ESD-TR-71-221

MTR-2052

# CONCURRENT DATA SHARING PROBLEMS IN MULTIPLE USER COMPUTER SYSTEMS

J. B. Glore
L. B. Collins
J. K. Millen

JULY 1971

Prepared for

DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS
ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
UNITED STATES AIR FORCE
L. G. Hanscom Field, Bedford, Massachusetts

Approved for public release; distribution unlimited.

Project 5550
Prepared by
THE MITRE CORPORATION
Bedford, Massachusetts
Contract F19(628)-71-C-0002

AD731752

ESD-TR-71-221

MTR-2052

# CONCURRENT DATA SHARING PROBLEMS IN MULTIPLE USER COMPUTER SYSTEMS

J. B. Glore
L. B. Collins
J. K. Millen

JULY 1971

Prepared for

DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS
ELECTRONIC SYSTEMS DIVISION
AIR FORCE SYSTEMS COMMAND
UNITED STATES AIR FORCE
L. G. Hanscom Field, Bedford, Massachusetts

Project 5550
Prepared by
THE MITRE CORPORATION
Bedford, Massachusetts
Contract F19(628)-71-C-0002

## FOREWORD

This report presents the results of analyses conducted by The MITRE Corporation, Bedford, Massachusetts in support of Project 5550 under contract F19(628)-71-C-0002. Dr. John B. Goodenough (ESD/MCDT-1) was the ESD Project Monitor. The report provides a technical baseline guiding further research in this area; additional reports on this topic will be published in the future.

## REVIEW AND APPROVAL

This technical report has been reviewed and is approved.

EDMUND P. GAINES, JR., Colonel, USAF
Director, Systems Design & Development
Deputy for Command and Management Systems

## ABSTRACT

This report summarizes work performed to date under the FY'71 Project 6710 Multi-User Data Management System task. It reviews major problems associated with the sharing of data among multiple concurrent users, and tentatively suggests promising strategies to cope with them. It discusses the importance of solving, ameliorating, or avoiding such problems to the effective development of Air Force systems of this kind. It also outlines desirable future work under this task.

## ACKNOWLEDGMENTS

Several persons have contributed substantially to the development of this report. Dr. Millen and Mr. Collins provided material for Section IV; Mrs. J. A. Clapp revised it. Mr. Collins drafted Section III and contributed to Section I. Mr. Glore wrote Section II and completed the other sections. Mrs. Clapp, Mr. C. M. Sheehan, and Mr. E. W. Williamson critically reviewed the draft. Miss Dorothy Statkus and Miss Cynthia Smith typed and proofread the original material and its many revisions.

TABLE OF CONTENTS

TABLE OF CONTENTS (Concluded)

## LIST OF ILLUSTRATIONS

# SECTION I

# INTRODUCTION

## OBJECTIVES

This report of work performed under the FY'71 Project 6710 Multi-User Data Management Systems Task has three prime goals. The first is to outline the major problems associated with the sharing of data among several concurrent users and processes in multiple user computer systems; e.g., multiprogramming systems executing computer programs concurrently for more than one user. The second main goal is to show the importance of solving, ameliorating, or at least avoiding these problems in Air Force systems of this kind. The third basic objective is to sketch desirable future work to refine further these problems' definition, to examine their implications, to develop alternate solutions to them, and to evaluate these solutions.

## PROBLEM OVERVIEW

Informal explanations and illustrations of concurrent data sharing and its problems are included here to introduce these somewhat unfamiliar concepts to the many persons who have yet to encounter them. More thorough treatment, including definition of certain essential terms, may be found in Section II, entitled TECHNICAL ANALYSIS. The use of certain standard information processing vocabularies(1)(2) is suggested to clarify technical terminology not explicitly defined in this report.

Data are shared whenever two or more persons or processes access them. (A typical process is a sequence of computer program instruction executions or other machine operations.) There is nothing new about data sharing as such. Data have been shared more or less successfully for thousands of years. More recently, but prior to the introduction of computers, satisfactory manual and semi-automatic information-handling systems involving shared data were routinely worked out, installed, and operated in many business and government offices. Such common data were used in part to communicate among these systems' segments. These systems' principles, somewhat modified, were adapted to early computer applications, especially in business data processing. The same principles are embodied today in thousands of successful batch processing computer applications that routinely produce a variety of files and reports shared by many users.

These obvious successes tend to cast doubt on data sharing problems as serious hazards in computer application design. Yet, such difficulties can occur, causing trouble when they do, especially (but not exclusively) in systems that allow processes to share data <u>concurrently</u>; i.e., without strict time separation of all occasions of use.

## Characteristics of Vulnerable Computer Systems

The past incidence of concurrent data sharing problems in operational computer systems has been slight, for reasons discussed in Section III. There is now a trend, however, both within and outside the Air Force, toward development or acquisition of computer systems especially vulnerable to such problems; e.g., AABNCP, MACIMS. These susceptible systems typically contain related groups of large and dynamically changing files often updated by real time inputs and frequently referenced by several concurrent users at on-line terminals. Such users <u>cooperate</u> through rapid interaction with common data. Sometimes a user may need only to insert a few values or obtain a few facts from a shared data base. On other occasions he may perform complex and time-consuming analyses, interacting with special algorithms and data shared with others in heuristic problem-solving attempts; e.g., in tactical planning.

In such a system no user's action may be allowed to garble shared data or cause another user to obtain incorrect results. Nor may the actions of one or more users excessively degrade the system's response time to other users' requests. As a rule system response time requirements preclude the relatively safe sequential and batch approaches to change and information request processing. Instead, highly concurrent access by several processes to common data may be needed to meet response time requirements. As this report shows, the major data sharing problems result from failure to regulate these conflicting requirements.

Types of computer systems subject to one or more data sharing problems are identified in the subsection below entitled TYPES OF SUSCEPTIBLE SYSTEMS, and discussed more fully in Section III. First, however, we briefly classify and illustrate the major problems themselves.

## Classification of Shared Data Problems

Shared data problems can be divided into _system error_, _system performance_, and _user procedure_ problems. System error problems can be broken down further into (1) those that garble the data base, (2) those causing no data base damage but which may cause erroneous output, and (3) those that permanently prevent certain processes from proceeding. System performance problems yield no wrong results but impair response times. User procedure problems afflict the computer system's operational users even though the computer system itself may be said to operate without error and performs acceptably. Each group of problems is illustrated below.

## System Error Problems

Two or more processes may alter a common field inconsistently; e.g., when a second process' action begins based on the field's old value before another's modification is complete. Similarly a related set of fields, e.g., fields of a conventional serial file and of indexes that refer to them, can be changed inconsistently if all significant results of one process' actions are not recorded before another refers to them. If the second process obtains portions of an index before the first has revised and replaced them, the second may either change them wrongly or use them to reference the wrong serial file field. Also, the first process may overwrite the index as adjusted by the second, effectively erasing the record of the second process' actions. At its worst, such interference can not only pollute the data base, but can also cause a process to destroy itself or others if it uses inconsistent control information to determine its behavior.

The sets of fields that are implicitly related in updating even a simple file can be surprisingly large. For instance, insertion or deletion of a record may entail adjustment of a record count or displacement of other records within a block in storage. The count and other records in the block must therefore be considered part of the related set.

Limiting to but one process at a time the right to update data can prevent garbling, although system throughput usually suffers in consequence. However, one or more processes allowed to read data that one other process is concurrently updating may still obtain wrong or inconsistent results, even though the updating is done correctly. For example, a reader might fetch a single field's contents either before or after its alteration, without knowing which state of the field the value obtained represented. Similarly, a summary prepared during updating is likely to be time-inconsistent, including some values read before modification and some read afterward.

Inappropriate access-limiting algorithms can cause deadlock. In its simplest form, deadlock occurs when each of two routines has been allowed exclusive use of a resource (e.g., a subset of common data) needed by the other to continue execution. For example, deadlock will occur in the following circumstances. Suppose process 1, updating a file of two records, A and B, first obtains and locks A and then requests B (e.g., of the operating system). Further, suppose process 2, concurrently summarizing the same file, has obtained and locked B and then requests A. Neither can continue, since each needs data locked by the other.

## System Performance Problems

Prohibiting both multiple concurrent updating and all concurrent reading during updating can prevent inconsistent results. These restrictions may unacceptably delay processing of important information requests, however. Even multiple concurrent reading without updating can impair system performance (although it more often permits improved performance).

Access control techniques devised to prevent system error problems tend to cause system performance problems. The locks require storage space, and their use entails otherwise unnecessary storage accesses. Inappropriate computer processor design may entail clumsy programmed lockout algorithms. Since locks themselves are prime examples of shared data, special computer instructions may be needed to inspect and set a lock indivisibly so that no other programs can interfere before the first program can act on it. Most programmed lockout strategies actually used also tend to impair system performance by constraining the possible concurrent operations allowed.

## User Procedure Problems

A multiple user computer system that consistently yields results correct according to specifications and within prescribed response time limits may still suffer from user procedure problems. Confusion resulting from multiple concurrent use that would not occur during sequential or batch processing typifies such problems. For example, two agents using an on-line airline reservation system might try to book the last seat on the same flight after each had just requested seat availability and had been notified, correctly, that one seat remained open. One agent's booking request, however, would be rebuffed causing user inconvenience and unnecessary computer processing. The systems's action would be correct according to specifications, although one might argue that the specifications, and the computer program, should be changed.

Again, after a typical period of multiprogrammed computer operation, badly disciplined data sharing might combine with other factors to erase the exact history of significant events, provided each input was not logged, tagged with source and time of entry. Such confusion could prevent reconstruction of information needed for debugging, audit, or system recovery. Consequent inability to pinpoint errors could ultimately destroy user confidence in system operation.

Consequent inability to recover correctly from system failure could render system use infeasible. Yet the logged information might be quite adequate to rerun a sequential or batch processing system.

## TYPES OF SUSCEPTIBLE SYSTEMS

At least three groups of computer programs that need to share data on a highly concurrent basis may be distinguished. The first comprises application-specific programs, such as the central on-line routines of certain military command control systems, airline reservation systems, and air traffic control systems. The second group consist of much-desired on-line multi-user generalized data management systems. Multiprogramming and multiprocessor computer operating systems designed to share resources dynamically among several concurrently executing tasks comprise the third group.

The operational SAC Control System and the planned Advanced Airborne Command Posts exemplify military command control systems with different approaches to the management of data sharing. In the former, updating normally occurs in small batches during whose processing all retrieval is shut out. Some of the advantages and difficulties of this approach with respect to data sharing are mentioned in a joint SAC/System Development Corporation study report.[1] Advanced Airborne Command Post on-board data processing subsystem operation for both SAC and NEACP will, as currently planned, emphasize serial instead of small-batch data base updating both by several persons at on-line consoles and automatically via communication link. Adequate response times without extensive parallel task

[1]See Osajima, et al.,[(3)] pp. 23-24, and 28-32.

execution are unlikely. The discipline of data sharing in this system has yet to be explored in detail, and represents a challenging prospect for application of this task's research.

Successful on-line airline reservation systems[2] such as the IBM SABRE and PARS systems, have been operational for a few years, with an advanced version of the latter now under development. Limited-objective fixed-transaction solutions to data sharing problems are characteristics in such systems. Among the techniques needing successful transfer between such systems is the discipline of data sharing. The Military Airlift Command (MAC) is now planning an on-line reservation system, as part of the MAC Information Management System (MACIMS), which will require procedures and mechanisms to control data sharing. We believe that the MAC design effort could benefit substantially from results of this research.

On-line multi-user generalized data management systems such as ADAM,[3] TDMS,[4] and GIM(8) built to date have generally either ignored data sharing problems or else partly avoided them through

[2]A succinct introduction to airline reservation system functions and some of their problems may be found in Martin,(4) Chapter 15.

[3]The description of ADAM's capabilities in the CODASYL survey,(5) Chapter 3, is the best generally available, although it does not stress data sharing.

[4]TDMS is also described in the CODASYL survey,(5) Chapter 10, and in other generally available papers(6)(7) which again do not address data sharing.

restrictions on concurrent use. For example, ADAM executed only one on-line user terminal command at a time; meanwhile any other terminal commands entered were merely queued. TDMS similarly restricts concurrent operations involving the same data base. None of these computer programs is operational in an environment requiring rapid response to several concurrent on-line users. Consequently, the delays caused by these deficiencies have only been annoying. Extrapolation to severe trouble in operational use is easy, however. Desired future on-line generalized data management systems must provide better response to more users, and must also protect the large integrated data bases from garbling by the ill-coordinated actions of several users. No known generalized data management system now provides these capabilities.

Multiprocessor systems, and single central processor multiprogramming systems that allow dynamic resource sharing, provide for communication among tasks (and among processors, in multiprocessing systems) in part via shared data. They must also prevent deadlock over shared resources,(9)(10) a problem common to systems sharing data.

## APPROACH

The effort to date has been largely conceptual. It has involved the following initial activities:

1. review of technical literature that refers to concurrent data sharing and allied problems;

2. formulation of a classification of the problems associated with data sharing;

3. collection and postulation of techniques to avoid or ameliorate some of these problems;

4. review of certain existing or planned computer applications which have had data sharing problems, which could have such problems in the future, or in which data sharing problems have been avoided at some cost in system effectiveness;

5. qualitative estimation of improvements in these systems obtainable through various shared data control techniques; and

6. postulation of feasible activities to more clearly define data sharing problems and to investigate possible solutions.

Further effort along these lines is anticipated.

## PLAN OF THIS REPORT

The subsequent sections of this report reflect these activities and the report's objectives stated earlier. Section II comprises an initial technical analysis of the major problems associated with concurrent data sharing. It discusses important effects of both inadequate and excessive constraints on data sharing, and reviews several promising approaches to their resolution.

Section III examines user procedure problems associated with data sharing and briefly reviews several existing and proposed systems for instances of these problems or ways to avoid them. These examples are drawn either from specific Air Force systems or else from others with functions very similar to Air Force applications. It distinguishes three general approaches to such solution attempts and indicates how several of the systems surveyed have applied them.

Section IV states the aims of, and approach to, a desirable future technical work program. It also outlines the major activities and tasks comprising the program and suggests the kinds of products to be generated during the effort.

Finally, a list of technical reports and papers found useful during our preliminary analysis and referenced in this report is included.

## SECTION II

## TECHNICAL ANALYSIS

This section of the report analyzes the major system error and system performance problems of concurrent data sharing thus far considered, after first discussing several basic concepts pertinent to the analysis. It then briefly explores a few promising approaches to these technical problems' solution. User procedure problems are discussed in Section III. Section II-III treat in greater depth many of the same problems touched on in Section I.

### BASIC CONCEPTS

Before analyzing specific cases, we first explain and informally define several basic concepts. These are deliberately made no more precise than necessary to distinguish the situations discussed.

#### Users, Tasks, Processes, and Processors

For our purposes it is sufficient to define a user as some person for whom a computer system (comprising both equipment and software) is operated. Users may submit work for off-line computation or communicate with an on-line system by means of remote batch stations or via interactive on-line terminals, local or remote.

A computer system accomplishes each user's work by performing a computation(11) or task. A task is a coordinated set of processes. Each process is a sequence of machine operations; i.e., those directed by a sequence of one or more indivisible steps: each a central processor instruction or a data channel command.[5] Any process of two or more steps can be divided into consecutive subprocesses, each a sequence of one or more steps, and thus a process.

A process is active when it is driving a processor. Otherwise, a process is inactive or suspended. A processor is a device able to interpret a sequence of one or more machine instructions or commands and to initiate and control a set of machine operations for each element of the sequence.

In a typical multiprogrammed computer system, there are usually two basic types of processors: (1) so-called central processors, which execute most of the arithmetic, logical, and control operations, mainly on operands stored in main memory; and (2) (data) channels, specialized input-output processors that mediate transfer of blocks of data between main memory and either auxiliary storage or peripheral devices.

A channel is actually operated as a slave of the one or more central processors in the system. A central processor instruction is required to initiate each block's transfer (which may be considered a process). The channel then directs the details of transfer operation without further central processor attention, signalling the central processor only when transfer has ended (normally or otherwise). Thus, a computer that includes a single central processor and one or

[5]Dennis and Van Horn(11) (p. 145) define a process (in the same sense that we use it here) as "...that abstract entity which moves through the instructions of a procedure as the procedure is executed by a processor."

more channels actually multiprocesses data, although we reserve the term multiprocessor for a system that includes at least two central processors. Most so-called high-speed channels process but one transfer at a time. Multiplexor channels, however, can concurrently control several, by interleaving the handling of small packets of data. This interleaving is strictly equipment-directed, involving no program execution.

A typical central processor executes the machine instruction sequence of but one process at a time. Normally, this active process can be suspended (or terminated) and another process activated, in one of two ways. Either the first process gives up control of the central processor by deliberately causing an interrupt, or else control is seized from it as a result of an interrupt caused by an event external to the process (e.g., a channel signalling completion of a block transfer, a clock interrupt ending a time-slice). In either case, the central processor then begins a different process (usually an operating system function). This second process, if correct and allowed to terminate normally, may either (1) return control to the original process or (2) activate a different (third) process.

## Time Relations of Processes

Thus far we have used the term "concurrent" rather loosely to refer to active processes that overlap somewhat in time. In this sense, two processes are concurrent if and only if one starts before the other finishes. Processes which are not concurrent will be termed consecutive. Figure 1 depicts consecutive processes.

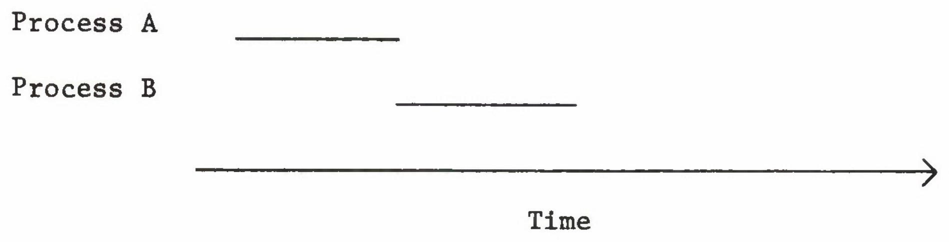


Figure 1. Consecutive Processes

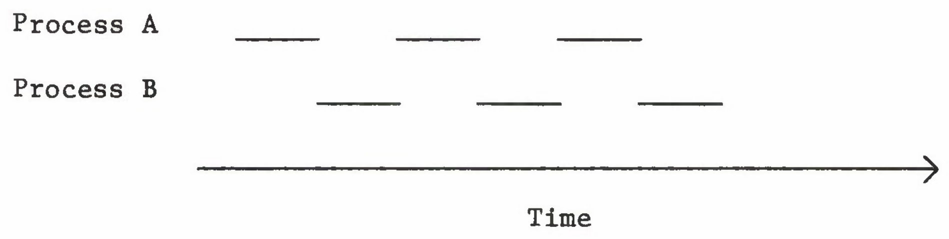


Figure 2. Interleaved Processes

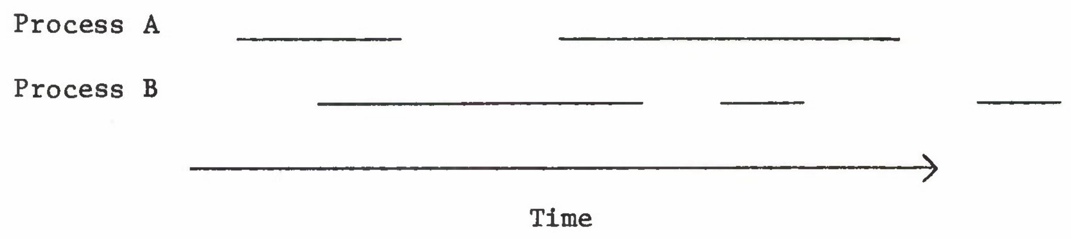


Figure 3. Overlapped Processes

Since most processes entail execution of a sequence of machine instructions (and the possible occurrence of other events), two important cases of concurrent processes can usefully be distinguished: _interleaved_ and _overlapped_ execution. During interleaved execution, concurrent processes time-share a processor. Only one process at a time is active, however. Use of the processor oscillates between processes as a function of programmed and external conditions (e.g., interrupts). As a result, at least one process is suspended and then resumed. Figure 2 shows two interleaved processes.

In contrast, during overlapped execution, both processes are jointly active during at least some small time interval, as Figure 3 illustrates. Overlapped activity of two processes requires two processors in simultaneous operation. Overlapped execution typifies effective multiprocessing. Many conventional computers also provide for overlapping a single central processor's operation with those of one or more channels.

Overlapped execution always involves some interleaving; i.e., times when only one process may be active. Although efficient use of cooperating processors requires their overlapped operation most of the time, their occasional coordination and communication are necessary to effect cooperation.

## Fields

We define a _field_ as the contents of a contiguous sequence of computer storage elements with well-defined storage address boundaries (e.g., start, and length or end), a specific user-defined meaning, and a single definite value on each occasion referenced. (Fields may take on different values on different occasions.) Fields are the smallest addressable units of information, from which all data structures are composed. It is important to note that a field is a conceptual entity, unlike, e.g., a register. The definition given covers fixed-length and most variable-length fields, including the common cases where an initial count or a terminal bit configuration defines a field's length. It excludes certain esoteric configurations in which parts of a value are separately represented; e.g., non-contiguous overflow bits, length specification disconnected from variable-length value. However, all such cases can be treated satisfactorily as related sets of fields, discussed shortly. The definition also excludes consecutive bit sequences containing subsets sometimes treated separately. Examples of the latter include the fixed-point part and exponent of a floating point number; the month, day and year of a date; and the individual characters of a character string. To data sharing analysis, such a subset may be ignored if it is never the independent operand of concurrent processes. Otherwise, it may be considered part of a related set.

## Data Structures

Fields logically related or frequently referenced in closely spaced time intervals are often grouped into _records_. A conventional _serial file_ is composed of a sequence of records. A _parallel file_ (fully) _inverted file_, or an _indexed serial file_ may each be regarded as a group of related subfiles, each of which is a serial file. The entire collection of (machine readable) files is often called the _data base_. Hence, these more complex structures are also ultimately composed of records. Temporary storages (e.g., queues), vectors, and the rows of matrices (stored by row) may similarly be considered serial files.

## Access-Related Data Properties

We next note several properties of data based on the ways in which they may be accessed. Both data sharing problems and promising approaches to their amelioration depend substantially on these properties.

Some data may not be subject to change by any of the particular set of processes in operation during a time period of interest. Such read-only data can often be distinguished advantageously from alterable data in planning these processes operation.

Two or more fields that have the same meaning and value (but distinct storage locations) are termed copies of one another. Two data structures are copies if each field of one has exactly one copy in the other, and if the fields of each data structure have the same relative storage locations. In a sense, a copy is made whenever data are moved from auxiliary storage to main memory, but in this paper the notion of "copy" will be limited to a duplicate of data in both form and content such that a process can use either and obtain the same results. A field or data structure that has no copy we term unique.

A field or group of fields that but one process may reference (i.e., read or write) is said to be private to that process. In contrast, data that two or more processes may reference are said to be shared by or common to those processes. Common data may comprise one or more copies, but usually a set of processes shares a unique set of fields. (As subsequently discussed, data are often copied to avoid data sharing.) Except as otherwise noted, we shall henceforth mean unique shared data when speaking of shared or common data.

Common data may be shared consecutively or concurrently. In consecutive data sharing, i.e., during either conventional batch processing or seriatim processing (in which transactions are processed one at a time), each of the sharing processes has exclusive access to the data during its entire period of operation. Consecutive data sharing presents no unusual problems. In concurrent data sharing, however, two or more processes desire overlapped or interleaved access to one or more common fields. The many concurrent data sharing problems are the subject of this research.

## Related Sets

The exact set of fields that a process references we term that process' related set.[6] Related set membership may differ from process to process, even among those executing the same procedure. A process' related set may include read-only and alterable data, private data, and data the process shares with one or more other processes. A related set may also include copies. During different steps in a process' operation its related set may include different values of the same alterable field, changed either by the process itself or by other, cooperating, processes. Related set membership seldom corresponds precisely to file membership. A related set may include fields in the same record, in different records in the same or related subfiles, or in different files.

[6] Related set is a generalization of consistent set, a concept found in Waghorn,(12) page 204.

To illustrate, several simple examples are presented of processes and the elements of their related sets:

1. The values of the control field of the records in a batch comprise the related set of a process summing them and checking the result against a control total.

2. With respect to posting a payment record entering an accounting system, the payment record values referred to, the resulting entries made in all accounts, and the corresponding control totals, comprise a related set.

3. The items of a linked list comprise a related set with respect to any list-processing operations.

Some members of a process' related set may be difficult to identify. Many processes' main functions entail other actions, conditionally and perhaps infrequently, which can in turn cause even more remote events, perhaps in several stages. These cascading operations we call _rippling_ and their results _ripple effects_. All the data they reference are part of the process' related set. Identifying such obscure members of a related set requires especially careful analysis.

Also, a process' related set may include some data that it accesses only to reference other values. For example, to alter a partial-word field, a process must normally handle the entire machine <u>word</u>[7] of which the value desired is a part. Again, a process may

[7] A (machine) <u>word</u> is a (fixed-length) unit of information that can fill an entire main memory register. Main memory register length is typically fixed for a given model memory. Lengths of 16, 24, 32, 36, and 48 information bits are common. Whenever a memory register is addressed (by a processor) to obtain information, its <u>entire</u> contents are fetched. Similarly, the entire memory register is rewritten during any store operation. Thus, information is always fetched from or stored into conventional main memory in word-length quanta. On analysis, apparent exceptions turn out to be spurious, entailing more complex operations. To obtain part of a word, a processor must first fetch (copy into a <u>processor</u> register) the entire word and then isolate the desired portion there. To store a partial-word field, a processor must first fetch the entire prior contents of the memory register that will eventually hold the result, combine it with the partial-word field in a processor register, and store the full-word result into memory. To obtain adjacent information from two consecutive memory registers, a processor must first fetch the entire contents of both. A processor performs multiple word fetches and stores (e.g., multiword internal data transfers) as sequences of single-word memory fetches and stores.

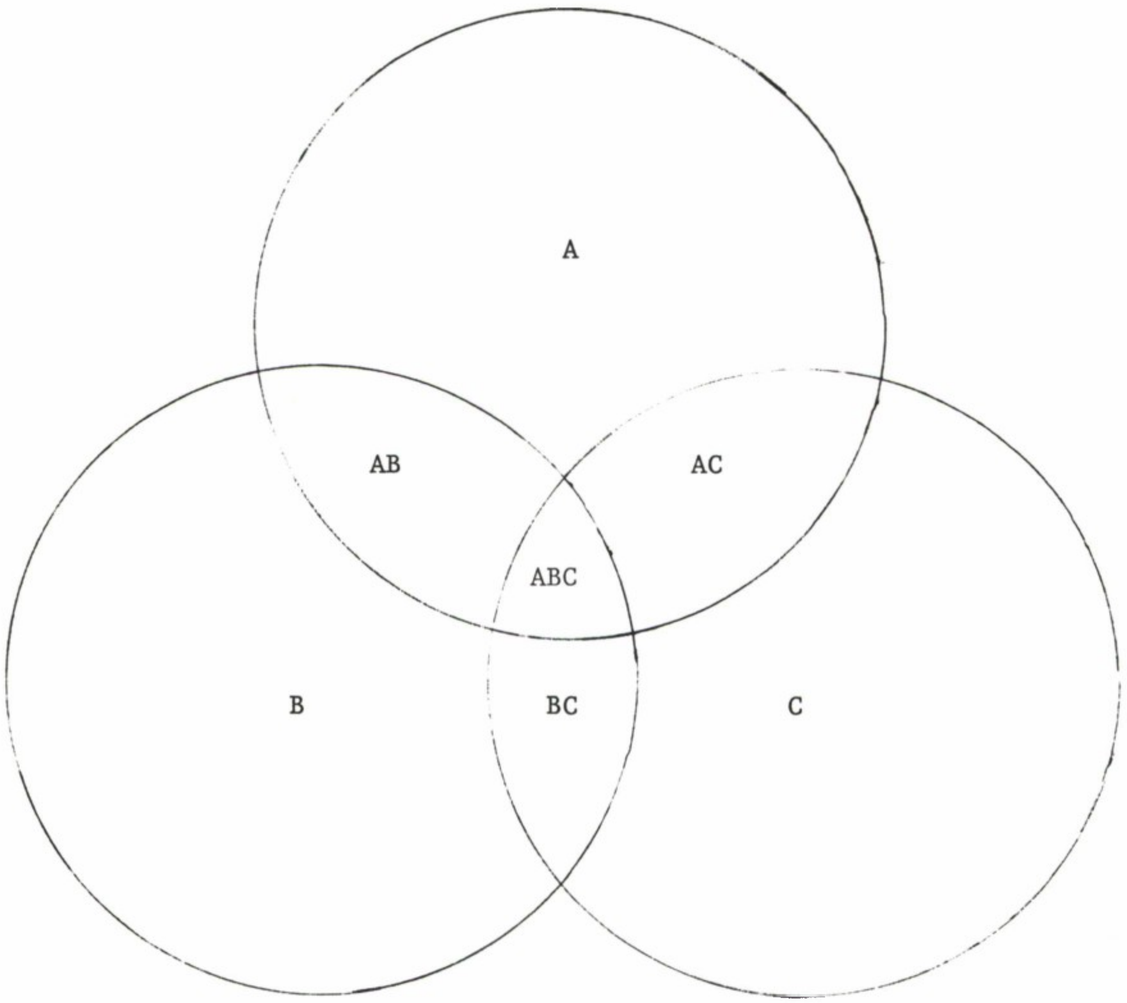
need to read in an entire auxiliary storage block[8] in order to examine a single record in that block.

## Related Set Intersection

A process may share one or more of its related set members with one or more other concurrent processes. Those common data comprise the processes' related set intersections. Figure 4 illustrates the intersecting related sets of three processes, A, B, and C. The alterable members of a related set intersection comprise a critical set.

Processes' related sets may intersect for several reasons. In some situations a shared field may be used to communicate among processes. In other cases different processes may use the same field for different purposes. In any case the use of critical sets must be coordinated.

[8]A block is a unit of information read from or written on an auxiliary storage device as a result of a single basic central processor request of a channel. In this respect, an auxiliary storage block is analogous to a main memory word. However, typical blocks are larger (e.g., 10-1,000 words) and may be variable in length. Also, the time needed to complete transfer of a block between main memory and an auxiliary storage device, the transport time, is about 4-5 orders of magnitude larger (approximately 10-400 ms.) than main memory access time, even if a separable request (i.e., a seek command) can be given to reduce latency through partially prepositioning the auxiliary storage device before a block transfer command is issued. As defined by Denning(13) (p. 157), a block's transport time includes time spent waiting (1) in a queue for issuance of its channel command, (2) to complete positioning of the (mechanical) auxiliary storage device (latency), and (3) for information transfer to end. We largely ignore (1). (2) can be substantial, forcing large block size. (3) uses modest fraction (about 1/20 to 1/2) of the main memory cycles. Even where minimum block size is small, the need to amortize substantial latency per block over many words usually forces a much larger block size, to minimize overall input-output time.



Legend:

| | |
|---|---|
| A, B, C | = the related sets of three concurrent processes. |
| AB, AC, BC | = the two-process related set intersections. |
| ABC | = the intersection of all three processes. |

Figure 4. Related Set Intersection

Related set intersections have several important implications for data sharing analysis. Processes whose related sets are disjoint share no data and can be run independently, other resources permitting. Conversely, concurrent processes can conflict to yield erroneous results over their critical sets, but not over other members of the related sets. Inappropriate methods of policing such conflict can cause system performance problems or permanently block certain processes' completion. Thus, accurate determination of related set boundaries and their intersections is fundamental to avoidance or reduction of data sharing problems. Most of the data sharing improvement strategies suggested below under PROMISING APPROACHES involve minimizing related set intersections.

## Periods of Membership

To reduce further the extent of data sharing, the notions of related set, related set intersection, and critical set must be time-constrained. To assure a process' correct operation, a field need not join a process' related set until the point in the process when another process' reference to that field could cause either process to err. Also, a field may leave a process' related set when its reference by any other process could affect neither process' correctness. Each interval between the time a field joins a process' related set and the time it leaves, we term a period of membership in that related set. In principle, a field can have one or more periods of membership in a process' related set. Contention among data sharing processes may be reduced if fields' periods of membership in related sets can be precisely established rather than assumed always to coincide with the duration of the entire process.

## An Alternate View of Related Set Membership

A complementary view of related set membership results from noting that a process' related set is the union of its subprocesses' (usually smaller) related sets, and that a process may be divided arbitarily into consecutive subprocesses of one or more steps each. By appropriately partitioning a process into subprocesses, its related set (and thus its related set intersections) can be approximated as closely as desired, even if its periods of membership as defined above are ignored and if one assumes instead that each subprocess' complete related set belongs to that subprocess during its entire period of execution.

## Critical Sections

That part of a process (or possibly the entire process) which uses a critical set can be termed a critical section.[9] To maintain properly the data used by critical sections their execution must be constrained so that two critical sections which have the same critical set cannot operate concurrently. In other words, a critical section must be made indivisible with respect to reference by other processes to its critical set.

A process may include zero, one, or more critical sections, each required to safely access the same or a different critical set. Although the critical sections that access the same critical set must occur consecutively, critical sections that access different critical sets may run concurrently.

[9]After Dijkstra,[(14)] p. 53.

A critical section may comprise a single step or many steps. Some critical sections are very long. Since most central processors can be interrupted between any two instruction executions, special precautions must be taken to prevent interruption, or else to bypass its effects during a critical section. These are discussed next.

## Access Control Mechanisms

All access control mechanisms aim to allow but one process at a time access to a common field or related set intersection. Such denial of concurrent access is often termed _lockout_ or _locking_. Hence, access control mechanisms may also be considered _locking mechanisms_. Two groups of locking mechanisms can be distinguished: _hardware locking_ and _programmed locking_. Hardware locking, as the term implies, is performed entirely by computer hardware. Hardware locking mechanisms include the primitives (basic operations) necessary to accomplish programmed lockout. Programmed locking mechanisms are algorithms that implement a lockout strategy using hardware locking and programmed locks as primitives.

A (programmed) _lock_ is a field whose value indicates whether a process may access certain data or enter a critical section, or whether the process must suspend execution. Other information; e.g., a list of suspended processes, may be associated with a lock. What a lock guards we term its _domain_ or _scope_, either a group of shared data items, one or more routines, or both. A lock's domain is established by convention. A _process lock_ guards a critical section and thus indirectly some set of shared data. A _data lock_ guards some set of data explicitly, ideally a related set intersection but often an entire file. A lock is itself a prime example of shared data. The use of locks with unnecessarily large domains we term _overlocking_.

Hardware locking mechanisms include interlock, locking instructions, and storage protection mechanisms. Interlock is entirely equipment-determined. Lock instructions and storage protection, in contrast, can be specified as components of programmed locking algorithms.

## Interlock

Interlocks associated with each active shared storage-processor pair protect that storage from access by the same or another processor for some minimum time period. In well-designed equipment, the interlock time is just enough to let the information just read or written "settle" from disturbance by the access. Interlock time is usually constant (or varies within narrow limits) for a particular processor-storage device pair and a fixed amount of information transferred. Thus, while one processor accesses a conventional destructive readout (DRO) core memory bank shared with another processor, the other processor is denied access for an entire memory cycle (on the order of 1-4 us.), the time needed to read out and restore, or to read out and replace (but not both) the accessed memory word. During this memory cycle, access is also denied to any other word in the same core bank (but not to words in other banks). (This denial, which we term secondary interlock, results from core memory equipment limitations. It would not necessarily occur in other types of main memory.)

Access to auxiliary storage is usually interlocked for much longer than access to main memory for several reasons. First, a single high-speed channel may connect several auxiliary storage devices to main memory. During a block transfer such a channel is dedicated to a single auxiliary storage device and is interlocked against access to any other device (except; e.g., for seek and rewind commands). A multiplexor channel, however, can interleave the concurrent block transfers of several auxiliary storage devices (provided their total data rate is not too high). Second, the auxiliary storage device itself is normally interlocked during an entire block transfer.[10]

In contrast, a central processor is normally interlocked only for the duration of a single instruction execution. In many machines this is not long enough to assure execution of a critical section. For this purpose, one or more locking instructions, discussed next, may be defined.

## Locking Instructions

In a single central processor multiprogramming system, interleaving of processes occurs as a result of interrupts, which in general can occur between any two machine instructions' execution. Thus, in such a computer system, interrupt masking (interrupt disabling and enabling) instructions are commonly used to assure uninterrupted critical sections within the operating system. Interrupt masking could theoretically protect critical sections elsewhere. Since, however, interrupt masking instructions are privileged (i.e., executable only in the supervisor state) in most computers, only operating system processes can execute them. Other processes would have to call on the operating system to mask interrupts.

[10] Some configurations, however, allow simultaneous connection of the same main memory module and auxiliary storage device via two or more channels. In such cases, two or more data block transfers may be in process concurrently depending on factors such as the number of independent drum read-write heads. Here separate sectors of a drum track are interlocked independently, allowing parallel drum access. Similarly, the two or more channels time-share the main memory on a word-at-a-time basis.

To implement a critical section as a single machine instruction also assures its uninterrupted completion[11] in a single central processor multiprogramming system. One example is the Test and Set instruction,(15) which indivisibly stores in a processor register an indication of a field's value while setting the field to a (fixed) new value. Correct lock management needs some such instruction because a lock must be indivisibly tested and set, as next shown. A process wishing access to a lock's domain (1) reads the lock and examines the value obtained. If this indicates the lock's domain is accessible, the process (2) writes into the lock a value meaning "domain in use" to rebuff another concurrent process and then accesses the lock's domain. If the reading and writing were not indivisible, two processes could interleave them as follows:

1. process A could do step one;
2. process B could do step one;
3. process A could do step two;
4. process B could do step two.

As a result, the second process would wrongly access the lock's domain when only the first should have done so. Therefore, to support programmed lockout, an indivisible locking instruction, such as Test and Set, must be provided.

In a multiprocessor system, interrupt masking cannot alone protect critical sections, unless each processor can feasibly disable <u>all</u> processors' interrupts. Also, in such a system, making a critical

[11]Except for interrupts resulting from detection of equipment malfunction.

section a single machine instruction will not assure its indivisibility, if that instruction has multi-word operands, since main memory interlock denies access to but one word (or core bank) at a time for but one memory cycle. Consequently, one process can sometimes access another's operand between memory cycles of the first process' instruction. The Test and Set instruction, which has a single-byte operand, has been used successfully, however, as the basic primitive for programmed lockout in a multiprocessor system.(15)

### Storage Protection

Storage protection is another form of hardware lockout available on many computers. Privileged instructions allow specification of ranges of main memory locations that a process may not access or that it may only read or execute. Hardware detects any reference to such locations in violation of these restrictions. Intended primarily to prevent erroneous sharing among concurrent processes, storage protection mechanisms, or modifications of them, could be useful primitives in certain programmed lockout strategies.

## Deadlock

Deadlock occurs when each of two or more concurrent processes has been allocated exclusive control of a resource needed by another which it cannot release, and each, to continue, needs at least one such resource held by another. As a result, none of the deadlocked processes can complete its operation. Each set of locked shared data and each critical section, like the storage space it occupies, may be such a resource.

## PROBLEMS

System error and system performance problems that can afflict concurrent data sharing are discussed below in terms of the basic concepts just presented. First, possible adverse effects of ill-coordinated data sharing are examined. Next, difficulties that can result from imposing excessive data sharing controls are reviewed. Third, we discuss deadlock, a special hazard risked by inappropriate shared data controls. Finally, we point out a few other miscellaneous difficulties.

### Effects of Ill-Coordinated Data Sharing

Three groups of problems are examined below. In the first, concurrent processes contend to read common data but none modify it. In the second, one process writing shared data conflicts with others reading it. In the third, multiple processes conflict while concurrently writing the same field or members of intersecting related sets. As a rule, each group's effects are worse than its predecessor's. In addition, the second group includes the first group's effects, and third group those of both the first and second groups.

#### Contention During Concurrent Reading

Two or more uncoordinated concurrent processes that are only reading common data cannot garble it and will obtain correct results, provided equipment interlocks work properly. Such processes' contention may indirectly cause computer system performance degradation, however.

Only the possible adverse effects of read-only data sharing are discussed here. The benefits, which usually outweigh the costs, are treated subsequently under PROMISING APPROACHES.

The basic effects of read-only data sharing are those of sharing certain resources. The resources of prime interest to analysis of data sharing are main memory, auxiliary storage, and the channels that connect them. Data sharing can impair performance primarily by deranging storage access patterns planned to reduce average access time.

For example, several years ago a sort routine was written that stored partially-ordered sequences on a large disk and repeatedly read, merged, and rewrote information there. For this disk system one of four track-to-track access times was possible, depending on the type of access mechanism motion entailed. The sort contained an algorithm to select the next track for output which reduced track-to-track access time substantially below average, when the program ran alone. When multiprogrammed with other jobs, however, the sort's performance degraded because extra time was usually needed to reposition the disk access mechanism after each use by a concurrent job. Even though these jobs shared no data, they did share a data-containing device.

Since the concurrent processes in question are uncoordinated, the usual effect of access pattern derangement is to randomize access, although worse than random access patterns sometimes reults. As a general rule, the less uniform the access time to a storage device is, the more careful programming can improve it and the more unsynchronized data sharing can impair it. Consequently, concurrent access to data in main memory usually degrades access time only slightly.

However, concurrent access can increase total magnetic drum access time substantially, total magnetic disk access time seriously and total magnetic tape access time intolerably.

## Read-Write Conflict

A single otherwise correct process allowed to alter common data while other uncoordinated processes read it will normally change the data correctly, but one or more of the readers may obtain incorrect results. If, however, these results are used to direct processing operations, further damage may result. Also, duplicating results may be impossible: the same processes run at slightly different relative times may yield different results. Consequently, debugging may be very difficult. Examples illustrate typical problems.

A process that reads a field while another updates it will obtain the field's old value if the reading reference occurs before, and the new value if reading occurs after, the change has been made. Unless the different processes are deliberately coordinated, the exact order of their instructions' execution cannot be guaranteed, and so results will vary unpredictably. Although either result could be considered correct depending on whether the old or the new value was desired, such variation could disturb certain operational users and cast doubt on system correctness, even where systems' requirements allow such variations.

A process that prepares a summary, or that otherwise examines more than one field (or the same field more than once) of a file that another process is concurrently changing, risks more serious trouble:

incorrect results. The reader's results may represent neither the original version nor the new version of the file, if some data is read before and some after its alteration. If these results are used to control processing operations, instructions whose execution comprises a process or processes may be garbled. Through a complex chain of events thus started, widespread damage to both data and programs can follow. Alternatively, the effects can be more subtle; e.g., decreased performance or deadlock.

## Concurrent-Write Conflict

An attempt by two uncoordinated concurrent processes to change one or more fields of their intersecting related sets can also have serious adverse effects. Such concurrent-write conflict can introduce incorrect values into individual fields and prematurely alter related sets, causing processing errors during subsequent computing, retrieval, and updating.

Often a process examines a single field to decide whether or how to alter the same field or other members of a related set. Error can occur if another process intervenes between this examination and the consequent action. Both operations must be coordinated; i.e., done within a critical section, to assure correct results. For example, suppose that an on-line airline reservation system is programmed to book, if available, the number of seats requested on a flight, but otherwise to respond that space is unavailable. Further suppose a specific flight has but one seat left and that agents A and B each request that seat. As a result process A operates for agent A's request and process B for agent B's. If the two processes are interleaved, the following sequence of events could occur:

1. Process A loads into an accumulator the flight's seat count.

2. Process B loads into an accumulator the same seat count.

3. Process A compares the loaded seat count to the number requested, finds enough spaces, decreases the loaded count, stores the result in memory, and completes the availability processing.

4. Process B does the same.

Here process B "errs" because it was allowed to operate during a critical section. In addition, the file is left with incorrect data. The problem could occur in a multiprocessor system or in a single central processor multiprogramming system if an interrupt occurred after each of steps 1 and 2 and the processes were subsequently re-activated in the same order in which they were interrupted.

When a related set is inconsistently updated, results can be more complex. In an indexed file the substantive data items and the corresponding index fields could be changed inconsistently if all significant results of one process's actions were not recorded before another process referred to them. If the second process obtained portions of an index before the first had revised and replaced it, the second might either change it wrongly or reference the wrong serial file field (via an obsolete index pointer). Also the first process might overwrite the index as adjusted by the second, erasing the record of the second process's actions. To eliminate risk of data base damage (and consequent possible loss of processing control) all concurrent-write conflicts must be avoided.

The major adverse effects of ill-coordinated data sharing have so far been outlined and illustrated in this subsection. These include potential performance degradation, incorrect results read when one or more reading processes contend for access with a single writing process, and inconsistent data when two or more concurrent writing processes conflict.

## Effects of Excessive Data Sharing Control

Three general approaches are commonly followed, singly or in combination, to avoid the erroneous results of inadequate coordination, and sometimes to improve performance. First, processes that might usefully run concurrently are required to run consecutively. Second, processes are run concurrently but constrained to operate mainly on distributed data (see p. 38); the identities, storage locations, types, and amounts of common data permitted are strictly limited. Third, data locks or process locks are used to force consecutive access to the intersections of processes' related sets. It appears possible to classify all known data sharing control methods, good or bad, as instances of these approaches or their combination.

For any particular application, some well-designed mixture of these approaches appears to be the strategy most appropriate to effective data sharing. Some are suggested under PROMISING APPROACHES, below. However, any one such approach pursued excessively or a poorly designed combination can cause one or more of the following adverse effects. Desirable cooperation among concurrent processes may be prevented. Computer equipment may be used inefficiently. Performance may degrade. Response times may lengthen. Finally, deadlock may occur. Several of these practical problems of excessive data sharing control are illustrated below.

## Impact of Consecutive Processing

Processing changes and information requests in batches or in strict succession permits a system to enforce a desirable order on such inputs. In the SAC Control System (SACCS),(3) for instance, a batch of changes and information requests is accumulated until one of three events occurs: (1) a fixed period of time has elapsed, (2) a critical batch-size limit is reached, or (3) a high-priority message enters the batch. Any one of these three events will trigger a processing cycle, during which all changes in the batch are first applied to the data base in a conflict-free order. The information requests in the batch are then processed to obtain latest results. Changes and information requests that enter the system during processing of one batch are held for the next.

The small batch approach avoids the effects of uncoordinated data sharing just discussed. Neither write-write, read-write, nor read-read conflict can occur, provided program logic is correct and the different types of input in the batch are recognized and arranged appropriately. The method's disadvantages are basically those of large batch processing, although smaller in scale: (1) responses to information requests are delayed while multiple changes are processed, and (2) the information retrieved is seldom truly current because it does not reflect unprocessed changes accumulating for the next batch. As a result the approach is seldom suitable for systems in which several users cooperate by dynamic interaction with a shared data base.

These disadvantages can be minimized by reducing batch size. They disappear in the limiting case, seriatim processing (where "batch size" = 1). Seriatim and very small batch processing may use computer system equipment quite inefficiently, however. The consequent reduced throughput can cause long queues of changes and information requests to form awaiting processing and lead to increased response times.

## Disadvantages of Data Distribution

A group of fields is said to be distributed or segregated when it is divided into subsets which are stored sufficiently apart to allow their uncoordinated access by concurrent processes. Each segregated subset may consist entirely of unique fields, solely of copies, or may contain some of each. Whether two fields are distributed depends on the operations to be performed on them. For example, two fields in separate core memory banks are distributed with respect to individual main memory fetch and store operations but are not necessarily distributed with respect to a block transfer spanning both core banks.

As discussed below under PROMISING APPROACHES, data distribution can speed the operation of concurrent processes, simplify their design, and reduce debugging effort. The approach has certain disadvantages, however, noted next.

Distributing data to minimize its sharing can entail redundant operations and increase resource requirements. Separate copies of data made to avoid access conflicts (1) need additional storage, (2) require extra execution time to prepare, and (3) may entail parallel updating. Also, additional computer instructions must be prepared and debugged to perform the copy and update functions to keep copies compatible. Segregated unique fields may require explicit links to relate them when these relationships might otherwise have been implicit. Such links may also be needed to relate copies. Finally, excessive processing may be required to reformat and distribute data; e.g., if each block read were to be dispersed and each block written collected, during frequently repeated on-line processing.

## Problems of Locking Data

By excessively restricting concurrent access, programmed data lockout can impair performance and may contribute to deadlock. Design and debugging of reasonably efficient locking procedures is difficult and complex. Programmed locks need storage, and their manipulation requires extra instruction executions. These difficulties are discussed next.

Since the precise composition of a related set of data may be difficult to establish (e.g., because of ripple effects), designing and debugging an optimal locking algorithm may require substantial effort. (Ideal locking algorithms would protect exact related set intersections for minimum times.) Sets of programmed locks may be required in some cases to release subsets of data no longer needed and to avoid prematurely locking other subsets, which can cause deadlock as well as delays. Even determining exactly where locks are needed may be difficult. Because related set membership is sensitive to details of processing, each process may need a somewhat different optimal locking algorithm, and whenever a process is changed, its locking algorithm may need to be reviewed for corresponding alterations.

Precision locking may also require the storage of many programmed locks. Each such lock typically requires several bits of storage (e.g., a byte, or even an entire memory register to index the queue of waiting processes associated with the lock). Thus, total lock storage requirements may not be trivial. The mechanisms of effective programmed lock manipulation are far from simple, either, unless special locking instructions are provided. Thus, lock management can contribute to performance problems.

To avoid these complications and especially to reduce development effort, cruder but simpler locking schemes tend to be devised in practice. Unfortunately, these crude methods overlock, thus denying other concurrent processes legitimate access to shared data and causing performance degradation or contributing to deadlock. As extreme examples, the ADEPT-50 executive (under which TDMS(6) runs) locks an entire integrated data base at a time, and IBM's OS/360(16) may lock an entire set of files (called "data sets") for an entire job.

Process locking is sometimes used instead of data locking to reduce the number of locks required. Although process locking can <u>sometimes</u> save lock storage and may simplify locking algorithms, it too can cause overlocking. For example, two processes that could safely read a common dictionary concurrently may do so only consecutively if constrained to access it via a common procedure.

Discovery of efficient generalized precision locking mechanisms adaptable to program change, or of ways to recompile equivalent specialized locking mechanisms when altering a program, could greatly reduce the tendency to overlock and would thus contribute substantially to more efficient multiple user computer system design.

## Deadlock

Deadlock is one of the most serious potential effects of inappropriate locking algorithm design. Deadlock has been well described by Havender,(9) and Habermann(10) has outlined a way to avoid it in some cases. There seems considerable current professional interest in the subject, so one may expect improved methods to be developed and published in the near future. The methods now known are either restrictive(9) or complex and time-consuming(10) to implement and execute, especially if the number of elements (e.g., lock domains) is large.

Where a system design does not include a foolproof deadlock avoidance algorithm, the problem may occur often or rarely, under conditions difficult to predict exactly. Overlocking may cause deadlock among processes that would otherwise be more likely to operate without interference on logically independent subsets of a set of data. On the other hand, more precise locking might allow certain processes to begin concurrent operation, only to become deadlocked later over other serially usable resources.

Deadlock can also be hard to detect. For example, in a multiprogramming system running three concurrent processes, if two become deadlocked, but the third does not, the problem may go unnoticed for quite a while. What to do when deadlock _is_ detected is also far from clear. In theory, an operating system could abort deadlocked processes one by one, releasing at each stage resources that might allow one or more of the remaining processes to continue. This approach is not feasible, however, without giving the operating system information normally not available, about the relative worth of the deadlocked processes, and the probable effect of aborting each on the continuance of the others and on any files it may be updating. Most important, to abort any writing process risks garbling the data base. This is generally an intolerable risk. Because of these difficulties, development of better avoidance algorithms appears more fruitful at this writing that pursuit of schemes to detect and break deadlock after it occurs.

### Other Problems

Data sharing can aggravate other problems that arise in a multiprogramming environment, although it does not cause them. These are the problems of version identification, audit, debugging, and restart. In each case data sharing aggravates the difficulty by introducing multiple possible sources of change, inquiry, and error which are absent when one process at a time refers to data. These problems are mentioned here because they must be dealt with satisfactorily in any system employing shared data.

## PROMISING APPROACHES

In this subsection we mention informally several promising general approaches to further research into data sharing control. Subsequent research effort should pursue them more deeply, although other worthwhile approaches undoubtedly exist. The approaches suggested below have neither been developed in complete detail nor subjected to thorough evaluation and trade-off analysis.

An ideal data sharing strategy would fully protect common data from inconsistent alteration, and reading processes from inconsistent results, while minimizing interference and avoiding deadlock. Earlier, three basic approaches to data sharing control were suggested, and their problems were discussed: (1) consecutive processing, (2) data distribution, and (3) data and process locking. We now suggest several strategies based on them, stressing advantages instead of difficulties, many of which have been already discussed under PROBLEMS.

## Consecutive Processing

Consecutive processing avoids conflict over common data by separating in time all occasions of shared data access. Either of the two general approaches discussed below may be valuable in parts of certain applications. A third approach, seriatim processing, can seldom be afforded in any system involving multiple concurrent users.

### Maximum Use of Off-Line Processing

Certain functions associated with a multiple user on-line computer system can usually be done off-line, in advance of concurrent on-line access, using well-known batch processing techniques. Examples of functions often appropriately performed off-line include initial preparation of computer program and data files for on-line use, their subsequent restructuring, and regular massive data file updating. As a rule the more such functions can be done off-line, the simpler and more effective the on-line system. For example, if a system's requirements allowed performing all updating off-line in nightly batches, its daily on-line use would reduce to retrieval, and its data sharing problems to contention associated with concurrent reading, a relatively trivial difficulty.

Batch processing has tremendous potential advantages wherever its usually excessive response times can be tolerated. First, operational control, version control, and program logic can together prevent inconsistent updating. Extensive error checking is feasible. For example, a control field in each transaction can be summed and compared against a prelisted batch control total, with batch rejection and detailed investigation of the cause following on disagreement. This technique is commonly used to detect replicate or missing transactions. Transactions affecting multiple data base records can be transformed into several simple transactions, each affecting only one data base record. Multiple changes to the same record can be sorted for application to the data base in any desired order; in particular their time-consistent application can be assured. Other consistency checks (e.g., for duplicates, for illegal sequences of transaction types) can also be built easily into batch update program logic.

Second, the same mechanisms can segregate a batch's changes from its information requests, so all changes to a record precede any information request involving that record. Thus, time-consistent retrievals, summaries, and other analyses of a file can be guaranteed.

Third, the multiple information requests for a batch can be expanded, reformatted, and sorted for optimum retrieval efficiency. Thus, batch processing can be designed to preclude performance degradation resulting from contention during concurrent reading.

Fourth, the methods of interference prevention just outlined preclude any need for programmed locks and thus avoid the overhead and other problems normally associated with their use. They also preclude deadlock.

Fifth, the batches' transactions and the saved source data base (or the corresponding checkpoint data if file maintenance is destructive) are preserved, and available to repeat updating and retrieval; e.g., for data base reconstruction, audit, or more dynamic debugging investigations.

To summarize, batch processing procedures enforce certain principles essential to satisfactory data sharing. These are:

1. correct order of operations;

2. time-separation of indivisible operations;

3. efficient order of operations;

4. precise identification of events and accountability for actions; and

5. reproducibility of results.

## Small-Batch Processing

The long response times that result from processing large batches can often be cut by reducing batch size, provided the updating method does not reproduce[12] the data base. Where a batch size can be selected that will yield both satisfactory throughput and acceptable response times, small-batch processing is a proven, effective way to avoid data sharing problems. Variations of the SAC Control System's method discussed under Impact of Consecutive Processing may reasonably determine batch size. Allowing a priority message to terminate a batch risks degeneration into seriatim processing, however, unless priority assignment is carefully controlled. Also, because batch processing efficiency tends to diminish as batch size decreases, small-batch processing will not necessarily yield both adequate throughput and response times short enough to satisfy application requirements.

## Data Distribution

Data can be partitioned in several ways prior to high-frequency online use to avoid or at least reduce concurrent data sharing and its problems. One technique stores separately data that may change during a period of concurrent sharing from read-only data, permitting concurrent reading of the latter. Another (complementary) technique separates private data from data shared concurrently among different processes. A third method replicates common data so that each process

[12] One of two alternate methods is used in batch updating. In the first, termed reproductive file maintenance, a complete copy of each updated file is written in a storage area distinct from the source data base. In the second, non-reproductive file maintenance, the source data base is merely altered in place. Reproductive file maintenance preserves the source data base as backup data while non-reproductive file maintenance destroys it. Nevertheless, reproductive file maintenance is a very slow way to do small-batch updating, because the entire set of altered files must be written during each batch's processing.

may access a copy more freely. All tend to minimize the scope, complexity, and interference of programmed lockout, and may contribute to more efficient storage utilization. (Disadvantages of Data Distribution, above, mentions corresponding problems.)

## Segregating Read-Only Data

It is often possible to design an application so that much of its data (e.g., the geographical location of cities, aircraft performance data) is not subject to change during a period of multiple concurrent use. Not only large sets of data (e.g., reference tables), but also certain fields of typical records may never be altered by any process invoked during such a period, although some of these fields may be changed on other occasions. Segregating (e.g., into distinct files or subfiles) and reformatting most of this read-only data can simplify concurrent on-line processing.

Apart from easing data sharing there are many advantages to storing read-only data separately from alterable data. For instance, read-only data can often be formatted more efficiently than alterable data (e.g., organized for more efficient search, blocked without spare space) because ease of updating can be ignored. Also, segregated read-only data can be omitted from checkpoints, provided its original source is available for restart. If the volume of such segregated data is large, considerable checkpointing time can be saved.

Segregating common read-only data can be especially valuable. Read-only data may require storage protection against inadvertent change but needs no programmed lockout to avoid error. Any number of processes may freely access common read-only data concurrently with no risk of error.

How "far" from other data to store segregated common read-only data depends on the types of protection and access conflict avoidance desired and on characteristics of the storage devices and access mechanisms used. Since the entire contents of a main memory register is fetched, stored, and interlocked as a whole, segregation at least by word is normally required. Main storage hardware read or write protection must typically be applied indivisibly to groups of consecutively addressed memory registers; e.g., modulo 512-word groups. To protect read-only data against accidental writing in such a main memory would require storing it in one or more such groups of memory registers, distinct from others storing alterable data. On auxiliary storage, segregation of common read-only data in blocks distinct from blocks containing other data generally contributes both to ease of access control and to reduced contention.

Removing segregated read-only data from the scope of imprecise programmed locks is trivial. Consequently processes cannot be slowed in accessing such data by imprecise programmed lockout intended to control access to other data. Nor can deadlock occur over segregated read-only data because no process is prevented from reading it. By storing elsewhere the read-only data items, the storage organization and logical relationships of the remaining writable data _may_ sometimes be simplified, permitting consequent simplification of the locking mechanisms controlling it.

Contention (e.g., for main memory access) inevitably results from concurrent access, but (for data stored in main memory) other advantages usually outweigh its costs. For example, to share a single main memory copy of a table read by two concurrently executing programs may take less total elapsed time than to transport it twice from disk. Further, sharing the single copy will consume fewer memory cycles than first copying it to a separate main memory area for separate reference, unless both processes use the common table intensively and unless the copy and the original can be stored in separate core banks to avoid secondary interlock. More important, when too little main memory is available to replicate common read-only data, the ability of processes reading it to run concurrently at all may depend on their right to read the same copy.

In applications involving many concurrent on-line users, similar processes (e.g., translation of information requests, preparation of output in similar form) must often be run for each. Hence, much of the data they need can often be formatted as common read-only data. In such situations data sharing may satisfactorily substitute for increased main memory capacity, whose acquisition may be technically or economically infeasible.

## Partitioning Common and Private Data

If each process' private data is segregated from common data, each process can access its private data without programmed lockout and its disadvantages. These different processes may use the same or different routines. For example, different processes assigned to update separate subsets of a geographically organized file may apply the same routine to distinct groups of fields. On the other hand, different routines may be used to update data in separated financial and personnel subfiles.

Well-known buffering techniques perhaps best illustrate the advantages of segregating private data. If one process assembles data for output a block at a time in some fixed main memory area, and then initiates a block transfer (a channel process) to move it to auxiliary storage, the first process must wait for the block transfer to end before starting to prepare the next block. Otherwise the first process may incorrectly write into main memory registers not yet read by the channel process. If the first process copies an assembled block to a separate buffer area from which the channel can transfer it, however, the first process can prepare the next block in the original area while the channel operates on the first block. Even the copy time can be avoided if the memory areas allocated to data assembly and output are alternated, as in double-buffering.

Hardware storage protection, if available and sufficiently flexible, may be feasibly applied to segregated private data to prevent erroneous access by other processes. Also removing private data from the scope of programmed locks could simplify lockout design and operation for the remaining shared data.

Control of the remaining, common, data can sometimes be simplified, and interference due to programmed lockout may be reduced, if data common to each <u>pair</u> of processes is segregated and locked separately from data shared by more than two concurrent processes. This approach can be pursued further, to segregate and separately lock the data shared by each triplet, quartet, etc., of concurrent processes. At some point, however, the strategy's complexity defeats its advantages.

## <u>Replication</u>

Under some conditions net improvements in processing time and programmed lockout simplication may result if data common to two or more processes is replicated, so that each process can use its own copy, or so all read-only processes can share a copy, without explicitly synchronizing each reference with those of the others. A process using such a copy can escape the problems of programmed lockout provided the data are not used to communicate with other processes.

Replication is most valuable to ameliorate read-write conflicts by allowing one or more read-only processes overlapped or interleaved access to the intersections of their related sets with that of exactly one writing process. When but one copy of such an intersection must be shared, to avoid inconsistent reading results the process wishing to write must wait for all readers to finish. Similarly, while the writer has access all readers must wait. If, however, two separate copies are made available, the readers can safely access one while the writing process alters the other.

For example, the use of alternate copies of a file to permit safely overlapped retrieval and updating during small-batch processing has been suggested as part of SAC Control System upgrading.(3) Here the current version of the file would be preserved throughout each processing cycle, which would produce in a distinct storage area an entire new version of the file reflecting the batch's changes. Meanwhile the batch's information requests would be processed against the current version, concurrently with updating. Results of information requests would thus reflect the current rather than the latest copy of the data base, but would hopefully be available sooner, since the wait for update completion otherwise required would be avoided. The method's other advantages include simplified checkpointing and a less complex file structure than alternatives such as one discussed below.

The approach just outlined requires copying the entire file during each small-batch processing cycle, a time-consuming operation.[12] It would also almost double the file storage space otherwise needed, assuming but two versions of the file would be kept at any time. For typical small-batch processing these difficulties would almost always outweigh the advantages.

Variations of an approach suggested by Waghorn,(12) involving selective replication, appear to merit investigation. Essentially the method would require time-tagging each file record and each change or information request. During updating each modified file record would be marked with the current time, and multiple versions of just those records subject to reference by retrieval requests pending or in progress would be retained. Each retrieval process would use the version of a referenced record last prepared before the retrieval's inception. When through with an old version of a record, a retrieval process could release it (causing it to be purged from the file), provided no incomplete retrieval needed it. To supplement elimination of old records during updating or retrieval, optional scavenger processes could be run periodically to purge the data base of extra record versions no longer needed to satisfy incomplete information requests.

In effect, the method would retain at a low storage cost a version of the file to supply each information request with data current as of the request's initiation. Properly implemented, it could substantially reduce programmed lockout between change and retrieval requests, especially where numerous slow multiple-field retrieval requests would otherwise frequently lock out update access. The method's major disadvantages should also be noted, however. It would substantially complicate file structure and would increase dynamic storage management requirements. It would also complicate and slow both update and retrieval processes, requiring them to sometimes search through multiple versions of the same record (in-line or via chains). These disadvantages should be traded off against the advantages in the light of particular applications' needs.

Replication is seldom appropriate to bypass concurrent-write conflict, however. If different changes were applied concurrently to separate copies of current data, the differently altered copies would need frequent collation to preserve their consistency. To avoid read-write conflict an additional read-only copy would be needed, as discussed above. The collation processes entailed could easily consume all time saved by separate updating. Such distributed updating could also cause incorrect updating results, since the occurrence of some changes can affect the processing of later ones. To apply all changes in parallel to all copies of replicated data would replicate the processing load (unless such parallel updating could be done by appropriate parallel processors). Thus, replication to avoid concurrent-write conflict would seldom be practical.

Replicating read-only data is seldom worthwhile, either. As stated under Segregating Read-Only Data, replication costs copy time and requires extra storage, not always available. Read-only data replication in main memory seldom significantly improves overall efficiency because the storage access conflicts avoided rarely justify the replication costs. Also, unless the copies are stored in separate (core) memory banks, secondary interlock will prevent any improved core memory access time.

Replicating certain frequently loaded read-only data in independently accessible auxiliary storage devices may avoid enough contention over a common storage device to justify its costs. In this case however these advantages and costs should be compared to those of other alternatives, such as use of a single fast auxiliary storage device for such data.

## Combining the Data Distribution Approaches

The three techniques just outlined could be combined advantageously in some systems. For example, an on-line computer program design might provide for the following organization of data in main memory, perhaps accomplished by reformatting data as they were read into main memory.

1. A single copy of all read-only data common to two or more concurrent processes would be stored, segregated from all other data and hardware-protected against erroneous writing.

2. Each process' private data would be segregated from common data and from every other process' private data. Storage protection mechanism design permitting, each such set of private data would be hardware-protected from accidental read or write access by any other process.

3. Except for certain data needed for inter-process communication, the remainder (i.e., alterable data common to two or more concurrent processes) would be replicated selectively as outlined in preceding paragraphs. Thus each read-only process could access a de facto version of the data base current as of the reader's inception.

Programmed lockout would be needed only to force consecutive access to the subset of the data base alterable by two or more on-line processes, or data alterable by at least one such process and also read by at least one other process unable (for whatever reason) to read a copy.

## Summary

All approaches to data sharing control thus far discussed are ways to avoid programmed lockout altogether or at least to reduce its scope. After all such approaches are applied there may still be a residue of data that must be locked: data shared by two or more concurrent processes and alterable by at least one of them. Also, it may sometimes be preferable to lock certain data than to incur the costs of replicating it, otherwise segregating it, or grossly constraining its use to sequential processes.

It appears possible to group all promising approaches to more effective lockout now known, including both process and data locking, into two subsets: (1) lockout scope reduction techniques, and (2) lockout mechanism improvement methods. The proposed work program will address these topics.

## SECTION III

## OPERATIONAL PROBLEMS AND SOLUTIONS

### INTRODUCTION

#### Approach

In Section II, potential problems in shared data use are discussed from the point of view of the data processing system designer. Section III reviews these problems in slightly different ways: (1) from the point of view of the system user (customer, operational user) as they affect his total operation including both the data processing system and its environment, and (2) in a cursory review of actual or proposed systems in search of actual occurrences of these problems or methods used to prevent or allay them. Thus, Section III deals with user procedure problems (as defined in Section I) rather than system error or system performance problems.

In this section, "users" will refer to the people who make operational use of the products of a data processing system, control the character and quality of its operational inputs, and in general, control the environment within which the data processing system operates. They are distinguished from those concerned only with design and with proper and efficient operation of the data processing system. "Data processing system" will mean the complex of equipment, operators, operating system, data management system, and application programs, which in toto perform data processing operations.

## Summary

A data processing system can be alleged to have user procedure problems whenever its users make an unduly large number of mistakes, perform inefficiently, or express substantial annoyance with system behavior, even though the system operates correctly and rapidly enough as measured by its specifications. In general, the trend toward high performance systems that process large data bases quickly with real time input/output streams, multiple on-line terminals, complex file inter-relationships and high reliability requirements, leads to a great increase in all potential shared data problems, including user procedure problems that result in part from users' reduced ability to understand what a system is doing. The situation is exacerbated by concurrent attempts to gain economy in program preparation or efficiency in use of resources through the application of computer programs such as standard operating systems, generalized data management systems, and time-sharing monitors. These programs are generally more susceptible to shared data problems than the consecutively operated specialized programs which characterized some of the earlier high-performance real time systems. (The latter also generally operated on much smaller data bases and had much less flexibility.)

On the other hand, although operational requirements imposed by the user often lead to system designs with a high problem potential, such operational constraints can also either prevent or alleviate problems, or else determine the solution. User procedure problems can be solved, or solutions can be attempted, at three levels: (1) changes in procedures, (2) a redesign of the overall information handling system (including communications as well as data processing), or (3) redesign of the computer programs to prevent or ameliorate the occurrence of such procedural problems outside the system.

The results of our brief survey can be easily summarized: data sharing has not generally, in the past, caused serious problems. Known occurrences, and specific design activities to prevent such problems, are few. The reasons for this are fairly clear: (1) virtually no operational multiprocessing systems have actually run in a multiprocessing mode on significant problems, (2) few operational multiprogramming systems have used true shared data bases, and (3) in the systems where shared data problems could occur, they may have been prevented by operational procedures or constraints external to the data processing system, but not expressly designed to prevent shared data problems. Shared data problems will become more critical in the near future, however, as requirements to process large data bases quickly, in real time high performance systems (e.g., FAA/NAS, AABNCP, MACIMS), lead to increasing reliance on multiprogramming involving extensive concurrent data sharing.

## USER PROCEDURE PROBLEM CHARACTERISTICS

An example, commonly given, from the airline reservation business illustrates user procedure problems. Mr. Able at the AMEX office requests space on Eastern flight 551 to Washington. One space is available, and a reply goes to AMEX stating so. Meanwhile at MITRE, the same request is sent in and the same answer is received. AMEX sends in a reserve order, followed by MITRE. MITRE, of course, receives a reply of "no more space" and is disappointed (assuming the data processing system is working properly and uses first-in-first-out (FIFO) request queuing). The point illustrated is that although the situation is irritating to one or many customers, and may result in overall system degradation as re-requests, rejections, and other associated messages follow each other around and clutter up the system, the operations of the data processing system are perfectly correct and its performance is not necessarily degraded from a strictly data processing point of view. The system is just processing unnecessary data as a result of a <u>user procedure</u> problem.

## Alternative Solutions

Three types of solutions to user procedure problems were suggested above: (1) procedural changes, (2) overall information handling system design changes, and (3) data processing system design changes. The following illustrate each kind of solution for the airline reservation system example: (1) send in only positive reservation requests rather than simple availability requests, (2) add buffer storage and batch requests external to on-line system operation, assigning tentative space reservation to the first request and deferring response to the second request until confirmation of the first, and (3) flag the data base element with a tentative reserve and lock out subsequent requests until the first is resolved.

## Operational Constraints

In an investigation of techniques for resolving shared data problems, one goal should be to make more explicit the relationships or trade-offs between operational constraints and major system design characteristics. In some cases the operational requirements may lead to system solutions which are particularly prone to shared data problems. For example, the need to make a wide variety of flexible, quickly set up queries on a large data base may lead to generalized file structures very susceptible to error if accessed concurrently (as in SACCS/DMS). The demand for better real time performance may lead to multiprocessing as a way to increase processing speed (as in the FAA/NAS), with a higher probability of concurrent access.

The importance of operational constraints, and the effects of altering them on data processing system design and behavior, are not always appreciated. Several examples illustrate this point. As mentioned above, in earlier real time programs (e.g., SAGE and BUIC), the need to get maximum speed out of a single central processor system (among other considerations) led to rigidly sequenced program control, leaving no room for random program operation resulting in data sharing errors. Discussion of the SACCS system later in this section mentions that the need for efficient processing of inputs led to buffering and batching of updates, reducing the potential for data base errors since updating is completed before further queries are processed.

In development of a tactical system an even more radical sort of operational constraint was considered. In this system, one class of input, weather, is so important and may modify so many outputs, that a mode of operation was considered in which if a weather input had come in, outstanding or in-process queries would have been delayed until the weather input could be incorporated in the data base.

In other applications, one use of the system may be so infrequent in relation to other competing uses, that procedural constraints can be imposed on that use without degrading system performance too much, even though they might not generally be desirable. An example of this may be the NMCS-NIPS use of on-line terminal queries, which lock out files from access by the update programs. In this system, on-line terminal operations are infrequent and low in volume in comparison with the off-line batch processing operations. Thus, restrictions imposed only during their operation may not degrade overall performance excessively.

In most current system development activities we no longer start from scratch but are constrained by an existing system and its historical development. In only a few cases are we actually allowed to design or drastically redesign a "total system", but instead deal only with some vertical or horizontal slice such as a new data processing facility. Such requirements to mesh with existing systems and established procedures comprise a major class of operational constraints.

## Importance of Operational Considerations

User procedure problems are extremely important for two reasons, both inherent in the fact that the real system to be designed or altered at the start of a development project is the total information handling system applied to a given set of functions. First, in future development projects, the solution to a problem may be accomplished either within the data processing system or through external procedures. System designers aware of applicable techniques in both areas may be able to trade them off advantageously. Second, in certain existing systems, potential shared data problems were prevented by procedures external to the data processing system. It is most important to note that these procedures may not have been designed to prevent such problems, but may have been implemented for other operational reasons entirely (e.g., organizational relationships, security, response time, processing efficiency). Again, they may just have been established arbitrarily that way during the evolutionary development characteristic of most systems. There is a general need to develop more explicitly the ways in which the user's requirements and constraints affect the choice of system configurations, the resulting potential for shared data problems, and the related costs and benefits involved in relieving them in alternative ways.

## SURVEY OF DATA SHARING IN EXISTING SYSTEMS

This survey is a quick look at the state of data sharing in early 1971. It involves two levels of investigation. One level examines the facilities built into existing systems to prevent shared data problems (i.e., lockout facilities). Another (and much more difficult) level surveys the users to see how these facilities, or other means, have actually been used. Determining whether shared data problems have actually occurred or have been designed away in advance entails detailed discussion with operators on specific applications and may involve decisions which were implicit and not documented. The present report is based only on readily available knowledge of systems and to a lesser extent, on applications.

### Types of Concurrent Data Sharing Systems

Preliminary investigation has identified several types of existing or planned data processing systems in which concurrent data sharing problems could occur or in which mechanisms for avoiding or ameliorating such problems have been incorporated. These are (1) computer operating systems, (2) generalized data management systems with on-line capabilities, and (3) certain application-specific programs. All are of potential interest to the Air Force. Some are part of current or planned Air Force applications. Examples of each type are listed below, and some are discussed more fully subsequently, where documentation is available.

In discussing the operating systems and generalized data management systems we can only define their _potential_ for shared data problems and the methods or facilities which they provide which _might_ be used in solutions in various applications. Several of the application-specific programs should be studied in more detail to determine problem occurrence and solutions. As pointed out under <u>Importance of Operational Considerations</u>, to determine the shared data characteristics of actual applications may be quite difficult due to lack of documentation and to the fact that the problem may have been resolved in the operational environment of the user rather than explicitly in the data processing system.

## Specific Examples

### Operating Systems

| | |
|---|---|
| OS/360: (MVT) | Developed by IBM for single central processor System/360 computers, OS/360 includes a multiprogramming option with dynamic core allocation (MVT). MVT has been further modified for use in a multiprocessing mode (see M65 next). |
| M65: | An experimental modified version of OS/360 MVT that controls dual IBM 360/65s in a multiprocessing mode. |
| ADEPT Executive: | A time-sharing operating system developed by System Development Corporation under ARPA sponsorship that controls a single central processor IBM System/360-50 computer. |
| GECOS-III: | A GE-developed operating system used to control GE 635 computers. |

## Generalized Data Management Systems

| | |
|---|---|
| NIPS: | A group of computer programs, developed by IBM for the Department of Defense (DoD), oriented to batch maintenance and batch retrieval of large files; originally restricted to off-line operation, its IBM System/360 version has recently been upgraded to include limited on-line capabilities. |
| TDMS: | A time-shared data management system, developed by SDC, which operates under control of the ADEPT-50 operating system. |
| GIM: | A group of general purpose data management programs developed by TRW to run on IBM System/360 computers. |

## Applications

| | |
|---|---|
| SACCS: | The SAC Control System - a communications and computer complex applied to quick response control of SAC forces. The part of the system of immediate interest is an experimental on-line data handling and display facility provided by a modified version of TDMS called SACCS/DMS. |
| NMCS: | The National Military Command System in Washington, which includes a data handling facility using the IBM System/360 version of NIPS with on-line terminal update and retrieval capabilities. |

| | |
|---|---|
| ANSRS: | A Defense Intelligence Agency (DIA) system for on-line manipulation of large and small data files to support intelligence analyst activities, implemented on GE 635 equipment using an extensively modified version of GECOS-III. |
| FAA/NAS: | A recently installed Federal Aviation Agency real time air traffic control system which operates on IBM 9020 computers, special versions of IBM System/360 machines, in a multiprocessing mode. |
| AABNCP, MACIMS: | These are planned command control systems under development by the Air Force in which the design approach to shared data problems should be subjected to continuing investigations as the design process continues. Both systems may include multiprogramming and multiprocessing capabilities with large shared data bases. Present designers of both systems are aware that potential shared data problems exist, but specific solutions have not been developed. |

## Discussion

### Operating Systems

The OS/360 Job Control Language(16) provides for data base control and limited data sharing. This control occurs at the job level and is designed primarily to avoid deadlock.(9) To OS/360 the major defined unit in the data base is the "data set", roughly equivalent to a file in other systems. If a processing program is to use a data set, the latter must be represented by a data definition (DD) statement in the job control set for the program. The DD statement controls the handling of new input data, access to existing data, and data output or modification. A DD statement parameter, "DISP" specifies the data set status and disposition. Several values of DISP pertain to existing data sets: The key value is "SHR", which is relevant in multiprogramming modes. If DISP = SHR, the data set may be accessed by other concurrent jobs. However,

> "Once a data set has been given the status of SHR, every reference to that data set within the job must specify the same status or the data set is considered unusable to concurrently operating jobs."(16)

The M65 multiprocessor operating system(15) designers were very sensitive to the problem of multiple use of control routines and control data, and thus lockout of control routines is included as one of the major changes to OS/360 MVT. Indeed, the designers state "The problem of lockout pervades the entire system and is crucial from the point of view of system performance". For example, to prevent erroneous operation of control routines

> "at strategic places in the control program, the processor will test a flag in main storage (the "lock") to determine if...it must wait for the other processor to complete its execution of the code".

Note that this lock controls <u>program</u> execution (i.e., a critical section) rather than data access as such. The more general problem of controlling concurrent access to a shared data base (outside the control programs) is not specifically treated in M65 per se. Prevention of erroneous control program execution is obviously critical to multiprocessor operating system success; in a parallel way, the successful operation of some data sharing application may be equally dependent on similar control of the use of common data.

The GECOS-III time-sharing system provides a facility to state the intent of a program to "CHANGE" a file and allows any other programs access to the file unless a program having a CHANGE status for that file is operating. In GECOS-III,(17) permission for different users to read or write can be attached to any file's "catalog":

> "Multiple concurrent readers (or executors) of a file are allowed by the file system, but any other combination of access-modes are mutually exclusive. The file system accepts or rejects subsequent access-requests for the same file, based on the permission(s) requests by the initial accessor of the file."

Rebuffed requestors must try on their own initiative to get access again; the system does not provide for queuing and retrying rebuffed requests. Also, in GECOS-III, lockout occurs at the file level. As Section II mentioned, such gross lockout can often deny legitimate concurrent access.

## Generalized Data Management Systems

Some data management systems, although designed for time-shared use, avoid concurrent access by effectively completing each task before processing the next. ADAM(5) and GIM(8) are representative of this approach. They essentially allow only "single-thread" processing. NIPS(18) allows multiprogrammed (but not concurrent data sharing) operation. Only one on-line user at a time may access a TDMS data base, because an entire TDMS data base is stored in a single ADEPT file, and the ADEPT Executive allows only one user at a time to access a file. Data management system application to specific problems is discussed below.

## Applications

One major SACCS data handling application has recently been converted from locally generated programs to a more generalized data management system (but not yet installed), using a version of TDMS drastically modified to meet SAC requirements, called SACCS/DMS.(19) Application of TDMS to the SACCS problem offers some insight both into the initial design approach(3) and into the handling of shared data problems in a real operational environment. The initial problem was to apply a general-purpose time-sharing data management system to a real time operation with rigorous response time requirements; a wide but known set of functions; and highly structured data inputs, files, displays, and reports. One significant conversion problem was to improve the data management system's response time in processing the high rate of inputs possible under conditions of great activity. Another was to prevent a shared data problem arising from non-synchronous file updating and retrieval.

High-volume file updates are processed in batches, either off-line (using the MAINTAIN module) or on-line (using the RAPID UPDATE module) from buffered data received from the communications system. "Single-thread" updating using the DYNAMIC UPDATE is also possible. Output requests for displays or hard copy are processed as received, one at a time. SACCS/DMS uses a hierarchical, "inverted" data base structure with many tables cross-related by pointers. As a result, changing the data base during concurrent assess by another program could cause errors not only in individual data items, but in the control tables giving access to them. One proposed solution[14] was to buffer updates and any retrieval requests until the update request presently being processed was completed. Purely from the point of view of successful system operation, this interlocking could be performed at the lowest level of "complete" data base operation, perhaps one message entry.

In actual operation response time requirements for efficient update program execution led to batching of updates, resulting in a system of update, retrieval, update. It should be noted that this mode is forced on the system by operational considerations (response times and system performance), rather than by data sharing requirements. An alternative to speed retrieval was suggested in the TDMS Capability Study(3) and discussed in Section II: a dual file system. That is, a duplicate set of tables would be saved for retrieval while updating was in progress. Besides its expense in storage and copy time, the method would have raised an operational question: would a user not want to wait for update completion rather than retrieving what might be out-of-date data? For whatever reason, the suggestion was not applied. As actually implemented SACCS/DMS is driven by an interface routine which queues all requests and presents them to SACCS/DMS for seriatim processing.

In the NMCS IBM 360 NIPS installation batch processing is multiprogrammed with on-line terminal retrieval from the same data base. A potential for shared data problems exists since equivalent sets of

[14] See(3) p. 32.

NIPS programs may be operating concurrently, one servicing a batch job and another the on-line terminals. The current operational solution is to allow only one on-line terminal at a time and to lock out any file being referenced by the on-line terminal. The on-line capability is an "add-on" to the basic NIPS facility and may not represent optimum design. Use of the on-line capability is limited in relation to the total volume of transactions, mostly batch updates and large scale retrievals, so that restrictions imposed by on-line lockout may not cause significant performance degradation.

The ANSRS system was developed by the Defense Intelligence Agency (DIA) to provide an analyst with on-line file modification and retrieval capability. ANSRS is a multi-terminal system implemented on GE 635 equipment using a modified GE Time-Sharing System under GECOS-III. It is interesting to note that DIA replaced the GECOS file manipulation routines with a subsystem called RAMS (Random-Access Management System) because of weaknesses in GECOS' security provisions. RAMS lets the systems programmer assign a unique identifier to each unit of information filed by the system and allows that data's subsequent manipulation by name. RAMS is designed to operate in a multi-user environment and furnishes certain data set and file interlock facilities. The system will protect data at the record level. That is, users may concurrently access the same data set, but the system locks their access sufficiently to maintain integrity at the record level.

The FAA/NAS system(20) encountered a significant threat to data integrity resulting from simultaneous use of common data areas. Locking was instituted to overcome this problem. Locking in turn caused system interference because subprograms requiring the use of a common data area were delayed by current users. The effect of data locks was also a subject of the NAS simulation study.(21)

In the NAS System both "monitor locks" (i.e., locks of critical sections of the control program) and common data area locks are defined. Both represent resources that cannot be duplicated and to which only serial access is allowed. Monitor lockout is generally of short duration but common data area lockout can persist for a long time and represents "...the most critical system resource".(20) The simulation study indicated that "ultimate system capacity is directly dependent upon the ability of the software to eliminate and reduce locks".(21)

As one of the first true on-line multiprocessing systems with a high potential for shared data problems, this system should be investigated in depth during subsequent shared data research.

## SECTION IV

## WORK PROGRAM

### OBJECTIVES

The prime objective in carrying out the activities indicated below is to help system designers provide data sharing capabilities in multi-user and multiprocessor systems which operate efficiently and correctly. To meet this objective, it is necessary to provide guidelines to system designers for determining the extent to which data will be shared, the potential conflicts in the use of shared data, and the impact of various solutions for correct management of shared data on the system design methodology as well as the system performance. These objectives and the research planned to meet them are discussed in subsequent subsections. The corresponding activities and planned products are next summarized.

To determine the extent to which data will be shared within a computer system, methods will be developed to identify critical sections of processes, the processes' related sets, and the potential related set intersections. A technical report will be produced describing these techniques, and illustrating them with concrete MACIMS application data.

To control the correct use of shared data requires investigation of techniques for managing shared data, including various locking methods. Several such methods will be described and modeled in software or firmware. Their costs in storage space and execution time will be formulated, and their potential for deadlock evaluated. A technical report will summarize the results of this work, and will outline the better deadlock avoidance algorithms and their relationships to different data sharing techniques.

To determine the impact of various approaches to data sharing management is a far more ambitious goal. Consequently, the approach will be illustrative rather than synoptic. A series of several technical reports will be produced. One such report will discuss selected cases, in each case analyzing the impact of different data sharing disciplines on system design, system performance, and user procedures. Other technical reports will discuss the primitives needed for effective data sharing control, how they should best be incorporated into a system of programs designed for effective data sharing, and how data should be structured for optimal data sharing and minimal interference among concurrent processes.

## APPROACH

The work program outlined below maintains a dual emphasis: (1) the development of generalized solutions to shared data problems will provide outputs of maximum and continuing usefulness to existing and planned Air Force programs; and (2) close coordination with specific Air Force developmental programs will gain a double benefit through the injection of realism into the generalized effort and the opportunity to directly affect and improve a system which will become operational. The two systems which seem well-suited to this latter purpose, because of schedule and propinquity, are the MACIMS system and the AABNCP.

## DETERMINATION OF REQUIREMENTS FOR DATA SHARING

In order to correctly manage data sharing in a computer-based, multi-user system, it is necessary to be aware of the potential for the concurrent use of data by more than one process. Such a potential can develop as a part of the operational requirements of a system and as a result of design decisions. In technical terms, the critical sections of processes must be identified, and the related set intersections which may occur during concurrent operations must be specified. This task will relate these requirements to the kinds of information which are incorporated into system specifications and collected during system design. Guidelines will be produced to show system designers what information is necessary and how that information is used.

The conduct of this work will be in close conjunction with the MACIMS design effort which is currently collecting information on data usage. Although this association serves to provide an operational context for this task, it may also aid the MACIMS design task. Static descriptions of the usage of data and of proposed data structures will be used to identify related sets. Other requirements specifications about the frequency and time distribution of events which involve data sharing will be used to estimate the extent to which data sharing might actually occur under operational conditions. For such estimates, a model would be highly desirable. Such a model could initially assume there is no penalty for sharing data and merely measure the number of instances. In tasks described below, the actual costs of data sharing, in terms of techniques used, could be added to the model to show their operational impact.

## TECHNIQUES FOR CORRECT MANAGEMENT OF SHARED DATA

An explicit enumeration is needed of the techniques and algorithms for managing the concurrent sharing of data in computer-based systems. One part of the effort will identify, develop, and describe a variety of locking methods. The implementation of various algorithms will be studied to determine the appropriateness of hardware for realizing them. Use of firmware will be made to model hardware approaches. The rules governing use of various locking methods will also be developed and demonstrated.

A second subject for consideration will be prevention of deadlock due to shared data. Algorithms such as Habermann's(10) must be studied in more detail to determine whether they are adequate and suitable control mechanisms. Comments such as those of Holt(22) on Habermann's work indicate that further analysis is required and further refinement possible. Unpublished exploratory work by Silver of MITRE has also shown that such algorithms must be considered in the context of data structures as well as at the level of resources.

The approach to this task will be to survey existing published techniques and to identify the most promising disciplines while filling in any missing information to make such descriptions more rigorous. Where new techniques with special theoretical or practical advantages suggest themselves, further development work will be carried out.

## QUALITATIVE AND QUANTITATIVE CONSEQUENCES OF DATA SHARING

### Techniques

A decision to incorporate particular data sharing disciplines into a system should depend on whether the consequences are desirable or permissible and on the extent of resources required to build into the system and maintain that discipline. The impact of data sharing disciplines is felt on three levels: the way in which the system design may be affected; the quantitative effect on system performance, in terms of storage space and throughput; and the consequences to the user of the system in terms of his mode of operation. The objective of this task is to determine this impact. The results of this task will be used to benefit the MACIMS and AABNCP systems, if these results are available at an appropriate time.

The application of data sharing disciplines may affect the methodology used for system design as well as the design itself. There appear to be two alternative, but not exclusive, approaches to employing data sharing disciplines: process control and data access control. The exact relationship between these two approaches will be examined as a result of the performance of the following sub-tasks:

## Control of System Processes

The work in this area will answer the following kinds of questions using the techniques described below.

1. What are the primitives needed to describe controls over shared data in the design of a system? Reference will be made to work done in defining semantics for concurrent or parallel processing such as Dennis and Van Horn(11) and Wirth.(23)

2. What effect does data sharing have on the structure of a system design; e.g., how functions are assigned to modules in order to exercise control over their sequence of operation, the amount of data which they might share and the length of time they might require exclusive use of some set of data? Consideration must be given to the importance of identifying critical sections and its effect on the structure of the system. Since the task on Highly Reliable Programming in Project 5550 of the Air Force Systems Command, of which this work is also a task, is involved with the structure of system designs, any tie-in with the results of that activity will be explored.

3. What new functional requirements are necessary in the design of a system to manage the sharing of data, and what is their quantitative effect on system performance? A prime example is the function of preventing deadlock. By making use of the techniques which are developed by the preceding tasks, implementation or simulation can be used to determine the time and space requirements levied by including these functions in the system.

## Data Access Control

How do different logical and physical data structures increase or decrease the potential for data sharing and the extent to which such sharing can be controlled? It is necessary to apply locking mechanisms to typical data structures to show quantitatively the cost of using them. A model appears to be the feasible means for varying data structures and observing their effect on the size of related sets and on the cost of controlling access to related sets with different locking disciplines. Searches will be made to see if there is an appropriate model in existence, such as the model produced by IBM for the AABNCP, or the FOREM model of Senko.(24) If none is suitable, or modifications are required to a model, then this work can be carried out by a subcontractor.

## REFERENCES

1. American National Standard Vocabulary for Information Processing, ANSI X3.12-1970, New York, American National Standards Institute, Inc., 1970.

2. IFIP-ICC Vocabulary of Information Processing, Amsterdam, The Netherlands, North-Holland Publishing Company, 1966.

3. Y. R. Osajima, D. L. Thomas, Lt. Col. C. E. McKusick, and Maj. J. A. Gill, SACCS/TDMS Compatibility Study, System Development Corporation, TM-3941, Santa Monica, California, 31 July 1968.

4. James Martin, Design of Real-Time Computer Systems, New York, Prentice-Hall, 1967.

5. Codasyl Systems Committee, A Survey of Generalized Data Base Management Systems, New York, Association for Computing Machinery, May 1969.

6. A. H. Vorhaus and R. D. Wills, The Time-Shared Data Management System: A New Approach to Data Management, System Development Corporation, SP-2747, Santa Monica, California, 13 February 1967.

7. R. E. Bleier and A. H. Vorhaus, File Organization in the SDC Time-Shared Data Management System (TDMS), System Development Corporation, SP-2907, Santa Monica, California, 1 August 1968.

8. TRW Systems Group, GIM System Summary, TRW Systems, TRW Document No. 3181-A Revision 1, Redondo Beach, California, 15 August 1969.

9. J. W. Havender, "Avoiding Deadlock in Multitasking Systems", IBM Systems Journal, 2 (1968), 74-84.

10. A. N. Habermann, "Prevention of System Deadlocks", Communications of the ACM, 12, 7 (July 1969), 373-377 and 385.

11. Jack B. Dennis and Earl C. Van Horn, "Programming Semantics for Multiprogrammed Computations", Communications of the ACM, 9, 3 (March 1966), 143-155.

12. W. J. Waghorn, "Shared Files", File Organisation, Amsterdam, The Netherlands, Swets & Zeitlinger N.V., 1969, 199-210.

13. P. J. Denning, "Virtual Memory", Computing Surveys, 2, 3 (September 1970), 153-189.

14. E. W. Dijkstra, "Co-operating Sequential Processes", Programming

15. Bernard I. Witt, "M65MP: An Experiment in OS/360 Multiprocessing", Proceedings of 23rd ACM National Conference, Princeton, N.J., Brandon/Systems Press, Inc., 1968, 691-703.

16. IBM System/360 Operating System Job Control Language, IBM Corporation, SRL Publication GC28-6539-9, Poughkeepsie, N.Y., July 1969.

17. GE-625/35 GECOS-III Time-Sharing Programming Reference, General Electric Company, CPB-1514A, Phoenix, Arizona, November 1968.

18. System Description - System/360 Formatted File System, IBM Federal Systems Division, Arlington, Virginia, 30 September 1969.

19. SACCS/DMS Study Group, An Analysis of the SACCS Data Management System, The MITRE Corporation, ESD-TR-70-367, (MTR-1967), Bedford, Massachusetts, 31 August 1970.

20. J. F. Keeley, et al., "An Application-Oriented Multiprocessing System", IBM System Journal, 6, 2 (1967).

21. Reino A. Merikallio and Fred C. Holland, "Simulation Design of a Multiprocessing System", AFIPS Conference Procedings, 33 (1968 FJCC) 1399-1410.

22. Richard C. Holt, "Comments on Prevention of System Deadlocks", Communications of the ACM, 14, 1 (January 1971), 36-38.

23. Niklaus Wirth, "On Multiprogramming, Machine Coding, and Computer Organization", Communications of the ACM, 12, 9 (September 1969), 489-498.

24. Michael E. Senko, Vincent Y. Lum, Philip J. Owens, "A File Organization Evaluation Model (FOREM)", IFIP 68, Amsterdam, The Netherlands, North-Holland Publishing Co., 1969, 514-519.

Security Classification

DOCUMENT CONTROL DATA - R & D

*(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)*

| | |
|---|---|
| 1. ORIGINATING ACTIVITY *(Corporate author)* The MITRE Corporation Bedford, Massachusetts 01730 | 2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED |
| | 2b. GROUP |

3. REPORT TITLE

CONCURRENT DATA SHARING PROBLEMS IN MULTIPLE USER COMPUTER SYSTEMS

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*

5. AUTHOR(S) *(First name, middle initial, last name)*

John B. Glore, Laurence B. Collins, Jonathan K. Millen

| | | |
|---|---|---|
| 6. REPORT DATE JULY 1971 | 7a. TOTAL NO. OF PAGES 88 | 7b. NO. OF REFS 24 |

| | |
|---|---|
| 8a. CONTRACT OR GRANT NO. F19(628)-71-C-0002 | 9a. ORIGINATOR'S REPORT NUMBER(S) ESD-TR-71-221 |
| b. PROJECT NO. 5550 | |
| c. | 9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)* MTR-2052 |
| d. | |

10. DISTRIBUTION STATEMENT

Approved for public release; distribution unlimited.

| | |
|---|---|
| 11. SUPPLEMENTARY NOTES | 12. SPONSORING MILITARY ACTIVITY Electronic Systems Division, Air Force Systems Command, L. G. Hanscom Field, Bedford, Massachusetts 01730 |

13. ABSTRACT

This report summarizes work performed to date under the FY'71 Project 6710 Multi-User Data Management System task. It reviews major problems associated with the sharing of data among multiple concurrent users, and tentatively suggests promising strategies to cope with them. It discusses the importance of solving, ameliorating, or avoiding such problems to the effective development of Air Force systems of this kind. It also outlines desirable future work under this task.

DD FORM 1 NOV 65 1473

Security Classification

Security Classification

| 14. KEY WORDS | LINK A | | LINK B | | LINK C | |
|---|---|---|---|---|---|---|
| | ROLE | WT | ROLE | WT | ROLE | WT |
| COMPUTERS | | | | | | |
| DATA PROCESSING | | | | | | |
| DATA STORAGE | | | | | | |
| INFORMATION THEORY | | | | | | |
| REAL TIME OPERATIONS | | | | | | |
| SYSTEMS ENGINEERING | | | | | | |

Security Classification