Database Consistency under UNIX*

H.L. Viktor

Department of Computer Science
University of Stellenbosch
7600 Stellenbosch

Abstract

A recovery technique to facilitate the recovery from system failures should ensure a consistent database state at all times. The UNIX *write* system call uses a delayed-write policy. Data blocks are kept in a buffer cache and are later asynchronously written to disk. If a system failure occurs, logically complete transactions may be lost and the database may contain incorrect old data values. If a physical write was being executed when the failure occurred, the database will be inconsistent, since an update in place policy is used when writing to disk. Differential files can provide database consistency efficiently.

CR Categories: D.4.3, H.2.2

Keywords: Consistency, recovery, system failure, UNIX.

Introduction

The NRDNIX distributed database management system (DDBMS) [5], currently under development at the University of Stellenbosch, utilizes a redo, no-undo recovery algorithm to facilitate the recovery from system failures. In the event of a system failure, the contents of main memory are considered lost. A previous consistent database state is constructed and a logical logging facility is then used to redo transactions that have since been committed. The DDBMS is implemented on top of the UNIX operating system and accesses the physical database through standard system calls.

The UNIX *write* system call uses the delayed-write policy. To reduce the amount of I/O traffic, accessed data are kept in a buffer cache. When a *write* system call is issued, the corresponding buffer is marked accordingly. The actual propagation of the data is performed at a later stage. If a system failure occurs, there could be logically complete but physically incomplete I/O requests in the buffers. These will be destroyed, causing inconsistency between the user's view and the physical database. If a failure occurs while a physical write operation is taking place, both the old and the new data values may

*UNIX is a trademark of the AT&T Bell Laboratories.

be lost. In some cases the determination of a previous consistent database state may be impossible.

A mechanism, using tagged differential files, which is implemented to maintain a consistent database state even in the event of a system failure, is described. The basic idea is to ensure the existence of a previous consistent database state at which the database was consistent. This eliminates the problem associated with the UNIX delayed-write and update-in-place policy.

In the following section a description of the UNIX file system components is given and the inconsistency problem is outlined. This is followed by an overview of the recovery techniques implemented in the NRDNIX DDBMS. The implementation of the differential file technique is described. The recovery processing algorithm is illustrated by means of an example. It is concluded that the proposed method provides the application with a consistent view of the database.

1 The UNIX Kernel

One of the main objectives of the designers of the UNIX operating system was to provide a small, simple yet effective, operating system. This gives users the freedom to design their own user interfaces and other utility programs as needed. The operating system consists of 10 000 lines C code and about 1 000 line of assembler code. The operating system is often called the kernel, emphasizing its isolation from user programs. The kernel consists of two main components: the process subsystem and the file subsystem.

1.1 Processes

Application programs are executed in an environment called a user process. A user process consists of a text, data and stack segment and executes in user mode. Various processes are concurrently executed in an interleaved fashion as determined by the kernel scheduling algorithm. These processes communicate via system calls. Processes are protected from interference from others by ensuring that one process may not read or write the data or stack of another. The read-only text segment may be shared by various processes.

When a system function is required the user process calls the system as a subroutine via one of the standard UNIX system calls. This causes a distinct switch of environment and the process enters kernel mode until the completion of the kernel subroutine. A separate stack is used for the user and kernel modes, thus isolating the user from the operating system routines.

1.2 The File System

The UNIX kernel implements a *hierarchical file system*. A file system consists of a sequence of logical blocks, each containing any convenient multiple of 512 bytes.[1] Directories are

[1] This paper will use the term 'block' to refer to a logical block consisting of 512 bytes.

used to implement the hierarchical structure. The nodes of the hierarchy are directories and the leaves are files, with the root directory as the parent node of the system. A directory is a file consisting of file names as well as information to access these files. Directories thus provide the mapping between the names of files and files themselves. Below the hierarchical file subsystem a *flat file subsystem* is implemented by means of a list of file definitions called an i-list. These file definitions, called inodes, contain the relevant file information to access and modify a file. Below this flat file system is the *I/O subsystem* which carries out physical I/O operations on devices. Below this is the *physical hardware* [1]. Figure 1 shows the relationship between the various subsystems.

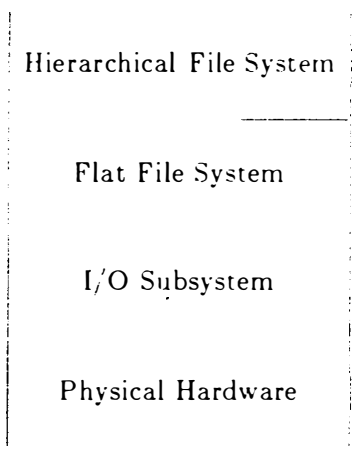

Figure 1: The various file subsystems.

A directory entry contains a file name and a pointer to the file, called the i-number of the file. When a file is accessed, its i-number is used as an index to the i-list stored in a known part of the device on which the directory resides. The inode found thereby contains a description of the file. A copy of the inode is kept on disk when the file is not active. When a file is accessed, the inode is read to main memory. The in-core inode is kept locked to ensure its consistency during update. The concept of inodes simplifies the design and maintenance of the file system.

1.3 The I/O Subsystems

The I/O subsystem allows processes to communicate with peripheral devices such as disks, tapes and printers. The kernel modules that control devices are known as device drivers. An application interfaces devices through the file system, accessing the device by means of a name similar to a file name.

1.4 The Disk Device Driver

The actual I/O to disk is controlled by the device driver. The driver supervises the transmission of data between primary storage and disk. A disk may be partitioned into several units, each containing one or more file systems. The kernel deals on a logical level

with file systems rather than with disks. The conversion between logical device addresses and physical device addresses is done by the disk driver. The disk driver translates a file system address, consisting of a logical device number and a block number, to a particular sector on disk. A disk may be accessed in either block or character mode. The standard *read* and *write* system calls use the block I/O system.

1.5 The Buffer Cache

The kernel attempts to minimize the frequency of disk access by keeping a pool of internal buffers, called the buffer cache, containing the recently accessed disk blocks. The cache acts as interface between the file system and the block device driver.

When executing *read* or *write* system calls, the kernel uses the file descriptor to access the file descriptor table, follows pointers to a global file table and inode table entry, and, from the inode, finds the data file. Figure 2 shows the execution path of a *read* or *write* system call.

When a *read* system call is executed, the kernel first attempts to read from the buffer cache. If the accessed data are located in the cache, the kernel does not have to read from disk. If not, the disk block on which the data are located is read to the cache. The data are copied from the cache to the user address space and the corresponding cache buffer is released. The kernel locks the corresponding inode for the duration of the call. This ensures that no other process may update the block of data during the *read* system call. The kernel unlocks the inode between various system calls. If another process changes the file between two *read* system calls, the first process may read unexpected data although the kernel data structure remains consistent. The system does not guarantee that the data in a file remains the same after the file has been opened.

Similarly, data written to disk are cached for enhanced retrieval time. If the kernel informs the disk driver that it has a buffer whose contents should be output, the disk driver schedules the block for I/O. When a delayed-write is executed, the buffer is marked as waiting to be written and is not immediately written to disk. If the file does not contain a block corresponding to the byte offset to be written when issuing the *write* system call, the kernel allocates a new block corresponding to the logical byte offset to be written. Blocks are iteratively written to disk. During each iteration, it is determined whether the whole block should be written to disk or only part of it. If only part of it is written, the block is first read from disk to avoid overwriting the part that will remain the same. Otherwise, the whole block is written to disk.

1.6 File System Inconsistencies

The design of the I/O system could result in an inconsistent file system, causing the loss of information.

- The asynchronous nature of the delayed-write algorithm makes error reporting and meaningful user error handling almost impossible, resulting in possible undetectable inconsistencies.

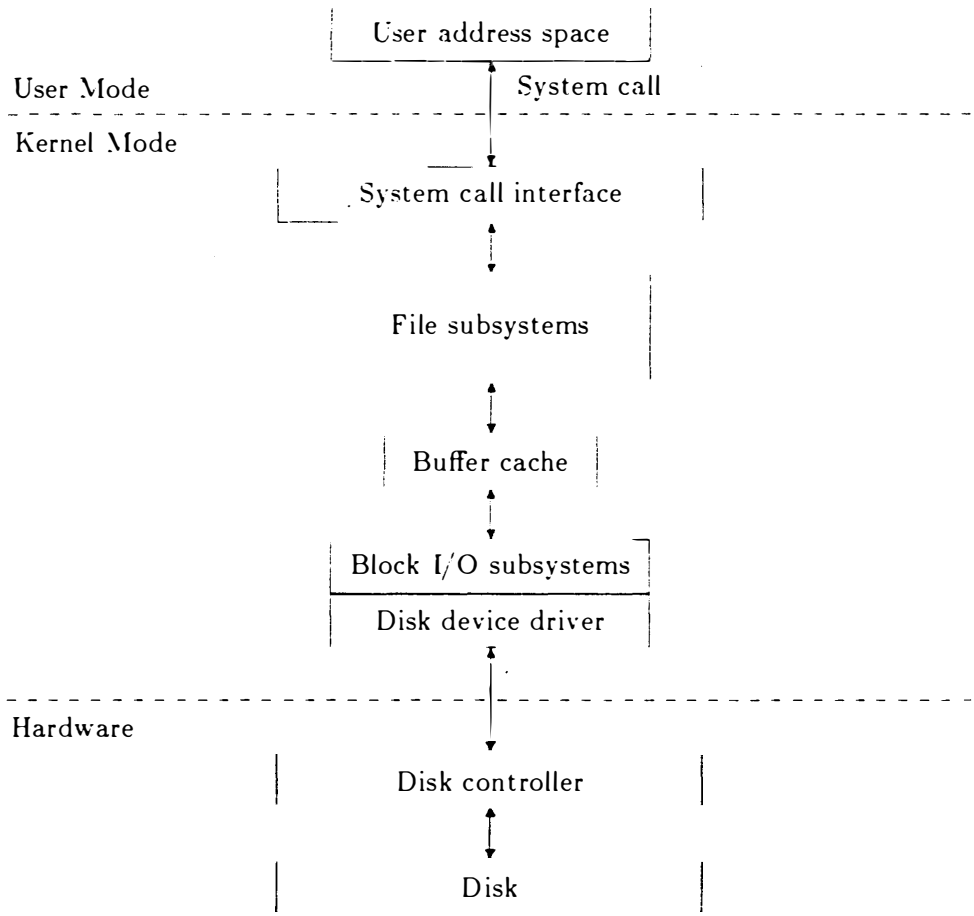

Figure 2: The execution path of a disk access.

- The physical write is delayed as long as possible, saving extra I/O operations. Since the kernel does not immediately write data to disk, the system is vulnerable to crashes that leave the disk data in an incorrect state. If a system failure occurs there could be logically complete, but physically incomplete I/O in the buffers. The standard I/O packages available to C language programs include a *fflush* call. This function call flushes data from buffers in the user address space into the kernel. Periodic use of this function helps, but does not solve the problem, since the application still does not know when the kernel writes the data to the disk.
- An update- in- place policy is used when writing a block of data to disk. If a system failure occurs during this process, the resulting file can be in any of the following states:
  - The write was logically complete, but the physical write had not been initiated. This results in loss of information.
  - The write was in progress when the failure occurred, resulting in an inconsistent block of data.

– The write was both logically and physically complete.

An application issuing a UNIX *write* system call is not notified when the data are physically written. The number of disk writes are reduced at the expense of possible loss of information in the event of a system failure. This necessitates a recovery mechanism to eliminate the loss of information.

Example 1

A simple example is used throughout this paper. A client at a departmental store is identified by a number and is further described by name, age and balance. A bonus of R50 is given to clients older than 65. with a balance greater than R100, resulting in the following transaction:

BEGIN TRANSACTION;

UPDATE client
SET balance = *balance* + 50
WHERE age > 65 *AND*
balance > 100;

COMMIT;

The record values of the corresponding clients prior to the update are depicted by figure 3. The records of Client 1 and Client 2 are located on page 1 of the database file and those of Client 3 and Client 4 on page 2. Each client's account has not yet been incremented by R50.

PAGE	CLIENT	AGE	BALANCE
Page 1	Client 1	67	950
	Client 2	68	2100
Page 2	Client 3	77	40000
	Client 4	89	840

Figure 3: Data values before the execution of the update transaction.

The resulting values of the successful execution of the update transaction is shown in figure 4 .

PAGE	CLIENT	AGE	BALANCE
Page 1	Client 1	67	1000
	Client 2	68	2150
Page 2	Client 3	77	40050
	Client 4	89	890

Figure 4: Data values after the execution of the update transaction.

In general, if a system failure occurs during the execution of the update transaction, the resulting database can be in any of the following states:

- The writes were logically complete, but the physical writes had not been initiated. The resulting database is in the state depicted by figure 3, with the accounts of all the clients containing the values prior to the transaction.
- The write was both logically and physically complete, as in figure 4. Here, no loss of information occurs. The accounts of the clients all contain the extra R50 as result of the transaction.
- The write was in progress when the failure occurred, resulting in an inconsistent block of data, with some current and some invalid old data, as in figure 5. The data values on page 1 are updated before those on page 2. Consider the situation when a failure occurs during the update, immediately after the situation shown in figure 5 and assume that no recovery algorithms are utilized. The write to page 2 was in progress when the failure occurred, resulting in an inconsistent block of data, with some current and some invalid old data. The amounts located on page 1 for Client 1 and Client 2 are up to date. The amount reflected in the account of Client 3 (R40 050) is current, but not in that of Client 4 (R840). As a result of the failure, an amount of R50 has been lost. The state of the system no longer reflects a consistent database. In addition, it is not possible to reconstruct a previous database state from the current state. If the transaction is re-executed, the accounts of the first three clients will contain R50 too much. Alternatively, if the effects of the transaction are removed from the database, the account of client 4 will incorrectly be decremented by R50.

PAGE	CLIENT	AGE	BALANCE
Page 1	Client 1	67	1000
	Client 2	68	2150
Page 2	Client 3	77	40050
	Client 4	89	840

Figure 5: Data values during the update prior to crash.

Since no additional data is recorded, it is not possible to determine in which one of the above three states the current database is. An incorrect re-execution or rollback of a transaction will lead to an inconsistent database state. It follows than a mechanism should be implemented to facilitate the recovery from this type of situation, bearing in mind the uncertainty incurred by the UNIX delayed-write policy.

Possible Solutions

A possible solution would be to bypass the UNIX file system. Although application programs can bypass the regular UNIX file access method and access the data directly, it is not advised. These programs are dangerous because they can destroy the consistency of the file system data. The file system algorithms coordinate disk I/O operations to maintain a consistent view of the disk data structure, including the list of free blocks and inode information. The list of free blocks maintained by the file system may become incorrect. The user program may write to blocks that are either considered free by the file system or used by the file system. This will normally not be detected by the file system, resulting

in inconsistent data values. This solution should therefore be avoided.

The UNIX system does provide a file system maintenance algorithm, which performs consistency checks to repair inode references and free block lists after a system failure. This algorithm aims to regain the consistency of the i-list and the list of free blocks, making use of salvation techniques. However, no provision is made for the recovery of data which was residing in main memory when the failure occurred. This data will be lost if the application does not explicitly utilize an algorithm which eliminates this problem.

A third solution is to implement a recovery mechanism to supplement the UNIX kernel. This mechanism utilizes the facilities offered by the system and ensures the consistency of files on disk. Such a mechanism in used in the NRDNIX DDBMS.

2 The NRDNIX DDBMS

The NRDNIX distributed database management system is implemented as a number of processes, called managers, on a version of UNIX System V. Interested readers are referred to Rennhackkamp [5] for a detailed description of the system.

Access to a local physical database is attained through the UNIX kernel. The database manager of the local DBMS interacts with the UNIX kernel via system calls. The kernel uses the block I/O interface when accessing the physical database files on behalf of the database manager. All low level operations on the database files are therefore controlled by the UNIX file system, thus abstracting the DBMS from the physical access.

Operations on the database are grouped into logical units called transactions. A transaction consists of a number of read and write operations on the database records. The database manager of the NRDNIX DDBMS controls user space memory management and physical database access. If access to a record is requested, the database manager issues the *read* system call. The read operation is performed, the block is read to cache and the transaction is executed. The operations performed on the database are logically written to the log prior to their actual execution. Upon a *write* system call, the updates to data values are first written to the buffer cache and then to the differential file.

2.1 The NRDNIX Recovery Mechanism

Recovery data is accumulated during normal processing. This data is used by the recovery manager when recovering from failures. Various recovery techniques are combined to recover to a consistent database in the event of a system failure.

2.1.1 Logical Log Facility

A *write ahead logical logging facility* is implemented. The decomposed relational algebra statements are recorded in a log file prior to the actual execution thereof. Additional information, including the transaction identifier and timestamp, is also logged. New log information is appended at the end of the log file. No physical representation of the data

items are logged, thus reducing storage overhead. The log information is written to disk using the UNIX *write* system call and is therefore subject to the problem incurred with delayed-writes. To minimize this, the buffer associated with the log is flushed at the end of every transaction. If a failure occurs when logically complete log information had not yet been written to the physical log, this can simply be ignored due to the no- undo policy. A *log garbage collection facility* [3] is implemented to remove redundant data from the log.

Transaction orientated checkpointing is used to reduce the amount of work when recovering from a failure. When a transaction commits, a checkpoint record is logged. It indicates the currently active transactions and transactions committed and aborted since the previous checkpoint. In addition, a flush of the database cache[2] is forced. Checkpoint generation is performed at the end of each transaction to provide a highly consistent database. In the case of an application where the transactions are short, the intercheckpoint interval is lengthened, otherwise the decreased performance incurred by the checkpointing may become unacceptable.

2.1.2 Differential Files

The abstraction of the log information from the physical data, achieved by making use of the logical logging facility, implies that no record of the previous state of a data item is kept. This increases the complexity of undoing uncommitted updates and could result in inconsistent data values. *Checkpoint orientated differential files* are implemented to overcome this problem.

Here, only the results of committed transactions are written to the database and the currently active records are maintained in a differential file. Since only the effects of committed transactions remain in the differential file, and only these values are eventually merged with the physical database, the undo of uncommitted values is eliminated.

At each checkpoint, a new differential file is opened. The currently active transactions are read from the current differential file and written to the new differential file. The blocks corresponding to these records are then deleted from the current database. The contents of the current differential file will therefore only include the results of committed transactions. The current file is closed and merged with the database at a later stage. The new differential file now becomes the current copy and updates are performed to this file.

A tagging mechanism is implemented to overcome the delayed- write problem. The last operation prior to the closure of the differential file consists of appending a sequence of bytes to the end of the file. This sequence (the *tag* of the differential file) consists of the unique current timestamp associated with the corresponding checkpoint as well as a predefined sequence of reserved characters. The tag uniquely identifies the differential file. A tag at the end of a file guarantee the completeness of the file.

The tagging mechanism ensures that all the required physical write operations have been performed when a differential file has been successfully closed. The differential file can then be merged with the physical database using the UNIX *read* and *write* system calls.

[2]This cache is part of the database manager and should not be confused with the UNIX buffer cache

The Bloom Filtering Algorithm

When access to a record is required, the location (i.e. in the differential or main database file) of the record is obtained by making use of a presearch Bloom filtering algorithm. When a data record is accessed, the filtering algorithm is used to determine whether the block associated with the record is located in the differential file or the main file. This reduces the amount of additional access time usually associated with the differential file method.

A bit vector of length M is kept in main memory. This vector is initiated to 0. In addition, X different hash functions is implemented. The identifier of a record r is used as a seed to each of the different hashing functions, thus yielding x different values. When a record is accessed in the main file and copied to the differential file, the values of the positions corresponding to various values of x is set to 1 in the bit vector. To determine whether a record is in the differential file, the identifier is hashed and the XOR of the corresponding bits are determined. If the value is equal to 1, the record is probably in the differential file. (It is not certain, since filtering errors may occur.) Otherwise, the record is fetched from the main file. The frequency of filtering errors can be controlled by manipulating the values selected for M and X by using a formula described by Severance and Lohman [6].

The Merging Algorithm

The merging of the differential file and the main file is performed only after the differential file has been successfully closed. The merging algorithm reads a block of data from the differential file by using the *read* system call. This block is then placed in the UNIX buffer cache. The block is written to the physical database using the delayed-write policy implemented by the *write* system call.

A failure may occur while the merge of the differential file is taking place. The resulting database after the failure may have been in one of the following states:

- The write was successfully executed. The database file contains the current up to date values which correspond to the values contained in the differential file.
- The write was not yet executed. The physical database contains the old values. The logical complete updates are not included in the physical database, resulting in an inconsistency between the user view and the actual state.
- The write operation was in the process of execution when the failure occurred. The contents of the physical database is therefore inconsistent, since it contains both old and new values.

A merging algorithm is implemented which provides a mechanism to indicate the current differential file block that is being merged. The merging algorithm proceeds as follows:

- Before the block of data are read from the differential file, a flag which indicates the logical block number in the differential file, is written to a predetermined location on disk.
- The differential file block is read to the buffer cache and written to the corresponding database file block.

- The flag is changed to a predefined reserved block sequence indicating that the merge was successful. The flag will always correspond to this sequence if no merge operation is taking place.

The flag is written to a predetermined location on disk. In the event of a system failure, the recovery manager determines which logical database file block was being updated. This ensures that the state of the database prior to the failure can be determined if the failure occurred during the execution of a merge.

2.2 Recovery Processing

Upon recovery from a site failure, a redo, no-undo algorithm is executed. All transactions that have been committed, but possibly not yet propagated, are redone.The following recovery steps are taken:

- If a merge operation was taking place when the failure occurred, an unmerge algorithm is performed.
  - If the failure occurred while writing the flag to disk, the predetermined location will either contain the reserved sequence or the logical block number. If it contain the reserved sequence, the recovery manager will re-initiate the merging algorithm. If the flag was successfully written to disk prior to the failure, only the merging of the block is redone.
  - If a differential file block was in the process of being merged with the database file, the existence of the flag will indicate this. The block corresponding to the flag address is freed and the merging is redone. Since the data values will be safely recorded in the differential file, no information is lost.
  - If a failure occurred while unflagging, the unmerging algorithm consults the inode information and concludes that the merge was successful.
- A previous consistent state, consisting of an old copy of the physical database, is constructed. The corresponding differential file is obtained. All closed but not yet merged files are subsequently merged into the main file.
- The checkpoint record corresponding to the latest differential file is consulted to obtain a list of the transactions to be redone.These transactions are re- executed.
- Global system recovery is performed. One of the advantages of a distributed database versus its centralized counterpart is that sites continue processing in the event of failures of other sites. This implies that the updates on local copies of the data must be obtained and executed by the site. Upon recovery, the site obtains the missed updates from other sites and performs the necessary updates to the database [7].
- After the successful completion of the recovery algorithm normal processing is resumed.

Example 2

The use of the NRDNIX recovery algorithm is illustrated by the following example.

Consider again the update transaction of example 1:

BEGIN TRANSACTION;

UPDATE client
SET balance = balance + 50
WHERE age > 65 AND
balance > 100;

COMMIT;

The record values of the corresponding clients prior to the update are depicted by figure 6 and the result by figure 7. The records of Client 1 and Client 2 are located on page 1 of the database file and those of Client 3 and Client 4 on page 2. The resulting values are located on pages 1d and 2d in a differential file.

PAGE	CLIENT	AGE	BALANCE
Page 1	Client 1	67	950
	Client 2	68	2100
Page 2	Client 3	77	40000
	Client 4	89	840

Figure 6: Data values before the execution of the update transaction.

PAGE	CLIENT	AGE	BALANCE
Page 1d	Client 1	67	1000
	Client 2	68	2150
Page 2d	Client 3	77	40050
	Client 4	89	890

Figure 7: Data values after the successful completion of the update transaction.

If a system failure occurs during the update, the database may be in any of the following states:

- The writes were all logically complete, but the physical writes have not been initiated. The accounts of all the clients therefore still contain the old values as in figure 6. Upon recovery, the timestamp of the last successfully closed differential file is consulted. The result of the update will not be found in the differential file since it had not yet been written to disk. It will be therefore correctly be concluded that the results of the transaction had not yet been propagated. The transaction will be redone by using the information recorded in the local log, resulting in the correct values written to all the accounts.
- The physical write was in progress when the failure occurred, as in figure 8. The accounts of the first two clients contain the new values R950 and R2100 and those located on page 2d still contain the old values. These pages are located in the differential file, since only the effects of closed differential files are merged with the main database file. The physical database was not influenced. Upon recovery, the contents of the incomplete differential file are discarded and the update is redone by

using the log information.

PAGE	CLIENT	AGE	BALANCE
Page 1d	Client 1	67	1000
	Client 2	68	2150
Page 2d	Client 3	77	40000
	Client 4	89	840

Figure 8: Data values during the execution of the update transaction.

- The write was both logically and physically complete, but the merge had not yet been executed. Again, the updated records were written to the differential file and the main database was not effected. The differential file thus contains the correct amounts for all the relevant clients shown in figure 7. If the differential file was already closed by using the previously described method, it is merged with the main database file, the result being a consistent and current database. If not, the database is rolled back to the state prior to the beginning of the differential file and all the relevant transactions, including the update, are redone by using the log.

PAGE	CLIENT	AGE	BALANCE
Page 1	Client 1	67	1000
	Client 2	68	2150
Page 2	Client 3	77	40050
	Client 4	89	840

Figure 9: Data values during the execution of the merging algorithm in the main database file.

- If the differential file and the main database file were being merged as in figure 9 when the failure occurred, the effects of the update are recorded in the local log. The unmerge algorithm is executed. The database is rolled back to the state prior to the beginning of the differential file and all the relevant transactions, including the updates, are redone by using the log.
- If the merge was successfully completed as in figure 10, the data values in the accounts of all four clients are up to date. The transaction has not to be redone and its effect are valid and secure.

PAGE	CLIENT	AGE	BALANCE
Page 1	Client 1	67	1000
	Client 2	68	2150
Page 2	Client 3	77	40050
	Client 4	89	890

Figure 10: The resulting data values in the main database file after the merge process.

The example illustrates that the recovery algorithm is sufficient to recover from any of the above situations. The amounts in the clients' accounts are correct and no information is lost when using this scheme.

Conclusion

The loss of information due to the delayed-write policy of the UNIX buffer cache is eliminated by the use of logical logging and differential file recovery techniques. Committed, unpropagated transactions are redone by executing the described recovery algorithm. The merging algorithm is used to construct a previous consistent database state by combining the main database file and all the closed, but not yet merged, differential files. The use of a tagging mechanism ensures that only the effects of committed transactions are written to the physical database. The tag also indicates the last complete differential file in the system. The timestamp of this differential file is then used to obtain the corresponding checkpoint record, thereby ensuring that the necessary transactions are redone. In the event of a failure during the merge process, the unmerge algorithm is used to construct a previous consistent state. This ensures the reconstruction of a previous consistent database state.

Differential files can be used when implementing an incremental dumping recovery facility to facilitate the recovery from disk failures. Closed differential files are written to auxilary storage together with a previous consistent database file. In the event of a disk failure, a previous consistent database is constructed by executing the merging algorithm on the files located on the auxilary storage. This increases the reliability of the DDBMS.

The above mechanisms overcome the shortcomings of UNIX and are implemented as user processes. They interface with the UNIX kernel through standard system calls. It was not necessary to modify the UNIX kernel. The DDBMS, developed in C, is therefore highly portable and system independent. One of the main objectives of the NRDNIX DDBMS, namely to provide a consistent, but highly portable system, is therefore met.

References

1. Anyanwu, J.A., *A Reliable Stable Storage System for UNIX*. Software - Practice and Experience, Vol 15(10), October 1985.

2. Bach, M.J., *The Design of the Unix Operating System*. Prentice-Hall, 1986.

3. Deacon, A., *Global Consistency using Logical Logs*. Technical Report, Department of Computer Science, University of Stellenbosch, 1988.

4. Ritchie, D.M., *The UNIX I/O System*. UNIX Programmer's Manual, Seventh Edition, January 1979.

5. Rennhackkamp, M.H., *The NRDNIX Distributed Database Management System*. Technical Report, Department of Computer Science, University of Stellenbosch, 1989.

6. Severance, D.G. and Lohman, G.M., *Differential Files: Their Application to the Maintenance of Large Databases*. ACM TODS, Vol 1, No 3, September 1976, p256-267.

7. Viktor, H.L., *Distributed Database Recovery*. Proceeding of the RCP Group Conference for M.Sc. and Ph.D. Students, 1988.