

FILE SHARING:
AN IMPLEMENTATION OF THE MULTIPLE WRITERS FEATURE

by

MARY KENNEY
B.A., Wichita State University, 1974

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

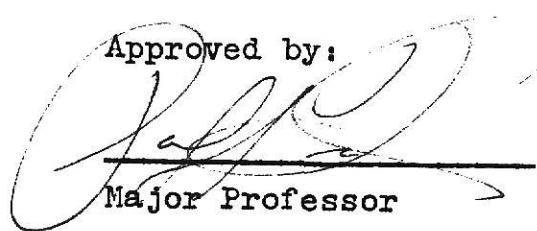
Department of Computer Science

KANSAS STATE UNIVERSITY

Manhattan, Kansas

1981

Approved by:



Major Professor

SPEC
COLL
LD
2668
.R4
1981
K46
C.2

TABLE OF CONTENTS

| | | |
|---------|--------------------------------------|----|
| 1.0 | Introduction | 1 |
| 1.1 | Overview | 1 |
| 1.1.1 | Purpose | 1 |
| 1.1.2 | Direction of Research | 1 |
| 1.1.3 | The Implementation | 1 |
| 1.2 | Definitions | 4 |
| 1.2.1 | Operating System | 4 |
| 1.2.2 | Disk Logical I/O | 4 |
| 1.2.3 | Application Programs | 5 |
| 1.2.4 | File Sharing | 7 |
| 1.2.5 | Multiple Writers | 8 |
| 1.2.6 | Locking Mechanisms | 8 |
| 2.0 | Review of Literature | 10 |
| 2.1 | General File Sharing Concepts | 10 |
| 2.1.1 | Motives for Sharing | 10 |
| 2.1.1.1 | Eliminate Duplication | 10 |
| 2.1.1.2 | Synchronize Operations | 11 |
| 2.1.1.3 | Cost and Efficiency | 11 |
| 2.1.2 | Access Control and Protection | 12 |
| 2.2 | Related Concepts and Implementations | 15 |
| 2.2.1 | Distributed Systems | 15 |
| 2.2.2 | Data Base Management Systems | 17 |
| 2.2.3 | Concurrent Programming | 18 |

| | | |
|-----------|--|----|
| 2.3 | Problems | 21 |
| 2.3.1 | Controlling Access | 21 |
| 2.3.2 | Locking Parts of Files | 21 |
| 2.3.3 | Ensuring Reliability | 22 |
| 3.0 | Implementation of Multiple Writers Feature | 23 |
| 3.1 | Description of the System | 23 |
| 3.1.1 | Hardware | 23 |
| 3.1.2 | Software | 23 |
| 3.1.2.1 | Operating System | 23 |
| 3.1.2.2 | Files | 24 |
| 3.1.2.2.1 | Organizations | 24 |
| 3.1.2.2.2 | Access Methods | 27 |
| 3.1.2.2.3 | Open Modes | 28 |
| 3.1.2.2.4 | Functions | 29 |
| 3.1.2.3 | Data Structures | 29 |
| 3.1.3 | Compatibility Issues | 34 |
| 3.2 | Description of Implementation | 36 |
| 3.2.1 | Sector Lock and Unlock | 36 |
| 3.2.2 | Impact on Functions | 39 |
| 3.2.2.1 | Sequential Files | 39 |
| 3.2.2.1.1 | Open | 39 |
| 3.2.2.1.2 | Write | 42 |
| 3.2.2.1.3 | Read | 44 |
| 3.2.2.1.4 | Rewrite | 46 |
| 3.2.2.1.5 | Close | 49 |

| | | |
|-----------|---------------------------------|----|
| 3.2.2.2 | Relative Files | 50 |
| 3.2.2.2.1 | Open | 50 |
| 3.2.2.2.2 | Write | 51 |
| 3.2.2.2.3 | Read | 57 |
| 3.2.2.2.4 | Rewrite | 59 |
| 3.2.2.2.5 | Delete | 61 |
| 3.2.2.2.6 | Start | 63 |
| 3.2.2.2.7 | Close | 63 |
| 3.2.2.3 | Indexed Files | 64 |
| 3.2.2.3.1 | Write | 64 |
| 3.2.2.3.2 | Read | 65 |
| 3.2.2.3.3 | Rewrite | 66 |
| 3.2.2.3.4 | Delete | 67 |
| 3.2.2.3.5 | Start | 68 |
| 3.2.3 | File Sharing Problem Resolution | 69 |
| 3.2.3.1 | File Access Control | 69 |
| 3.2.3.2 | Sector Locking | 70 |
| 3.2.3.3 | Data Integrity | 71 |
| 3.2.4 | Testing | 74 |
| 4.0 | Conclusion | 74 |
| 4.1 | Advantages and Disadvantages | 74 |
| 4.2 | Alternatives | 75 |
| | Bibliography | 77 |

LIST OF FIGURES

| | | |
|--------------------|---|----|
| Figure 1.2.2-1 | The Relationship Among the Disk Logical I/O Submodules | 6 |
| Figure 3.1.2.1-1 | Processes in the System | 25 |
| Figure 3.1.2.2.4-1 | Legal I/O Functions | 30 |
| Figure 3.1.2.2.4-2 | List of Statuses | 31 |
| Figure 3.1.2.3-1 | Relationship Among Logical I/O Data Structures | 33 |
| Figure 3.2.3.2-1 | Summary of Sector Lock Impact | 72 |

ACKNOWLEDGEMENT

This report is based largely on work-related experience at NCR Corporation. Terry Johnson and Linda Lallement Rehme provided encouragement and thoughtfully reviewed the report. My advisor at KSU, Rod Bates, assisted me in my pursuit of a master's degree at KSU and was especially helpful with the master's report requirement. Alan, my husband, encouraged me throughout my master's program. My thanks to all of these people, to NCR and to the Department of Computer Science at Kansas State University for their support during this work.

1.0 Introduction

1.1 Overview

1.1.1 Purpose

File sharing capabilities have been included in a variety of computer systems, from large time sharing computers to small microprocessors. The purpose of the report is to discuss file sharing and in particular to describe the incorporation of file sharing into the operating system of a specific microprocessor.

1.1.2 Direction of Research

An attempt was made to find documentation on the way file sharing has been implemented in systems other than those built by NCR. However, except for some general descriptions found in sales brochures, it was not possible to do so since such documentation is licensed by the companies.

Research for the report was pursued in the direction of uncovering both descriptions of file sharing and related concepts and also discussions of problems that must be resolved when more than one user is allowed to access a file at a given time.

1.1.3 The Implementation

A major portion of the report deals with a programming effort that was carried out for NCR Corporation. The

feature of allowing more than one writer to a file at one time was added to an operating system of a microprocessor.

The author worked as part of a team of four to design, code and test the new feature. Writing the report and doing the associated research, however, was not part of the requirements of the work done at NCR. Low-level design of the feature was begun in February, 1980, and unit testing was completed the following July. The work associated with implementing the file sharing feature was done at NCR Engineering and Manufacturing in Wichita, Kansas.

The operating system that was modified first came into existence in early 1979. It evolved from an earlier operating system which was written for the same microprocessor. The earlier system supported a single workstation machine, whereas the later one supported a multi-workstation system, allowing up to four keyboard-CRT workstations. Initially, the logical I/O portion of the later system allowed applications running on different work-stations to access the same file as readers. Only one writer was allowed to access the file at a given time. Allowing multiple writers to a file was a new feature added to the already existing operating system, and the addition of the feature caused a number of changes to be made to the disk logical I/O part of the operating system.

One of NCR's goals in developing software is to have a certain amount of compatibility among systems of the same

family. Therefore, some of the decisions made about implementation were resolved in the direction of being compatible with two other operating systems. Basically, being compatible means that an application program written for one operating system will run on another and vice versa.

1.2 Definitions

Some of the terms used in the report are defined below.

1.2.1 Operating System

An operating system is a computer program, usually large and composed of many subprograms or modules, that provides a number of services for the users of the computer system. Some of the services provided by the operating system are: scheduling jobs, allocating resources, dispatching programs, communicating with the operator, recovering from incidents, recording statistics, and storing and retrieving data [4].

1.2.2 Disk Logical I/O

A major component of an operating system is the input/output (I/O) subprogram. The prominent I/O activities are: allocating I/O devices, processing user control statements, opening and closing files, file cataloging, allocating direct access storage device (DASD) space to files, maintaining device directory labels, storing and retrieving data, organizing files, data staging, and overall I/O supervision [4].

For convenience, the I/O subsystem can be divided into two parts: physical I/O and logical I/O. Physical I/O comprises the I/O device drivers and device support routines. Logical I/O is at a higher level and makes use of a number

of data structures to provide application programs with the ability to read and/or write to files or devices.

Disk is the DASD on the operating system that was modified. A large portion of the logical I/O system exists to provide support for the disk device. Disk logical I/O includes the following sub-modules:

- 1) Disk Directory - routines that allow control and maintenance of the disk directory,
- 2) Disk Common - a set of common routines used by the other sub-modules,
- 3) Indexed File Support - logical I/O support for indexed files,
- 4) Relative File Support - logical I/O support for relative files,
- 5) Sequential File Support - logical I/O support for sequential files,
- 6) Lock Routines - sector lock and unlock routines used by the other sub-modules, and
- 7) Spooler - routines that allow print files to be spooled to disk for later de-spooling to printer.

Figure 1.2.2-1 shows the relationship among the disk logical I/O sub-modules.

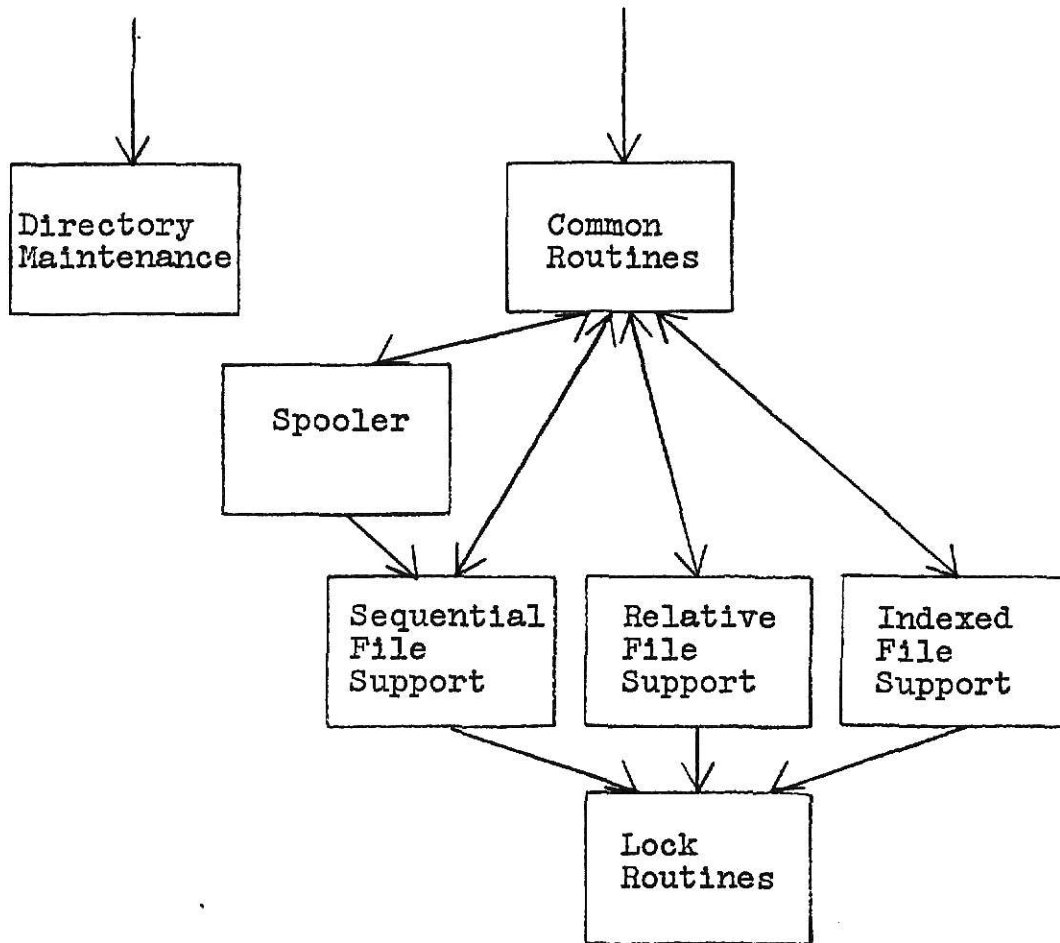
1.2.3 Application Programs

The terms user, application program, and application user will be used interchangeably. The means of accessing

**THIS BOOK
CONTAINS
NUMEROUS PAGES
WITH DIAGRAMS
THAT ARE CROOKED
COMPARED TO THE
REST OF THE
INFORMATION ON
THE PAGE.**

**THIS IS AS
RECEIVED FROM
CUSTOMER.**

Figure 1.2.2-1 The Relationship Among the Disk Logical I/O Submodules.



files or I/O devices is through application programs. User programs are executed under control of that part of the operating system that provides the environment and exercises the controls required to fulfill the user's requests [4].

Application programs written in COBOL, BASIC, and 8080 assembly programming languages may be executed in the operating system environment to be described in the report.

1.2.4 File Sharing

The term file sharing as used in the report means the capability of a computer system to allow more than one user to access the same file at the same time. Users are application programs running at different workstations or terminals connected to the system. The users may be readers of the file or writers to the file.

It may be helpful to distinguish file sharing from several similar concepts. Multiprogramming is a general term which refers to the executing of two or more programs concurrently using a single CPU [4]. The operating system interleaves the processing of the programs. File sharing takes place in a multiprogramming environment and is the special case in which the concurrent programs access the same data files.

Concurrent programming must resolve many of the same problems that should be addressed by a system that has file sharing. A concurrent program is a program that has several

parts in execution at a given time [6]. The mechanisms that allow concurrent programming are usually structures that are features of particular programming languages. As in multi-programming, the main concern in concurrent programming is scheduling access to resources. The emphasis in file sharing is similar, allowing data files to be accessed by multiple users while preserving the integrity of the files.

Preserving the integrity of data is also a major concern of a data base management system. A data base management system can be thought of as a large file sharing system. The same protection of data while it is accessed by many users must be provided by both types of systems.

1.2.5 Multiple Writers

Special problems can occur when more than one writer accesses a file at a given time. Measures must be taken to ensure that the data in a file remains in a consistent, useable state. These special problems and their methods of resolution are the subjects of the report and will be addressed throughout.

1.2.6 Locking Mechanisms

Specifically, the way data is protected in a multiple writer environment is with the use of locking mechanisms. There are many different implementations of data locking/unlocking, but the primary goal is to provide a means of

shared or exclusive access to the data [6]. One locking mechanism will be described in the report.

2.0 Review of Literature

2.1 General File Sharing Concepts

2.1.1 Motives for Sharing

Sharing of resources including data files has a number of advantages. The following three categories summarize most of the reasons for sharing resources:

- 1) eliminate duplication, yet provide accessibility;
- 2) allow the synchronization of operations; and
- 3) reduce cost and improve performance of system use [9].

2.1.1.1 Eliminate Duplication

File sharing capability is not required in a batch environment, but would be helpful in an interactive environment that supports multiple users. The requirements of an interactive system are such that files should be able to be updated immediately.

If two users need to access or update the information contained in a file and file sharing is not allowed in the computer system, then one of two alternatives exist. Either the users must wait for each other to finish accessing the file [8], or the file must be duplicated giving each user a copy [12]. Both alternatives create problems for the user. In the first, valuable time is wasted while one user waits for the other. In some application environments such inaccessibility of the data could cause serious problems

and therefore would not be acceptable.

The second case eliminates the problem of inaccessibility, but if both users are making changes to the data, then neither user has access to the most up-to-date information all the time. The problem of periodically updating all copies of the file must also be resolved. Once again, there are environments in which users depend upon the information supplied by other users. Therefore, the second alternative would not be acceptable.

2.1.1.2 Synchronize Operations

Sharing of files allows synchronization of data operations. For example, all modifications of an individual's bank balance, whether by human or automatic tellers, should be made to the same file even if the updates coincide in time [9]. The computer system will have to provide the necessary synchronization controls so that one update does not destroy the other updates that may occur at the same time. In addition, the file will always be current if the operations are synchronized in a file sharing environment.

2.1.1.3 Cost and Efficiency

Duplication of files is costly in terms of secondary storage and in terms of the overhead involved in periodically updating all the copies. If the files can be shared with all operations for the same information being carried out on one copy of the file, the cost of duplication can

be eliminated [9].

The computer system is more efficiently utilized if files can be shared by users than if the users must wait for each other to finish before being able to access the files.

2.1.2 Access Control and Protection

In a system that allows file sharing, access to files must be controlled so that the data can be protected. Several modes of access control are sometimes found within the same computer system [12].

The most basic kind of access control is the "all or nothing" mode. In this case the user can either perform any operation on the data (read, write, etc.) or the user is not allowed to access the data at all. Certain files may be able to be designated as private, meaning that only one user can have access to the file at any given time. In addition, when a file is being created the user setting up the file initially may be provided with exclusive use of the file at least until the file's characteristics can be established.

Some systems provide elaborate modes of protection. Certain information can be accessed by certain users in a "read-only" mode while other users may be able to write to the data. MULTICS [10] differentiates the following access attributes: read, write, execute, append, and trap. These

attributes are stored in the directory and are a means of controlling access to the files. MULTICS also makes use of a logical control structure called "Rings of Protection". Location within a particular ring dictates the information that may be accessed and the mode of access for a program or system module.

Protection keys specifying access permissions are used in the UNIX Time-Sharing System [7]. The keys are part of the file directory. Usually, data files have general read/write permissions making their contents sharable between processes.

If two users are allowed to access the same data in "write" mode, there must be a way of controlling access to portions of the file at critical times. Sections of the file must be able to be protected temporarily so that only one user at a time may be allowed access while updating is taking place. Usually, some type of locking controls are used for shared data [10]. Locking mechanisms vary from simple to quite complex. A process or program may be allowed to have only one portion of a file locked at a time or may be allowed to place multiple locks. The locks may be exclusive access locks, allowing no other process to read or write to the data, or they may be a combination of types of locks, including exclusive locks as well as locks that allow other readers but not other writers to access the data.

Access control and ensuring the integrity of data are

major concerns of any system that allows the sharing of data. The value of a concurrent file system to be used in a business environment is determined by how well the system protects the data and ensures its integrity.

2.2 Related Concepts and Implementations

2.2.1 Distributed Systems

Woodstock File System (WFS) [13] is a shared file system which operates in a distributed network. The user is provided with a limited repertoire of atomic operations which may be used to create files or modify pages of a file. The file directory contains access control information specifying whether or not the file may be shared. Each page also contains certain status information such as a "dirty bit" which indicates that a change has been made to the page if the bit is set.

In WFS "a client may lock a file, preventing access by anyone without the proper key. The lock operation returns a key that must be supplied with all subsequent operations on the file, until either the client issues an unlock operation or the lock breaks. WFS will break a file's lock if no operation has been performed on the file for a minute or so. A system restart breaks all locks. A key of zero fits an unlocked file. A client can detect a broken lock because the non-zero key will not fit the lock on an unlocked file.

| <u>key</u> | <u>lock</u> | <u>access</u> | <u>file state</u> |
|------------|-------------|---------------|-------------------|
| 0 | 0 | allowed | unlocked |
| 0 | X | denied | locked |
| X | X | allowed | locked |
| X | Y | denied | locked |
| X | 0 | denied | unlocked |

These locking operations provide primitives that are adequate

to implement completely safe sharing mechanisms" [13, p.11].

To support shared access to files in WFS, the following categories of file system information exist:

- 1) "long-term information" which includes such things as the file itself, directory information, and system allocation tables. This information exists throughout the file's lifetime;
- 2) "medium-term information" contains the timeout lock that enables the sharing of data;
- 3) "short-term information" contains the information that must be kept during the execution of an operation.

The approach to sharing by WFS is very primitive but has been shown to provide reliable shared access to user files.

Paxton [11] discusses mechanisms for maintaining data integrity in a distributed transaction processing environment. He identifies the following important system properties:

- "1) The consistency property: although many clients may be performing transactions simultaneously, each will get a consistent view of the shared data as if the transactions were being executed one at a time.
- 2) The atomic property: for each transaction, either all of the writes will be executed or none of them will, independent of crashes..." [11, p.18].

The lock mechanism described by Paxton is similar to that of WFS in which a key is used to gain access to a file. The lock status of data files and of intentions files (used

for back-up and recovery) are shown to be useful in recovering from crashes.

2.2.2 Data Base Management Systems

Sharing of data is usually allowed in a Data Base Management System (DBMS). A system provided by Texas Instruments [14], the DBMS-990, for example, allows users to share data files. The means by which data is protected during critical updates is provided in the user program languages. Records can be explicitly locked in both the assembly language that the system supports and in COBOL.

In a discussion of the "concurrent update problem", Tsichritzis and Lochovsky [15] identify a number of the issues that complicate locking of data to prevent access to data that is being modified. Tradeoffs must be made in the area of how much of the data base should be locked at one time for one operation. Many DBMS allow various amounts of the data base to be locked, from a record, to a record type, to a view, to the entire data base, depending upon the operation to be performed.

What happens when a locking request cannot proceed is another issue that must be resolved. There are a number of ways to deal with the request. Most systems cause the process to wait until the lock can proceed, however a fake modification could be performed in which the changes would be placed in a temporary file until they could be made to the

data base.

Most DBMS allow more than one lock to be placed on the data base by a process at one time. This ensures that all related data can be updated to provide consistency. However, allowing more than one lock can cause deadlocks. A deadlock occurs when a process has a resource locked that another process needs and vice versa. So, both processes are waiting for each other to release a resource.

Another issue involved with locking is the length of time locks are in place. "To maximize concurrent use of a data base, it is important that items are locked for the minimum amount of time possible" [15, p.260].

2.2.3 Concurrent Programming

As mentioned earlier, many of the concerns of concurrent programming are also concerns of implementing file sharing. Also, many of the problems faced by each are the same. File sharing could be implemented by writing parts of an operating system in a programming language that contains concurrent structures.

Basic to concurrent programming is the concept of critical sections. While shared data is being updated by a process, no other process should be allowed to execute those same statements, updating the data [6]. A number of synchronization primitives have been described in the literature. These primitives have to do with causing a process to wait

while another process executes a set of program statements. After the process completes execution, the waiting process is signaled that it may begin execution of the statements. The synchronization primitives are primarily used to allow only one process at a time to execute some set of code, thus protecting the shared data from simultaneous update.

An example of the similarity of the goals of concurrent programming and an implementation of operating system support for file sharing can be found in the discussion of "readers and writers" presented by Courtois, et al [3]. The problem centers around controlling and providing mutual exclusion for processes accessing critical sections of code. Using P and V semaphore operations, the authors demonstrate two ways of providing the "reader" processes with the ability of sharing a resource with other readers, while at the same time providing the "writer" processes with exclusive access to the resource. This is exactly what is required of a system that proposes to allow the sharing of files by both readers of the files and writers to the files.

Deadlock can occur in concurrent programming as in DBMS since several processes compete for a limited number of resources. To control allocation of resources, the program structure called a monitor may be used. A monitor can be thought of as a scheduling primitive [1,2,5].

There are a few differences between concurrent programming and the implementation of file sharing discussed in

the report. For one thing, concurrent language structures were not available to help with the implementation. In addition, concurrent programming is concerned with data in memory, whereas the data being protected in the file sharing implementation resides on disk. And finally, concurrent programming manipulates data that may not change in type or size at runtime. Nevertheless, protection of critical sections of code, synchronization of operations, and scheduling of resources are all part of the underlying operating system that supports sharing of files.

2.3 Problems

Many of the problems that must be addressed when file sharing capabilities are added to an operating system have already been mentioned. To summarize, the problems may be categorized as follows:

- 1) controlling access to files,
- 2) locking portions of files, and
- 3) ensuring reliability.

2.3.1 Controlling Access

An operating system that supports file sharing must not only make it possible for several users to access the same file but should also be able to prevent more than one user from accessing a particular file. Some files should be allowed to be private, and the user should be able to "own" a file, that is, the user should be able to be allowed exclusive access to a file if necessary.

2.3.2 Locking Parts of Files

A whole group of issues must be resolved under the category of file locking. Some are listed below:

- 1) how many locks per process at a given time may be in place,
- 2) what amount of data may be locked,
- 3) how long may locks be in place,
- 4) how is deadlock prevented/detected,

- 5) how are requests blocked if the data is already locked by another process, and
- 6) how long are blocked requests allowed to wait.

2.3.3 Ensuring Reliability

A system should provide the means for ensuring data integrity. This is especially crucial and difficult to achieve when more than one user is allowed to access the same data file at the same time. If two or more processes, for example, are allowed to simultaneously update the same record [4], the results will be unpredictable.

In addition, if a hardware or permanent error occurs or the system crashes while a file is being used, the operator should be given some indication that the file may be unreliable. At best, recovery procedures could be built into the system, so that the file could be restored to a consistent state.

3.0 Implementation of Multiple Writers Feature

3.1 Description of the System

3.1.1 Hardware

The system that was modified is an interactive micro-processor. It has from 64K to 256K bytes (8 bits/byte) of main memory and secondary storage of four to eight disks, one of which is removeable. The system includes one to four keyboard-CRT workstations and one or two printers. One of the printers may be a ledger printer. In addition, a data cartridge drive is part of the system and is used primarily for system generation and file back-up. Other peripherals that may be included are up to four flex disk drives and up to two cassette drives.

3.1.2 Software

3.1.2.1 Operating System

The operating system includes software support for all of the devices listed in section 3.1.1. The operating system is a virtual memory system, making use of variable demand pages (VDP) and overlays. Scheduling of processes is accomplished with time slicing in a round-robin fashion.

Processes in the system are resources from which jobs may be run. Each workstation has the capability of allowing jobs to be initiated from a primary or a secondary mode or

to be placed on a batch queue. When secondary mode is entered, the job running in primary mode is suspended. Therefore, a system configured with four workstations and batch has nine processes, five of which may have jobs running concurrently (figure 3.1.2.1-1).

The operating system is written in 8080 assembly language and supports the creation and manipulation of various types of files. Application programs written in COBOL, BASIC and 8080 assembly language may be executed. A number of system utilities are also provided which allow the user to move files from one device to another, inspect device directories, inquire into the contents of files, etc.

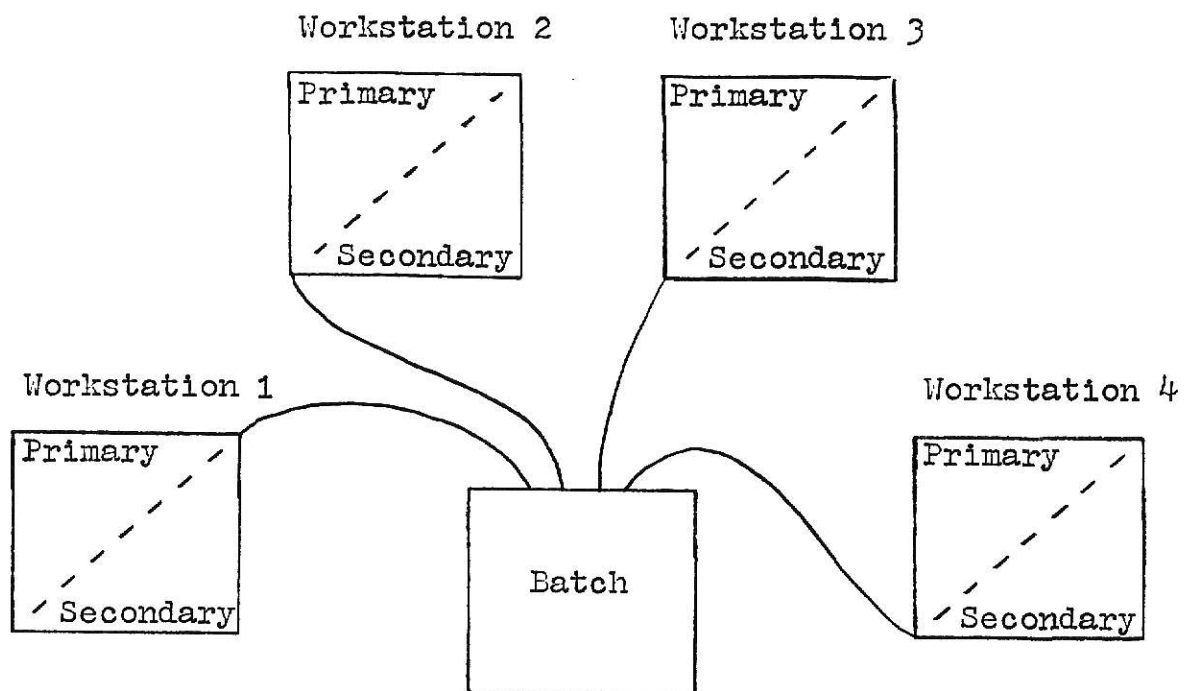
The various options available are configured into an operating system through a system generation process. A new software option provided as a result of the multiple writers feature is a sector lock timer value. This is the length of time in seconds that a process may be delayed while attempting to lock a sector that another process has locked.

3.1.2.2 Files

Disk logical I/O, which is part of the operating system, provides support for three types of logical file organizations and three types of record access methods. The components of disk logical I/O were listed in section 1.2.2.

3.1.2.2.1 Organizations

Figure 3.1.2.1-1 Processes in the System



- - - Secondary mode entered via Break-key Suspend

— Batch jobs placed on queue via Submit Command

The file organizations supported are sequential, relative and indexed.

Sequential files are characterized by the consecutive placement of records within the file. Sequential files may only be accessed sequentially, and the legal open modes for sequential files are open for input, open for output, open for input or output (I/O), and open for extension (to be defined in section 3.1.2.2.3). Records may be fixed or variable in length. If the records are fixed, they may be between 1 and 512 bytes; if variable, they may be from 1 to 510 bytes in length, excluding a 2 byte variable length index (VLI). Maximum blocksize is 512 bytes and the minimum is the record size for fixed length records, the maximum record size plus two for variable length records. There is one block per sector and records may be blocked (buffered) in memory during processing if the length of record (plus VLI) times 2 is less than the blocksize.

In relative files records are identified by their position in the file relative to the beginning of the file (relative record number). Relative files may be accessed by any of the three access methods, and the legal open modes are open for input, open for output, and open for I/O. Records may be fixed in length only. Record size is from 1 to 512 bytes and blocksize is from record size to 512. Records may be blocked if the blocksize is at least two times the record size. There is one block per sector.

Indexed files are characterized by the use of keys which identify the records and are part of the records. Keys are of fixed size and fixed offset within the records. Any of the three access methods may be used, and the legal open modes are open for input, open for output, and open for I/O. Records may be fixed or variable in length with the same size constraints as sequential files. The key size may be from 1 to 246 bytes in length but may not be longer than the record size. There is one block per sector and blocking may occur as in sequential files.

Indexed files differ from sequential and relative files in that a key index directory is contained after the data portion of the file. The actual data records are placed in the file in the order of their creation. The index directory is built at the end of the file, starting with the last sector. The data part and index part of the file are like two stacks growing toward each other. The index directory is created and maintained in ascending order according to the value of the keys.

3.1.2.2.2 Access Methods

Disk logical I/O supports these access methods: sequential, random and dynamic.

Sequential, relative and indexed files may all be accessed sequentially. If sequential and relative files are accessed sequentially, the records are read/written in the

order of their occurrence starting at the beginning of the file. In indexed files, records are read/written in ascending order by key value.

Random access is valid for relative or indexed files. Records are accessed by relative record number in the case of relative files and by key in the case of indexed files.

Dynamic access is also valid for relative or indexed files and is a combination of sequential access and random access. Records may be obtained by relative record number/key or they may be obtained by a "read next" which will return the next relative record in the file or the record associated with the next index key.

3.1.2.2.3 Open Modes

Two of the I/O functions are operations on files. They are "open" and "close". The other functions are performed to manipulate records. There are four mutually exclusive open modes: open for input, open for output, open for input/output (I/O), and open for extension.

Open for input allows the records in the file to be read. Records may not be written to the file while it is opened for input, and the file retains the same characteristics throughout processing.

Records may only be written to a file that is opened for output. File characteristics may be established or changed in this open mode. A file that is opened for output is con-

sidered empty at the beginning of processing. Any records that may have been contained in the file are over-written.

In the open for I/O mode, records may be read or written (for sequential files, read and rewrite are allowed, see figure 3.1.2.2.4-1). If the file is empty when opened for I/O, its characteristics may be changed. However, if the file is not empty, it retains the same characteristics throughout processing.

Open for extension is legal for sequential files only. Records may only be written and are added to an existing file. If the file is empty, it may be reorganized, otherwise it may not.

3.1.2.2.4 Functions

The I/O functions that may be performed on records include read, write, rewrite, delete, read next, and start. Figure 3.1.2.2.4-1 shows which functions are allowed within each access method for each file type and open mode. It should be noted that in the sequential access method, rewrite and delete must be preceded by a read of the record to be rewritten or deleted. Figure 3.1.2.2.4-2 is a list of the statuses that may be returned to applications after attempting to perform the various functions.

3.1.2.3 Data Structures

A number of data structures are used by the operating

Figure 3.1.2.2.4-2 List of Statuses

Successful completion/good status
End of data reached
Index file keys out of sequence
Duplicate key
No data record for given key
Record number exceeds actual number of keys
Permanent error
Boundary violation file extent reached
File not available or not assigned
File not closed
File not opened
Invalid operation
Invalid open
Invalid device
Record size outside minimum/maximum value
Insufficient memory available to perform function
Time out on sector lock

system to keep track of necessary information. Four are important to disk logical I/O. They are: file descriptors, I/O function control structures, general file information structures, and process file information structures.

File descriptors reside in the disk directory and contain such information as file name, location on disk, type of file, length of file, blocksize, and record size.

The application sets up the information in the I/O function control (IOFC) structures. The operating system uses this information to carry out the user's requests.

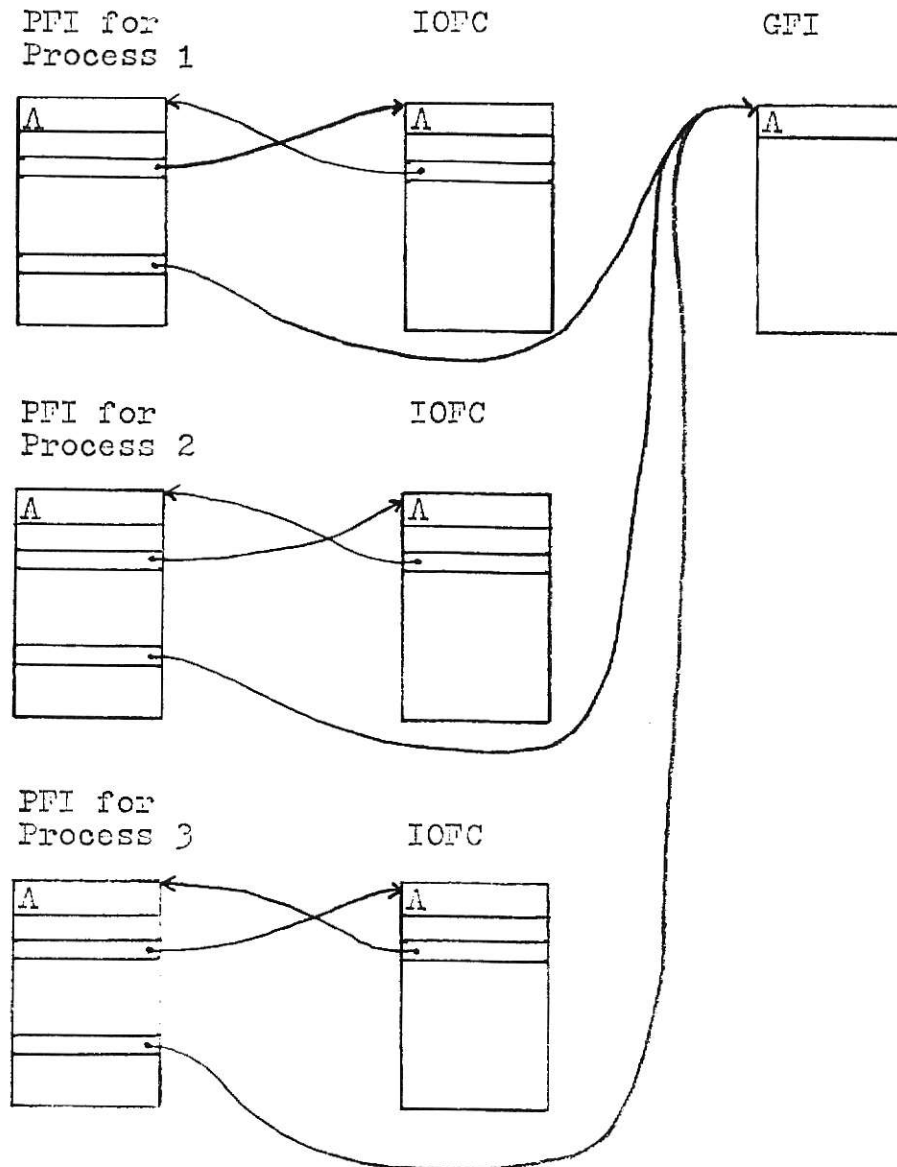
There is one general file information (GFI) structure in existence for each file that any process has assigned. The information in this structure is essential for providing file sharing among processes and is global to all processes.

Each file that a particular process has assigned has a corresponding process file information (PFI) structure. The information in this structure is readily available to a process and is necessary for carrying out user's requests. Pointers to the appropriate IOFC and GFI are contained in the PFI.

The pseudocode that follows is written with reference to these data structures. The structure is given first, followed by a period and the offset within the structure. Figure 3.1.2.3-1 shows the relationship among the IOFC, PFI, and GFI data structures.

Figure 3.1.2.3-1 Relationship Among Logical I/O Data Structures

Three processes have file A open:



3.1.3 Compatibility Issues

While the multiple writers feature was being designed and throughout its implementation, consideration was given to being compatible with other systems as described in section 1.1.3. Some of the requirements of the implementation that were determined by the goal of compatibility are:

- 1) Only one sector of a file may be locked by a process at any given time. When a process tries to lock a new sector, the old sector (if any) will be automatically unlocked.
- 2) A process may have a sector locked for each file it has assigned.
- 3) Processes assigning a file as read-only (used for a file that is being executed) or opening the file for input, will not lock sectors for that file but will honor sector locks of other processes. (The actions of checking for a locked sector and then performing the read are indivisible, therefore another process will not be able to lock the sector between the two actions.)
- 4) Processes that own a file will not lock sectors for the owned files. A process that opens a file for output must have the file assigned as owned.
- 5) The sector lock routine will issue a timer if a process attempts to lock a sector that is already locked by another process. Locks failing due to the

timer expiring will return a timeout status to the application. In other words, the process attempting to lock a sector is timed not the process that has the sector locked.

- 6) How long a process has a sector locked is not timed. The sector may be kept locked indefinitely.
- 7) Any time an error (status other than good) is returned to the application while processing a file, if the process has a sector of the file locked, it will be automatically unlocked.

3.2 Description of Implementation

3.2.1 Sector Lock and Unlock

The mechanism used to allow exclusive access to records was provided in a new disk logical I/O sub-module that contained routines to lock and unlock sectors. The actual resource locking/unlocking was done by routines existing in the supervisory part of the operating system with which disk logical I/O interfaced. If the sector was actually locked, the sector number was put into the PFI. If the sector was unlocked, zero was put in that field in the PFI. The PFI is not listed as a parameter because it is available to all disk logical I/O routines and its fields may be modified by any of the routines at any time. The logic involved in the lock and unlock routines is given below:

```
ROUTINE:                Sector Lock
INCOMING PARAMETERS:    Sector number to be locked
RESULT PARAMETERS:      Return status (good or timeout)
PSEUDOCODE:
BEGIN SECTOR-LOCK
    status := good
    IF PFI.shared file = true THEN
        IF PFI.read only = true OR IOFC.open mode = input THEN
            check lock status (sector number, locked status)
            WHILE locked status = locked by another process AND
                status = good DO
```

```

    delay lock (timer,status)
    IF status NOT = timeout THEN
        check lock status (sector number,locked status)
    ENDIF
ENDDO
ELSE
    IF sector number NOT = PFI.locked sector THEN
        IF PFI.locked sector NOT = 0 THEN
            unlock resource (PFI.sector locked)
        ENDIF
        lock resource (sector number,timer,status)
        IF status ≠ good THEN
            PFI.locked sector := sector number
        ENDIF
    ENDIF
ENDIF
ENDIF
RETURN
END SECTOR-LOCK

```

ROUTINE: Sector Unlock

INCOMING PARAMETERS: None

RESULT PARAMETERS: None

PSEUDOCODE:

BEGIN SECTOR-UNLOCK

```

    IF PFI.shared file = true AND (PFI.read only = false OR
        IOFC.open mode NOT= input) THEN

```



```
IF PFI.locked sector NOT = 0 THEN
  unlock resource (PFI.locked sector)
  PFI.locked sector := 0
ENDIF
ENDIF
RETURN
END SECTOR-UNLOCK
```

3.2.2 Impact on Functions

The incorporation of the multiple writers capability changed the logic of each file I/O function. Therefore, a description of each function will be provided, with particular attention given to the changes made. The functions will be presented according to file organization.

Critical sections are used to insure that the operator cannot interrupt the updating of critical control structures by pressing the break key. Break key is a special function that allows the operator to interrupt a process to abort a program, to enter secondary mode, or to perform other similar operations. The entering and exiting of critical sections will not be explicitly stated in the pseudocode.

As mentioned before, the PFI is available to all of the following routines and will, therefore, not be listed as a parameter. Access can be gained to the other data structures (IOFC and GFI) through the PFI. The pseudocode is greatly simplified to provide only the general logic of the functions.

3.2.2.1 Sequential Files

3.2.2.1.1 Open

The open routine checks organization, blocksize, record size and record type against the directory information to ensure that the user is opening the file properly. To prepare for the file being opened for extension by this process or a

later process, the last data sector of the file is read and the last record is located so that the system will know where to put new records. The logic for open is as follows:

ROUTINE: Sequential Open

INCOMING PARAMETERS: None

RESULT PARAMETERS: Status

PSEUDOCODE:

BEGIN SEQ-OPEN

status := good

check for legal open (status)

IF status = good THEN

set open flags (status)

GFI.open count := GFI.open count + 1

IF IOFC.open mode = output, I/O or extend THEN

GFI.writer count := GFI.writer count + 1

ENDIF

IF status = good THEN

IF GFI.open count = 1 AND IOFC.open mode NOT = output
THEN

read GFI.end of data sector - 1 (status)

IF status = good THEN

find last record in buffer

ENDIF

ENDIF

ENDIF

ENDIF

```

IF status = good THEN
  IF IOFC.open mode = extend AND
    GFI.end of data sector NOT= PFI.start sector THEN
    PFI.current sector := GFI.end of data sector - 1
    PFI.position in buffer := last record in buffer
  ELSE
    PFI.current sector := PFI.start sector - 1
    PFI.position in buffer := PFI.end of buffer
  ENDIF
  set up funtion table (status)
ENDIF
RETURN (status)
END SEQ-OPEN

```

The open routine does not lock sectors or honor locked sectors. The only purpose for the I/O operation that is performed is to set up pointers, and since the I/O operation is only done in the case of the first process opening the file, there should be no contention for the sector. The major impact to open due to the multiple writers feature is that the last data sector must be read and pointers set up by the first open, regardless of whether the open mode is for input, I/O, or extension, in case a later process opens the file for extension. (Open for output does not have to do the read since the file will be owned and therefore no other process may have access to it.) Prior to the implementation, only open for extension had to set up the pointers since only one

writer at a time was allowed to have the file opened.

3.2.2.1.2 Write

Sequential write became two routines: sequential write for owned files and sequential write for shared files. This was done because the logic for the two was quite different and to have combined them would have affected the performance of sequential write for owned files. Only the logic for sequential write for shared files will be presented here. That for owned files remained much as it had been. The write routine places the record which is in the user's area into the file in the next record position. Records are written immediately to the disk (whereas, if the file is owned, records are added to a memory buffer until the buffer becomes full and then are written to disk. This is known as blocking.).

ROUTINE: Sequential Write for Shared Files

INCOMING PARAMETERS: None

RESULT PARAMETERS: Status

PSEUDOCODE:

BEGIN SEQ-WRITE-SHARED

status := good

lock end of data sector (status)

IF status = good THEN

determine next sector to write (status) "updates the
PFI.current sector"

IF status = good THEN

```

IF PFI.current sector NOT = GFI.end of data sector THEN
    read PFI.current sector (status)
ELSE
    PFI.position in buffer := PFI.buffer address
ENDIF
IF status = good THEN
    move record from IOFC.user's area to buffer
    fill rest of buffer with delete characters
    write PFI.current sector (status)
    IF status = good THEN
        IF PFI.current sector = GFI.end of data sector THEN
            PFI.current sector := GFI.end of data sector
            GFI.end of data sector := GFI.end of data sector+1
        ENDIF
        PFI.position in buffer := PFI.position in buffer +
            IOFC.record length
    ENDIF
ENDIF
ENDIF
ENDIF
unlock sector
RETURN (status)
END SEQ-WRITE-SHARED

```

Sequential file writes are always performed at the end of data. Therefore, the end of data sector is the sector that must be locked. If two or more processes are writing to the

file, it is very possible that by the time a process gets the sector locked, it may no longer be the end of data sector. The logic, then, for the special "lock end of data sector" is as follows:

```
ROUTINE:                Lock End of Data Sector
INCOMING PARAMETERS:    None
RESULT PARAMETERS:      Status (good or timeout)
PSEUDOCODE:
BEGIN LOCK-EOD
    status := good
    WHILE PFI.sector locked NOT = GFI.end of data sector AND
        status = good DO
        sector := GFI.end of data sector
        lock sector (sector,status)
    ENDDO
    RETURN (status)
END LOCK-EOD
```

3.2.2.1.3 Read

Sequential read also became two routines for the same reasons as sequential write. The logic for sequential read for shared files will be presented. The read routine locates the next record in the file and moves it into the user's area. A sector is read only if it is not currently locked by the process (the sector is in a memory buffer already).

```
ROUTINE:                Sequential Read for Shared Files
```

```

INCOMING PARAMETERS:      None
RESULT PARAMETERS:        Status
PSEUDOCODE:
BEGIN SEQ-READ-SHARED

    status := good

    IF PFI.position in buffer = PFI.end of buffer THEN
        determine next sector to read (status)      "updates the
        PFI.current sector"
    ENDIF

    IF status = good THEN
        REPEAT
            IF PFI.current sector NOT = PFI.locked sector THEN
                lock sector (PFI.current sector,status)
                IF status = good THEN
                    read PFI.current sector (status)
                ENDIF
            IF status ≠ good THEN
                locate next record in block (flag)
                IF flag = found THEN
                    move record into IOFC.user's area (status)
                    IF status = good THEN
                        IF PFI.organization = relative THEN
                            increment relative record number
                        ENDIF
                    PFI.length of last record read := IOFC.
                        record length
                ENDIF
            ENDIF
        UNTIL status ≠ good
    ENDIF
END SEQ-READ-SHARED

```



```

        ENDIF
        RETURN (status)
    ELSE
        determine next sector to read (status)
    ENDIF
ENDIF
ENDIF
ENDIF
UNTIL status NOT = good
ENDIF
RETURN (status)
END SEQ-READ-SHARED

```

This routine is used for the sequential access method of both sequential and relative files. Therefore, deleted records could exist within the file and several sectors may have to be read before the next record in the file is found. It should be noted that the read function leaves the sector locked upon return. This is important for the implementation because sequential deletes and rewrites must be preceded by a read of the record and the sector must be locked until the delete or rewrite is completed. This could also cause sector contention problems for users since a process could read a record and keep the sector locked for any length of time.

3.2.2.1.4 Rewrite

Sequential rewrite replaces a record just read by the record in the user's area. The previous I/O operation must have been a read. The replacement record must be the same length as the old record.

ROUTINE: Sequential Rewrite

INCOMING PARAMETERS: None

RESULT PARAMETERS: Status

PSEUDOCODE:

BEGIN SEQ-REWRITE

status := good

IF PFI.length of last record read = 0 OR PFI.length of
last record read NOT = IOFC.record length THEN

status := invalid operation

ENDIF

IF status = good AND PFI.shared file = true THEN

IF PFI.current sector NOT = PFI.locked sector THEN

status := invalid operation

ENDIF

ENDIF

IF status = good THEN

IF PFI.record length = variable THEN

IOFC.record length := IOFC.record length + 2

ENDIF

PFI.position in buffer := PFI.position in buffer -

IOFC.record length

IF PFI.shared file = true THEN

```

compare record in IOFC.user's area to record in buffer
IF equal THEN
    PFI.length of last record read := 0
    unlock sector
    PFI.position in buffer := PFI.position in buffer +
        IOFC.record length
ELSE
    move record from IOFC.user's area to buffer
    write PFI.current sector (status)
    unlock sector
ENDIF
ELSE
    move record from IOFC.user's area to buffer
    PFI.buffer changed := true        "used for blocking"
ENDIF
ENDIF
RETURN (status)
END SEQ-REWRITE

```

This routine is also used for both sequential and relative files in the sequential access method. While the design of multiple writers was taking place, another compatibility issue was brought to our attention. Programs written for another operating system were doing rewrites after reads quite often for the sole purpose of unlocking sectors. No data was being changed, the programs simply wanted to know what a record contained and did not want to

tie up the sector. The other operating system was asked to make a change to rewrite, to determine if a record had changed before writing it to the disk and thereby avoid doing unnecessary I/O operations, thus improving performance. We, therefore, made the same change in our operating system.

3.2.2.1.5 Close

The close routine makes sure no I/O for the file is outstanding. It releases buffer memory. The appropriate flags are changed to indicate that the file is closed. The file descriptor in the directory is updated if needed. The only change to the close routine was that if the process had a sector locked for the file, the sector was unlocked. Therefore, the logic for close will not be included in the report.

3.2.2.2 Relative Files

3.2.2.2.1 Open

Like sequential open, relative open checks file organization and record type. Relative open does not need to do a read I/O operation to find the last record in the file because relative files cannot be opened for extension. The logic for open is:

```
ROUTINE:                Relative Open
INCOMING PARAMETERS:    None
RESULT PARAMETERS:      Status
PSEUDOCODE:
BEGIN REL-OPEN
    status := good
    IF IOFC.open mode = extend OR IOFC.record length =
        variable THEN
        status := invalid open
    ENDIF
    IF status = good THEN
        check for legal open (status)
        IF status = good THEN
            set open flags (status)
            GFI.open count := GFI.open count + 1
            IF IOFC.open mode = output, or I/O THEN
                GFI.writer count := GFI.writer count + 1
            ENDIF
        ENDIF
    ENDIF
```

```

IF status = good THEN
    PFI.current sector := PFI.start sector - 1
    PFI.position in buffer := PFI.end of buffer
    PFI.records per block := IOFC.blocksize/IOFC.
        record size
    set up function table (status)
    IF status = good THEN
        PFI.relative key := 0
    ENDIF
ENDIF
ENDIF
ENDIF
ENDIF
RETURN (status)
END REL-OPEN

```

The changes to relative open to implement multiple writers consisted in removing code to prevent the open if the file was already opened by a writer. Instead, the writer flag became a counter. These same changes were made in sequential and indexed open routines. It should be noted that if a process has a file opened for output, no other process will be able to assign (access) the file.

3.2.2.2.2 Write

Relative random write locates a record's position by key number and moves the record from the user's area into the buffer. Records are written immediately to disk. No

record may already exist at the record's position in the file prior to the write.

ROUTINE: Relative Random (and Dynamic) Write

INCOMING PARAMETERS: None

RESULT PARAMETERS: Status

PSEUDOCODE:

BEGIN REL-RAN-WRITE

locate relative record position (status,in memory flag,
sector number)

IF status NOT = invalid key THEN

IF status ≠ beyond EOD THEN

random write to EOD (status,flag,in memory flag,
sector number)

IF flag = handled as EOD THEN

RETURN (status)

ENDIF

ENDIF

ENDIF

IF status = good THEN

IF in memory flag = false THEN

lock sector (sector number,status)

IF status = good THEN

read sector number (status)

ENDIF

ENDIF

IF status = good THEN

```

    check for deleted record (status)
  IF status = good THEN
    check for legal record size (status)
    IF status = good THEN
      move record from IOFC.user's area to buffer
      write sector number (status)
    ENDIF
  ENDIF
ENDIF
ENDIF
ENDIF
END REL-RAN-WRITE

```

The logic of two of the routines called by this routine should be presented for the sake of clarification. One of the routines, "locate relative record position", will be used by the other relative random routines. The other routine, "random write to EOD", was separated from random write to make the routines more modular and easier to understand.

```

ROUTINE:          Locate Relative Record Position
INCOMING PARAMETERS:  None
RESULT PARAMETERS:   Status, sector number if found,
                     in memory flag

```

PSEUDOCODE:

```

BEGIN LOC-REC

```

```

  status := good

```

```

  IF PFI.records per block > 1 THEN

```

```

    IF PFI.records per block < IOFC.relative key THEN

```



```
    quotient := IOFC.relative key/PFI.records per block
    remainder := modulo (IOFC.relative key/PFI.records
        per block)
    IF remainder = 0 THEN
        quotient := quotient - 1
        remainder := PFI.records per block
    ENDIF
ELSE
    quotient := 0
    remainder := IOFC.relative key
ENDIF
remainder := remainder - 1
PFI.position in buffer := PFI.buffer address +
    (remainder * IOFC.record size)
ELSE
    quotient := IOFC.relative key - 1
    PFI.position in buffer := PFI.buffer address
ENDIF
sector number := PFI.start sector + quotient
PFI.relative key := IOFC.relative key
IF sector number = PFI.current sector THEN
    IF PFI.shared file = true THEN
        IF PFI.current sector = PFI.locked sector THEN
            in memory flag := true
        ENDIF
    ELSE
```

```

        in memory flag := true
    ENDIF
ELSE
    IF sector number ≥ GFI.end of data sector THEN
        status := beyond EOD
    ENDIF
    IF sector number ≥ PFI.end of file sector THEN
        status := beyond EOF
    ENDIF
    IF status = good THEN
        PFI.current sector := sector number
        in memory flag := false
    ENDIF
ENDIF
RETURN (status,sector number,in memory flag)
END LOC-REC

ROUTINE:                Random Write to EOD
INCOMING PARAMETERS:    Sector number, status
RESULT PARAMETERS:      Status, flag, in memory flag,
                        sector number

PSEUDOCODE:
BEGIN RAN-WRITE-EOD
    flag := not handled as EOD
    WHILE status = beyond EOD DO
        status := good
        PFI.current sector := sector number
    
```

```

sector number := GFI.end of data sector
IF PFI.shared file = true THEN
    lock end of data sector (status)
ENDIF
IF status = good THEN
    IF PFI.shared file = false OR sector number =
        PFI.locked sector THEN
        fill from EOD sector (GFI.end of data sector) to
            PFI.current sector with delete characters
        check for legal record size (status)
        flag := handled as EOD
        IF status = good THEN
            move record from IOFC.user's area to buffer
            write PFI.current sector (status)
            GFI.end of data sector := PFI.current sector + 1
            unlock sector
        ENDIF
    ELSE
        locate relative record position (status,in memory
            flag, sector number)
    ENDIF
ENDIF
sector number := PFI.current sector
ENDDO
RETURN (status,flag,in memory flag,sector number)
END RAN-WRITE-EOD

```

There were two major changes to relative random write as a result of the implementation of multiple writers. First of all, if the file was shared, the sector had to be read into memory if it was not already in memory (not a change) or if it was already in memory but the sector was not locked by the process (change). Secondly, if the record position was found to be past the end of data, by the time the process was able to get the end of data sector locked, the end of data sector could have been changed by another process and the record position may or may not be past the new end of data sector. These conditions were handled in the implementation and described in the pseudocode.

3.2.2.2.3 Read

Relative random read locates a record by key number and moves it into the user's area. The routine ensures that the record exists.

ROUTINE: Relative Random (and Dynamic) Read

INCOMING PARAMETERS: None

RESULT PARAMETERS: Status

PSEUDOCODE:

BEGIN REL-RAN-READ

 locate relative record position (status,in memory flag,
 sector number)

 IF status = beyond EOD THEN

 status := invalid key

```

ENDIF
IF status = good THEN
    IF in memory flag = false THEN
        lock sector (PFI.current sector,status)
        IF status = good THEN
            read PFI.current sector (status)
        ENDIF
    ENDIF
ENDIF
ENDIF
IF status = good THEN
    check for deleted record (status)
    IF status = good THEN
        move record from buffer to IOFC.user's area (status)
    ENDIF
ENDIF
RETURN (status)
END REL-RAN-READ

```

Read, as mentioned before, does not unlock the sector upon return. The only change to the read routine was to require that the sector be locked and read into memory if the file was shared and the sector was not already locked and in memory. Note that if the file is read-only or opened for input, read does not lock sectors but honors sectors locked by other processes, hence each read will require an I/O operation if the file is shared.

The read next function is the same as sequential read

which was described in section 3.2.2.1.3.

3.2.2.2.4 Rewrite

Relative random rewrite locates the record by key number and replaces it with the record in the user's area. The record must exist in the file in order to be rewritten.

ROUTINE: Relative Random (and Dynamic) Rewrite

INCOMING PARAMETERS: None

RESULT PARAMETERS: Status

PSEUDOCODE:

BEGIN REL-RAN-REWRITE

locate relative record position (status,in memory flag,
sector number)

IF status = beyond EOD THEN

status := invalid key

ENDIF

IF status = good THEN

IF in memory flag = false THEN

lock sector (PFI.current sector,status)

read PFI.current sector (status)

ENDIF

ENDIF

IF status = good THEN

check for deleted record (status)

IF status = good THEN

IF PFI.shared file = true THEN

```
compare record in buffer to record in IOFC.user's
  area
IF equal THEN
  unlock sector
ELSE
  check for legal record size (status)
  IF status = good THEN
    move record from IOFC.user's area to buffer
    write PFI.current sector (status)
    unlock sector
  ENDIF
ENDIF
ELSE
  check for legal record size (status)
  IF status = good THEN
    move record from IOFC.user's area to buffer
    write PFI.current sector (status)
  ENDIF
ENDIF
ENDIF
RETURN (status)
END REL-RAN-REWRITE
```

The rewrite routine includes the change requiring the sector to be locked and also the rewrite compatibility change discussed in section 3.2.2.1.4. Random rewrite,

unlike sequential rewrite, does not require the previous function to have been a read of the record.

3.2.2.2.5 Delete

Both relative random delete and relative sequential delete will be described in this section. The record to be deleted is located by key number in the case of random delete. Sequential delete requires that the previous operation be a read of the record to be deleted. In both routines the record must exist to be deleted.

ROUTINE: Relative random (and Dynamic) Delete

INCOMING PARAMETERS: None

RESULT PARAMETERS: Status

PSEUDOCODE:

BEGIN REL-RAN-DELETE

locate relative record position (status,in memory flag,
sector number)

IF status = beyond EOD THEN

status := invalid key

ENDIF

IF status = good THEN

IF in memory flag = false THEN

lock sector (sector number,status)

IF status = good THEN

read PFI.current sector (status)

ENDIF

ENDIF


```

ENDIF
IF status = good THEN
    check for deleted record (status)
    IF status = good THEN
        fill record in buffer with delete characters
        write PFI.current sector (status)
    ENDIF
ENDIF
unlock sector
RETURN (status)
END REL-RAN-DELETE

ROUTINE:                Relative Sequential Delete
INCOMING PARAMETERS:    None
RESULT PARAMETERS:      Status
PSEUDOCODE:
BEGIN REL-SEQ-DELETE
    status := good
    IF PFI.length of last record read = 0 THEN
        status := invalid operation
    ENDIF
    IF PFI.shared file = true THEN
        IF PFI.current sector NOT = PFI.locked sector THEN
            status := invalid operation
        ENDIF
    ENDIF
    IF status = good THEN

```

```

fill record in buffer with delete characters
PFI.buffer changed := true          "used for blocking"
PFI.length of last record read := 0
IF PFI.shared file = true THEN
    write PFI.current sector (status)
    unlock sector
ENDIF
ENDIF
RETURN (status)
END REL-SEQ-DELETE

```

Both delete routines require that the sector be locked if the file is shared and both unlock the sector upon return.

3.2.2.2.6 Start

Start is valid for a user of a relative file in sequential or dynamic access modes. The file pointer is positioned to the record specified by the relative key number. The only change made to relative start was if the process had a sector locked for the file, it was unlocked. Therefore, the pseudocode will not be included.

3.2.2.2.7 Close

Relative file close is the same routine used for sequential files. See section 3.2.2.1.5.

3.2.2.3 Indexed Files

Many of the implementation considerations for indexed files have already been encountered in the discussion of sequential and relative files. For this reason, and because much of the code for indexed files involves maintaining the index (which is not the topic of the report), the description of indexed file functions will be very brief and the pseudocode presented will be even more simplified than that given for sequential and relative files. Open and close will not be discussed because the changes are similar to those described in the sections on relative and sequential files.

3.2.2.3.1 Write

Write is a valid function for indexed files in all three access methods. Whenever the index is manipulated, the file is locked, which means that any other process wishing to access the file must wait. This was not new, that is, it was not a result of the implementation of multiple writers. However, locking the file during processing did somewhat aid in the implementation of the new feature. The logic for write, in general, is:

Indexed Sequential Access Write:

no sectors are locked because the file must be
owned

Indexed Random and Dynamic Write:

```

    IF new end of data OF end of data not locked THEN
        lock end of data sector
    ENDIF
    lock file
    search index
    write key
    move record from user's area to buffer
    write sector
    write high level index
    unlock file
    unlock sector

```

3.2.2.3.2 Read

Read is also a valid function in all access methods.

Indexed Sequential Read and Dynamic Read Next:

```

    lock file
    search index
    unlock file
    IF new sector THEN
        lock sector
        read sector
    ENDIF
    move record from buffer to user's area

```

Indexed Random and Dynamic Read:

```

    lock file

```

```

search index
unlock file
IF new sector OR sector not locked THEN
    lock sector
    read sector
ENDIF
move record from buffer to user's area

```

3.2.2.3.3 Rewrite

As in the case of sequential rewrite, indexed sequential access rewrite requires that the previous operation be a read of the record. In addition, the compatibility issue described in sections 3.2.2.1.4 and 3.2.2.2.4 also affected indexed rewrite but will not be specifically depicted in the brief pseudocode.

Indexed Sequential Access Rewrite:

```

move record from user's area to buffer
write sector
unlock sector

```

Indexed Random and Dynamic Rewrite:

```

lock file
search index
IF new sector OR sector not locked THEN
    lock sector
    read sector
ENDIF

```

```

move record from user's area to buffer
write sector
unlock file
unlock sector

```

3.2.2.3.4 Delete

Delete is similar to rewrite except that the index key must be deleted as well as the record.

Indexed Sequential Access Delete:

```

lock file
search index
delete key
read sector
delete record in buffer
write sector
unlock file
unlock sector

```

Indexed Random and Dynamic Delete:

```

lock file
search index
IF new sector OR sector not locked THEN
    lock sector
ENDIF
delete key
read sector
delete record in buffer

```

- write sector
- unlock file
- unlock sector

3.2.2.3.5 Start

Start is valid only in sequential and dynamic access methods.

Indexed Start:

- unlock sector
- lock file
- search index
- unlock file

3.2.3 File Sharing Problem Resolution

3.2.3.1 File Access Control

When a new file is created in the operating system environment, it may be given the characteristic of "private". This is a permanent characteristic and means that any process assigning the file must assign the file as "owned". "Owned" is an attribute that may be specified when a file is assigned and it means that while a process has the file assigned, no other process may assign the file. A file that is assigned new (created by the assign) is assigned "owned" unless specified otherwise. In addition, for the sake of compatibility with other systems, a process that opens a file for output must have the file assigned as "owned".

The requirement that a file opened for output be assigned "owned" had not been in existence until the addition of the multiple writers feature. Therefore, it created an incompatibility with previous issues of the operating system. Previously valid procedure files would not work in the new system. (Defaulting the own/share option on the assign was interpreted by the system as "shared" originally.) To resolve the conflict, if the own/share option was defaulted and the file was opened for output by the process, the default would be interpreted by the system as "owned" providing no other process had the file assigned. If another process has the file assigned, the process attempting to

open for output would receive an "invalid open" status. This does not entirely solve the problem, of course. To guarantee that a file may be opened for output, the operator should assign the file as "owned".

3.2.3.2 Sector Locking

Most of the implementation decisions concerning the locking mechanism were dictated by compatibility issues (see section 3.1.3). Addressing specifically the questions raised in section 2.3.2, the implementation required:

- 1) only one lock per file may be in place for a process at any given time,
- 2) the amount of data to be locked is one disk sector which corresponds to one block of data,
- 3) a process may have a lock in place for any length of time,
- 4) resource contention is detected by the use of a timer in the sector locking mechanism. A timeout status is returned to the application if the process is unable to lock a sector because another process has the sector locked and the timer expired. Deadlock, in the true sense of the word, does not occur because processes are limited to only one lock per file and receiving a timeout status will cause the sector, if any, that the process has locked to be unlocked. However, a condition of "deadly embrace" can occur if an application

stubbornly attempts to perform a function that tries to lock a sector that another process keeps locked. The avoidance of deadly embrace is the responsibility of the user.

- 5) as mentioned above, requests to lock sectors are blocked if the sectors are locked by other processes. If a timer expires while a process is waiting to lock the sector, then a timeout status is returned to the application.
- 6) the value of the timer is established when the user's operating system is configured and generated.

The impact of sector locking on the functions of shared files is summarized in figure 3.2.3.2-1.

3.2.3.3 Data Integrity

The method used by the operating system to inform the user that a file may be unreliable is very primitive. When a file is opened by any process, a flag is set in the directory entry corresponding to the file and remains set until the last process closes the file. If a system failure occurs while the file is open, the flag remains on. No process may assign the file again until the file is CHECKED. CHECK is a system utility which allows the open flag to be reset. In addition, while a file is being processed, if a permanent or hardware error is encountered, no further functions may be performed by any process on the file except the

Figure 3.2.3.2-1 Summary of Sector Lock Impact

| <u>FUNCTION</u> | <u>SEQUENTIAL ACCESS</u> | <u>RANDOM/DYNAMIC ACCESS</u> |
|-----------------|---|---|
| open | no sector locking | no sector locking |
| write | lock EOD sector process unlock sector | lock sector process unlock sector |
| read | lock sector process | lock sector process |
| rewrite | process unlock sector | lock sector process unlock sector |
| delete | process unlock sector | lock sector process unlock sector |
| read next | not valid | lock sector process |
| start | unlock sector process | unlock sector process |
| close | unlock sector process | unlock sector process |

close function. When the file is closed after such an error, the open flag will remain set and the file must be CHECKed before it can be assigned again. Any time access to a file is prevented because of the open flag, the file should be investigated for any possible corruption and rebuilt if necessary, using system utilities provided for that purpose.

3.2.4 Testing

Each of the operations affected by the implementation were tested after coding was completed. Some minor design errors were found and corrected during testing. The design presented in the report contains those corrections. Testing was accomplished primarily with the use of a program written in 8080 assembly language and another written in COBOL. These programs had been designed specifically to test I/O functions. In addition, a customer's application program was borrowed and used in testing. Performance changes due to the new feature were measured.

4.0 Conclusion

4.1 Advantages and Disadvantages

A number of advantages in having file sharing capabilities were given in section 2.1.1. The major disadvantage is that changes that must be made to an operating system to allow multiple readers/writers affect performance from the viewpoint of a single user. If a user has exclusive access to a file, then portions of the file do not have to be locked and operations do not have to be regulated by the system. However, if a user is allowed to share a file with other users, then some overhead has to be added to the system to allow this capability. More I/O operations may have to take place to insure that the

user is working with the latest information, and the user may have to deal with some resource contention.

The reported implementation of file sharing is somewhat restrictive. These restrictions make the system straightforward and easily understood but are also limiting. The major restriction is that a file may not be shared in the open for output mode. For sequential files, the user can get around the restriction by opening files for extension rather than for output. Multiple writers may share a relative or indexed file by accessing it in random or dynamic access methods.

The requirement that the read operation honor sectors locked by other processes in the case of the file being read-only or opened for input, may have been unnecessary and affects performance very much. It was included in the implementation to be compatible with other operating systems.

4.2 Alternatives

The trend in the computer industry is away from batch processing and toward multiple-operator, interactive processing. Because of this trend, it is likely that file sharing in one form or another will be included in many future systems.

As indicated in the review of literature, the various aspects of file sharing may be implemented in a number of ways. One implementation has been described in the report.

It is a rather primitive implementation but does allow the sharing of files by multiple writers. The file sharing feature has passed a minimal amount of testing and is now being tested for certification by NCR's quality assurance staff.

Two implementation alternatives that have already been mentioned are: 1) allowing a process to place multiple locks on a file, and 2) making the user responsible for data integrity by providing record locking/unlocking operations in the user language. The first alternative is useful when the file is a large data base but has the disadvantage that deadlocks must be handled. The second alternative simplifies the operating system but makes user applications more complex. In addition, the compiler/assembler would have to make certain that write commands were preceded by lock commands.

BIBLIOGRAPHY

- [1] Brinch Hansen, P., Operating System Principles, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1973.
- [2] Brinch Hansen, P., The Architecture of Concurrent Programs, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1977.
- [3] Courtois, P.J., Heymans, F., and Parnas, D.L., Concurrent control with "readers" and "writers". Communications of the ACM 14,10, October 1971, 667-668.
- [4] Freeman, D.E. and Perry, O.R., I/O Design: Data Management in Operating Systems, Hayden Book Company, Inc., Rochelle Park, New Jersey, 1977.
- [5] Habermann, A.N., Introduction to Operating System Design, Science Research Associates, Inc., Chicago, Illinois, 1976.
- [6] Holt, R.C., Graham, G.S., Lazowska, E.D., and Scott, M.A., Structured Concurrent Programming with Operating Systems Applications, Addison-Wesley Publishing Company, Reading, Massachusetts, 1978.
- [7] Lycklama, H. and Bayer, D.L., UNIX Time-Sharing System: The MERT operating system. The Bell System Technical Journal 57,6, July 1978, 2049-2086.
- [8] Lorin, H., Parallelism in Hardware and Software: Real and Apparent Concurrency, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1972.
- [9] Madnick, S.E. and Donovan, J.J., Operating Systems, McGraw-Hill Book Company, New York, New York, 1974.
- [10] Organick, E.I., The Multics System: An Examination of its Structure, The MIT Press, Cambridge, Massachusetts, 1972.
- [11] Paxton, W.H., A client-based transaction system to maintain data integrity. Proceedings of the Seventh Symposium on Operating Systems Principles, The Association for Computing Machinery, Inc., December 1979, 18-23.

- [12] Shaw, A.C., The Logical Design of Operating Systems, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1974.
- [13] Swinehart, D., McDaniel, G., and Boggs, D., WFS: A simple shared file system for a distributed environment. Proceedings of the Seventh Symposium on Operating Systems Principles, The Association for Computing Machinery, Inc., December 1979, 9-17.
- [14] ———, Texas Instruments DS990 Commercial Computer Systems: General Information, Texas Instruments, Inc., Houston, Texas, 1978.
- [15] Tsichritzis, D.C. and Lochovsky, F.H., Data Base Management Systems, Academic Press, New York, New York, 1977.

FILE SHARING:
AN IMPLEMENTATION OF THE MULTIPLE WRITERS FEATURE

by

MARY KENNEY
B.A., Wichita State University, 1974

AN ABSTRACT OF A MASTER'S REPORT
submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1981

ABSTRACT

Aspects of file sharing are discussed, particularly the aspect of allowing more than one writer access to a file concurrently. File sharing has been implemented in various degrees and in a variety of environments, however certain problems are common to file sharing implementations and must be resolved in one way or another. Some implementations and file sharing problems and problem resolutions are described. In addition, the incorporation of the multiple writers feature into an existing operating system for a multiple workstation microprocessor is presented in detail. The design changes made to each function are included. How typical file sharing problems were resolved is addressed as well as the resolution of problems encountered during design and implementation of file sharing in this particular system.