

2004

Files as first-class objects in fault -tolerant concurrent systems

Robert Edwin Matthews

College of William & Mary - Arts & Sciences

Follow this and additional works at: <https://scholarworks.wm.edu/etd>



Part of the [Computer Sciences Commons](#)

Recommended Citation

Matthews, Robert Edwin, "Files as first-class objects in fault -tolerant concurrent systems" (2004).
Dissertations, Theses, and Masters Projects. Paper 1539623456.
<https://dx.doi.org/doi:10.21220/s2-qa9t-5z25>

This Dissertation is brought to you for free and open access by the Theses, Dissertations, & Master Projects at W&M ScholarWorks. It has been accepted for inclusion in Dissertations, Theses, and Masters Projects by an authorized administrator of W&M ScholarWorks. For more information, please contact scholarworks@wm.edu.

FILES AS FIRST-CLASS OBJECTS IN FAULT-TOLERANT
CONCURRENT SYSTEMS

A Dissertation

Presented to

The Faculty of the Department of Computer Science

The College of William & Mary in Virginia

In Partial Fulfillment

Of the Requirements for the Degree of

Doctor of Philosophy

by

Robert Edwin Matthews

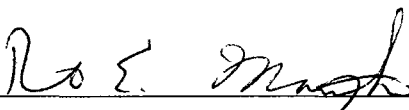
2004

APPROVAL SHEET

This dissertation is submitted in partial fulfillment of


the requirements for the degree of

Doctor of Philosophy

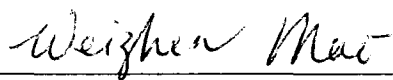


Robert E. Matthews

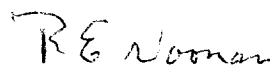
Approved, August 2004



Phil Kearns
Dissertation Advisor



Weizhen Mao



Robert Noonan



Xiaodong Zhang



Jean Mayo
Michigan Technological University

To My Parents
William Lee Matthews
and
Patricia Ann Matthews
and
to the Many Gifted Teachers and Mentors Who Have Influenced My Life

"... to know true ends from false, and lofty things from low ..."

Table of Contents

Acknowledgments	x
List of Tables	xi
List of Figures	xiv
Abstract	xv
1 Introduction	2
1.1 Concurrent Systems	2
1.2 Background	4
1.3 Organization of Paper	5
2 Fault Tolerance	7
2.1 The Beginnings of Fault-Tolerant Computing - Self Stabilization	7
2.2 Checkpointing and Rollback-Recovery Schemes	9
2.2.1 Terminology of Fault-Tolerance with Checkpointing and Rollback-Recovery	10
2.2.2 Consistent Global States	11

2.2.3	Checkpointing and Rollback-Recovery	13
2.2.4	Pure Checkpointing	14
2.2.5	Uncoordinated Checkpointing	16
2.2.5.1	Pessimistic Recovery	17
2.2.5.2	Optimistic Recovery	18
2.2.5.3	Causal Recovery	23
2.3	Checkpointing and Rollback-Recovery Schemes and Files	25
2.3.1	Plank and Litzkow	26
2.3.2	The SCR Algorithm	26
2.3.3	AIPC	28
2.3.3.1	The Operation of AIPC	28
2.3.3.2	Problems with AIPC	29
2.4	Using Files in Concurrent Computations	30
2.4.1	Biotechnology	31
2.4.2	Astrophysics	32
2.4.3	Applied Mathematics	33
2.5	Concluding Remarks	35
3	Log-Structured File Systems	36
3.1	The Unix File System	37
3.1.1	Terminology	37
3.1.2	A Unix File System	39
3.1.2.1	The Superblock	40

3.1.2.2	Inodes	41
3.2	Log-Structured File Systems	43
3.2.1	The Organization of a Log-Structured File System	46
3.2.2	Crash Recovery	48
3.2.3	Cleaning	49
3.2.4	Previous Implementations of Log-Structured File Systems	50
3.2.5	Log-Structured File Systems are History Preserving	50
4	Checkpointing and Rollback-Recovery with Files	52
4.1	Our Scheme	52
4.2	Vector Time	54
4.3	File System Checkpointing and Process Checkpointing	57
4.4	Restoring System Consistency after Process Failure	58
4.4.1	Vector Time and Checkpointing	59
4.4.2	Recovery	61
5	Implementation	65
5.1	Design Goals and System Operation	66
5.2	API	68
5.2.1	Mkvlfs	69
5.2.2	Mount	70
5.2.3	Unmount	70
5.2.4	Open	70
5.2.5	Close	71

5.2.6	Read	71
5.2.7	Write	72
5.2.8	Unlink	73
5.2.9	Sync	73
5.2.10	Rollback	74
5.2.11	Print_stats	74
5.3	The File System Layer	76
5.3.1	The Virtual Disk	76
5.3.2	The Disk Cache	77
5.3.3	Vector Time	78
5.3.4	The Inode	79
5.3.5	The Imap	81
5.3.6	The Checkpoint Region	81
5.3.7	The Superblock	81
5.4	The Syscall Sequence	82
5.5	File System Rollback	83
5.6	The Process Control Layer	85
5.6.1	Thread Checkpointing	85
5.7	System Recovery After a Failure	87
5.7.1	Thread Restart	91
5.7.2	Thread Rollback and Rollforward	94
5.8	Application Rules for Using the System	101
5.8.1	Restart Check	101

5.8.2	Thread Registration	102
5.8.3	Checkpointing	102
5.8.4	Thread Deregistration	102
5.8.5	An Application Program Skeleton	102
5.9	POSIX Thread Synchronization	104
5.9.1	Mutexes	104
5.9.2	Condition Variables	106
5.9.3	Thread Rollback with POSIX Synchronization Variables	111
6	Formalisms	118
6.1	Concepts and Definitions	118
6.2	Recovery After Failure	120
6.3	Reclaiming Checkpoints and Logs - The Domino Effect	123
6.4	Messages	126
6.4.1	Recovery Protocol	126
6.4.2	Rollforward	126
6.5	Unusual Failure Modes	130
6.5.1	Concurrent Failures	131
6.5.2	Failures During Recovery	134
6.5.2.1	Failures Without the Assumption of Independence	135
6.5.2.2	Failures During Rollforward	138
7	Performance Evaluation and Experimental Results	140
7.1	File System Theoretical Performance	141

7.1.1	Sync() Evaluation	141
7.1.1.1	Fixed Cost Component of Sync()	141
7.1.1.2	Variable Costs Component of Sync()	143
7.1.2	Rollback() Evaluation	145
7.2	File System Empirical Evaluation	146
7.2.1	The Columnsort Algorithm	147
7.2.2	Effects of Buffer Cache Size on Performance	150
7.2.3	Failure-Free Overhead in Columnsort	150
7.2.4	A Comparison of the Prototype with AIPC	154
7.3	Log Clearing	156
7.3.1	Discardable Checkpoints	157
7.3.2	Pessimistic and Optimistic Log-Clearing	158
7.3.3	Optimal Log-Clearing	161
7.4	Dedicated File Systems for Fault-Tolerance	162
8	Conclusion	166
8.1	Future Directions	167
A	Sync() Syscall Implementation	169
B	The CHECKPOINT() Macro	175
	Bibliography	178

ACKNOWLEDGMENTS

I would like to acknowledge my adviser, Dr. Phil Kearns, for his tremendous contributions to this work, and for his saintly patience, and my dissertation committee for their time and valuable suggestions.

List of Tables

4.1	Vector Time Propagation Rules	60
5.1	Block and Byte Ranges Accessible Via Inode and Indirect Blocks With A Default Block Size of 1K.	80
5.2	The <code>Ucontext_t</code> Structure for the X86 Architecture.	86
7.1	Fixed Costs of a <code>Sync()</code> Operation	143
7.2	Total Worst-Case <code>Sync()</code> Overhead	145
7.3	Cost of <code>Rollback()</code> System Call	147
7.4	Average Disk Writes Caused by a Single <code>Sync()</code> in <code>ColumnSort</code>	152
7.5	Performance of the Seagate Cheetah TM 10K.6 37GB Disk Drive	152
7.6	Per-thread Block Reads, Block Writes, File Copies and Total Disk Block Accesses Performed by AIPC During <code>ColumnSort</code>	156

List of Figures

2.1	Consistent and Inconsistent Cuts in a Distributed System	12
2.2	Pure Checkpointing	14
2.3	The Domino Effect	17
2.4	State Intervals of a Recovery Unit	19
2.5	Optimistic Recovery	21
2.6	Causal Recovery	24
3.1	A Unix File System	40
3.2	An Inode	42
3.3	An Indirect Block Scheme for Increasing the Number of Addressable File Blocks	44
3.4	A Log-Structured File System	47
3.5	A Log Structured File System After the Log has been Written to Disk . . .	48
3.6	The On-Disk Layout of a Log-Structured File System	49
3.7	The On-Disk Layout of a Log Structured File System After the Log has been Written to Disk	49

4.1	Vector Time in a Three Process System	55
4.2	The Partial Order Induced on Events by Vector Time	56
4.3	Sharing a File in a Concurrent System	58
4.4	Vector Time Propagation Between Processes and the File System	61
4.5	The Propagation of Vector Time in a Three Process System with a Shared a File System	62
4.6	Rollback of a Failed Process	63
4.7	Rollback of Non-Failed Processes	63
4.8	Roll Forward of Non-Failed Processes	64
5.1	The Layers of the Prototype	66
5.2	A Logical Disk Built on Top of Physical Files	77
5.3	The In-Memory Representation of an Example Vector Time Object	78
5.4	The On-Disk Representation of an Inode	79
5.5	The On-Disk Representation of the Imap	81
5.6	The On-Disk Representation of the Checkpoint Region and its Associated Structures	82
5.7	The On-Disk Representation of the Superblock.	83
6.1	Finding Discardable Checkpoints	124
6.2	Message Replay During Recovery	128
6.3	Rollback After Concurrent Failures	134
6.4	A Possible Failure Mode if We Drop the Assumption of Independence.	135
6.5	A Second Possible Failure Mode if We Drop the Assumption of Independence	137

6.6	A Failure During Rollforward	139
7.1	Experimental Results of Sync()ing a Virgin File System on Disks of Various Sizes	144
7.2	Preparing Data for Columnsort	148
7.3	Columnsort Phase 1	148
7.4	Columnsort phase 5	149
7.5	Effect of Buffer Cache Size on the Number of Disk Writes Performed by Columnsort, With and Without Checkpointing	151
7.6	Discardability of Checkpoint $C_{k,q}$	158
7.7	Pessimistic Log-Clearing Average Log Size	160
7.8	Optimistic Log-Clearing Average Log Size	161
7.9	Optimal Log Clearing	162
7.10	A Comparison of Pessimistic, Optimistic and Optimal Log-Clearing Strate- gies	163
7.11	A File System with No Freelist	164

ABSTRACT

Concurrent systems are used in applications where multiple processors are needed to complete tasks within a reasonable amount of time, or where the data sets involved will not fit within the main memory of a single computer. Because of their reliance on multiple machines, such systems are proportionally more vulnerable to both hardware and software induced failures. Fault-tolerance schemes are used to recover some earlier consistent state of the system after such a failure.

One important technique used to achieve fault-tolerance is checkpointing and rollback-recovery. In this thesis, we present a method for efficiently and transparently incorporating the part of the process state contained in the file system into process checkpoints, and we show how recovery of consistent versions of the file system and processes may be done after a failure. We present the details of a prototype system which implements our method.

We show that by using the special properties of the log-structured file system, the class of programs which are amenable to checkpointing and rollback-recovery schemes can be expanded to include those that use files. We impose no a priori restriction on the types of file system operations that can be done, and we demonstrate that our scheme does not impose significant failure-free overhead on the computation.

FILES AS FIRST-CLASS OBJECTS IN FAULT-TOLERANT
CONCURRENT SYSTEMS

Chapter 1

Introduction

*Thus in the beginning the world was so made
that certain signs come before certain events.*

Cicero, *De Divinatione*. i. 118

1.1 Concurrent Systems

Concurrent systems are those in which separate processes cooperate to achieve some goal. It is useful to categorize concurrent systems by the amount of hardware resources that the processes share. In tightly coupled systems, the processes share all or most hardware resources. In loosely coupled systems, they share few or no local resources.

A multiprocessor system is an example of a tightly coupled system. Processes share all resources except the CPU's and their associated caches.

At the other end of the spectrum, we find the loosely coupled models, such as distributed systems. In a distributed system, processes share few if any local resources. Each system has

its own memory, I/O channels, and system clock. Coordination between the components of the system is done via message passing.

Concurrent systems provide several advantages over the traditional model of computation. Mayo [37] gives three primary advantages:

- They provide greater computational power than that provided by traditional systems.

An example of this type of system is the Berkeley Network of Workstations (NOW) architecture [5]. A NOW implementation consists of individual computer systems connected by a fast network. The systems communicate via message passing. A unique characteristic of the NOW architecture is that the file system is serverless. Workstations cooperate as peers to provide file system services.

- Loosely coupled implementations of these systems may be geographically dispersed.

Examples of this type of system are point-of-sale terminals connected to a central credit card database. In these types of systems, we may even find that the database itself is dispersed geographically.

- They allow for redundancy. By replicating data or computational processes on different systems, we can introduce a primitive form of fault-tolerance. The loss of one node of the system does not result in the loss of data or computational state.

While concurrent systems have advantages over traditional monolithic systems, their use introduces a number of new problems. The inherent parallelism of the model makes them difficult to program using traditional languages, since the underlying machine model is so different. The lack of shared resources makes the implementation of such basic constructs as “atomic operations” and “sequential execution” difficult. Finally, as we introduce

more subsystems into a more complex computational model, these systems become more vulnerable to both transient software failures (such as those caused by network congestion and programming errors) and hardware failures (such as the failure of mechanical parts and power outages.)

The general term which describes dealing with hardware and software failures in concurrent systems is “fault-tolerance”. A fault-tolerant system is one which detects and recovers from these failures. This recovery process may or may not be successful in a given circumstance, and the recovery itself may not be invisible when the system is viewed from the outside.

1.2 Background

The goal of a fault-tolerant system is to detect so-called “illegitimate states” [13] of the system and transform such a state to a “legitimate state” using some mechanism. An illegitimate state is considered to be one where an unexpected condition has occurred.

Since the late 1980’s, most research in this area has focused on a single method, called “rollback/recovery”. In a nutshell, this method attempts to reconstruct a legitimate state from an illegitimate state by periodically taking snapshots of some subset of the state of the computation. Once an illegitimate state has been detected, the system discards the current state, and builds a new, legitimate state from one or more previously taken snapshots. The computation can then proceed from this newly constructed legitimate state. The term “rollback” comes from the fact that the computation is said to roll backward from the illegitimate state to a legitimate state.

There are many questions in rollback/recovery that are rich research areas. For example:

- What is the most efficient way to take snapshots?
- How can we guarantee that a set of snapshots that form a legitimate state exist?
- How and where should such snapshots be stored?
- What subset of the global system state should a single snapshot capture?
- What optimizations can be done to make the snapshot process more efficient?
- What is the most efficient way to re-instantiate a legitimate state from a set of snapshots?

Our research has focused on the fourth and fifth questions listed above. In particular, we have discovered a method for including in the snapshot the part of the system state resident in the file system. Our system is more efficient and robust when compared to previous attempts. We use the special properties of the log-structured file system [48] to achieve this efficiency.

1.3 Organization of Paper

In chapter 2 we discuss the history of fault-tolerant computing and the current state of research in the area. We conclude the chapter with a discussion of previous attempts to incorporate files into fault-tolerance schemes.

In chapter 3 we discuss log-structured file systems. We begin by looking at the history of file systems from the introduction of the Unix Fast File System [38] to the development

of the first log-structured file system. Our work depends on the special characteristics of the log-structured file systems.

Chapter 4 introduces our work on fully incorporating files into fault-tolerant concurrent systems.

In chapter 5 we discuss the prototype system that we have developed. The system allows arbitrary file system operations to be performed by fault-tolerant concurrent computations. The chapter includes sections on the implementation of our prototype, and a discussion of both the file system layer and the process control layer. We conclude by discussing how a concurrent systems programmer would develop code for our system.

In chapter 6 we prove a series of results that show our prototype correctly implements a variant of optimistic logging, and that the scheme does not suffer from the domino effect. We also discuss a minor result we have obtained on log-clearing algorithms. The chapter concludes with a discussion of some unusual failure modes that can occur, and how the system we have developed might deal with them.

Chapter 7 discusses the performance of our prototype both in theoretical and experimental terms. It also includes an empirical evaluation of our log-clearing scheme.

We conclude in chapter 8 by summarizing our results and discussing some areas of future directions.

Chapter 2

Fault Tolerance

*The greatest of faults, I should say,
is to be conscious of none.*

Thomas Carlyle, *The Hero as Prophet*

We begin this chapter by discussing Edsger Dijkstra’s extraordinary paper on what he called “self-stabilizing” systems. We then describe the terms and definitions used to discuss modern fault-tolerance schemes, and look at the three major types of fault-tolerance schemes which have been developed. We conclude the chapter by looking at attempts to incorporate files into these methods.

2.1 The Beginnings of Fault-Tolerant Computing - Self Stabilization

The earliest mention of what we call fault-tolerant computing was made in 1974 by Dijkstra [13]. Dijkstra proposed a simple model of distributed computing consisting of a

connected graph, where each node contains a finite-state machine and each edge represents a two-way communication channel over which the finite-state machines may transmit their current state to neighbors.¹

Each node in the graph has a list of “privileged” and “unprivileged states”. These states are combinations of each node’s own internal state and the internal states of its neighbors. At each step in the operation of this model, a single node is chosen and, if a privileged state exists, the node changes state. (Dijkstra assumes the existence of a central daemon which chooses the node to act. This central daemon might be as simple as a signaling clock. Machines without a central daemon, which are the focus of research today, are discussed below.)

We can think of privileges as “work to do”. For example, if the nodes of the graph are the components of a bus architecture, some components may have pending work. This work is indicated by a signal from another component.

A global criterion exists in such a system which indicates whether the overall state of the system is legitimate or illegitimate. Dijkstra defined such a system as “self-stabilizing” if, and only if, after a finite number of moves, and regardless of the privilege chosen at each move, the system eventually found itself in a legitimate state.

Dijkstra provided a proof that such self-stabilizing machines exist in the restricted case where the nodes are connected in a ring. The case for tree-structured systems was solved in 1979 by Kruijer [27]. For the proof that such machines exist in the general case of an arbitrary connected graph, Dijkstra noted “the appreciation is left as an exercise to the reader.” Ironically, it was not until fourteen years later, in 1986, that the general case was

¹Presciently, Dijkstra speculated that such a system might be useful for modeling “a worldwide network”.

solved - by Dijkstra [14].

In his 1983 keynote address to the 3rd ACM Symposium on Principles of Distributed Computing, Lamport, speaking of Dijkstra's original paper, says:

I regard this as Dijkstra's most brilliant work - at least, his most brilliant published paper. It's almost completely unknown. I regard it to be a milestone in work on fault tolerance. The terms "fault tolerance" and "reliability" never appear in this paper [29].

2.2 Checkpointing and Rollback-Recovery Schemes

The requirement of a central daemon that Dijkstra assumes strikes one immediately as an overly restrictive assumption. A central daemon implies some sort of centralized control, and thus excludes from the discussion truly distributed systems. Most of the work in fault-tolerance since the late 1980's has concentrated on systems without a central daemon [53].

Early work in fault-tolerant systems concentrated on certain ad-hoc methods of recovering a system after a transient failure. In the early 1990's a more well-defined approach became the accepted method of studying such schemes. The most comprehensive survey of rollback-recovery protocols in systems which use message passing and have no special hardware support for fault-tolerance is found in Elnozahy et al [16] [17].

Below, we discuss the modern methods which rely on checkpointing and rollback-recovery. These modern schemes can be divided in to four areas, known as "pure checkpointing", "pessimistic recovery", "optimistic recovery" and "causal recovery".

2.2.1 Terminology of Fault-Tolerance with Checkpointing and Rollback-Recovery

Fault-tolerance schemes which employ checkpointing and rollback-recovery use the following vocabulary and assumptions about the logical machines. These definitions and assumptions are adapted from Strom and Yemini [56].

The Logical Machine A cluster of processes running under a fault-tolerant scheme is referred to as the logical machine. These processes may each run on a single machine, or they may be distributed over a series of physical machines.

Recovery Unit The logical machine is partitioned in to a fixed number of recovery units (RUs). RUs communicate with one another through message passing. An RU may consist of a single process, a collection of Posix threads, or multiple processes. Whenever a thread or process is created, it is assigned to a particular RU. The literature on fault-tolerance uses the terms recovery unit and process interchangeably when there is no possibility of confusion. We adopt that practice here.

Failure-Free Overhead Any resources consumed by an RU solely to support the underlying fault-tolerance scheme are referred to as the failure-free or fault-free overhead. This is the amount of computational effort that must be expended even if no failures occur. We use failure-free overhead to evaluate the performance of fault-tolerance schemes.

Commit When a permanent, undo-able state change occurs, we refer to that as a commit. An example of a commit is a physical change to data on a disk.

We make the following assumptions about the logical machine.

Reliable, FIFO Communication Channels: RUs communicate over reliable channels.

Messages are always sent in order. We assume nothing about the arrival order of messages sent from two different sources.

Fail-stop: Failures are detected immediately and result in the halting of failed RUs and the initiation of the recovery protocol under discussion. That is, we exclude Byzantine failures from our discussion. The fail-stop model was first described in 1983 by Schlichting and Schneider [52].

Independence: Failures will not reoccur if a failed RU is re-executed on another machine.

Stable Storage: The current state of each recovery unit is stored in volatile storage. The information needed to recover an RU after failure is kept in stable storage [30].

Spare Processing Capacity: It is always possible to relocate a failed RU to some working processor which has access to the logical machine's stable storage.

No Shared Memory or Global Clock: Individual RUs communicate only via channels, and do not share any local resources.

2.2.2 Consistent Global States

The idea of “consistent global state” is central to reasoning about distributed systems. The idea was formalized by Chandy and Lamport [11].

In a distributed system, an *event* is a state change by a process, or the sending or receiving of a message. We say that *event a directly happens before event b* if and only if

1. a and b are events in the same recovery unit, and a occurs before b , or
2. a is the sending of a message m by RU R_i and b is the receipt of m by RU R_j .

The transitive closure of the directly happens before relation is the *happens before* relation. We use the notation $a \rightarrow b$ to denote “ a happens before b .” For two events a and b , if $a \not\rightarrow b \wedge b \not\rightarrow a$, then we say that events a and b are concurrent.

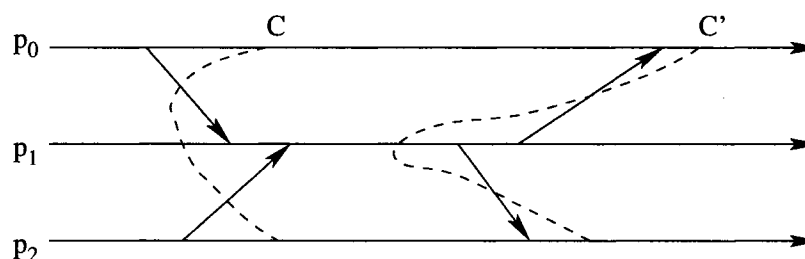


Figure 2.1: Consistent and Inconsistent Cuts in a Distributed System

We depict events in a distributed system using a time diagram. In a time diagram, each RU is represented by a horizontal line, with time moving from left to right. Events are represented as points on an RU’s time line. Messages between RUs are represented by diagonal lines from one RU to another. See figure 2.1.

A *consistent global state* of a distributed system is a snapshot of the system in which every event before the system snapshot happened before every event after the snapshot. We typically refer to these global snapshots as cuts. Figure 2.1 shows two cuts in a distributed system. Cut C represents a consistent global state, while cut C' is inconsistent since it represents a state where the receipt of messages occurred before their corresponding sends.

Intuitively, a distributed system is in a consistent state if every pair of processes in the system agree on which communications between them have taken place, and which have

not.

2.2.3 Checkpointing and Rollback-Recovery

The goal of checkpointing and rollback-recovery schemes is to bring the system to a consistent global state after the failure of an RU. This goal is achieved through a two- or three-step process, depending on the particular scheme used.

The first step in the process is checkpointing. During a checkpoint, an RU makes a copy of its internal state, and saves this state to stable storage. This internal state consists of copies of the processes memory image (including code, data, stack and dynamically allocated memory), the CPU registers (general purpose registers plus special purpose ones, such as the stack register and instruction pointer), along with variables external to the RU but never-the-less part of the RU's state (such as the list of signals the process has arranged to catch or ignore, open communication channels, etc.)

The second step of the scheme occurs only when an RU fails. Should an RU fail, it is reinstated from the saved state. This reinstatement is called rollback because the RU appears to be rolling back to an earlier point on the time diagram. Depending on the scheme used and the state of the other RUs in the system, it may also be necessary to rollback the non-failed RUs.

A third step may be required depending on the scheme being employed. This third step is called rollforward and is required because some schemes do not attempt to guarantee that RU checkpoints are mutually consistent. We discuss the three most common schemes below.

2.2.4 Pure Checkpointing

The simplest checkpointing and rollback-recovery scheme is called *pure checkpointing*.² Under pure checkpointing, the RUs cooperate to make sure that whenever a set of checkpoints is taken, the checkpoint set forms a consistent global snapshot. This guarantee means that whenever a rollback occurs, the resulting system state is automatically globally consistent. In the pure checkpointing scheme shown in figure 2.2, P_0 fails, and each RU is rolled back to its most recent checkpoint.

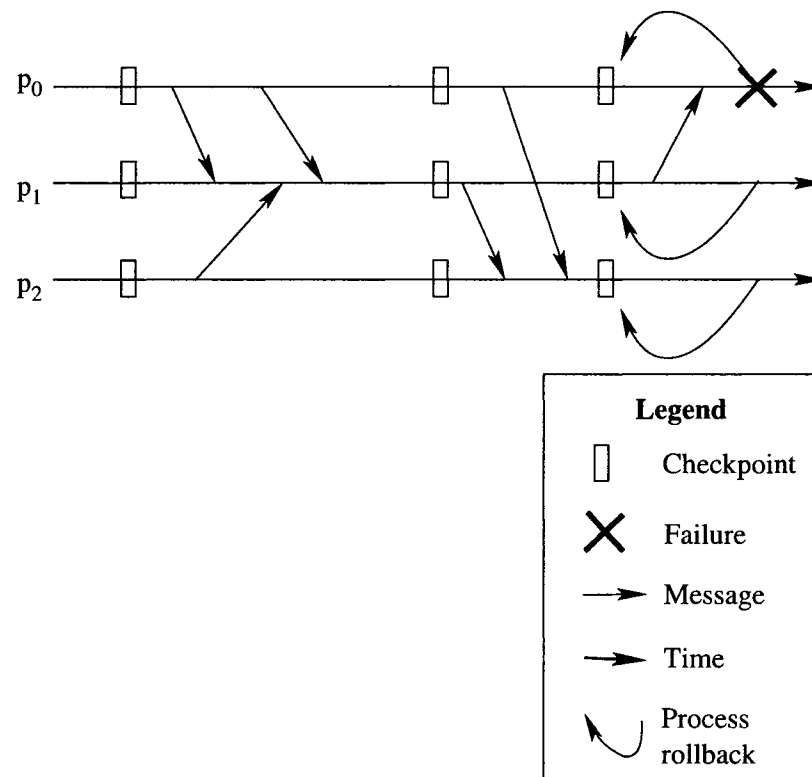


Figure 2.2: Pure Checkpointing

Pure checkpointing has several advantages over other types of schemes.

²The literature sometimes refers to what we call pure checkpointing as “consistent checkpointing” or “synchronous checkpointing.”

1. Since the system guarantees that snapshot sets form a globally consistent state, the rolled back system is also guaranteed to be globally consistent.
2. At most one set of snapshots needs to be kept on stable storage. When a new set is taken, the previous set can be discarded.
3. The scheme is has a relatively simple implementation.
4. The scheme does not suffer from the domino effect (see the section on pessimistic recovery, below.)

Despite its advantages, pure checkpointing suffers from some severe disadvantages. The most frequent criticism of the scheme is the requirement that sets of checkpoints form a consistent global state. To ensure this requirement, the RUs must coordinate their checkpointing activity. Once the decision to checkpoint is made, all RUs must cease computation and wait for the system to pause.

Typically this pause is implemented by using a barrier. A barrier is a coordination technique that restrains an RU at some point in the computation. In order to pause at a barrier, an RU must cease all normal computation, and wait for sent messages to be delivered. It is this lack of “in-flight” messages during the global checkpoint which guarantee that the resulting global checkpoint is consistent.

Once each RU has paused at the barrier, the system waits for some predicate to be satisfied before restarting the computation. In pure checkpointing, this predicate is “all RUs have checkpointed”.

This coordination increases the failure-free overhead in two ways. First, a distributed barrier typically requires a large number of messages to implement. Second, a substantial

delay is introduced in the computation as RUs busy-wait at the barrier.

2.2.5 Uncoordinated Checkpointing

In uncoordinated checkpointing, we do not require RUs to coordinate their checkpoints. Each RU checkpoints independently of the others. As in pure checkpointing, a failed process is rolled back to a previous checkpoint. Since the uncoordinated checkpoints generated with this scheme may not be mutually consistent, we need some way to bring the system to a global consistent state after re-instantiating the failed process.

One solution to this problem tried in early systems which used uncoordinated checkpointing was to notify the non-failed processes in the system that a process had failed and was being restarted. The recovery mechanism then detected those non-failed processes which were not consistent with the recovered process. Those processes were then restarted from their checkpoints.

If the resulting system state is not globally consistent, another set of checkpoints is selected, and the RUs which generated those checkpoints were rolled back. This process was repeated until the system state is globally consistent.

While it eliminates the barrier needed in consistent checkpointing (and thus the associated busy-waiting), this scheme is vulnerable to a phenomenon known as the “domino effect” [44] [49]. In the domino effect, an attempt to recover results in an unbounded cascade of rollbacks resulting from the recovery mechanism’s attempt to locate a set of consistent checkpoints. This phenomenon is depicted in figure 2.3. In this figure, process p_0 fails at time t and is restarted from checkpoint $C_{0,3}$. The system is in an inconsistent state because p_1 has received message m_7 , but p_0 ’s new state does not reflect sending this message.

Rolling back p_1 to checkpoint $C_{1,2}$ results in a system where p_0 received message m_6 which was never sent, causing p_0 to roll back to $C_{0,2}$, and so on.

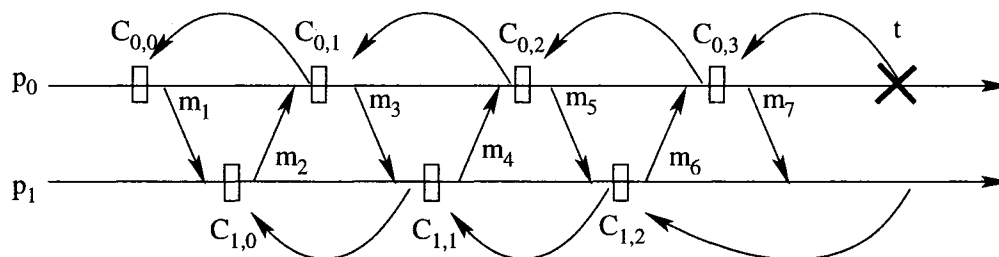


Figure 2.3: The Domino Effect

2.2.5.1 Pessimistic Recovery

To address the disadvantages of pure checkpointing, and the problem of the domino effect, a system formally called “checkpointing and rollback with pessimistic recovery” was developed. This scheme was first described by Borg, et. al [7]. We refer to this scheme simply as pessimistic recovery.

In pessimistic recovery [7] processes synchronously log message activity to stable storage. We use these logged messages to roll the failed process forward after a crash. In order to prevent the loss of messages, we synchronously log messages to stable storage. Thus, such a system never creates orphan messages, and guarantees that we can recover all messages after a system crash.

If a process failure occurs, we re instantiate the process from the checkpoint. After re instantiation, the log of received messages is “replayed”. That is, the recovery mechanism feeds the process the logged messages, and the recovering process consumes them in the normal manner. Any messages sent by the process are discarded. Once the log is exhausted,

the process state is consistent with those processes with which it had been communicating.

An important advantage of this method is that processes which do not fail do not have to be rolled back. Because of the synchronous logging requirements, we can guarantee that every message a process received before its failure is available for replay after restarting from a crash. In addition, no latency is incurred in sending messages to the “outside world”, since every past state of the computation is recoverable from information recorded on stable storage.

The major disadvantage of pessimistic recovery is the synchronous logging requirement which results in high failure-free overhead. Each message received by the process causes it to block while the message is logged to stable storage.

2.2.5.2 Optimistic Recovery

Optimistic recovery [56] is a fault-tolerance technique which allows computation, communication, checkpointing and committing to proceed asynchronously. As in pessimistic recovery protocols, optimistic recovery protocols save enough information to reconstruct a consistent state after a failure.

Optimistic recovery protocols ensure that the externally visible behavior of a system is equivalent to some failure-free execution. That is, they do not guarantee that the internal behavior of all processes is identical during every execution. Rather, they only guarantee that the same ordered set of messages is sent outside the system in both failure-free and recovery modes.

Optimistic recovery is based on the idea of dependency tracking. Dependency tracking allows a process to detect that it has performed some computations that causally depend

on the states that some other failed process has lost. Such computations are referred to as orphans.

Fundamental to the idea of dependency tracking is the concept of state intervals. Suppose that a recovery unit RU has already processed $n - 1$ messages and is ready to process its n th message $M(n)$. Its volatile storage is in some state $S(n)$. Given $M(n)$ and $S(n)$, the recovery unit will perform some series of operations, which may include sending messages to other RUs. Eventually, an RU will be ready to dequeue message $M(n + 1)$. We call the time between the receipt the message $M(n)$ and the time when the RU is ready to dequeue $M(n + 1)$ state interval $I(n)$.

Optimistic recovery schemes need to distinguish between identical states occurring before and after a failure. To do so, they consider the state interval to be a pair, $[\iota, \mu]$. ι is the incarnation number of the state, that is, the number of times this state has occurred previously due to rollback, and μ is the scalar state interval defined above.

The live history of an RU is the sequence of state intervals of the RU which have not been rolled back. A sequence of state intervals is shown in figure 2.4. Here we see the state intervals of an RU which has been restarted twice. The ordered pairs $[\iota, \mu]$ represent the incarnation number and scalar state interval.

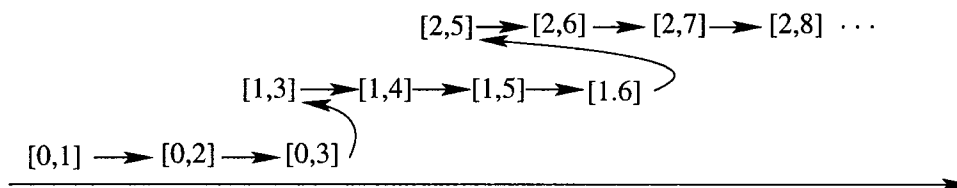


Figure 2.4: State Intervals of a Recovery Unit

The live history of this RU is the sequence $[0, 1], [0, 2], [1, 3], [1, 4], [2, 5], [2, 6], \dots$. The

live history of a recovery unit imposes a total order on the state intervals of the recovery unit. We use the notation $[\iota, \mu] \prec [\kappa, \nu]$ if $[\iota, \mu]$ precedes $[\kappa, \nu]$ in the live history.

In addition, we say that state interval $[\iota_i, \mu_i]$ of RU_i *immediately causes* state interval $[\iota_j, \mu_j]$ of RU_j if a message sent by RU_i during state interval $[\iota_i, \mu_i]$ is dequeued by RU_j at the beginning of state interval $[\iota_j, \mu_j]$.

The union of the relations \prec and immediately causes induces a partial order on the set of all state intervals in the system which are part of any RU's live history. The transitive closure of the union of these two relations over the set of all live state intervals is called the *causal precedence* or *dependency*. It is the key point to understanding optimistic recovery.

In a distributed system with m recovery units, any state of a particular RU will have at most m causal predecessors. We represent these causal predecessors as a vector $\langle d_1, d_2, \dots, d_m \rangle$. Each RU maintains this dependency vector as part of its internal state. The dependency vector is updated each time an RU moves to a new state interval or receives a message from another RU. It is in this way that an RU tracks those other RUs upon which it depends.

Should an RU fail, it is restarted from its earliest saved checkpoint. It then replays its log until either an orphan message is sent or the end of the log is reached. (It should be noted that orphans will occur only if some other RU has failed during this RU's restart. In the case where a single RU fails, optimistic recovery guarantees that no orphans are generated.) Once the RU has rolled forward, it begins a new incarnation by incrementing its incarnation number and broadcasting a recovery message to the other units in the system. This recovery message includes the new state interval of the RU.

RUs which receive the recovery message inspect their dependency vectors to determine if their current state interval causally depends on any state intervals lost by the recovering

process. If so, they restart and roll forward using the same procedure.

Consider the example shown in figure 2.5. Here we have two recovery units operating under the optimistic recovery scheme. Each RU is shown using two lines. The top line shows the message activity and state interval. The bottom line shows checkpointing activity and state-interval logging progress. RU_0 receives message $M(6)$ and enters state interval $[0, 6]$. Eventually during the processing which takes place in this state, RU_0 sends a message to RU_1 .

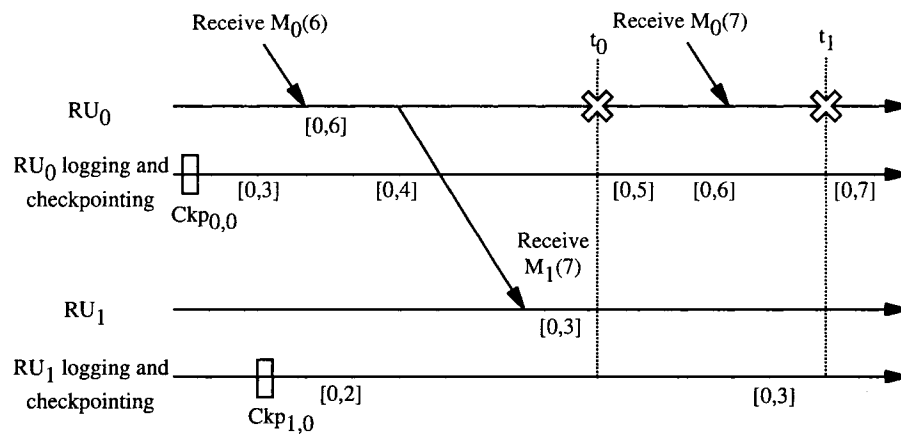


Figure 2.5: Optimistic Recovery

Suppose that RU_0 fails at time t_0 . RU_0 will restart from checkpoint $C_{0,0}$ and begin to replay its message log. At the time of its failure, RU_0 had only logged enough information to enable it to roll forward to state interval $[0, 4]$. Once RU_0 has replayed the log to that point, it will broadcast a recovery message to RU_1 indicating that it has rolled forward and is restarting at state interval $[1, 5]$.

RU_1 notes from its dependency vector that it is causally dependent on state interval $[0, 6]$ of RU_0 , since it received message $M_1(7)$ sent from that state interval. This message

has been lost. At this point, it rolls back to checkpoint $\text{Ckp}_{1,0}$ and then rolls forward.

Suppose on the other hand that RU_0 does not fail until time t_1 . After rolling forward, its broadcast recovery message will inform RU_1 that it is restarting at state interval $[1, 7]$. RU_1 's dependency vector indicates that its current state is not causally dependent on state $[1, 7]$ of RU_0 . Thus, RU_1 does not need to roll back.

Recovery units log input messages to stable storage in the background. In order to determine which of its computations are committable, and when a checkpoint and a portion of the log can be discarded, an RU must know which of its own messages have been logged as well as the status of logged-messages in other RUs. To this end, each RU maintains a “log vector” which lists a particular state interval for each recovery unit in the system. For RU_k , the log vector LV_k is a list of state intervals $[\iota_k, \mu_k]$ of RU_k such that all of $[\iota_k, \mu_k]$'s live predecessors have been logged.

Recovery units update their log vectors in the background. New log vectors are computed by inspecting the state of an RU's log and the logs of the other RUs in the system. How often a log vector is transmitted to the other RUs in the system is a tunable parameter. On receiving a new log vector, $\langle [\iota_1, \mu_1], [\iota_2, \mu_2], \dots, [\iota_m, \mu_m] \rangle$, RU_k computes the new log vector by taking the pointwise maximum of the old and new log vectors. Let $\text{LV}_k = \langle [I_1, M_1], [I_2, M_2], \dots, [I_m, M_m] \rangle$. Then the new log vector $\text{LV}_k(i) \leftarrow \max([I_i, M_i], [\iota_i, \mu_i]); 1 \leq i \leq m$ where \max is defined on pairs using the normal lexicographical ordering.

Once a recovery unit determines that a particular checkpoint or particular set of log entries are no longer needed (that is, no other RU is causally dependent on the state intervals spanned), these checkpoints and log entries may be discarded.

2.2.5.3 Causal Recovery

Causal recovery schemes combine the failure-free performance advantages of optimistic recovery with most of the advantages of pessimistic recovery schemes. It was first proposed by Alvisi and Marzullo [4]. Causal recovery

- does not require coordinated checkpoints,
- avoids the synchronous logging requirements of pessimistic recovery schemes, and
- never creates orphans, as optimistic recovery schemes do.

Causal logging protocols ensure what is called the “always-no-orphan” property by assuring that the determinant of each event that causally precedes the state of an RU is either located on stable storage, or is locally available to that RU. In such a system, non-failed RUs are able to guide the recovery of failed RUs using these determinants.

Consider figure 2.6. Messages m_5 and m_6 may be lost on the failure of either RU_1 or RU_2 . However, RU_0 will have either logged or have access to the determinants that causally precede its state. These events consist of the delivery of messages m_0, m_1, m_2, m_3 and m_4 .

Figure (a) represents the maximum recoverable state. Messages m_0, m_1, m_2, m_3 and m_4 have been logged to stable storage. Message m_5 and m_6 have been lost. Figure (b) shows the antecedence graph of RU_0 at state S . (Figures adapted from Elnozahy et al [17]).

As in other schemes, the message sender has logged the message content. Since RU_0 knows the order in which the messages were originally sent, it is able to “guide” the recovery of RU_1 and RU_2 . It is able to do this because it knows the order in which RU_1 should replay

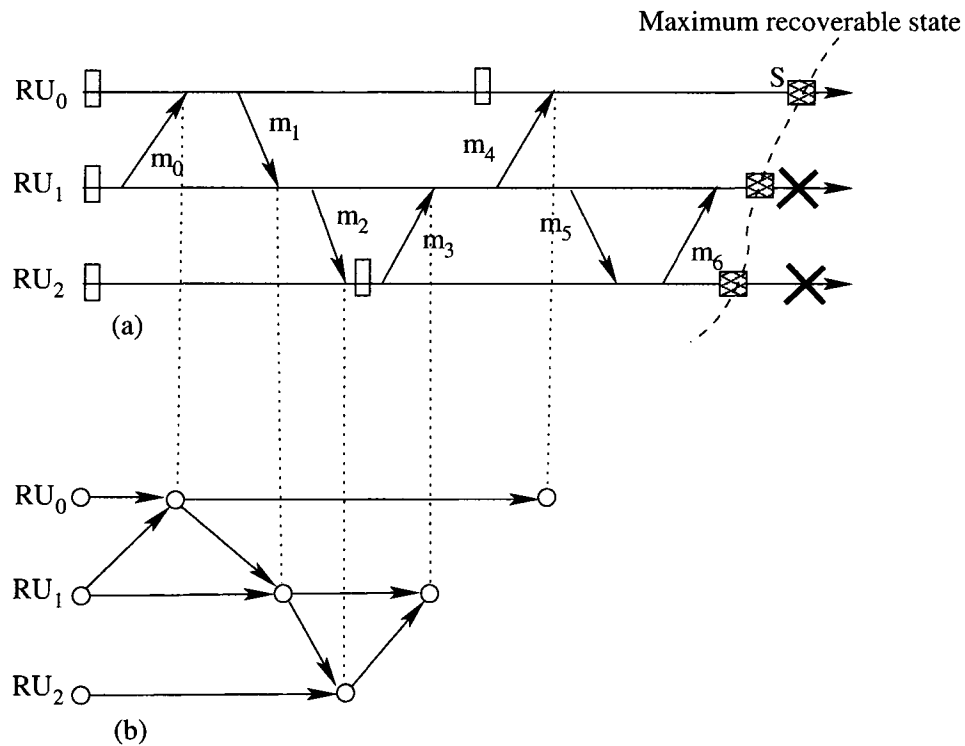


Figure 2.6: Causal Recovery

m_0 and m_2 to reach the state from which m_4 is sent. Similarly, RU_0 knows the order in which RU_2 should replay m_2 and m_4 .

Notice that information about m_5 and m_6 is not available anywhere. These messages may be regenerated and resent during recovery (perhaps in a different order), or they may not be sent at all, depending on how the computation unfolds during this incarnation. However, since they have no effect on the surviving recovery units, the resulting system state is still consistent.

Causal recovery protocols implement the always-no-orphan requirement by having RUs piggyback the determinants that have not been logged to stable storage on top of the messages they send to other RUs. One method of representing these determinants is by

using an antecedence graph. This is the method used in the Manetho system [18].

Figure 2.6b shows the antecedence graph built by RU_0 at the point where message m_4 was received. In an antecedence graph, the nodes represent nondeterministic events (in this case, simply checkpoints and message receipts) while the edges correspond to the “directly happened-before” relation on the events.

The major drawback of the causal recovery protocol as compared to other uncoordinated checkpointing schemes is the amount of failure-free overhead incurred by transmitting the entire antecedence graph with each message. In practice, there are several optimizations that can be performed which allow the recovery units to transmit just a subgraph of the antecedence graph [3].

2.3 Checkpointing and Rollback-Recovery Schemes and Files

Little work has been done on incorporating files into checkpointing and rollback-recovery schemes. Typically, files are treated as entities external to the computation. The canonical example of an entity external to a computation is an output device. It is difficult to “unprint” a page, once the computation has committed the output to the printer. Similarly, care must be taken before committing output to the file system since it is difficult to “unwrite” information once it has been written to a file system.

In most systems, a read operation is viewed as a message pair; a request message is sent to the file followed by a corresponding message receipt containing the result of the read operation. A write operation is treated similarly. Current checkpointing and rollback-recovery schemes which wish to fully incorporate files are required to perform a commit

before each destructive file operation.

Most of the previous work in this area concentrated on ad-hoc methods for recovering files after a crash. Some authors have proposed schemes which provide full support for all the usual file operations, but only under restricted types of recovery mechanisms. Below we review the main approaches to incorporating files into fault-tolerance schemes.

2.3.1 Plank and Litzkow

Early work on including files in concurrent systems treated the files as discrete external entities, separate from the state of a process. In the Libckpt package for Unix [42], Plank records just the process's file table in the checkpoint structure. A restarted process knows only the names, disk locations and offsets of file pointers. Thus, Libckpt requires files to remain open and unchanged over the course of the computation. This essentially limits the computation to using a static set of read-only files.

Litzkow [34] imposes similar restrictions in a package which performs checkpointing outside of the kernel. The viability of this scheme was demonstrated in the Condor system [33].

2.3.2 The SCR Algorithm

Wei and Ju [59] propose a scheme known as "SCR Algorithm" which provides for the full integration of file system operations in concurrent systems. Under the SCR algorithm, the file system is implemented on top of stable storage, and all processes sharing a file must coordinate their checkpointing operations to produce a set of consistent checkpoints. If the processes in a concurrent system have full access to a file system, this in essence means the scheme is restricted to operating only under pure checkpointing.

In the SCR algorithm, file operations are classed as idempotent (`read()`, `stat()`, etc.) and non-idempotent (`write()`, `creat()`, `link()`, etc.) We prefer to call these two classes of operations “non-destructive” and “destructive.”

Each machine in the system maintains an “undo stack” which is initially empty. In addition, the file system of each machine is protected by a “manager process” through which all file system operations are performed. A process wishing to perform a non-destructive operation contacts the manager process with its request, which immediately performs the operation and returns the result to the calling process.

If a process wishes to perform a destructive file operation, it contacts the manager process. The manager creates an “undo entry”, which contains enough information to undo the requested operation. It then, atomically, performs the operation and places the undo entry on the undo stack. The result can then be returned to the calling process.

The creation of the undo entry can be an expensive operation. For example, the entry for a `write()` operation that overwrites 10KB of a file needs to contain the file name, the offset of the write and the part of the original file overwritten. The undo entry for an `unlink()` must contain the entire contents of the original file, plus a copy of the file’s parent directory. Wei and Ju discuss several optimizations which reduce the size of the undo structure under specific circumstances.

From time to time, processes accessing files perform a coordinated checkpoint. After these checkpoints are written, the manager resets its undo stack.

If a process failure occurs, all of the processes in the group accessing the same file system as the failed process are halted. The manager process begins popping undo entries off the undo stack and applying the changes to the file system. Once the stack is empty, the file

system state is identical to the state at the time of the last global checkpoint. The halted processes are rolled back to the last gang-checkpoint and restarted. At this point, the computation continues.

2.3.3 AIPC

The most comprehensive proposal contained in the literature is that of Alagar, Rajagopalan, and Venkatesan [2]. They propose a scheme to completely integrate files into checkpointing and rollback-recovery schemes. Under their scheme, which they call “Accessing files through IPC”, or AIPC, a server process is created for each file accessed in a system.³

2.3.3.1 The Operation of AIPC

Since it is not always possible to predict which files will be used in a concurrent system, these server processes are created (and destroyed) dynamically. AIPC builds one server process for each file used in the system. This server handles all access requests for the file. Since the server does not know *a priori* which types of requests it will be required to serve, the file is opened with all permissions. A part of the server’s task is to discover whether processes attempting access to the file have the appropriate permissions.

When the server is created, it performs initializations, sets up the inter-process communication facilities necessary to communicate with the other processes in the system, checkpoints its own state, and then checkpoints the file. At this point the server is ready to accept requests to access the file. If a process wishes to perform a file operation, it contacts

³The authors are not clear on the need for a server process for *each* file in the system. For example, it is not clear whether the executable images of the programs themselves are rendered by server processes.

the server for that file. The server is responsible for all file operations normally performed in the kernel.

During a checkpointing operation, each process must save a list of the files in use. This information is used to reestablish communication with the server process for each file.

The server process checkpoints itself just as a regular system process does. To do so, it must make a copy of the files it is responsible for, and store the backup copies with the checkpoint information. The server's checkpoint structure also contains information about the system processes accessing the file. When the file server process restarts, it opens the checkpointed version of the file and establishes communication with the set of processes using the file.

Of particular advantage in AIPC is its flexible design. It can be used either for coordinated or uncoordinated checkpointing, and in schemes which require message logging.

2.3.3.2 Problems with AIPC

The primary difficulty with AIPC occurs during the checkpointing phase. During a checkpointing operation, the server process is required to make a copy of the file it serves. This is, in general, a time-consuming operation. In addition, the server must ensure that the backup copy of the file is consistent. Thus it can not serve any destructive requests for the duration of the checkpoint operation.

The authors suggest several possible solutions to this problem. An obvious optimization is to have the server checkpoint the file only if it has been modified. The also speculate that using an incremental checkpoint scheme, or checkpointing the file asynchronously, might improve the speed of the checkpoint operation and reduce the time the server is unavailable

to serve destructive requests. We analyze the performance of AIPC in section 7.2.4.

2.4 Using Files in Concurrent Computations

It is reasonable to ask if the methods mentioned above, as well as the scheme we propose in chapter 4 have any interest beyond that of an academic exercise. It turns out that there are a number of situations drawn from the field of scientific computing that might make use of the ideas we propose. In this section, we look at some of these practical applications.

Many of these applications fall into the class of problems referred to as “out-of-core” computation. Such a computation is loosely defined as one involving a data set that will not fit into the main memory of a single machine. Typically these applications do not perform any exotic operations (at least from an algorithmic standpoint.) Rather, the difficulty in programming them comes from the fact that many traditional versions of algorithms assume that all data resides in main memory for the entire course of the computation.

If some portion of the data set resides on disk, sorting, for example, becomes problematic since traditional sorting algorithms assume that any element in the data set can be accessed or modified in the same amount of time as all other elements⁴. The problem thus becomes one of decomposing the data set into blocks that will fit into main memory, and then modifying algorithms to operate with increased locality of reference on these decomposed sets. The cost of moving data between secondary storage and main memory can make a particular algorithm impractical if done indiscriminately.

⁴This is the traditionally assumed “RAM” model of computation [1].

2.4.1 Biotechnology

A recent example from the field of biotechnology appears in Samatova, et al [50]. An important problem in biotechnology is the modeling of the metabolic networks of different organisms. This modeling attempts to show how metabolic reactions and their products interact with one another within an organism.

While modeling small organism's metabolic networks can be done using traditional approaches, modeling genome-scale networks is currently infeasible because of the amount of data generated during the intermediate steps of the computation.

Two closely related techniques are currently used to model large metabolic networks: "elementary flux modes" [54] and "extreme pathways" [51]. Both of these methods rely on a mathematical technique called "convex analysis" borrowed from the field of linear algebra.

Convex analysis concerns itself with finding the steady-states of a system of matrices. Let S be an m by n matrix and let v be an n -element vector. A steady state solution to this system is any vector v such that $S \cdot v = 0$. In convex analysis, we are typically given the matrix S and asked to find all non-trivial values of v . We can use this method to model metabolic networks by letting S be a row of m metabolites and n reactions. V is a vector of reaction rates of the metabolites, called the flux vector.

Unfortunately, as the number of reactions involved in the metabolic network increases linearly, the size of the state space to be searched increases exponentially. The problem of finding all values of v is known to be polynomial-time reducible to the problem of finding all vertices of a convex polyhedron embedded in n -space. The vertices problem itself is known to be complete for NP.

Samatova, et al present a new algorithm for finding steady-states which can be effectively scaled to any number of processes. Their algorithm relies on a result in algebraic theory called the “reduced conical independence theorem”. This theorem reduces the number of references which must be made to intermediate results during the state-space search. By using this result in their algorithm, they are able to effectively keep large portions of the intermediate result data sets on disk for extended periods of time, thus reducing the amount of I/O traffic needed by their algorithm.

2.4.2 Astrophysics

“Hydrodynamic simulation” is an active field of research in the area of cosmology. Hydrodynamic simulation is used to process observations obtained from radio-telescopes to see if these observations agree with existing and new models of the large-scale structure of the universe. Trac and Pen [57] have recently implemented a new, more efficient version of a method used to process data obtained from these observations. This version takes advantage of out-of-core computational techniques.

The observed data used in this modeling falls into two categories. Short-range, high-resolution observations, typically obtained with non-terrestrial telescopes, and long-range, low-resolution observations obtained with series of earth-based telescopes. The size of the data sets involved in this work are in the multi-terabyte range.

The majority of the computation done in hydrodynamic simulation requires performing the fast-fourier transform (FFT) on observed data. Because the FFT references its data in a non-local fashion, locality of reference during such computations is low, and the sizes of the working sets of such computations approximate the size of the data.

Trac and Pen adopt a technique called the “two-level mesh scheme” first proposed by Couchman [12]. The two-level mesh scheme effectively partitions the data into small blocks. These small blocks represent short-range forces captured by the observations. After these short-range forces are analyzed, they undergo a coarse-grained sample, and from this sample, long-range forces are analyzed. The resulting analysis gives a high precision radio-picture of the large scale structure of the universe. The precision can be controlled by the granularity of the long-range force sample.

Trac and Pen have developed an implementation of this scheme wherein the data sets reside on secondary storage, but the short-range force observations fit into the memory of a single Compaq Alpha server. The algorithm works by repeatedly reading in short-range samples, computing the FFT of the sample, and then storing the result back to disk. After all the short range samples have been processed, the long range sample is read into memory. Depending on the granularity selected, the long-range sample may not fit into memory, resulting in heavy disk I/O load during this phase of the computation. Nevertheless, the authors point out, dividing the computation into short- and long-range phases and storing the intermediate results on disk significantly reduces the overall amount of I/O traffic required during the computation.

2.4.3 Applied Mathematics

QR factorization is a mathematical technique for decomposing a single matrix into two matrices with specific properties. Given an M by N matrix A , a qr factorization of A is a pair of matrices Q and R with the following properties:

1. $A = QR$.

2. Q is an M by M orthogonal matrix.
3. R is an M by N upper-triangular matrix.

QR factorization is a useful technique for computing the least-squares approximation from a collection of observed data. The technique is particularly useful in problems with large numbers of parameters and observed data values, because it converges faster than traditional methods of least-squares approximation.

One particular application of qr factorization is in the earth sciences, where problems can involve tens of thousands of parameters and millions of observations. Gunter, et al [24] cite an example where the technique was used to analyze data and develop a more accurate estimate of the earth's magnetic field.

When large data sets are involved, the matrices and intermediate sub-matrices often times will not fit into primary memory simultaneously. Since the intermediate sub-matrices are referenced multiple times in later steps of the computation, this makes the problem an ideal candidate of out-of-core computation.

Gunter, et al [23] have developed and implemented an algorithm which is able to minimize the number of times a particular sub-matrix must be referenced. They do this by first simplifying the matrix using what is known as the "Householder transformation". This enables them to store intermediate results on disk, while minimizing the disk I/O involved when the sub-matrices are referenced at later steps in the factorization.

2.5 Concluding Remarks

All of the schemes so far proposed to fully integrate files into fault tolerance schemes suffer from deficiencies.

- They severely restrict the types of operations that can be performed on files, as in LibCkpt.
- They are restricted to the simplest form of rollback-recovery, as in SCR.
- They are inefficient (AIPC.)

The scheme we discuss in chapter 4 suffers from none of the problems described above.

- It does not restrict the types of operations which can be performed on files.
- It is flexible in that it can be used in any type of checkpointing and rollback-recovery scheme because it incorporates the part of the RU state stored in the file system in to the checkpoint.
- Its efficiency is comparable to schemes which do not incorporate files.

Our scheme relies on the special properties of the log-structured file system. In the next chapter, we briefly review the history of file systems and then discuss the log-structured file system and the special properties it possesses that we use in our scheme.

Chapter 3

Log-Structured File Systems

It's a poor sort of memory that only works backwards.

Lewis Carroll, *Alice's Adventures in Wonderland*

We begin this chapter with a review of the Unix file system, and discuss its influence on modern file system design. We then look at Rosenblum's Log-Structured File System (LSFS) and discuss:

- the ideas which motivated the development of the first LSFS,
- the advantages which LSFSs promised and those that have been born out,
- a brief survey of the modern file systems influenced by LSFS design concepts, and
- the special properties of the LSFS which we use in the scheme discussed in chapter 4.

3.1 The Unix File System

What is traditionally referred to as “the Unix file system” was part of the original release of Unix (called Version III, or V3) by AT&T [46]. A later version, called the Unix Fast File System (Unix FFS) [38], shared all of the basic design elements with the original AT&T version, and became popular in the 1980’s with the release of 4.2BSD. The 4.2BSD file system is still influential today.

Both of original and 4.2BSD Unix file systems were themselves based on the Multics [19] file system. Today, the term “Unix file system” refers to those elements common to both the original release and the Unix FFS.

The Unix file system introduced several concepts to file system design, and many modern file systems are indebted to it. Among the file systems which owe their basic design to the Unix file system are Sun Microsystems’s Solaris file system, SGI’s Irix file system, the Linux Ext2 [10] and Ext3 [58] file systems, and the Be file system [22].

3.1.1 Terminology

We begin our discussion by introducing some terminology.

Block A disk block is the smallest amount of information that can be read or written to a disk at one time. Everything that a file system does is composed of operations on blocks. If there is a possibility for confusion, we distinguish between a copy of a block in memory and one on disk by using the terms “in-memory block” and “disk block”, respectively. The generic term “block” refers both to the physical space on disk, and the data it contains.

Superblock The superblock is special disk block that describes, in general terms, the file system which resides on the disk. The superblock also serves as the starting point for locating information on the disk.

Disk cache The disk cache is an area of memory dedicated to holding copies of disk blocks after they are read from, or before they are written to disk. Mechanical disk drives are slower than solid state devices such as RAM. If the file system can successfully keep needed information in the buffer cache, the overall speed of the file system will improve, since the physical device will be accessed less often. The disk cache is sometimes referred to as the “buffer cache”.

Metadata Metadata is information about the attributes and locations of data stored on the disk. Examples include a directory’s size in bytes, or a file’s access permissions.

Inode The block where the file system stores all the necessary metadata about a file is called the inode. In particular, it stores the physical locations of the file’s data. The term inode is a contraction of “information node” [6]. An inode is sometimes referred to as a “file control block” or “FCB”.

Indirect block The file system may need a long list of block addresses to record the locations of a file’s blocks. If this information can not fit into a single disk block, we store the additional information in a tree structure. The blocks that make up this tree structure are called indirect blocks.

Directory Modern file systems are designed with a hierarchical structure. In Unix type file systems, each level of the hierarchy is called a directory. A directory is a list of

the files at a particular level, and the directories of the next level of the hierarchy.

Extent If the blocks which comprise a disk can be allocated contiguously on the physical device, inode space can be conserved by storing the starting address and the total length of the contiguous blocks, rather than a pointer to each individual block. These contiguous runs of blocks are called extents.

Journal A journal is a list of modifications that have been performed on a file system, but that are not yet reflected in the on-disk copy of the file system. The journal guarantees the consistency of the on-disk copy of the file system after a system crash. Just exactly how such a system guarantees consistency is discussed in section 3.2.

3.1.2 A Unix File System

In this section, we discuss a simple file system layout that includes all the basic structures of a typical Unix like file system. Figure 3.1 shows the basic components. The diagram illustrates the structures what comprise the top level, or root, of a simple Unix file system. The top level, or root, contains File A, Directory B, ..., File Z. File A consists of a single block. Directory B needs multiple blocks of storage. File Z is large enough to need an indirect block to store pointers to all the blocks in the file.

Figure 3.1 is somewhat simplistic in its representation of directories. Rather than storing a pointer to the block containing the inode, directories store an inode's unique number. We discuss this matter further in section 3.2.1.

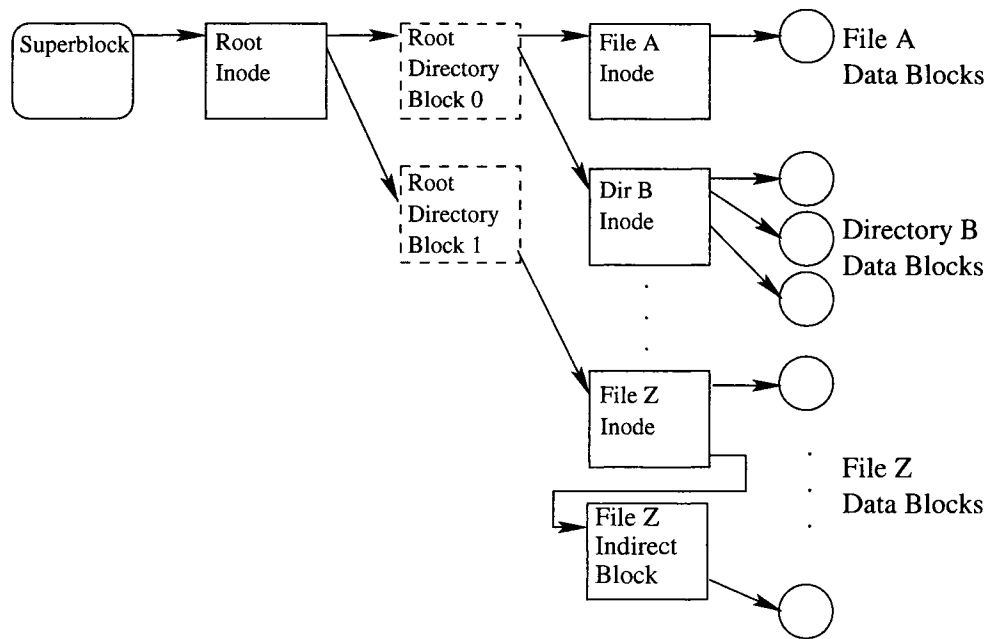


Figure 3.1: A Unix File System

3.1.2.1 The Superblock

The superblock structure contains information about the file system as a whole. As the name indicates, the superblock fits into one disk block. The superblock must contain enough information to allow the operating system to successfully access the data on the disk. The superblock typically contains

- a “magic number” indicating the type and version of the file system on the disk,
- information about the number of physical blocks on the disk,
- the file system creation date,
- the date the file system was last attached to or detached from the operating system (the “mount” or “unmount” time),

- the file system attributes (such as “read-only”, , “don’t update file time-of-last-access”¹, or “encrypted”,)
- a pointer to the block containing the root inode, and
- a list of used and free blocks on the disk, either in the form of a pointer to a bitmap (used in the Ext2, Ext3, NTFS and Be file systems), or a pointer to the head of a list of free blocks (used in the Unix FFS.)

Since an operating system must be able to find the superblock of a disk when mounting it, the superblock is typically stored in a fixed location on the disk. Because the file system essentially becomes inaccessible should the superblock be damaged, many file systems duplicate the information contained in the superblock at multiple fixed locations on the disk.

3.1.2.2 Inodes

Inodes are the basic metadata structure associated with each file and directory on the disk. (Unix does not distinguish between directories and other types of files except that users are not allowed to make arbitrary changes to the contents of a directory.) A typical inode is shown in figure 3.2.

The inode typically contains

- the creation, modification and last access times of the file (referred to as the ctime, mtime and atime,)

¹This option is useful on devices like Usenet news spools where the system administrator does not care about file access times.

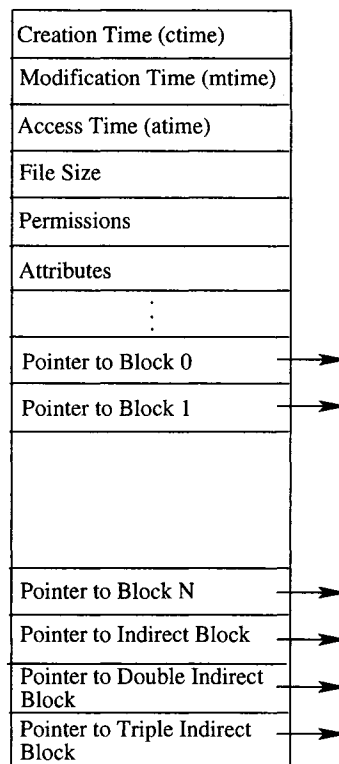


Figure 3.2: An Inode

- the number of bytes in the file,
- the permissions of the file,
- any special attributes that the file may have (such as being a symbolic link and/or a directory,)
- a group of block pointers that give the physical locations of the first few blocks which comprise the disk.
- pointers to the indirect blocks of the file.

For efficiency, we want to be able to transfer an inode between disk and memory with

a single read or write, so an inode must fit into one block. This requirement constrains the number of block pointers that can be stored in an inode. To overcome this limitation, Unix file systems use a series of direct, indirect and multiply indirect pointers to hold the locations of disk blocks.

As an example, suppose that a block is 512 bytes in length. If the metadata of an inode uses 24 of those bytes, and we assume disk block addresses are 32 bits, then the inode has room for at most 122 block pointers. This limits the maximum number of blocks in the file to 122, giving a maximum file size of 61KB.

The solution is to reserve several of the block pointers at the end of the inode and redefine them to point, not to blocks containing file data, but to blocks containing pointers to file data, or blocks containing pointers to blocks containing pointers to file data. This multiple indirect scheme allows a much larger addressable file space, at the cost of some extra overhead when accessing those distant blocks. Figure 3.3 shows an example of a scheme which uses single and double indirect blocks.

The ext2 and ext3 file systems available with the 2.4 kernel versions of Linux operating system use a triple indirect scheme. This allows an addressable file size of approximately 4 terabytes. However, due to limitations in the block device layer of the 2.4 kernel, the maximum file system size is 1 terabyte.

3.2 Log-Structured File Systems

Traditional Unix file systems have proven to be remarkably successful. The file system has been used on machines with disk sizes ranging from just a few megabytes to systems

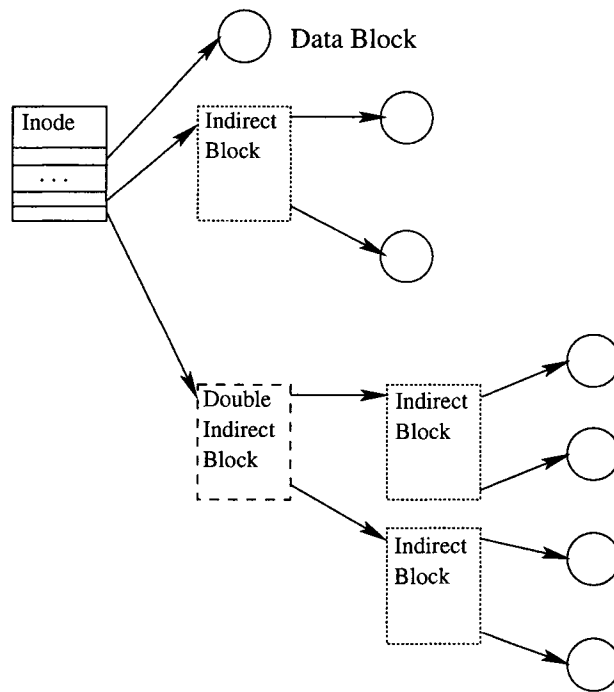


Figure 3.3: An Indirect Block Scheme for Increasing the Number of Addressable File Blocks

with hundreds of gigabytes of storage. By the late 1980's, however, changes in primary memory and disk storage capacities and relative speeds begat a reconsideration of file system implementations.

One such reconsideration, first proposed by Rosenblum, made its appearance in the Sprite Network Operating System developed at U.C. Berkeley [40]. What Rosenblum called a “log-structured file system” (LSFS) was first described in detail in 1991 [48]. His dissertation [47] on the subject won the 1992 ACM/doctoral dissertation award.

Rosenblum made three key observations about hardware that motivated his work.

1. Traditional file systems were not taking full advantage of the large main memories that had become available.

2. The gap between the speeds of solid-state memory and mechanical disks was increasing, and, barring some unforeseen breakthrough in storage technology, would continue to do so.
3. The decreasing cost per byte of mechanical storage meant that systems would be equipped with large and larger disks. The time required to recover the file systems on such disks after a crash was becoming unacceptable.

These observations led Rosenblum to several conclusions. First, as the size of main memories increased, the amount of memory that could be dedicated to the disk cache would increase. This increased size would mean that a larger portion of the spacial locality of actively used files would be captured by the cache. As the cache is filled, fewer physical disk reads would be needed to satisfy logical application reads, since the files would already be in the cache. Thus, traffic to the disk would become dominated by writes.

Second, the factor most responsible for limiting the speed of modern disks under a typical workload is the seek time. Reducing the number of seeks and the average seek distance would increase the observed transfer rate of the disk.

Third, traditional methods of restoring consistency to a file system after a crash require scanning all the metadata structures on the disk. This time grows linearly as the size of the disk increases.

The LSFS attempts to solve all three of these problems. It does so by adding a special data structure called *the log* to the file system. The log is an append-only structure that holds copies of all modified disk blocks. As disk blocks are modified, they are written to the end of the log, rather than being updated in place as in a traditional file system. Thus

an LSFS never overwrites any block on the disk. (There is a single exception to this rule, which we discuss below.)

Typical scientific and engineering workloads contain many writes that are smaller than the block size [41]. The LSFS converts these small writes into one long sequential transfer. Since only seeks to adjacent tracks occur, this write occurs at nearly 100% of the disk's theoretical transfer rate.

3.2.1 The Organization of a Log-Structured File System

The LSFS retains the concepts of the superblock, inode and the indirect block scheme from traditional Unix file systems. Two additional structures are needed, however, to support the log.

Since we never overwrite disk blocks, each write will place a block at a new location on the disk. This scheme presents no problems when recording the locations of regular data blocks, since we simply update their locations in the inode or indirect block.

However, rewriting inodes introduces a complication. Traditionally, space for some fixed number of inodes is reserved near the beginning of the disk when the file system is created. This allows the system to store the inodes contiguously, and makes locating a particular inode easy since it can be done by a simple arithmetic calculation using the inode's number.

In an LSFS, however, we never overwrite an existing inode, so the location of an inode may change. Thus, we need an additional structure to remember the locations of inodes. This structure is called *the inode map*, or simply just *the imap*.

Since the data stored in the imap changes as inodes are written to new places on disk, the imap itself may change. Thus, we need some way of tracking the changing location of

the imap. We store the location of the imap in *the checkpoint region*. The location of the checkpoint region is stored in the superblock. The superblock is the only structure on disk whose location does not change. Whenever the log is written to disk, we write the data blocks, inodes, imap and checkpoint region to disk, in that order.

Figure 3.4 shows the logical relationship between these structures. Figure 3.5 shows these structures after the log has been written to disk. In the LSFS represented in figure 3.5, file *A* has been deleted, and block 1 of file *B* has been modified.

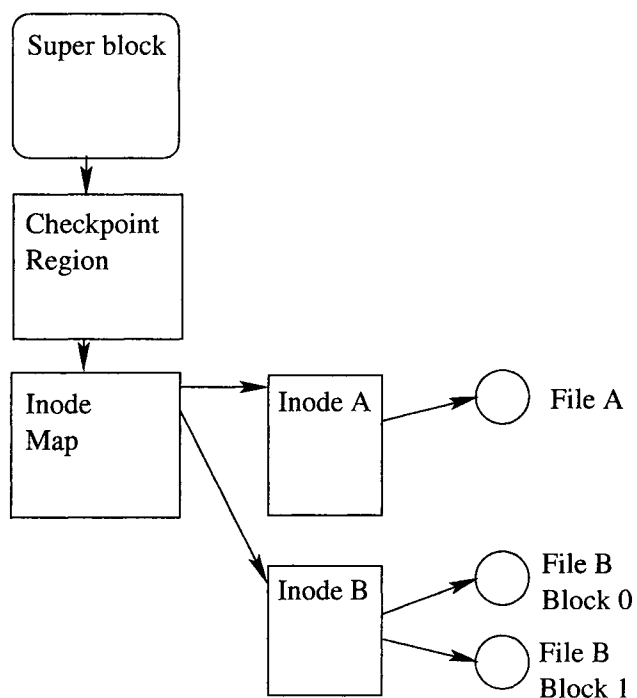


Figure 3.4: A Log-Structured File System

An example of the physical layout that these structures might have on disk, both before and after the log has been flushed, is shown in figures 3.6 and 3.7.

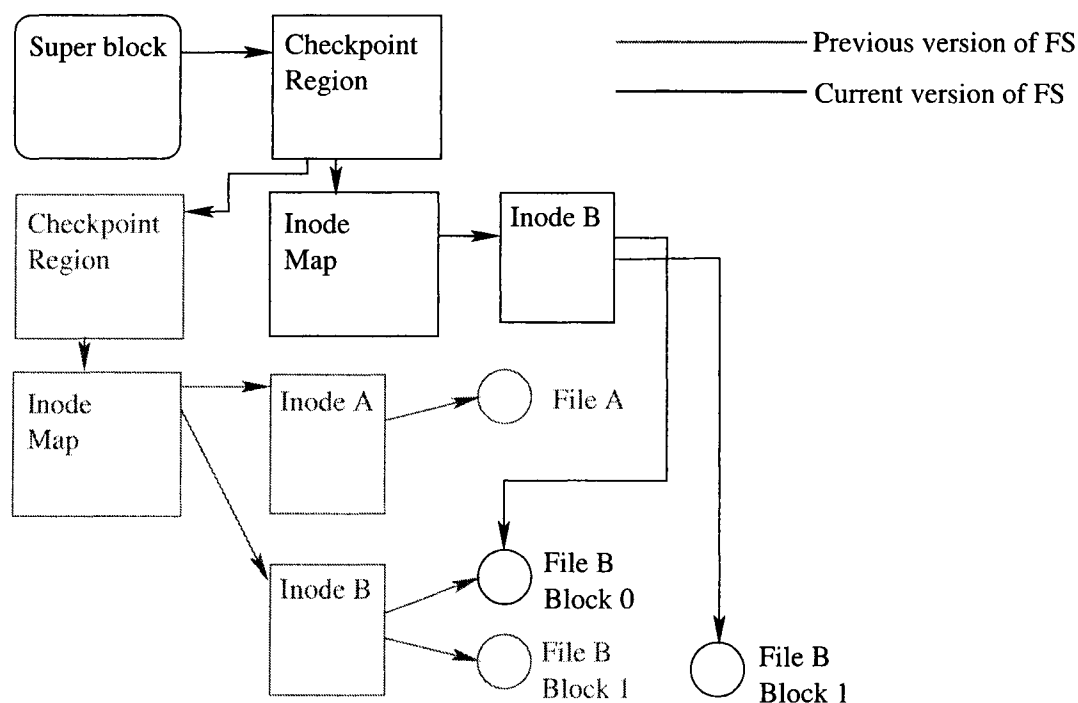


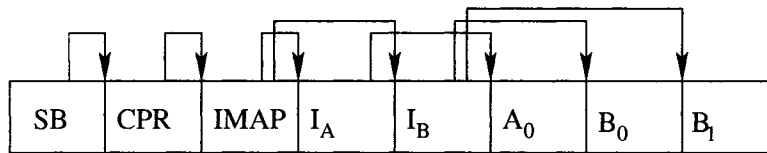
Figure 3.5: A Log Structured File System After the Log has been Written to Disk

3.2.2 Crash Recovery

The log-structured file system uses the redundant data on the disk to accelerate crash-recovery. After a system crash, the file system may have been left in an inconsistent state. For example, the contents of a new file may have been written to disk, but the system crashed before the associated inode could be updated.

In traditional file systems, the system cannot easily determine which file system structures are consistent with the data on the disk, so it must scan all the meta-data structures on the disk to restore consistency. In a log-structured file system, all of the most recent changes are easy to find; they are at the end of the log.

To effect a file system recovery after a crash, the system notes (using a flag in the



SB = Superblock

CPR = Checkpoint Region

I_A = Inode A

A_0 = File A, Block 0

Figure 3.6: The On-Disk Layout of a Log-Structured File System

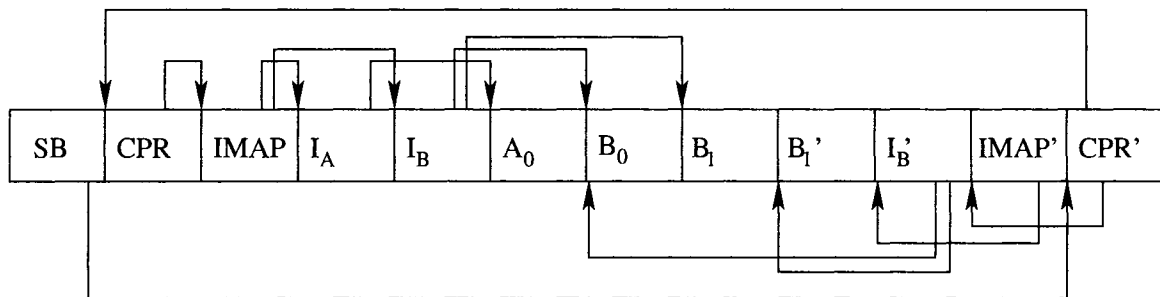


Figure 3.7: The On-Disk Layout of a Log Structured File System After the Log has been Written to Disk

superblock) whether the file system was properly unmounted before the system was powered off. If it was not, the system locates the most recent checkpoint region, reinstantiates the file system from that checkpoint, and then updates the file system from the log. If the checkpoint region is corrupted (due to a crash while it was being written), the LSFS consults the previous checkpoint region. It is for this reason that the LSFS always keeps two checkpoint regions on the disk. It needs a valid location for the imap even if a crash occurs which leaves a checkpoint region in an inconsistent state.

3.2.3 Cleaning

The performance of a log-structured file system depends on having large contiguous segments of disk space in which to write the log. Since the disk will eventually become full as

the size of the log grows, it is necessary to have some mechanism to remove old data from the log. As this old data is removed, the disk will tend to become fragmented, and so a method is needed to coalesce the free and used blocks.

Both of these tasks are typically performed during a procedure called cleaning. Rosenblum studied several methods for implementing the cleaning daemon. The cleaner can either run continuously as a background process, or it can be called from time to time when the machine is idle, or when the number of contiguous free blocks falls below some threshold. Rosenblum [48] speculates that the long-term performance of an LSFS is closely tied to the cleaning policy used, but his research did not discover an optimal cleaning policy.

3.2.4 Previous Implementations of Log-Structured File Systems

Rosenblum [48] described the original implementation of the log-structured file system, and gives performance and experimental results when it is used with the Sprite network operating system [40]. Seltzer et al. [55] describe an implementation for the 4.4 BSD operating system. The Network Appliance Corporation currently provides a log-structured file system implementation which they couple with non-volatile memory and a RAID-4 disk in their FAServer family of NFS servers [25]. It is a standalone network attached storage device which provides high performance NFS service.

3.2.5 Log-Structured File Systems are History Preserving

Since the checkpoint region completely encapsulates the information necessary to access a snapshot of the filesystem, we speculated that an LSFS could be modified to preserve a series of checkpoint regions, and thus a series of snapshots of the file system. Since, with

the exception of the superblock, no disk block in an LFS is ever overwritten, each series of disk snapshots is exactly preserved in the version of the file system rooted in the associated checkpoint region.

In the next several chapters, we discuss how we have leveraged this history preserving mechanism to create a prototype system that allows for the efficient incorporation of files into fault-tolerant concurrent systems.

Chapter 4

Checkpointing and Rollback-Recovery with Files

We are what we repeatedly do.

Aristotle

In this chapter we describe a method for efficiently and transparently incorporating files into systems which use checkpointing and rollback-recovery for fault-tolerance. We begin by discussing the concept of vector time, which we use to track dependencies between processes and the file system. We then describe the checkpointing, rollback, and rollforward processes.

4.1 Our Scheme

The previous attempts to integrate files into checkpointing and rollback-recovery schemes have all had certain inherent limitations. If they treat files as external entities, then the

frequency of commit operations severely limits the amount of parallelism among the nodes in the system. Attempting to reduce the number of commits required restricting the operations that can be performed on files, effectively making files second-class objects in the computation. Unfortunately, fully integrating files into fault-tolerance schemes has limited these schemes to pure checkpointing only, and in addition has imposed a severe performance penalty on such systems.

Attempting to take a snapshot of a traditional file system results in excessive failure-free overhead because these schemes need to make disk-to-disk copies of large amounts of data in order to make file operations undoable. This is the key observation which has motivated our work.

The goal of our work was three-fold. We wanted to develop a scheme that addressed all of the above problems. Thus, the scheme must have all the following characteristics:

Application Transparency The scheme should allow a program to access files using the standard set of Unix file operations.

Fault-Tolerance-Scheme Independence The scheme should be general enough to allow it to be used with any checkpointing and rollback-recovery scheme. That is, the scheme must fully support checkpointing, rollback and rollforward of the file system.

Efficiency The scheme must be efficient enough so that including file systems in the checkpoint does not make the failure-free overhead prohibitively large.

We chose to use the log-structured file system as the basis for our scheme. The history mechanism of the LSFS gives us an efficient method of arbitrarily taking snapshots of the file system. Since nearly the same amount of computational resources are used to snapshot

an LSFS as to `sync()` a traditional file system, the LSFS essentially gives us the ability to checkpoint an entire file system while expending very little extra computational resources.

Since LSFSs guarantee that the on-disk state of the file system is always self-consistent after a snapshot, the scheme can easily be integrated into existing checkpointing methods. Combining this with the timestamping mechanism described below allows us to easily and efficiently roll the file system forward and backward through time. Thus, the scheme is independent of the method used to provide fault-tolerance.

Since multiple processes can concurrently access a single file under our scheme, we need some way to ensure that we can roll the file system forward and backward and still maintain consistency with all the processes accessing it. To do so, we require the file system state to maintain a strong sense of temporal causality with the processes accessing it. Vector time allows us to capture exactly this characteristic.

4.2 Vector Time

Vector time was proposed independently by Mattern [35] and Fidge [20, 21] to provide a characterization of causality among processes. It is a generalization of Lamport's logical clocks [28].

A concurrent system consists of a set of N communicating processes. In vector time, each process p_i has a vector $V_i^j, 1 \leq i, j \leq N$. We refer to this value as the vector time or vector timestamp of the process. All processes in a concurrent system implicitly agree on an initial timestamp. Typically this timestamp is $[0_1, 0_2, \dots, 0_N]$.

Let e_i^k be the k th event occurring in process p_i . We increment V_i^i before each event in

a process, so when the k th event occurs, $V_i^i = k$. We use the notation $V_i(e_i^k)$ to denote the clock value of an event e_i^k in process p_i .

When process p_i sends a message to process p_j , we piggyback p_i 's timestamp on top of the message. P_j updates its vector time clock to reflect both p_i 's and its own idea of the current vector time.

More formally, the following rules are used to maintain vector time clocks:

1. When the k th event in p_i , e_i^k , occurs, p_i increments the i th component of its vector time clock. Thus, $V_i^i \leftarrow V_i^i + 1$. For $1 \leq l \neq i \leq N$, V_i^l is not changed.
2. If s is a send event in p_i and r is the corresponding receive event in p_j , then the clock of p_j is updated to reflect p_i 's knowledge of the clocks in all other processes. That is, $V_j^k \leftarrow \max(V_i^k, V_j^k)$, $1 \leq k \leq N$.

An example of a three process system is shown in figure 4.1. Processes increment their vector time before each send and receive event by incrementing the part of the timestamp that represents their own process. After the receipt of a message, a process updates the entire timestamp by forming the pairwise maximum of their own clock and the sender's clock.

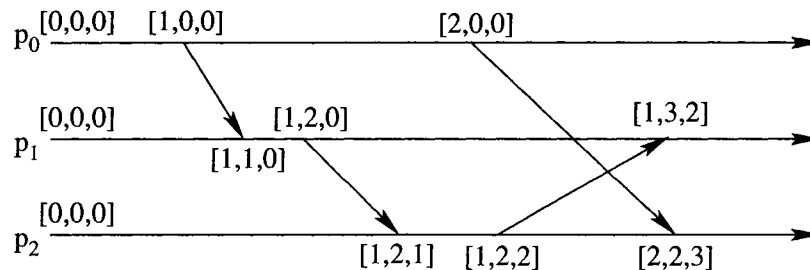


Figure 4.1: Vector Time in a Three Process System

Mattern defines the following properties of vector time clocks: For any two clocks V_i and V_j :

- $V_i = V_j \iff \forall k : V_i^k = V_j^k$;
- $V_i \leq V_j \iff \forall k : V_i^k \leq V_j^k$;
- $V_i < V_j \iff (V_i \leq V_j) \wedge (V_i \neq V_j)$;
- $V_i \parallel V_j \iff (V_i \not\leq V_j) \wedge (V_j \not\leq V_i)$.

The ordering of vector time clocks exactly captures the causal order between two events e_i and e_j occurring in processes p_i and p_j . In particular, recalling that \rightarrow is Lamport's "happens before" relation: $e_i \rightarrow e_j \iff V_i(e_i) \leq V_j(e_j)$. Figure 4.2 shows the partial order induced on the send and receive events shown in figure 4.1 by the vector time clock relation.

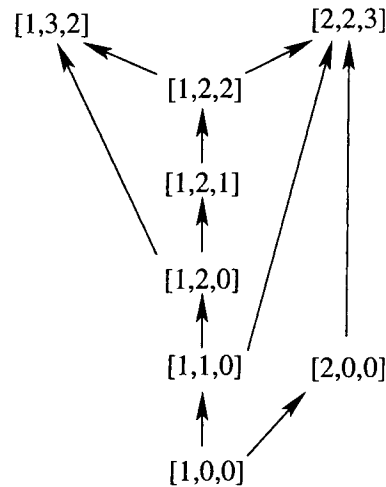


Figure 4.2: The Partial Order Induced on Events by Vector Time

4.3 File System Checkpointing and Process Checkpointing

As noted in chapter 3, the LSFS maintains a continuous history of the operations performed on the file system. Each time data blocks are written to disk, we write the blocks to new physical locations. We also write the updated inodes and any associated indirect blocks to new locations, thus preserving the old state of the file system by leaving the previous copies on disk undisturbed. Writes of the imap and checkpoint region also lay down new copies on the disk.

Thus, to checkpoint a LSFS, we need only flush all dirty blocks from the disk cache, and then update the disk's metadata structures. Of these operations, only the writing of the imap and checkpoint region are not performed during similar operations by traditional Unix file systems.

Process checkpointing has been well studied [49] [7] [34] [16] [8] [32] [43]. We describe our implementation of processes checkpointing, along with an implementation of an LSFS in chapter 5.

In our system, we extend process checkpointing to include a file system checkpoint. We do this by initiating a file system checkpoint concurrently with the process checkpoint. We then store the two together in a list associated with each process. Included in the checkpoint is information about the processes' interaction with the filesystem, such as the list of files the process has open at the time of the checkpoint.¹

¹In our prototype system, we open each file with both read and write permissions. In addition, we require application processes to manage their own file pointers in user space. In a real system, both of these pieces of information are managed by the operating system and would need to be stored in the checkpoint.

4.4 Restoring System Consistency after Process Failure

We face a particular difficulty in rolling a file system forward from a checkpoint after a crash. To wit, each of the processes accessing the file system will have a part of its state contained not only within local memory but also in the file system as well. This is much different from the typical concurrent or distributed system model where process state is subsumed by the process itself.

Consider the example shown in figure 4.3. This example illustrates a simple producer-consumer situation, where the processes synchronize using message passing, and a file serves the role of the buffer. One process reads the file while the other writes to it. The processes synchronize by exchanging messages. Assume for this argument a buffer large enough to hold two items of data.

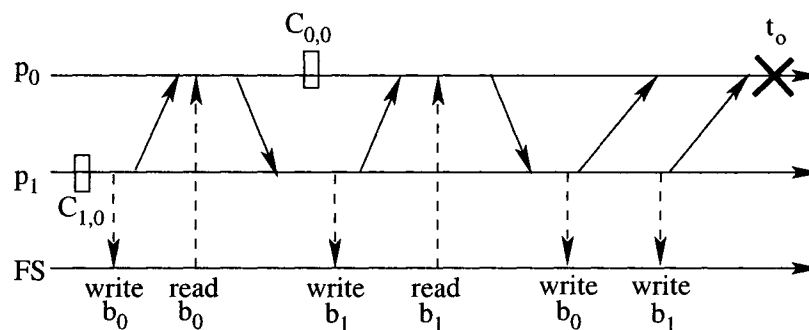


Figure 4.3: Sharing a File in a Concurrent System

At time t_0 , process p_0 crashes. We can not simply rollback p_0 to $C_{0,0}$ without rolling back the file system. To see why, assume that p_0 is rolled back to checkpoint $C_{0,0}$ but the file system is left untouched. At this point, p_0 begins to replay its log of received messages. These messages will instruct the process to read from the buffer. Unfortunately, the data

which the process should be reading has already been overwritten by p_1 .

The obvious solution then is to rollback the file-system along with process p_0 . Unfortunately, this introduces additional difficulties. After the process and file system have been rolled back, data written by p_1 after checkpoint $C_{0,0}$ was created will have been lost. The solution here then is to rollback process p_1 to checkpoint $C_{1,0}$, and then roll it forward by replaying its message logs. Note that we do not rollback the file system to its state preserved in checkpoint $C_{1,0}$.

There is a subtle difficulty illustrated in this example: Since both processes in the system are asynchronously replaying messages from their logs, they proceed independently. We need some way to coordinate access to the file while the two processes are rolling forward. Our solution is similar to other logging recovery protocols in that we require each process to log not only received messages, but also data which has been read from the file. We then control the extent of roll forward using vector time.

4.4.1 Vector Time and Checkpointing

In our protocol, vector time controls the extent of process rollforward. For our purposes, any file system access will be considered an event and cause the process vector time to increment. Particular applications may define additional events.

We assign a vector time timestamp to the shared file system and each process in the system. Processes piggyback these timestamps on messages sent to other processes. When a process issues a destructive file system request (a `write()`, `close()` or `unlink()`), it passes its timestamp along with the request to the file system. The file system updates its vector time before returning the result. When a non-destructive request is made of the file

system (an `open()` or `read()`) the file system returns its vector timestamp along with the result, and the process updates its vector time clock using this information.

Thus, we update the process vector time on an internal event, a non-destructive file operation or a message receipt. We update the file system vector time on a destructive operation, including a file system snapshot. Our motivation in attaching timestamps to file operations is to have the file system coordinate the vector time among processes communicating through the file system. Table 4.1 summarizes the rules for updating the vector times of the file system and processes which initiate internal events and file system operations.

Vector Time Propagation Rules		
<i>operation</i>	<i>initiating process</i>	<i>file system</i>
non-destructive	update process timestamp	no change
destructive	no change	update file system timestamp
internal event	increment V_i^i	no change

Table 4.1: Vector Time Propagation Rules

For example, during a write, the file system was modified by some event in the process, so the file system's idea of the time should be at least as late as the process's. Alternately, a read from the file system is the last step in a possible communication between processes, so the read event should causally succeed the latest file system event. Events per se do not occur in the file system; the file system only reflects the vector time of processes which change it. See figure 4.4.

Because we want our system to be adaptable to schemes other than pure checkpointing, we require the processes to log received messages. In addition, we also require processes to log any information read from the shared file system.

Finally, we require each process in the system to take an initial checkpoint before the

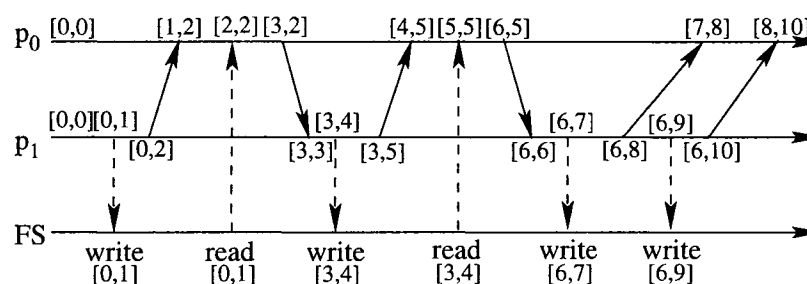


Figure 4.4: Vector Time Propagation Between Processes and the File System

computation begins. A formal checkpoint is not actually needed since we can always restart processes from the beginning, but a later argument is simplified by assuming at least one checkpoint always exists.

4.4.2 Recovery

During the recovery stage, the process which crashed is restarted from its latest checkpoint. This process notifies others with which it shares the file system that a restart is in progress and that they should stop normal computation and wait for restart instructions.

During the resurrection of the failed process from the checkpoint, we rollback both the process state and the file system. The process then notifies the other processes with which it shared the file system that it has restarted, and informs them of the vector time contained in the rolled back file system.

We now restart the non-failed processes. For each of these other processes, we choose the most recent checkpoint which was taken before the last checkpoint of the crashed process, and restart them. Note that as we rollback these non-failed processes, we do not rollback the file system.

The non-failed processes then roll forward. As they do, they replay received messages

and the results of non-destructive file requests from their logs. Requests by the process to perform destructive file operations are ignored, and sent messages are discarded. We roll the processes forward until their vector clocks indicate that they are about to perform some event which is not causally earlier than the vector clock of the file system, or until the log is exhausted. At this point, these processes resume normal operation.

Consider figure 4.5. It shows a three-process system accessing a shared file system. Each process reads from and writes to the file, and intermittently checkpoints itself and the file system.

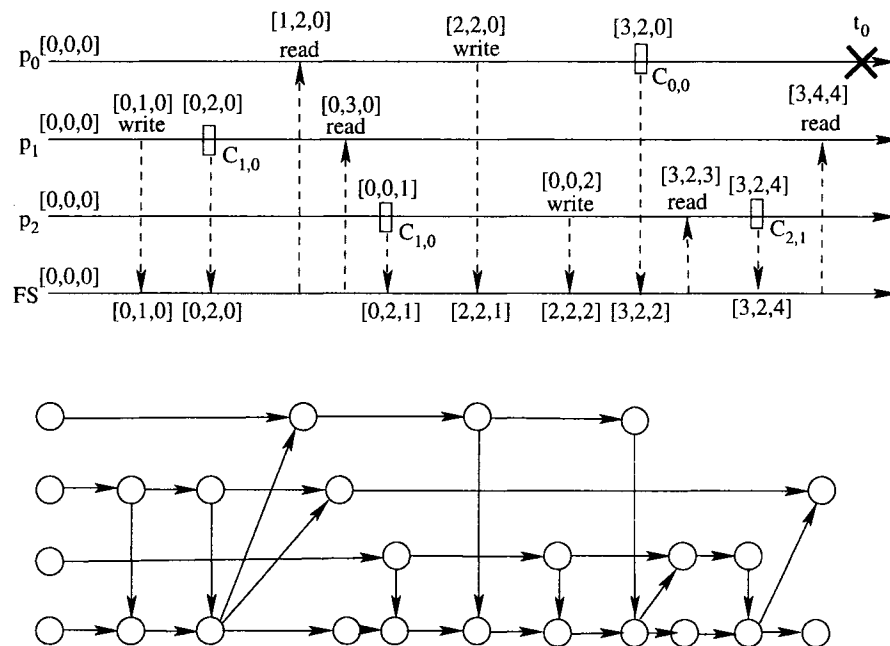


Figure 4.5: The Propagation of Vector Time in a Three Process System with a Shared File System

At time t_0 , process p_0 crashes and is restarted from checkpoint $C_{0,0}$. The process is rolled back to its state at vector time $[3, 2, 0]$, while the file system is rolled back to vector time $[3, 2, 2]$. See figure 4.6.

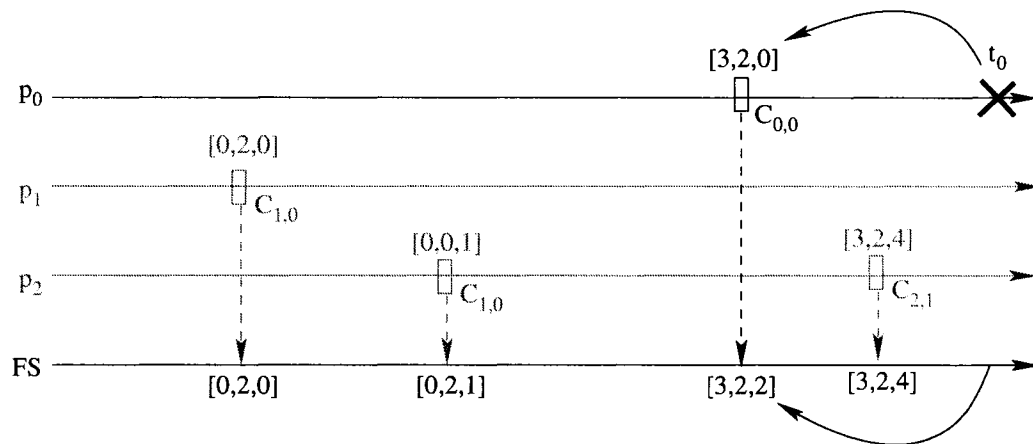


Figure 4.6: Rollback of a Failed Process

Since both process p_1 and p_2 share the file system with the failed process, we must restart these processes from checkpoints which causally precede checkpoint $C_{0,0}$. P_1 is restarted from $C_{1,0}$ and p_2 is restarted from $C_{2,0}$. See figure 4.7.

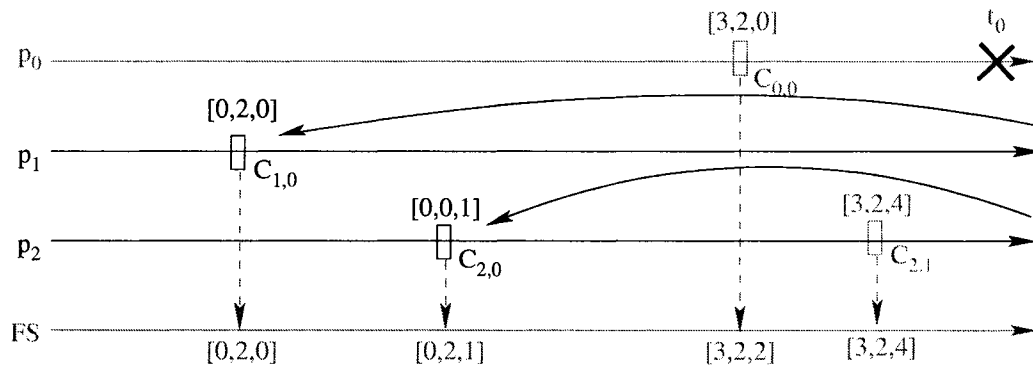


Figure 4.7: Rollback of Non-Failed Processes

Both processes now roll forward, driven by their logs. The rollforward stops when the vector time of each process reaches a point which indicates that it is not before (in the Lamport sense) the vector time of the file system. For p_1 , this occurs when its vector time reaches $[3, 4, 4]$. For p_2 , this occurs with the read at $[3, 2, 3]$. At this point, all the processes

are mutually consistent with each other and the file system. See figure 4.8.

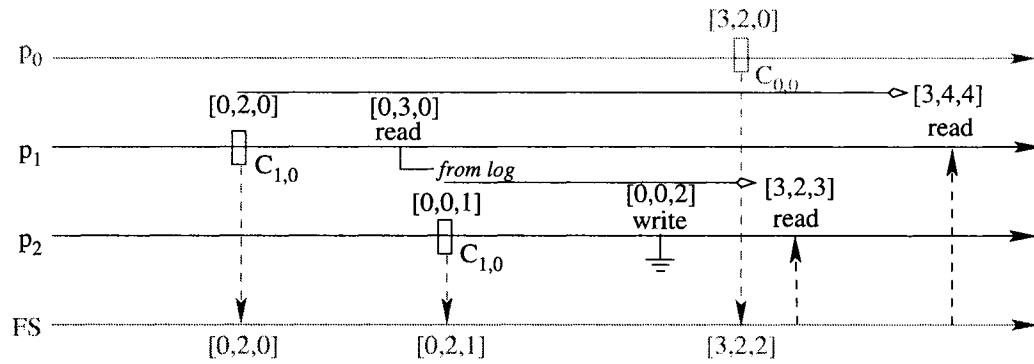


Figure 4.8: Roll Forward of Non-Failed Processes

In the next chapter we discuss a prototype implementation that demonstrates the feasibility of the scheme. In chapter 6, we provide a proof that our protocol does indeed allow the system to recover from a process failure, and that it does not suffer from the domino effect.

Chapter 5

Implementation

When we mean to build,

We first survey the plot, then draw the model...

Shakespeare, *King Henry IV, Part II*, Act I, Scene 3

In this chapter we discuss our prototype implementation of the system which demonstrates the practicality of our scheme. The implementation is divided into two parts: the file system layer and the process control layer. See figure 5.1.

We begin the chapter with a discussion of our design goals and an overview of the prototype. We discuss the API that our prototype presents to application programs, and then discuss the implementation of the two layers of the system. In section 5.8.5 we give an example of how an application would use our system. We conclude the chapter with a discussion of POSIX thread synchronization primitives, and the special difficulties they present to applications using checkpointing and rollback-recovery for fault tolerance.

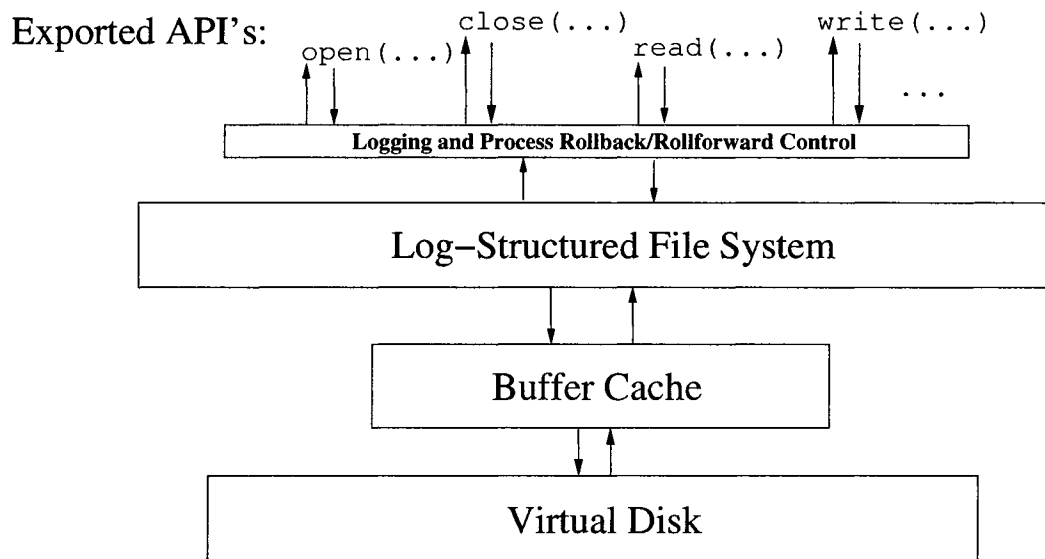


Figure 5.1: The Layers of the Prototype

5.1 Design Goals and System Operation

Since our intention was to build a prototype, our design goals were different than those of an implementor building a production system. Our primary focus was on correct operation and a full set of features rather than efficiency. Nevertheless, some design decisions were affected by questions of algorithmic efficiency. When designing a file system, many data items will have two separate representations: in-memory and on-disk. Being able to efficiently convert between these two representations makes development easier and contributes to the efficiency of the implementation.

Our implementation is built using the POSIX thread library supplied with the 2.4 version of the Linux kernel, version 2.2 of the GNU C standard library (libc) and version 0.9 of the GNU pthread library. This thread implementation was originally designed in 1996. Because of it was designed in conjunction with the early 2.x versions of the Linux kernel,

the implementation suffered from some non-POSIX compliant issues. From our perspective, the most glaring of these was the implementation of signal handling by the kernel and library. Drepper describes the implementation as “non-compliant and fragile” [15].

In the original implementation of the threads library, each thread is assigned to a separate process. An extra manager process coordinates activities among the threads.

The POSIX standard requires threads to be able to share almost all resources. This was accomplished in Linux by the use of the non-standard `clone()` syscall. `Clone()` is similar to the standard Unix `fork()` syscall which is used to create new processes. While `fork()` makes a copy of the calling process, `clone()`'d processes share almost all of their address space with the parent process. The exception's are the kernel and user stack. While this makes sharing resources easy and efficient, it leads to problems when using signals.

In particular, the result of sending a signal to a single thread is not well-defined. In the traditional Unix programming model, the `kill()` syscall is used by a process to send a signal to another process. The sending process specifies a process ID and a signal number. Upon signal receipt, the receiving process can either choose to ignore the signal, catch the signal and perform some action, or terminate.

The original implementation of signals in the thread library perverted their traditional semantics in several ways. Some types of signals behaved in the expected ways. For example, a `SIGALRM`, the alarm clock signal, behaves according to its traditional meaning. Other signals, however, may either be lost completely (`SIGINT`, the keyboard interrupt signal), or may be delivered to every thread in the group (for example, `SIGCONT`.) For example, a `SIGTERM` kills every thread in the group rather than just the thread to which it was sent.

In 2002, the open source development community introduced a new threading model

in Linux. This threading model is available on systems running the 2.4.20 or later kernel versions along with version 2.3 of GNU libc and version 0.10 of the GNU pthreads library.

Our prototype system was implemented using the original thread library and consists of approximately 8,100 lines of C++ code. Another 7,900 lines of code was written for the purposes of debugging, testing and evaluating the system.

5.2 API

The API of a system is the set of library and system calls (syscalls) available to programmers who develop applications for that system. From a syntactic standpoint, there is no difference between syscalls and library calls, so we refer to the elements of our API as syscalls. The syscalls exported by our prototype include eight traditional Unix syscalls, one syscalls which has modified syntax and semantics, and three new syscalls.

`int mkvlf()` Create a log-structured file system on a virtual disk.

`int mount()` Mount a file system.

`int unmount()` Unmount a file system.

`int open(int fd)` Open a file and return a file handle for use in subsequent operations.

`int close(int fd)` Deallocate a file handle.

`int read(int fd, char *buffer, long count, unsigned long offset)` Read from a file.

`int write(int fd, char *buffer, long count, unsigned long offset)` Write to a file.

`int unlink(int fd)` Remove a file from the file system namespace.

`int sync(vector_time *×tamp)` Snapshot the file system and commit file system changes to disk. Return a pointer to the timestamp associated with the snapshot to the calling program.

`int rollback(vector_time *×tamp)` Discard the current state of the file system and roll it back to the state which existed at time `timestamp`.

`void print_stats(char *switch)` Print statistics on file system activity.

`void noop()` Enter the file system without performing any file system activity. This purpose of this system call is explained in section 5.5.

Our implementation is written in C++, and the exported system calls and associated data types are available to application programs which link against the compiled code of our prototype. Below we explain the semantics of each syscall.

5.2.1 Mkvlf

`int mkvlf()`

The `mkvlf()` syscall formats the virtual disk (see section 5.3.1). It creates a superblock on the disk at block 0, and initializes on-disk copies of the checkpoint region and imap. It then allocates and initializes the on-disk copy of the block bitmap, which indicates free and used disk blocks.

In the applications we used to test and evaluate our prototype, the applications call `mkvlf()` once at the beginning of a computation. It is possible to format the file system in

one thread group, unmount the file system, and then mount it in another application some time in the future, since the file system is implemented on a virtual disk that lives as long as the associated physical files remain on the disk.

5.2.2 Mount

```
int mount()
```

The `mount()` syscall makes the file system accessible to user programs by opening the physical files associated with the virtual disk, and reading the superblock, checkpoint region and imap into memory.

5.2.3 Unmount

```
int unmount()
```

The `unmount()` syscall updates all the on-disk data structures from the copies in memory, and then deallocates the memory used by those data structures.

5.2.4 Open

```
int open (int file_number)
```

The `open()` syscall takes an integer argument specifying a file number and makes the file accessible to application programs. It does this by reading the associated inode and indirect blocks (if any) from disk. No file data is read from the disk by the `open()`. Existing data blocks are only read from disk when they are accessed by a subsequent `read()` or `write()` syscall. If the specified file does not exist, the `open()` creates an empty file.

The prototype supports a total of 64 files. We made this design decision because an imap entry is 16 bytes long and we wanted the imap to fit into a single disk block. See figure 5.5.

In the design of our prototype system, we decided not to implement directories. In Unix file systems, directories are for all practical purposes simply files which contain lists of (file or directory name, inode) pairs. We decided that supporting directories was not necessary to demonstrate the viability of our scheme.

5.2.5 Close

```
int close(int file_number)
```

The `close()` syscall deallocates a file handle previously allocated to an application by an `open()`. As in traditional Unix file systems, the `close()` essentially does nothing. However, if some previous application deleted the file using the `unlink()` syscall, and this application is the last to close the file, the inode and associated indirect and data blocks are marked as free and available for reuse in the imap and free list.

This unusual behavior is true to the semantics of traditional Unix file systems in that `unlink()`ing a file removes the file from the file system namespace, but the associated data and metadata are not deleted until the last application releases the file with a `close()`.

5.2.6 Read

```
int read (int file_number, char *buffer, unsigned long nbytes, unsigned long
offset)
```

The `read()` syscall copies the number of bytes specified by the `count` argument into

the buffer pointed to by argument `buffer`. The behavior of the `read()` is undefined if the buffer is not large enough to hold the number of requested bytes.

In traditional Unix file systems, a `read()` always begins at the position specified by the file's file pointer. The file pointer is an internal value specifying the offset from the beginning of the file where the next read or write will begin. We decided to omit the concept of the file pointer and require applications to explicitly specify it in each read or write syscall via the `offset` argument.

Each file has a file size parameter associated with it. When a file is first created, the file size is set to zero. An attempt to read beyond the end of the file does not fail, but returns only the number of bytes available. If such an attempt is made, the syscall returns the value `-EOF` to the calling application.

It is possible to create files with "holes" in them. This occurs, for example, if the first write to a file occurs at somewhere other than offset 0. Any subsequent attempt to read data that is before the end of the file but which has never been written returns null bytes for the unwritten data. This behavior is consistent with both the Linux and Solaris `read()` syscalls.

5.2.7 Write

```
int write (int file_number, char *buffer, unsigned long nbytes, unsigned long
offset)
```

The `write()` syscall transfers `count` bytes of data from `*buffer` to the file system beginning at `offset`.

5.2.8 Unlink

```
int unlink (int file_number)
```

The `unlink()` syscall removes the file from the file system namespace. If no applications have the file open, the inode is marked as unallocated, and the disk blocks containing the associated metadata and data are returned to the free list.

5.2.9 Sync

```
int sync (vector_time *&timestamp)
```

The `sync()` syscall is used to commit to the disk any file system changes which occurred since the file system was mounted or last `sync()`ed to disk. It returns the vector time of the file system to the calling processes.

The `sync()` syscall is similar to the traditional Unix `sync()` call, but performs some additional steps to ensure that the file system can be rolled back to its current state at some point in the future if necessary.

These additional steps commit the imap to disk at a new location. The imap location is stored in the checkpoint region along with the vector time of the file system. The checkpoint region itself is then written to disk. A new checkpoint region is set up with pointers to the list of old checkpoint regions. Finally, the old checkpoint region's disk location is stored in the superblock, and the superblock is written to disk. See figures 3.4 and 3.5.

The `sync()` operation is fundamental to the correct operation of an LSFS. Pseudocode for the `sync()` syscall is shown in appendix A.

5.2.10 Rollback

```
int rollback (vector_time *timestamp)
```

The `rollback()` syscall is used to roll back the file system to an earlier state. It takes a single argument, which is a pointer to a `vector_time` variable containing the timestamp of some earlier `sync()` of the file system.

On the initiation of a rollback, the file system waits for any pending I/O operations to complete, and then locks itself to prevent any thread from accessing its internal data structures. It then scans the list of checkpoint regions looking for one with a matching timestamp. If no such timestamp is found, the system unlocks itself and returns an error.

Once a checkpoint region with a matching timestamp is found, the current state of the file system is discarded. The matching checkpoint region is read into memory along with the associated inode map. At this point the syscall returns.

One unusual characteristic of the rollback operation is that it does not re-open any files which were open at the time of the `sync()` which created the file system checkpoint. Threads in the system store a list of the files they have open when they checkpoint. Once individual threads have rolled back, the thread instantiation process re-opens the file.

5.2.11 Print_stats

```
void print_stats (char *switch)
```

The `print_stats()` syscall is used to gather statistics on the behavior of the underlying file system. It takes a string of characters chosen from the set `[bcCdisv]`. These characters act like switches, and control which sets of statistics are printed. A null string passed as an argument defaults to printing all available statistics.

The individual switches control the following sets of statistics:

- b, the block bitmap:** number of allocated and free blocks on the virtual disk.
- c, the checkpoint region:** total number of checkpoint regions on the disk and the timestamp of each region.
- C, the disk cache:** total number of used and free buffers in the disk cache, the number of sweeps of the buffer replacement clock hand, the number of disk blocks read and written to the cache, and the number of cache blocks filled and flushed
- d, the virtual disk:** number of blocks and bytes read from and written to the disk.
- i, the imap:** number of times the imap has been read from or written to disk, and the total number of inode to physical disk block translations that have occurred.
- s, information about the superblock:** number of times the superblock has been read and written to disk.
- v, general information about the file system:** number of times this file system has been mounted and unmounted, the number of file `open()`'s, `close()`'s and `unlink()`'s performed since the last mount, and the total number of times this file system has been `sync()`ed and rolled back.

In the next section, we describe the data structures and algorithms that comprise the file system layer.

5.3 The File System Layer

In this section, we discuss the various subsystems which make up the file system layer of our prototype system.

5.3.1 The Virtual Disk

The file system layer of our prototype is implemented on top of the actual Linux file system. At the bottom of the file system layer is a “virtual disk”. This virtual disk can be configured to contain a maximum of 2^{24} blocks. The block size of the disk can be modified at compile time and supports block sizes which are a power of two and range from 512 to 8192 bytes. With a block size of 1K (the default size we used in all of our experimental work) this gives a maximum file system size of 16GB.

While the Ext3 file system (which is the default file system for many Linux distributions) supports files of up to 1TB in size, support for these large files is not enabled by default; it must be explicitly requested when the kernel is compiled. Most Linux distributions provide precompiled kernels which default to a maximum file size of $2^{31} - 1$ bytes of addressable file space.

To support 2^{34} bytes of addressable file system space on top of these files, we implemented a system of segmented physical files to contain the data in our logical file system. The logical disk layer translates block addresses to (file, offset) pairs. See figure 5.2.

Since the main job of the logical disk is to satisfy block read and write requests from higher levels of the file system, we chose to use 16 1GB files to implement our default virtual disk. These files are created when an object of type `vdisk` is created by an application

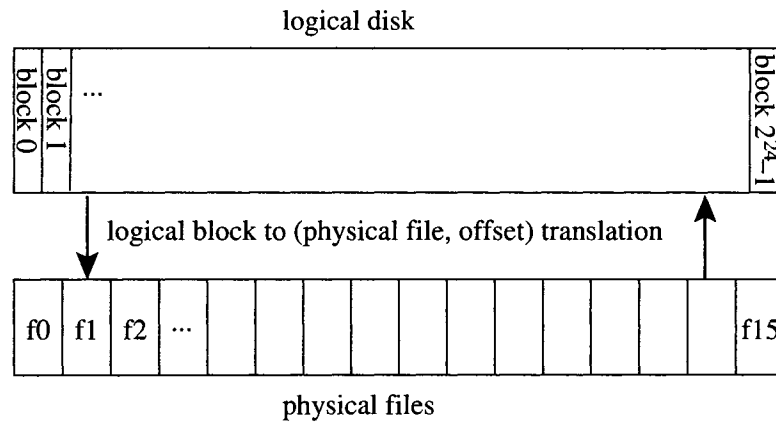


Figure 5.2: A Logical Disk Built on Top of Physical Files

program. We chose to use files of 1GB in size because this makes translation from virtual disk block read and write requests to physical file system read and write requests simple and fast.

With a 16GB virtual disk, the total number of blocks of size 2^S bytes is $\frac{2^{34}}{2^S} = 2^{34-S}$, and each file contains $\frac{2^{34-S}}{16} = 2^{30-S}$ blocks. The upper $30 - S$ bits of each 32 bit block address gives the physical file number, while the lower S bits give the offset within the file.

For example, suppose a request to read block N is made to a virtual disk with block size 2^S . The virtual disk satisfies this request by reading from file $N \gg (30 - S)$ at offset $N \% 2^{30-S}$. Since the block size 2^S is specified at compile time, the compiler can translate these operations into a bit shift and a bitwise *and* operation, respectively.

5.3.2 The Disk Cache

Our caching scheme for the prototype is simple. We employ a fixed size cache which uses an LRU replacement strategy. The size of the cache can be modified at compile time.

5.3.3 Vector Time

Because vector timestamps are one of the most frequently manipulated objects in our prototype, we evaluated several different implementations. We finally settled on a dynamically allocated list of (thread id, time) pairs to represent a single vector timestamp. The pairs in the list are kept sorted using the thread id as the key. An additional `int` associated with each timestamp contains the number of pairs in the list. Figure 5.3 shows an example vector time object. The object contains a timestamp describing three threads. Thread 10000's time is 10, thread 10020's time is 0, and thread 10300's time is 23.

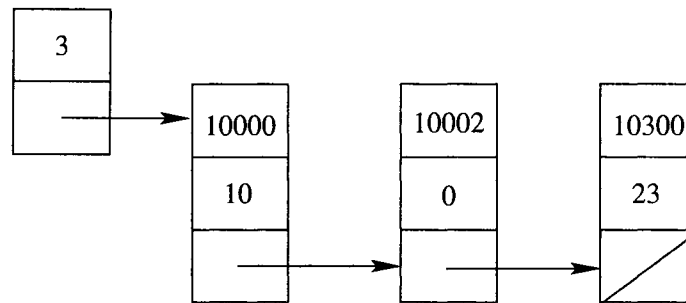


Figure 5.3: The In-Memory Representation of an Example Vector Time Object

When we write a vector time object to disk, we want to be able to store it in a single disk block. The vector time of a single process needs 12 bytes (8 for the process id and 4 for the time). This limits the number of (process id, time) pairs which can be written to a single block to 85. Thus, 85 is the upper limit on the number of processes the system will support.

5.3.4 The Inode

The inode class is the primary ADT of the file system. Figure 5.4 shows the on-disk structure of an inode.

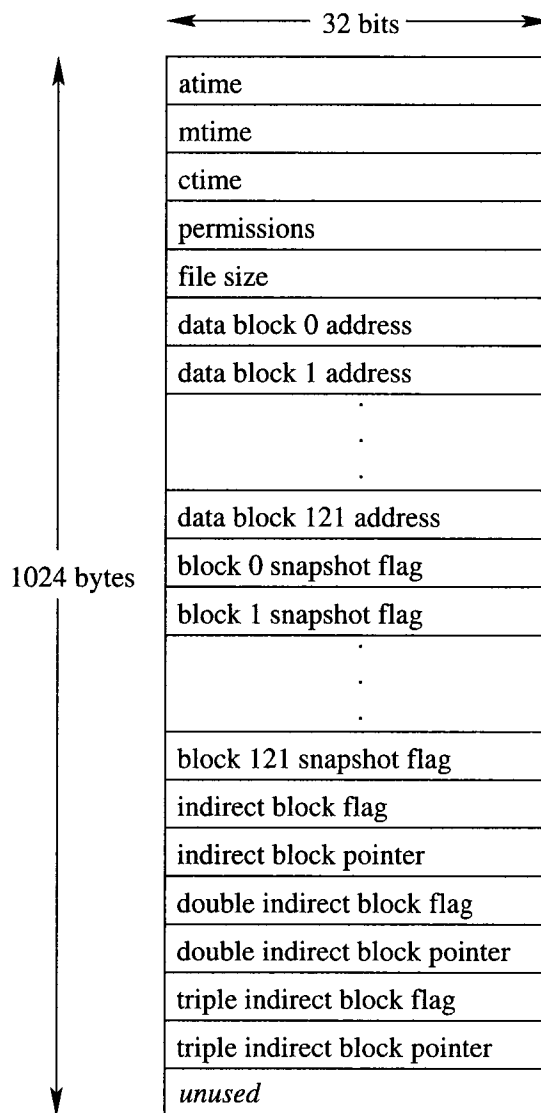


Figure 5.4: The On-Disk Representation of an Inode

The inode uses 20 bytes of metadata to store the access, modification and creation times, the file permissions and the file size. The next 968 bytes contain pointers to the first 122

blocks of file data. Associated with each block of file data is a “snapshot flag”. This flag indicates whether the corresponding data block on disk is part of a filesystem snapshot. This information is used when a block is evicted from the disk cache and when a file system snapshot is taken, since both of these operations force a block to disk.

We use a triple indirect block scheme (see figure 3.3) to expand the maximum addressable file size. There are 122 blocks accessible immediately from addresses stored in the inode. The first indirect block contains the addresses of an additional 128 blocks. The double indirect block contains the addresses of 128 indirect blocks, giving access to $128^2 = 16384$ disk blocks. Finally, the triple indirect blocks contains the addresses of 128 double indirect blocks, which in turn contain the addresses of 16384 indirect blocks. Thus, the triple indirect block gives access to an additional $128 \cdot 16384 = 2,097,152$ data blocks. Thus, the total number of addressable blocks is 2,113,786. Assuming the default block size of 1K, this gives us a maximum file size of 2,164,516,864 bytes, or approximately 2.02 GB. Table 5.1 shows the block and byte ranges encompassed by each level of indirection.

accessible via	# of blocks	block range	byte range
inode	122	0 - 121	0 - 124927
indirect block	128	122 - 249	124,928 - 255,999
double indirect block	16384	250 - 16633	256,000 - 17,033,215
triple indirect block	2,097,152	16634 - 2,113,785	17,033,216 - 2,164,516,863

Table 5.1: Block and Byte Ranges Accessible Via Inode and Indirect Blocks With A Default Block Size of 1K.

5.3.5 The Imap

The imap is a table of 64 (flag, block number) pairs. Entry `imap[i].flag` indicates whether inode i exists. If so, entry `imap[i].block` gives the physical disk block containing the associated inode. The schematic of a disk block containing an imap is shown in figure 5.5.

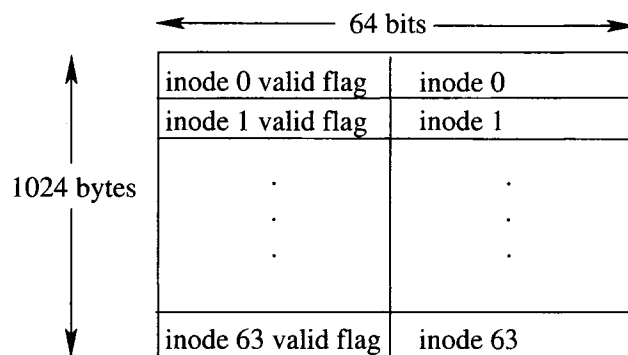


Figure 5.5: The On-Disk Representation of the Imap

5.3.6 The Checkpoint Region

The checkpoint region contains four pieces of information: a pointer to the disk block containing the imap, a pointer to the disk block containing the vector time of the checkpoint, a pointer to the previous checkpoint region's location on disk, and a list of pointers to the tree structure indicating the location of the freelist. See figure 5.6.

5.3.7 The Superblock

The superblock contains two pieces of information: the time of the file system was `mount()`, and the location of the first checkpoint in the checkpoint region list. See figure 5.7.

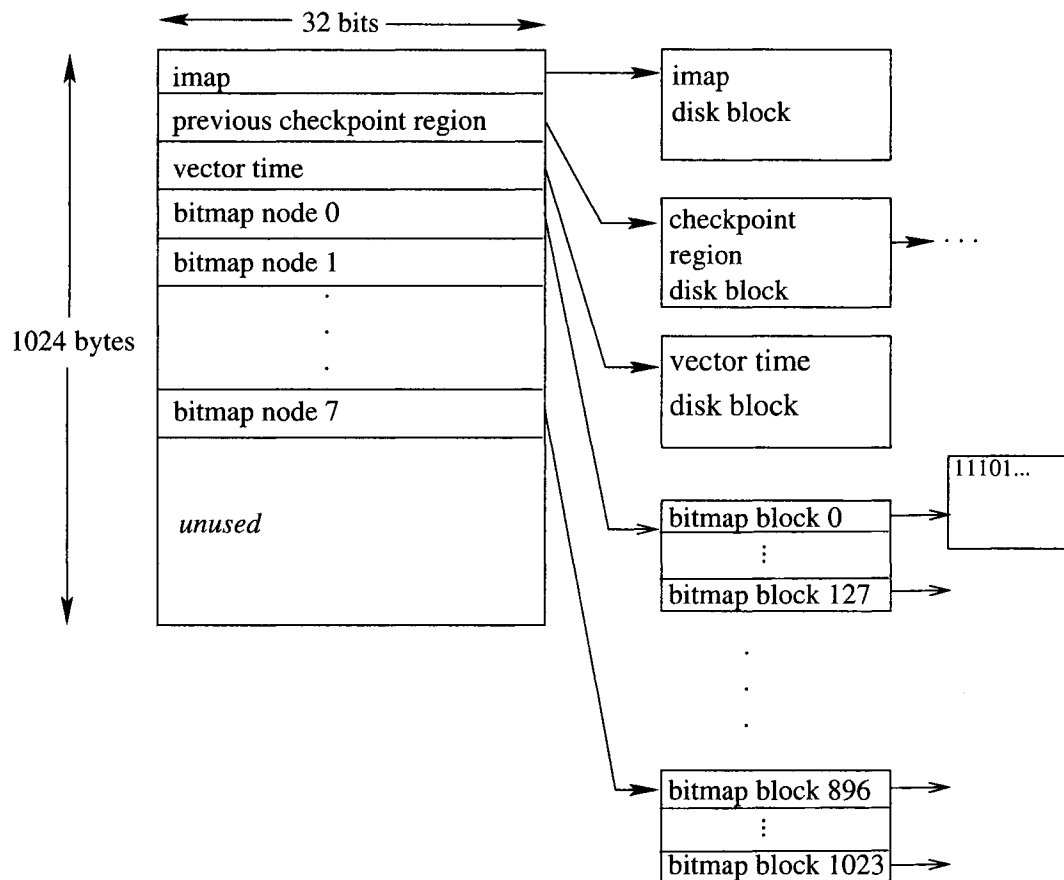


Figure 5.6: The On-Disk Representation of the Checkpoint Region and its Associated Structures

5.4 The Syscall Sequence

When a thread makes a syscall, the first step is always to lock the file system. All the structures of the file system are protected by this single lock. The main deficiency of using such course grained locking is that it limits the amount of parallelism among the threads. Nevertheless, it provides several advantages for a prototype system: It reduces the number of locks which must be managed, and eliminates the possibility of deadlock within the file system code which could occur if multiple locks are taken out of order.

Once a thread obtains the file system lock, it performs several housekeeping chores. It

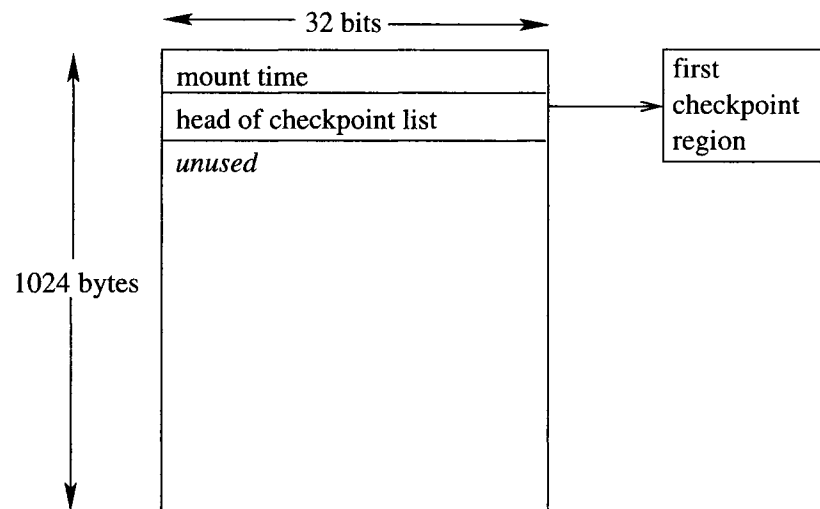


Figure 5.7: The On-Disk Representation of the Superblock.

first looks up the calling thread in the system area to obtain access to the thread's global data. It then checks a series of thread-specific flags. The most important of these flags is the `system_pausing` flag, which indicates that some thread in the system has failed, and that this thread needs to take the appropriate measures to restart from an earlier checkpoint. We discuss these measures in detail in section 5.7.1.

The thread next checks the type of syscall requested and updates either its own vector time or the vector time of the file system according to the rules given in table 4.1. After the appropriate vector time update has been performed, the actual syscall is performed. The thread then releases the file system lock and returns.

5.5 File System Rollback

Once the system detects that a process has failed, it must halt all the other threads in the system in some known state. Because of the difficulty in using signals to achieve this

goal, we opted to use an asynchronous notification system. This asynchronous notification is accomplished by setting a flag in each thread's local memory. This flag is inspected whenever a thread makes a syscall.

One of the tasks a thread performs when it enters the file system is to inspect its `system_pausing` flag. If this flag is set, the thread sends a message to the system indicating that it has paused. It then releases the file system lock and waits for the next phase of the system restart. We discuss this process in detail in section 5.7.2.

Threads can only pause during a syscall, and the system must wait for all threads to pause before continuing with the system restart. Thus, a compute-bound thread could conceivably stall the system restart for a long time if it performs no file system operations. To prevent this from happening, application programmers can insert the `noop()` syscall in compute bound areas of their code.

Once all processes in the system have paused, the system restart can occur. The first step in the restart is to rollback the file system to the vector time contained in the last checkpoint taken by the failed process.

To perform the file system rollback, the system locates the last complete checkpoint taken by the failed thread. It extracts the timestamp of this checkpoint and then searches the list of file system checkpoint regions on disk until it finds one with a matching timestamp. (Recall that each process checkpoint is accompanied by a file system sync, and that the timestamp of this sync is recorded in the process checkpoint.)

Once the appropriate checkpoint region is found, we discard the current state of the file system. The correct checkpoint region is read from disk, and then the associated `imap` and the block bitmap are read. At this point, the file system state is exactly as it was when the

file system snapshot was taken.

5.6 The Process Control Layer

In this section we discuss the process control layer. This layer of the prototype handles communication between threads, and the thread checkpointing process, and it coordinates thread restart, rollback and rollforward.

5.6.1 Thread Checkpointing

Thread checkpointing is a relatively straightforward operation, but some complications arise due to the particular thread library we are using, and the fact that we are checkpointing from the context of the thread itself rather than from the operating system kernel.

A thread checkpoint must save three types of information: The contents of the thread-local variables, the instruction pointer, stack pointer and other CPU registers, and the location of the associated file system snapshot.

Because a userspace thread does not have a simple means of accessing its call stack, we restrict threads to checkpointing only in the main thread routine. This places some restrictions on the application programmer, who must divide the task into logical chunks, making certain that she will not want her application to take a checkpoint within some subroutine. But it simplifies the checkpointing process because we know a priori which static and dynamically allocated variables need to be copied at each checkpoint and can ignore the variables which reside on the call stack. We believe this restriction simplified the implementation of our prototype enough to make it justifiable.

The typical way of saving and restoring the instruction pointer in a userspace C program is through the use of the `setjmp` and `longjmp` C standard library calls. These functions will not work on threaded programs however, since the `jmp_buf` structure manipulated by these calls does not contain enough context to completely restore a Linux thread. In particular, pointers to the thread-global variables and thread manager processes are not stored.

The pthreads library, however, contains equivalent library calls designed especially for threads. These are the `getcontext` and `setcontext` calls. A call to `getcontext` copies vital thread information into a `ucontext_t` structure allocated by the program. The contents of the `ucontext_t` structure are shown in table 5.2.

Structure Component	Contents
<code>unsigned long uc_flags</code>	The x86 EFLAGS register
<code>ucontext_t *uc_link</code>	A pointer to another <code>ucontext_t</code> object; not used.
<code>stack_t uc_stack</code>	Stack pointers SP and BP.
<code>mcontext_t uc_mcontext</code>	General purpose and floating point registers.
<code>__sigset_t uc_sigmask</code>	An array of longs containing current signal masks. Not used.
<code>__fpregs_mem</code>	Floating point state registers. Not used.

Table 5.2: The `Ucontext_t` Structure for the X86 Architecture.

We want application processes to be able to take a checkpoint with a single function call. However, an actual `checkpoint()` function call implies that the checkpoint operation would be called in a subroutine context, which we wish to avoid for previously stated reasons. To get a function-like syntax without actually performing a function call, the checkpoint operation is done inline via a macro implemented with a C `#define`. The checkpoint macro is shown in appendix B.

The checkpoint operation itself works as follows:

1. The thread checks that it is not in failure recovery mode. If it is, the checkpoint is skipped and the macro terminates.
2. The thread takes the file system lock. This prevents other threads from modifying the file system while the checkpoint is in progress.
3. A file system snapshot is taken via the `sync()` call.
4. The thread checkpoint is set up. The thread copies its variables and a list of open files to the checkpoint area. The checkpoint itself is hooked up to the thread's checkpoint list in the thread global area.
5. The thread calls `getcontext()` to save CPU registers in the thread checkpoint.
6. The file system is unlocked, the newly created checkpoint is marked as valid, and the checkpoint macro terminates.

5.7 System Recovery After a Failure

When a thread failure occurs, the system goes into recovery mode. In recovery mode, the system executes the recovery protocol to bring the system into a state which is consistent with the system state which existed when the failed thread took its last checkpoint. The recovery protocol consists of 7 steps.

1. Bring the non-failed threads in the system to some known, quiescent state.
2. Select an appropriate checkpoint from which to restart the failed thread.

3. Rollback the file system to the state which existed when the checkpoint selected in step 2 was created.
4. Set the appropriate system flags to indicate to threads that recovery is under way.
5. Recreate the failed thread and give it access to the information it needs to recreate the state which existed when the checkpoint from step 2 was created.
6. Rollback the non-failed threads and give them access to the information they need to recreate some state which causally precedes the recovered state of the failed thread.
7. Rollforward the non-failed threads, processing file system operations from the log rather than the disk. The rollforward phase of a particular thread stops when either the log is exhausted, or the log-operation's timestamp indicates that an orphan file system operation is about to be replayed.

Pseudocode for performing the above steps is shown below

```
// A failed thread has been detected. We pass to the system
// recovery code the thread and process ids of the failed threads
// via the pid_thread argument, along with a pointer to the
// function that the failed thread was executing via the kf
// argument.
//
// The function returns the thread id of the new instantiation
// of the failed thread.
pthread_t vlfs::system_restart (pid_thread_t pid_thread,
```



```

                                void * (*kf)(void *)
{
    int ppos;

    thread_checkpoint *last_checkpoint;

    pthread_t restarted_thread_t;

    // Get access to the thread group global data for the failed
    // thread.

    ppos = lookup_process (pid_thread.pid)

    // Access the last valid checkpoint created by the failed
    // thread.

    struct process_checkpoint *last_checkpoint =

                                psupport[ppos]->extract_ckp_list_head();

    // Set a flag telling other threads to pause at the next syscall.

    system_pausing = 1;

    // Wait for the non-failed threads to pause.  Each thread sends
    // the special value RECOVERY_PAUSE after it has noticed that
    // a restart is in progress.

    for (i = 0; i < n_threads; i++) {

        // Skip the failed thread.

```

```
    if (i == ppos)
        continue;
    recv (get_recovery_sock(i), &answer);
}

// At this point, the system is quiescent.

// Rollback the file system
rollback (last_checkpoint->vtime);

// Create the information necessary for the failed thread to
// restart. The resurrected thread needs to know that it
// is indeed resurrected, what its idea of the vector time
// should be, and the context from which it should execute.
startup_info *si = new startup_info;
build_startup_info (si, RESTART, ppos, last_checkpoint->vtime,
                   last_checkpoint->open_file_list,
                   last_checkpoint->vars,
                   last_checkpoint->context);

// Create new thread to replace failed one.
pthread_create (&restarted_thread_t, NULL, kf,
               (void *) si));
```

```

// Initiate rollback of other threads.
for (i = 0; i < n_threads, i++) {
    // Skip failed thread.
    if (i == ppos)
        continue;
    send (get_recovery_sock(i), RECOVERY_CONTINUE);
}

// Failed thread is restarted and non-failed threads are
// rolling back. Return id of restarted thread.
return restarted_process_thread_t;
}

```

5.7.1 Thread Restart

When a thread fails, we need to create another thread and restore the thread context from the appropriate checkpoint. Before beginning execution, each thread tests its local “restart_flag” variable which indicates whether the thread is restarting. This flag is only set for a failed thread which has been recreated via `pthread_create()`.

If this variable is set, the function `process_restart()` is called. This function sets up a new checkpoint list for the thread, restores the thread’s vector time from the checkpoint, reopens the set of open files saved in the checkpoint, and restores the thread’s variables to the values saved in the checkpoint. The thread then calls `setcontext()` to restore the

context saved in the checkpoint. The `setcontext()` call restores the threads stack and CPU registers to the state saved when the corresponding checkpoint was taken. When the thread returns from the `setcontext()` call, it begins executing in the place where the matching `getcontext()` call was made at the end of the `CHECKPOINT()` macro. The thread then continues execution exactly where it had after the `CHECKPOINT` macro was called.

Pseudocode for the thread restart operation is shown below.

```
// After a process fails, we create another thread
// executing the same function using the pthread_create()
// library call. If the thread received a RESTART flag as a
// parameter when it was created, this segment of code is
// executed.
//
// At this point, the checkpoint for restarting this thread
// has already been located, and the information necessary for
// restarting this thread from the checkpoint has been extracted
// and placed in the ‘‘thr’’ structure.
//
// Note that we are executing in the context of the restarted
// thread at this point.
int process_support::process_restart()
{
    uncontext_t *restored_context = new uncontext_t;
```

```
// A restarted thread resumes normal file system operations
// immediatly. This flag tells the thread to log its
// file system operations rather than replay them from the log.
mode = LOGGING;

// Restore the thread's local variables.  ``get_laddr()``
// returns the starting address of this threads locals.
restore_locals (get_laddr(), thr->ckp->vars, vsize);

// Reopen files the thread had open at the time this
// checkpoint was taken. The open file list is just
// a bitmap.
reopen_files (thr->ckp->open_file_list);

// Extract the thread's context. We must do this before
// restoring the CPU registers.
*restored_context = thr->ckp->checkpoint_address;

// Extract the CPU registers. This call restores all of the
// registers, EXCEPT the IP and SP, which are restored during
// the setcontext call below.
restore_regs (thr->ckp->gregs);
```

```

// Restore this thread's context.  It will emerge from this
// library call in the middle of the CHECKPOINT macro invocation
// which created the checkpoint.

setcontext (restored_context);

// If we reach this point, setcontext has gone haywire.

cout << "Setcontext returned during process_restart!" << endl;

return -1;
}

```

5.7.2 Thread Rollback and Rollforward

Non-failed threads are restarted in a similar manner. Instead of creating a new thread however, the system sets a flag in the each non-failed thread's local memory. This flag is checked whenever a flag makes a syscall. If the flag is set, the `process_rollback()` function is called by the thread. This function causes the thread to suspend normal computation. It then finds an appropriate checkpoint from which to restart, and restores its local variables, vector time, open files and CPU registers from the context saved in the checkpoint. The thread reemerges from the end of the `CHECKPOINT()` call which created the checkpoint, and precedes with the rollforward phase of the recovery.

Pseudocode for this operation is shown below.

```

// Rollback the process to the last checkpoint before
// rollback_start.

```

```

//

// We want to select the latest checkpoint which is before
// the rollback_start time. To do this, we scan the checkpoint
// list.

//

// If the current checkpoint doesn't have a successor, we use
// the current checkpoint.

//

// If the current checkpoint is concurrent with the
// rollback_start time, use the current checkpoint.

//

// If the current checkpoint's successor is before or
// concurrent with the rollback_start, make the successor the
// current checkpoint under inspection and start the loop again.

//

// Rf_start is a global vector_time object indicating the
// time of the rolled back file system.

int process_support::process_rollback ()
{
    struct process_checkpoint *candidate_ckp =
        (struct process_checkpoint *) NULL;

    struct process_checkpoint *cpptr;

    int found = 0;

```

```
// Find the latest checkpoint which was taken prior to the
// rollback time.

candidate_ckp = get_next_checkpoint();

while (!found)) {

    // No checkpoint! Error.
    if (!candidate_ckp) {
        cout << "Error: No checkpoint to rollback "
            << "to in process_rollback!" << endl;
        return -1;
    }

    // Is candidate_ckp the last checkpoint taken?
    // If so, restart from candidate_ckp.
    cpptr = get_next_checkpoint();
    if (!cpptr) {
        found = 1;
        continue;
    }
}
```



```
// Is candidate_ckp concurrent with the rf_start?
if (candidate_ckp->vtime->concurrent (rf_start)) {
    found = 1;
    continue;
}

// We've checked all the corner cases. Now, check to see if
// candidate_ckp is the latest checkpoint before rf_start.
if (candidate_ckp->vtime->before (rf_start)) {
    found = 1;
    continue;
}

// Candidate_ckp is not the correct one for restarting
// this thread. Check the next one.
delete_checkpoint (candidate_ckp);
candidate_ckp = cpptr;
}

// Setup a 'thr' structure for this thread to rollback to.
extract_thread_context (&thr->ckp, candidate_ckp;

// Allocate a context to rollback to.
```

```
ucontext_t *restored_context = new ucontext_t;

// Clear out the log up to but not including the first
// operation performed after the checkpoint.
log->clear_until (candidate_ckp->get_vtime());

// This thread must replay FS operations from its log.
mode = REPLAY;

// Restore local variables.
restore_locals (get_ladder(), thr->ckp->vars, vsize);

// Reopen files
reopen_files (thr->ckp->open_file_list);

// Extract the thread's context.
*restored_context = thr->ckp->

// Restore general purpose registers
restore_regs (thr->ckp->gregs);

// Rollback. The thread will emerge in from this library call
// in the middle of the CHECKPOINT macro invocation which
```

```

// created the checkpoint.

setcontext (head->checkpoint_address);

// Not reached unless setcontext barfed.

cout << "Setcontext returned during process_rollback!"
      << endl;

return -1;
}

```

Once a rolled back thread emerges from the CHECKPOINT macro, it rolls forward by replaying file system operations from its log. Each time a thread that is rolling forward makes a syscall, its vector time is compared against the vector time to which the file system was rolled back. If the thread's current time is before the time of the rolled back file system, the actual file system operation is skipped. Instead, the data generated by the call is extracted from the thread's log. The vector time of the thread continues to update according to the rules given in table 4.1.

Once a thread makes a syscall which indicates that its notion of time is either concurrent with or after the file system, or its log is depleted, the thread resumes normal operation.

Below we show pseudocode for the `open()` system call where a particular thread determines if it should actually perform the system call, or read the call from its log. Similar code to extract the next operation from the log is found at the beginning of every syscall.

```

// Current_ps is a pointer to thread global data for the executing

```

```
// thread.

int vlfs::open (long inode_num)
{
    ...

    // Are we replaying the log?
    if (current_ps->mode == REPLAY) {

        // Check the timestamp on the next log operation.  If this
        // is an orphan FS operation, replay is over.  Switch to
        // normal operation.

        log_op = current_ps->retrieve_next_op (&syscall_return,
                                                &inode_num);

        // Have we exhausted the log or rolled forward far enough?
        //
        // Check_mode_change compares the vector time of this logged
        // operation to the vector time towards which we a rolling
        // forward.

        if (!log_op || current_ps->check_mode_change(log_op)) {
            current_ps->mode = LOGGING;

            break;
        }
    }
}
```

```

// This message is not an orphan. Return the logged result.

else {

    // Delete the entry from the log

    delete_log_tail();

    return syscall_return;

}

}

// Not replaying from log. Continue normally.

...

}

```

5.8 Application Rules for Using the System

Applications which use our prototype are not prohibited from any type of operation found in a normal C++ program. However, they must add some additional machinery to take checkpoints and ensure that they are able to restart or rollback correctly. In this section, we describe those additional requirements.

5.8.1 Restart Check

Before beginning execution, each thread checks its `restart` flag. This flag indicates whether the thread is simply beginning its execution, or is restarting. See section 5.7.1. The flag is tested by a call to `RESTORE_REGS`, a macro. This macro in turn calls `process_restart()` if appropriate. If the flag is not set, threads continue with the normal startup.

5.8.2 Thread Registration

The second step in thread startup is for threads to inform the system that they have joined the computation. They do this by calling the `process_register` function. This function registers the thread with the system, sets up a global checkpoint list for the thread, and informs other threads that a new process has joined the group so they can allocate space for the thread in their vector timestamps.

5.8.3 Checkpointing

Threads checkpoint by calling the `CHECKPOINT()` macro. The checkpoint macro is listed in appendix B.

5.8.4 Thread Deregistration

When a thread finishes its execution, we do not allow it to terminate. The reason for this is that some other thread in the group may still fail, and the finished thread will have to be rolled back.

When threads finish execution they call the `process_deregister()` function. This function counts the number of threads which have called it, and pauses a thread if there are still others executing. Once each thread in the system has deregistered itself, the threads are released and terminate.

5.8.5 An Application Program Skeleton

To show how a typical program uses our system, we present the following skeletal outline, with calls to the support code inserted into the correct places in the application.

```
void *some_thread (void *args)
{
    // Check the global restart flag.  Restore registers and
    // context, if this thread is restarting.

    RESTORE_REGS;

    ...

    // Register the thread with the system.
    system->process_register(void *global_variables, ...);

    ...

    // Normal computation begins here

    ...

    // Take a checkpoint
    CHECKPOINT(system);

    ...

    CHECKPOINT(system);

    ...

    // Thread has completed
    system->process_deregister()
}
```

5.9 POSIX Thread Synchronization

When threads are cooperating to achieve some computational goal, they need some way of communicating and coordinating their actions. Communication is most commonly done by message passing. Coordination can also be done via message passing, but because threads share memory, POSIX thread libraries provide simpler and more efficient ways of coordinating.

The two simplest thread coordination primitives are *mutexes* and *condition variables* [9]. In this section, we discuss the use of these two primitives, and how they are dealt with during thread rollback.

5.9.1 Mutexes

The most basic synchronization problem between computational processes which share memory is ensuring that all access to shared memory is mutually exclusive. That is, we must be able to guarantee that no thread attempts to write to a memory location while another thread is reading or writing that memory location.

POSIX provides objects of type `mutex` that can be used to guarantee exclusive access to one or more memory locations. Protected memory locations may be as small as a single bit, or may consist of more complex data structures such as trees or graphs. In our implementation, we treat the in-memory data structures which describe the file system as a single object. The amount of data protected by a single mutex is referred to as the “granularity” of the mutex.

To enable a program to use mutexes, programmers include the `pthread.h` header in

their programs and then link against the pthread library. Linking against this library gives programmers access to the data type `pthread_mutex_t` and the library calls which implement the basic operations used to manipulate mutexes. These 5 basic operations are creation, initialization, deletion, locking, and unlocking. These operations are performed by the following declarations and library calls:

Creation A mutex is created by declaring an object of type `pthread_mutex_t`.

Initialization Once a mutex has been created, it is initialized with a call to the library

function `pthread_mutex_init`.

```
int pthread_mutex_init (pthread_mutex_t *mutex,
pthread_mutexattr_t *attr);
```

The `pthread_mutexattr_t` argument defines the characteristics of the mutex. In the version of the Linux kernel and pthread libraries we use in our prototype, the only attributes available are the NULL attributes.

Deletion Mutexes are deleted by calling the destroy function `pthread_mutex_destroy`.

```
int pthread_mutex_destroy (pthread_mutex_t *mutex);
```

Locking In order to gain exclusive access to the memory protected by a mutex, the programmer must first lock the mutex associated with the shared memory. Locking is done with the `pthread_mutex_lock` call.

```
int pthread_mutex_lock (pthread_mutex_t *mutex);
```

If a thread attempts to lock a mutex that is already locked by another thread, the thread sleeps on the mutex, waiting for it to be unlocked.

Unlocking Once a programmer has finished accessing the shared data protected by the mutex, the mutex should be unlocked to allow other threads to access the data.

Unlocking is accomplished with the `pthread_mutex_unlock` call.

```
int pthread_mutex_unlock (pthread_mutex_t *mutex);
```

If any threads are waiting to lock the mutex, exactly one is awoken and succeeds in locking the mutex.

5.9.2 Condition Variables

In some circumstances, coordinating threads may be interested in more than just exclusive access to shared data. They may also be interested in exchanging information about the data's state. For example, threads sharing a stack may be interested in knowing whether or not the stack is empty. If the stack is empty, a particular thread may have no work to do and will want to sleep until the stack is pushed.

A thread could access the stack by locking the mutex associated with a stack and checking the stack size. If the stack has data on it, it would pop the stack, release the mutex, and process the data. If, however, the stack is empty, the thread would release the mutex and return without modifying the stack.

Unfortunately, this type of behavior leads to a condition called busy-waiting. The thread must repeatedly check the stack, even if it is empty, to determine whether there is work to be done. This behavior wastes processor cycles, since the thread must enter the stack critical section and check the stack size, even if there is no work to do.

The solution to this busy-waiting problem is to use *condition variables*. Condition variables allow a thread to block until some predicate is satisfied. In our stack example, the

predicate of interest is *the stack is not empty*.

Associated with each condition variable are three subordinate objects: a mutex, a special variable of type `pthread_cond_t` and a predicate. In POSIX parlance, the predicate is called a *condition*, thus giving rise to the name “condition variable”. When a thread executes a condition variable test, it locks the mutex to gain exclusive access to the condition variable and associated data. It then tests the condition. If the condition is true, the thread modifies the protected structure as needed, unlocks the mutex, and then leaves the condition. When these operations occur, a condition variable has semantics very much like a mutex.

The difference occurs when the tested predicate is false. If a thread finds that the condition is not satisfied, it releases the mutex and goes to sleep on the predicate condition. To avoid race conditions, POSIX condition variables release the mutex and sleep in a single atomic operation.

Eventually, (in correctly written code), the condition protected by the condition variable becomes true. When this happens, the thread which caused the condition to become true uses a special library call to wake all the threads waiting on the condition.

There are six basic operations that can be performed on a condition variable: creation, initialization, entering the condition, exiting the condition, sleeping, and signaling a wakeup. These operations are performed by the following declarations and library calls.

Creation A condition variable is created by declaring an object of type `pthread_cond_t`.

Since a condition variable always has an associated mutex, the mutex is typically created at the same time as the condition variable¹.

¹Since condition variables always have a mutex associated with them, it is curious that the POSIX thread committee did not provide a way to declare both of these structures in a single statement. We suspect the

Initialization A condition variable is initialized by a call to `pthread_cond_init`.

```
int pthread_cond_init (pthread_cond_t *cond,
pthread_condattr_t *condattr)
```

The `pthread_condattr_t` argument defines the attributes of the condition variable. Again, as in mutexes, the only attributes available in our development environment was the `NULL` attribute. Typically, the associated mutex is initialized at this point as well.

Entering the Condition A condition is entered by locking the mutex associated with the condition.

Exiting the Condition Similarly, exiting the condition requires the program to unlock the mutex.

Sleeping If a thread needs to sleep because the predicate associated with the condition is not true, it calls `pthread_cond_wait`.

```
int pthread_cond_wait (pthread_cond_t *cond, pthread_mutex_t *mutex);
```

A call to this function puts the calling thread to sleep and, atomically, unlocks the mutex that was locked when the condition was entered.

Signaling a Condition When a condition variable's predicate becomes true, those threads sleeping on the condition are awoken by a call to either `pthread_cond_signal` or `pthread_cond_broadcast`.

```
int pthread_cond_signal (pthread_cond_t *cond);
```

reason for this omission was their desire to keep the standard as simple as possible by providing only one method to declare a mutex.

```
int pthread_cond_broadcast (pthread_cond_t *cond);
```

`Pthread_cond_signal` wakes a single thread that is sleeping on the condition, while `pthread_cond_broadcast` wakes all threads sleeping on the condition. The thread which caused the condition variable's predicate to become true is responsible for calling the appropriate signaling function.

As an example of the use of a condition variable, let us extend our stack example from above. We will protect our stack with a mutex variable called `stack_lock`. The predicate we are interested in is: "The stack is not empty." We associate the condition variable `stack_item_avail` with this predicate.

The first step in using our condition variable is to declare and initialize the appropriate condition variable and mutex.

```
// Declare, allocate and initialize condition variable, mutex,
// and stack
pthread_cond_t *stack_item_avail = new pthread_cond_t;
pthread_mutex_t *stack_lock = new pthread_mutex_t;
pthread_cond_init (stack_item_avail, NULL);
pthread_mutex_init (stack_lock, NULL);
stack_t s = new stack_t;
```

Some threads will wish to periodically check to see if the stack has any items on it. If so, they remove the item and process it. If not, they sleep until an item is available. Such a thread would execute the following segment of code.

```
// Lock the critical section
```

```
pthread_mutex_lock (stack_lock);

// Test the predicate.  If false, release the lock and sleep
// on the condition.
while (s.isempty())
    pthread_cond_wait (stack_item_avail, stack_lock);

// OK, we're awake and we hold the lock again.  Do some work.
item = s.pop();

// Release the lock and exit the critical section.
pthread_mutex_unlock (stack_lock);
```

The purpose of having the `pthread_cond_wait` in a loop is two-fold. First, some buggy POSIX thread implementations (including the original Linux implementation) generate spurious wakeups on condition variables. Second, it is more flexible, since those threads which generate wakeups do not need to track the number of threads waiting on the condition and so can use either `pthread_cond_signal` or `pthread_cond_broadcast`.

Threads which generate items to be placed on the stack would execute the following code.

```
pthread_mutex_lock (stack_lock);

s.push(item);
```

```
pthread_cond_broadcast (stack_item_avail);
```

```
pthread_mutex_unlock (stack_lock);
```

5.9.3 Thread Rollback with POSIX Synchronization Variables

When a thread fails, our recovery algorithm requires us to bring the system to a known quiescent state before the restart and rollback process can commence. We require threads to check if recovery is underway before they begin any syscall. If so, the thread pauses before the syscall commences.

However, the introduction of thread synchronization introduces additional difficulties. Suppose that a thread is waiting on a mutex lock that is held by a second thread that fails. The failed thread will never release the lock, and the waiting thread will never progress. Since the system must quiesce before recovery can begin, the non-failed, waiting thread will block the recovery process, resulting in a deadlock.

Similarly, several threads may be waiting on a condition variable. The death of the thread which was to signal the satisfaction of the predicate will prevent any of these threads from becoming quiescent, again resulting in a deadlock.

To deal with these issues, we developed a user-level synchronization classes which effectively serve as wrappers around the mutex and condition variable types. Rather than manipulating mutexes and condition variables directly, threads manipulate them through these classes. This allows the system to keep track of which threads use synchronization primitives, and to deal with the various deadlock issues during recovery. We call the two

data types provided by this class `ulevel_mutex_t` and `ulevel_cond_t`.

The class `ulevel_mutex_t` has the following definition:

```
class ulevel_mutex_t
{
private:
    pthread_mutex_t *lock_lock;

    pthread_mutex_t *the_lock;

    // The process id of the thread currently holding this lock.
    pid_t holder;

public:
    ulevel_mutex_t();

    ~ulevel_mutex_t();

    int take_lock();

    int release_lock();
}
```

The variable `the_lock` is the actual application level mutex used by the thread. The variable `lock_lock` is a mutex which protects the `ulevel_mutex_t` object itself. Instead of declaring objects of type `pthread_mutex_t`, threads declare a `ulevel_mutex_t`. This declaration allocates space for the lock and initializes it.

When a thread wishes to lock a mutex, it calls the `take_lock()` method. Similarly, the `release_lock()` method is used to unlock a mutex. Under normal circumstances, the se-

antics of both `take_lock()` and `release_lock()` are the same as `pthread_mutex_lock()` and `pthread_mutex_unlock()`, respectively.

However, during recovery mode, both methods perform additional work. Once a failure is detected, the failure recovery code checks to see if the failed thread was holding any mutexes. If so, the mutex is released. If any of the non-failed threads attempts to take a lock after a failure has been detected, the mutex lock fails. Instead, the thread is shunted into code to await a rollback message. Below we show pseudocode for the `take_lock()` and `release_lock()` operations.

```
ulevel_synchronize::take_lock()
{
    pthread_mutex_lock (*lock_lock);

    \\ System_restart is a thread global variable which indicates that
    \\ a failure has been detected.
    if (system_restart) {
        pthread_mutex_unlock (*lock_lock);

        \\ The noop() function is a syscall which allows a thread to check
        \\ the status, including whether a restart is in progress.
        noop();
    }

    else {
        holder = get_pid();
        pthread_mutex_lock (*the_lock);
        pthread_mutex_unlock (*lock_lock);
    }
}
```

```
    }  
  
    return 1;  
}  
  
ulevel_synchronize::release_lock()  
{  
    pthread_mutex_lock (*lock_lock);  
    pthread_mutex_unlock (*the_lock);  
    holder = -1;  
    pthread_mutex_unlock (*lock_lock);  
    return 1;  
}
```

The class `ulevel_cond_t` has the following definition:

```
class ulevel_cond_t  
{  
    private:  
        pthread_cond_t *the_cond;  
        pthread_mutex_t *the_mutex;  
        pthread_mutex_t *cond_lock;  
        int *condition;  
        int init_value;  
        int release_value;
```

```

pid_t waiting [MAX_THREADS];

// Condition is an integer value.  When the condition variable is
// initialized, its value is init_value.  When condition ==
// release_value, the predicate is considered to be true.
ulevel_cond_t (int *condition, int init_value, int release_value,
               pid_t holding);

~ulevel_cond_t ()

void enter_cond ();

void exit_cond();

void release_all();
}

void ulevel_cont_t::enter_cond()
{
    // Take lock and check to see if we are restarting.
    pthread_mutex_lock (the_mutex);

    if (system_restart) {
        pthread_mutex_unlock (the_mutex);

        noop();
    }

    // Not restarting.  Test the condition.

```

```

while (*condition != release_value) {

    // Predicate is not true. Record ourselves as waiting and
    // go to sleep.

    waiting[getpid()] = 1;

    pthread_cond_wait (the_cond, the_mutex);

}

void ulevel_cond_t::exit_cond()

{

    // We've done the work necessary. Remove ourself from the
    // list of waiters for this condition, drop the lock and
    // get on with life.

    waiting[getpid()] = 0;

    pthread_mutex_unlock (the_mutex);

}

```

Our code actually implements only those condition variables with predicates that have truth values determined by the comparison of two integer values. We have found, however, that most of the predicates we wish to write can, in fact, be expressed simply as the relationship between two integers. The purpose of the `init_value` variable in the private section of the class is to allow the system to reset condition variables to their default values after a system restart.

To use the `ulevel_cond_t` class, a thread group declares a variable of type `ulevel_cond_t`, and initializes it by passing the address of the integer variable that the condition is protect-

ing, the initial value of that variable, and the integer value that indicates that the condition is satisfied.

A condition can then be expressed using the following generic code.

```
ulevel_cond_t *some_cond;

...

some_cond->enter_cond();

// At this point, the predicate is satisfied.

// The thread accesses the critical section of the object protected
// by the condition variable.

some_cond->exit_cond();
```

When a thread fails, we set the global variable `system_restart` and then wake up all the threads waiting on the condition by executing `pthread_cond_broadcast (the_cond)` on all user level conditions.

The process control layer keeps a list of all the `ulevel_mutex_t` and `ulevel_cond_t` created by threads executing in the system. This list is updated during calls to the constructors and destructors of the respective classes.

Chapter 6

Formalisms

The proof ...

consists of little bubbles or beads

which appear on the surface ... after agitation.

E. Cobham Brewer, *Dictionary of Phrase and Fable*

The implementation presented in chapters 4 and 5 is an adaptation of the optimistic logging protocol [56]. In this chapter, we present theorems showing our scheme operates correctly and that it does not suffer from the domino effect.

We also discuss the concept of checkpoint log clearing and present an interesting result on a space-optimal method of clearing the log. We conclude the chapter by looking at how our system might be extended to deal with rare or unusual failure modes.

6.1 Concepts and Definitions

We supplement and extend the definitions of section 2.2.1 with the following vocabulary from the literature.

State Intervals A *state interval* is series of computations occurring between the execution of two file system operations. Given two file system operations $F(n)$ and $F(n + 1)$, state interval $I(n)$ is the series of operations occurring between the commencement of $F(n)$ and the commencement of $F(n + 1)$. $S(n)$ is the *process state* which occurs immediately before $F(n)$.

Repeatability We assume the behavior of each recovery unit is repeatable. That is, the state of a recovery unit can be restored by rolling a recovery unit back to a checkpoint and replaying file system operations from the log. Repeatability implies that given a process state $S(n - d)$ and file system operations $F(n - d)$ through $F(n - 1)$, we can restore state $S(n)$ by rolling back the process to state $S(n - d)$ and replaying the file system operations in order.

Orphaned File System Operations In optimistic recovery schemes, messages that are either lost because of an RU failure or are casually dependent on a lost state are referred to as orphan messages. In our scheme, we have the parallel concept of orphaned file system operations. A file system operation is an orphan if it occurred after the last checkpoint of the failed RU, or if some other RU is causally dependent on the file system operation. An simple example of an orphaned file system operation occurs when RU_i performs a file system write, RU_j subsequently reads some of the modified bytes from a file, and RU_i then fails.

Committable States If a state interval is lost due to a failure, or if it is casually dependent on a lost state interval, it is called an orphan state interval. State intervals that will never become orphans are called *committable states*.

Sequential Semantics A file system exhibits sequential semantics (sometimes referred to in the literature as Unix semantics) if the result of any file system operation is immediately visible to all other processes in the system. Our implementation uses sequential semantics.

6.2 Recovery After Failure

After a failure, the failed RU is restored from its latest checkpoint and the file system is restored from the snapshot taken during that checkpoint. The system then sends a recovery message to the other recovery units in the system informing them of the vector time of the restored file system. The failed RU then resumes computation.

When a non-failed RU receives a recovery message, it searches its list of checkpoints. As the RU scans the checkpoint list, it searches for the latest checkpoint that represents a committable state. That is, it is looking for the most recent checkpoint which is not causally dependent on the restored state of the failed RU.

This search is accomplished by comparing the vector times associated with each RU's checkpoints against the vector time of the recovered file system. Once an RU finds a checkpoint which is not causally dependent on the recovered file system, the RU's state is recovered from this checkpoint and roll forward commences.

The correctness of this protocol depends on two assumptions.

1. For each RU, we can always find a checkpoint which is not causally dependent on the state of the recovered file system.

2. Once recovery units have rolled forward, the entire system state, consisting of the rolled back file system, the restarted RU, and the rolled forward RUs, is consistent.

Theorem 6.1 *For each RU, we can always find a checkpoint which is not casually dependent on the latest checkpoint of any other RU.*

Proof: We require each RU to take a checkpoint before it performs any file system operations. Since the RU has not performed any file system operations, this checkpoint is not casually dependent on any checkpoint created by another RU. \square

Theorem 6.2 *After a failure, the restarted instantiation of the failed RU is consistent with the file system.*

Proof: During the checkpointing process, a recovery unit locks the file system, takes a checkpoint of the process state and then takes a snapshot of the file system. The checkpointing RU's vector time is updated from the timestamp taken from the file system. Since the file system was locked during the entire `sync()` operation, it can not have processed any file system operations generated by another recovery unit. Thus, there are no potentially orphan file system operations underway during the checkpoint.

When an RU fails, both the state of the RU and the state of the file system are rolled back to the state taken during the associated checkpoint. Thus the state of the restarted recovery unit and the file system are mutually consistent.

Theorem 6.3 *After restart of the failed process, rollback of the file system, and roll forward of the non-failed RUs, the system wide state is consistent.*

Proof: We need to show that each RU is pairwise consistent with all other RUs and the file system after recovery. We do this by showing

- A** The restarted RU is consistent with the recovered file system.
- B** Each non-failed RU is consistent with the restarted RU and consistent with the file system after roll forward.
- C** Each non-failed RU is pairwise consistent with every other non-failed RU.

A) By theorem 6.2, the state of the recovered RU is consistent with the state of the file system.

B) Now consider some non-failed recovery unit, RU_i . By theorem 6.1, there exists a checkpoint of RU_i which is not causally dependent on the state of the restarted recovery unit. The protocol rolls the state of RU_i back to the state contained in this checkpoint.

When RU_i rolls forward, it replays file system operations from its log until either the log is exhausted, or it finds an orphan file system operation. Potential orphans can be detected since the system attaches a time stamp to each file system operation, and this timestamp is saved with the operation in the log. In either case, the rollforward stops before an orphan file system operation is processed by RU_i . Thus, RU_i is consistent with the file system after rollforward, and is thus also consistent with the recovered RU.

C) Now consider any two non-failed recovery units, RU_j and RU_k . We must show that these two recovery units are mutually consistent with each other after rollforward. (Note that it is not sufficient to show that each non-failed recovery unit is consistent with the restarted RU after roll forward. There exist systems where some RU is pairwise consistent with all other RUs, but the resulting system is not globally consistent [39].)

As the RUs roll forward, they each replay file system operations from their logs. We claim that no rolled back RU will process an orphaned file system operation. First, note that the replay algorithm guarantees that no RU will replay a file system operation that occurred after the file system snapshot.

Since the file system is locked during the time of the snapshot, there are no file system operations “in-flight” while the snapshot was taken. By assumption, the file system exhibits sequential semantics, so no RU’s state can be affected by any file system operation performed by another RU after the the snapshot was taken. Thus, there are no orphaned file system operations and so RU_j and RU_k are mutually consistent.

Since every pair of recovery units is mutually consistent, and each recovery unit is consistent with the file system, a consistent global state exists. \square

6.3 Reclaiming Checkpoints and Logs - The Domino Effect

During normal operation, an RU will accumulate a growing list of checkpoints and log records. The question arises: when can an RU discard a checkpoint and the log records which precede it?

A checkpoint and the preceding log records can be discarded when the recovery unit which created them will never be required to restart from or roll back to that checkpoint. A checkpoint may potentially be used for restarting or rolling back under the following conditions: (1) an RU is re-instantiated from that checkpoint, or (2) an RU must be rolled back after the failure of another RU for the purposes of redoing one or more file system operations.

A failed RU always restarts from the latest valid checkpoint. Thus, the set of *possibly* discardable checkpoints consists of all the checkpoints in the system, except for the last checkpoint taken by each RU.

For each recovery unit, we need to identify which set of checkpoints will be referenced should any process in the system fail. In figure 6.1, checkpoint $C_{2,0}$ is discardable. In this figure, we assume that if one checkpoint appears at an earlier time than another, it was taken at an earlier time (in the Lamport sense.)

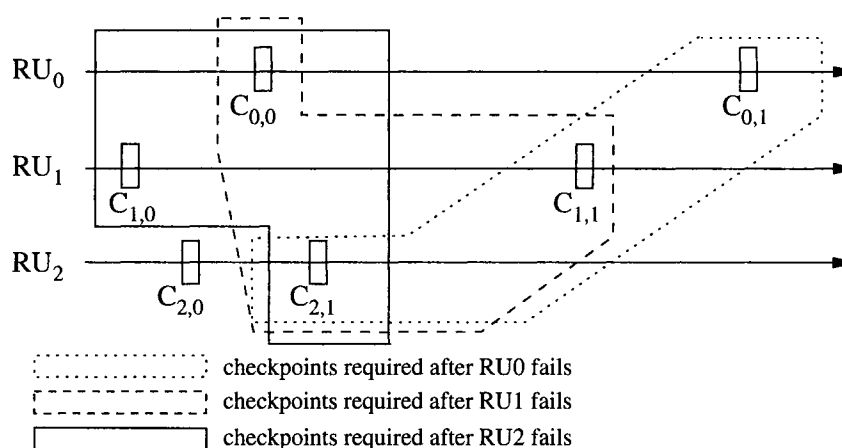


Figure 6.1: Finding Discardable Checkpoints

Theorem 6.4 *If some state interval I of RU_k is committable, and if all file system operations of RU_k performed after interval I are recoverable, then a system-wide consistent state can always be found without having to back out of interval I .*

Proof: Since state interval I is committable, by definition I does not depend on any orphan file system operations. Therefore, it is possible to recover all other recovery units to some point where state interval I is not causally dependent on them. That is, RU_k will never have to be backed out of interval I to undo orphan file system operations.

Our scheme guarantees that all file system operations performed by RU_k after state interval I are stored in RU_k 's log and are thus recoverable. \square

Theorem 6.5 *Provided that no RU indefinitely delays*

1 *logging its file system operations,*

2 *updating its vector time from the file system after a checkpoint, and*

3 *taking another checkpoint,*

then each RU will eventually be able to safely discard its oldest checkpoint.

Proof: Each recovery unit will eventually take another checkpoint (assumption 3). That checkpoint will be causally dependent on some set of states from other RUs. But each of these other RUs will eventually take another checkpoint (assumption 3) and update its vector time from the file system (assumption 2). Thus, each checkpoint will eventually become committable. Since each RU logs its file system operations before returning from the associated system call (assumption 1), both conditions of theorem 6.4 will eventually be met for discarding the oldest checkpoint. \square

Theorem 6.6 *There is no domino effect.*

Proof: By definition, the rollback of each non-failed RU is bounded by its earliest non-discardable checkpoint. From theorem 6.5, every checkpoint will eventually be discarded. Thus, there is a finite bound on the amount of rollback that the system must perform after a failure. \square

6.4 Messages

While our prototype implementation does not support interprocess communication via messages, real concurrent systems do send messages. In this section, we show that our algorithm can be adapted to systems which support message passing.

In the following discussion, we assume that the machine on which recovery units execute supports reliable, FIFO channels between recovery units. In addition, we assume a logging scheme for messages similar to the one that stores file system operations. Messages are logged at both the sender and the receiver. While schemes which require both sender and receiver based logging are unusual, they are not unheard of. Rao, et al.[45] discuss a hybrid protocol which logs both sent and received messages and is a variant of the traditional optimistic recovery protocol. We continue to assume that the log of a recovery unit is kept in volatile memory.

6.4.1 Recovery Protocol

As with file system operations, we require the system to piggyback the vector time of the sending RU on top of each message that is sent. When logging a sent message, the sending RU records its own vector time along with the message in the log. Upon receipt of a message, the receiving RU first updates its own vector time clock from the timestamp contained in the message, and then logs the message to its logging unit.

6.4.2 Rollforward

When a non-failed process enters the rollforward phase of the recovery protocol, it replays file system operations and messages from its log. When a message receipt is encountered,

the timestamp of the message is inspected. If the message was sent either prior to or concurrently with the recovery time of the rolled back file system, the receipt is processed. If the received message was sent after the recovery time, it is discarded.

When a log entry corresponding to a send is encountered during rollforward, the RU inspects the message's destination RU. If the message is destined for a non-failed RU, the send is discarded. Similarly, if the timestamp of the send is not concurrent with the recovery time, the send is discarded. However, if the destination of the message is the failed RU, and the timestamp indicates that the message was sent concurrently with the recovery time, the rolling forward RU must resend the message.

We require the RU that is rolling forward to resend the message to the failed RU, because such a message is necessarily an orphan in the recovering system. We know that the message is an orphan because the vector timestamp indicates that the message was not received by the failed RU. That is, if the message had been received by the failed RU before the recovery checkpoint was taken, the failed RU would have updated its clock from using the message timestamp, and when the failed RU took its last checkpoint, the filesystem time would have been later than the message send time.

An example is shown in figure 6.2. During replay, message m_0 's send is discarded by p_1 . Message m_1 's send is discarded by p_1 , and its receipt is replayed by p_2 . The sends of m_2 and m_3 are replayed by p_1 and p_2 since these messages are orphans.

We extend the definitions from section 6.1 as follows.

Recovery Unit A recovery unit now incorporates the log of sent and received messages, in addition to the operation log, the process checkpoint and the file system snapshot.

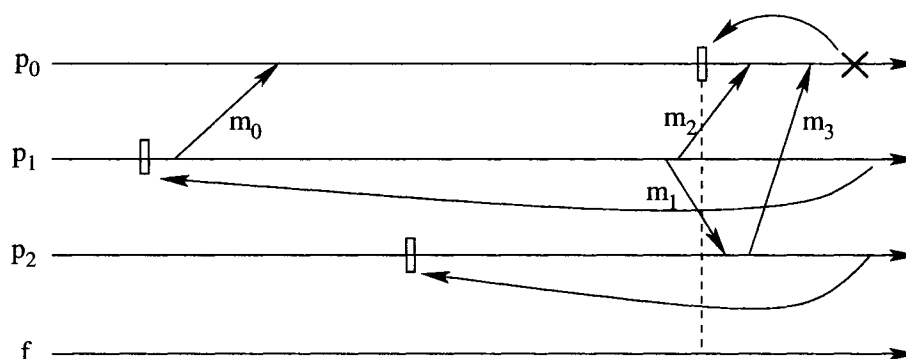


Figure 6.2: Message Replay During Recovery

State Intervals A state interval is the series of computations occurring between the execution of any two file systems operations, any two messages, between a message and a file system operation, or between a file system operation and a message.

Orphaned Messages An orphaned message is a message which is lost either because of the failure of the sender or receiver of the message, or because the state of the sender or receiver is causally dependent on a lost state.

As in theorem 6.3, we need to show that our algorithm with the addition of messages under these assumptions results in a system wide consistent state after recovery.

Theorem 6.7 *Assume we have a system which operates according to our algorithm, logs both sent and received message in volatile memory, and follows the protocol described above during rollforward. After the recovery phase, the system wide state is consistent.*

Proof: Again as in theorem 6.3, we need to show that each RU is pairwise consistent with all other RUs after recovery.

By theorem 6.2, the state of the restarted RU is consistent with the state of the file system.

Now consider some non-failed recovery unit, RU_i . We need to show that RU_i is consistent with the failed RU after recovery. By theorem 6.1, there exists a checkpoint of RU_i which is not causally dependent on the state of the restarted recovery unit.

All messages sent from the failed RU to RU_i are recoverable since they are stored in the log. RU_i will not receive any orphaned messages from the failed RU, since the corresponding message receipts are timestamped later than the recovery time and will be discarded. The RU which is rolling forward can detect any orphan sends in its log, since such messages will have a timestamp which is concurrent with the recovery time of the file system. Since all of these messages are retransmitted to the failed RU, no orphan messages exist. Thus the failed RU is consistent with each rolling forward RU.

Now consider any two non-failed recovery units, RU_j and RU_k . As in theorem 6.3, we must show that these two RUs are mutually consistent after rollforward.

We claim that RU_j will not perform a file system operation or send or receive a message that results in an orphaned state for RU_k . Again, since the file system snapshot acts as a barrier, no orphaned file system operations will be performed. Each RU inspects the timestamp of received messages and discards any which were sent after the recovery time. Since sends between non-failed RUs are discarded, no orphan messages are generated. Thus, RU_j and RU_k are mutually consistent.

We thus have the following: the restarted RU is consistent with the file system after restart, and each non-failed RU is consistent with the restarted RU and all other non-failed RUs after rollforward. Thus, each RU is pairwise consistent with all other RUs after

recovery, so a consistent global state exists. \square

6.5 Unusual Failure Modes

In order for a checkpointing and rollback-recovery technique to be considered robust, it must operate correctly for all expected modes of failure. In order to minimize the set of expected modes of failure, most techniques referenced in the literature make two important assumptions about the behavior of executing processes. They assume independence: that is, they assume that a failed process will not re-fail if it is re-executed using spare processing capacity. And second, they assume that failure is a relatively rare occurrence. Both of these are *prima facie* reasonable arguments.

If the probability of a single recovery unit failing during some computation is f , and there are n recovery units, then the probability of at least one recovery unit failing during the computation is

$$\sum_{i=1}^n \binom{n}{i} (f)^i (1-f)^{n-i}.$$

Since this is just the binomial distribution with parameter f , the probability that no recovery unit fails is

$$1 - (1 - f)^n.$$

As an example, let us assume a hypothetical system where the probability of a single recovery unit failing is 0.001 over the life of some long running concurrent computation. Then the probability that a computation consisting of 100 RUs will not suffer any failures

over the life of the computation is 0.90480. The probability that exactly one RU will fail is, of course, 0.001.

In this section, we are interested in the failure of two or more processes during the failure-recovery stage of the protocol. As an illustrative example, let us assume that a computation takes one month to run, and that recovery from a failure takes 10 minutes. The probability that a single recovery unit will fail during any 10 minute interval is then $0.001 \cdot \frac{10 \text{ minutes}}{1 \text{ month}}$. The probability that two or more RUs will fail during any single ten minute period is then

$$\begin{aligned} P(\text{failures} \geq 2) &= \sum_{i=2}^{100} \binom{100}{i} \cdot \left(0.001 \cdot \frac{43200}{10}\right)^i \cdot \left(1 - \left(0.001 \cdot \frac{43200}{10}\right)\right)^{100-i} \\ &= 2.6523e - 10 \end{aligned}$$

Despite the minuscule probability of these types of failures, it is nevertheless interesting to consider what types of multiple failures our scheme can deal with. Below, we consider two types of such failures. In the discussion that follows, we use the term “incarnation” to describe the execution of recovery units after some failure.

6.5.1 Concurrent Failures

Concurrent failures are failures of multiple RUs that happen simultaneously, in the vector time sense. Concurrent failures occur when an RU failure is detected and the system initiates the rollback-recovery protocol. Before the protocol can notify the non-failed RUs that recovery is underway, one or more additional RUs fail.¹ In section 6.5.2 we look at the

¹Our implementation does not handle concurrent failures. When the recovery protocol is initiated, the system sets a flag in each non-failed RU’s address space indicating that recovery is underway. This flag is

behavior of the system if we allow failures to occur during the recovery process.

Even under this failure mode, recovery using our scheme can still commence. Once the non-failed RUs pause, the file system is rolled back to an appropriate checkpoint, and the vector time of the rolled back file system is transmitted to each non-failed RU. These RUs then identify an appropriate checkpoint in their checkpoint list from which to restart. They perform the restart and begin the rollforward.

It is interesting to ask what modifications we would need to make to the system to support concurrent failures. In addition, if such support is added, the question of how many concurrent failures can be tolerated must be addressed.

In order to handle concurrent failures, we must extend our protocol as follows:

1. The fault catching code of the system must continue monitoring non-failed RUs after catching the first failure. This monitoring must continue until all non-failed RUs are quiescent. In the prototype, the fault catching code simply waits for system quiescence at this point, rather than continuing to monitor for failure.
2. The recovery module must be extended to determine which of the failed RU's most recent checkpoints is the oldest. See figure 6.3. This is the checkpoint from which the file system is reconstructed, and toward which all other RUs must rollforward. In the current implementation, this checkpoint is always the latest checkpoint of the single failed process.

not inspected until the next file system call is executed by the non-failed RU.

Because we can not rollback the file system while the non-failed RUs are accessing it, we must wait for these RUs to pause. There is currently no provision in the prototype recovery module to monitor additional RUs for failure while waiting for quiescence.

In figure 6.3, RU_0 fails. Before the system becomes quiescent and the rollback can begin, both RU_1 and RU_2 fail. The set of most recent checkpoints from each process consists of $C_{0,1}, C_{1,1}$ and $C_{2,2}$. The most recent of these (in the Lamport sense) is $C_{1,1}$ and the file system is rolled back to this point. RU_0 rolls back to $C_{0,0}$ and RU_2 rolls back to $C_{2,1}$. The non-failed recovery unit, RU_3 would rollback to its most recent checkpoint which causally precedes $C_{1,1}$; in this case $C_{3,1}$.

Assuming the modifications listed above are made to the implementation, we now ask: How many concurrent failures can the protocol handle? Surprisingly, the state recovery protocol can be used to recover from any number of process failures in the system, including a total failure of all processes.

Theorem 6.8 *A system consisting of N recovery units which uses the recovery protocol described above can recover from any $i \leq N$ concurrent failures.*

Proof: Let $RU_0, RU_1, \dots, RU_{i-1}$ be the set of recovery units which have failed concurrently, and let C_0, C_1, \dots, C_{i-1} be the set of checkpoints most recently taken by the corresponding processes. Let C_k be the element of the set of recent checkpoints which has the property $\forall i, \text{time}(C_k) \leq \text{time}(C_i)$. That is, C_k is the earliest checkpoint (in the Lamport sense) among the set of latest checkpoints taken by all concurrently failed RUs. Note that there may be more than one checkpoint with that property. If so, chose one arbitrarily.

Theorem 6.1 tells us that each RU_l such that $l \neq k$ has an associated checkpoint which is not causally dependent on the latest checkpoint of any other RU. C_k is such a checkpoint, since it was chosen from the set of most-recent checkpoints for each process. If we rollback the file system to its state when C_k was taken, then we can rollback all other RUs to some

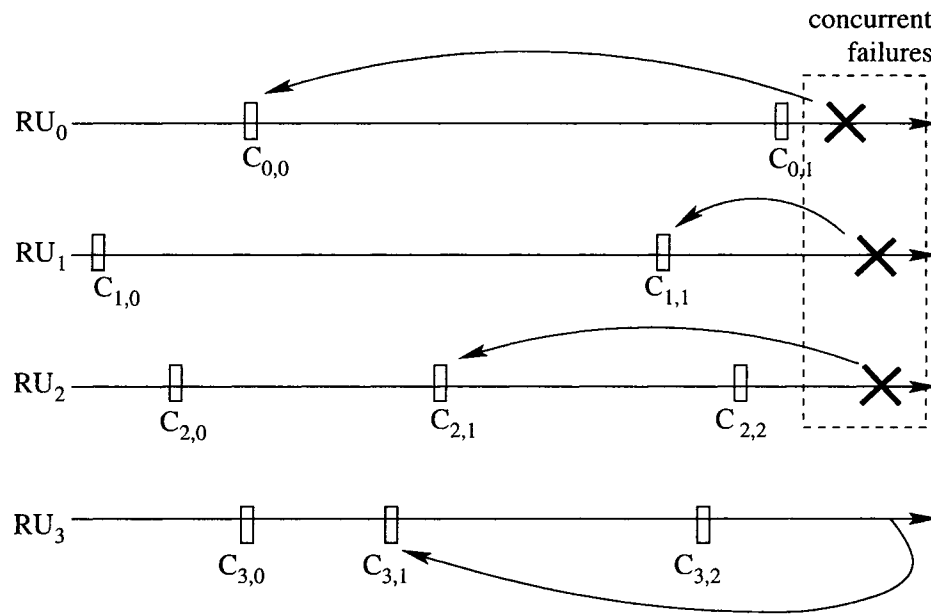


Figure 6.3: Rollback After Concurrent Failures

state that is not causally dependent on the state stored in C_k .

Since for each RU we can find a checkpoint which is not causally dependent on the state stored in C_k , it follows immediately from theorem 6.3 that a system wide consistent state can be achieved by rolling any non-failed RUs forward. \square

In this section, we dealt with the failure of more than one process before the recovery algorithm began to execute. In the next section, we look at the problem of failures during the recovery process.

6.5.2 Failures During Recovery

There are two types of failures that can occur during the recovery phase of the protocol. We can have a second failure a failed process before the recovery completes, or one of the non-failed processes which has been rolled back may fail during the rollforward phase of the

recovery. In this section, we address these two types of failures.

6.5.2.1 Failures Without the Assumption of Independence

The first type of failure we wish to consider is this: Suppose that RU_k fails. The failure is detected and the recovery protocol begins to execute. Sometime after the reinstatement of RU_k , RU_k fails again. This type of failure is only possible if we remove our assumption of independence. (See section 2.2.1.) We need to consider two possibilities.

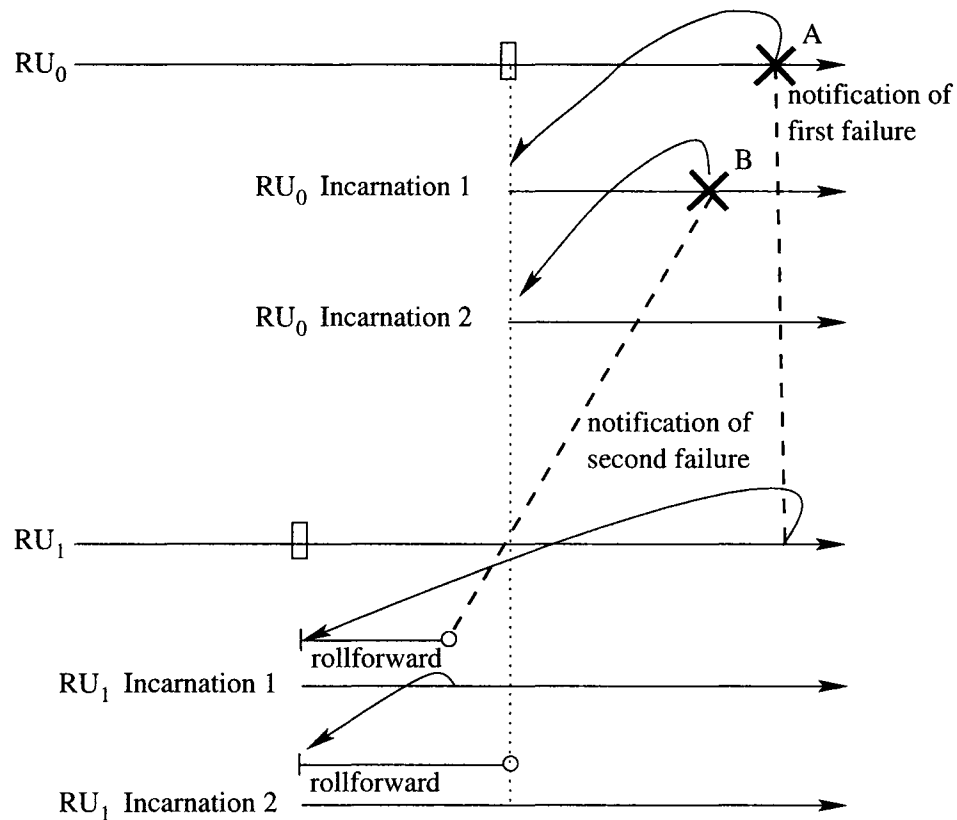


Figure 6.4: A Possible Failure Mode if We Drop the Assumption of Independence.

The first possible mode of failure occurs when the failed RU fails for a second time before the non-failed RUs have finished their rollforward, as illustrated in figure 6.4. In

this diagram, RU_0 fails at time A . At time B , RU_0 fails again before RU_1 completes its rollforward.

Under this mode of failure, we must notify RU_1 that a refailure has occurred. Note that under this failure mode, the system state is consistent during the entire process between the restart of RU_0 and the second failure at time B . This is because RU_1 is replaying file system operations from its log.

The second possible mode of failure occurs when the failed RU again fails for a second time, but the non-failed RUs have finished their rollforward. This failure mode is shown in figure 6.5. At time A , RU_0 fails. At time B , RU_0 fails again, but RU_1 has already finished its rollforward and has resumed normal computation.

Note that the second failure mode is the simpler of the two. Under the second failure mode, where the non-failed RUs have finished their rollforward, the system state is indistinguishable from a normal single-process, independent failure. All RUs are executing normally and accessing the file system in the usual way. Each RU has a valid checkpoint on disk. The fact that this checkpoint has already been used for a restart is immaterial.

We now formalize the arguments presented above.

Theorem 6.9 *Let RU_k be an RU which fails repeatedly. If the number of failures is bounded by some finite value F , then the protocol will always bring the system to a consistent state some time after the last failure of RU_k .*

Proof: The proof is by induction on F , the number of failures.

Inductive Basis: Choose $F = 1$ as the inductive basis. Note that this is exactly the failure mode discussed in section 6.2. We showed in theorem 6.3 that under this failure

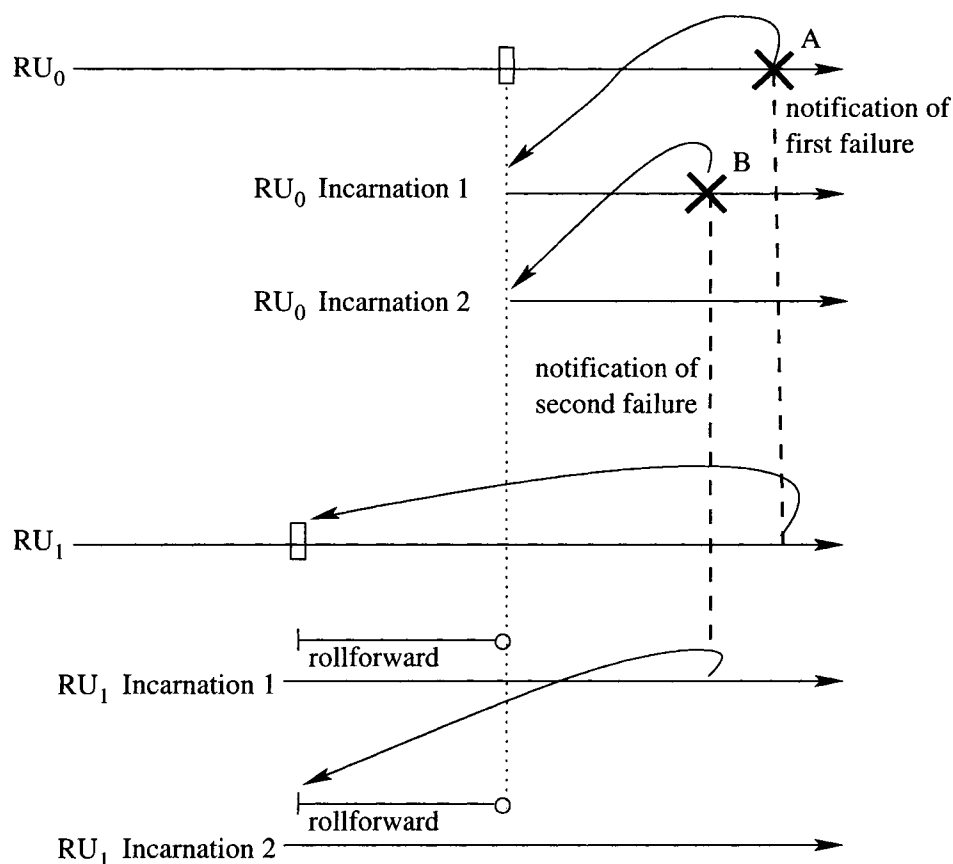


Figure 6.5: A Second Possible Failure Mode if We Drop the Assumption of Independence

mode the resulting system state is consistent.

Inductive Step: Now assume that some RU_k has failed $F - 1$ times for $F \geq 2$, and consider what happens on failure F . There are two cases to consider, depending upon whether the non-failed RUs have finished their rollforward, or are still replaying file system messages from their logs.

Case 1: Failure F occurred before the non-failed RUs finished replaying their logs. (This is the scenario shown in figure 6.4.) In this case, the non-failed RUs reconstitute themselves from the checkpoints from which they just restarted. Such a state is causally consistent,

since the protocol chooses checkpoints for each non-failed RU to restart from that causally precede the checkpoint from which the failed RU restarted. Since this is the last failure, the non-failed RUs will roll themselves forward according to the protocol until their logs are exhausted or until they are about to replay an orphaned file system operation. By theorem 6.3, such a non-failed RU is consistent with all other non-failed RUs, the restarted RU and the file system.

Case 2: Failure F occurred after the non-failed RUs finished replaying their logs. This is the simpler case, as shown in figure 6.5. Since we know that the system state was consistent after failure $F - 1$, the state was also consistent before failure F , since failure $F - 1$ preceded failure F . We thus have a failure in a system which is causally consistent. Again by theorem 6.3, the resulting system state is consistent. \square

6.5.2.2 Failures During Rollforward

The last type of failure mode which we wish to consider is this. Suppose that RU_k fails. The non-failed RUs are notified, the file system is rolled back and RU_k is restarted. However, at some time during the rollforward of some non-failed RU_j , another failure occurs. This scenario is shown in figure 6.6. In this diagram, RU_0 fails and begins the recovery protocol. During the rollforward, RU_1 fails. It is restarted from the checkpoint which causally precedes the one from which RU_0 restarted. No other recovery units need to be notified of the failure. (Note that we do not need to consider the case where the RU fails after rollforward is complete, since such a system is identical to one where no failure has taken place, in terms of causal consistency.)

During such a failure, we simply require the RU which failed during rollforward to restart

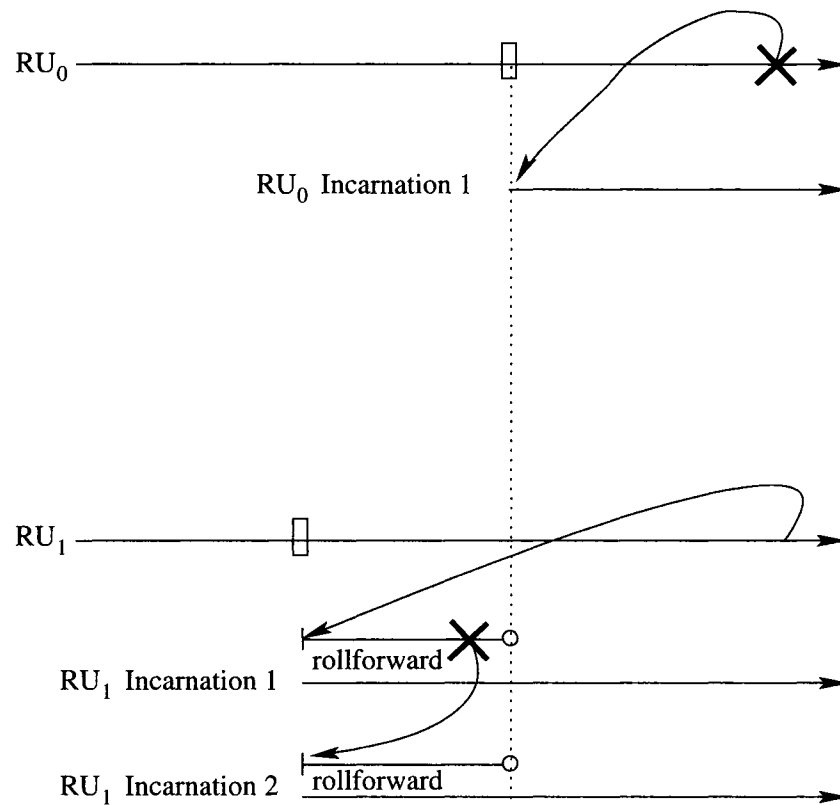


Figure 6.6: A Failure During Rollforward

from the checkpoint that it had previously restarted from. Since the RU which failed was still rolling forward, it was replaying file system operations from its log. Thus, it had not modified the file system, so it can not have effected the state of any other RU.

Once this RU completes its rollforward, its state will be consistent with that of the rolled back file system, and thus (by theorem 6.3) the resulting system state will be consistent.

Chapter 7

Performance Evaluation and Experimental Results

*Which of you, intending to build a tower,
sitteth not down first, and counteth the cost
whether he have sufficient to finish it.*

Luke, 14:28, *KJV*

In this chapter, we discuss the theoretical and empirical performance of the file system by analyzing the algorithms used in file system checkpointing and rollback. We present experimental results of a practical application which uses our system in a fault-tolerant environment. We conclude the chapter by looking at an interesting result obtained by experimenting with different log clearing schemes.

7.1 File System Theoretical Performance

For application programmers to want to use a file system, it must give acceptable performance under a wide variety of applications. Because our implementation is based on the Unix FFS [38], we know that the basic structures and algorithms of our implementation are sound and have found favor with programmers over many decades. The two parts of our implementation that have not received widespread use, however, are the `sync()` operation, and the performance of the `rollback()` system call.

In this section, we evaluate both of these system calls. We begin with an analysis of the algorithms used. We then present experimental results which confirm our analysis. We continue to use the notion of failure-free overhead (see section 2.2.1 to evaluate the performance of our system. As is standard in the literature which discusses file systems, we will use the concept of “disk block accesses” as the metric.

7.1.1 Sync() Evaluation

The cost of a `sync()` has two main components. A fixed cost, which is incurred each time a `sync()` is performed, and a variable cost, which is the cost of writing to disk those blocks which have been modified since the file system was last `sync()`ed or `mount()`ed.

7.1.1.1 Fixed Cost Component of Sync()

Certain structures associated with the file system must be written to the disk each time a snapshot is taken. These structures are the superblock, the checkpoint region, the `imap`, the file system vector time and the free-list. The first four of these structures each use one block on disk. As shown in figure 5.6, we use a series of up to eight trees of depth 2 to store

the free-list. The number of trees used depends on the size of the underlying virtual disk. Thus, the fixed costs are truly fixed for a virtual disk of a particular size, but increase as the size of the underlying disk increases.

A single 1K disk block contains $2^{14} = 8192$ bits, and can indicate the allocation status of 8,192 disk blocks. A single tree in our structure can contain pointers to the addresses of $2^7 = 128$ disk blocks. Therefore, each tree encompasses enough disk blocks to store a bitmap of size $2^{14} \cdot 2^7 = 2^{21}$ bits, and so a single tree is large enough to hold the status of two million disk blocks. Again, assuming a block size of 1K, a single tree can describe the allocation status of a 2GB file system.

Whenever the number of blocks in the underlying disks surpasses a multiple of 2GB in size, we allocate another tree in the checkpoint region to hold the freelist bitmap. Writing this additional tree requires writing a single additional block for the index, and an additional 128 blocks for the bitmap itself. See figure 5.6. Thus, for a disk with N 1K blocks, the total cost of writing the freelist to disk is $\lceil \frac{N}{2^{21}} \rceil + \lceil \frac{N}{2^{13}} \rceil$. Combining this with the cost of writing the superblock, etc., the total fixed costs of `sync()`ing a disk consisting of N 1K blocks is $4 + \lceil \frac{N}{2^{21}} \rceil + \lceil \frac{N}{2^{13}} \rceil$.

Table 7.1 lists the fixed costs incurred when `sync()`ing a file system on disks of various sizes.

We verified this formula by repeatedly creating virtual disks ranging from 1GB to 16GB in size. After a disk was formatted, we performed a `sync()` operation on the file system and instructed the file system to report the number of blocks written by the `sync()`. Figure 7.1 shows the results of this experiment.

Disk Size	Block Writes
1 GB	133
2 GB	261
3 GB	390
4 GB	518
5 GB	647
6 GB	775
7 GB	904
8 GB	1032
9 GB	1161
10 GB	1289
11 GB	1418
12 GB	1546
13 GB	1675
14 GB	1803
15 GB	1932
16 GB	2060

Table 7.1: Fixed Costs of a `Sync()` Operation

7.1.1.2 Variable Costs Component of `Sync()`

The variable cost of the `sync()` counts the cost of writing modified data and metadata to the file system. It is bounded above by the number of files and disk blocks modified since the file system was mounted or the last `sync()` was performed.

If only a small amount of data is written to the file system, the fixed costs of the `sync()` will dominate. However, as more and more data blocks on the disk are modified, the total contribution of the fixed costs to the failure-free overhead diminishes.

Table 7.2 lists the worst case cost of `sync()`ing a 16GB file system when varying amounts of data is written to the file system. Note that when few data blocks are modified between `sync()` operations, the fixed costs dominate. But as larger amounts of data are written, the fixed costs are amortized over more disk writes and contribute a smaller and smaller

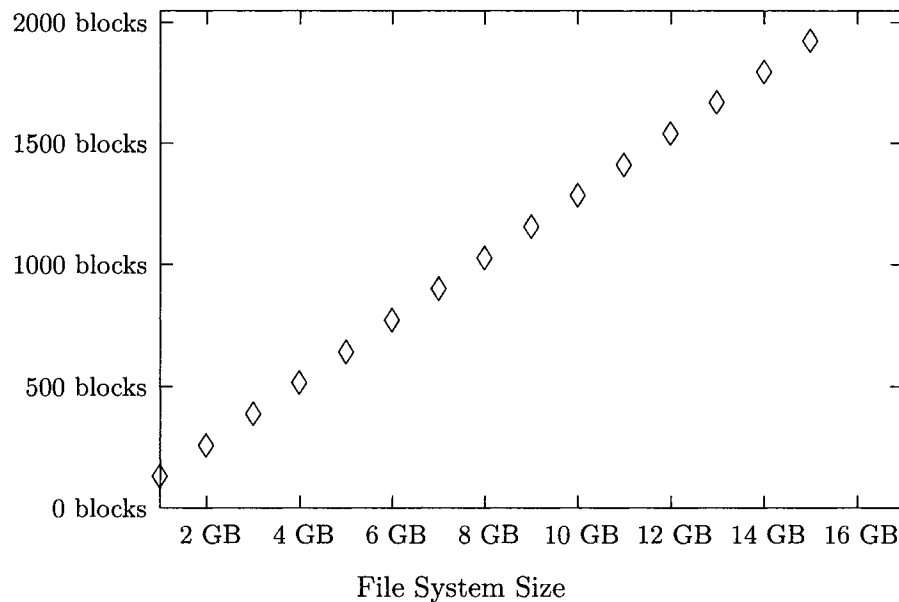


Figure 7.1: Experimental Results of `sync()`ing a Virgin File System on Disks of Various Sizes

percentage to the total cost of the `sync()` operation.

The data in table 7.2 was generated by a program accessing a file system built on a 16GB disk. The program opens 15 different files, writes an equal amount of data to each file, and then closes the files. The number of disk writes was then extracted from the file system, a `sync()` was performed, and the file system was again instructed to report the number of disk writes.

The table shows the total worst-case overhead of a `sync()` operation as the amount of data written between `sync()`'s increases. Column two details the number of blocks written during the `sync()` call. Column three shows the percentage of the total disk traffic caused solely by the `sync()`.

Note that some of the cost attributed to the `sync()` in column two is larger than the total `sync()` overhead. This figure also includes the cost of flushing any modified data

blocks from the disk cache, along with the cost of writing the inode and indirect blocks, operations which a `sync()` is required to do. As the data sets grow larger, most of the data is forced from the cache and written to disk before the actual `sync()`. The experiment was performed with a cache size of 1MB. In an actual system, the disk cache size would probably be much larger than our experimental value of 1MB, so the values would be proportionally increased.

Data Modified	Sync() Overhead (in blocks written)	Sync() Cost (as a % of blocks written)
15 KB	2090	99.3 %
30 KB	2105	98.6 %
60 KB	2135	97.3 %
120 KB	2195	94.8 %
240 KB	2315	90.6 %
480 KB	2555	84.2 %
1 MB	3114	61.9 %
3 MB	3144	45.0 %
7 MB	3174	29.2 %
15 MB	3234	17.4 %
30 MB	3354	9.8 %
60 MB	3594	5.5 %
120 MB	4074	3.2 %
240 MB	5034	2.0 %
480 MB	5960	1.2 %
1 GB	6050	0.3 %
3 GB	6170	0.2 %
7 GB	6410	0.1 %
15 GB	6890	0.0 %

Table 7.2: Total Worst-Case Sync() Overhead

7.1.2 Rollback() Evaluation

When a `rollback()` operation occurs, the system discards the current state of the file system, including any modified data blocks in the disk cache, locates the correct checkpoint

region, and reinstantiates the file system using this checkpoint region.

Locating the correct checkpoint region requires the system to follow the chain of checkpoint regions, looking for the last checkpoint taken by the failed RU. Once the checkpoint is read from disk, the system locates and reads the block bitmap, the imap, and the file system vector time. The most expensive of these operations in terms of disk reads is the reinstantiation of the correct bitmap.

Assuming that the system must read C checkpoint regions, and the disk consists of N 1K blocks, the total cost of the `rollback()` operation is $C + 2 + \lceil \frac{N}{2^{21}} \rceil + \lceil \frac{N}{2^{13}} \rceil$. If we assume that RUs checkpoint with approximately the same frequency, we would expect a system of R RUs to need to search, on average, $\frac{R}{2}$ checkpoint regions to locate the appropriate one for the failed RU. Table 7.3 shows the number of disk block reads required to rollback file systems of various sizes for a system with eight RUs. We assume in this table a disk block size of 1KB, with four checkpoint regions inspected during the rollback.

7.2 File System Empirical Evaluation

In order to evaluate the performance of the system under a “real-world” workload, we programmed a distributed sorting application that used our fault-tolerant file system scheme. The particular algorithm we used is known as “columnsort”. Columnsort was invented in 1984 by Leighton[31] while he was investigating bounds on parallel sorting. It is a generalization of the Knuth calls the “odd-even merge”[26]. We chose columnsort because it admits itself to unbounded parallelism, subject to certain constraints on the data set size, and it is easily adaptable to large, out-of-core data sets, making it a good candidate to run

File System Size	Disk Block Reads
1 GB	134
2 GB	262
3 GB	390
4 GB	518
5 GB	646
6 GB	774
7 GB	902
8 GB	1031
9 GB	1159
10 GB	1287
11 GB	1415
12 GB	1543
13 GB	1671
14 GB	1799
15 GB	1927
16 GB	2056

Table 7.3: Cost of Rollback() System Call

on top of a fault-tolerant file system.

7.2.1 The Columnsort Algorithm

The columnsort algorithm divides the sort into eight phases, numbered 0-7. Phases 0, 2, 4 and 6 are sorting phases. Phases 1, 3, 5 and 7 transform the data in particular ways. To sort N data items, the data is organized into an R by C matrix, padding the data as necessary so that the following set of constraints is met.

- $N = RC$
- $\frac{R}{C}$ is an integer
- $R \geq 2(C - 1)^2$

See figure 7.2.

$$[a, b, c, \dots, r] \longrightarrow \begin{bmatrix} a & g & m \\ b & h & n \\ c & i & o \\ d & j & p \\ e & k & q \\ f & l & r \end{bmatrix}$$

Figure 7.2: Preparing Data for Columnsort

The columnsort then precedes as follows:

Phase 0: Sort The data in each column is sorted.

Phase 1: Transpose and Reshape The transpose of the matrix is computed, and the matrix is then reshaped back into an R by C matrix. See figure 7.3.

$$\begin{bmatrix} a & g & m \\ b & h & n \\ c & i & o \\ d & j & p \\ e & k & q \\ f & l & r \end{bmatrix} \xrightarrow{\text{transpose}} \begin{bmatrix} a & b & c & d & e & f \\ g & h & i & j & k & l \\ m & n & o & p & q & r \end{bmatrix} \xrightarrow{\text{reshape}} \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \\ j & k & l \\ m & n & o \\ p & q & r \end{bmatrix}$$

Figure 7.3: Columnsort Phase 1

Phase 2: Sort The data in each column is resorted.

Phase 3: Reshape and Transpose This step is the inverse of phase 1. The data in the columns is reshaped into a C by R matrix, and the transpose is then computed, yielding an R by C matrix.

Phase 4: Sort The data in each column is resorted.

Phase 5: Shift Down Form an R by $C + 1$ matrix, by shifting the data in each column

down by $\frac{|R|}{2}$ elements. Pad as necessary at the top of column 1 and the bottom of column $C + 1$ with $-\infty$ and ∞ , respectively. See figure 7.4.

$$\begin{bmatrix} a & g & m \\ b & h & n \\ c & i & o \\ d & j & p \\ e & k & q \\ f & l & r \end{bmatrix} \xrightarrow{\text{shift down}} \begin{bmatrix} -\infty & d & j & p \\ -\infty & e & k & q \\ -\infty & f & l & r \\ a & g & m & \infty \\ b & h & n & \infty \\ c & i & o & \infty \end{bmatrix}$$

Figure 7.4: Columnsort phase 5

Phase 6: Sort The data in each column is resorted.

Phase 7: Shift Up Shift the elements of each column up by $\frac{|R|}{2}$ elements, discarding the $-\infty$'s and ∞ 's.

At the end of phase 7, the matrix is sorted in column major order.

To evaluate the performance of our system, we conducted two sets of experiments. In both sets of experiments, the columnsort algorithm was used to sort 2MB of data using four threads. The experiment worked as follows:

- Two MB of pseudo-random data was generated and written to four separate files. Each file contained 512 KB of data. The threaded columnsort application treats the four files as a single 524,288 by 4 matrix. Files 0, 1, 2 and 3 (i.e., columns 0, 1, 2 and 3) are maintained by threads 0, 1, 2 and 3, respectively.
- Each thread is responsible for manipulating one column of data. During the sorting phases, each thread reads the data from its associated file, sorts the data, and then writes the data back to the file.

- During the transformation phase, each thread reads the data from its associated file, calculates the correct position in the proper file for each data item after the transformation, and then writes the data item to that location. The reading and writing of the files is coordinated with a condition variable to ensure that no data items are overwritten.
- After all eight phases of the algorithm complete, thread 0 makes a final pass over all the data to verify that it is sorted in ascending order.

7.2.2 Effects of Buffer Cache Size on Performance

Because a log-structured file system must flush all dirty blocks from the buffer cache on each `sync()` operation, the cost of a `sync()` is sensitive to the buffer cache size. To evaluate this effect, we measured the number of disk writes performed during the experiment described in section 7.2.1.

Figure 7.5 displays the average number of physical block writes performed over 5 executions of the `columnsort` algorithm, both with and without checkpointing. The horizontal axis shows the disk cache size in kilobytes, and the vertical axis shows the average number of disk block writes performed for the given cache size.

7.2.3 Failure-Free Overhead in Columnsort

To evaluate the failure-free overhead of our scheme in a practical problem, we again measured the performance of `columnsort`. In this experiment, we calculated the number of disk block writes caused solely by `sync()` operations.

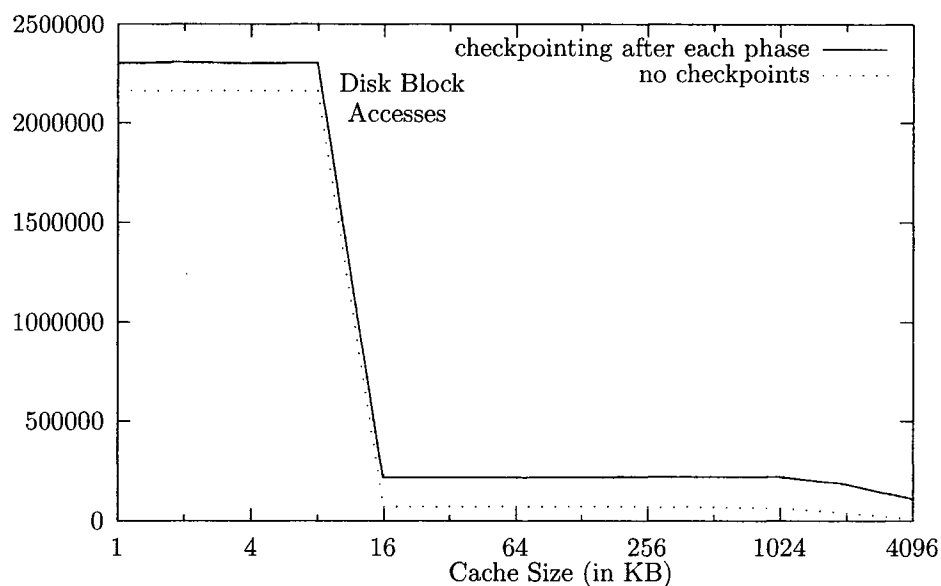


Figure 7.5: Effect of Buffer Cache Size on the Number of Disk Writes Performed by Columnsort, With and Without Checkpointing

Prior to each CHECKPOINT by a thread in the fault-tolerant implementation of column-sort, we locked the file system and had it report the total number of disk block writes done by the system so far. We then executed a checkpoint operation, and again instructed the system to report the number of disk writes before unlocking the file system. The difference between these two numbers is the total number of disk writes caused solely by a checkpoint operation.

Fifty executions of the column sort were run as described in section 7.2.1. Since each of the four threads takes eight checkpoints over the course of the sort, we measured a total of 800 checkpointing operations. Table 7.4 shows the observed mean, observed standard deviation, and the 95% confidence estimate of the mean number of disk writes.

In order to estimate what type of performance an end-user might be able to expect from our system, we obtained a data spec sheet from a hard drive manufacturer's website. We choose as an example hard drive the CheetahTM10K.6 server drive available from Seagate.

	Disk Writes
Observed Mean	3879 blocks
Observed Standard Deviation	2015 blocks
95% Confidence Estimate of Mean	3879 \pm 118 blocks

Table 7.4: Average Disk Writes Caused by a Single Sync() in Columnsort

The 10K.6 is advertised as a high-end server disk. We believe its performance is typical of currently available disks in that category. The disk is available in 37, 73 and 147GB configurations. All of these configurations contain the same basic read/write head and platter mechanical controls. The increasing capacities are obtained by adding additional platters to the basic configuration. Each configuration is available with UltraSCSI and FibreChannel interfaces. The relevant performance statistics of the 37GB version are summarized in table 7.5.

Track-to-Track Seek Time	0.55 msec
Average Seek Time	5.30 msec
Sector Write Time	0.05 msec
Average Latency	2.99 msec
Sectors per Cylinder	1438

Table 7.5: Performance of the Seagate Cheetah™10K.6 37GB Disk Drive

In a traditional Unix file system, disk blocks become scattered all over the surface of the disk as the file system matures.¹ Thus, each block access requires a seek followed by one-half of a platter rotation before the desired block can be read or written by the disk

¹Modern Unix file systems attempt to palliate this scattering effect with the use of “block groups”. A file system which uses block groups places groups of inodes at regularly spaced intervals over the logical disk, rather than placing them in a single, logically contiguous location on the disk. When new blocks are written, the file system attempts to write the data and metadata related to each file as near as possible to the block group which contains the file’s inode in order to reduce the average seek time between the inode and the data or metadata.

head. The cost of writing N disk blocks under such a system is

$$N \cdot (\text{average seek time} \cdot \text{average latency})$$

Contrast this value with the cost of writing N disk blocks in a log-structured file system. Since all disk blocks are written to free areas of the disk, and since an LSFS ensures that large contiguous areas of free space always exist, the disk needs only to seek to the beginning of the free space on the disk before writing. At this point, the writing can proceed at the disk's theoretical maximum. The cost of writing N disk blocks on an LSFS where the target blocks are physically adjacent is then

$$\begin{aligned} & \text{Average Seek Time} + \text{Average Latency} \\ & + N \cdot \text{Sector Write Time} \\ & + \frac{N}{\text{Sectors per Cylinder}} \cdot \text{Adjacent Track Seek Time} \end{aligned}$$

The fourth term in the sum above refers to the time required to seek to an adjacent track once the blocks of a cylinder become completely allocated. In practice, this term is so small that it can be discounted.

Applying the above formulas to the performance values of the Cheetah 10K.6 gives us the following measures:

Time to write N blocks in a traditional Unix file system

$$\begin{aligned} & \approx N \cdot (5.30 \text{ msec} + 2.99 \text{ msec}) \\ & = N \cdot 8.29 \text{ msec} \end{aligned}$$

Time to write N blocks in a log-structured file system

$$5.30 \text{ msec} + 2.99 \text{ msec} + N \cdot 0.05 \text{ msec} + \frac{N}{1438} \cdot 0.55 \text{ msec}$$

$$\approx (8.29 + N \cdot 0.05) \text{ msec}$$

In our implementation of column sort, a total of 34,816 disk access are needed to read data at the beginning of each phase and write it at the end of each phase. Applying the numbers from table 7.5 to a system which uses a traditional Unix file system with no fault-tolerance gives us a total block access time of

$$34,816 \cdot 8.29 \text{ msec} \approx 288.62 \text{ seconds}$$

Adding in the additional block writes induced by the `sync()` operations gives a total block access time for the column sort on an LSFS-based fault-tolerant file system of

$$34,816 \cdot 8.29 \text{ msec}$$

$$+ 32 \cdot (8.29 \text{ msec} + 3879 \cdot 0.05 \text{ msec} \pm (118 \cdot 0.05 \text{ msec}))$$

$$\approx 295.10 \pm 0.19 \text{ secs}$$

Thus, failure-free overhead of our prototype increases the total disk access time by approximately 2.3%.

7.2.4 A Comparison of the Prototype with AIPC

As we discussed in section 2.3.3, the most comprehensive attempt to integrate files into checkpointing and rollback-recovery schemes which appears in the literature is the AIPC

scheme of Alagar et al [2]. In this section, we compare the failure-free overhead of our prototype with AIPC.

One disadvantage of AIPC is that each file which is opened requires the creation of a separate server process to manage access to the file. Our scheme requires no such additional processes. However, since operating systems vary greatly in the amount of work which must be done to create a new process, we simply note that particular difference here and do not discuss its effects on failure-free overhead.

Of more interest here is the number of disk block accesses generated by AIPC. During a checkpointing operation, an AIPC system performs the following steps:

1. Take a snapshot of the process state. Included in the snapshot is a list of files currently opened by the process.
2. Inform the manager process for each file currently opened by the process to perform a snapshot.
3. Each manager process takes a self-checkpoint.
4. Each manager process makes a copy of the file it is responsible for.

Table 7.6 shows the number of block read, write and copy operations performed by AIPC when executing a fault-tolerant version of columnsort. Because the block copying operations occurs within the per-file servers and not in the threads, we assume that the server carries out the copy by reading and writing each block.

AIPC performs 188,388 block accesses, compared to the observed number of disk block accesses of $158,944 \pm 3776$ performed by our prototype. Thus AIPC performs, on average,

Per Thread Block Reads, Block Writes and File Copies Performed by AIPC					
Phase(s)	Thread(s)	Reads	Writes	File Copies	Disk Block Accesses
0, 2, 4	all	512	512	1	12 · 4320
1, 3	all	512	512	4	8 · 8064
5	0, 3	512	1024	2	2 · 6152
5	1, 2	512	512	2	2 · 5640
6	0	1024	1024	4	9232
6	1, 2, 3	512	512	1	3 · 4356
7	0	1024	1024	4	9232
7	1, 2, 3	512	512	2	3 · 5640
Total:					188,388

Table 7.6: Per-thread Block Reads, Block Writes, File Copies and Total Disk Block Accesses Performed by AIPC During Columnsort

118.5% \pm 2.0% \pm more disk accesses than our prototype. The fact that these numbers are relatively close is not surprising, given that both schemes essentially require all modified data to be flushed to disk during the CHECKPOINT operation. The extra reads performed by the file manager threads account for most of this extra cost in AIPC.

If we apply the numbers from table 7.5, we find that the total amount of time AIPC spends accessing disk blocks is 1561.74 seconds executing on a traditional Unix file system as compared to 295.10 seconds for our prototype executing on a fault-tolerant, LSFS based system. Thus, AIPC spends approximately 429% more time performing physical disk accesses than does our prototype.

7.3 Log Clearing

As a fault-tolerant application executes, the RUs will accumulate more and more checkpoints and logged file system operations. A natural question to ask is: When can a checkpoint and the associated log entries be deleted?

The obvious answer to that questions is: A checkpoint can be discarded when no recovery unit will ever be required to restart from it. Since our algorithm does not suffer from the domino effect (see section 6.3,) we know that every checkpoint will eventually become discardable (see theorem 6.5). We address two questions in this section: “How do we identify discardable checkpoints?” and “How often should we look for discardable checkpoints?”

7.3.1 Discardable Checkpoints

Identifying discardable checkpoints turns out to be straightforward. A checkpoint is not discardable if it may potentially be used for restarting a failed process, or if it may be used for rolling back a non-failed process during failure-recovery. Below, we identify exactly which checkpoints meet these two criteria.

1. A checkpoint may possibly be used to restart an RU if it is the last checkpoint generated by an RU.
2. A checkpoint may possibly be used to rollback an RU if the checkpoint timestamp indicates that it happened before the last checkpoint of every other RU, and it is the latest checkpoint of this RU with that property.

Suppose we have a system on N RUs. Associated with each recovery unit RU_i is a set of checkpoints $\{C_{i,0}, C_{i,1}, \dots, C_{i,p}\}$. Let $T(C_{k,q})$ be the vector time associated with checkpoint $C_{k,q}$. By definition, $T(C_{i,0}) < T(C_{i,1}) < \dots < T(C_{i,p})$. We can state the two conditions above more formally.

Theorem 7.1 *Checkpoint $C_{k,q}$ is discardable if and only if*

$$\forall l \neq k, \exists C_{k,r}, \exists C_{l,s} \text{ such that } T(C_{k,q}) < T(C_{k,r}) < T(C_{l,s})$$

Proof: To prove the “if” part of the theorem, let us assume that checkpoints $T(C_{k,q})$, $T(C_{k,r})$, $T(C_{l,s})$ exist and that $T(C_{k,q}) < T(C_{k,r}) < T(C_{l,s})$, $l \neq k$. See figure 7.6.

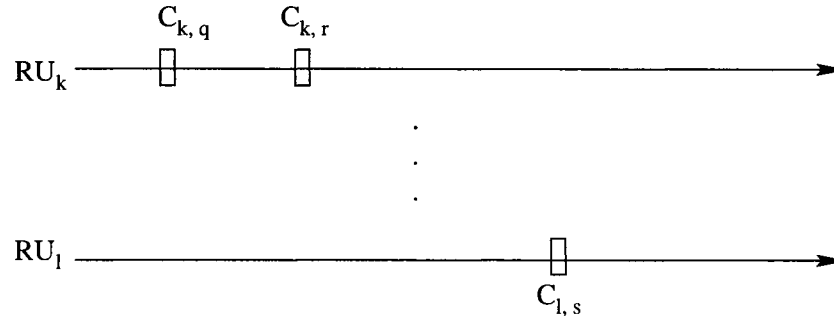


Figure 7.6: Discardability of Checkpoint $C_{k,q}$

If RU_k fails, it will restart from checkpoint $C_{k,r}$, since it is the latest checkpoint taken by the recovery unit. If some other RU_l fails, then RU_k will rollback to $C_{k,r}$. Thus, $C_{k,q}$ will never be used to restart or rollback an RU, and is thus discardable.

To prove the “only if” part, assume that checkpoint $C_{k,q}$ is discardable. Since this checkpoint is discardable, RU_k must have some checkpoint with a later timestamp from which it can restart if it fails. Thus, $\exists r$ such that $T(C_{k,q}) < T(C_{k,r})$. Again, since this checkpoint is discardable, then for every other RU_l , there must exist a checkpoint $T(C_{l,s})$ such that $T(C_{k,r}) < T(C_{l,s})$, since RU_k will be required to restart from $C_{k,r}$. Thus, $T(C_{k,q}) < T(C_{k,r}) < T(C_{l,s})$. \square

7.3.2 Pessimistic and Optimistic Log-Clearing

Theorem 7.1 gives us a predicate that indicates whether a particular checkpoint is discardable. It does not, however, give us an easy way to determine when a particular checkpoint moves from potentially being needed to discardable. The question arises: “How often should

we search the list of checkpoints looking for discardable checkpoints?”

Since each new checkpoint generated by a system of RUs can potentially make one or more checkpoints discardable, one possibility is to search for discardable checkpoints each time any RU generates a new one. We call this type of search for discardable checkpoints *pessimistic log-clearing*.

Another possibility is to search the checkpoint list only after every RU in the system has generated an additional checkpoint. We call this type of search for discardable checkpoints *optimistic log-clearing*.

To evaluate these two methods, we implemented both types of log-clearing in a simple application. Which of the two methods is used is controlled by a compile time parameter.

The application is a simple producer consumer problem. One set of consumer processes generates random 20 by 20 matrices, and writes these matrices to a queue in the file system. Another set of consumer processes removes these matrices from the queue as they become available and computes the square of the matrix.

In our particular experiment, we created two matrix producers and four matrix consumers. Each producer checkpointed when it had produced an additional 75 matrices. Consumers checkpointed each time they had consumed an additional 50 matrices. A total of 5000 matrices were produced and consumed before the application terminated. After each pass of the log-clearing algorithm, we calculated the total size of all checkpoints and log entries in the system.

We ran the application 100 times for both types of log-clearing strategies. Each time the log-clearing algorithm was invoked, we scanned for and deleted discardable checkpoints, calculated the total size of the checkpoints and log, and recorded this value in an external

file. After the 100 executions of the application, we calculated the average size of the log on each pass of the algorithm. Figure 7.7 shows the average size of the checkpoint and log in KB, after each pass of the pessimistic log-clearing algorithm, averaged over 100 executions.

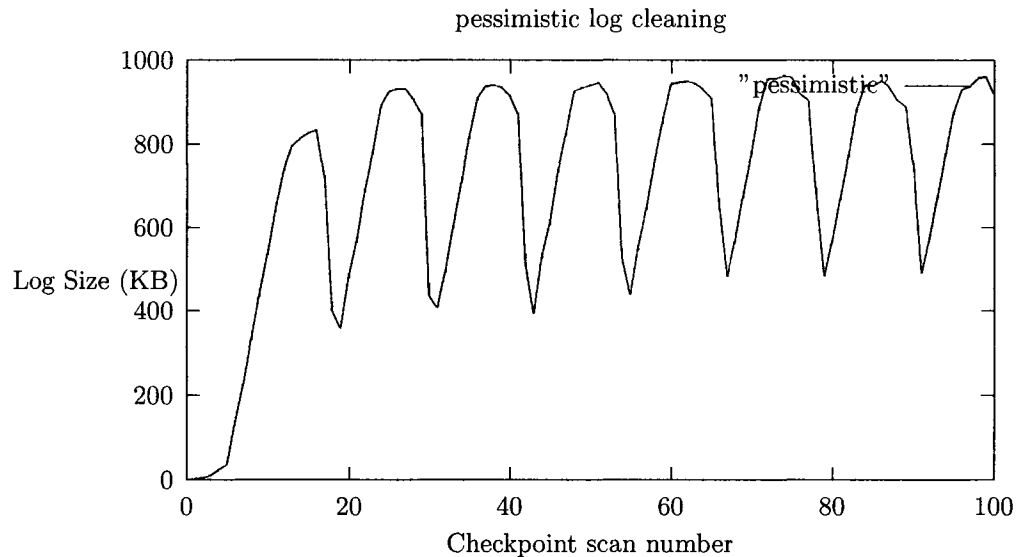


Figure 7.7: Pessimistic Log-Clearing Average Log Size

The “wave” pattern shown in the figure suggests that pessimistic log clearing scans the list too often, since few of the checkpoint scans succeeded in decreasing the memory used by the checkpoint/log list.

Figure 7.8 shows the size of the checkpoint/log list when we scan using optimistic log-clearing, again averaging the total size of the checkpoint and log after each pass over 100 executions.

While optimistic log clearing decreases the number of scans of the checkpoint list, it does so at the expense of the average size of the log.

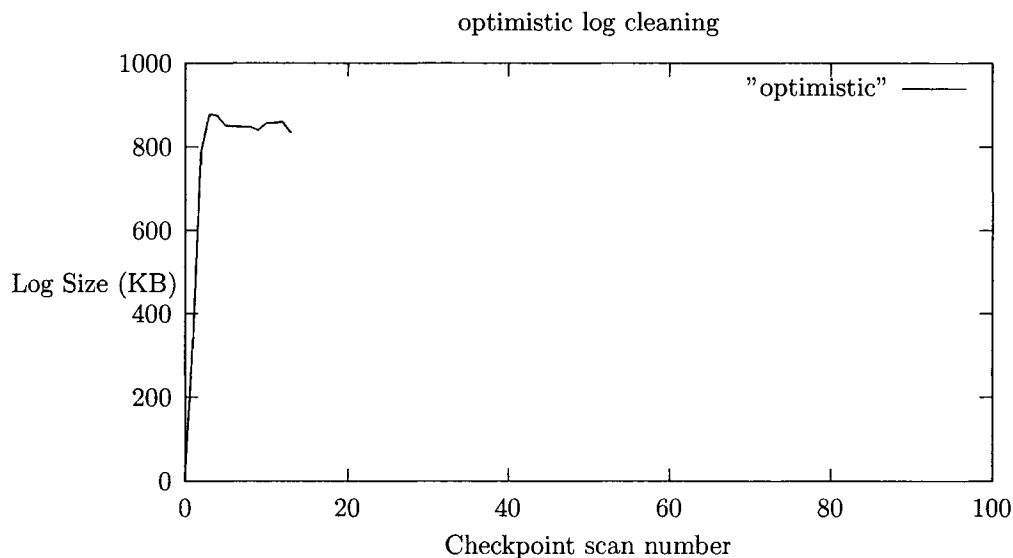


Figure 7.8: Optimistic Log-Clearing Average Log Size

7.3.3 Optimal Log-Clearing

Theorem 7.1 suggests an alternative method. Instead of scanning the list after every new checkpoint is created, or after all RUs have created an additional checkpoint, we run the checkpoint/log clearing algorithm only when we know a checkpoint has become discardable.

Let $\{C_{0,r_0}, C_{1,r_1}, \dots, C_{i,r_i}\}$ be the set of most recently taken checkpoints for all the RUs $\{RU_0, RU_1, \dots, RU_i\}$ in the system. By definition, none of these checkpoints is discardable. However, we can tell exactly when each of these checkpoints will become discardable. This occurs when a particular checkpoint is the oldest in the above set, and the process which owns the oldest checkpoint takes another. By tracking the set of oldest checkpoints, and watching when the RU which generated the oldest checkpoint takes a new checkpoint, we know exactly when a particular checkpoint becomes discardable.

This method has two advantages. First, it tells us exactly which checkpoint is discard-

able, so no other checkpoints need to be inspected. Second, since no other checkpoints need to be inspected, it eliminates the need to scan the entire list of checkpoints.

Figure 7.9 shows the size of the log when this algorithm is executed. To compare these three methods, figure 7.10 shows the results of all three experiments superimposed on the same axes.

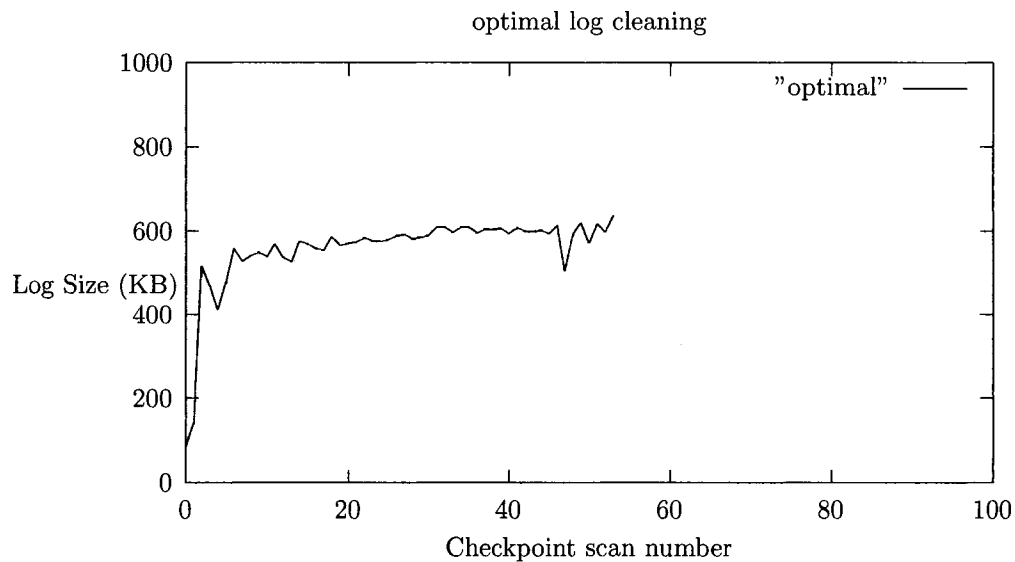


Figure 7.9: Optimal Log Clearing

7.4 Dedicated File Systems for Fault-Tolerance

In the design of our prototype, we have assumed that the underlying file system is a general purpose one that will be used both for fault-tolerant computing, and normal computation not requiring the ability to checkpoint the file system. It is interesting to ask if any advantage could be gained by dedicating a file system to a particular computation that requires fault-tolerance.

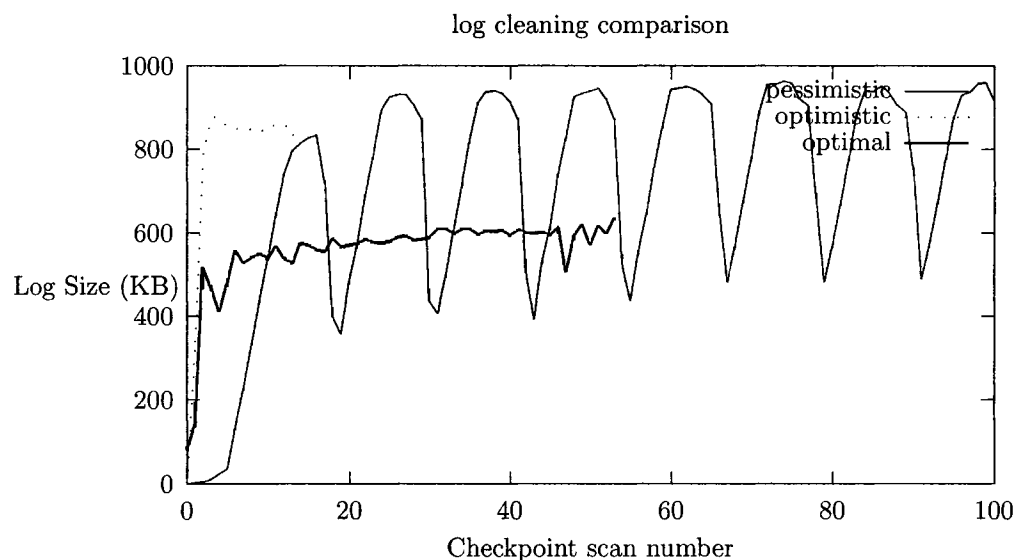


Figure 7.10: A Comparison of Pessimistic, Optimistic and Optimal Log-Clearing Strategies

Imagine a long-running computation with a dedicated fault-tolerant file system, such as the one in our prototype. This file system could be built on a logical partition of a physical disk, or it could have a disk dedicated solely to the computation. In such a setup, the computation would first mount and format the file system. Computation would then proceed as normal. At the end of the computation, the contents of the file system would be discarded. The next long-running computation to use the partition or disk would recreate a virgin file system on the disk for its own use.

Suppose that the programmer who designs such a long-running application is able to calculate the maximum amount of disk space that might be used by the application. This would include not only the space used for files, but also the overhead used by the file system metadata and the multiple copies of disk blocks created by the `sync()` syscall. If a partition or disk of this size was allocated to the computation, such a computation would never have

to reuse disk blocks.

The primary advantage is that a file system running on such a dedicated disk would not need to store a list of allocated and unallocated disk blocks. Since, by assumption, the disk will always have enough free blocks to satisfy allocation requests from the file system for the duration of the computation, a simple pointer could be used to indicate the location of the next free block. The pointer itself indicates the location on the disk separating the set of unallocated blocks from the set of possibly allocated blocks.

Figure 7.11 shows a file system with no freelist. The “free pointer” demarcates the part of the disk containing possibly used disk blocks (before the pointer) from allocatable disk blocks (after the pointer.) The unallocated areas of the disk before the pointer represent data and metadata for files that are not part of any checkpoint. Such files were created and then destroyed between two successive `sync()` operations.

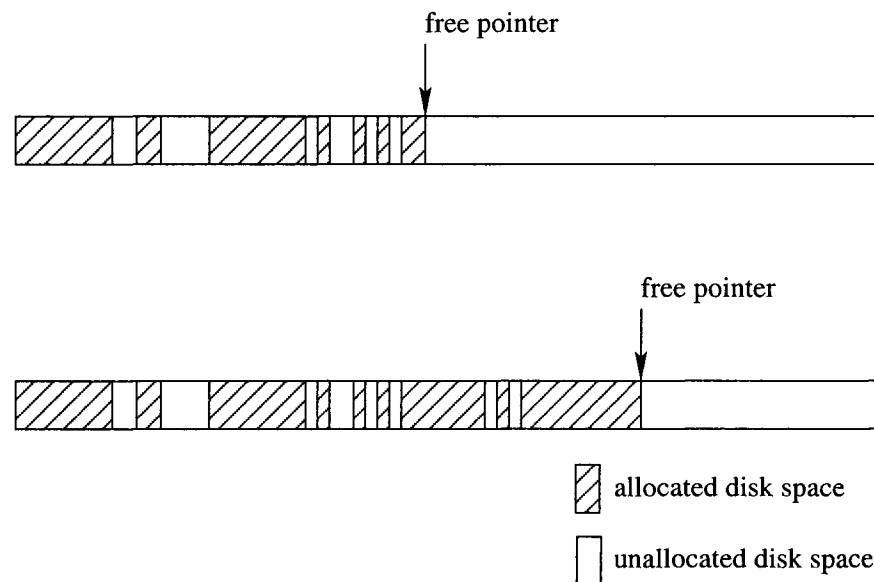


Figure 7.11: A File System with No Freelist

The primary advantage of such a setup is that it greatly reduces the number of blocks which need to be written and read during the `sync()` and `rollback()` operations. As stated in section 7.1, the majority of the costs of these two operations is the number of disk accesses needed to store or retrieve the freelist bitmap from the disk.

If we eliminate the need for a block bitmap by replacing it with a simple pointer, then the pointer can be stored in the checkpoint region during a `sync()`. See figure 5.6. This optimization reduces the cost of both the `sync()` and `rollback()` syscalls to four blocks, greatly reducing the failure-free overhead of the scheme. In addition, this makes both operations work in constant time, regardless of the size of the underlying disk.

Chapter 8

Conclusion

What's done we partly may compute,

But know not what's resisted.

Robert Burns, *Address to the Unco Guid.*

Our work demonstrates that it is possible to efficiently and transparently include files in schemes which use checkpointing and rollback/recovery to provide fault-tolerance, and it demonstrates that the implementation of such a scheme is possible. It includes a working prototype that implements a log-structured file system and the system support necessary to do application-level thread checkpointing, restart, rollback and rollforward.

In particular, our work extends the class of programs which can be run in a fault-tolerant manner to those that include arbitrary file system operations. Unlike earlier techniques, it places no a priori restrictions on the type of file operations that may be performed by the programmer. Users are not restricted to any subset of traditional Unix file operations, nor are file access patterns restricted. In addition, the system does not expend

computational resources on unnecessary file copying.

In addition, it does not limit the type of checkpointing and rollback/recovery schemes that may be used. The scheme can be used with pure checkpointing, optimistic, pessimistic and causal recovery techniques and, indeed, any scheme that uses restart and rollforward through logging to recover system state after a failure.

In addition, we presented a rigorous criterion for bifurcating checkpoints into discardable and non-discardable sets, and adapted that criterion to an algorithm which guarantees space-optimal checkpoint and log clearing.

8.1 Future Directions

There are several questions raised by our research that we believe may be fruitful areas of study in the future.

- Our scheme is a prototype, and we would like to extend it to a production system. Can commercially available log-structured or journaling file systems be modified to use our scheme? If so, what types of modifications are required?
- One important module missing in our file system implementation that keeps it from being applicable to production use is the cleaning daemon. We suspect that a cleaning daemon could be closely coupled with the log-clearing algorithm (see section 7.3.1.) Can we apply the same algorithms to file system cleaning as we do to checkpoint and log cleaning to build an optimal file system cleaner?
- We have already investigated one additional area of application for log-structured file systems[36]. Can LFSs be adapted to fault-tolerance schemes that do not depend

on logging to restore state?

Applications which use a distributed model of computation are becoming more critical to our society's economic well being and security. As concurrent and distributed workloads become commonplace (in both pure distributed implementations, and client-server systems) we believe that fault-tolerance will take on even greater importance. We humbly offer our work to the world.

Appendix A

Sync() Syscall Implementation

```
int vlfs::sync(vector_time *&sync_time)
{
    checkpoint *new_cpr = new checkpoint;

    inode_map *new_imap;

    vector_time *new_vtime;

    // Lock the file system.

    enter_fs(SYS_SYNC);

    // Sync inodes and associated files.  Skip unmodified inodes/files.

    for (i = 0; i < N_INODES; i++) {

        if (i_list[i] && i_list[i]->is_touched()) {

            if ( (inode_block = inode_list[i]->sync()) < 0) {

                cerr << "Error syncing inode " << i << endl;
            }
        }
    }
}
```

```

        return inode_block; // The error
    }

    imap->set_valid_d (i);

    imap->set_entry_d (i, inode_block);

    i_list[i]->clear_touched();

}

}

// Sync imap and make a copy. Old imap on disk must not be modified
// from here on in.

if ( (imap_block = imap->sync(disk)) < 0) {

    cerr << "Error " << imap_block << " syncing imap " << endl;

    return err;

}

*new_imap = new inode_map(*imap);

// Sync vector time and make a copy

if ( (vtime_block = fs_vtime->sync()) < 0) {

    cerr << "Can't sync vector time" << endl;

    return vtime_block;

}

```

```
new_vtime = new vector_time (*fs_vtime);

// Sync checkpoint region

// First, allocate bitmap indirect blocks
for (int i = 0; i < MAX_BITMAP_INDIRECT_BLOCKS; i++) {
    int newblock;
    if ( (newblock = d->alloc_block()) < 0) {
        cerr << "Can't alloc new blocks for bitmap indirect blocks "
             << "during sync" << endl;
        return -1;
    }
    cpr->set_bitmap_block (i, newblock);
}

// Set up pointers to imap and vector time blocks in
// checkpoint region.
cpr->set_imap_d (imap_block);
cpr->set_vtime_d (vtime_block);

// Sync checkpoint region.
if ( (cpr_block = cpr->sync(disk)) < 0) {
    cerr << "Error " << err << " syncing checkpoint region" << endl;
}
```

```
    return err;
}

// Make superblock point to new checkpoint region on disk.
super.set_checkpoint_d (cpr_block);

// OK, we have allocated all the blocks we need to complete the sync.
// Sync the bitmap. The bitmap index is stored in the checkpoint region.
if ( (err = d->bitmap_write(cpr->get_bitmap_blocks_addr())) < 0) {
    cerr << "Error syncing disk bitmap" << endl;
    return err;
}

// Sync super
if ( (err = super.sync(disk)) < 0) {
    cerr << "Error " << err << " syncing superblock" << endl;
    return err;
}

// At this point, the old FS is synced to disk. Setup new in-memory
// checkpoint and hook up to superblock.

// Make new checkpoint the head of the checkpoint list by pointing
```

```
// it to the checkpoint we just wrote to disk.
new_cpr->set_next_checkpoint_d (cpr_block);
new_cpr->set_next_checkpoint_m (cpr);

// New imap is a copy of old, but not yet written to disk.  Imap
// does not hit disk until next sync.
new_cpr->set_imap_d (-1);
new_cpr->set_imap_m (new_imap);

// New vtime is a copy of old, but not yet written to disk.  This
// vtime does not hit disk until next sync.
new_cpr->set_vtime_m (new_vtime);
new_cpr->set_vtime_d (-1);

// Make superblock point to new checkpoint region in memory.  Again,
// this checkpoint region is not yet on disk, and won't be until
// next sync().
super.set_checkpoint_d (-1); super.set_checkpoint_m(new_cpr);

// Make imap point to new imap,
imap = new_imap;

// and checkpoint to new checkpoint.
cpr = new_cpr;
```

```
// Return sync time
sync_time = new vector_time (*new_vtime);

// Update total sync count.
nsyncs++;

// Update number of checkpoints this process has taken.
ps->n_checkpoints_taken++;

return 1;
}
```

Appendix B

The CHECKPOINT() Macro

Because checkpointing is done via a C macro, and C-style macros are limited to one line, each of the following lines (including comment lines) is followed by the C line-continuation character “\” in the actual code. We have omitted that character here for clarity.

We use the `do { ... } while(0);` construct here to allow the macro to declare macro-local variables outside the scope of the calling application.

```
#define CHECKPOINT(fs)

do {

    /* Register storage */
    ucontext_t *context;

    /* Thread-local variable access */
    process_support *current_ps;
```

```

int ppos;

vector_time *sync_time; /* Used for sync */

if ( (ppos = fs.lookup_process(getpid())) == -1)

    cerr << "Can't find psupport struct for process "
          << getpid() << endl;

else {

    current_ps = fs.psupport[ppos];

    /* Don't checkpoint during rollforward or restart */

    if (current_ps->mode == REPLAY)

        continue;

    if (current_ps->get_rollback_flag())

        continue;

    /* Must lock fs here so vtimes are atomic within */
    /* the (setup, sync, vtime) triple. In addition, */
    /* we hold the lock when we return from a */
    /* from a rollback or restart. */

    fs.lock_fs();

    fs.sync (sync_time);

    current_ps->setup_checkpoint (context, sync_time);

```



```
delete sync_time;

fs.cleanup_checkpoint_list (LOG_CLEARING_MODE);

/* The thread magically reappears here after a */
/* rollback or restart. */

getcontext (context);

fs.unlock_fs();

}

} while (0);
```

Bibliography

- [1] ALFRED V. AHO, JOHN E. HOPCROFT, AND JEFFREY D. ULLMAN. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
- [2] SRIDHAR ALAGAR, RAMKI RAJAGOPALAN, AND S. VENKATESAN. Integrating files and processes: A comprehensive approach to checkpointing. In *Proceedings of the Fifth International Conference on Advanced Computing*, pages 453–458, Dec. 1997.
- [3] LORENZO ALVISI. Understanding the message logging paradigm for masking process crashes. Ph.D. Thesis, Cornell University, Dept. of Computer Science, 1996.
- [4] LORENZO ALVISI AND KEITH MARZULLO. Message logging: Pessimistic, optimistic and causal. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, pages 229–236. IEEE Computer Society Press, Jun. 1995.
- [5] THOMAS E. ANDERSON, MICHAEL D. DAHLIN, JEANNA M. NEEFE, DAVID A. PATTERSON, DREW S. ROSELLI, AND RANDOLPH Y. WANG. Serverless network file systems. *Operating Systems Review*, 29(4):109–126, Dec. 1995.
- [6] MARICE J. BACH. *Design of the Unix Operating System*. Prentice Hall PTR, 1987.
- [7] A. BORG, W. BLAU, W. GRAETSCH, F. HERRMANN, AND W. OBERLE. Fault-tolerance under unix. *ACM Transactions on Computer Systems*, 7(1):1–24, Feb. 1989.
- [8] GREG BRONEVETSKY, DANIEL MARQUES, KESHAV PINGALI, AND PAUL STODGHILL. Automated application-level checkpointing of mpi programs. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 84–94. ACM Press, 2003.
- [9] DAVID R. BUTENHOF. *Programming with POSIX Threads*, chapter 3. Addison-Wesley, 1997.
- [10] RÉMY CARD, THEODORE Y. TS’O, AND STEPHEN TWEEDIE. Design and implementation of the second extended filesystem. In *Proceedings of the 1994 Amsterdam Linux Conference*, 1994.
- [11] K. M. CHANDY AND L. LAMPORT. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, Feb. 1985.

- [12] H. M. P. COUCHMAN. Mesh-refined p m : a fast adaptive n-body algorithm. *Astrophysics Journal*, 368(23), 1991.
- [13] EDSEGER W. DIJKSTRA. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, pages 643–644, November 1974.
- [14] EDSEGER W. DIJKSTRA. A belated proof of self-stabilization. *Distrib. Comput.*, 1(1):5–6, 1986.
- [15] ULRICH DREPPER AND INGO MOLNAR. The native posix thread library for linux. Technical report, Red Hat, Inc., 1804 Varsity Dr., Raleigh, NC 27611, Jan. 2003.
- [16] ELMOOTAZBELLAH. N. ELNOZAHY, LORENZO ALVISI, YI-MIN WANG, AND DAVID B. JOHNSON. A survey of rollback-recovery protocols in message passing systems. Technical Report CMU-CS-96-181, Carnegie Mellon University, Oct. 1996.
- [17] ELMOOTAZBELLAH. N. ELNOZAHY, LORENZO ALVISI, YI-MIN WANG, AND DAVID B. JOHNSON. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, 2002.
- [18] ELMOOTAZBELLAH N. ELNOZAHY AND WILLY ZWAENEPOEL. Replicated distributed processes in manetho. In *Proceedings of the 22nd Annual International Conference on Fault-Tolerant Computing*, pages 18–27, 1992.
- [19] R. J. FEIERTAG AND E. I. ORGANICK. The multics input-output system. In *Proceedings of the Third Symposium on Operating System Principles*, pages 35–41. The Association for Computing Machinery, Oct. 1971.
- [20] C. J. FIDGE. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the 11th Australian Computer Science Conference*, pages 56–66, 1988.
- [21] C. J. FIDGE. Logical time in distributed computing systems. *IEEE Computer*, 24(8):28–33, 1991.
- [22] DOMINIC GIAMPAOLO. *Practical File System Design with the Be File System*. Morgan Kaufmann Publishers, Inc., 1999.
- [23] BRIAN GUNTER, W. C. REILEY, AND ROBERT VAN DE GEIJN. Parallel out-of-core cholesky and qr factorizations with poodlapack. In *Proceedings of the 15th International Parallel and Distributed Process Symposium (IPDPS)*, San Francisco, CA, 2001. IEEE Computer Society.
- [24] BRIAN GUNTER, B. D. TAPLEY, AND ROBERT VAN DE GEIJN. Advanced parallel least squares algorithms for grace data processing. In *Proceedings of the 15th International Association of Geodesy (IAG) Conference*, Budapest, Hungary, 2001. International Association of Geodesy.
- [25] D. HITZ, J. LAU, AND M. MALCOLM. File system design for an nfs file server appliance. In *Proceedings of the Winter 1994 USENIX Technical Conference*, pages 235–245, San Francisco, CA, Jan. 1994. The USENIX Association.

- [26] DONALD KNUTH. *The Art of Computer Programming, Volume 3: Sorting and Searching*, chapter 5. Addison-Wesley Publishing, Company, San Francisco, CA, 2nd edition, 1998.
- [27] H. S. M. KRUIJER. Self-stabilization in tree structured systems. *Information Processing Letters*, 8(2):2–79, 1976.
- [28] LESLIE LAMPORT. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):223–228, Jul. 1978.
- [29] LESLIE LAMPORT. 1983 invited address: Solved problems, unsolved problems and non-problems in concurrency. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*, pages 1–11. ACM Press, 1984.
- [30] B. LAMPSON. Atomic transactions. In *Distributed Systems: Architecture and Implementation*, B. Lampson, M. Paul, and H. Seigert, editors, volume 105 of *Lecture Notes in Computer Science*, chapter 11, pages 246–265. Springer-Verlag, 1983.
- [31] TOM LEIGHTON. Tight bounds on the complexity of parallel sorting. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 71–80. ACM Press, 1984.
- [32] CHI-YI LIN, SZU-CHI WANG, AND SY-YEN KUO. An efficient time-based checkpointing protocol for mobile computing systems over mobile ip. *Mob. Netw. Appl.*, 8(6):687–697, 2003.
- [33] MICHAEL J. LITZKOW, MIRON LIVNY, AND MATTHEWS MUTKA. Condor - a hunter of idle workstations. In *The Eighth International Conference on Distributed Computing Systems*, pages 104–111, San Jose, CA, Jun. 1988. IEEE Computer Society Press.
- [34] MICHAEL J. LITZKOW AND MARVIN SOLOMON. Supporting checkpointing and process migration outside the unix kernel. In *Proceedings of the Winter 1992 USENIX Conference*, pages 283–290, San Francisco, CA, 1992. The USENIX Association.
- [35] F. MATTERN. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms: Proceedings of the International Workshop on Parallel and Distributed Algorithms*, Michel Cosnard et al., editors, pages 215–226, Gers, France, Oct. 1988. Elsevier Science Publishers B.V.
- [36] ROBERT MATTHEWS AND PHIL KEARNS. On-line rollback in log-structured file systems. In *Proceedings of the ISCA 16th International Conference on Computer Applications in Industry and Engineering*, pages 11–16. The International Society for Computers and Their Applications - ISCA, 2003.
- [37] JEAN MAYO. Global state predicates in rough real time. Ph.D. Thesis, The College of William and Mary, Dept. of Computer Science, 1997.
- [38] MARSHALL KIRK MCKUSICK, WILLIAM N. JOY, SAMUEL J. LEFFLER, AND ROBERT S. FABRY. A fast file system for unix. *ACM Transactions on Computer Systems*, 2(3):181–197, Aug. 1984.

- [39] ROBERT H. B. NETZER AND JIAN XU. Necessary and sufficient conditions for consistent global snapshots. *IEEE Trans. Parallel and Distributed Systems*, 6(2):165–169, Feb. 1995.
- [40] JOHN K. OUSTERHOUT, ANDREW R. CHERENSON, FREDERICK DOUGLIS, MICHAEL N. NELSON, AND BRENT B. WELCH. The sprite network operating system. *IEEE Computer*, 21(2):23–36, 1988.
- [41] JOHN K. OUSTERHOUT, HERVE DA COSTA, DAVID HARRISON, JOHN A KUNZE, MIKE KUPFER, AND JAMES G. THOMPSON. A trace-driven analysis of the unix 4.2 bsd file system. In *Proceedings of the 10th Symposium on Operating System Principles*, pages 15–24. ACM, 1985.
- [42] JAMES S. PLANK, MICAH BECK, GERRY KINGSLEY, AND KAI LI. Libckpt: Transparent checkpointing under unix. In *Proceedings of the Winter 1995 USENIX Technical Conference*, San Francisco, CA, Jan. 1995. The USENIX Association.
- [43] FRANCESCO QUAGLIA AND ANDREA SANTORO. Ccl v3.0: Multiprogrammed semi-asynchronous checkpoints. In *Proceedings of the seventeenth workshop on Parallel and distributed simulation*, page 21. IEEE Computer Society, 2003.
- [44] B. RANDELL. System structure for software fault tolerance. In *Proceedings of the International Conference on Reliable Software*, pages 437–449, 1975.
- [45] SRIRAM RAO, LORENZO ALVISI, AND HARRICK M. VIN. The cost of recovery in message logging protocols. *IEEE Transactions on Knowledge and Data Engineering*, 12(2):160–173, Mar./Apr. 2000.
- [46] D. M. RITCHIE AND K. THOMPSON. The unix time-sharing system. *Communications of the ACM*, 17(7):365–375, July 1974.
- [47] MENDEL ROSENBLUM. *The Design and Implementation of a Log-Structured File System*, chapter 6. Kluwer Academic Publishers, Norwell, MA, 1995.
- [48] MENDEL ROSENBLUM AND JOHN K. OUSTERHOUT. The design and implementation of a log-structured file system. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, volume 25 of *ACM SIGOPS Operating Systems Review*, pages 1–15. ACM, Oct. 1991.
- [49] D. L. RUSSELL. State restoration in systems of communicating processes. *IEEE Transactions on Software Engineering*, SE-6:193–194, Mar. 1980.
- [50] NAGIZA F. SAMATOVA, AL GEIST, GEORGE OSTROUCHOV, AND ANATOLI V. MELECHKO. Parallel out-of-core algorithm for genome-scale enumeration of metabolic systemic pathways. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 249. IEEE Computer Society, 2002.
- [51] C. H. SCHILLING, D. LETSCHER, AND B. O. PALSSON. Theory for the systemic definition of metabolic pathways and their use in interpreting metabolic function from a pathway-oriented perspective. *The Journal of Theoretical Biology*, 203:286–306, 2000.

- [52] RICHARD D. SCHLICHTING AND FRED B. SCHNEIDER. Fail-stop processors: an approach to designing fault-tolerant computing systems. *ACM Trans. Comput. Syst.*, 1(3):222–238, 1983.
- [53] MARCO SCHNEIDER. Self-stabilization. *ACM Computing Surveys*, 25(1):46–67, Mar. 1993.
- [54] S. SCHUSTER, T. DANDEKAR, AND D. A. FELL. Detection of elementary flux modes in biochemical networks: A promising tool for pathway analysis and metabolic engineering. *Trends in Biotechnology*, (2):53–60, 1999.
- [55] M. SELTZER, K. BOSTIC, M. K. MCKUSICK, AND C. STAELIN. An implementation of a log-structured file system for unix. In *Proceedings of the Winter 1993 USENIX Technical Conference*, pages 307–326, San Francisco, CA, Jan. 1993. The USENIX Association.
- [56] R. E. STROM AND S. A. YEMINI. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, Aug. 1989.
- [57] HY TRAC AND UE-LI PEN. Out-of-core hydrodynamic simulations for cosmological applications. *accepted by New Astronomy*, 2004.
- [58] STEVEN TWEEDY. Journaling the linux ext2fs filesystem. <http://www.kernel.org/pub/linux/kernel/people/sct/ext3/>.
- [59] XIAO-HUI WEI AND JIU-BIN JU. Scr algorithm: Saving/restoring states of file systems. *ACM SIGOPS Bulletin*, 33(1):26–33, Jan. 1999.

VITA

Robert Edwin Matthews

Robert (Bob) Matthews was born and raised in Wapello, Iowa, where he graduated from Wapello High School in 1980. He received a B.A. degree with a major in Mathematics from Simpson College in Indianola, Iowa in 1984.

After serving for two years as a U.S. Peace Corps volunteer in the Republic of Botswana, he enrolled in the Master's degree program at Iowa State University in Ames, Iowa, where he received an M.S. degree in Computer Science in 1989.

He served as an Instructor and Assistant Professor of Computer Science at Armstrong State College (now Armstrong Atlantic State University) in Savannah, Georgia from 1989 to 1995. In 1995, he entered the College of William and Mary as a graduate teaching assistant in the Department of Computer Science.

From 2000 to 2002, he was an engineer at Red Hat, Inc. in Raleigh, North Carolina. Bob defended his dissertation in August of 2004. Currently, he resides in Raleigh where he is an Assistant Professor of Computer Science at St. Augustine's College.