The

performance characteristics of these three architectures are discussed in [Rou91] in detail.

2.1.1 Advantages of Client/Server Databases

The primary advantages of a Client/Server system arise from the splitting of processing between the client system and the database server. Since most of the database processing is done at the back-end, the speed of the server is not tied to the speed of the workstation.

This division of work also reduces the load on the network. Instead of sending the entire database file over the network, only the required data is sent, thus reducing the network traffic.

Another benefit of separating the client from the server is application independence; users aren't limited to one type of system. Users can access the database from different front-ends and from different systems.

Another major advantage of a client/server database is the preservation of data integrity. The DBMS provides transaction processing, which tracks changes to the database and helps correct errors in the database in case the server crashes. Transaction processing system keeps a running log of all modifications made to the database over a period of time. These capabilities make Client/Server systems ideal for large multiuser databases, which allow simultaneous modifications to the data. Locking and deadlock avoidance are handled by the DBMS.

2.1.2 Disadvantages of Client/Server Databases

The major disadvantage of Client/Server systems is the increased cost of administrative and support personnel who maintain the database server. There is also increase in hardware costs as compared to single-user systems.

The complexity of the Client/Server system is high. Having multiple front-ends to the database increases the amount of programming support needed, because more

and varied program code must be developed and maintained. When there is any change made to the structure of the database, it becomes a longer and more complex process to make necessary changes to different front-end applications.

2.2 Developing a Client-Server DBMS for PC Platforms

The implementation of client-server systems depend on the platforms on which the front and back ends run on, and the degree to which the processing is split between the two. PCs are the most widely used platform today, and lot of research is being done on making PCs suitable for client-server architecture . This section deals with the requirements and issues for PC environment and later getting the DBMS onto the Internet.

2.2.1 Requirements and Issues

Only in recent years have IBM-compatible PCs become an acceptable platform for Client/Server databases. The advent of high-powered 32-bit 80486 and pentium systems, hard disks in gigabyte range, and stable network operating systems make these PCs able alternatives to the RISC workstations and minicomputers that have been the traditional platforms for resource-intensive DBMSs. This section discusses the hardware and software requirements for developing a PC based Client/Server database.

Hardware

A PC based on a 80386 CPU can perform adequately as a database server, but for sheer power and future growth potential, the best system to start with is one based on a 66Mhz 80486. Newer PCs come with the CPU on a replaceable card, which makes it easier to upgrade the power of the system.

A RAM (random access memory) of 12Mb gives adequate performance, but depending on the number of simultaneous users and the DBMS, more RAM (upto 32MB) will be required.

The most critical component for database performance is the hard disk subsystem, since the bulk of the DBMS's activities involve reading data from or writing data to it. Benchmark tests have shown that the best choice for speed and expandibility is a hard disk based on the small computer system interface (SCSI) standard. A single SCSI board can support up to seven attached drivers, each as large as 3G, giving tremendous room for expansion.

Finally, the type of internal bus (data connection system for add-on cards) on the server should be decided. A 32-bit bus will speed up both disk processing and network access when 32-bit interface cards are used, eliminating the most serious bottlenecks a server faces.

Multiprocessor (MPU) systems, an emerging technology in the PC world, may have a significant impact on the performance and capabilities of Client/Server databases in the future. However, at this time few MPU systems are available, and little or no support for them exists among the various C/S products.

Operating System Software

OS/2 is a preemptive multitasking, multithreaded OS designed expressly for PCs. It provides superior capabilities over DOS for the resource-intensive database server software; its multitasking lets multiple services run on the same system, including both LAN software and the DBMS. It has native support for up to 512Mb of virtual memory (VM), but this OS is not widely used.

Novell's NetWare network operating system [Chr90] provides nonpreemptive multitasking capabilities. It has the ability to run applications on the File Server as NetWare Loadable Modules (NLM).

Some versions of UNIX run on 80486 systems, and can be used as the OS for PC-based Client/Server databases. However, running UNIX on PCs is not that common, and it is generally better to go with a hardware platform specifically designed for it if you must run C/S software that only comes in UNIX versions.

Communications

A Client/Server Database System depends on splitting the processing between an intelligent front-end system and the database server. Networks allow communication between these two parts of the overall system.

The client systems communicate with the server through a network that consists of a combination of hardware and software. The hardware which connects the PC to the network's wiring consists of a network interface card (NIC) that is added to the PC workstation, and server. A network of PCs, workstations and servers is referred to as a local area network (LAN) if all the systems fall within a small area. When a LAN extends across a wide area of distance, then entire network is referred as Wide Area Network (WAN). Three LAN topologies (cabling schemes) are in common use today: Ethernet, ARCnet, and Token Ring.

LAN uses network protocol for communication between two machines. Many proprietary network protocols exist today, but only a few are relevant to designing a Client/Server Database System.

NetWare LANs use the Novell IPX/SPX protocol [Chr90]to provide communications between workstation(s) and server(s). Microsoft's LAN Manager and IBM's LAN Server use variations of the NetBIOS protocol [Sch85, Sco88], and DEC-based LANs use the DECNet protocol. IBM mainframes primarily use System Network Architecture (SNA) to communicate with terminals and LAN Gateways, and IBM's LAN also support the Data Link Control (DLC) protocol for direct communication between the LAN PCs and the mainframe front-end processors.

The most common cross-platform protocol is the Transmission Control Prototcol/Internet Protocol (TCP/IP). Originally developed as the UNIX networking protocol. TCP/IP is now available for almost every platform and operating system.

Two Network Operating Systems (NOS) and their protocols are discussed below. LANtastic [Mon91] is discussed because, it is used at ourdevelopment site, and the prototype of CANDIDE multi-user system will implemented on this LAN. Novell's Netware [Chr90] is discussed because, it is the most widely used LAN and this multi-user system might have to be ported to Novell's NetWare. The discussion includes the protocols and their compatibility with other LANs.

2.3 LANtastic

2.3.1 Network Operating System

LANtastic NOS [Mon91] is a peer-to-peer LAN. LANtastic is designed as a bus type network. Standard CSMA/CD scheme is used for sending and receiving data and is implemented by LANtastic's NetBIOS. All programs of LANtastic fall into the following five categories: low-level drivers, NetBIOS, the redirector program, the server program, and network utilities.

The purpose of the low-level driver is to establish an interface between the software and the adapter installed in the computer. NetBIOS [Sco88] performs all basic network functions. The low-level driver provides the low-level communications function, and the NetBIOS provides the high level functions. The *redirector* is the software that intercepts commands and printer output that normally would be handled by DOS and routes them to the appropriate location on the network. Figure 2.1 shows how commands are redirected from a workstation to a server. The server program allows the computer to share its disk drives and printers with other computers and to manage other services like print spooling. LANtastic comes with several network

utilities that provide optional functions like disk caching and adapter diagnostics, used for special situations and to improve the network performance.

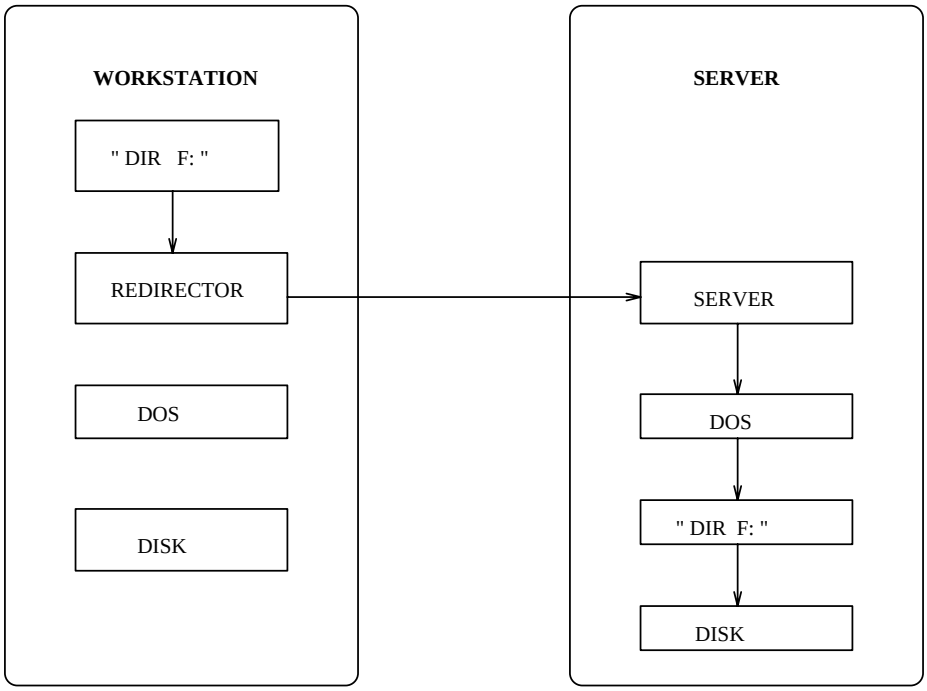

Figure 2.1. Commands are redirected from a workstation to a server

Features of NetBIOS are discussed because the prototype CANDIDE multi-user system uses this protocol for client server communication.

2.3.2 NetBIOS

NetBIOS is the software that allows the network operating system to work with the network adapter boards, and Artisoft's LANtastic is based on the NetBIOS protocol. It also allows network utilities and networked application programs to function with the network hardware.

NetBIOS performs four main functions: creating and managing sessions, creating and transmitting datagrams, keeping track of network names, and providing point-to-point communications.

Session

Before two network nodes can communicate in both directions, NetBIOS must establish a session between them. This is the dialogue between the two nodes (at the session layer of the OSI model) that establishes some ground rules for the communication about to take place. Among other things, it determines if the two computers will exchange data simultaneously or if they will take turns sending data.

Datagrams

In addition to the data packets that are sent between nodes on the network, NetBIOS is able to assemble and send datagrams. A datagram is the simplest type of network communication. It is a small block of data that can be transmitted to any single username, any group of usernames, or to all users (called a broadcast datagram). Unlike a session, a datagram is a one-way communication and is less reliable because there is no verification that the data was received.

Network Names

NetBIOS creates and manages network names so that a node does not have to be created with a specific user. Network names are created and deleted as needed, allowing a node to be used by different people at different times. The names are also used for sending messages and datagrams.

Communication

NetBIOS also provides for point-to-point communication on the network. All of the nodes on a network are linked through a *virtual circuit*. This virtual circuit is the way that drives and printers are shared on the network. In addition to these four types of functions, NetBIOS is able to monitor the status of all adapters on the network.

2.4 NetWare

NetWare is a client/server Network Operating System [Chr90] offered by Novell. It provides the core functions needed to maintain the most basic network operations like network file system, memory management and the scheduling of processing tasks. Among the important functions performed by the client/server operating system are coordinating the use of all resourses and services available from the server, ensuring the reliability of data stored on the server and managing server security.

2.4.1 Network Operating System

The Network Operating System resides on the file server and controls all the networking functions. DOS resides at the workstation and controls the workstation devices. Since the Netware and DOS are two separate operating systems, a small program called SHELL enables the workstation to talk to the file server. Anytime the workstation makes a request, the shell directs it to Netware or the DOS depending on the type of request. It also controls the security of the network. It must also be able to locate the addresses on the network. It must also have a variety of network management tools for the network administrator. Finally it must also provide support for the peripherals such as printers, bridges, gateways and other network devices.

Whenever the workstation is booted up, the Netware Shell is loaded into a workstation's RAM as a RAM resident program. The shell includes the following files.

- NETX.COM
- IPX.COM
- SPX.COM

The **IPX.COM** file is the communication protocol which creates, maintains and ends connection between network devices (workstations, file servers, etc). IPX addresses and routes outgoing data packets across a network. For returning data, IPX

reads the address and directs the data to the proper area in a workstation's or server's operating system. IPX is closely linked with other programs and routines that help in the data transmission process.

IPX can be adapted to different network boards by running WSGEN or DCONFIG, so IPX can route and accept data packets through physically different networks. Once the IPX.COM is run, a workstation can communicate with network. Without the IPX.COM in the workstation, there is no way workstation can communicate with the file server.

SPX.COM is also a Novell NetWare's communication protocol that ensures successful delivery. SPX enhances IPX by supervising data sent across the network. SPX tracks data transmissions consisting of separate packets. It also requests and returns acknowledgments from a communication partner, ensuring successful data delivery.

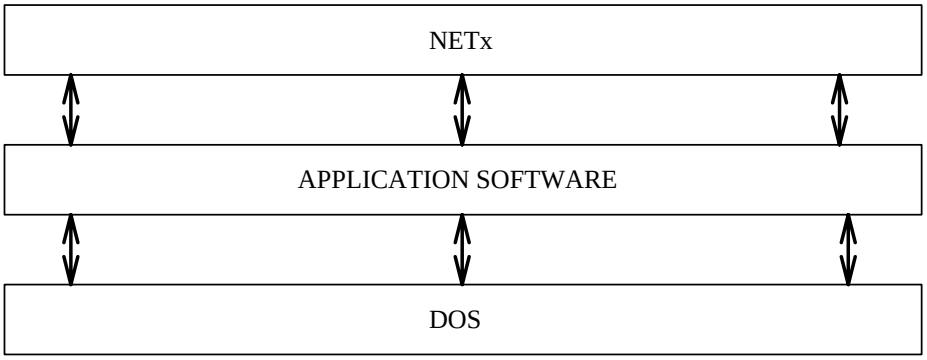

Figure 2.2. Interaction of DOS, Application Software and NETx

If an acknowledge request does not bring any response within a specified time, SPX retransmits it. After a number of retransmissions are not acknowledged, SPX assumes the connection has failed and warns the operator of the failure.

NETx.COM Netware shell program works with IPX, SPX, and a LAN driver to convert a stand alone computer into a network station. It is loaded into the RAM each time a workstation boots and begins network transmission. The NETx program

lies on top of workstation operating system between the application layer and DOS. It monitors all the data transmission moving in and out of DOS or the application layer.

If an application request, such as a call for files, needs to be handled by the file server, NETx intercepts the request and begins protocol conversion and transmission to the file server.

NETx intercepts requests by taking over software interrupts 21h (used to call standard DOS functions), 24h (DOS critical error handler vector) and 17h (used to send data to local printer ports). The shell intercepts and inspects all Int 21h DOS requests. Then the shell either passes the request on to the regular DOS interrupt routine or handles the request itself. If the shell keeps the request, it converts it into the Netware Core Protocol and hands it to IPX for transmission to the file server. For data returning from the file server, the conversion of requests is handled in reverse order. Whether the request is handled by local DOS of the file server is transparent to the application and to the user.

NCP Service protocol The process of requesting service from a file server begins in the workstations RAM where the NETx.COM forms requests according to the definitions of the file server's Netware Core Protocol. The shell then hands the requests to the IPX. IPX then transmits the requests to the file server after attaching a header. Upon receiving the request, the file server removes the IPX header and reads the request. NCP service protocol exists for every service a workstation might request from a file server. These are the procedures that a file server's operating system follows to accept and respond to workstations requests. The common requests include are:

- creating and destroying a service connection.
- manipulating the directories and files.

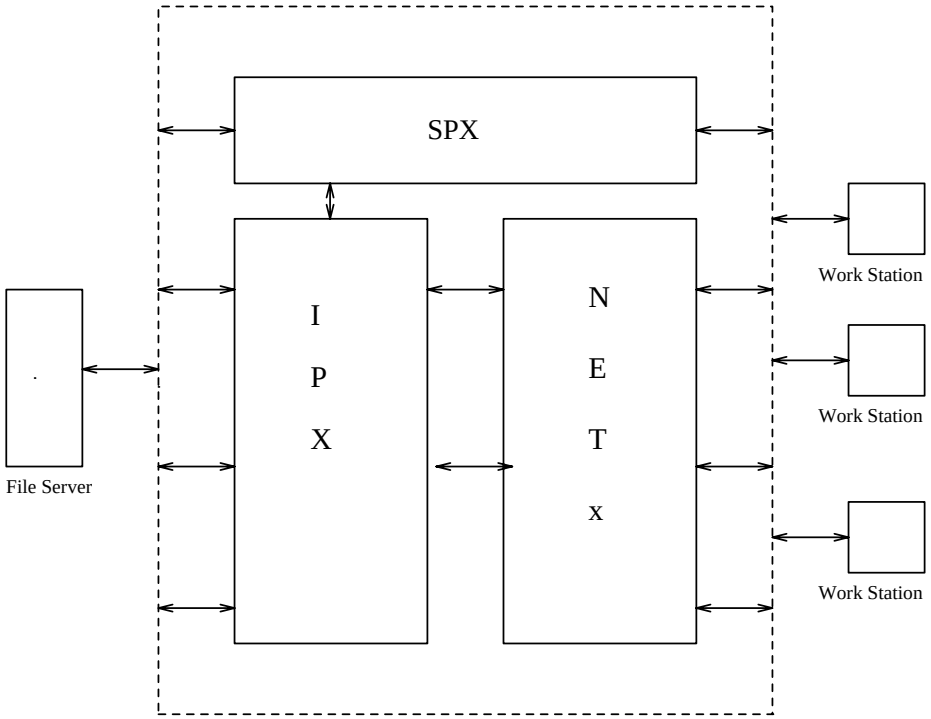

Figure 2.3. Communication Process in Netware

- opening semaphores.
- altering the bindery (drive mappings and security)
- printing.

2.4.2 Transaction Tracking System

NetWare offers System Fault Tolerance (SFT) tools for protecting the system against faults. Netware provides the following protective measures for your data:

- file allocation table [FAT] corruption protection
- hot fix
- disk mirroring
- disk duplexing

- UPS monitoring system
- transaction tracking system (TTS)

We will discuss only TTS in detail, because this is what we are trying to incorporate in CANDIDE. TTS prevents database corruption if the system fails while data is being written to database files. TTS is designed to work with sequential databases.

TTS views the entire sequence of database changes as a single transaction that must be fully completed or fully backed out, (as if no changes made at all). Only after all files have been correctly updated will the transaction be completed and released. This is the atomicity property, one of the ACID properties of a transaction.

If the system fails during a transaction, TTS performs an automatic roll back. That is, TTS will undo all database changes made during the transaction and returns the database to its original state. The database files and underlying system information are left as they were before the transaction began. The lost transaction will have to be re-entered, but all files and system information will be intact and consistent.

TTS accomplishes all of this through physical and logical record locks and the TTS attribute. TTS attribute is a flag which tell the system to track the transaction. TTS ensures that each transaction will be tracked to ensure that

- either all of the write request will be written to the datafile on a drive, or
- none of the data will be written to the datafile

Without TTS it would be almost impossible to determine the status of a datafile after an interruption such as a power failure, a hung workstation, a file server failure, and so on.

2.5 Concurrency Control Mechanisms

Concurrency control has been studied extensively for traditional database applications. Database management systems implement concurrency control mechanisms

based on the concepts of transaction and serializability. A transaction is an atomic unit that encloses database operations that logically belong together. Users' interact with a DBMS by executing transactions. In traditional DBMS trancations serve three distinct purposes: (1) They are logical units that group together operations comprising a complete task; (2) they are atomicity units whose execution preserves the consistency of the database; and (3) they are recovery units that ensure that either all the steps enclosed within them are executed or none are. Thus, by definition, if the database is in a consistent state before a transaction starts executing, it will be in a consistent state when the transaction terminates. In a multiuser system, users execute their transactions concurrently. The DBMS must provide a concurrency control mechanism to guarantee that consistency of data is maintained in spite of concurrent accesses by different users.

Serializability is a correctness criteria that guarantees noninterference among concurrent transactions. A schedule of concurrent transactions is said to be serializable if it is equivalent to a serial schedule, that is, one in which each transaction ends before the next one begins.

There are several techniques for serializing transaction schedules, and they are locking, time stamp ordering and optimistic approach. All these are discussed briefly below.

2.5.1 Locking

One way to ensure serializability is to require that access to a data item by a transaction is allowed only if it is currently holding a lock on that item. There are two modes in which a data item may be locked: Shared mode (read lock) and Exclusive mode (read/write lock).

Two-Phase Locking

The two-phase locking mechanism (2PL) [Esw76] is commonly used concurrency control mechanism in conventional DBMSs. This protocol requires that each transaction issue lock and unlock requests in two phases:

- Growing Phase: A transaction may obtain locks but may not release any lock
- Shrinking Phase: A transaction may release locks but may not obtain any new locks.

If a transaction tries during its growing phase to acquire a lock that has already been acquired by another transaction, it is forced to wait, which could lead to deadlocks. This protocol is adopted when working on individual items. There are situations where, release of locks earlier than the shrinking phase will not affect serializability. In such cases, 2PL does not provide maximum concurrency. In order for this protocol to do better, we need additional information about the order in which the data items are accessed.

Tree Protocol

In the absence of information about how and when the data items are accessed, however, 2PL is both necessary and sufficient to ensure serializability by locking. In some applications, it is often the case that the DBMS has prior knowledge about the order of access of data items. The DBMS can use this information to ensure serializability by using locking portocols that are not 2PL. One such protocol is the tree protocol [Hen91], which uses the information on the order of access to form conflict serializable schedules. The advantage here is unlocking may occur earlier, which might lead to less waiting time on locks and increase concurrency. Also, deadlocks can be avoided. The disadvantage of this method is that a transaction may end up locking items which it will not use.

Multiple Granularity Locking

The concurrency control mechanisms discussed so far operate on individual data items to synchronize transactions. There are circumstances, however, where it would be advantageous to group several data items (e.g., all instances of a class) and treat them as one individual synchronization unit.

Multiple granularity concurrency control protocol introduced by Gray et al. in [Gra88] aims at minimizing the number of locks used while accessing a set of objects in a database. This model [Gra88] organizes data items in a tree where small items are nested within larger ones and each nonleaf item represents the data associated with its decendents. Orion [Gar88] adapted this method for concurrency control mechanism for object oriented databases. The root of the tree represents the whole database. Transactions can lock nodes explicitily which in turn locks descendants implicitly. Apart from Shared (S) and exclusive (X) mode locks, there exists an *intention* lock mode. Intention mode lock on a node implies that explicit locking is being done at a lower level of the tree. A nonleaf node is locked in intention-shared (IS) mode to specify that descendant nodes will be explicitly locked in Shared (S) mode. A shared and intention-exclusive (SIX) lock on a nonleaf node implies that the whole subtree rooted at the node is being locked in shared mode and that explicit locking will be done at a lower level with exclusive mode locks.

2.5.2 Timestamp Ordering

One of the problems of locking mechanisms is the potential for deadlock. This problem can be solved by assigning each transaction a unique number called a timestamp, chosen from a monotonically increasing sequence (usually a function of the time of the day). Using timestamps, a concurrency control mechanism can totally order requests from transactions according to the transactions' timestamp. This method avoids deadlock situations.

2.5.3 Optimistic Nonlocking Mechanism

The locking approach has the following disadvantages [Hae82, Kun81]:

- Lock maintanence and deadlock detection represent substantial overhead
- There are no locking mechanisms that provide high concurrency in all cases
- Not permitting locks to be released except at the end of transaction, which although not required is always done in practice to avoid cascading aborts, decreases concurrency.
- Most of the time it is not necessary to use locking to guarantee consistency since most transactions do not overlap; locking may be necessary only in worst cases.

To avoid these disadvantages, *optimistic* concurrency control mechanism was introduced. Here, each transaction goes through three phases: a read phase, a validation phase, and possibily a write phase. During the read phase all reads take place on local copies. During the validation phase, the local changes are made global if these changes do not affect the serializability with respect to all committed transactions. Else the transaction is either rolled back or aborted. This a serious disadvantage, because work done by a transaction will go waste. Thus this method would work well when there are few read-write conflicts. And only during the write phase the changes become accessible by other transactions.

Locking schemes guarantee one consistent image of the database at every commit point. Also they provide the facility of selecting an appropriate level of control. Optimistic concurrency control schemes create several private data copies during the transactions execution for the sake of enhanced concurrency, but face difficulties at the COMMIT stage, when these copies do not match. This approach is chosen in

applications where conflicts are unlikely. Since locking also behaves quite well in such an environment (no wait or deadlock conflicts) there seems to be little reason to introduce a specialized control mechanism. Chapter 4 discusses the concurrency control mechanism for CANDIDE in detail.

CHAPTER 3
CANDIDE SINGLE USER SECONDARY STORAGE MANAGER

This chapter describes the working of and major data structures used in the implementation of the Version I of the CANDIDE single user secondary storage manager. The reasons for transition from Version I to the improved Version II, is explained next. Finally the concepts and ideas used in Version II are discussed.

3.1 CANDIDE Single User Secondary Storage manager (CSUSSM) Ver. I

3.1.1 Object Representation

CANDIDE uses two different representations for its objects:

Relocatable Objects: Relocatable objects are designed to provide maximum compaction and speed. They can be rapidly swapped between storage devices (especially between main memory and disk). These objects can be located physically at any memory address. Relocatable objects are variable length blocks of characters. Within the object there are pointers to information stored in that object, and these pointers are all relative offsets from the first character in the block. Since these are compacted binary data structures, they are difficult to manipulate.

Instantiated Objects: This representation uses C++ classes and structures to represent all the components of the object, and is much easier for application programs to create and modify these objects. Relocatable objects can be transformed to instantiated objects by picking apart the object internals and instantiating each component into a C++ instance. These objects take up a lot of space, they are not relocatable, and the process of instantiating them is very slow.

3.1.2 Object Storage Management

The object storage manager is designed to retrieve, save, modify and remove objects from the secondary storage. The secondary storage manager mainly uses relocatable objects, since it is not concerned with object internals. Instantiated objects are not handled by the secondary storage manager.

3.1.3 Overview of the Files and Data Structures

Each object is identified by a unique number called the OID. The OID of an object exists in two forms, the string OID and integer OID. Most database functions utilize the integer OID, but the string OID is maintained for ease of readability. Functions are available for convertion between string and integer oids.

The basic operations are retrieving and storing an object, given the object's OID. The major data structures and files include the classes Database, Object_Buffer, String_Buffer, EMS_Buffer and files Strhash.dat, OID.dat, OBJ.dat and fileindx.dat. The Database class provides the general programming interface to the storage manager. The interface functions are described at the end of this section.

There are three tables which hold all the necessary details for locating objects. The files which hold these tables are

- String Buffer
- Object Buffer
- Fileindex
- EMS Buffer

String Buffer is a buffer which is divided into buckets of size 512 bytes each. Each bucket is identified by a bucket number. The bucket contains the type (class or instance), interger OID, and the string OID.of each object hashing into that bucket.

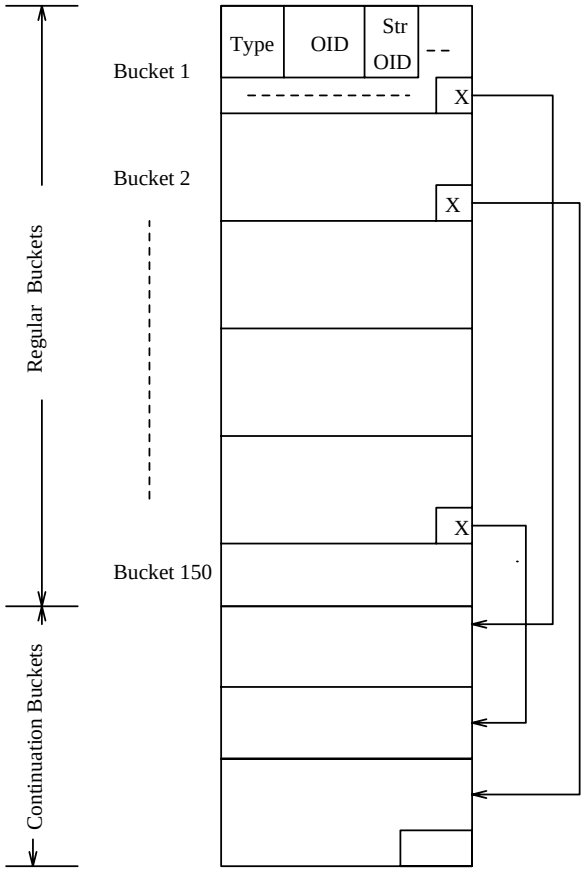

Figure 3.1. Structure of Buckets in Hash Table

The string OID hashes to its bucket number in which its details are stored. When more entries are to be stored in the hash table bucket than that would fit, a chain of buckets are used. The arrangement of the buckets in the string buffer is as follows. There is a fixed number (max-bucket-number) into which the string OIDs can hash into. While loading the buffer into the memory, extra buckets are created to be used as chain buckets. The first bucket of any chain must be one of the max-bucket-number buckets and the rest of the buckets of a chain are buckets with numbers greater than max-bucket-number. The bucket number of next bucket in a chain is maintained in the last byte of each bucket. The whole arrangement is is depicted in the Figure 3.1.

Object Buffer manipilates the OID table. The OID table is used to locate objects in the object file and in the string hash buffer. This table is keyed on integer

oid. The entry corresponding to each OID contains a file ID of the OBJ.dat in which the object is located, the offset into the object file and the exact location of the string OID in the String buffer hash table. The purpose of maintaining the location of the OID in the hash table is to speed up the lookup process for that string OID. This would avoid sequential search in the bucket in the hash table. The structure of the OID table is shown in Figure 3.2.

Fileindex translates file IDs to physical filenames. The database on disk is contained in one or more object files (OBJ.dat). A particular object can exist in one and only one object file, which is identified by the file ID. Multiple object files are supported to allow for multiple projects. This enables a particular project to store all of its objects in one file. Object_Buffer class performs all of the operations on objects, such as retrieving, storing and deleting them from the database.

EMS buffer: EMS (Expanded Memory Service) is used as a buffer to cache objects from disk. OID table and hash table are also maintained in the EMS buffer. The class EMS_Buffer performs all operations on the EMS buffer. When an object is requested, the EMS buffer is searched first, if the object is not found in the buffer, then the disk is accessed. Each time the disk is accessed, BUFFER-SIZE (currently 8K) bytes are brought into EMS buffer. Thus, the number of I/O done is reduced.

The Database interface functions are explained below.

unsigned long oid(char *stroid): This function takes a string OID and returns the corresponding integer OID. The bucket number of the string oid is identified using the hash_value(str) function. Once the bucket number is obtained, the bucket is searched sequentially. All buckets in this chain are also searched. If the string entry in the bucket matches the given string, then the corresponding integer OID is returned.

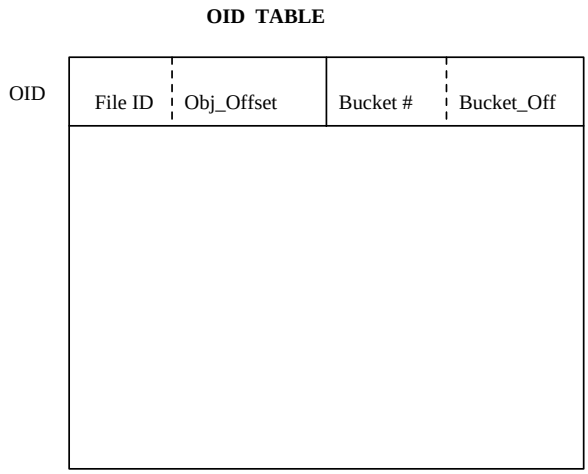

Figure 3.2. Data Structure of the Tables

char *stroid (unsigned long oid): This function is the inverse of the above operation. It takes in the integer OID and returns the pointer to the string version of integer OID. This function gets the location (offset) of the OID in the hash table from the OID_table. Using this offset, the bucket number and the exact location of the OID in the bucket is obtained. The corresponding string oid is returned.

unsigned long newoid(char *stroid): This function creates a new oid for the string pointed by stroid and stores the string oid, integer oid and the type of the object in the string hash table and makes an entry in the OID_table. The location of the new oid in the string hash table is stored in the OID_table. The function sting_buffer− >new_oid(stroid, type, oid) returns this location. This function identifies the bucket number of the OID and stores the type, oid and stroid. If the bucket is full, it is extended using the bucket chain concept.

char *retrieve(unsigned long *p): When an object of OID is requested, the location of the object in the object file is obtained from the OID_table. The EMS buffer is searched first, if the object is not found then the object file is accessed to retrieve the object.

int exists(unsigned long *p): This function takes in an integer OID and checks to see if that object exists in the database. If there exists a pointer corresponding to the OID in the OID_table the object exists and this function returns 1, else it returns 0.

void store(char *obj): This function stores the object "obj" in the secondary storage. From the obj structure, the OID and the length of the object are obtained. This object is stored at the end of the object file and the corresponding entry in the OID table is changed to the new location. Also, update of this object in the EMS buffer is done, if it is part of the buffer.

void delete(unsigned long oid): This function removes the pointer for the given oid from the OID_table and removes that object from the RAM. Note the object is not removed from the object file. It is currently left as dead space and the OID still exists.

3.2 CSUSSM Ver. II

The most frequently performed operations are sequential access and taxonomic subsumption. Version I of CSUSSM does not have techniques to speed up these operations. The improved CSUSSM - Version II has incorporated the following to optimize the above mentioned frequently performed operations.

- buckets – the object file consists of a set of fixed-sized buckets identified sequentially by bucket number. Each bucket contains a number of relocatible objects. The bucket becomes the unit of transfer instead of an relocatable object. So, in a single I/O operation, a number of objects are brought into the memory, thus reducing the number of I/O operations performed.

- memory cache – buckets are cached into main memory to speed up sequential access (one of the frequently used operations). Thus, when a request is made

for an object, the storage manager first checks for the object in main memory, and only when it is not found, it is accessed from the disk to retrieve it.

- multiple copies - an object may belong to more than one class. To speed up sequential access, multiple identical copies of an object are stored. Though it slows down update process, it speeds up sequential access.

- physical clustering - this is the most important optimization technique used for speedup in the system. It attempts to locate objects which are logically clustered in the same class near the same physical location on the disk.

This improved version of CSUSSM is designed to optimize two frequently performed operations. *Sequential access*, for retrieving all the objects within a particular class. *Taxonomic subsumption*, for determining whether object B is below object A in the class taxonomy. Sequential access is used for classification during query processing and taxonomic subsumption is used in determining object type. Other operations include single-object retrieval used in browsing and object induction.

3.2.1 Object-Oriented Virtual Memory Management

The secondary storage manager is based on a physical clustering algorithm which attempts to locate objects which are logically in the same class near the same physical location on disk. The secondary storage manager is an object-oriented virtual memory manager (OOVMM). Objects are cached in main memory, and when a request is made for an object that is not in main memory, an *object-fault* occurs and the disk is accessed to retrieve that object.

Three levels of memory are identified; main memory, memory cache, and disk. The main memory contains the application program and is the source and destination of objects generated or stored. Memory cache is main memory set aside for

buffering objects in random access memory (RAM). Under DOS, this is implemented in expanded memory. The disk is the secondary storage manager.

The data on disk are contained in one or more object files. A file ID identifies the object file which contains the desired object. A file index translates file ids to physical file names. Each object file has a root which is the name of a database class. All objects at or below that class in the taxonomy are stored in that object file.

The object file is composed of fixed-size buckets. Bucket numbers are used to identify each bucket. A bucket is the unit of transfer between disk and main memory. A bucket can contain a number of relocatable objects and according to the physical clustering algorithm, all these objects are members of the same class. When the bucket containing the desired object is loaded into main memory, many of its neighboring objects are also loaded, thus speeding sequential access. Here the probability that the next object required is already in the main memory is very high. A bucket buffer is used to cache buckets into main memory. This bucket buffer consists of recently used buckets. When this buffer is full and a new bucket is to be brought into memory, LRU (Least Recently Used) policy is used to accommodate the new bucket.

Another concept incorporated into the improved version of CSUSSM is the multiple identical copies. CANDIDE supports multiple inheritance, in which an object can belong to more than one class. To speed up sequential access multiple identical copies of an object are stored. One copy of the object is stored for each of its parent classes. Though this speeds up sequential access it has the disadvantage of slowing updating, since all copies of the object must be updated. This trade off is made to speed up query processing and assuming that there will be only few updates.

3.2.2 Physical Clustering Algorithm

Database accesses in an OODBS include relation-like scans of sets or collection of objects, and navigation-like access among related objects. Such inter object references would lead to random disk I/O if the objects are in different buckets. Physical clustering algorithm attempts to reduce the I/O overhead by storing the related objects in the same unit of storage, a bucket.

Physical clustering should also be based on the most frequently performed operations. In CANDIDE the most frequently performed operations are sequential access and taxonomical subsumption [Beck89]. Thus according to the physical clustering algorithm, objects of the same class are located near the same physical location on disk. Physical clustering is done on two occasions.

1. when the database has not yet been physically clustered and
2. when updating an already physically clustered database

A resursive function *pcluster(oid)* is used. This function physically clusters the subtree rooted at class oid. Intially pcluster is called with the root of the taxonomy. There are two values involved in the clustering process:

1. β - maximum size that can fit in a bucket
2. α - minimum size required to create a bucket

The algorithm is given below

pcluster(oid)

{

Get size of subtree rooted at oid.

while(size $> \beta$)

{

Create a list containing the subtree rooted at each subclass of oid.

If the size of one of these subtrees is $> \beta$, call pcluster(soid) where soid is the subclass of oid which is the root of this subtree.

Else keep removing the smallest subtrees until the combined size of the remaining subtrees is between α and β. Then store the remaining subtrees in a bucket.

}

If oid is the root of the taxonomy, then store the remainingsubtree rooted at oid in a bucket and return.

Else return without storing (remaining subtree passes to next higher class).

}

It is desired that the stratergies for updating an existing database which has already been physically clustered should be optimum. Reclustering may be needed when an object is created, deleted or modified. In that case the following strtergies are adapted

- Recluster a subtree containing the updated object plus all of its ancestors
- Recluster the taxonomy only in the immediate neighborhood of the updated object

CHAPTER 4
DESIGN OF MULTIUSER CANDIDE

Chapter 2 discusses the general requirements, issues and concepts involved in developing a multiuser database system. This chapter describes the design of CANDIDE multiuser secondary storage for the Version I. The concepts discussed in Chapter 2 are used. First, the architecture of CANDIDE multiuser secondary storage manager is described. Next transaction processing along with client and server operations are described. Then the design of concurrency control mechanism for CANDIDE is described in detail. Finally a multiuser design for Version II is discussed.

4.1 Multiuser Architecture for CANDIDE Secondary Storage Manager

Multiuser CANDIDE secondary storage manager will have a client/server architecture, the configuration of which is shown in Figure 4.1. This architecture can be broadly categorized under RU [Rou91] architecture in the sense that the object manipulation and query processing are done in the client and low level operations like locking are done in the server.

The client makes a request to the server by sending it a message that contains the information necessary to satisfy the request. The server, a dedicated server, responds to the request by adding information to the message and returning the message to the client. The server provides a variety of services to multiple clients, which include I/O, transaction management and concurrency control.

Here the application program which resides on the client machine links with the client module library. The interface client library provides routines for accessing the Database interface functions. Initialization and transaction support routines are also

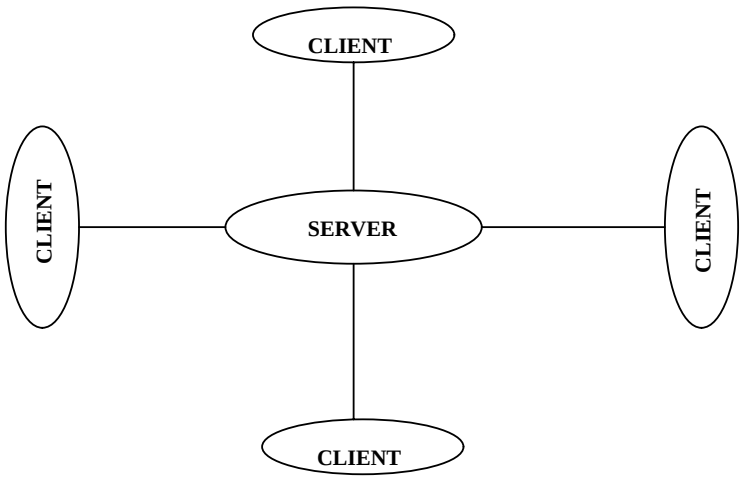

Figure 4.1. Client/Server Configuration

included. Thus the application program does not access the server directly, but it calls the interface routines of the client module. The client module communicates with the server as necessary.

The client-server connection is established at the start of a client application. Since an application can have more than one transaction it is advantageous to establish the connection at the start of the client application rather than at the beginning of each transaction. Thus we would incur the connection establishment cost only once for an application.

4.2 Client and Server Operations

A client begins by performing some initialization and then starts a transaction. During the scope of the transaction, various files and objects can be accessed and modified. The client declares the start of transaction with a Begin_Work() and end with a Commit_Work() or Abort_Work(). All operations performed by the program between Begin_Work() and Commit_Work()/Abort_Work() will be part of this transaction. Thus a flat transaction model is chosen which supports these functions.

Nested transactions and multi-level transaction models which provide more concurrency are expected to be supported by later versions of the CANDIDE multiuser secondary storage manager.

Begin_Work(): The client requests a transaction ID (TID) from the server. All future data and lock requests sent to the server in the scope of the transaction will contain this TID. After a client has started a transaction, it can begin accessing objects and files. An application requests an object along with the desired lock mode for the object. The server will either send the desired object to the client if there is no lock conflict, or make the client wait on that lock for that object. Every object modified during a transaction's life span is flagged, to be able to locate it at the termination of a transaction.

Commit_Work(): At 'normal' termination of a transaction or an explicit 'commit transaction' request by an application, a transaction is committed. At the commit of a transaction all dirty objects are sent back to the server and a commit is requested. Also, all the locks held by the transaction are released and a check is done to find out if other transactions are waiting for the locks on objects held by the transaction, and if so they are released depending on the lock compatibility.

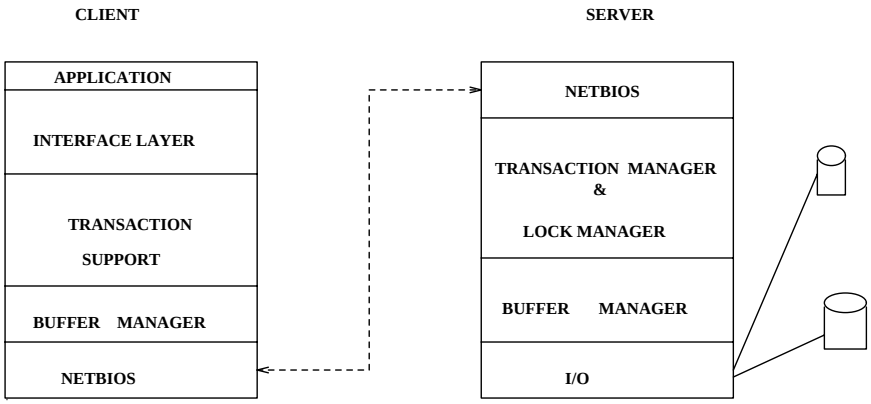

Figure 4.2. CANDIDE Multiuser Architecture

Abort_Work(): An 'abnormal' termination of a transaction or an explicit 'abort transaction' request by a user application results in the transaction being aborted. All modifications of an object done by that transaction are not sent back to the server for storage, instead they are discarded. All locks held by the aborting transaction are released. The transactions waiting on the locks held by the aborting transaction are processed to see if they can be freed. The waiting transactions are freed based on the compatibility of locks held by the previously freed transactions. The database is brought back to the state in which it was at the start of this aborting transaction.

To achieve multiuser capability, an efficient concurrency control mechanism has to be provided. A concurrency control mechanism suitable for CANDIDE data model is designed and described in the next section in detail.

4.3 Concurrency Control for Object-Oriented Databases

To the best of our knowledge, there are only two systems, ORION and O2, which have adapted locking schemes and exploited the semantics of the data model to provide concurrency control. So, we discuss briefly the approaches adapted in these two systems and then explain our approach for CANDIDE. This section comprises of the following subsections:

1. Concurrency Control in O2
2. Concurrency Control in ORION
3. Concurrency Control in CANDIDE

In the first subsection the locking scheme in O2 is discussed briefly. In the second subsection, the concurrency control in ORION is explained. The final subsection is organized as follows: first various operations performed on the CANDIDE are highlighted; why the approach proposed in ORION and O2 is not suitable for our

DBMS is discussed next; and finally the modifications and extensions to the ORION approach to make it suitable for CANDIDE are discussed.

4.3.1 Concurrency Control in O2

O2 has two concurrency control mechanisms: one at the object level based on the read-write semantics and one at the schema level based on the schema information. An O2 transaction maps directly to a WiSS (Wisconsin Storage System) transaction. And, concurrency on O2 objects is handled by WiSS on the server by a two-phase locking algorithm on pages and files.

Concurrency on the schema is handled differently from concurrency on objects. Semantic information such as compatibility of methods, independence of objects are taken into consideration to allow increased parallelism. This approach identifies a hierarchy of abstraction levels and provides parallelism at each level. The bottom level is the page level (physical level) and the intermediate level is the representation level and the top level is the O2 object level. O2 objects are usable through methods (O2 object level) and these in turn invoke operations (representation level), which requires pages from the bottom level.

The approach proposed for O2 provides concurrency control at each of the above mentioned abstraction levels. Compatibility of methods is identified based on the *real access* (access to object representation) and *virtual access* (other objects related to the real access object which do not require access to their representation). Access to the classes and instances are controlled by locks. Granularity locking is adapted for this. [Car90] do not discuss lock conversion.

Finally [Car90] discuss the impact of one-level transaction model and multi-level transaction model and concludes that the multi-level transaction model is best suited to their system.

4.3.2 Concurrency Control in ORION

ORION applications require locking on three types of hierarchy: i) granularity hierarchy for logical entities, devised to minimize the number of locks to be set, ii) the class-lattice, and iii) composite object hierarchy. ORION extends the theory of locking for the granularity hierarchy to satisfy its locking requirements. The lock modes and the DAG protocol are summarized below.

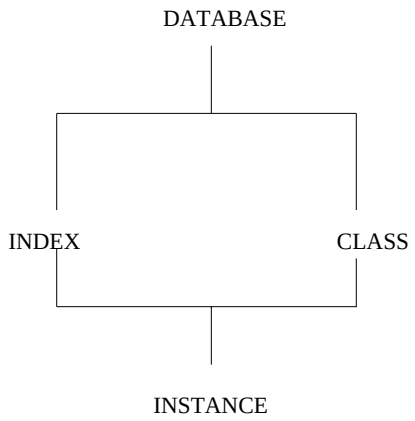

Figure 4.3. Hierarchy of Lock Granules in ORION.

Figure 4.3 shows the unit of locking. Instances are locked only in S or X mode, indicating whether they are to be read or updated respectively. Class objects may be locked in any of the five modes.

IS : on Class means that instances of the class are to be explicitly locked in S mode as necessary.

IX : on Class means that instances of the class will be explicitly locked in S or X mode as necessary.

SIX : on Class means that class definition is locked in S mode, and all instances are implicitly locked in S mode and instances to be updated (by transaction holding the SIX lock) will be explicitly locked in X mode.

S : on

- Class means that class definition is locked in S mode, and all instances of the class are implicitly locked in S mode - no update is allowed.
- Instances means read lock on that instance

X : on

- Class means that class definition and all instances of the class may be read or updated.
- Instances means write lock on that instance.

Standard DAG lock protocol:

- To set an explicit S lock on a lockable granule, first set an IS lock on all direct ancestors, along ANY ONE ancestor chain, of the lockable granule on the DAG.
- to set an explicit X lock on a lockable granule, first set an IX or SIX lock on all direct ancestors, along each ancestor chain, of the lockable granule on the DAG.
- set all locks in root-to-leaf order.
- release all logical locks in any order at the end of the transaction, or in leaf-to-root order before the end of a transaction.

Figure 4.4, gives the compatibility matrix for the granularity locking modes. Two lock requests for the same node by two different transactions are compatible if they can be granted concurrently.

[Gar88] describes two locking protocols for locking a class lattice. The first protocol simply requires the system to set explicit locks on all subclasses on a class lattice rooted at the class to be accessed. For example, if the definition of a class is to be modified, the system sets update locks on the class and all its subclasses. Further

REQUESTED MODE

CURRENT MODE	IS	IX	S	SIX	X
IS	yes	yes	yes	yes	no
IX	yes	yes	no	no	no
S	yes	no	yes	no	no
SIX	yes	no	no	no	no
X	no	no	no	no	no

yes - compatible no - not compatible

Figure 4.4. Compatibility Matrix

more, if a class lattice rooted at a particular class is to be accessed for query processing, the system sets a read lock on every class in the class lattice rooted at the class. This protocol works well if the class to be accessed is near the leaf of a class lattice, since it requires explicit locks on the class and all its subclasses, but incurs a high lock overhead if the class is near the root level of a deep class lattice.

The second protocol which is efficient for accessing a class near the root of a class lattice, introduces two lock modes, R lock (read-lattice lock) and W lock (write-lattice lock).

R: on a class means an explicit S lock on that class and implicit S lock on the subclasses of that class.

W: on a class means an explicit X lock on that class and implicit X lock on all the subclasses of that class.

The definition of the X lock is modified to

X: on class allows updates to instances of the class and it allows the definition of the class to be read but not updated.

Since R and W cause implicit locking of subclasses, its superclasses are locked in intention modes IR and IW respectively. Further, all lock modes discussed earlier (IS, IX, SIX, X, S), require the IR and IW mode for their superclasses of the class being locked. This is illustrated with an example. The notations used in the examples are as follows: C_A represents Class A, C_B represents Class B and so on. I_1A represents Instance 1 of class A, I_2B represents Instance 2 of class B, and so on. T1 represents transaction 1, T2 represents transaction 2, etc. In Figure 4.5, to read instance I_1D, IR lock is set on C_C and C_A and IS lock on C_D and an S lock on the instance I_1D.

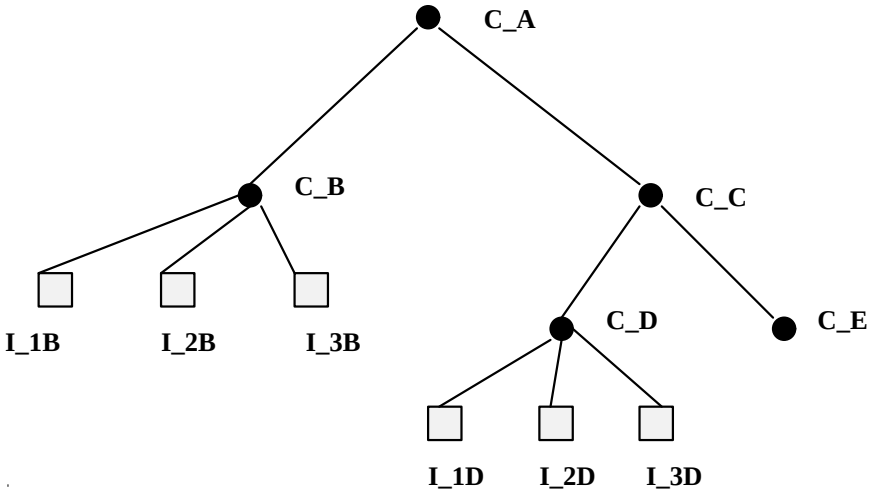

Figure 4.5. Example 1

This protocol fails on a class-lattice, in which a class may have more than one superclass. For example, in Figure 4.6, if T1 reading C_B sets an IR lock on C_A and an R lock on C_B, implicitly locking C_C, C_D and C_E in the R mode. Now, if T2 updating C_G, sets an IW lock on C_F and W lock on C_G, implicitly locking

C_D, C_E and C_H in W mode, giving rise to a read-write conflict on C_D - implicit conflict, which is not detected using this protocol.

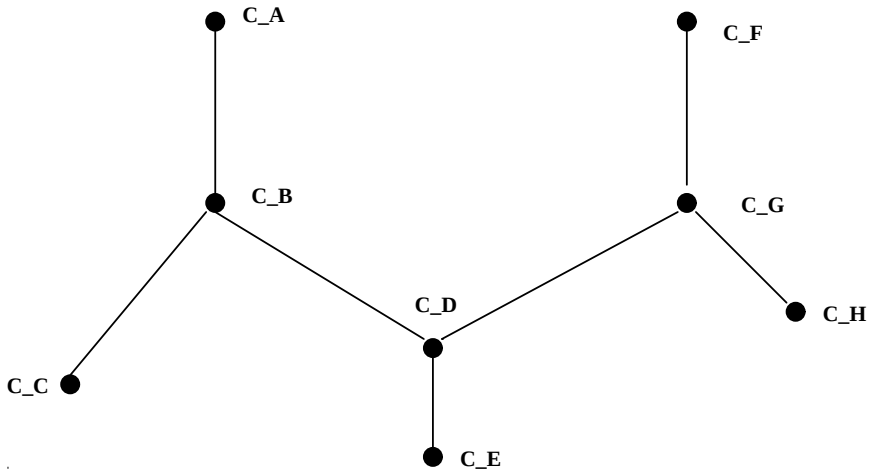

Figure 4.6. Class Lattice Example

The solution [Gar88] proposes is, to set explicit R or W lock on all subclasses with more than one superclass of a class acquiring R or W lock. Thus the second protocol is - to acquire a R or W lock on a class, lock the superclasses along any one chain in the intentional mode and all subclasses with more than one superclass in the R or W mode explicitly. In addition to these lock modes for classes and instances, ORION introduces new lock modes (ISO, IXO, SIXO) for locking composite objects. The protocol treats composite object as a lockable granule.

4.3.3 Concurrency Control Protocol for CANDIDE

CANDIDE implements an object-oriented data model, since the application can be represented in a taxonomy. The taxonomy has a lattice structure for the class taxonomy. Various operations performed on CANDIDE are

- read an instance of a class
- access all/few instances of a class
- add/delete/modify instances of a class(es)

- add/delete/modify class(es)
- taxonomic surgery - moving the position of a class in the taxonomy
- sequential class traversal
- modify the schema - updating the attributes of a class
- classification - finding the position of an object in the class taxonomy using taxonomic subsumption

All these operations can be explicitly done by the user or can occur as a result of classification. Most frequently performed operations are sequential class traversal and taxonomic subsumption.

The O2 approach cannot be adapted for CANDIDE because, CANDIDE is not based on methods. Orion's approach can be adapted, but with modifications to increase concurrency. The approach used in ORION does not exploit the independence of operations on instances and class definitions. The following example illustrates this using Figure 4.5:

T1: reads class C_B.

T2: updates instance I_1B.

According to the protocol proposed in ORION, T1 acquires R lock on C_B and IR lock on C_A. T2 needs IX lock on C_B and IW lock on C_A. Here, R and IX are incompatible. Updates on instances of a class and reading the class definition of that class are two independent operations and should be allowed to occur in parallel. Moreover, classification function needs to read class definitions often. So, another transaction updating instances of a class should not stop the classification process.

To allow this, the definition of R and W locks are modified. This is done to seperate the class and instance operations. The modified definitions are:

R : on a class means that an explicit S lock on the *definition* of that class and implicit S lock on the *definition* of all the subclasses of that class.

W : on a class means that an explicit X lock on the *definition* of that class and implicit X lock on the *definition* of all the subclasses of that class.

IR : on a class means that, one of its subclasses' definition is locked in the R mode.

IW : on a class means that, one of its subclasses' definition is locked in the W mode.

The Intention modes of R and W locks are required to ensure that when a class and its instances are being accessed, the definition of the class's superclass (and their superclasses) are not modified. Thus when you are trying to get R/W on the definition of a class, you need to get IR/IW locks on its superclasses.

The definition of IS, S, IX, X and SIX lock modes are carried over for the instance operations. The definition of these modes are stated again:

IS : on Class means that instances of the class are to be explicitly locked in S mode as necessary.

IX : on Class means that instances of the class are to be explicitly locked in S or X mode as necessary.

SIX : on Class means that class is locked in S mode, and all instances are implicitly locked in S mode and instances to be updated (by transaction holding the SIX lock) will be explicitly locked in X mode.

S : on Class means that that all instances of that class are implicitly locked in S mode - no update is allowed.

on Instances means read lock on that instance

X : on Class means that all instances of the class are implicitly locked in the X mode.

on Instances means write lock on that instance.

Note, that the IS, IX, S, X and SIX locks on the classes must not be confused with the IR, IW, R and W locks on the classes, since they have a different meaning. IR, IW, R and W locks control the access to the representation (definition) of class, while IS, IX, S, X and SIX control access to the instances of the class. Figure 4.7 gives the lock compatibility matrix for the class definition locks. The W mode is not compatible with any of the IS, IX, S, X and SIX modes. Thus the operations on instances and operations on classes are separated.

REQUESTED MODE

CURRENT MODE	IR	R	IW	W
IR	yes	yes	yes	no
R	yes	yes	no	no
IW	yes	no	yes	no
W	no	no	no	no

Figure 4.7. Compatibility Matrix For Locks On Class Definitions

It is seen from the lock compatibility matrix that the concurrency has been increased by the introduction of IR, IW, R and W locks and redefining them. The granular hierarchy for CANDIDE is shown in Figure 4.8.

The protocol is as follows:

- To set an explicit S/R lock on an instance/class, first set an IS/IR lock on all direct ancestors along each ancestor chain.

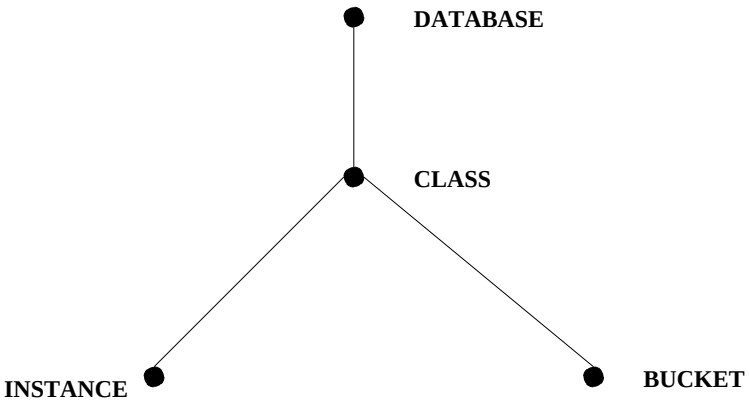

Figure 4.8. Hierarchy of Lock Granules in CANDIDE.

- to set an explicit X/W lock on an instance/class, first set an (IX or SIX)/IW lock on all direct ancestors, along each ancestor chain.
- locks can be set in any order, since there will be only one task which will be accessing the lock table at any given point of time.
- release all logical locks in any order at the end of the transaction.

The following illustrates the use of all these locks. Referring to Figure 4.5, Suppose:

1. If, T1 wants to update I_1B, and T2 wants to read I_2B, then T1 gets IX locks on C_A and C_B and X lock on I_1B, and updates I_1B, while T2 gets IS lock on C_A and C_B (IS is compatible with IX) and S lock of I_2B, and reads I_2B.
2. If, T1 wants to read all instances of class C_D, then T1 gets IS locks on C_A, C_C, and S lock on C_D. This S lock on C_D implicitly locks all instances of C_D in S mode, thus reducing the number of locks.

Concurrency control using these lock modes for CANDIDE class lattice can be achieved by the following protocols.

1. Explicit Locking Protocol

2. Subclass Locking Protocol

3. Run Through Protocol

Explicit Locking Protocol

This is the the simple protocol of ORION, where all the subclasses are explicitly locked. As discussed earlier the locking overhead is very high.

Subclass Locking Protocol

The original protocol is modified such that explicit R or W lock is set on all subclasses of a class acquiring R or W lock which have more than one superclass (multiple parent subclass). This is the class-lattice protocol discussed in Garza and Kim [Gar88].
The following illustrates these protocol

1. To change the definition of C_B
   (a) lock C_B in W mode and all C_B's superclasses in IW mode
   (b) lock each subclass of C_B which has *more than one superclass* in W mode (class C_D)
2. To read the class C_G
   (a) lock C_G in R mode and its superclass C_F in IR mode
   (b) lock each subclass of C_G which has *more than one superclass* in R mode (class C_D)

Advantages: This protocol supports partial implicit locking. Thus the number of locks required to access a set of objects is reduced considerably.

Disadvantages: The existing system maintains the ancestor list and if we need to maintain a list of multiple parent subclass, then the bookkeeping needed to maintain this information for each class would prove very expensive.

Problems faced by this protocol while building the lock table are discussed in Chapter 5.

Runthrough Protocol

We propose a third way of handling the class lattice structure. This protocol postpones the implicit conflict detection till the multiple parent subclass is *actually* accessed. The first transaction *actually* accessing the multiple parent subclass is granted the desired lock mode, while all other transactions involved in the implicit conflict will detect the conflict when they try to *actually* access the multiple parent subclass. While acquiring the lock on an object when it is actually accessed, the whole ancestor list need not be checked. The waiting transactions are blocked or aborted depending on the situation. The following highlights this situation:
Referring to Figure 4.6, suppose T1 reads class C_B and T2 writes class C_G. T1: gets IR lock on C_A and R lock on C_B. R lock on C_B implies implicit R lock on C_C, C_D and C_E,and T2: gets IW lock on C_F and W lock on C_G. W lock on C_G implies implicit W lock on C_D, C_E and C_H.

The R/W conflict at C_D is not detected since there is no lock on multiple parent subclasses. Both the transactions are granted their desired lock modes. The conflict is detected while *actually* accessing class C_D. There are two cases:

Case 1: T1 accesses C_D first.

R lock on C_D is granted to T1. Now if T2 tries to actually access C_D there is a conflict and T2 waits on C_D. Here it is not necessary to abort T2 because T1 is only reading and this does not change the taxonomy.

Case 2: T2 accesses C_D first.

W lock on C_D is granted to T2. Now if T1 tries to actually access C_D there is a conflict and T1 waits on C_D. Most of the times T1 can be allowed to continue after T2 commits. But when there is a taxonomic surgery and if C_D has to move , then it is no longer a subclass of C_B. So, T2 has to delete C_D from C_B's subclass list. To do this T2 needs to acquire a W lock on C_B. At this point, T2 waits for an object held by T1 and vice versa. Thus a deadlock situation is reached and one of the transactions is aborted.

The disadvantage of this protocol is that there is a possibility for a transaction to proceed down the taxonomy and still be aborted. The complications of building a lock table for this is discussed in Chapter 5. Here, the intention locks need to be acquired along each ancestor chain.

Thus CANDIDE's concurrency control mechanism is aimed at providing maximum concurrency, by exploiting the semantics of the data model. Implicit locking supported by this mechanism reduces the number of locks required while accessing a set of objects.

Lock Conversion

In the previous section we described the various lock modes and locking protocols that satisfy the locking requirements for CANDIDE applications. But the issue of lock conversion was not addressed. In this section we discuss lock conversion for locks for both class (IR, R, IW,W) operations and instance operations (IS, IX, R, SIX, X).

A transaction sometimes needs to convert a lock it currently holds to a more exclusive mode. This is done to increase concurrency and reduce the amount of bookkeeping in the lock table. For example, suppose a transaction classifies an object into the taxonomy, it first reads the class definitions to find the exact location of the object and then inserts it. It is reasonable for the transaction to set R locks while reading the classes and then request W lock when it actually inserts the object

into the taxonomy. Instead, if the transaction sets a W lock at the beginning of the classification operation, it will lock objects for a long time, thus blocking many transactions.

Figure 4.9 and Figure 4.10 provides lock conversion matrix for the class operation locks and instance operation locks respectively. Since the operations on classes and instances require seperate locks, a transaction can hold two locks on a class, one for class operation and one for instance operation. Thus, if a transaction holding a W lock requests a S lock on one of its instances, it is granted.

CURRENT MODE \ REQUESTED MODE	IR	R	IW	W
IR	IR	R	IW	W
R	R	R	W	W
IW	IW	W	IW	W
W	W	W	W	W

Figure 4.9. Lock Conversion Matrix for Class Operation Locks

4.4 Preliminary Thoughts on the Design for CANDIDE Version II

The previous discussion dealt with the design of locks for multiuser secondary storage manager for the Version I of CANDIDE. This section deals with the preliminary thoughts on the design for multiuser secondary storage manager for Version II.

REQUESTED MODE

CURRENT MODE	IS	IX	S	SIX	X
IS	IS	IX	S	SIX	X
IX	IX	IX	SIX	SIX	X
S	S	SIX	S	SIX	X
SIX	SIX	SIX	SIX	SIX	X
X	X	X	X	X	X

Figure 4.10. Lock Conversion Matrix for Instance Operation Locks

Architecture

Version I is based on object-server architecture [Ban91], where objects are sent to a client on demand basis, i.e., one object at a time. So each time a client requests an object, the lock table is checked for compatibility and then the object is sent to the client. This architecture simplifies the implementation of concurrency control mechanisms as it is completely centralized in the server. Furthermore, the implementation of object-level [OID] locking is straight forward, as locks are got on the OID and there is no need for bucket level locking. This design suffers from several disadvantages. First, in the worst case there may be one server access per object reference, thus increasing the communication cost. Second, the architecture of the server becomes complicated and as a result of which the load can become high.

In an environment where many objects must be frequently accessed, efficiency becomes a principle design criterion. One approach to improving performance is clustering related objects in a bucket. Thus a bucket has logically related set of objects which the client is expected to access during a transaction. Now the bucket is

chosen as the unit of transfer for objects between client and server and from secondary storage to main memory. This is similar to page-server architecture [Ban91]. Greater system performance results from preloading required objects, but complicates concurrency control mechanisms. Object-level locking may be difficult to implement, and problems occur when two clients update two different objects on the same bucket. The bottom line is that the performance of this design would depend on the effectiveness of the clustering mechanism.

Lock and Bucket Interaction

When a client requests an object in a particular mode from the server, the server checks the lock table for its compatibility, and if it is compatible then the bucket containing that object is locked in the requested mode and the whole bucket is sent to the client. The bucket can be locked either in the S or X mode. Before acquiring a lock on a particular bucket, it should be made sure that no other transaction holds a lock in a conflicting mode on that bucket. If the bucket is locked in the conflicting mode then the transaction waits on the object.

In Version II the client can also maintain the lock mode details of the object it is accessing. This information is useful while accessing other objects in a bucket the client already has. This would reduce the number of times the client has to consult the server to access an object.

Situations where this method is advantageous: Suppose a transaction is reading a class A, it holds an R lock on this class A. Acquiring R lock on a class implicitly locks all its subclasses in the R mode according to the semantics of R mode. Now if the transaction wants to access the subclasses which are in the bucket already in the client, then the client need not go to the server. The probability that this situation would arise often is high due to the following reasons:

- class traversal access is one of the most frequently performed operations in CANDIDE
- physical clustering algorithm [Bec93] places a class and its subclasses physically together

The locks on buckets are released at the end of the transaction. This might make objects not used by this transaction unavailable to others untill it commits.

Situations where this method is disadvantageous: Suppose a transaction needs to access a single object (single object retrieval), still the whole bucket containing that object is sent to the client. Other objects in that bucket are unnecessarily locked. Some improvement can be achieved by reducing the size of buckets. An alternative is to release the locks on the buckets as soon as an object is read or written. This would lead to complication in physical clustering and maintaining the consistency of data.

Physical Clustering and Locks

Physical clustering is done at commit stage of a transaction. The strategy for updating an existing database which has already been physically clustered is optimized so as to affect minimum number of buckets.

Reclustering is needed when an object is created, deleted or modified. First, the system tries to insert the object into the same bucket. If it fits then no other bucket is affected. The transaction is committed and the client is informed of the commit. If the object does not fit into that bucket, then the clustering proceeds to the next step. The next step is to combine neighboring buckets and then try to fit in all the objects. Now if the bucket required for physical clustering is used by another transaction, then the clustering algorithm doesn't proceed, it creates a temporary bucket and stores the modification of that transaction. Thus the clustering is deferred till there

is no traffic on that project file. Since the database is divided into projects, each project can be clustered independently. Thus an *optimistic method* is chosen while reclustering. The main disadvantage is that if it is found that a bucket required for clustering is in use after moving high up in the taxonomy then lot of work done goes to waste.

4.5 Deadlock Handling

Locking schemes are prone to deadlocks. And, the locking scheme proposed for CANDIDE is no exception. This section briefly discusses the suitability of various deadlock handling schemes for CANDIDE.

There are two principle methods for handling deadlocks. One is *deadlock prevention* and the other is *deadlock detection and recovery.* As the name indicates, deadlock prevention scheme ensures that the system will never enter the deadlock state. This is commonly used if the probability of the system getting into a deadlock state is high. Otherwise the deadlock detection and recovery approach is used, which detects the deadlock state and then resolves it.

The prevention approach either causes many objects to be locked for a long time or causes some unnecessary rollback of transactions [Hen91]. This approach would decrease the concurrency in CANDIDE. The alternative is to adapt the deadlock detection and resolution scheme. There are two ways to detect deadlocks, one is by using timeouts and the other using the wait-for-graph [Hen91]. In the timeout approach, each lock request is given a time limit for lock wait, and, if the request times out, that transaction is aborted. This is a very simple and inefficient way of handling deadlocks.

A more sophisticated way, is to use wait-for-graph. Once the deadlock state and the transactions involved in deadlock are detected using the wait-for-graph, the transaction(s) to be aborted (victim) can be chosen based on the semantics and cost

of the transactions involved in deadlock. Some of the factors which determine the cost of an abort are, how many objects the transaction has used and how many more it need to complete, how long the transaction has computed and how much longer the transaction will compute and how many transactions will be involed in the abort. Apart form these factors, the semantics of the read and write operations on the class objects can be used to choose the victim. A write operation on a class can result in change in class definition or in taxonomic surgery. If such an operation is involed in the deadlock then, it would be better to allow the write operation to proceed and restart the transaction with read operations. This is suggested because, it would be expensive to abort and restart these operations. The disadvantage of this scheme is that, a read transaction could always be picked as the victim and as a result, this transaction never gets to complete its designated task. This situation is called *starvation*. This can be avoided by including the number of times a transaction has been restarted as a factor for selecting the victim. Future work can be done to design an efficient protocol for handling deadlocks in CANDIDE.

CHAPTER 5
IMPLEMENTATION OF MULTIUSER CANDIDE DBMS

This chapter discusses the implementation of concurrency control mechanism using the design presented in the previous chapter. The first section discusses the choice of a Network Operating System and the network protocol for implementing this client/server system. The second section discusses the issues relating to providing connectionless or connection oriented communication. The third section discusses extending CANDIDE to other platforms. Finally in the fourth section the data structures and algorithms required for implementing the multiuser system is discussed in detail.

5.1 Lantastic VS NetWare

Chapter 4 discusses both LANtastic and NetWare Network Operating Systems. This section discusses the pros and cons of these NOS and the reasons for choosing LANtastic to implement this multiuser DBMS.

Novell's NetWare has a client/server architecture, and the security system provided by it is much better than LANtastic. Since LANtastic is a peer-to-peer local area network, all its resources (hard drive, printers, etc) can be shared over the network. Security provided by LANtastic is with respect to the machine from which a user logs in to the server. So, if a user is denied access to certain files on the server from his machine, he still can gain access to those files by directly working on the server, rather than trying to access it across the network. Since CANDIDE multi-user system has a client/server model and requires maximum possible security, NetWare would be preferred.

One important feature in NetWare is its TTS (Transaction Tracking System). The features of TTS are what we are trying to implement in the CANDIDE multi-user system. If we use Novell we can use this feature to keep track of all the transactions, thus saving development time. However, it is better to avoid writing code that uses facilities specific to a particular LAN vendor for portability reasons. Even though TTS functtionality meets the design rrequirements of our system, to run our system on non-Novell LANs, we have to provide our own substitute for TTS. In that case we should implement the system at a lower layer.

NetBIOS is compatible with most of the LANs. LANtastic is designed around the NetBIOS standard used by many networks. Also, it is able to work with a wide variety of network hardware and software. By using this standard for network communication, LANtastic allows use of other manufacturer's adapter boards. LANtastic allows third party network utilities and programs to be used on it. This will be an added advantage of implementing on LANtastic, as our stand-alone system already has its own buffer management policies.

As for NetBIOS on NetWare LANs, it is possible to run an application written using NetBIOS on Netware. NetBIOS services can be provided on NetWare, either in the form of Novell emulator or a device driver. This would take up extra memory on a Novell NetWare workstation, and IPX and SPX communications can still be used.

Both networks provide reasonable performance, though Novell has an edge. Thus it would be much easier to port if we use NetBIOS. Also it would be much easier to tune our system to meet to our requirements and have full control (coding) of our system.

5.2 Datagrams Vs Sessions

Irrespective of which NOS is chosen, a decision has to be made as to which mode of communication is suitable – connectionless or connection oriented. This section discusses the pros and cons of these two modes.

Connection-oriented communication protocol (session service) is used to establish connection with the server. An initial setup phase is used to setup a fixed route for all packets exchanged during a session between users. Packets use relatively short headers, as compared to datagram service. Enhancements, such as error control guaranteed delivery, and sequencing of packets are provided by the session services. In datagram services, enhancements of basic service must be provided. Datagram service does not guarantee delivery of packets.

In a client-server DBMS, delivery of data should be guaranteed, thus session service is chosen to implement the system, though it has an initial setup time. Also, if the session is going to last for a long time, which is usually the case, it is more advantageous to use session service.

Another reason for choosing the session service is that, up to 65,535 bytes long message record can be sent and received during a session, whereas datagrams can carry only 512 bytes long message records. Since the size of a bucket (unit of transfer) is more than 512 bytes, session service will be more suitable and efficient than datagram service.

5.3 Extending CANDIDE to other platforms

As discussed earlier TCP/IP is the standard cross-platform protocol. Using this protocol our database can be accessed over the Internet. In this case, we can have a SUN Unix machine as our server. Unix OS provides multi-programming and multi-threading capabilities and the IPCs are much powerful and efficient than NetBIOS

and IPX/SPX. More disk space and memory are available. The cross-platform capability of TCP/IP allows any machine to be the client and in our case, we can have PCs as clients. The Server does all the low-level DBMS functions like maintaining lock table, I/O, etc, so it would be efficient to use a powerful machine as the server. One disadvantage of using Unix is its complexity and the cost of maintaining the whole system. Recently a standard protocol to support NetBIOS services in a TCP/IP environment [RFC] has been proposed. This protocol can be used to access CANDIDE database on the Internet.

5.4 Implementation of the Multiuser System

In this section, implementation of the following components are discussed in detail

- Server
- Client
- Lock Table
- Buffer Management

The session service provided by the NetBIOS software is used for communication between the client and server.

5.4.1 Server:

The server is first initialized by checking if the following conditions are true:

- The DOS Version is 3.0 or above
- The machine name has been set
- NetBIOS is active

Then NetBIOS is asked to add its name (SERVER) to the local name table. If the SERVER does not encounter any problem, it issues a NetBIOS *listen* command [Appendix B]. This command specifies

- Listen for a call to SERVER
- The call can be from any client
- The Post routine [Appendix B] background_listen is called
- A time-out value for the *send* and *receive* [Appendix B] commands issued in the upcoming session

The server maintains a queue of requests (called JOB_QUEUE) from various clients and processes it in the FIFO order. After a session is established between the client and the server, the job request from the client is queued and the server starts to execute the jobs from the head of the queue. Now if a *call* from another client is received, the background (asynchronous) process is triggered. A session is established and the SERVER receives the job from the client. If the request is a Commit_Work() or Abort_Work() then they are put at the head of the JOB_QUEUE. These two functions are given priority because their execution results in the release of locks on many objects. All other requests are added to the end of the queue. We have proposed FIFO order for processing of requests from the client for the sake of simplicity. But to improve the performance, the semantics of CANDIDE operations can be used to prioritize the processing of the transaction requests.

5.4.2 Client

When the application program links to the client module library, it intialiazes by checking the DOS version and making sure that NetBIOS is active, and adds its name to the local name table. Then the client establishes a session by issuing

the *call* [Appendix A] command. If the *call* command is successful, the client can begin to send its requests. When the transaction, is completed, the client requests to Commit_Work(). Once this command is successful, the client can start a new transaction or if it is the end of the application program, the session is closed.

5.4.3 Lock Table

When a request from the client is received, the server first checks the lock table for lock compatibility and if the lock modes are compatible it returns the desired object, else it is queued on the LOCK_WAIT_QUEUE.

LOCK_WAIT_QUEUE is a queue of transactions waiting to acquire a lock on an OID. The server scans the LOCK_WAIT_QUEUE to check if a transaction is waiting for a lock on an object, which was released during the Commit_Work or Abort_Work operations. If so, the lock is granted to the transaction and the transaction is inserted into the JOB_QUEUE after the Commit_Work/Abort_Work requests.

We could have used the JOB_QUEUE to queue up transactions waiting for locks on objects. Thus there would have been only one big queue. The transactions waiting for a lock could be added to the head of the queue. Thus, before processing any request, the server first checks to see if any lock could be granted to waiting transactions. This is called busy wait. This method of implementation is not efficient because it is required that a check be made for release of locks continuously. But it is sufficient that this check is made when there is an commit or abort. This could be done efficiently by using the LOCK_WAIT_QUEUE. The LOCK_WAIT_QUEUE is also used for deadlock detection.

Building of Lock Table for CANDIDE

This section discusses the step wise building of an efficient lock table for CANDIDE. First, a conventional lock table is described and the inefficiency of this table for CANDIDE is discussed. Then lock table for CANDIDE is described.

Conventional Lock Table: The lock table is a dynamic data structure (shown in Figure 5.1) which maintains lock information. of all objects accessed by active transactions in a system. Typically, it is a hash table. Each table entry has a pointer to a chain of locks, each with a name that maps to that hash bucket. Also all locks held by a transaction are linked for speedy release of locks during the commit of a transaction [Gra93b].

The lock table for CANDIDE is a hash table keyed on the OID of an object. The OID hashes to a bucket. The entry in the table is a pointer to a linked list. Typically, every object that hashes to that bucket will have a node in the linked list. The structure is shown in Figure 5.1. Thus, if two transactions are holding locks on an object(in compatible modes), then there would be two node in the linked list, one for each tranasction. As in a typical lock table structure, all objects held by a transaction are linked to speed up the lock release process at commit or abort stage.

Problems in Adapting this Structure for CANDIDE: The locking protocol of CANDIDE acquires locks not only on the object required, but also on its ancestors and in some cases on its subclasses as well. Since we have intentional lock modes on class objects, the number of locks (compatible modes on a class object is considerably high. Hence, if we use the conventional structure (Figure 5.1), there will be a node in the linked list for each lock held by a transaction on objects that hash to that bucket. This increases the search time on each bucket.

Anchored Hash Table (AHT): We propose 'Anchored Hash Table' (AHT) [Bad93], keyed on OID and type for CANDIDE. The pair (OID,type) hashses to a bucket. The entry in the bucket is a pointer to a linked list of anchor nodes. Each anchor node represents an object that hashes to that bucket. Each anchor has a linked list structure (transaction list) for transactions holding locks on that object. Figure 5.2 shows the data structure of the anchored lock table, and Figure 5.3 the

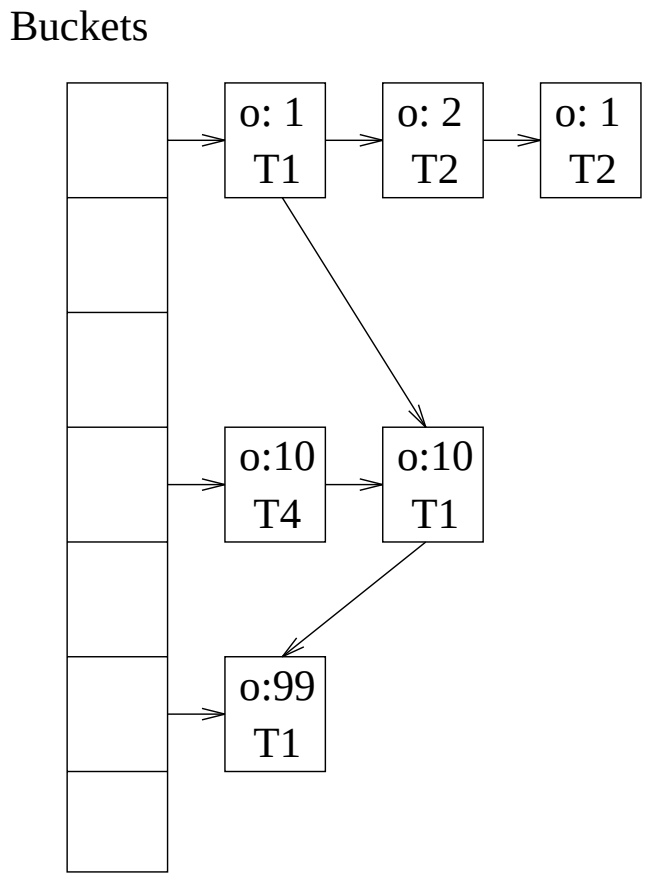

Figure 5.1. Conventional Lock Table Structure

structure of the nodes. Thus the maximum number of anchor nodes for a bucket will be the number of objects that hash into that bucket. Our aim is to build an efficient locktable.

The first time an object is accessed, we establish the ancestor class hierarchy in the lock table. This is done by having parent pointers in the anchor node which points to its parents. Thus, if another transaction needs to access an OID already in the lock table it need not build the ancestor hierarchy again. Since we have a lattice structure, we need more than one parent pointer. A linked list of parent pointers is maintained in the anchor node.

The ancestor hierarchy is built in the lock table for the following reasons:

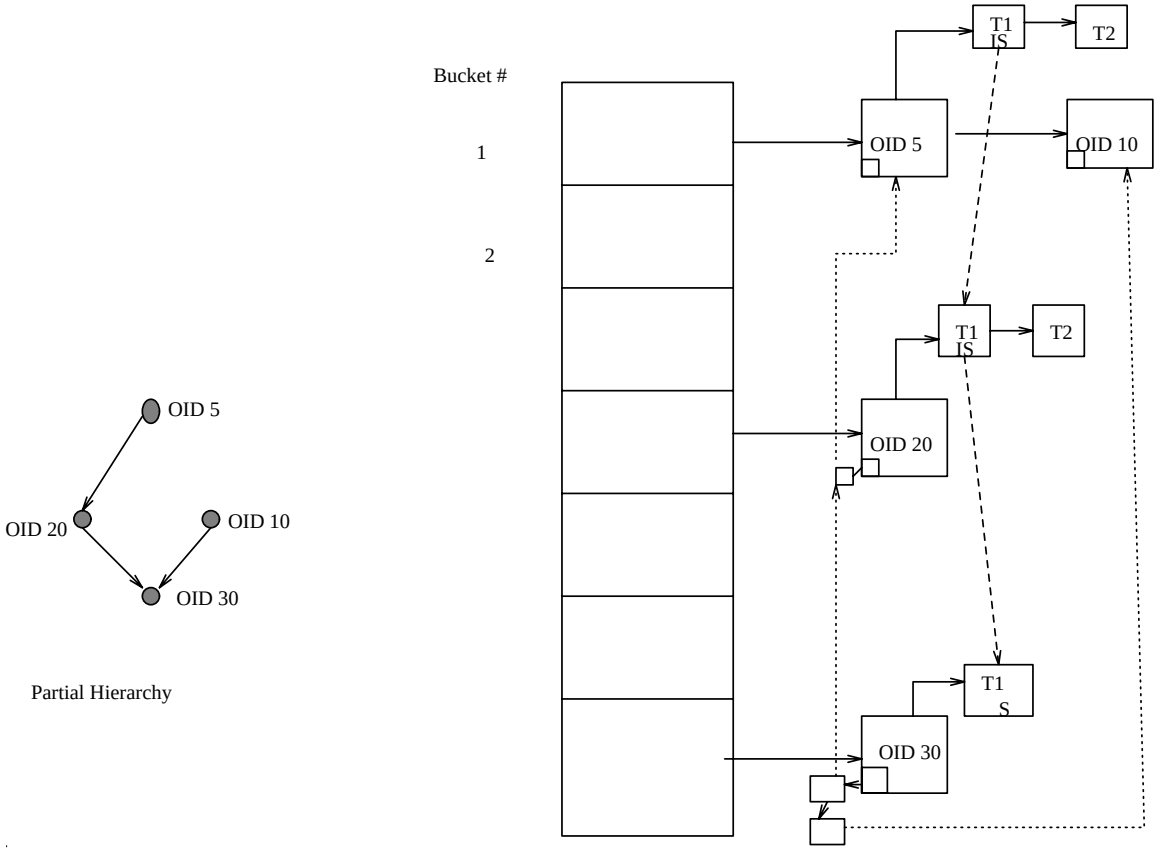

Figure 5.2. Lock Table Structure

1. Our locking protocol requires that for each lock acquired on an object, all its ancestors are locked in the intention lock mode.

2. it reduces the number of times the same hierarchy is built.

3. it reduces the number of I/Os.

4. it reduces the search time while acquiring locks.

It is guaranteed that for each OID present in the OID table, there exists its ancestor hierarchy in the lock table. The amount of time saved by building the ancestor hierarchy in the lock table is illustrated below:

Let n = number of instances accessed by a transaction.

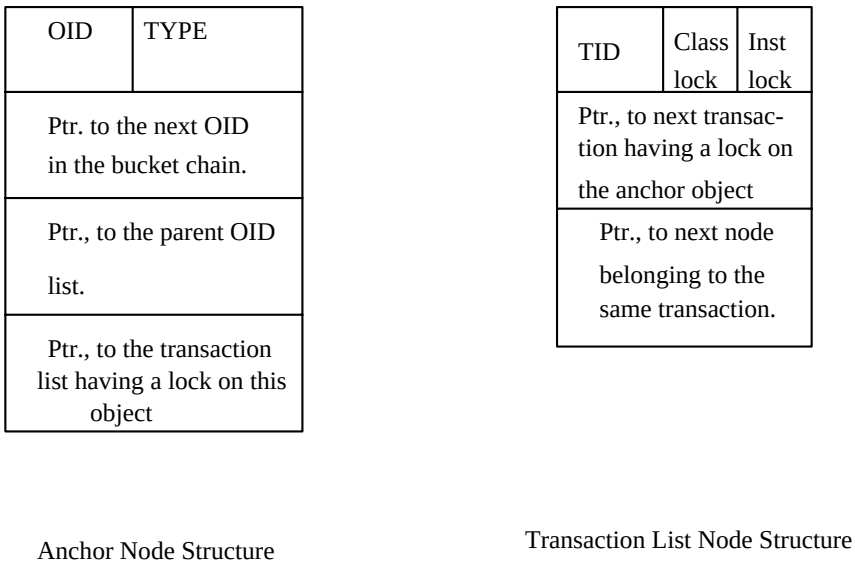

Figure 5.3. Node Structure in Lock Table

t1 = time taken (including I/O) to build the ancestor hierarchy and acquire locks on them.

t2 = time taken to search the established ancestor hierarchy for each instance and acquire locks on them. t1 >> t2

Total time taken to access all n instances:

(a) When ancestor hierarchy is not built = n*t1

(b) When the ancestor hierarchy is built and retained = t1 + (n-1)*t2

Thus the total time saved = (n-1) (t1-t2) = (n-1)*t1 as t2 is small compared to t1.

Thus building the ancestor hierarchy seems to be more efficient as t1 includes several I/Os and n can be in hundreds.

Seperating Class and Instance OID tables:

We have two OID tables, one for class objects and one for instance objects. This distinction is made for the following reasons:

- The number of instances is much more than the number of classes in the database. If the two tables are merged then the average search time for both

instances and classes will increase. Whereas, if they are seperate, the size of the class lock table will be considerably small and the average search time for both instances and classes is reduced.

- Since it is required to acquire intention locks on all ancestors for both the class and instance operations, the number of lock modes on classes are more than on instances. Thus for the class objects AHT structure is preferred, whereas a conventional hash table would suffice for the instance objects.

The difference between the class lock table (CLT) and instance lock table (ILT) are:

- CLT has an AHT structure, whereas instance lock table has an conventional hash table structure.
- The parent pointer in CLT points to its superclass(es) in the CLT, whereas in the ILT, it points to its parent class in the CLT. This is depicted in Figure 5.4.

5.4.4 Effect Of The Lock Table Structure On The Protocols

The effect of the above lock table structure on the protocols discussed in the previous chapter is discussed in this section.

Subclass Locking Protocol

This protocol requires to put explicit R/W lock on all subclasses of a class with multiple parents in addition to putting implicit locks on ancestors. Now, the locking protocol requires locks on some of the subcalsses too. We need to keep the subclass lattice information in addition to the ancestor lattice. There are two ways of incorporating this into the lock table.

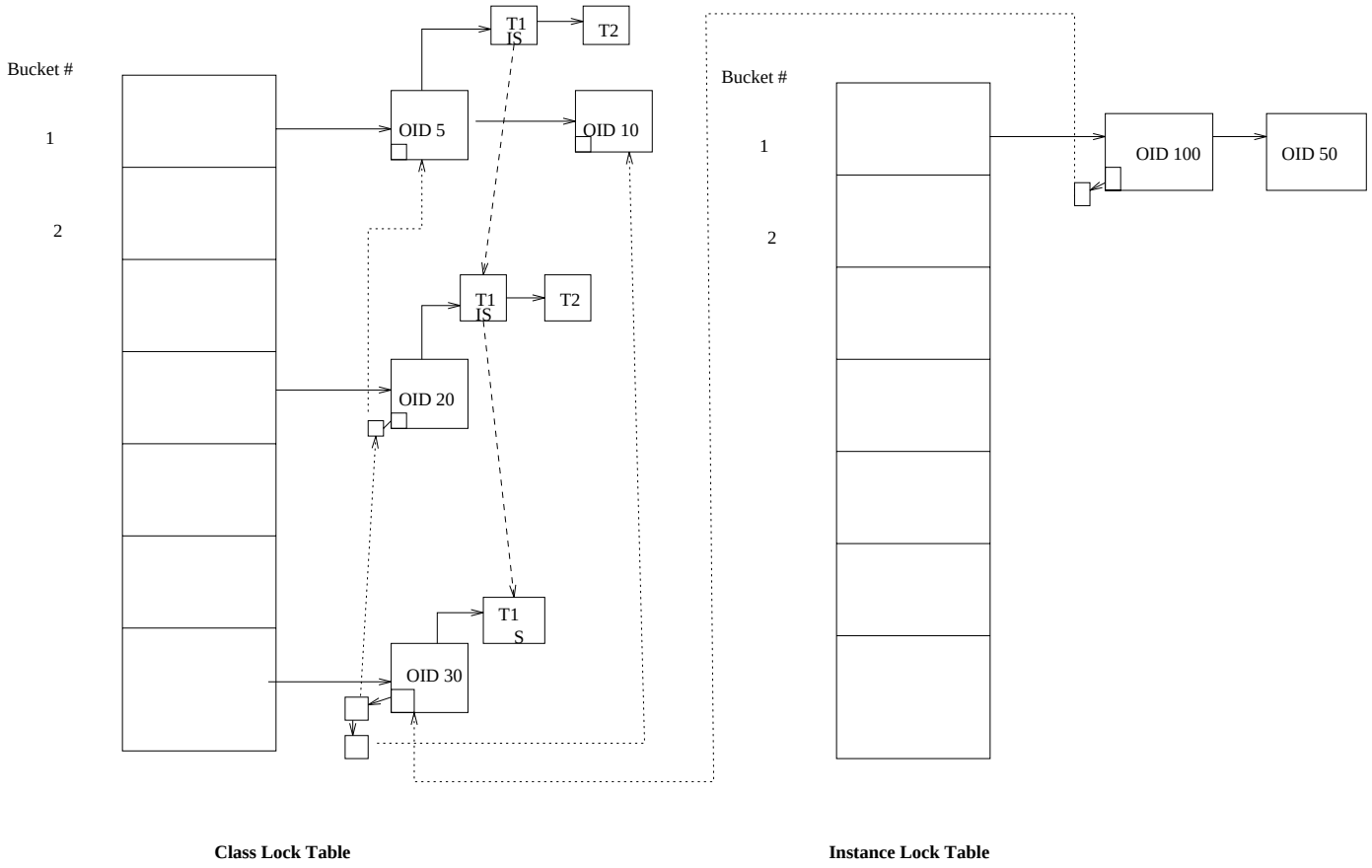

Figure 5.4. Class and Instance Lock Table

1. we can insert only the subclasses with multiple parents into the Class OID table. But then our claim that, there exists in the lock table a consistent lattice for all OIDs having an entry in the lock table, does not hold.

2. the alternative is to put explicit locks on all subclasses of the class and thus build the whole superclass/subclass structure for that class. The locking overhead can become heavy, and moreover, the notion of implicit locking is lost for class level operations.

Runthrough Protocol

The lock table structure need not be modified for this protocol. Whenever the transaction holding a R/W lock on a class *actually* accesses its subclass, a check

for conflict for that class alone is made and if there is no conflict an entry for that subclass is made in the lock table and the ancestor connection is established and the lock is granted. It need not check the conflict along the ancestor chain.

5.4.5 Use of Buffering Techinque for Commit/Abort:

Each time the disk is accessed BUFFER-SIZE (currently 8K) bytes consisting of objects are loaded into the EMS. Whenever an object has to be accessed, the EMS buffer is checked first and the disk is accessed only when the object is not found in the buffer. *Only* the desired object and not BUFFER-SIZE bytes is then sent to the client. This is done as there is no guarantee (without physical clustering) that the next required object will be in the BUFFER-SIZE buffer. Moreover, this buffer might contain an already deleted object, as the current system does not reuse the space of deleted objects. The client maintains a linked list called Objectlist. Each node in the Objectlist is an object requested by the client for this transaction. Objects modified by this transaction are flagged. At commit stage, this list is scanned and only those objects which are flagged are sent to the server for storage. Objects are also written back on the secondary storage before commit stage, but only on overflow.

In Version I the object file is an append file. All objects whether old or new are appended to the file. In case of new objects an entry is created in the OID table and the location of the object is stored in the table. In case of existing objects the new location is stored in the OID table. The old object is not deleted and remains as garbage. For the purpose of restoring the database to the original state, both the current location and the location of the object before the start of the a transaction is maintained. If the transaction commits, the current location is made permanent, and if the transaction aborts, the location before the start of the transaction is made permanent.

The original location of the object (location at the start of a transaction) is already available in the OID table. The OID table can be extended to hold the current location (location of the most recent write) of the object. The final location of the object is stored in the original entry in the OID table, depending on the commit or abort of a transaction.

5.5 Lock Table Operations for Lock-Aquisition and Lock-Release

As described earlier, there is a JOB-QUEUE where all the requests are queued and a LOCK-WAIT-QUEUE where the blocked transactions are queued. The operations that are mapped to operations on the lock table are read-class(c-oid), write-class(c-oid), read-all-instances-of-class(c-oid), write-all-instances-of-class(c-oid), read-instance(i-oid), write-instance(i-oid). Where c-oid and i-oid are class integer oid, and instance integer oid respectively. If a string oid is given then the system fetches the corresponding integer oid using the function OID(char *stroid) and then calls the above function.

The Protocols for these operations are summarized below:

Read/Write Class(c-oid):

- get IR/IW lock on all ancestors of c-oid
- get R/W lock on c-oid

Read/Write instance(i-oid):

- get IS/IX locks on c-oid and all ancestors of c-oid, where c-oid is the class, i-oid belongs to.
- get S/X lock on i-oid.

Read/Write all instances of class(c-oid):

- get IS/IX lock on all ancestors of c-oid
- get S/X lock on c-oid.

```
process-JOB-Q()
{

  while (end of JOB-Q)
  {
    process-request(operation)  /* operation is the node in JOB-Q */
    operation = operation->next
  }

}


process-request(operation)
{

  Switch(operation)
  {
    case(read-class(c-oid)):
              {acquire-class-lock(tid, c-oid, class, R, IR) }
              break;

    case(write-class(c-oid)):
              {acquire-class-lock(tid, c-oid, class, W, IW)}
              break;

    case(read-all-instances-of-class(c-oid)):
              {acquire-class-lock(tid, c-oid, class, S, IS)}
              break;

    case(write-all-instances-of-class(c-oid)):
              {acquire-class-lock(tid, c-oid, class, X, IX)}
              break;

    case(read-instance(i-oid)):
              {acquire-instance-lock(tid, i-oid, instance, S, IS)}
              break;

    case(write-instance(i-oid)):
              {acquire-instance-lock(tid, i-oid, instance, X, IX)}
              break;

   }
  return;
 }


acquire-class-lock(tid, c-oid, type, lock-mode, intention-lock-mode)
{

  Hash into CLT keyed on (c-oid,type);
  Search the anchor node list of that bucket for c-oid;
```

```
  if (present)
  {
    found = 1;
    c-oid-node = the node in the anchor node list with c-oid;
    if ( check-class-compatibility(tid, c-oid, type, found,
                    lock-mode, intention-lock-mode) )
    {
      Read the object from the disk;
      Send it to the client;
    }
  }
  else
  {
    found = 0;
    object = read the object from the disk;
    ancestor-list = the list of ancestor oids in the object.
    if ( check-class-compatibility(tid, c-oid, type, found,
                    lock-mode, intention-lock-mode) )
      Send object to the client;

   }
}


check-class-compatibility(tid, c-oid, type, found, lock-mode,
                intention-lock-mode)
{
  if(found)
  {
    Scan the transaction list in the anchor-node for c-oid and
                    then check;
    if( tid holds a lock 'existing mode' on c-oid)
    {
      if the 'existing node' is held for class operation, then
      using the conversion matrix find the 'new-mode' and the
      corresponding intention mode
         if (check-conflict(tid, c-oid, type, found, new-mode,
                  intention-lock-mode))
           Update the existing tid node to the new-mode and
           corresponding ancestors in the intentional mode,
           by traversing the parent pointers;

           return 1;
      else if(check-conflict(tid, c-oid, type, found, lock-mode,
                   intention-lock-mode))
             Make an entry in tid node for lock-mode and in
             its ancestors in intentional modes, by traversing
             the parent pointers;
             return 1;
    }
    else  if(check-conflict(tid, c-oid, type, found, lock-mode,
                intention-lock-mode))
```

```
            Create tid node with lock-mode for c-oid and for
            all their ancestors with intention-lock-mode in their
            respective transaction list and include them in the
            'same tid' list. This can be done by traversing the
            parent pointer list;
            return 1;
  }
  else
   if(check-conflict(tid, c-oid, type, found, lock-mode,
                          intention-lock-mode))
      {
       /*  Build the ancestor hierarchy */

       Create the anchor node for c-oid and create the tid node
       with its lock modes in the transaction list, and include
       tid in the 'same tid' list;

       Create the ancestor nodes and the tid nodes with intention-
       lock-mode in the respective anchor node transaction list;
       include tid in the 'same tid' list using PATH;

       return 1;
      }

  return 0;
 }



acquire-instance-lock(tid, i-oid, type, lock-mode,
                        intention-lock-mode)
{

  Hash into ILT keyed on (i-oid,type);
  /*ILT does not a AHT */

  Search the node list of that bucket for i-oid;
  if (present)
  {
    found = 1;
    i-oid-node = the node in the bucket linked node list
                    with i-oid;
    if ( check-instance-compatibility(tid, i-oid, type, found,
                        lock-mode, intention-lock-mode) )
    {
      Read the object from the disk;
      Send it to the client;
    }
  }
  else
  {
    found = 0;
```

```
    object = read the instance object from the disk;
    c-oid = the oid of the class the instance i-oid belongs to,
                obtained from object structure;
    /*ancestor-list = the list of ancestor oids in the object.*/
    if ( check-instance-compatibility(tid, i-oid, c-oid, type,
                        found, lock-mode, intention-lock-mode) )
      Send object to the client;

   }
}

check-instance-compatibility(tid, i-oid, c-oid, type, found,
                      lock-mode, intention-lock-mode)
{

  if(found)
  {
      Scan the node linked list for i-oid and check
      if( tid holds a lock 'existing mode' on i-oid)
      {
        Find the 'new-mode' from the conversion matrix based on the
        lock-mode and 'existing-mode';
        if (check-conflict(tid, i-oid, type, found, new-mode,
                  intention-lock-mode))
        {
              Update the existing i-oid node to the new-mode
              in the ILT;
              Update the corresponding ancestors's tid nodes
              in the intentional mode in CLT, by traversing the
              parent pointers;
              return 1;
        }
      }
      else  if(check-conflict(tid, i-oid, type, found, lock-mode,
                  intention-lock-mode))
            Create tid node with intention-lock-mode for c-oid
            and for all ancestors in their respective transaction list
            and include them in the 'same tid' list in CLT. This is be
            done by traversing the parent pointer list;

            Create and insert a node for i-oid in its bucket linked
            list in ILT and make the parent class connection in CLT
            and include this node in the 'same tid' list in ILT;

            return 1;
   }
  else
   if(check-conflict(tid, i-oid, type, found, lock-mode,
                        intention-lock-mode))
     {
       Build the ancestor hierarchy - create the anchor nodes
       for c-oid and its ancestors; and create the tid node with
```

```
       the intention-lock-mode in their respective transaction list in
       CLT. Also establish the 'same tid' connection;

       Create and insert a node for i-oid in its bucket linked list in
       ILT and make the parent class connection in CLT and include this
       node in the 'same tid' list in ILT;

       return 1;
     }

  return 0;
 }



check-conflict(tid, c-oid, type, found, lock-mode,
                   intention-lock-mode))
 {

  if (found)
  {
    Trace the parent pointers and check for conflits for
    intention-lock-mode in the tranasction list for each of
    the ancestors nodes.
    if(any one of them are in conflicting mode)
    {
      put that request in the LOCK-WAIT-Q;
      return 0;
    }
    else
    {
      check for conflict with lock-mode on c-oid for class
      operations and on i-oid for instance operations;

      if(no conflict)
        return 1;
    }
  }
  else
  {
   Hash on each of the ancestor to check for conflicts for
   intention-lock-mode;

   if any of the ancestors is already in the CLT then
      traverse the parent pointer to check for implicit locking
      and conflict;

   /* PATH is a data structure which keeps the OIDs and hash
      value for the ancestor path. It also keeps information about
      the OIDs in the ancestor list already in CLT. This information
      is used to traverse the parent pointer */
```

```
   Include the hash value and the OID in the PATH data structure.
   If the OID is already present then flag it in PATH to indicate
   that it already exists in CLT and its parent pointer can be
   traversed.

   Check for conflict with lock-mode on c-oid

   if(any one of them are in conflicting mode)
    {
      put that request in the LOCK-WAIT-Q;
      Discard PATH;
      return 0;
    }
    else
        return 1;

  }

}


lock-release(tid)
{

    Traverse the linked list connecting all 'tid' nodes and
          remove it from the lock table (both CLT and ILT);

    Scan the LOCK-WAIT-QUEUE and put all nodes with locks
    on all oids on which the locks were released by 'tid' in
    front of the JOB-Q;
}
```

An example is taken to illustrate the working of the lock-table-structure. Referring to Figure 4.5, suppose

1. T1:update I_2D

2. T2:read class definition of C_D

3. T3:read class definition of C_E

T1 gets IS locks on C_D, C_C, and C_A and an S lock on I_2D. Since the class hierarchy does not exist for C_D in the lock table at the beginning of T1, the ancestor hierarchy for C_D, C_C and C_A is built in the lock table.

T2 gets IR lock on C_C and C_A and R lock on C_D. Since the ancestor hierarchy already exists for C_D, the parent pointer list is traversed to create the T2 nodes.

The lock table structure at this point is shown in figure 5.5.

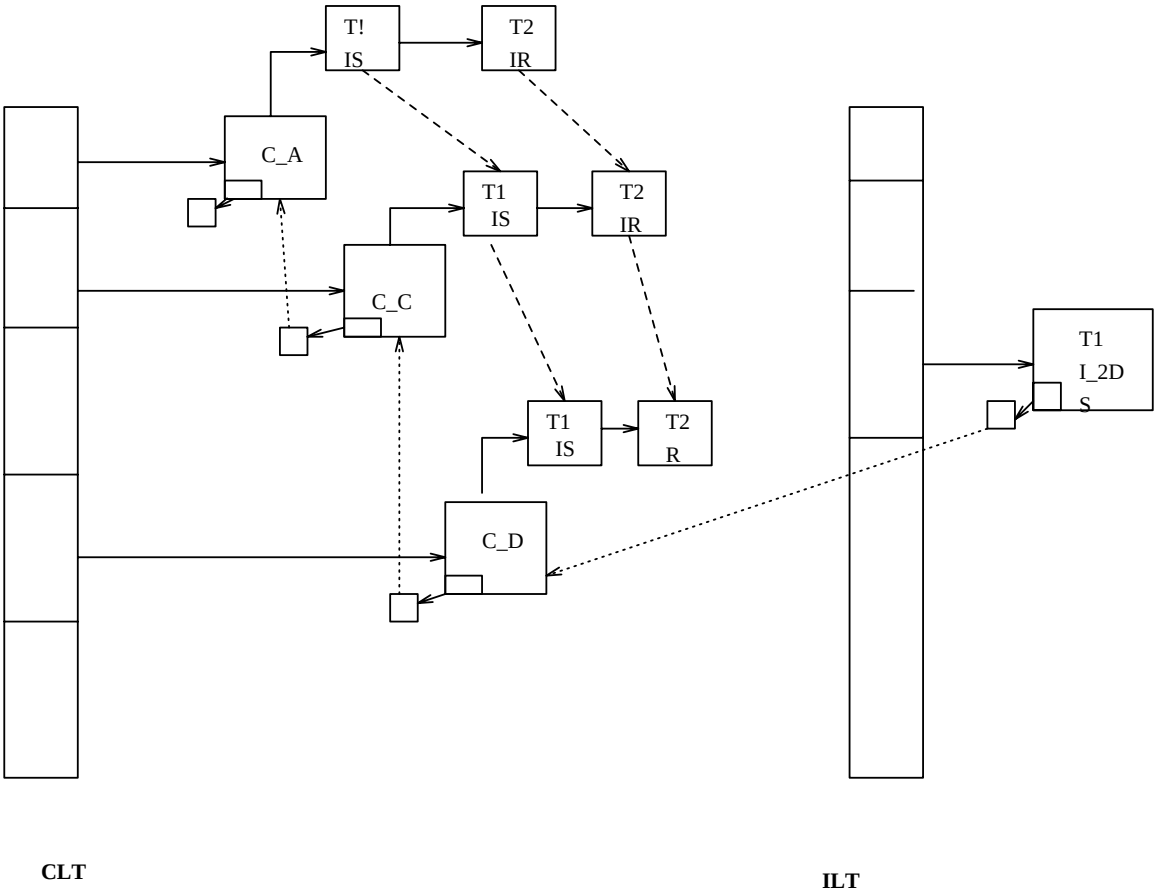

Figure 5.5. Lock Table Structure after T1 and T2 Acquired Locks

Now if T3 wants to read the definition of class C_E, it needs to only establish only the C_E - C_C connection. The rest of the ancestor hierarchy already exists, so it traverses through the parent pointer and sets locks.

If, T2 commits, the link connecting nodes of the T2 transaction is traversed in both CLT and ILT and all T2 nodes are removed. Thus, the various stages of lock table construction is illustrated through this simple example.

In summary, this chapter describes in detail the implementation of various components in the Version I of multi-user CANDIDE. First, the implementation of client

and server is explained, the building of the lock table for CANDIDE is explained next, and the use of the existing buffering techinique to provide recovery is described. Finally the working of the lock table was illustrated with an example.

CHAPTER 6
CONCLUSION

6.1 Summary

We have made three major contributions through this thesis towards making CANDIDE a multiuser system.

1. We have designed the multiuser CANDIDE for Version I and discussed the complete implementation of Version I. We chose the client/server architecture for its suitability for our system. Also, we discussed the preliminary thoughts for the design for Version II

2. We have designed an efficient concurrency control (CC) mechanism based on the theory of granularity locking and designed an efficient lock table to suit our CC mechanism. This CC mechanism is aimed at providing maximum parallelism.

3. Finally, we have demonstrated the feasibility of implementing this system on PCs. NetBIOS communication software was recommended to implement this system on PCs, as it is compatible with most of the existing LANs.

6.2 Future Work

As this is the first step towards making CANDIDE a multiuser DBMS, there are lot of extensions required to make the system efficient and complete. Some of them are listed below.

- Recovery has not been addressed in this thesis. It can be easily implemented for version I using the OID table. For Version II, the next step would be to design an efficient recovery protocol for CANDIDE.

- Extensions can be made to provide efficient buffer management. There exists and buffer management policy in Version II, this thesis does not discuss the effect of CC on this buffer management policy.

- One important future work would involve the design of the deadlock manager for CANDIDE.

- Version II stores multiple copies of an object to speed up access. The effect of this on CC mechanism has not been discussed in this thesis and this would be another area of interest in future.

- The effect of physical clustering on CC mechanism at the implementation level has to be studied.

- Finally, research can be done towards having multiple servers.

APPENDIX A
THE CLIENT PROGRAM

```
#include <stdio.h>
#include <iostream.h>
#include <dos.h>
#include <io.h>
#include <string.h>
#include <mem.h>
#include <fcntl.h>
#include <sys\stat.h>
#include <process.h>
#include <share.h>

#include <netbios.h>

NCB             send_ncb;
NCB             add_name_ncb;
NCB             delete_name_ncb;
NCB             call_ncb;
NCB             cancel_ncb;
NCB             hangup_ncb;

// NetBios machine must be of 16 chars. If the name is not of 16 chars
// the it is padded to the right with blank spaces.

void expand_to_16_chars(char *name)
     {
     char *p;
     char tmp[16];
     int i;

     memset(tmp, ' ', 15);
     p = name;
     i = 0;
     while (i < 15 && *p)
           {
           tmp[i] = *p;
           i++;
           p++;
           }
     tmp[15] = '\0';
     strcpy (name, tmp);
     }
```

```
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
/*
 *  Build the 'add_name' NCB and send it out
 *  across the network.
 *
*/
void    net_add_name(char *name)
        {
        memset(&add_name_ncb, 0, sizeof(NCB));
        add_name_ncb.NCB_COMMAND = ADD_NAME;
        strcpy(add_name_ncb.NCB_NAME, name);
        expand_to_16_chars(add_name_ncb.NCB_NAME);
        NetBios(&add_name_ncb);
        }

/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
/*
 *    Build the 'delete_name' NCB
 *
*/
void    net_delete_name(char *name)
        {
        memset(&delete_name_ncb, 0, sizeof(NCB));
        delete_name_ncb.NCB_COMMAND = DELETE_NAME;
        strcpy(delete_name_ncb.NCB_NAME, name);
        expand_to_16_chars(delete_name_ncb.NCB_NAME);
        NetBios(&delete_name_ncb);
        }

/* ---------------------------------------- */
/*
 * "Call" another workstation.
*/

void    net_call(char *who, char *us, unsigned char rto, unsigned char sto)
        {
        memset(&call_ncb, 0, sizeof(NCB));
        call_ncb.NCB_COMMAND = CALL;
        strcpy(call_ncb.NCB_NAME, us);
        strcpy(call_ncb.NCB_CALLNAME, who);
        expand_to_16_chars (call_ncb.NCB_NAME);
        expand_to_16_chars (call_ncb.NCB_CALLNAME);
        call_ncb.NCB_RTO = rto;
        call_ncb.NCB_STO = sto;
        NetBios(&call_ncb);
        }

/* ---------------------------------------- */
/*
 *  Build the "Send" NCB and send it across the network.
*/
```

```
void net_send(unsigned char lsn, void far *packet_ptr, int packet_len)
     {
     memset(&send_ncb, 0, sizeof(NCB));
     send_ncb.NCB_COMMAND = SEND;
     send_ncb.NCB_LSN = lsn;
     send_ncb.NCB_LENGTH = packet_len;
     send_ncb.NCB_BUFFER_PTR = packet_ptr;
     NetBios(&send_ncb);
     }


/* - - - - - - - - - - - - - - - - - - - - - - - - */
/*
 *    Build the 'cancel' NCB and send it out
 *    across the network.
 *
*/
void    net_cancel(NCB *np)
        {
        memset(&cancel_ncb, 0, sizeof(NCB));
        cancel_ncb.NCB_COMMAND = CANCEL;
        cancel_ncb.NCB_BUFFER_PTR = np;
        NetBios(&cancel_ncb);
        }


/* - - - - - - - - - - - - - - - - - - - - - - - - */
/*
 *    Build the 'hang up' NCB and send it out
 *    across the network.  Wait for completion.
 *
*/
void    net_hangup(unsigned char lsn)
        {
        memset(&hangup_ncb, 0, sizeof(NCB));
        hangup_ncb.NCB_COMMAND = HANG_UP;
        hangup_ncb.NCB_LSN = lsn;
        NetBios(&hangup_ncb);
        }



int main(void)
{

 char           machine_name[16] = "xxx_big_kahuna";
 char           local_name[16];
 char           my_name[16];
 unsigned char  *netbios_name_number, local_session_number;


 int            i;                  // Initialising
```

```
char           destination[16];
char           mes_buffer[] = "Hello BIG_K";


// The first step would be to test if NetBIOS is loaded.
// If not exit.

Step_1:

       // Testing if NetBios is loaded!!!!
        if ( is_netbios_loaded() == 0)
           {
           printf( "Netbios is not loaded \n");
           exit(1);
           }
        else
            printf(" NETBIOS is loaded \n");

// The second step would be to "add_name_ncb"
// This is how you would be recognised to the others

Step_2:

       get_machine_name(local_name, netbios_name_number);
       printf (" This machine is %s stops here \n", local_name);
       gets(my_name);
       net_add_name(my_name);
       while (add_name_ncb.NCB_CMD_CPLT == 0xFF)
             ;
       if (add_name_ncb.NCB_CMD_CPLT != 0)
          {
          printf("ERROR. \"add_name_ncb\" NetBios says : %s. \n",
net_error_message[(int) add_name_ncb.NCB_CMD_CPLT]);
          exit(1);
          }
       printf(" Added the Machine name to the local name table = %s \n",
my_name);
       getchar();


// Step_3 would be to establish a session with the server using the CALL
// command.


Step_3:

       strcpy(destination, machine_name);
       expand_to_16_chars(destination);
       net_call(destination, my_name, 20, 20);
       while (call_ncb.NCB_CMD_CPLT == 0xFF)
             ;
```

```
        if (call_ncb.NCB_CMD_CPLT == 0)
           {
           local_session_number = call_ncb.NCB_LSN;
           printf("LSN = %d \n", local_session_number);
           }
        else
            {
            printf("Error:  \"call_ncb\" NetBios says : %s. \n",
net_error_message[(int) call_ncb.NCB_CMD_CPLT]);
            exit(1);
            }
        printf("Connection Established with the server.\n");
        getchar();
        getchar();


 // Once you establish a connection then send messages to the server
 // using the send messge

Step_4:

        strcpy(destination, machine_name);
        expand_to_16_chars(destination);
        net_send(local_session_number, mes_buffer, strlen(mes_buffer));
        while (send_ncb.NCB_CMD_CPLT == 0xFF)
              ;

        if (send_ncb.NCB_CMD_CPLT != 0)
            {
            printf("Error:  \"send_ncb \" NetBios says : %s. \n",
net_error_message[(int) send_ncb.NCB_CMD_CPLT]);
            exit(1);
            }
        printf("Sent the message to the server.\n");
        getchar();
        getchar();


// After sending the message cancel all pending operations using
// the Cancel command.


Step_5:

       net_cancel(&call_ncb);
       while (cancel_ncb.NCB_CMD_CPLT == 0xFF)
             ;
       if (cancel_ncb.NCB_CMD_CPLT != 0)
            {
            printf("Error:  \" cancel-ncb\" NetBios says : %s. \n",
net_error_message[(int) cancel_ncb.NCB_CMD_CPLT]);
            exit(1);
```

```
            }
          printf(" Canceled all pending operations. \n");
          getchar();
          getchar();


// After cancelling all pending operations the program should
// hangup the session it created


Step_6:

        net_hangup(local_session_number);
        while (hangup_ncb.NCB_CMD_CPLT == 0xFF)
              ;
        if (hangup_ncb.NCB_CMD_CPLT != 0)
             {
             printf("Error:  \"cancel_ncb\" NetBios says : %s. \n",
net_error_message[(int) cancel_ncb.NCB_CMD_CPLT]);
             exit(1);
             }
          printf(" Hung Up. \n");
          getchar();
          getchar();



 // Once you have sent a message to the other machine,
 // delete your name from the local table

 Step_7:

          net_delete_name(my_name);
          while (delete_name_ncb.NCB_CMD_CPLT == 0xFF)
                ;
          if (delete_name_ncb.NCB_CMD_CPLT != 0)
             {
             printf("ERROR. \"delete_name_ncd\" NetBios says : %s. \n",
net_error_message[(int) delete_name_ncb.NCB_CMD_CPLT]);
             exit(1);
             }

          printf(" Deleted the name from the local name table. \n");
          getchar();
          getchar();

return 0;
}
```

APPENDIX B
THE SERVER PROGRAM

```
#include <stdio.h>
#include <iostream.h>
#include <dos.h>
#include <io.h>
#include <string.h>
#include <mem.h>
#include <fcntl.h>
#include <sys\stat.h>
#include <process.h>
#include <share.h>

#include <netbios.h>

NCB             receive_ncb;
NCB             add_name_ncb;
NCB             delete_name_ncb;
NCB             listen_ncb;
NCB             cancel_ncb;


char            buffer[15];
unsigned char   local_session_number;

// NetBios machine must be of 16 chars. If the name is not of 16 chars
// the it is padded to the right with blank spaces.

void expand_to_16_chars(char *name)
     {
     char *p;
     char tmp[17];
     int i;

     memset(tmp, ' ', 15);
     p = name;
     i = 0;
     while (i < 15 && *p)
           {
           tmp[i] = *p;
           i++;
           p++;
           }
     tmp[15] = '\0';
     strcpy (name, tmp);
```

```
    }

/* - - - - - - - - - - - - - - - - - - - - - - - - */
/*
 *  Build the 'add_name' NCB and send it out
 *  across the network.
 *
*/
void    net_add_name(char *name)
        {
        memset(&add_name_ncb, 0, sizeof(NCB));
        add_name_ncb.NCB_COMMAND = ADD_NAME;
        strcpy(add_name_ncb.NCB_NAME, name);
        expand_to_16_chars(add_name_ncb.NCB_NAME);
        NetBios(&add_name_ncb);
        }

/* - - - - - - - - - - - - - - - - - - - - - - - - */
/*
 *    Build the 'delete_name' NCB
 *
*/
void    net_delete_name(char *name)
        {
        memset(&delete_name_ncb, 0, sizeof(NCB));
        delete_name_ncb.NCB_COMMAND = DELETE_NAME;
        strcpy(delete_name_ncb.NCB_NAME, name);
        expand_to_16_chars(delete_name_ncb.NCB_NAME);
        NetBios(&delete_name_ncb);
        }


/* - - - - - - - - - - - - - - - - - - - - - - - - */
/*
 *    Build the 'listen' NCB and send it out
 *    across the network. */

void    net_listen(char *caller, char *us,
                           unsigned char rto, unsigned char sto)
        {
        memset(&listen_ncb, 0, sizeof(NCB));
        listen_ncb.NCB_COMMAND = LISTEN;
        strcpy(listen_ncb.NCB_NAME,     us);
        strcpy(listen_ncb.NCB_CALLNAME, caller);
        expand_to_16_chars(listen_ncb.NCB_NAME);
        expand_to_16_chars(listen_ncb.NCB_CALLNAME);
        listen_ncb.POST_FUNC = NULL;
        listen_ncb.NCB_RTO = rto;
        listen_ncb.NCB_STO = sto;
        NetBios(&listen_ncb);
        }
```

```
/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
/*
 *    Build the 'cancel' NCB and send it out
 *    across the network.
 *
*/
void    net_cancel(NCB *np)
        {
        memset(&cancel_ncb, 0, sizeof(NCB));
        cancel_ncb.NCB_COMMAND = CANCEL;
        cancel_ncb.NCB_BUFFER_PTR = np;
        NetBios(&cancel_ncb);
        }

/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - */
/*
 *    Build the 'receive' NCB and send it out
 *    across the network.  When the operation completes,
 *    let NetBIOS call the POST routine to handle it.
*/
void    net_receive(unsigned char lsn,
                        void *packet_ptr, int packet_len)
        {
        memset(&receive_ncb, 0, sizeof(NCB));
        receive_ncb.NCB_COMMAND = RECEIVE;
        receive_ncb.NCB_LSN = lsn;
        receive_ncb.NCB_LENGTH = packet_len;
        receive_ncb.NCB_BUFFER_PTR = packet_ptr;
        receive_ncb.POST_FUNC = NULL;
        NetBios(&receive_ncb);
        }


int main(void)
{

 char           local_name[16];
 unsigned char  *netbios_name_number;
 int            i;                  // Initialising
 char           source[16] = "Dot_machine";



 // The first step would be to test if NetBios is loaded

 Step_1:

        if ( is_netbios_loaded() == 0)
           {
           printf( "Netbios is not loaded \n");
           getchar();
```

```
           exit(1);
           }
        else
            printf(" NETBIOS is loaded \n");


 // Step_2 would be to add the name to the local name table

 Step_2:

        //get_machine_name(local_name, netbios_name_number);
        printf("Enter the local table name: ");
        gets(local_name);
        printf (" \n This machine is %s stops here \n", local_name);
        net_add_name(local_name);
        while (add_name_ncb.NCB_CMD_CPLT == 0xFF)
              ;
        if (add_name_ncb.NCB_CMD_CPLT != 0)
           {
           printf("Error. NetBios said %s.\n", net_error_message[(int)
add_name_ncb.NCB_CMD_CPLT]);
           exit(1);
           }
        printf(" Added the name to the local name table %s. \n", local_name);


Step_3:

// The third step would be to LISTEN for calls
       net_listen(*, local_name, 20, 20);
       printf("Listening for the workstation\n");
       while (listen_ncb.NCB_CMD_CPLT == 0xFF)
              ;

        if (listen_ncb.NCB_CMD_CPLT == 0)
           {
           local_session_number = listen_ncb.NCB_LSN;
           printf ("LSN = %d \n", local_session_number);
           }
        else
            {
            printf("Error:  \"listen_ncb\" NetBios says : %s. \n",
net_error_message[(int) listen_ncb.NCB_CMD_CPLT]);
            exit(1);
            }
        printf("Connection Established with the server.\n");
        getchar();
        getchar();



Step_4:
```

```
// Once the listen command is over the server is ready to receive.
// This is done using the RECEIVE command.

        net_receive (local_session_number, (void far *) buffer, sizeof(buffer));
        while (receive_ncb.NCB_CMD_CPLT == 0xFF)
              ;

        if (receive_ncb.NCB_CMD_CPLT != 0)
            {
            printf("Error:  \"receive_ncb\" NetBios says : %s. \n",
net_error_message[(int) receive_ncb.NCB_CMD_CPLT]);
            exit(1);
            }
        printf("The message is \" %s \" \n ", buffer);
        getchar();
        getchar();
 Step_5:

 // The final step would be to delete the name from the local name table

        net_delete_name(local_name);
        while (delete_name_ncb.NCB_CMD_CPLT == 0xFF)
              ;

        if (delete_name_ncb.NCB_CMD_CPLT != 0)
           {
           printf("ERROR. NetBios said:  %s.\n", net_error_message
[(int)delete_name_ncb.NCB_CMD_CPLT]);
           exit(1);
           }
//        printf("Received the message from the source\n");
        getchar();


 return 0;
}
```

APPENDIX C
THE BNF GRAMMAR OF CANDIDE

```
<class>       ::= <classname> CLASS <primative-flag>
            [SUPERCLASSES <superclass>+]
                [SUBCLASSES <subclass>+]
                [INSTANCE-LIST <inst>+]
                [ATTR-CONSTRAINTS <attr-constraint>+]


<instance>        ::= <inst-name> INSTANCE[<iparent>+]
                      [ATTR <attr-value>+]

<attr>            ::= <attr-name> ATTR [<attr-parent>] <ve>

<disjoint-class>  ::= <disjoint-name> DISJOINT
                         <classname> <disj>+

<primative-flag>  ::= PRIMITIVE|DEFINED

<attr-constraint> ::= <attr-name> <constraint>+

<attr-value>      ::= <attr-name> <type-i>+

<constraint>      ::= <max> | <some> | <exactly> | <all>

<max>             ::= ATMOST <integer>

<some>            ::= ATLEAST <integer> <ve>

<exactly>         ::= EXACTLY <integer> <ve>

<all>             ::= ALL <ve>

<ve>              ::= (DOMAIN <type-c>)|(VALUE<type-i>)|NIL

<type-c>          ::= (CLASS <classname>)|STRING|INTEGER|
                         |REAL|(RANGE <range>)|(SET <type-c>+)|
                      (SETDIF <classname> ',' <classname>)|
                         (COMPOSITE <attr-constraint>+)

<type-i>          ::= (CLASS <classname>)|
      (INSTANCE <inst-name>)|
      STRING <string>)|
      (INTEGER <integer>)|REAL <real>)|
                         (RANGE <range>)|(SET <type-i>+)|
      (SETDIF <classname> ',' <classname>)|
```

```
      (COMPOSITE <attr-value>+)

<range>           ::= (( '(' | '[') <num> |NIL)','
                      ( NIL | <num> ( ')' | ']'))

<num>             ::= <real> | <integer>

<superclass>      ::= <classname>

<subclass>        ::= <classname>

<inst>            ::= <inst-name>

<iparent>         ::= <classname>

<attr-parent>     ::= <attr-name>

<disj>            ::= <classname>

<classname>       ::= <string>

<inst-name>       ::= <string>

<attr-name>       ::= <string>

<disjoint-name>   ::= <string>
```

REFERENCES

[Art94] Artisoft's Bulletin Borad Service. The DOS SHARE Command Explained. Share.TXT, 1994.

[Bad93] R. Badani. Nested transactions for concurrent execution of rules: Design and implementation. Master's thesis, Computer Information Sciences Department, University of Florida, Gainesville, FL, October 1993.

[Ban91] Francois Bancilhon, Claude Delobel, and Paris Kanellakis. Building an Object-Oriented Database System – The Story of O2. San Mateo, CA: M. Kaufmann Publishers, 1992.

[Bar91] Naser S. Barghouti and Gail E. Kaiser. Concurrency Control in Advanced Database Applications. *ACM Computing Surveys*, 23(3):269–317, Sep. 1991.

[Bec93] Howard W. Beck. FAIRS Database Design Manual. Technical report, University of Florida, Gainesville, Fl. Aug 1993.

[Bec94] Howard W. Beck, Tarek Anwar, and Shamkant B. Navathe. A Conceptual Clustering Algorithm for Database Schema Design. *IEEE Transactions on Knowledge and Data Engineering*, 6:396-411, June 1994.

[Beck89] Howard W. Beck, Sunit Gala, and Shamkant B. Navathe. Classification as a Query Processing Technique in the CANDIDE Semantic Data Model. *Proceedings International Conference on Data Engineering*, Los Angeles, CA., Feb 1989.

[Car90] Michele Cart and Jean Ferrie. Integrating Concurrency Control into an Object-Oriented Database System. *Proceedings, International Conference on Management of Data*, 3:363–377, June 1990.

[Chr90] Paul Christiansen. Networking With Novell NetWare: A LAN Manager's Handbook. Blue Ridge Summit, PA: Windcrest. 1990.

[Esw76] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The Notions of Consistency and Predicate Locks in Database Systems. *Communications of the ACM 19*, 10(11):624-633, Nov. 1976.

[Gar88] Garza J.F. and Won Kim. Transaction Managemanet in Object-Oriented Database System. *Proceedings, International Conference on Management of Data*, Chicago, Ill., pages 37–45, June 1988.

[Gra88] J. Gray, R. Lorie, G. Putzolu, and I. Traiger. Granularity of Locks and Degree of Consistency in a Shared Data Base. Readings in Database Systems, ed. M. Stonebraker. San Mateo, CA: Morgan Kaufmann, 1988.

[Gra93a] J. Gray and A. Reuter. Transaction processing: Concepts and Techniques. San Mateo, CA: Morgan Kaufmann, 1993.

[Gra93b] J. Gray and A. Reuter. Transaction Processing: Concepts and Techniques. San Mateo, CA: Morgan Kaufmann, 1993.

[Hae82] Theo Haerder and N. Pippengar. Observations on Optimistic Concurrency Control Schemes. Technical Report RJ 3645, IBM Thomas J. Watson Research Center, Yorktown Heights, NY., Oct 1982.

[Hen91] Henry F. Korth and Abraham Silberschatz. Database System Concepts. 1991. (Chapters 11 and 12 on Concurrency Control and Transaction Processing).

[Kun81] H. T. Kung and John T. Robinson. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[Mai85] David Maier, A. Otis, and A. Purdy. Development of an Object-Oriented DBMS. *Proceedings 1st International Conference on Object-Oriented Programming Systems, Languages, and Applications*. 472-486, Portland, Oregon, Oct. 1986.

[Mon91] Michael Montgomery. Networking with LANtastic. Blue Ridge Summit, PA: Windcrest, 1991.

[Nan90] Barry Nance. Network Programming in C. IN: QUE Corporation, 1990.

[Rou91] Nick Roussopoulos and Alexis Delis. Modern Client-Server DBMS Architectures. *Proceedings, International Conference on Management of Data*, 20(2):52–61, Sep. 1991.

[Sch85] W. David Schwaderer. C Programmer's Guide to NetBIOS. IN: H.W. Sams, 1985.

[Sco88] Currie W. Scott. LANS Explained: A Guide to Local Area Networks. New York: Halsted Press,1988.

[Zdo90a] Stanley B. Zdonik and David Maier. A Shared, Segmented Memory System for an Object-Oriented Database. In *Object-Oriented Database Systems*, pages 273–285. San Mateo, CA: Morgan Kaufmann, 1990.

[Zdo90b] Stanley B. Zdonik and David Maier. Fundamentals of Object-Oriented Databases. In *Object-Oriented Database Systems*, pages 1–33. San Mateo, CA: Morgan Kaufmann, 1990.

BIOGRAPHICAL SKETCH

Hema Kannan was born on August 15, 1970, at Shoranur, Kerela, India. She received her undergraduate degree in electrical and electronics engineering from Madurai Kamaraj University, India, in May 1992. She will receive her Master of Science degree in computer and information sciences from the University of Florida, Gainesville, in December 1994. Her research interests include object-oriented databases and implementation of client/server systems.