Roach - IPC

Multics Technical Bulletin MTB-369

To: Distribution

From: Michael Asherman

Subject: Multics Data Base Consistency

Date: 18 April 1978

Multics Data Base Consistency

contents:

Introduction

Data base and system administrators too frequently find themselves bothered by inconsistent files. The fraction of money, programming effort, and machine time spent just on this problem may well be half the cost the system, yet we still lack a satisfactory solution for certain infrequent, but inevitable hardware failures where volatile memory is lost. The following is a proposal to change Multics so that it can handle having its plug pulled. The solution comes as a by-product of protection from arbitrary causes of data destruction.

Purpose

The purpose of this document is to bring a serious problem to the attention of the small group of Multicians who can implement its solution. The first step must be to recognize that we have a real problem. I hope to persuade the reader that a complete, efficient solution also exists. The next step is yours.

Background

I offer only a sketch, because I am unfamiliar with the innermost workings of the system. My ignorance prevents me from making even a crude guess as to the amount of reprogramming or possibly hardware changes that will be required to implement these changes. Nevertheless, as the one responsible for maintainaining and improving vfile_, I am compelled to respond to increasingly frequent complaints from justifiably irate victims of system failures.

trouble reports

The rate of reports of damaged files showing signs of damage due to unsuccessful Emergency Shut Down (ESD) is approaching one per week; this figure will undoubtedly increase as more Multics systems are sold, and as the number of large shared data base applications grows.

MIT crash statistics

MIT has reported the following statistics for ESD failures:

1978	3 (as of 3/23/78)
1977	15

damaged files

More than one of these occasions damaged the permanent syserr_log. Similar difficulties have been reported elsewhere, for example at Phoenix on the heals_log, as well as the syserr_log.

Types of Inconsistencies

I treat the following potentially undesirable situations as distinct types of inconsistencies:

1. invalid snapshot

The data base appears to be in a state which does not correspond to a snapshot for any time; pages comprising it are unpredictably "out of synch" because of a failure to flush core before its loss. Having a snapshot implies that the exact sequence of modifications made by programs is preserved.

2. time lag loss

The data base is a valid snapshot, but not its most recent image.

3. interruption

The data base is a valid snapshot, but of an intermediate state of a complex operation that is supposed to appear atomic.

4. any other cause of bad data

The current image is unsatisfactory for whatever reason.

summary

Each case is considered below, but I stress the first class, because Multics is most deficient in this regard, and the solution to this problem will greatly improve our handling of the latter types of inconsistencies as well.

Illustration

type 1

example

Suppose that an initially empty segment is modified by turning on each bit, left to right. The only permissible images of such a file have the form:

111111111111111111111111...10...0000000000000000000000

i.e some number of 1's followed by zeroes to the end of the segment.

If concurrent references are causing paging activity on this file, then at any given moment the non-volatile disk image, when taken alone, may not be consistent in the above sense.

Specifically, the disk pages might show:

page 0: 000...0 (this page pinned in core by heavy use) page 1: 111...1 page 2: 111...1 page 3: 111...1 page 4: 110...0 page 5-end of segment: 000...0

If a page is written to the disk while a less recently modified page remains unwritten (because of a more recent reference, for example), then the disk image may cease to be a valid snapshot of the logical file for any point in time. An ESD failure at such a point might leave the only copy of the file in an unpredictable, seemingly impossible state.

an apparent solution

At first glance it might seem that we can get out of this trouble by using a least-recently-modified page removal algorithm, instead of the present LRU; unfortunately, this doesn't work, as can be seen from the following:

step 1: modify word 0 of page 1 step 2: modify word 0 of page 2 step 3: modify word 1 of page 1

If none of the modified pages have yet been written out, and at this point a page must be expelled, then we are in trouble, since the least recently used page also contains a more recent modification. In other words, neither page can be written without causing at least a transitory inconsistency of the disk image.

type 2 example

As an illustration of the second type of inconsistency, suppose that a ready message is printed or some other visible side-effect occurs indicating the completion of this operation. If the most recently modified page is not written out before an ESD failure, then the file is inconsistent in the sense of a time lag loss of information, even though it may be a valid snapshot.

type 3 example

If the user wishes to regard this as an atomic operation, then its partial completion must be regarded as an inconsistency of the third type, even though the most recent snapshot may reside on disk. In other words, the inconsistency lies in the file's being in an unallowable, albeit well-defined, state.

type 4 example

Finally, barring any of the other types of inconsistencies, there is still the possibility of file damage, for example because of an act of God, such as a disk struck by lightning. More typically, the user might have clobberred his file by typing the wrong command.

Type 1 Inconsistencies

summary

Briefly, I propose that files have generations of file maps, and that pages written out not immediately overrite their previous disk images. Each saved file map, except possibly the current one, would be guaranteed to describe a consistent image of the file residing entirely on disk. The number and frequency of previous files images retained could be controllable either by setting branch attributes, or by using explicit calls. For the sake of simplicity, we shall assume at most one additional file map in the following discussion.

directories and msfs

The task of supporting generations of directories and msf's is going to have to be tackled in order to handle large data bases. Implementing the needed hardcore changes might be a big job, but there should be no great conceptual difficulty in extending the idea of generations of consistent file maps to that of generations of sets of file maps.

recovery

After an ESD failure or any other possible cause of damage to a file, one of the snapshot images always must be used. The retension of a file generation guarantees that a valid snapshot is immediately available online at all times. Disregarding any other types of inconsistencies for the moment, recovery is therefore guaranteed and immediate.

implementation

file maps

Page control must note that a file map becomes consistent when all its pages have been flushed. Before the next modification to this file, a new file map may be initialized by copying the old one; each page table word (PTW) of the new map would then be marked to indicate that the corresponding page is shared by a previous image. A consistent snapshot can always be obtained without any delay by resetting the current file map and freezing the old one. The unwritten modifications belonging to the frozen image may then be written out to guarantee saving a consistent copy at least up to this point in time.

page control

In order to keep from overriting pages belonging to a previous file generation, page control could use the following scheme, assuming that a prior snapshot is always maintained:

passive references

Proceed with normal paging logic, leaving the PTW's shared bit unchanged.

first modification {shared bit is on}

Assign a new address for the current image of this page; then replace its PTW in the current file map with the shared bit set to zero.

subsequent modifications {shared bit is off}

Proceed with normal paging logic.

disk I/O

With regard to the first class of inconsistencies, there is no need wait for any I/O's to complete in order to resume activity on a file once a new map is initialized; i.e. maintaining prior disk snapshots does not interfere with normal file activity. Although there may be a number of modifications belonging to the frozen image, as well as the file map itself to write out, these I/O's can take place concurrently with modifications to the current file without jeopardizing the consistency of earlier snapshots.

usage

The use of generations would, of course, include the present (and only) option of having just one file map, i.e. the case of zero consistent prior images. Presumably one would use generations only for permanent data bases.

typical

The most common usage involves far more passive references than data base modifications. Consequently, there are many applications in which the natural paging behavior leads to an acceptable mean time between file generations.

highly active files

For extremely large, heavily modified files, the frequency of file generations might be deliberately regulated by periodically freezing the current file map and setting up a copy as the new current map. Note again that it is not necessary to wait for any I/O in order to accomplish this switch; the only requirement for snapshot consistency is that a map be correct when it is written out.

cost

extra virtual storage

page images

The amount of additional virtual storage required to save pages of a previous file generation is given roughly by the formula:

$X = T M$,

where: X is the number of extra pages of the previous snapshot which must be retained in addition to their counterparts in the current file map.

T is the mean time between generations, i.e. 1/generation frequency.

M is the average rate of page modifications.

This approximation assumes that references are in a uniform random distribution across the file, as one might expect in the limit of large files and small values of T; to the degree that this assumption does not hold, X will be smaller. Note that for sufficiently small T, X is limited by processor speed and becomes independent of the file's size.

old file maps

The extra online storage for old file maps can be kept small in comparison with the overhead for the file pages by which they differ. This may require the use of a difference representation for old file maps in the limit of large files and small T (i.e., few differences between generations). Basically, enough virtual storage for two file maps is required, but only the old one need ever be on disk. Only active file maps need ever be in core.

extra I/O

The number of additional I/O's to disk required for saving a snapshot of each new generation is given by the formula:

$I = 1 + P$,

where: I is the extra I/O's per generation.

P is the average number of modified pages pinned in core for a time exceeding T, the mean time between generations.

One additional I/O is required to save the frozen file map if the file is active most of the time.

extra processing

Each time a file map is frozen and a new one initialized, there will be some of additional processing required to make the switch. This expense is the limiting factor on setting arbitrarily small time intervals between generations. Since file maps are orders of magnitude smaller that the files they describe, it is reasonable to suppose that seemingly small time intervals on the order of a minute between generations will still be very large in comparison to the limit imposed by the cost of map initialization. Thus the extra processing to support generations with frequencies in the range corresponding to anticipated usage can be kept negligible.

summary

Even in relatively large active data bases, online backup copies probably can be maintained within seconds of the lastest version at a tolerable cost, without interference from/to normal file use. The additional storage requirement only applies to pages which differ between images. Thus a higher frequency of generating file maps reduces extra storage overhead, in exchange for a gradual increase in I/O and processing. Where the cost is greatest, in heavily modified files, the need for protection from inconsistency is most urgent, since these applications are most likely to suffer from eventual system failures.

example

locality of references

Consider a typical transaction processing application with the following paging behavior, described in terms of the distribution of page modifications by mean time before removal of the page from core:

maximum time core-resident	% of modifications
.1 second	50%
1 second	90%
10 seconds	99%
100 seconds	100%

estimated additional cost

Assuming that each transaction modifies one page, the following table estimates the extra cost of maintaining a consistent disk snapshot:

	T = 1 second		T = 10 seconds		T = 100 seconds	
Transaction Rate	pages	I/O's	pages	I/O's	pages	I/O's
1/sec	1	1.1/sec	10	.11/sec	100	.01/sec
10/sec	10	2/sec	100	1.1/sec	1000	

						.01/sec
100/sec	100	11/sec	1000	10.1/sec	10000	.01/sec

Type 2 Inconsistencies

summary

To entirely eliminate the risk of time lag inconsistencies, it is both necessary and sufficient to have after image journalization at the highest level of accepting transactions. One may also have lower levels of after image journalization and incremental file backups in order to speed up recovery, but this is not strictly necessary for any other reason.

recovery

The recovery procedure restores the latest available snapshot of the data base, and then must reapply any transactions journalized subsequent to the time of the snapshot. It is not necessary that the snapshot to be rolled forward be consistent in the third sense to do this, i.e. any snapshot will suffice, so long as one provides a separate mechanism for causing data base operations to behave atomically.

implementation

journalization

vfile_

An option will be provided allowing vfile_ users to specify a separate sequential journal file where after images are automatically written prior to each file modification. If the file being journalized is used within a transaction, after images would be saved at checkpoint time; otherwise, journalization occurs before each modification. It is necessary to wait for the completion of I/O's to the journal file if one expects to be assured of recovery to this point in time.

page control

One may also use a lower level form of journalization to accelerate recovery by providing a more recent snapshot than the latest available complete backup copy. This entails having page control write after images of modified pages to a separate sequential file before writing the the data base changes to disk. A mark must be made in the journal file at each point all modified pages are known to have been written out. In order to obtain a more recent snapshot of the file, the latest snapshot, whether an online generation or an incremental backup copy, would be rolled forward from journalized page modifications up to, but not beyond the latest mark defining a point of snapshot consistency. This procedure might be sped up by using an index into the journal file.

incremental backup

with online generations

Incremental backup is greatly simplified by having a frozen file generation online. The complete snapshot may be copied in one pass, without interference from or to activity on the current file image. Of course one must not free any old maps or their pages while backup is in progress.

with low level journalization

When low level after images are journalized, incremental backups can be produced entirely from the journal file and previous backup copies. An old backup copy would be rolled forward by applying all subsequent modifications journalized up to, but not beyond a more recent point of snapshot consistency marked in the journal file.

without online generations or page journalization

If the file is not frozen, saving complete file snapshots becomes somewhat trickier. After a complete pass through the file, some of the pages saved early in the pass may have undergone modifications subsequent to saving other pages later in the pass. This necessitates resaving those pages which were modified during the previous pass in another pass through the file map, etc., until a pass can be made during which no additional changes occur. Note that the number of pages that must be saved in each subsequent pass will tend to decrease, i.e. this procedure tends to converge, assuming a moderate, uniform rate of file modifications. Specifically, the mean number of modifications occurring in the time required to save one page must be less than one. Of course, one could intermittently lock and unlock files to reduce the rate of modifications to an acceptable level, but this sort of interference is undesireable.

usage

If one is willing to settle for limited time lag losses, high level journalization may be avoided. This compromise may be particularly attractive with the small time lags achievable through online file generations.

cost

normal use

journalization

highest level

The minimum overhead of protection from time lag losses in normal use is the cost of the highest level of journalization. Assuming that the accompanying data base changes are considerably more complex than the operation of saving their highest level specification, the greatest component of the cost of journalizing at this level is the additional I/O on the journal file. However, this is at most one extra I/O per transaction, and may be even further reduced if transactions are aggregated before their journalization and acknowledgement of receipt.

lower levels

Low level after image journalization generally implies at least as much extra I/O as required by the highest level of journalization. The only justification for this cost would be in those cases where the expense of saving complete snapshots at sufficiently frequent intervals is even greater without journalization. Having online generations eliminates the need for incremental file backup as long as we are disregarding the fourth class of inconsistencies. Otherwise it is strictly a question of trading off overhead in normal use versus the cost of recovering after a crash.

incremental backup

Incremental backup is an unsatisfactory substitute for using online file generations as a means of speeding recovery from time lag losses in large active files. The reason is that the cost of incremental backup is proportional to both its frequency and the file's size, which makes keeping fairly recent complete snapshots prohibitively costly in these limits. Furthermore, unless low level journalization is used, there is the additional problem of interference from/with current file usage. On the other hand, none of these factors have so direct an impact on the expense of maintaining online file generations.

recovery

The cost of recovering to the most recent file image consists of retrieving the most recent available snapshot and reapplying any transactions journalized after that point. If an online generation exists, there is no additional cost to obtain a snapshot, and the entire expense is therefore proportional to the mean time between generations, which can be kept small. Without generations, one must resort to offline retrieval of incremental backup copies, and rolling forward from low level journals as means of reducing the number of transactions that must be reapplied. This implies considerably more cost to retrieve a snapshot, and considerably more transactions to reapply unless frequent backups or low level journalization is used.

Type 3 Inconsistencies

summary

Using atomic operations is the key to avoiding inconsistencies from interruptions. The problem is achieving efficiency with minimal programming effort. Depending upon the level of programming involved, there are appropriate methods of guaranteeing atomicity. These techniques culminate in a general transaction encapsulation facility, which permits one to make arbitrarily complex atomic data base operations.

atomic operations

A modification is atomic if it has no detectable intermediate states. This appearance ultimately always derives from the hardware atomicity of a limited set of machine operations. Restricting access of data structures to a well-defined set of interfaces can create the appearance of atomicity with respect to this constraint. Thus, strictly speaking, even "absolutely" atomic hardware operations are atomic relative to the constraint that one use only the standard operations; sufficiently microscopic examination invariably will reveal a continuum of states.

recovery

Unless one entirely avoids the possibility of interruption by a specialized low level method, the higher level data base interfaces must be prepared to automatically detect and adjust partially completed operations. This is accomplished by checking for such states at every synchronization point, i.e. at opening time and at the start of every shared data base operation. The adjustment is performed by either undoing or completing any interrupted modification before proceeding with a new operation.

implementation

lowest level

At the lowest level, there are a few inherently atomic hardware operations. This means that their use can never lead to inconsistencies of the third type. In order to make a procedure atomic, the data representation and program logic are designed around the machine's instruction set. See appendix B for some examples of low level coding techniques.

intermediate level

Vfile_ uses a more general method of making atomic operations which imposes few constraints on the choice of algorithms and data structures. This technique, described in appendices A and B, requires that programs be modified from their basic logic so as to execute in either a normal or a "repeating" state, during which a prior operation is retraced to the point of its interruption, possibly bypassing certain "protected" procedures on the way. The normal execution is required to periodically increment a permanent tracking variable, and other program variables must be classified as either protected or reconstructed. By observing the rules outlined, arbitrarily complex procedures can be made atomic.

highest level

The new transaction processing facility permits users to make complex atomic procedures with essentially no reprogramming. For the initial implementation, the data base must be a collection of indexed files manipulated through vfile_. Records and index pages have a format which permits their having both a before and an after image. Transactions are made atomic by representing their state of completion through a flag on an index entry in a control file shared by users of the data base. See MTB-370 for details.

usage

lowest level

This method is most appropriate at the innermost level of system programming, e.g. for page control, where optimizing performance is the paramount concern, and the kinds of procedures involved are relatively simple. Because of the difficulty of this kind of programming, its use should be avoided when higher level techniques are available.

intermediate level

The intermediate solution belongs in the realm of those system programs which underlie the implementation of a high level transaction facility, but which are too complex to program at the lowest level.

highest level

Transactions are the method of choice for all applications where they can be used. This includes both user programs and any system functions that are built on top of vfile_. The purpose of the more cumbersome lower level techniques is to enable the system to support such a facility.

cost

lowest level

machine

When an operation is naturally expressible as a single atomic hardware instruction, the cost of atomicity is minimal, assuming the hardware is efficient. However, to the degree that such an implementation compromises the choice of an optimal algorithm and data representation, this is a deception.

programmer

Programming techniques at this level tend to be highly specialized, requiring the most effort and ingenuity, since the constraint on the available operations is greatest. As the complexity of procedures increases, eventually the cost of programming applications by such methods becomes unthinkable.

intermediate level

machine

Some additional processing and storage is needed in for this method of achieving atomicity, but experience has shown it to be small in comparison with the basic expense of the operations involved. For the case of vfile_, I would estimate that less than 15% of the cost of a modification is processing overhead needed to achieve atomicity. The additional storage requirement is negligible in comparison to the size of even a modest file. Furthermore, these costs are fixed per modification; they do not depend on either the size or degree of activity of the data base.

programmer

This solution is a compromise between the low and high level mechanisms, both in terms of efficiency and reprogramming. Its principle benefit over the former lies in the fact that one is not generally required to modify the basic logic and data representations, except in a superficial way, which is arrived at by systematic program transformation guided by a few general rules.

highest level

machine

Use of the transaction facility implies a small additional expense in storage and processing compared the the basic cost of performing typical operations. The storage overhead comes from reserving several words for each record and index page to keep track of before and after images. There is also a temporary need for additional storage during the course of a single transaction, resulting from the retension of before images of modified items to allow for a possible rollback. The extra processing requirement is associated with manipulations of the transaction control file and temporary reference lists. So long as transactions are short enough to complete before the pages they touch are removed from core, no additional disk I/O is required.

programmer

The advantage of transactions is that they permit one to construct atomic operations with the minimum conceivable reprogramming of any vfile_ application. All that is required of the user is to specify what procedure, e.g. a command line, is to be invoked as a single atomic operation.

Type 4 Inconsistencies

summary

Ideally, one would like to have an archive of all snapshots of every data base for their entire histories, if the cost could be neglected. Given the mechanisms for dealing with the first three types of inconsistencies, the only problem that remains is to adjust the parameters governing them, so as to arrive at an acceptable compromise between the cost of recovering and the overhead in normal use.

recovery

The recovery procedure consists of retrieving a previous complete data base snapshot, and selectively reapplying subsequently journalized transactions.

usage

Saving journalized transactions gives the finest resolution one really needs between old snapshots. However, the time to recover from the journal alone may be excessive, unless occasional complete snapshots are also made available. The user can control this expense through the following parameters:

number of online generations
frequency of generations
frequency of incremental backup
use of lower levels of journalization

If one is willing to accept less than perfect resolution among backup copies, the high level transaction journalization may be forgone. At the other extreme, paranoid users could have redundant journalization (in parallel), to compensate for any estimate of hardware unreliability.

cost

Storage is obviously going to be the biggest cost, which has to be weighed against the time and ability to recover an acceptable snapshot. Users control the the parameters deciding these factors, and they should bear the expense as an incentive to choose efficient solutions. For example, this applies to quota for online file generations. The space required for journal files is minimized by avoiding excessive low level journalization and using compact, difference representations for changes (e.g. field level as opposed to record level journalization).

Summary

There is at present a serious hole in Multics reliability. Reliability is like silence; any disturbance spoils it, however quiet the rest of the world may be. One cannot sensibly assign a cost to intermediate degrees of reliability. Either you've got it or you don't. People who really care are not going to take chances; they will go to whatever lengths they can afford to insure that any forseeable data base calamity is, for all practical purposes, impossible. This has led to a myriad of inefficiencies that have wrongly come to be associated with the idea of protection from file inconsistency.

Multics does not have to settle for limited reliability if we can take the E out of ESD. I have outlined a complete solution to the problem of data base inconsistencies, which hinges on solving this basic problem. The point of my broad presentation of this subject has been twofold: first, to show that there really is a complete, rigorous solution to the problem of consistency, thereby putting the particular problem of ESD in context as the key weakness in the present Multics system; second, I have made a reference guide to assist in producing an efficient implementation of reliability safeguards.

In contrast to what many users have come to expect, one does not have to pay a high price for reliability. Nor is it necessary to sacrifice the advantages of a virtual memory environment; in fact this feature makes Multics potentially more efficient than systems which must journalize before images. Another inefficiency often contemplated is the needless interference of consistency mechanisms with normal usage, which tends to become increasingly evident in large, shared, heavily modified files. It would be most unfortunate if a serious performance degradation were introduced in the limits for which Multics is otherwise so well suited. Fortunately, there seems to be a simple solution that does not suffer from interference, and which permits rapid recovery. Furthermore, the problem can be solved once and for all. No additional programming will be required to maintain data base consistency once we have a reliable transaction processing facility.

The highest priority must be given providing the missing elements of reliability underlying the file system. I have cited the particular problem of ESD as the crucial unsolved one. There may also be some inadequacies in the present backup and other file system programs, but I don't know to what degree this is the case. In particular, incremental backup must always obtain snapshots, and this should not suffer interference to or from current file use. The file system and underlying primitives should behave atomically, in an efficient and rigorously correct manner. Any system which depends upon heuristic salvaging as a solution to this problem is less than ideal. If we intend to support large, reliable transaction processing applications, then these changes must also be applied to directories and msf's. However much work is involved, we cannot ignore such a serious user concern.

Appendices

A. vfile_ interruption recovery program logic

The following is an excerpt from the Multics User Ring I/O System PLM, (AN57), The vfile_ I/O Module.

E. notes on interruption recovery

These notes were used for a presentation to a Multics staff meeting, in which I outlined a general solution for recovery from interrupted operations, as implemented in vfile_.

RECOVERY FROM INTERRUPTIONS

Introduction

It can happen that while vfile_ is modifying a file, its execution is interrupted and not resumed (e.g., the system crashes). This can leave the file in a state where new operations cannot be performed, e.g., a node has been split but the new entry has not yet been made in its parent node. The program vfile_ has been coded so that the next time the file is used, the interrupted operation is automatically completed.

This feature requires the use of a substantial portion of each file header and a separate restart procedure. The rest of the mechanism is embedded throughout the file-altering sections of vfile_.

A uniform strategy applies, except in a few simple special-case situations. File-altering operations are designed to execute in either of two states, normal or repeating. In the normal state, each operation keeps track of its progress by saving certain variables in the file header. When an interruption is detected, the restart procedure reinvokes the interrupted operation in the repeating state. This results in the completion of the interrupted operation, whereupon the restart procedure returns, and the operation that detected the inconsistency proceeds normally.

The Normal State of Update Processing

The distinction between the normal and repeating states is made through the variable indx_cb.repeating. At opening, its value is set to "0"b, indicating the normal state.

On each file alteration, a certain amount of additional processing is done that is extraneous to the actual transformation that results. This extra work guarantees that any intermediate state of execution can be reconstructed and correctly resumed, provided only that the file itself is preserved intact.

For this purpose, two kinds of data are periodically saved in the file header during each update operation. First, there is the information that keeps track of the nature and degree of completion of an operation. Second, various external variables are saved that might otherwise perish with the user's process, or perish because of a subsequent assignment during the current operation.

Tracking Variables

In order to keep track of each operation's progress, the following variables are used:

file_base.file_state_block.file_action
indicates which file-altering operation, if any, is currently in progress.

file_base.file_state_block.file_substate
is a counter indicating how far the current operation has come toward completion.

file_base.index_state_block.index_action
indicates which kind of index change, if any, is in progress.

file_base.index_state_block.index_substate
is like file_substate, but applies only to the index alteration phase of the operation.

For each update operation, there is a corresponding file-action code that is set just before and cleared immediately after the file transformation takes place. Similarly, each index alteration is associated with a nonzero setting of index_action.

The substate counters are zeroed and periodically incremented during every transformation. By minimizing the frequency of substate changes, additional processing is reduced. This optimization, as it turns out, is largely achieved through otherwise arbitrary choices in coding style, such as the order of independent assignments.

Other Header Variables

The action and substate variables just discussed make up only a small part of the file header. Somewhat more than one page is reserved for the rest of the recovery-related file variables.

The remaining header variables are used during normal execution to save copies of certain other variables. Arguments and other external nonpermanent information that can affect the subsequent operation, e.g., file position, must be saved before any inconsistency is introduced. This precaution is required by the condition that the recovery mechanism always completes an interrupted operation. The other variables that must be duplicated are those permanent file variables that are altered subsequent to their affecting the course of the transformation.

Several optimizations apply to the saving of variables during updates. For example, the record argument is not saved during write and rewrite operations. This exception is handled by automatically deleting or flagging the record after restarting. Although it may be necessary to save many variables in a single update, the duration over which a given value must be saved is often shorter than the entire operation. Thus, a single header variable can serve as a repository for any number of separate values during the course of one operation. Another optimization that substantially reduces the cost of saving variables takes advantage of the efficiency of multiword assignments on Multics hardware.

The Restart Procedure

Whenever an entry point sets a file's lock, the header variable file_action is tested before proceeding with the body of the operation. If file_action is nonzero, an inconsistency exists in the file as the result of the interruption of a previous update operation. This situation is detected upon opening and at the start of shared update operations. It is dealt with simply by calling the external procedure restart.

The restart procedure performs the following simple tasks:

1. Saves the process information describing the state of the current opening (variables in the structure iocb.open_data_ptr->indx_cb).
2. Restores some arguments and process information for the interrupted operation, using values saved in the file_header.
3. Sets the variable indx_cb.repeating to "1"b and reinvokes the appropriate entry in open_indx_file to complete the interrupted operation.
4. Finally, after returning from the restarted operation, the process information for the current opening is restored and a return is made.

For the write_record, rewrite_record, and record_status operations, some additional steps are taken. In the case of rewrite_record, the user may be alerted to the potential inconsistency of the record's contents. For the other two operations, the new record is automatically deleted immediately after finishing the interrupted operation. This special treatment is required on writes and rewrites because efficiency considerations preclude saving the buffer argument at the start of every update.

The Repeating State of Execution

The last major feature of the recovery mechanism is the alternate state of update processing, characterized by the setting of indx_cb.repeating to "1"b. This situation only arises as a result of the detection of an interruption and invocation of the restart procedure discussed in the previous section.

What will ultimately be shown is that the result of reinvoking any interrupted operation in the repeating state is the same as it would have been, had the operation run to normal completion. Furthermore, the process of recovery must also be completely restartable.

To guarantee the correctness of restarting as described, it is sufficient to show that some set of conditions exist such that the total machine state (relevant to an operation) that existed just prior to any interruption is somehow reconstructed. The term "machine state" refers to both the state of execution (level of procedure invocation, for example) and the values of all variables that can subsequently be referenced. Since we are presumably dealing with a deterministic system, the replication of any prior state must produce the same outcome.

The essential difference between the two states of processing is that certain portions of code are bypassed in the repeating state. Otherwise, the flow of control is identical to that of normal execution. In restarting an operation, the repeating state automatically reverts to the normal state before reaching the point of interruption. Thus, the repeating state only applies to portions of code previously executed.

Sections of code to be skipped in the repeating state are embedded in internal procedures of the following form:

```
(a "protected" procedure)
routine_x:proc;
     if indx_cb.repeating then do;
          call check_file_substate;
          return;
     end;

     (body of procedure
      executed only in
      the normal state)
          .
          .
          .
     file_base.file_substate=
          file_base.file_substate+1;
end routine_x;
```

where check_file_substate is the following procedure:

```
check_file_substate:proc;
     indx_cb.next_substate=indx_cb.next_substate+1;
     if file_base.file_substate=indx_cb.next_substate
     then indx_cb.repeating="0"b;
end check_file_substate;
```

Also, each update entry in open_indx_file starts with a call to the following internal procedure (some details omitted for clarity):

```
initialize_substate:proc;
     if indx_cb.repeating
     then if file_base.file_substate=0
          then indx_cb.repeating="0"b;
          else indx_cb.next_substate=0;
     else file_base.file_substate=0;
end initialize_substate;
```

Flow of Control

Half the problem of reconstructing the interrupted machine state is getting back to the right location in the code. If the program were completely linear, i.e., without any internal procedures or do loops, then a simple transfer would suffice. In general, the skipping mechanism used with the repeating state achieves the same end without the requirement of linear program flow. The correctness of this technique, however, does imply certain constraints.

To guarantee that flow of control returns to the point of interruption, it is required that the original path be followed, deviating only when it is certain that the bypassed code has already been completely executed, and in such cases always returning to the original path. Any control-altering statement that is repeated must therefore have the same outcome as before. This implies that any variables upon which a control-altering statement depends must be restored before the statement is repeated. Conversely, any control-altering statement that depends on a variable whose value can have changed must be skipped in the repeating state.

Reversion to the Normal State

The function of the internal procedure check_file_substate and the temporary counter indx_cb.next_substate is to ensure that the transition from repeating to normal execution takes place at the right moment. Strictly speaking, the right moment to stop skipping sections of normally executed code is the point after the last machine instruction executed before the interruption occurred. In general, however, some number of prior instructions can be repeated without altering the outcome. The permanent substate values delimit sections of code according to this property. Thus, for an interruption anywhere within a section of code corresponding to a single substate value, it is sufficient to revert to normal execution just prior to entering that section, or "logical block," of code.

The next_substate in the repeating state is initialized and periodically incremented so as to correspond to the normal substate value for the upcoming logical block. This practice allows the logical block of an interruption to be found simply by comparing next_substate with the permanent substate saved in the file_header. However, it should be noted that the mechanism for incrementing the next_substate described earlier introduces the constraint that such "protected" procedures not be nested. For this reason, a second permanent substate counter is used in the procedure change_index. Evidently, the use of multiple permanent substate counters effectively removes the constraint against nesting protected procedures.

Restoration of Variables

Having described the mechanism whereby flow of control returns to the point of interruption, it remains to be shown how the program variables are correctly restored to their previous values at the instant of reverting to normal execution. For this purpose, the variables are divided into two classes, distinguished by the constraints they impose upon protected procedures. All program variables upon which the completion of any update operation depends are required to fall into one of these classes.

Reconstructed Variables

A variable is "reconstructed" if every assignment to it is repeated and produces the same outcome as that prior to interruption. Thus a reconstructed variable cannot appear on the left of an assignment statement within a protected procedure. This definition guarantees that at any reference to such a variable while repeating, its value is the same as it was during previous normal execution. It follows, therefore, that when the reversion to the normal state takes place, all reconstructed variables have their former values, as required.

Protected Variables

A variable is "protected" if every assignment to it (except possibly the last) is skipped in the repeating state. Its value will therefore remain unchanged between the time an interruption occurs and normal execution is resumed. Protected variables must reside in the file, since only the file is assumed to be preserved.

A file variable can be protected first and then reconstructed, but not vice-versa. This constraint prevents any interrupted recovery from altering the protected value until it is no longer needed.

Statements that are repeated must have the same outcome in order to correctly reconstruct the interrupted machine state. This implies that no repeatable statement can depend upon any subsequently assigned protected variable.

The basis for subdividing the program into logical blocks, each corresponding to a substate value, lies in the dependencies on protected variables. Specifically, a single logical block is required to be independent of any protected variables subsequently altered in the same block. Otherwise, the outcome of reexecuting a block would depend on the point of interruption inside the block, which contradicts the defining assumption stated earlier.

Repeating State-Summary

Another point that was noted earlier is the requirement that the process of recovery from interruption itself be interruptible in the same sense. Fortunately, this problem has already been solved through the assumption that all variables are either reconstructed or protected. Since the file is thus constrained from changing its state until normal execution resumes, the only nontrivially distinct intermediate states are those associated with normal execution. Therefore an interrupted restart is always recoverable through the standard recovery mechanism.