This thesis should be referenced as:

Munro, D.S. “On the Integration of Concurrency, Distribution and Persistence”. Ph.D. Thesis, University of St Andrews (1993).

On the Integration of Concurrency, Distribution and Persistence

David S. Munro

Department of Mathematical and Computational Sciences

University of St Andrews

St Andrews

Fife KY16 9SS

Scotland

Abstract

The principal tenet of the persistence model is that it abstracts over all the physical properties of data such as how long it is stored, where it is stored, how it is stored, what form it is kept in and who is using it. Experience with programming systems which support orthogonal persistence has shown that the simpler semantics and reduced complexity can often lead to a significant reduction in software production costs.

Persistent systems are relatively new and it is not yet clear which of the many models of concurrency and distribution best suit the persistence paradigm. Previous work in this area has tended to build one chosen model into the system which may then only be applicable to a particular set of problems. This thesis challenges the orthodoxy by designing a persistent framework in which all models of concurrency and distribution can be integrated in an add-on fashion.

The provision of such a framework is complicated by a tension between the conceptual ideas of persistence and the intrinsics of concurrency and distribution. The approach taken is to integrate the spectra of concurrency and distribution abstractions into the persistence model in a manner that does not prevent the user from being able to reason about program behaviour.

As examples of the reference model a number of different styles of concurrency and distribution have been designed and incorporated into the persistent programming system Napier88. A detailed treatment of these models and their implementations is given.

Acknowledgements

There are a number of people who have directly and indirectly helped me in the production of this thesis. I would like to give special mention to :-

My supervisor Ron Morrison for his encouragement, good humour, patience and drive in getting me this far. He must also be acknowledged for employing me to do one job then spending eight years persuading me to do something completely different.

Richard Connor for his invaluable contribution in acting as a ‘second’. His enthusiasm, clarity and willingness to listen and suggest have been an inspiration.

Fred Brown for helping me with my first ventures down this path and for showing great patience in explaining the details of his store.

Al Dearle for his enthusiasm, gentle humour and swimming lessons.

Quintin Cutts and Graham Kirby for their contribution in encouraging me to strive for a room of my own.

Colin Allison for holding the fort.

Pete Bailey and Ray Carrick for contributions from the good old days.

Malcolm Atkinson for re-defining the meaning of enthusiasm and always providing reasons to be cheerful.

Some of the PISA project visitors including Dave Stemple, John Rosenberg, Chris Barter and Francis Vaughan.

Other people in the Department including Tony Davie, Dave McNally, Craig Baker and Carl Warren.

Gordon Robertson for getting me out of the Machar and into computing in the first place.

Finally, Erika and Paul for providing a happy home. My Mum for constant support and Day Wishart for always having sound advice.

Contents

1 Introduction

The persistence abstraction is concerned with the uniform treatment of data that is independent of its lifetime. In orthogonally persistent systems all data has the right to survive irrespective of its type. The same mechanisms operate on both short-term and long-term data, avoiding the traditional need for separate systems to control access to data of different degrees of longevity. Thus data may remain under the control of a single persistent programming system for its entire lifetime. The benefits of orthogonal persistence have been described extensively in the literature [ACC81, ABC+83, ABC+84, AM85, AMP86, AB87, Dea87, MBC+87, Wai87, AM88, Dea88, Bro89, MBC+89, Con90, MBC+90, Kir92, Cut92]. These can be summarised as :-

- improving programming productivity from simpler semantics;
- removing ad hoc arrangements for data translation and long term data storage; and
- providing protection mechanisms over the whole environment.

Persistent systems are relatively new and it is not yet clear which of the many models of concurrency and distribution best suit the persistence paradigm. The goal in this thesis is to devise a persistent framework in which all models of concurrency and distribution can be integrated in a manner that preserves understandability. Understandability is a key theme in this work and broadly means that the underlying system model never prevents the user from being able to reason about program behaviour.

1.1 Persistent Object Stores

Persistent systems generally rely on a persistent object store as the sole repository for all data. An object store has a number of desiderata namely infinite

speed, unbounded capacity, and total reliability. In implementation terms none of these is realistically achievable and hence store design is concerned with technological approximations. Concurrency is one way which may effect speed increase whilst distribution can be used as a method to extend store capacity.

The reliability of a persistent store depends in part on its resilience to failures. A number of techniques have been developed that enable a store to recover from system crashes. It is argued here that one method, namely shadow paging, is worthy of investigation in supporting recoverability in orthogonally persistent systems. A single-threaded persistent object store which is based on an efficient implementation of shadow paging is presented. It is shown how this shadow paging mechanism can be extended to support a concurrent store in a way that does not constrain how and when the data is accessed.

1.2 Concurrency

There are many different styles of concurrency ranging from atomic transactions at one extreme to models based on semaphores. Typically application-building systems employ one particular model which may be suitable for a specific set of problems. The approach presented in this thesis is to view models of concurrency as lying on a spectrum of understandability where points on the spectrum define the extent to which the programmer is responsible for maintaining global cohesion. The problem of integrating concurrency and persistence can then be seen as one of incorporating this spectrum into the persistence framework.

A persistent architecture for Napier88 that can support all styles of concurrency has been constructed. The system incorporates Stemple and Morrison's concurrency control specification system CACS system [SM92] into the architecture in a way that links the CACS concepts of data visibility with the persistent store and shadow paging scheme. As an example an atomic transaction

package has been constructed in Napier88 which makes use of the concurrent system.

1.3 Distribution

As the complexity of applications grows the available address space provided by a single node may be insufficient and hence data may be distributed over a number of nodes. Decentralisation of resources can enable sharing among a number of applications and may improve reliability and availability of systems since failure at one node may not necessarily prevent others from continuing.

Distribution may be provided at many levels in a computer system. For example systems such as NFS [SGK+85] provide distribution to the Unix file system whereas other systems such as Amoeba [MRT+90], Accent [RR81] and the Cambridge Distributed Systems [NH82] decentralise the operating system across a number of nodes. One of the design aims of these systems is to attempt to hide the distribution and create the illusion of a single system. In contrast many programming languages provide language features such as RPC [Nel81] or the Ada rendezvous that enable the programmer to exploit distribution. Models of distribution can thus be viewed in terms of the extent to which their use is transparent.

A model of distribution that is completely transparent to the user is undoubtedly the ideal one for orthogonally persistent systems. However there are a number of technological difficulties that make this hard to deliver on a large scale. What is presented here is an extension to the Napier88 system that extends its name space to include other global address spaces. This has been done in such a way as to enable browsing of remote object stores and for objects to be copied from these stores in a type-safe manner. Once copied these objects can then be manipulated just like any other Napier88 object.

At the implementation level communication with a remote object store is achieved through TCP/IP based sockets and therefore any Napier88 persistent object store residing on a machine on the internet can be accessed. The implementation of this service uses an underlying mechanism that is sufficiently generic to allow a number of different distribution models to be constructed. As an illustration of this a model has been constructed which enables Napier88 atomic transactions to take part in a two-phase commit across a number of nodes.

1.4 Integration

In order that the persistence abstraction can deliver its promise of reduced complexity in the construction and evolution of large scale systems and long-lived data, it must address the issues of integrating distribution and concurrency. In [Atk92], Atkinson states that :-

> "Perhaps the most challenging problem is to implement a store that meets the following combination of requirements :-
>
> - manage the various memories and storage devices, allocating and recovering space, giving the illusion of an indefinitely large store;
> - provide stable references (needed to implement all recursive types [Hoa75] and to provide persistent block retention semantics for procedures [Joh71]);
> - provide a reliable store that offers recovery after various kinds of failure [Bro89];
> - provide mechanisms for concurrent use of data; and
> - provide a transactional mechanism to allow programs to voluntarily withdraw updates that they have grouped together and to control their release of revised information."

In unifying the database and programming languages views of data, the persistence abstraction must then provide concurrency and distribution integration in a way that is suitable to both the database domain and the programming language domain. However these two domains often have conflicting views of concurrency and distribution that makes the provision of one single model in the persistence context inappropriate. This then can lead to a tension between the conceptual ideas of orthogonal persistence and the intrinsics of concurrency and distribution that makes the integration difficult. The orthogonal persistence abstraction which hides all physical properties of data may be too strong a constraint for some concurrency and distribution models.

The goal, then, is to find a solution to the integration that preserves *understandability* and confines and controls the relaxation of the orthogonality constraints of persistence to where and when it is required. Most of the previous work in this area has concentrated on the provision of just one model [MBC+88, Wai88, BF89, Bro89]. Instead, this thesis presents an attempt to provide a complete integration by capturing the spectra of concurrency and distribution abstractions in a persistent system. The approach taken is to view the levels of abstraction of concurrency and distribution in terms of the organisational support required in a persistent architecture. By an appropriate provision of language facilities and store primitives a persistent system can be given an infrastructure that enables the construction and support of a range of models.

1.5 Napier88

The Napier88 system [MBC+89] is used as the basis for achieving the integration of concurrency and distribution. Napier88 is a strongly-typed persistent programming language with a sophisticated type system, first-class procedures and environments. Napier88 is supported by a stable persistent object store through which all data is accessed. The use of Napier88 as a vehicle for this work has a number of advantages :-

- The Napier88 system provides no explicit language constructs or store primitives for the expression of concurrency or transactions. The language thus has no preconceived view of concurrency that might complicate the provision of a range of models.

- Similarly, the system has no built-in model of distribution. Since all data in a Napier88 system is through the persistent store then the system is constrained by the bounds of the store.

- The Napier88 system is based on a layered architecture [BDM+90] that was specifically designed to support cost effective experimentation with persistent store design, concurrency, transactions and distribution in a persistent environment. The layered architecture provides an explicit layer for each of the many logical levels of architecture required by a persistent system.

One of the principal design aims in this thesis is to incorporate concurrency and distribution into the Napier88 system in a way that retains these architectural abstractions, augments rather than alters the layered interfaces and does not require the introduction of new language constructs.

The architectural layering has been chosen to take advantage of the persistence abstraction by ensuring that programs are not able to discover details of how objects are stored. This divides the architecture between the architectural layers that provide the persistent object store and those facilities that may be programmed by a supported programming language. Thus, a data format can be altered by the compiler without the need to alter the persistent store. The architectural layering is shown in figure 1.1.

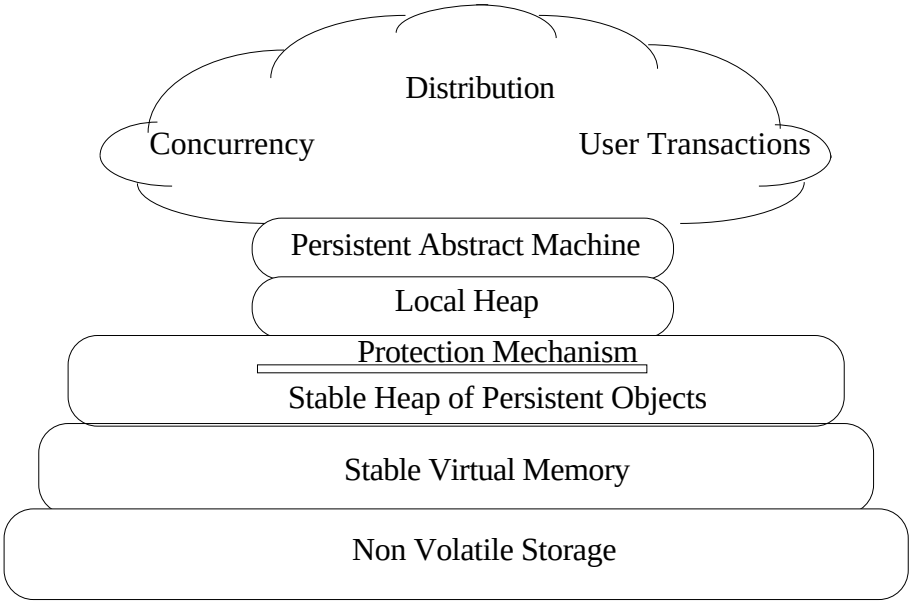


Figure 1.1: The basic architectural layers.

The division has an important consequence for the provision of concurrency, transactions and distribution. This allows experimental implementations to be constructed at the language level without the need to redesign the entire architecture. This is very different from convention where store primitives are provided that define the distribution and concurrency. However, once a particular implementation technique has been identified as essential one or more layers of the persistent store may be reimplemented to incorporate the mechanism. If a layer interface is changed the change is only visible to the layer immediately above thereby limiting the required reimplementation.

The design of this layered architecture and an initial implementation on Unix was produced by Brown [BR91]. Throughout this thesis references are made to this design and its implementation with appropriate detail in the relevant places. A brief summary of the function of each layer is given here as an overview.

The Napier88 compilation system maps programs onto an abstract machine, the Persistent Abstract Machine [CBC+89]. The abstract machine is built on a heap-based architecture that is designed as a convenient way of supporting the block retention needed for the use of first-class procedures and is responsible for

implementing the necessary primitives to support polymorphism and abstract data types. Since the abstract machine does not allow direct access to the persistent store it ensures that the compilation system is unaware of the implementation of object storage, thus separating the use of an object from the way it is stored. A design aim of the architecture provides the persistent object store as the only available storage for the abstract machine. This means that there is only one storage mechanism and one possible way of exhausting it.

To effect efficiency gains the implementation of the abstract machine makes use of a local heap for two reasons. Firstly the local heap is used as a cache of persistent objects that enables the abstract machine to work directly on virtual memory addresses and secondly it provides an area of storage where new objects can be created. Hence one of the local heap's principal functions is to control the movement of data to and from the stable heap. The local heap is constructed in such a way that it can be garbage collected independently of the stable heap and since a great many new objects are transient they can be efficiently collected on the local heap. Objects are faulted into the local heap on demand that causes an address translation, or *swizzle*, that replaces their persistent store address with a local heap address.

The stable heap interface provides a number of persistent object management functions that enable the abstract machine access to the persistent store. The stable heap layer provides a view of the persistent store that appears stable, is conceptually unbounded in size and may be uniformly addressed. These functions include the ability to create and delete objects, a procedure to stabilise the persistent store and a procedure to invoke the garbage collector. The stable heap is designed to work independently of the abstract machine and defines an object format that is not tied to any one programming language. This object format distinguishes object address fields from non-address fields but the stable

heap imposes no interpretation of an object. All objects in the heap are reachable from a single root object.

The stable virtual memory provides a contiguous range of addresses for use by the stable heap that can always be restored after a soft failure to a self-consistent state. Data in the stable virtual memory can be read and written through the interface functions along with a mechanism for establishing a new consistent state. One of the functions of this layer is to maintain a mapping between the stable virtual memory and non-volatile store. The combination of the stable virtual memory and the non-volatile storage layers is often referred to as the *stable store*.

1.6 Related Work

1.6.1 Persistence

The concept of persistence can be traced back to investigations by Atkinson [Atk78] into the integration of databases and programming languages. This led on to the production of the persistent language PS-algol [ACC81] which essentially added persistence to the S-algol [Mor79] programming language.

The PS-algol persistent store has been implemented by several systems, including: the CMS chunk management system [ACC83], which provided a simple transaction mechanism, concurrency control and manipulation of arbitrary sized chunks of data; the POMS persistent object management system [CAB+84]; the CPOMS which is a persistent object manager written in C [BC85].

The CPOMS architecture is a persistent store that is partitioned into databases. Objects are faulted from the persistent store to a local heap on demand. Programs then manipulate these objects, or create new ones, on a local heap and then commit objects back to the store. The commit atomically copies the transitive

closure of the committed object to the store. A pessimistic concurrency control mechanism allows programs to share data.

Much of the implementation techniques used in these systems originated in the development of the System/R [CAB+81] relational database project. The project produced a number of key contributions including the SQL query language, two-phase locking, serializability and shadow paging. System/R uses a combination of shadow paging and logging to support recovery in a concurrent accesses database.

The Shrines implementation [Ros83] of an object store for PS-algol used shadow paging running under the VAX/VMS system. This system operated by mapping a file holding the persistent store onto the virtual address space of a running program by directly manipulating the VMS page tables using a special purpose paging algorithm. The Monads architecture uses a similar shadow paging technique [RHB+90].

Stores that use logging as a basis for recovery include Argus [OLS85], Eos [GAD+92], and O2 [VDD+91]. Other systems such as Cedar [Hag87] incorporate logging as an auxiliary structure in the file system to speed up writes and recovery whilst in the log-structured file system of Rosenblaum and Outerhout [RO91] all data, persistent and transient, is kept in logs. The performance of recovery schemes has been analysed in [KGC85, AD85a, AD85b].

1.6.2 Concurrency

Work on concurrency in databases first identified the notion of atomic transactions [Dav73, Dav78, EGL+76] as a way of isolating the effects of one activity from another using a serializable schedule that preserved understandability. The concept was extended by Moss [Mos81] to enable nesting. In contrast the programming language domain developed constructs

such as semaphores [Dij65] critical regions [Hoa72], monitors [Hoa74], RPC and rendezvous to enable a programmer to design and control concurrency of co-operating processes.

The restrictions enforced by atomic transactions can often suppress potential concurrency and hence overall performance. This is particularly true when transactions are long lived. The more objects a transaction accesses the greater potential for conflict resulting in long blocking delays under a locking mechanism or an expensive abort in an optimistic scheme. The longer a transaction lives the more likely it will either encounter a system crash or incur deadlock thus raising the probability of abort.

One suggestion that has gained some attention has been to use semantic knowledge of an object to increase the amount of concurrency [Gar83]. For certain objects, especially simple ones, it is often possible to identify operations on these objects which are commutative and hence they can be processed in any order. Many such examples exist but is not clear that the method can be adopted in general. An adaptation of this semantic-based concurrency has been proposed by De Francesco et al. [DVM+92]. They suggest that linguistic constructs could be provided in an object-oriented database language such as Nuovo Galileo that permit the programmer to specify mutually commutative methods.

In the Sagas model [GS87] a possible solution to the problems caused by long-lived transactions is given. The serializability constraint is relaxed in Sagas where a long-lived transaction is split into a sequence of transactions that can be interleaved with other transactions. The mechanism guarantees that the transactions comprising the long-lived transaction either all commit or compensation transactions are executed to undo the effects of the partially completed sequence. It is important to stress that this is not a general solution to the problems caused by long-running transactions since not all such transactions can necessarily be broken down to a sequence of shorter ones or compensating

transactions created. The updates of a partial Saga are globally visible and the compensating transactions make no attempt to track other transactions that have seen these uncommitted updates. It is left to the programmer to determine if a long-lived transaction can be partitioned into a sequence of transactions and that compensating transactions can be constructed.

For some applications, especially those in the design and interactive systems [Sut91, EG90, NZ92], the issue is not just one of performance but that the serializability constraint of atomic transaction model is too restrictive. Many such systems require the global cohesiveness of the transaction model but require to interact with each other in a structured way because of their inter-dependence. For example two designers working on part of a large complex design may wish to view each others' changes prior to committing or even commit overlapping changes. In the cases where conflict arises the resolution is done by mutual agreement. Recovery in these *design* transactions can be difficult since there may be complex transaction interdependence. However cascade aborts to effect correctness might be unacceptable since it could eliminate a lot of useful work.

One approach to providing a suitable transaction model for design applications is transaction groups [FZ89, NSZ91, NZ92]. A transaction group is a tree hierarchy of groups whose leaves are co-operating transactions. Co-operating transactions within a group can read and modify the same, possibly uncommitted, objects. This is in contrast to the nested transaction model where serializability is preserved and subtransactions neither communicate nor share data. Each transaction group explicitly defines a sequence of operations called *patterns* that specify the correct execution for members within that group. Any operation or sequence of operations that is prohibited within a pattern is also specified. These are called *conflicts*. Patterns can be thought of as a more expressive form of path expressions [CH74] that can control who performs a sequence of operations as well as when such a sequence can be performed. The recovery mechanism is

designed to restrict the effects from transaction abort or system failure. Only the operations that form the parts of the patterns that are invalidated by the failure are undone and not all the groups' co-operative transactions. Co-operative transactions issue compensating operations to work together to recover. These compensating operations must adhere to the patterns and conflicts definitions for the group and are either coded into the transaction or are formed by user interaction.

ObServer [HZ87] is an object-oriented database that has been used as a base for implementing transaction groups. It facilitates co-operating transactions by providing non-restrictive locks and communication modes that enables non-serializable interleaving. However it was found that within ObServer it was difficult to control the visibility of intermediate results and specify correctness for a history of concurrent transactions. To explore the feasibility of transaction groups in a persistent system Cooper et al [CRW91] produced an implementation using the language DPS-algol [Wai88] which is an extended version of the persistent language PS-algol with support for concurrency and distribution. Their approach was to incorporate the locking mechanisms of ObServer into DPS-algol and use these as a basis for forming transaction groups. Their work included a number of sample applications using co-operative transactions running on different nodes. However, the implementation was not complete in that there was no recovery mechanism nor was there a mechanism for describing patterns and conflicts.

Concurrency in persistent systems has tended to be focused on one chosen model [GAD+92, VDD+91, Wai88, Lis84]. The work of Krablin’s CPS-algol [Kra87] is a notable exception. CPS-algol is an extension to the vanilla PS-algol that includes language constructs to support and manage concurrent processes. The concurrency model is essentially co-operative with procedures executing as separate threads and synchronising through condition critical regions. Krablin

showed that with these primitives and the higher-order functions of PS-algol a range of concurrency abstractions could be constructed including atomic and nested transactions as well as more co-operative models.

1.6.3 Distribution

There are a number of persistent systems which support a form of distribution. The Casper system [KSD+91] is a distributed architecture designed to support a number of Napier88 programs running on distinct nodes against a shared store. The system has been implemented using the multi-threading and external pager features of the Mach operating system [ABB+86] and employs a cache-coherency scheme to ensure consistency. The Monads project [HR91] provides a distributed recoverable shared virtual memory architecture across a network of Monads-PCs. Unlike the Casper model there is no central server and each node provides it own backing store for a portion of the address space. The DPS-algol system [Wai88] is an extension to the PS-algol language to include a model of concurrency and distribution that includes lightweight processes and remote-procedure call. The system enables the PS-algol heap to be distributed across a number of hosts. By default, the distribution is fully transparent but the language provides constructs that may be used to discover and influence where objects reside and processes execute.

The Mneme persistent store [MS88] provides a heap of objects that is designed to support co-operative, concurrent and distributed collections of data. Distribution in Mneme is not just concerned with the physical separation of data but also the decentralisation of the object space management. An object has a unique identifier and is uninterpreted except for distinguishing between pointers and non-pointers for the benefit of the automatic storage management. Within the heap, groups of objects can have particular store management strategies, such as a clustering policy. Two client languages, a persistent Smalltalk and a persistent Modula-3, use the Mneme store.

The Arjuna system [SDP91] is a framework for providing flexible models of distribution through reuse by multiple inheritance. Arjuna is an object based, fault tolerant, distributed programming system. Recoverability and concurrency control mechanisms based on locks are integrated with an object based framework by using inheritance. Objects become active when invoked by an atomic action otherwise they are deemed passive and are stored in a stable object store. Each node has its own stable object store, called Kubera, in which local passive objects are kept. When objects are activated a server process is created and the state of the object is copied into a volatile store. Access to the state of an object is controlled by the object itself. Each Kubera object is kept as a log corresponding to a version history. Versioning of objects is in support of alternative concurrency control mechanisms.

1.7 Summary

The principal aim of this thesis is to provide an architecture in which models of concurrency and distribution can be integrated with persistence. This has resulted in the development of a flexible persistent architecture for Napier88 in which any model of concurrency and distribution can be constructed and supported.

Integrating concurrency into Napier88 is achieved by mirroring the CACS data visibility structures with a concurrent shadow-paging mechanism and developing communication paths between the Napier88 architecture and the CACS system.

Distribution is integrated into Napier88 through the provision of a store-to-store communications interface within a client/server infrastructure that enables models to be constructed.

The development and implementation of this integrated system conforms to the Napier88 generic layered architecture and has been produced without the need to introduce new language constructs.

1.8 Thesis Structure

Chapter 2 examines the issues of the preservation of understandability from the viewpoint of reliability and in particular failure recovery. Two commonly used techniques, namely shadow paging and logging, are described in detail together with a discussion of their appropriateness in persistent systems. Chapter 3 describes the implementation of a new shadow paged stable store that forms the basis of a Napier88 persistent object store.

Chapter 4 discusses the problems of integrating concurrency into a persistent system and presents a new persistent architecture based on a combination of a concurrent shadow-paged store and a concurrency control specification system. The architecture presented has the flexibility to support a range of concurrency styles. Chapter 5 details an implementation of this architecture and shows how two contrasting models of concurrency, a co-operative threads package and atomic transactions, can be incorporated into the Napier88 system and supported by this architecture.

Chapter 6 concentrates on the integration of distribution and persistence. Models of distribution are categorised in terms of their control of the dimensions of distribution transparency. An implementation of one such model that has been constructed in the Napier88 system is presented. It is shown how this model can be simply extended to support a transactional two-phase commit protocol across a number of nodes.

2 Recovery Methods

2.1 Introduction

Central to the persistent architecture is the design of the object store. Many different object stores have been proposed and constructed [ACC83, Ros83, CAB+84, BC85, MS88, Bro89, HR91, GAD+92, BR92]. Because the models that these stores are attempting to support are motivated differently they vary considerably in their functionality and architectural design. However one factor that is common to all these designs is that they must address the desired properties of persistent object stores. These are unbounded capacity, infinite speed and total reliability. In implementation terms none of these are realistically achievable and hence store design is concerned with technological approximations. This accounts for the variety of available stores.

This chapter focuses on the problems of reliability and in particular recovery from failure. Recovery management is concerned with engineering solutions to failure that provide the required degree of reliability by automatically restoring a system to a state that is understandable and acceptable to the user. Failure may occur in a number of different ways. Examples include hardware malfunctions, operating system failure, incorrect computer operation, etc. It is important in designing systems to understand the types of expected failures and their impact on the user.

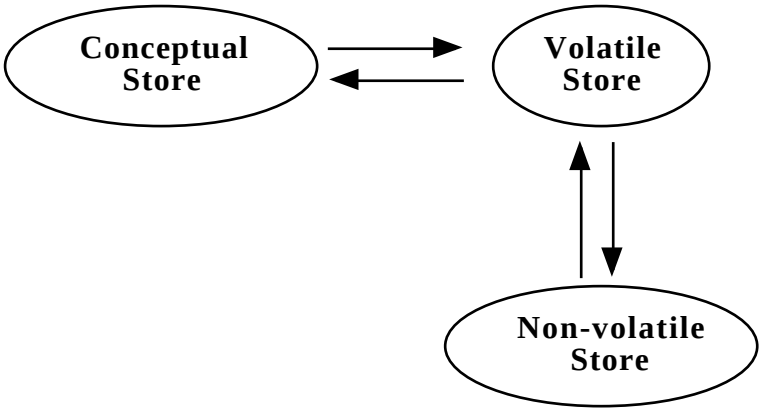


Figure 2.1: Conceptual store architecture

In the discussion in this chapter it is assumed that the user manipulates data through reads and writes to a conceptual store which is implemented on *non-volatile* storage, with *volatile* storage being used as a cache for performance reasons (figure 2.1). The volatile storage in a system is usually main store and cache memory and has the typical feature that the information it holds is lost after a system crash or on power fail. Non-volatile storage is distinguished from volatile storage in that the data it contains is expected to survive system crashes and power failures. One function of the system is to maintain the user's conceptual view of the store by effectively employing a coherency mechanism that ensures the movement of data between the stores is atomic and consistent.

Failure can arise from the loss or corruption of either or both the volatile and non-volatile stores. *Hard failures* occur from the irrevocable breakdown of a hardware component that potentially results in the loss of data from both the non-volatile storage as well as the volatile storage in a system. Disk and tape systems are commonly used as non-volatile storage and their failures are usually the result of a head crash or corruption to the recording medium. Such failures are often called *media* failures. *Soft failures* arise when only the volatile storage is lost or corrupted.

The discussion of recovery techniques in this chapter is restricted to errors that are detectable. Undetectable errors that corrupt either the volatile or non-volatile

storage can conceivably happen. To minimise the potential for undetected errors, hardware often uses error detecting codes, such as cyclic redundancy checks or parity checks [PB61]. Soft failures through power failure are reasonably easily detected. Other soft failures may be detected by the logic of the system software. For example, the system may determine that it is in a state from which it cannot recover or that it is unsafe to proceed and cause itself to crash.

The next section briefly describes some of the issues involved with recovery from hard failures. However the main emphasis in this chapter is to provide background to the issues of recovery techniques that deal with soft failures. A classification of recovery methods is formed and a detailed discussion of common approaches to implementing recovery methods. Comparisons of two popular mechanisms are discussed.

2.2 Recovery from Hard Failure

Recovery from hard failures involves maintaining data on *stable* storage. Stable storage is designed to protect data from hard failures by replicating information held on non-volatile storage to other devices that have an independent failure mode.

Most systems offer an approximation to stable storage through backup or archiving facilities. If the backup media is removable it may be sited in a different location to prevent simultaneous damage to both the original and the copy. The main problem with this method is granularity. The system is only reliable up to the point of its last successful backup. Recovery from hard failure effectively rolls the system back to the time of the last backup. Backups may require that the system is unavailable whilst making the archive and hence may not be done very frequently. Similarly the recovery of data from archive may also be time consuming and frequently requires human intervention.

Systems that require high reliability and availability of data need automatic and fast recovery from hard failure. A technique often used is disk mirroring [BT85] where every update written to disk is mirrored by performing the same update onto a second disk. Thus the second disk acts as a mirror copy of the first until there is a soft or hard failure. Should either disk suffer from hard failure the system can proceed using the information held in the remaining drive. If there is a soft failure a simple protocol is used to determine which drive successfully completed the last update and processing continues from that point.

In the case where both drives fail simultaneously then of course both copies of the data are lost. To circumvent this a third drive could be used in the mirroring and so on. Variations of mirroring include distorted mirrors [SO91] and doubly distorted mirrors [OS93] whilst other techniques such as RAID systems and replicated machines are sometimes used. Clearly absolute reliability is impossible to guarantee but an arbitrarily high degree can be achieved.

An important point about hard failure recovery is that it used to support recovery from soft failures. The techniques used to recover from soft failure described below utilise the non-volatile store under the assumption that it is stable.

2.3 Recovery from Soft Failure

After a soft failure such as a system crash, information held in the volatile store is lost. There is then a potential that the image of the conceptual store on non-volatile storage that survived the failure is in an inconsistent state. This can occur if some of the changes made by the user had not been moved from the volatile to non-volatile storage by the time the failure happened. The problem for the recovery manager is to ensure that there is sufficient information held on non-volatile storage to enable the regeneration of the conceptual store to a consistent state. Soft failure recovery techniques are thus concerned with what information

needs to be kept to allow a consistent state to be reconstituted from the non-volatile store and what the mechanisms are for handling this information.

One simple approach to maintaining synchronisation between the conceptual store and non-volatile store is to cause a change in the volatile store to be moved to non-volatile store before the next change can proceed. In this method the volatile store is no longer being used as a cache and hence the overhead is likely to be prohibitive. Instead several changes may be held in the volatile store and moved to non-volatile store in a batch. This then can lead to a mismatch between the conceptual store and non-volatile store. In order that the user can maintain a level of understandability the system provides the user with a mechanism, which shall be called *meld*, that synchronises the non-volatile store with the conceptual store. The effect of the meld operation is to move atomically, with respect to the conceptual store, a batch of changes from the volatile to non-volatile storage. The term *meld* is used to describe the action of making updates permanent rather than terms like *commit* or *stabilise* since they imply specific meanings in particular models. On recovery from soft failure the manager must be able to restore the conceptual store to the state it was at the previous meld point from the non-volatile store and proceed from there.

The atomicity of the meld with respect to the conceptual store assures the user that either all the changes appear to have reached non-volatile store or none have. In the provision of an atomic meld the recovery manager must take into account that the interface to the non-volatile store is non-atomic and hence writing out a batch of changes cannot be performed in a single action. A soft failure in the middle of a meld operation could potentially leave the non-volatile store in an inconsistent state. To circumvent this the recovery manager employs a controlled replication of data on non-volatile storage that ensures that after soft failure the old values for data that have been changed can be found or reconstructed or that values for changed data that should have occurred as part of the atomic meld can

be found or reconstituted. There are two procedures the recovery manager can perform to rebuild a consistent image of the conceptual store. After a soft failure the recovery system can **undo** the changes made to data in the non-volatile store by replacing them with their previous melded value. An undo operation requires that the previous melded value for some piece of data in the conceptual store has been copied to non-volatile store before being overwritten. The recovery system can also undertake a **redo** action on data in non-volatile storage so that it reflects the values it would have acquired had the failure not occurred. A redo operation requires that all the changes the user made to the conceptual store have been copied to non-volatile storage.

If the system were to crash again whilst in the middle of recovering from the last crash then the recovery manager must be able to replay the sequence of undo and redo operations that it was in the process of completing. This implies that the replay must be *idempotent*. That is, repeated executions of the replay mechanism must have the same effect as if it were executed once. Recovery strategies can be classified into four categories:-

- **no undo / redo**. The system is constructed in such a way that the recovery manager is never required to perform undo operations after a crash. This can be achieved by delaying changes to the non-volatile store until the user requires a new consistent state to be established. The changes are then applied to the store. On recovery, redo operations are required to update the non-volatile storage to make it consistent with the user's view of the store.

- **undo / no redo**. To avoid redo operations the system must ensure that all changes made by the user are recorded in non-volatile store. On recovery these changes must be undone to the last point where the non-volatile store and the conceptual store were synchronised.

- **undo / redo**. After a system crash the recovery manager may be required to perform undo operations for some data and redo operations for others to restore a consistent state.

- **no undo / no redo** To avoid redo all the changes made by the user must be on non-volatile storage by the time of the meld. To avoid undo none of the updates can be on stable storage before the meld. To avoid both requires that all the updates to the conceptual store are made permanent in one single atomic action.

The next two sections look at how each of these categories may be realised in an implementation. For the purpose of describing the recovery mechanisms a single-threaded operation is assumed. The additional problems incurred by concurrent access and possible optimisations to the recovery mechanisms are considered in section 2.6.

In many systems the recovery manager considers data as separate from the program's state in that no provision is made for restoring the program state following a soft failure. Because no program state is maintained then after a crash a program can only be restarted from the beginning of its execution. To ensure consistency the recovery manager may need to restore the data to the state before the program started. However this may have the adverse effect of eliminating a lot of useful work.

In some orthogonally persistent systems the process state is considered part of the data and hence the recovery manager must ensure that both the data and process state are synchronised so that they can be uniformly and consistently restored after a soft failure. The recovery manager can then restore both the program and the data to some point in the computation and continue from that point. The point is that systems that consider the process state as part of the store

can often restore a system after a crash to a point nearer the crash than would otherwise be possible thereby reducing the potential loss of information.

Obviously simple programs can be structured to preserve enough state information that allows the computation to proceed from some intermediate point. For example the user of an editor or word processor is often encouraged to make periodic saves so that in the event of a system crash the editing can proceed from the last save point. This save may not include all aspects of the editing environment so that on restarting the editor items such as the insertion point or the contents of the copy/paste buffer may be lost. This method of recovery is clearly not general and can add considerable complexity to a program.

The difference in effect on recovery between systems that save the program and data state to those that don't is not unlike the difference between the *restart* and the *sleep* commands on an Apple Macintosh PowerBook. The *restart* first quits all open applications usually giving the user an opportunity to save any changed data and then reboots the system. On startup applications are not restored to the state before the *restart* and the user has to recreate the environment that existed before. In contrast the *sleep* command effectively shuts down the machine but preserves the current state so that when awoken the user can immediately continue from the point just before the *sleep* was issued.

2.4 Logging

Logging [Dav73] is the most widely used recovery method especially in transaction processing systems. A log is held on stable storage and constitutes a journal of changes made to the conceptual store. In this discussion it is assumed that the log survives a crash and that writes to the log are stable. Before any object is updated a record is appended to the log that records the change being made. This *writeahead* log [GMB+81] can then be used after a crash to restore

the system to a known state by comparing the values for objects held in the log with values in the non-volatile store. Figure 2.2 gives an outline of the store architecture showing the log in non-volatile storage and pages copied between the non-volatile and volatile storage. Logging is usually implemented using one of two basic schemes described below.

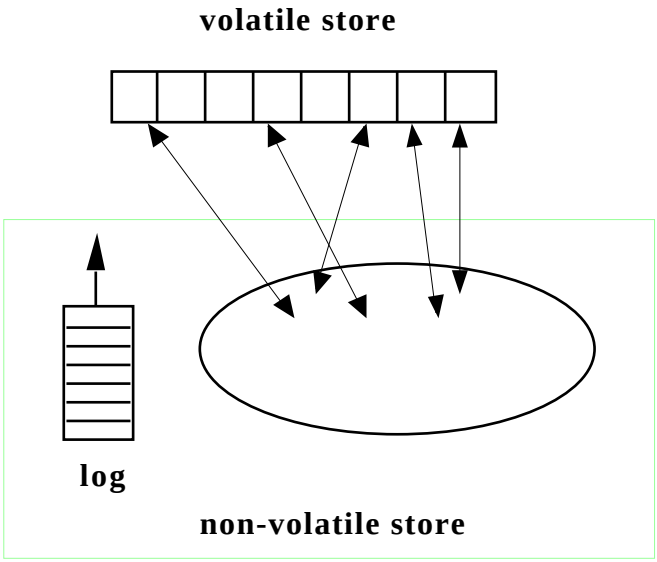


Figure 2.2: Layout of logging model

2.4.1 Writeahead Log with Deferred Updates

Updates made by the user are recorded in the log but the changes to the non-volatile store are deferred until the user melds. Each update causes a log record to be written that specifies the object involved in the update and its new value. When the user melds an entry recording the fact is written in the log. At this point the data is effectively melded. Following this the deferred updated objects can be copied to non-volatile storage by reading the changes from the log.

If the system crashes after the user has melded but before the updates to the non-volatile store have completed the log is used to **redo** the changes made by the user. Thus on recovery the information in the log can be used to restore the system to the last consistent state.

If the system crashes before the user melded then because of the deferred writes the non-volatile store still reflects the last consistent state. If the program state is part of the conceptual store then it can be automatically restored from the log by the recovery manager and its execution restarted. The program may continue execution from this point and will then effectively reconstruct the conceptual store to the state just prior to the crash and then carry on its computation. If the program state is not in the store then the recovery manager is not required since the state of the non-volatile store always reflects the state at the last consistent state. This is identical to the situation where the user discards the changes.

Writeahead logging with deferred writes can be classified as using a **no undo / redo** algorithm.

2.4.2 Writeahead Log with Immediate Updates

Before the user updates an object in main memory a record of the change is written to the log. The log entry records the object involved in the update, its new value and its old value. The update of the object can then take place. Unlike the previous case updates can be performed straight after the log record has been written and are not deferred until the user melds. Since the log records contain the old and new values for modified objects the pages of the volatile store can be flushed to non-volatile storage at any time. When the user melds a record is entered in the log.

Since the flushing of modified pages can be done independently of the user's updates and melds the log must be used after a system crash to ensure that the non-volatile storage is consistent with the conceptual state of the store. If the crash occurred after a modified page was written to non-volatile storage but before the user melded, the records in the log are used to **undo** the effects of the updates. If the crash occurred after a meld record was written to the log but before all the modified pages pertaining to that meld were flushed, the log

records are used to **redo** the effects. If the user discards then the log is again used to undo the effects of the updates.

Again if the program state is part of the store then after the recovery manager has completed its undo operation the execution of the program can be automatically restarted by redoing the log.

Writeahead logging with immediate updates can be classified as using an **undo / redo** algorithm.

2.5 Shadow Paging

In a shadow paging system a page replacement algorithm controls the movement of pages between non-volatile store and main store in such a way that a consistent state can be recovered from the non-volatile store after a crash. To effect this the system maintains a disk page table that records the mappings between the pages in the virtual address space and their associated blocks in non-volatile store. The first time the user modifies a page a shadow copy of the page is created so that there is always a retrievable copy of the page as it was before the modification in non-volatile storage. The system must also record that a modified page has been shadowed to avoid another shadow copy being created. Shadow paging employs a meld mechanism that atomically establishes a new global consistent state. This is executed when the user melds. There are two varieties of shadow paging :-

2.5.1 After-look Shadow Paging

With an after-look shadow paged scheme the mechanism makes sure that a modified page is never written back to non-volatile store to the same place it was read from. When a modified page is written back to non-volatile store an unused disk block is found and the disk page table updated to reflect the new mapping.

This is analogous to deferred-write logging. Figure 2.3 illustrates the after-look scheme showing the modified pages being shadowed to a different disk page.

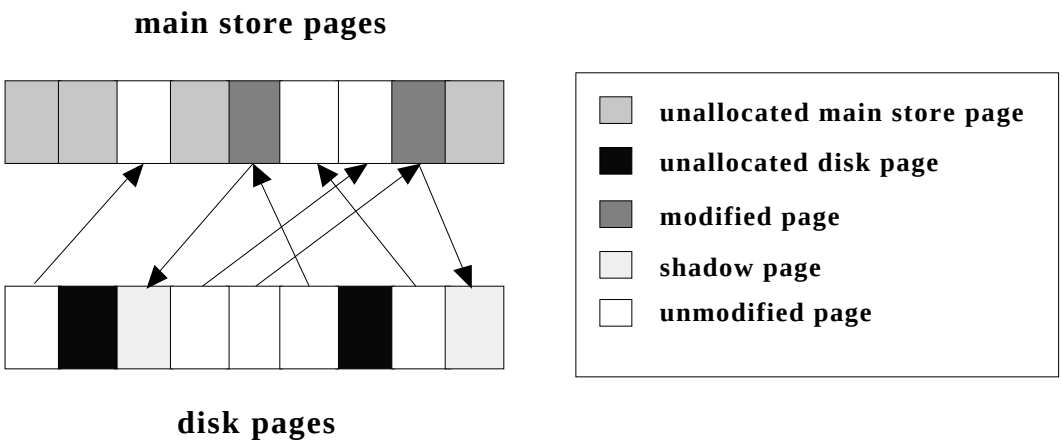


Figure 2.3: After-look shadow paging

The system uses a root block which resides at a known disk address. The root block is stable by mirroring and from this the disk page table can be located. At system startup the disk page table is interrogated to re-establish the state of the address space. In fact two versions of the disk page table are maintained; the version held on disk which reflects the stable state and another in main memory which is updated by the shadowing mechanism. This *transient* disk page table reflects the current state of the address space. Figure 2.4 illustrates the architecture. The stable disk page table records the mappings from the last consistent state whilst the transient disk page table in volatile store records the current mappings. The diagram shows that the third and fourth page have been shadowed to unused disk blocks. When the user melds all the modified pages are flushed to disk and then the in-memory version of the disk page table atomically replaces the disk version. This atomic update can be performed using an adaptation of Challis' algorithm [Cha78].

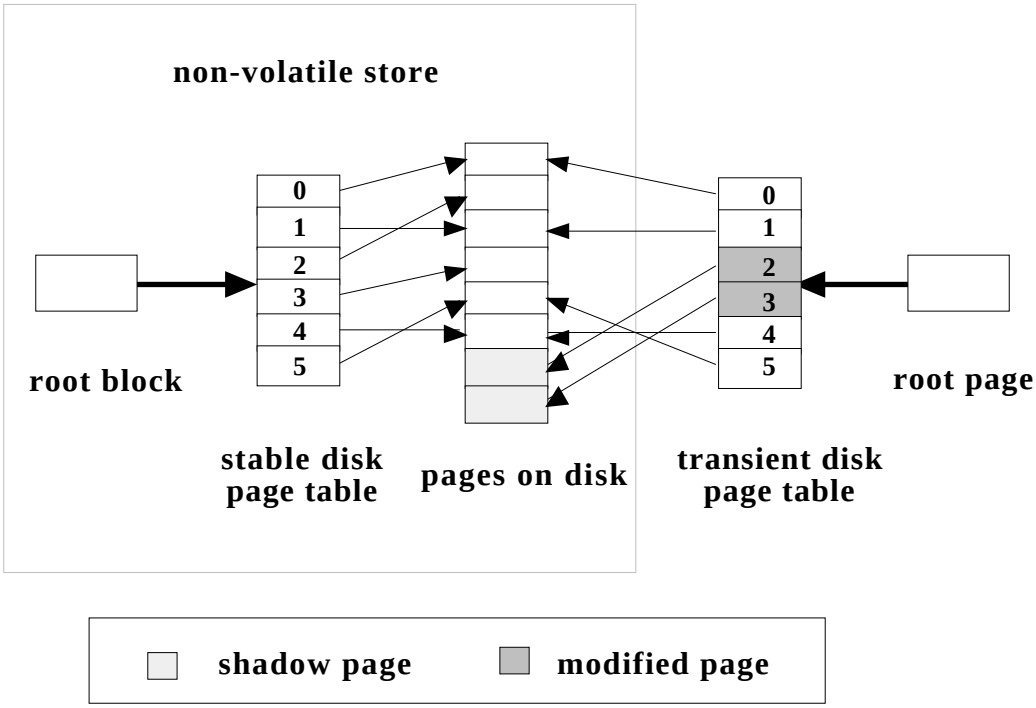


Figure 2.4: Layout of after-look shadow paging

On crash recovery the root block is read and the disk page table is recovered. This is used to re-establish the contents of the pages from their associated disk blocks. It will include the program state if it is considered part of the data. If this is the case then once the recovery manager has reconstructed the data the computation will automatically carry on from the last meld point. No changes to data in this mechanism get undone or rewritten and hence the after-look shadow paging recovery algorithm can be classified as **no undo / no redo**.

2.5.2 Before-look Shadow Paging

With before-look shadow paging, the first modification to a page causes a copy to be written to a new block on non-volatile store, i.e., its shadow page. In contrast to after-look shadow paging modifications then take place in the original. The disk page table is used to record the location of the shadow pages and must itself be on non-volatile store before any updates reach non-volatile store. This is similar to logging with immediate writes. The before-look scheme

is illustrated in figure 2.5 where a modified page is written back in place after a shadow copy of the original has been taken.

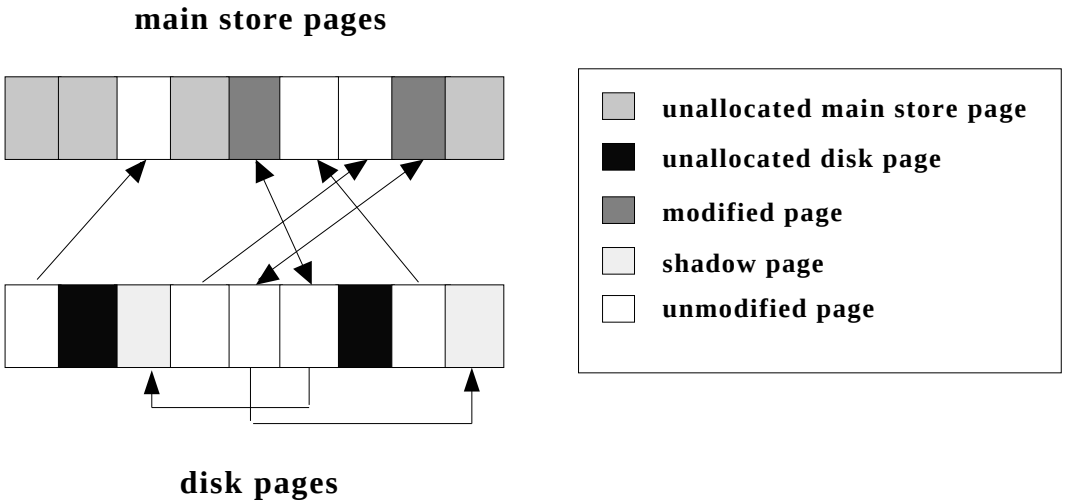


Figure 2.5: Before-look shadow paging

These shadow pages must be locatable after a crash and effectively form a coarse-grain log of the previous values of the modified pages. On meld the modified pages are written to disk and a new consistent state is established. This log of previous values is then discarded.

Recovery from a system crash occurring before a meld involves using this “log” to overwrite the pages that have been modified since the last meld with their previous values. The system is thereby established to the same state as it was at the last meld. This undo of the pages is clearly idempotent. If the program state is considered part of the data then once the recovery manager has completed the undo operation the computation will automatically proceed from the last meld point. Before-look shadow paging can be classified as having **undo / no redo** semantics.

2.5.3 Shadowing using Objects

Some systems adopt an alternate approach to shadowing in that they record different versions of an object rather that different versions of a page. The architecture for such a system could be constructed as a stable heap of objects

that is implemented on non-volatile store using main memory as a cache. Instead of maintaining page tables a mechanism is required that maps versions of an object. One possible approach is to reference all objects through object headers of the form shown in figure 2.6 as in Argus [OLS85]. When an object is first modified a shadow copy of it is created and used as the current version. All further modifications to the object affect the current version. When the user melds the current versions of all modified objects reachable from some root object it are written out to non-volatile storage and the pointers to the old versions updated atomically to point to the current versions. Thus the old version is always the value of the object at the last meld.

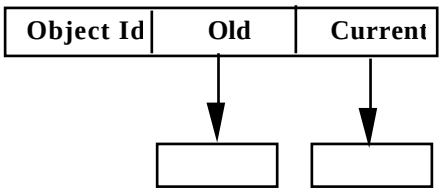


Figure 2.6: Object header format

The main attraction of shadowing using objects is the granularity of the shadow copy. Only objects that are modified require shadow copies rather than a whole page. This will particularly benefit programs that only modify a small number of objects. However creating shadow copies of objects could lead to problems of external fragmentation. The trade-offs of shadowing objects rather than pages are therefore similar to the arguments of segmentation versus paging. However hardware and operating systems support for paging is available in many systems which may favour a shadow paging solution.

2.6 Optimisations

The potential costs of maintaining a recoverable system can be high. The running costs involve space overhead either for the log records or for the shadow pages and speed overhead in writing these to non-volatile store. There is also the overhead involved in the time it takes to recover after a crash. There are a

number of possible optimisations that can help alleviate these problems without compromising the recoverability of the system. In general non-volatile store is implemented using hard disk technology and the optimisations centre around reducing the frequency of disk writes and minimising seek times. Other optimisations strategies can be used to aid the space overhead.

2.6.1 Optimisations to Logging

One of the perceived benefits of logging is that log maintenance involves only sequential disk writes to the end of the log. One obvious optimisation is to buffer log records and only force the log to disk when either it is essential to maintain recoverability or more eagerly when there is a block (or a number of blocks) of records to be written. Of course such buffering will compete with the store for main memory resources. A second consideration is to determine exactly when it is essential that the log records are on non-volatile storage. The stable log, that is the log on non-volatile storage, must contain a record of an update to an object only when the non-volatile store no longer contains the last melded version of the object. When using deferred write logging the last melded versions of changed objects are not overwritten until the meld record has been written to the stable log. Hence it is possible to also defer the writes to the stable log until the user melds as long as the buffers are large enough.

This approach will not work when using logging with immediate writes. In this case the mechanism must ensure that if the log records are buffered then the updates they record do not overwrite the melded versions in non-volatile store. Conversely if updates are flushed to non-volatile store then the log records associated with these updates must be on the stable log beforehand.

One important feature of logging is that there is no requirement at any time to force modifications made by the user to non-volatile store providing the stable log has recorded the changes. A user can modify and meld changes to a group of

objects several times without it being necessary to write these changes back to non-volatile store. All the necessary recovery information has been recorded in the log.

When using writeahead logging with deferred writes the update to an object can be prevented from reaching non-volatile store by pinning the page it resides on in main memory. After a meld these pages are unpinned and flushed to non-volatile store. A problem arises if the main memory becomes exhausted before a meld. The page-replacement mechanism must select a page and unpin it and write it back to non-volatile storage. However it must write it back to a different location on non-volatile store to avoid overwriting the last melded version. This then requires that the mechanism maintains a page table that determines where a page is mapped to.

One optimisation that is sometimes used in logging implementations is for the log to record differences, or *deltas*, between the old value and the new value of a modified object rather than the values themselves. This can reduce the amount of information written to the log especially where there are a frequent number of small changes to large objects.

2.6.2 Checkpointing

It should be clear from the above descriptions that the log size is unbounded. Unfortunately stable storage isn't and hence a mechanism is required that enables the log size to be reduced. Furthermore a large log could have an adverse effect on recovery time. To alleviate the problem periodic *checkpoints* are taken. A checkpoint operation synchronises the state of the conceptual store with the state of the stable log. This involves flushing any buffered log records and all modifications made by the user to non-volatile storage and then writing a checkpoint record to the log. This record logs if any unmelded changes have been made by the user at the time of the checkpoint.

A checkpoint enables any records associated with changes that were melded prior to the checkpoint to be removed from the log thus reducing its overall size. However retrieving the free space in the log may be a very expensive operation and in practice rewriting or compacting the log is slow and not frequently performed and hence the size of a log is typically quite large [Kol87].

Checkpointing reduces the amount of the log that must be searched on crash recovery.

Recovery from a system crash should only now involve searching the log up to the checkpoint record and undoing or redoing changes as necessary. There are 5 separate cases to be considered as illustrated in figure 2.7.

The first case, C1, is where the user made some changes that were melded before the checkpoint. No action is required by the recovery mechanism since the state of the non-volatile store has not changed since the checkpoint.

Case C2 shows that the user made some changes before and possibly after the checkpoint and melded these changes before the crash. The recovery manager must use the log information to redo the changes made by C2. Note though that it is only necessary to restore from the checkpoint forward since any changes prior to the checkpoint are still in effect.

Case C3 is where the user made changes prior to the checkpoint but the system crashed before these changes were melded. In this case the effects of C3 must be undone including the changes C3 made before the checkpoint. It may then safely be rolled forward.

In case C4 changes were made after the checkpoint and melded before the crash. Again the log records are used to restore the changes made by C4.

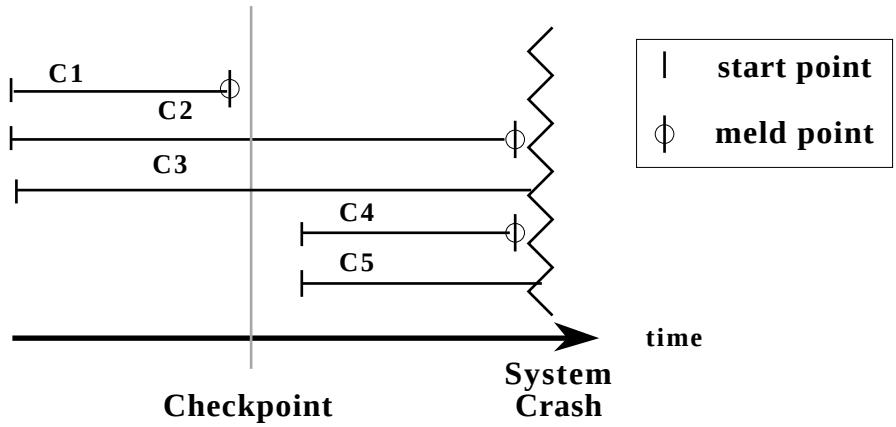


Figure 2.7: Checkpointing and recovery

Finally case C5 made changes after the checkpoint which were not melded before the system crash. Here the effects of C5 must be undone before they can safely be redone. Note that this is only for the case involving logging with immediate update. With deferred update any changes made by C5 would not have reached the non-volatile store and hence do not need to be undone.

If the program's state is considered part of the data then the above scenario is somewhat different. The checkpoint operation would include synchronising the program state with the data and hence this state could be re-established after a crash. In case C2 the changes made after the checkpoint would be automatically re-established by the program resuming computation from the checkpoint. The redo information would not need to be read from the log. However in cases where a lot of computation was involved between the checkpoint and crash point it may be cheaper to restore from the log. The is also true for case C3. Instead of undoing all its effects as described above C3 would be resumed from the checkpoint state and move forward automatically.

Note that in case C5 logging the program state would mean that after the undo operation had completed the program would automatically restart from the beginning and move forward. Logging of program state may be a fairly expensive operation given that each state change would require a new log record.

One cheaper alternative is to only log program state at a checkpoint. This would not deliver all the benefits that comes with logging every state change but could save some changes from being discarded after a crash. In the cases shown checkpointing the program state would allow case C3 to be redone rather than undone but could do nothing for case C5 since no log information about its program state had been taken.

2.6.3 Optimisations to Shadow Paging

With shadow paging, overheads are incurred by the necessity of maintaining a disk page table. The page table is frequently accessed and updated for every new shadow. One optimisation is to ensure that the page table is permanently resident in main memory where this is feasible. One problem that may arise is that with a very large store the disk page table may be too large to hold in main memory. A simple solution is to include the page table in the virtual address space. This then means that the page table itself is paged and modifications to the table will create shadow copies. One consequence of this is that it also allows the pages of the disk page table to be mapped to anywhere on disk. However there is still a requirement for a fixed point so that the disk page table can be located at system startup and on recovery. This can be accomplished by maintaining a secondary disk page table at a known location that records the mappings of the pages of the page table.

Significant performance increases can often be achieved by incorporating the shadow paging mechanism into the operating system's own paging scheme for virtual memory. However this kind of facility is not necessarily available on all operating systems but is becoming more widespread. For example the Mach [ABB+86] operating system allows access to the paging mechanisms through an external pager whilst SunOS [Sun90] and earlier versions of VAX/VMS [Dec78] permit the user to memory-map portions of the file system. Whilst memory-mapping in these systems does not provide access to the paging mechanisms it

can aid performance by providing hardware assist in address translations. Memory-mapping and its benefits are described in more detail in section 3.3.2.

With single-threaded computation checkpointing in a shadow paged store is identical to the action taken when the user melds.

2.6.4 Optimisations to After-look Shadow Paging

One problem with after-look shadow paging is that two logically adjacent pages in the conceptual store may be allocated physically distributed shadow pages causing increased seek time. The effect of this can be reduced by using physical clustering techniques such as suggested by Lorie [Lor77] where shadow pages are allocated within a cylinder where possible. With after-looks, shadow pages need only be allocated disk blocks when the user melds or when main memory is exhausted and pages must be written back. Hence this provides an opportunity to use such a clustering scheme.

2.6.5 Optimisations to Before-look Shadow Paging

One of the advantages of before-look shadow paging is that the updates occur in place and hence the logically adjacent pages will also be physically adjacent on non-volatile store. Optimisations to before-look shadow paging are similar to logging optimisations. When a page is first modified a shadow copy in main memory can be taken instead of a shadow copy on non-volatile store. The user can then make changes in place. It is only necessary to write the shadow page to non-volatile store when these changes are about to be written to non-volatile store. Further, it is only necessary to write back changes when the user melds or when main memory is exhausted and pages must be written back. Note that the disk page table which in before-looks records the location of the shadow pages must be recoverable before any modifications reach non-volatile store. The before-look mechanism thus requires more disk writes than the after-look

mechanism since the original page and the “log” must be on disk before any modifications are written back.

2.7 Concurrency

What impact does the introduction of concurrent operations on the conceptual store have on the recovery manager? If the model of concurrency is one of co-operation where all the processes or threads synchronise their activity then recovery will work as described above. The processes all agree on the state of the conceptual store and hence they agree on the state of the store after recovery from system crash.

In the case where the model is one of conflict concurrency the processes do not commonly agree on the state of the store. Recovery therefore involves restoring the conceptual store so it contains the effects of any individual processes that melded and does not contain the effects of any unmelded processes. To describe the problems associated with recovery from system crash in a concurrent system a model of atomic transactions is assumed. Recovery mechanisms for non-serializable transaction systems are very dependent on the particular concurrency model.

2.7.1 Concurrency and Logging

The logging techniques described naturally extend to handle concurrency atomic transactions. In fact the logging technique was designed for precisely this model. Since the log records a journal of changes all that is required is to add a transaction id to each log record.

Crash recovery and checkpointing uses the same procedure as described above. In the example given in figure 2.8 this would involve redoing transactions T2 and T4 and undoing T3 and T5. Note that T1 does not need restored from the log since it melded before the checkpoint. Also the redo of T2 need only be

performed from the point of the checkpoint forward. Any changes made by T2 prior to the checkpoint will still be in effect.

If the program state is logged then the same advantages that were described for the single-threaded case apply. That is, that transaction T3 would automatically be resumed from the checkpoint state and move forward. Similarly T5 would first have its changes undone and then move forward automatically.

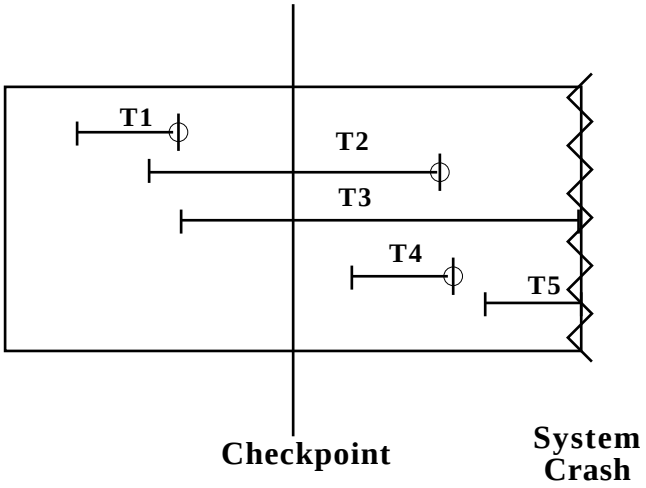


Figure 2.8: Checkpointing and transaction recovery

There are many possible algorithms for recovery using checkpoints. The following is the one described by Bacon [Bac92]. The recovery system after a crash constructs a *undo_list* and a *redo_list.* The log is searched backwards for the last checkpoint record. The checkpoint record in this algorithm is assumed to contain the identity of all transactions that were active at the time of the checkpoint. These transactions are added to the *undo_list*. The log is then read forwards from the checkpoint record to the end of the log. Each start record found for a new transaction adds that transaction to the *undo_list*. Each meld record found moves the transaction from the *undo_list* to the *redo_list*. The log is then read backwards undoing each transaction in the *undo_list* and finally read forwards from the checkpoint record to the end redoing transactions on the *redo_list*.

2.7.2 Concurrency and Shadow Paging

In contrast shadow paging does not readily extend to accommodate concurrent operation. The problem is that transactions may make conflicting requests to modify the same page. When one of the transactions melds the modified page is written to its shadow page on non-volatile store. This of course will include the changes made by any unmelded transaction that modified objects on the same page.

One solution is to use page-level locking whereby an atomic transaction obtains an exclusive lock on a page before shadowing the page [AD85b, Lor77]. Each transaction maintains its own page table of shadows and the locking guarantees that a page is never in more than one page table. The main drawbacks of this solution are firstly that it introduces phantom locking where two atomic actions are prevented from modifying different parts of the same page. Secondly it employs a built-in concurrency control mechanism at a low level. Problems of deadlock will also have to be addressed. An alternative approach is given in chapter 3.

One optimisation that can be used with concurrent shadow paging is *meld batching*. Requests to meld are not handled on a one-by-one basis. Instead they can be delayed until there are a number of such requests which are then serviced together. This can gain some performance advantages especially if the degree of multiprogramming is high.

2.8 Comparing Shadow Paging with Logging

Which of the recovery methods described above is the best? Comparisons between different mechanisms trying to achieve the same overall effect is often a difficult and usually inconclusive task. A number of (not necessarily independent) criteria have to be considered when making comparisons :-

- the tradeoffs in the time taken for recovery against the time and resources used in constructing a recoverable system.

- expected frequency of hard and soft crashes. In conflict concurrency systems the frequency of aborted actions is also a factor. This may also depend on the concurrency control implementation used, for example optimistic concurrency control may result in more transaction aborts than say two-phase locking.

.• frequency and cost of checkpoints.

- store architecture and its anticipated use. The issues here include the frequency of updates, locality of reference, object identity and addressing. Scalability of the recovery mechanism with respect to store size may also be of concern.

- hardware and operating system support.

Some systems have sophisticated recovery methods built in as an initial design decision. For example the Monads [RHB+90] architecture uses shadow paging of its persistent store to achieve stability. Other systems such as Cedar [Hag87] incorporate logging as an auxiliary structure in the file system to speed up writes and recovery whilst in the log-structured file system of Rosenblaum and Outerhout [RO91] all data, persistent and transient, is kept in logs. As pointed out earlier several operating systems provide access to the paging mechanisms giving potential for increased shadow paging performance.

Attempts have been made at analysing and comparing the cost of different recovery schemes [AD85a, KGC85]. The results of these efforts do not produce a clear winner. Other research [GMB+81, Kol87, AD85b] would suggest that logging is a better technique especially when the system needs to support conflicting actions. Implementations of shadow paging are not widespread and it

is believed by some to be an inappropriate technology for database applications. The implementors of System R used a complex combination of shadow paging and logging and claim that in hindsight they would have adopted a purely log-based recovery scheme. Furthermore they stated they were unable to perceive of an appropriate architecture based purely on shadows that could support transactions.

Agrawal and DeWitt produced a complex cost model used for comparing shadow paging with logging using a variety of concurrency control techniques. Their approach was purely analytical and their database simulations did not account for the costs of buffer management. The results for shadow paging in these simulations were poor when compared with logging. However closer inspection of their model reveals an unexplained assumption. In the logging case it is assumed that the size of the records that are written to the log for each page modified by a transaction is 10% of the page size. So if during a transaction's execution data is modified on 10 pages the assumption is that the size of the log records for that transaction amount to 1 page. This assumption may be valid in some models of computation. However if the transactions are generated from language systems that frequently update large objects, such as graphical objects, or higher order functions the assumption may not be sustainable.

In contrast the Predator project [KGC85] took an empirical approach to comparing the two methods. A realistic transaction-based database was constructed and logging and shadow paging recovery mechanisms implemented on stock hardware. A variety of transaction experiments were carried out using both recovery techniques and the results compared. The performance metrics were based on transaction throughput and mean response time. Their first observation is that there is no one best mechanism and that the choice of recovery method is application dependent. They concluded that shadow paging works best when there is locality of reference and where the page table cache is

large. By using meld batching, shadow paging outperformed logging as the number of simultaneous transactions increased. Another interesting observation they made was that the shadow paging imposes a more evenly balanced I/O load than logging. Under load a noticeable performance drop was observed in the logging scheme as the system is divided between requests for sequential disk writes for the log and page reads and writes for the database.

Most of the objections to shadow paging performance are founded on a belief that the cost of writing a journal of updates to a log will almost always be more efficient than the maintenance of shadow pages. This may be true for a class of problems but may not be true in general. Many of the measurements that this notion was founded on were based on simulations or were taken from tests run on machine architectures and configurations that are now obsolete. It may be fair to suggest that the results of the comparisons related to the limitations of technology and systems available at the time. For example the overhead of page-table lookups in shadow paging was considered very costly. However the size and speed of memory in an average workstation have risen dramatically over the last few years so that the page table even for a large store could reside in main memory.

Computational models too have changed, not least with the rise in popularity of database programming languages and persistent systems. These systems make different demands on a database or stable store with different patterns of use from conventional database accesses. For example programs and data are not treated differently in orthogonal persistent systems. It is not obvious how logging could be used to efficiently record program state. In contrast this is relatively straightforward to achieve in a shadow paged system by including the process state in the address space that is shadowed. These arguments suggest that the decision on a superior model of recovery is not so clear cut. It may be that shadow paging is a better alternative. Certainly it is clear that shadow paging

implementations can get great performance improvements from an operating system that provides an external pager or memory-mapping support. This support seems more forthcoming in stock systems than explicit support for fast logging.

It has been argued [AD85b] that on a small scale, locality of reference would seem to favour a log-based solution since the amount of information that requires to be written to the log is small compared with the overhead of copying a whole page. Furthermore with logging there is no requirement to write back modified pages after a meld and hence a frequently modified page can reside in main store through a number of transactions. Kent's [KGC85] experimental observations suggest the exact opposite. As locality increases the page table overhead in shadow paging is soon amortised. With logging the amount of modified data that must be saved increases. There quickly comes a point where a lot of locality, especially within one page, along with frequent updates to objects on the page tips the balance in favour of a solution that makes a one-off copy of a page rather than maintains a journal of changes. Furthermore if the objects themselves are fairly big then frequent modifications to them will have an adverse effect on the log size but not on a shadow page. This kind of locality is exactly the type of behaviour that might be exhibited in persistent systems with higher-order functions.

It was shown earlier that for systems that consider the program state as part of the data there is potential for restoring more information following a crash. Maintaining program state changes in a log-based system may be expensive since each state change potentially requires a log entry. This can be alleviated to some extent by only recording the program state at a checkpoint. In contrast shadow paging seems well suited to handling programs as data. Most state changes are probably fairly localised and so once the first state change has caused a shadow page to be created there is little extra overhead involved.

2.9 Conclusions

The issues of recovery management centre around the trade-offs between the cost of data recoverability against the expected frequency and impact of failure. The cost of recovery management not only involves the overhead of restoring data after a failure but also the time and space overhead required to maintain sufficient information during normal operation that ensures that data are recoverable.

In the case of soft failure recovery management data must be restored to a state that is acceptable to the user following the loss of the volatile store. The user can control the points at which his view of the data, the conceptual store, corresponds with its image on non-volatile store by melding. The atomicity of melding with respect to the conceptual store is not reflected in updating the non-volatile store since multiple writes to non-volatile store are not atomic. Techniques for soft-failure recovery enable a consistent state to be reconstructed after a crash by replicating data or changes to data on non-volatile storage. This information is used on recovery to restore consistency through a combination of undo and redo operations.

This chapter has given some background to the issues of recovery from soft failure and presented a detailed description of two common methods, namely logging and shadow paging. The logging and shadow paging mechanisms were discussed together with possible optimisation techniques and the complications arising from conflicting concurrent operation.

From comparative studies that have been carried out it is clear that no one particular method is superior. Rather the choice of the best recovery method is heavily dependent on the way the data is used. However there is some evidence to suggest that shadow paging may be the appropriate technique to use in orthogonally persistent systems. This view is based on the belief that such

systems are expected to exhibit a high degree of locality. It is also contended that shadow paging may be a more efficient technique than the alternatives for supporting recoverability in persistent systems that regard the program state as data.

The next chapter demonstrates how after-look shadow paging may be efficiently incorporated into the implementation of a persistent object store within the Unix framework. This store is primarily designed to support the persistent language Napier88 and has a single-user operation. Chapter 4 shows how the design of the store and the after-look mechanism can be extended to cater for concurrent operation.

3 Shadow Paging Implementation

3.1 Introduction

In the previous chapter the recovery techniques of logging and shadow paging were explored and the view was expressed that shadow paging may provide a more efficient soft failure recovery mechanism for orthogonally persistent systems. To this end an efficient implementation of a stable virtual memory system based on after-look shadow paging has been designed and built. The stable virtual memory provides a paged address space that can always be restored after a soft failure to a self-consistent state. The interface provides functions for reading and writing to the stable virtual memory along with a meld operation that atomically establishes a new consistent state. This system has been built within the persistent object store framework used to support Napier88.

This chapter examines the problems of implementing single-user shadow paging in virtual memory and presents a detailed description of how this stable virtual memory system was built on the SunOS operating system.

3.2 Implementation Issues

Paged virtual memory separates the user logical memory from the physical memory such that the logical address space can be much larger than the size of the main store. The virtual address space is divided into a number of fixed-sized pages that reside on disk blocks on backing store and the main store is divided into page frames of the same size. The operating system's memory management controls the movement of pages of the virtual memory on backing store to and from the page frames on main store and maintains a page table, the main memory page table, which records the allocation of page frames of main memory to pages of the virtual memory.

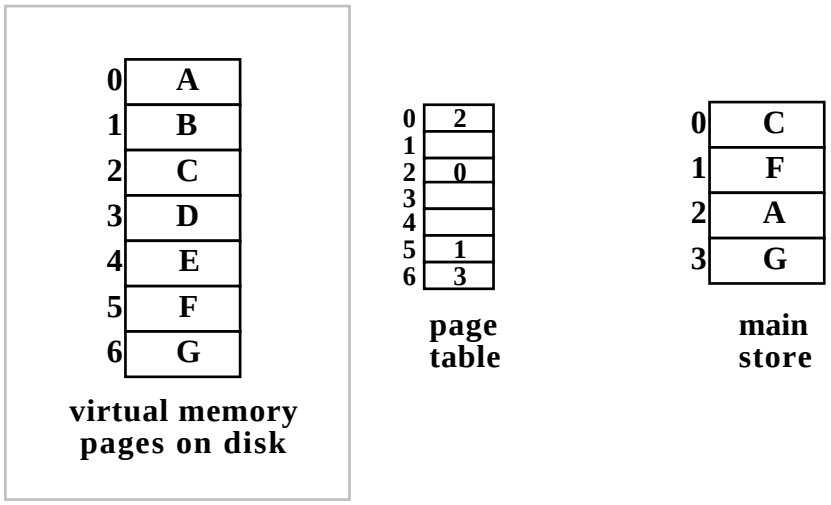


Figure 3.1: Virtual memory

Figure 3.1 depicts a virtual memory scheme with the pages of the virtual memory held on disk. The page table records the mappings of pages to main store page frames. Since a soft failure results in the loss of the main store then there is no point preserving the page table through system crashes and hence it is transient.

The operating system typically allocates the pages of the virtual memory to a contiguous range of disk blocks on backing store and records the disk address of the first disk block, the base address. Virtual memory addresses are usually relative offsets from the first page and hence the *i*th page can be found at the *i*th disk block from the base because the pages and disk blocks are always in one-to-one correspondence.

A shadow-paged virtual memory system is similar to the scheme shown in figure 3.1 where the pages of the virtual memory reside on backing store and a main-memory page table records the allocation of pages to physical memory page frames. The main difference in shadow paging is that the system ensures that before a modified page is written back to non-volatile store that there is always a retrievable copy of the original page on non-volatile storage.

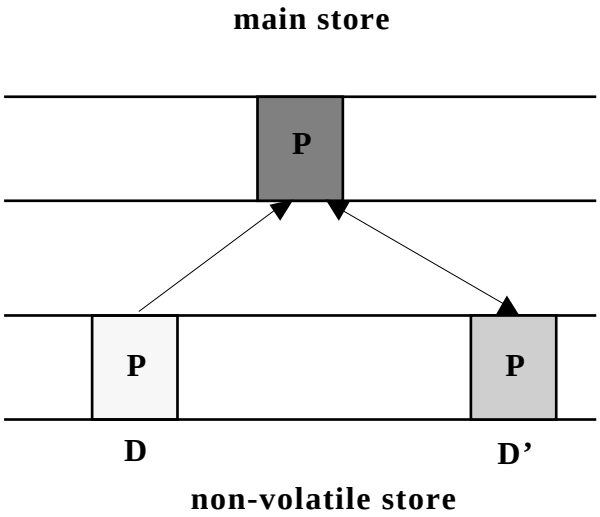


Figure 3.2: After-look shadowing

Figure 3.2 illustrates the after-look shadow paging mechanism showing that the virtual memory page P is initially located at disk block D. Page P is moved into main store and subsequently modified. When P is paged out it is written back to a different disk block D', its *shadow* page, leaving the original intact at the old location. This scheme then destroys the one-to-one correspondence between the pages of the virtual memory and disk blocks and so shadow paged virtual memory maintains a disk page table that records the mapping between pages of the virtual memory and disk blocks on backing store.

A further requirement in shadow paging is that this disk page table is stable. In order that a consistent state can be recovered from the non-volatile store after a crash the shadow page mechanism requires that the mappings of the disk page table survive a soft failure. The shadow paging scheme maintains two versions of the disk page table; a *stable* disk page table on non-volatile store which records the mappings from the last consistent state and a *transient* disk page table which reflects the current state of the mappings. Figure 3.3 illustrates the architecture

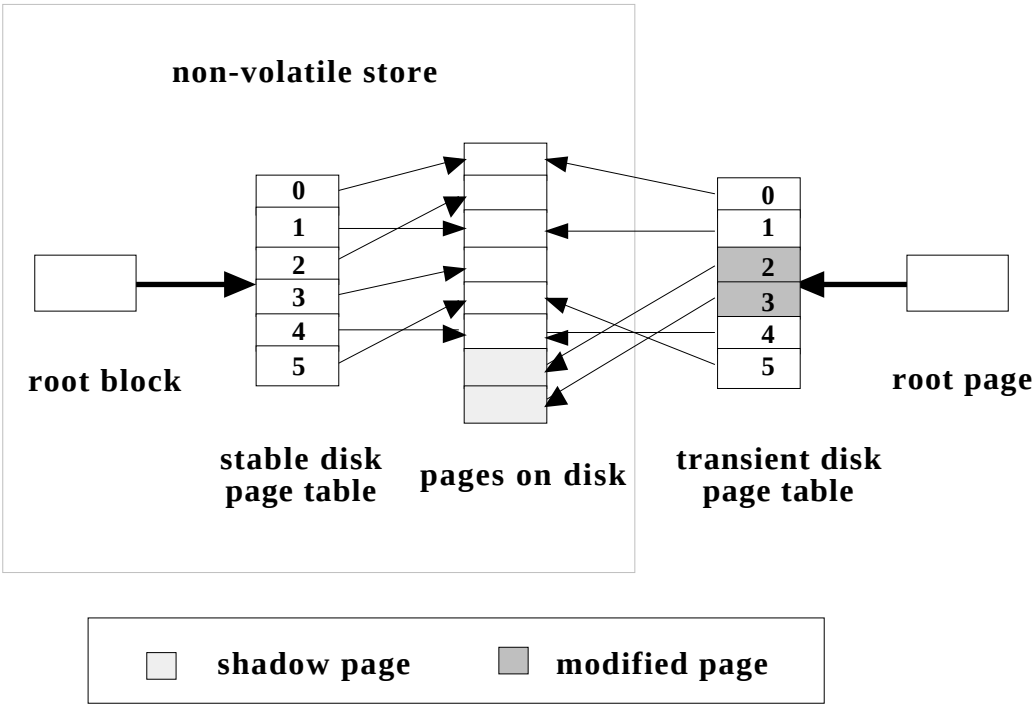


Figure 3.3: Layout of after-look shadow paging

The principal function of shadow paging is to make sure that there is always a recoverable consistent version of the virtual memory. To establish a new consistent state when the user melds, the shadow paging must first write out all modified pages to their shadows and then provide a mechanism that replaces the stable disk page table with the transient disk page table in a single atomic action.

Most operating systems do not support the requirements for a shadow paged virtual memory system outlined above and hence an implementation strategy must adopt its own mechanisms. A problem in describing implementation issues for shadow paging arises because the extent to which facilities of the paging systems can be accessed or manipulated by user program varies amongst operating systems. For example in Mach control of paging can be directed from a user-written external pager. SunOS takes a different approach by allowing some degree of paging control through memory-mapped files. An implementation strategy will be heavily influenced by what access and control to paging and system page tables is permitted by the operating system. For example an operating system may inform the process when a page in main store is being

written back to disk. This would allow the shadow paging implementation to delay the allocation of a shadow block until it is needed. This functionality could be exploited by an implementation to perform the commit batching optimisation outlined in 2.6.3.

Regardless of the accessibility to the system's page management an implementation of shadow paging in a traditional virtual memory operating system requires not only a main memory page table but also, as a direct consequence of the shadow pages, the maintenance of a disk page table. As stated above this disk page table must be stable and the implementation must provide a mechanism for atomically updating it.

3.3 Stable Virtual Memory Implementation in SunOS

3.3.1 Introduction

An overview of the generic layered architecture used to support Napier88 was given in chapter 1. One of the strengths of this layering is that it allows experimentation whereby different implementations of a layer may be interchanged without the need to alter the layers above or below. This section is concerned with the stable virtual memory layer and its interaction with the non-volatile storage.

The stable virtual memory (SVM) provides a contiguous range of addresses for use by the stable heap that can always be restored after a soft failure to a self-consistent state. Data in the SVM can be read and written through the interface functions along with a mechanism for establishing a new consistent state. One of functions of this layer is to maintain a mapping between the SVM and non-volatile store. The combination of the SVM and the non-volatile storage layers is referred to as the *stable store*.

The flexibility of this architecture has led to a number of implementations of the layers, in particular the stable virtual memory layer. This is not surprising since this is the level which interacts closely with the operating system for the provision of non-volatile storage and stable virtual memory.

Currently, there are three versions of the stable store. They can be summarised as follows :-

- The original store implementation by Brown [Bro89] uses block reads and writes for the movement of data between the SVM and the non-volatile storage and performs all the address translations. It employs a before-look shadowing scheme to ensure that a consistent store state is recoverable after a soft crash. This is the most portable and the least efficient of the versions.

- Brown also produced a second implementation of a before-look shadow-paged stable store. This version utilises the memory-mapping facilities of the SunOS operating system described below to implement the paging mechanisms.

- A new after-look shadow-paged stable store based on the Shrines [Ros83] model has been implemented as part of this thesis. This also uses the SunOS memory-mapping features.

The next section gives an overview of the SunOS memory-mapping facilities. The rest of the chapter then describes in detail the implementation of this third version, the new after-look shadow-paged stable store, how it makes use of the memory-mapping features of SunOS and gives some comparisons with Brown's second implementation. The discussion of these stores centres around the issues of :-

- Stable virtual memory and non-volatile address space layout

- Format of the root pages

- Stable store creation
- Stable store startup
- Stable store access
- Stable store checkpointing
- Stable store recovery

3.3.2 SunOS Memory-Mapping Facilities

The memory-mapping functions of SunOS are a set of system calls that allow the establishment of a mapping and a degree of control over the movement of data between pages in the virtual address space and blocks of a file.

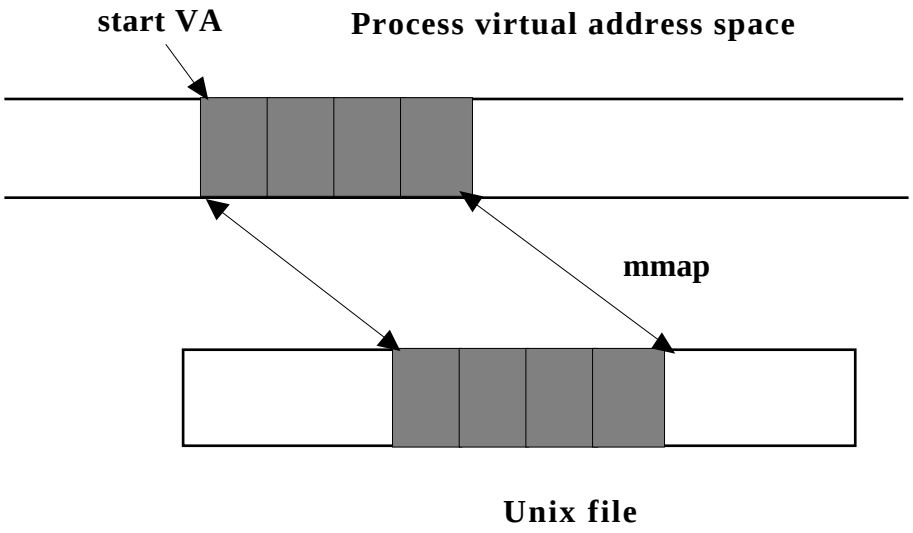


Figure 3.4: SunOS memory-mapping

The *mmap* function sets up a mapping between the given contiguous blocks of a file and the process virtual address space and returns the starting virtual address of the mapping (figure 3.4). An option of *mmap* allows the caller to determine where in the process virtual address space to place the mapping. Otherwise the system chooses an appropriate value. The whole file need not be mapped in one chunk. The function is flexible in that it allows the mapping of individual blocks of a file to specific pages in the virtual address space.

In addition to the *mmap* function there are related functions. *msync* writes all modified pages in the main store in the specified range of the virtual address space to their permanent storage locations. It can optionally invalidate any pages so that further references will force the pages to be read from their permanent locations. The *mprotect* function changes the access protections for a given address range to the specified protection. Protection options are read, write, execute or none. The *munmap* system call removes the mappings.

The main reason for using this memory-mapping facility is the gain in performance. Performance of the store is greatly enhanced since it uses the operating system's page-replacement mechanism and utilises the Sun memory management hardware to perform the address translations.

However the SunOS memory-mapping does not provide all the facilities needed to implement shadow paging. For example :-

- The memory-mapping does not provide access to the page-replacement mechanism. This means that the operating system has total control of the movement of pages between the main store and virtual memory and the user process cannot determine when and which pages are written back to disk. As a consequence the operating system requires that before a page is accessed in the virtual address space the page must have an associated disk block to write out to. This effectively eliminates many of the shadow paging recovery optimisations, such as commit batching.

- The memory-mapping functions do not allow access to the main memory page tables. The user then cannot determine which pages of process virtual address space are in main store at any given time or which pages have been modified. This implies that the stable store implementation needs to keep track of which pages have been modified.

- In memory-mapping disk blocks to virtual memory pages the SunOS operating system is effectively maintaining a disk page table. This table is maintained by the operating system on a per-process basis. The disk page table is transient not only because it is not maintained through soft failures but also because the mappings it contains are discarded when a process completes. The memory-mapping does not allow the user access to the disk page tables. Although the memory-mapping allows the user to create mappings between pages in the virtual address space and backing store it does not provide any facility to enable the user to query the operating system to find what the mappings are. This means that the stable store implementation needs to maintain its own transient disk page table as well as a stable disk page table.

3.3.3 After-look Stable Store Implementation

The stable store implementation uses memory-mapping to control and manipulate the pages of the process virtual address space and the non-volatile storage such that the state of the virtual address space can always be restored to a consistent state. The performance gains in using the memory-mapping features are so compelling that the after-look mechanism used in this implementation works round the limitations outlined in the previous section.

Non-volatile storage is provided through a Unix file where the blocks of this file are mapped into the virtual memory using the *mmap* call. This file is referred to as the *stable store file*. The after-look shadowing mechanism in this implementation effectively involves detecting the first modification to a page in the SVM, finding an unused disk block in the stable store file and establishing a new mapping between this block and the page. When the operating system pages out this page, it will be written to the new block thereby leaving the old block with an image of the page before it was first modified.

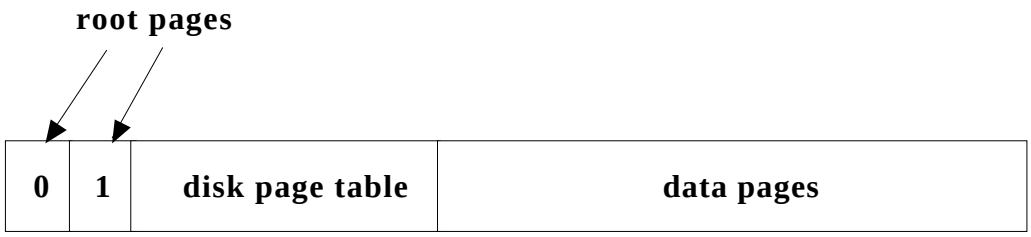


Figure 3.5: Layout of SVM address space

The layout of the stable virtual memory is given in figure 3.5. The first two pages are the root pages whose structure and function are described below. The data pages define the area of the address space that is used by the stable heap. The disk page table is used to record the current mappings between the disk blocks of the stable store file and the data pages. The format of a disk page table entry is given in figure 3.6. The disk page table entries are organised as an array of 32-bit words with one entry for each data page such that the *i*th entry of the table contains the entry for the *i*th data page. Each disk page table entry (PTE) records the block offset in the non-volatile storage that is mapped to the page or zero if no mapping exists. Each PTE also has a number of flags which record state information for the page. The disk page table is included in the SVM address range so that pages of disk page table are subjected to the same shadow paging scheme as the data pages. A secondary disk page table which is located in the root pages keeps an array of disk page table entries for each of the pages of the disk page table.

Figure 3.6: A disk page table entry

The number of data pages and hence the address space of the SVM is fixed at store creation time. The SunOS system imposes an upper limit on the number of pages that can be memory-mapped by a process. This limit is dependent on the operating system version and machine type. By experimentation the SVM

address range has been calculated and set to the maximum that can be used across the range of machines.

An illustration of the layout of the non-volatile store and how it is mapped into the SVM is given in figure 3.7. The root pages are mapped to fixed locations in non-volatile store as shown. For the rest of the SVM the system does not associate a disk block in non-volatile store with a page until the page is first accessed. Since these mappings are created on demand then the order of pages in the SVM address space is independent of the order of disk blocks of the non-volatile storage.

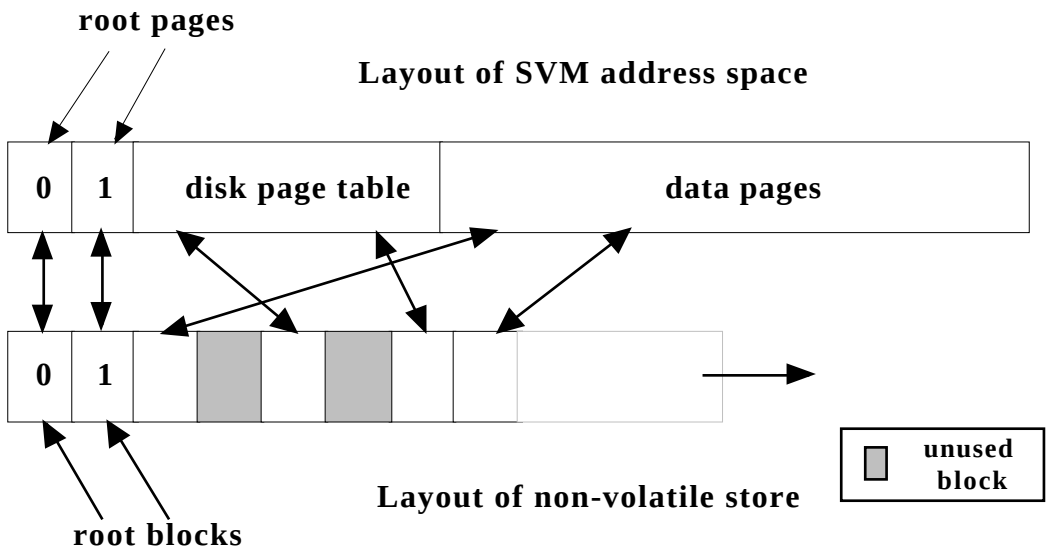


Figure 3.7: Stable store layout

The shaded areas in figure 3.7 represent unused disk blocks. On startup or after a checkpoint the system can establish that some of the mappings are no longer required and hence the disk blocks associated with these mappings can be re-used. If there is a request to access a previously unused data page a free block must be found. One of the features of this implementation is that if there are no free disk blocks available then the Unix file that is being used to represent the non-volatile storage layer can be extended to accommodate the request. A free space bitmap is constructed in local memory at start up to record which disk blocks are unused. The bitmap has a bit for each page of the SVM address space

rather than a bit for each disk block so that as the file expands the free list does not need to be thrown away and recreated. Hence in this implementation the size of the SVM address space is determined and fixed at store creation time whilst the non-volatile storage is variable in size and grows on demand as new data pages are accessed.

3.3.3.1 Root page layout

As shown in figure 3.7 above the disk page table resides in the stable virtual memory address space along with the data pages. Therefore the disk page table itself is paged and hence the architecture is required to maintain a disk page table for the disk page table pages. This *secondary* disk page table is recorded in the root page (figure 3.8) with one entry for each page of the *primary* disk page table. Each entry uses the same format as the primary disk page table entries shown in figure 3.6. The mappings of the entire stable store can therefore be found by a traversal of the secondary disk page table in the root page.

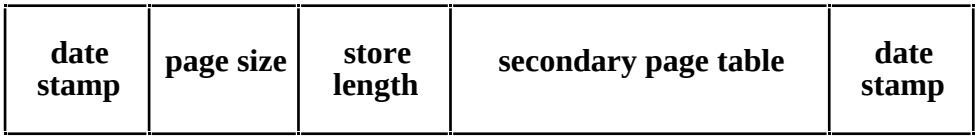


Figure 3.8: The root page layout

The implementation uses two root pages to achieve atomicity. Details of this checkpointing mechanism are given in section 3.3.3.5. The term checkpointing is used in the description of this implementation since in a single-user shadow-paged store it is identical to a meld operation as described in section 2.6.3. The date stamps on the root page are incremented when the page is written back on checkpoint. The date stamps are then used on startup to discover which root page was used in the most recent checkpoint. The two date stamps on the root page are there to detect if any root page corruption occurred when the page was written

back to disk. If the two date stamps on a root page are identical then it is assumed that the page was successfully written to disk.

The page size field records the size of a page in this store implementation and the store length field records the size in bytes of the data pages area of the SVM address space. The size of the primary disk page table area is not recorded since it can be easily calculated from the size of the data pages area. Similarly the size of the secondary disk page table in the root page can also be calculated from the length of the primary disk page table. The current length of the non-volatile storage file is not recorded in the root page since the size of a Unix file can readily be obtained through a system call.

3.3.3.2 Stable store creation

A new stable store is created by a program outwith the Napier88 system. Stable store creation involves creating the Unix file to be used for non-volatile storage, initialising the two root pages and writing them out to the file. The date stamps and the secondary disk page table entries are all set to zero. In the implementation the page size is 8192 bytes and the store length which records the maximum size of the data pages area is set to 384Mb. This means the data area has 49152 pages. The disk pages tables thus need 49152 disk page table entries each of 4 bytes so the disk page tables area is 24 pages long. The secondary disk page table in the root page therefore has 24 entries.

3.3.3.3 Store startup

The startup procedure begins by memory-mapping the root blocks of stable store file into the virtual address space. The date fields of the root pages are then interrogated to find which root page is the most recent. This root page is then copied into local memory, that is memory outwith the SVM. The root page in local memory, the *current* root page, will be modified as changes are made to the secondary disk page table between startup and checkpoint. Because the SVM has

no control over when the pages are written out the root page on disk, the *root block*, must not be overwritten until a checkpoint. In using a local copy of the root page the SVM can prevent the root block from being overwritten until checkpoint.

From the current root page the SVM is reconstructed to the state it was at the last checkpoint by re-establishing the mappings between pages in the SVM address space and their disk blocks. This is done by a traversal of the secondary disk page table. Each non-zero entry in this table specifies the block offset in the stable store file of a primary disk page table page. Each primary disk page table page is then memory-mapped, using the *mmap* system call, into the SVM from this block offset. Each primary disk page table page that is restored contains an array of disk page table entries for the data pages. These disk page table entries specify the block offsets in the stable store file for the data pages and hence this is used to map, again using the *mmap* call, the data pages into the SVM.

The free space list referred to in section 3.3.3 is created in local memory on store startup as a bitmap. For each disk block that is memory-mapped into the SVM at startup the free space list sets the appropriate bit to indicate an allocated disk block in the store file.

The last action of the startup process is to designate all the restored pages as read only using the *mprotect* function.

3.3.3.4 Store access

One of the conventions of the SVM interface is that the stable heap must “allocate” space from the SVM before using it for the first time. When the stable heap layer requests the use of a range of virtual addresses the page boundaries for this range are calculated. The SVM then ensures that each page in the range has an associated disk block to write back to. This then satisfies the requirements

of the SunOS memory-mapping facility that every page accessed in the process virtual address space must have a block to page out to.

If any of the pages in the request have no associated block then the free list is searched for an unused disk block. If there are no free blocks in the stable store file the store file is extended by writing a disk block to the end of the file. This new block is then memory-mapped with *mmap* to the requested page and the disk page table entry associated with this page updated.

When updating the disk page table entry a similar check must be made to ensure that the page that encompasses this disk page table entry also has an associated disk-block.

The after-look shadow paging mechanism requires that a shadow copy is created the first time a page is about to be modified since a checkpoint. To do this the SVM first searches the free space list for an unused block. If there is one, the page that is about to be modified is written out to this block otherwise the page is written out to the end of the stable store file. Either way a copy of the page is written out to its shadow. This block is then memory-mapped to the page and the disk page tables updated to reflect the new mapping. The disk page table entry records the new mapping of the page. The flags field of the disk page table entry records that the page has been shadowed so that any further modifications to this page will not create another shadow.

Updating the disk page table to record the new mapping of the shadow page of course modifies the disk page table entry's page. If this is the first modification to this page then it too must be shadowed. This shadow paging of the disk page table page is performed exactly as for a data page and causes an update to the secondary disk page table in the current root page. This of course is not shadowed and therefore the process stops here.

This is the essence of after-look shadow paging where changes to a page will be written out to a new disk block with the previous state of the page left intact in the old disk block.

The state at the last checkpoint can always be traced from the root page on disk whilst the current state can be traced from the local root page copy.

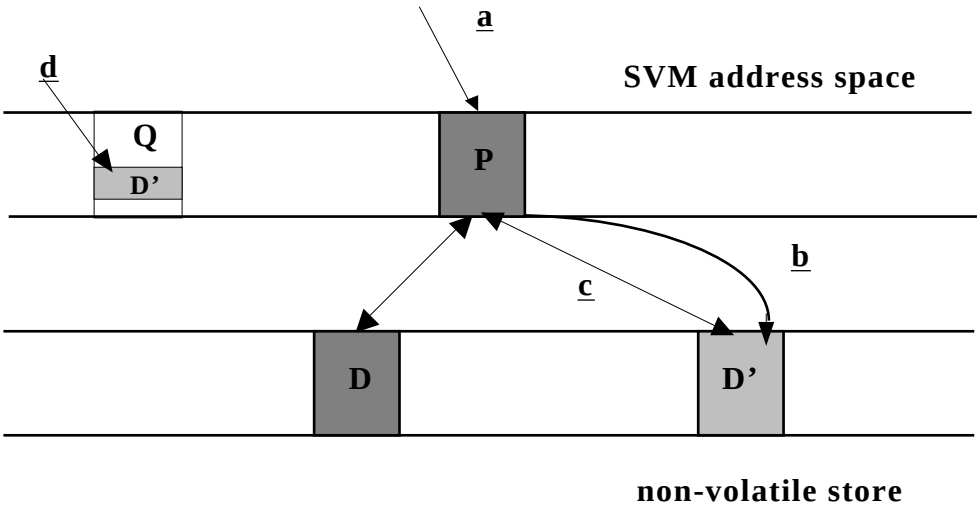


Figure 3.9: After-look shadow paging mechanism

The sequence of events, lettered a to d, that happen when a page is shadowed is illustrated in figure 3.9. Initially the disk block D is memory-mapped to page P and the disk page table entry for P resides in page Q.

a) A request is made to modify page P. If this is the first modification to P then a shadow copy of P must be taken. The free space list is searched for a free block in the stable store file extending the file if necessary. A free block D' is found.

b) Page P is written out to disk block D'.

c) Disk block D is unmapped from P using the *munmap* call. Disk block D' is then memory-mapped to P. There are two reasons why step b is performed before step c can be done. Firstly the *mmap* system call establishes a map between a disk block and a page, not the other way round. After the

unmapping and remapping the operating system may decide that page P must be overwritten with D'. Step b ensures that D' has a copy of P before establishing the mapping. Secondly the mapping of D' to P must happen before P is modified. The stable store has no way of detecting when the operating system may page out P and must ensure that disk block D is not overwritten.

d) The disk page table entry for page P is updated to record that P has been modified and shadowed and that D' is now mapped to P.

   If this is the first update to the page Q, the page containing the disk page table entry for P, then Q is also shadowed following exactly the steps a to c. The new mapping for Q is recorded in the secondary disk page table in the current root page.

3.3.3.5 Checkpointing

In an ideal situation a checkpoint would begin by writing out all modified pages in main store to disk. The only interface provided by the operating system to write back modified pages is the *msync* function call. The function is parameterised by the process virtual address range of pages to be written out to disk. Since, as stated earlier, the stable store cannot determine which pages in the SVM are currently in main store, the checkpoint mechanism must either *msync* the entire SVM range or individually *msync* each modified page. However many of the pages of the SVM have not been allocated disk blocks to write back to since they have never been accessed. An *msync* on the entire SVM first checks that each page has somewhere to write back to before deciding if the page needs written back and hence will fail. So the checkpoint mechanism works by finding all the modified pages and using *msync* call to write it back to disk.

Starting with the secondary disk page table in the current root page the SVM is traversed as before and each modified page is written back to disk. The date

stamps of the current root page are then incremented. The SVM then copies the current root page from local memory into one of the root pages in the SVM address space and flushes that root page to disk. On checkpoint the current root page is not copied back to the root page it was read from but is always copied to the “other” root page. In other words the root page holding the information about the previous checkpoint is preserved and the other root page is overwritten.

Because of the after-look mechanism no block in the non-volatile storage involved in the checkpoint is overwritten. The atomicity of the checkpoint is therefore only dependent on the atomic update of the root block. It is assumed that any error encountered in writing the root page to disk will be detected and can be acted on immediately. As a further precaution the date stamps at the beginning and end of the root blocks can be compared. Any difference in these indicates that the root block is corrupt.

Now that a consistent state has been established the original disk blocks that were mapped to pages that were shadowed at the checkpoint can now safely be re-used. For example the disk block D in the example shown in figure 3.9 can safely be overwritten after the checkpoint since page P has been flushed to D’ and the disk page tables and root pages updated. Finding such disk blocks is done by designating all the disk blocks unused by clearing out the free list and then reconstructing the ones that survived the checkpoint by traversing the store from the current root page. Note that this is exactly what happens when the store is started up. From the root page the disk page tables are traversed and the pages mapped to their recorded disk blocks.

3.3.3.6 Store recovery

Since the after-look mechanism ensures that nothing is overwritten there is always a consistent state of the store on non-volatile storage. After a soft-failure the recovery mechanism finds the most recent, consistent root block and from

there reconstitutes the mappings between the SVM and the non-volatile store. Note that this is exactly the same procedure as used for store startup. Because recovery from soft failure is identical to store startup this mechanism can be described as being no-undo/no-redo.

3.3.3.7 Optimisations

A number of optimisations have been made to the stable store implementation just described. The main changes are designed to reduce the cost of checkpoint. The checkpoint operation in the original implementation can be costly for two reasons. Firstly finding the modified pages that need to be written out to disk involves a lengthy traversal of the store through the secondary and primary disk page tables. Secondly, as each modified page is found it is written out to its associated disk block using the *msync* call. This operation may incur a heavy seek time cost since the order of modified pages in the SVM is independent of the disk block order. Both these issues are tackled using a *block list* which is effectively a log of pages that are modified between checkpoints.

The new layout of the SVM and non-volatile store is given in figure 3.10. As before there are two root pages at the start of the SVM that are mapped to specific disk blocks. The disk page tables are again located in the SVM after the root pages. This is followed by the block list which is the same size as the disk page tables.

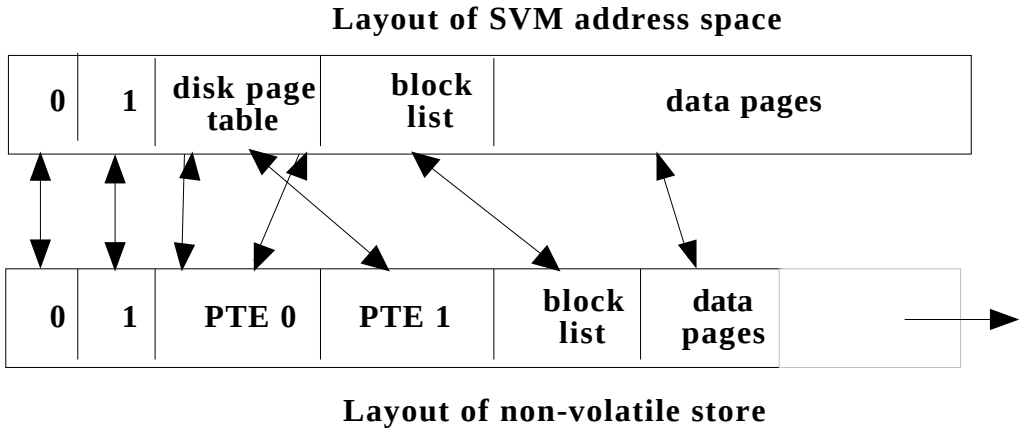


Figure 3.10: Stable store layout

Each time a page is shadowed it is allocated an unused disk block and the address of this page is added to the end of the block list. The block list can be thought of as a log of SVM addresses of modified pages. Finding an unused block involves a linear search of the free list and if none are found by extending the stable store file. Since this is in block order the block lists record a block-order mapping of modified pages that need to be written to disk on a checkpoint. It is sufficient on a checkpoint to use this block list to determine which pages need to be written out. Because the pages are written out in increasing block order the disk seek time should be minimised. The checkpoint procedure employs a further optimisation by using a form of run length encoding. Rather than just write out one modified page at a time the block list is inspected for a sequence of contiguous pages which can be written out in one system call. So a sequence of say five contiguous modified pages will be written out with one system call of five page lengths rather than five system calls of one page length.

When the stable store file is created disk blocks are allocated and reserved for the root pages and for the pages of the disk page table. A second set of disk blocks is also allocated to accommodate shadow copies of these disk page table pages. Space on the stable store file is also reserved for the pages of the block list.

In this optimised implementation shadow copying data pages is done as before but a different mechanism is used for shadow copying disk page table pages. When a data page is first modified a new, unused block is allocated for the shadow copy and mapped to the page. The disk page table entry for this data page then records the block offset that the page is now mapped to.

Each disk page table page has two disk blocks associated with it. The *i*th disk page table page is either mapped to the *i*th disk block in the first disk page table (marked PTE 0) or to the *i*th disk block of the second disk page table (marked PTE 2). As before the secondary disk page table in the root page records the mapping between the disk page table page and its disk block. When a disk page table page is first modified its entry is looked up in the secondary disk page table of the current root page. This determines which disk block the page is currently mapped to. The disk page table page is then shadow copied to its disk block in the other disk page table and the secondary disk page table updated to reflect this. The advantage of this method is that the blocks for the disk page table pages are pre-allocated and reasonably localised.

date stamp	**page size**	**store length**	**average modified pages**	**modified PTE bitmap**	**secondary page table**	**date stamp**

Figure 3.11: Root page

The format of the root page in this optimised implementation is given in figure 3.11. Two extra fields have been added. The modified disk page table entry bitmap is used to determine which of the disk page table pages have been modified between checkpoints. This bitmap has one bit for each of the pages of the disk page table. This is used when doing a checkpoint to limit the search for modified pages.

The average number of pages that were modified between checkpoints is recorded in the root page at each checkpoint. At store startup time the system ensures that the stable store file has this number of free blocks available for shadow copies extending the stable store file if necessary. This can speed up the store access time since a search for a free block in the stable store file is almost always going to be satisfied and hence there is seldom any need to extend the stable store file.

3.3.4 Before-look Stable Store Implementation

This section gives a short description of the stable store designed and built by Brown [BR91]. This stable store is a before-look shadow-paged store built using the same layered architecture and using the same interface functions.

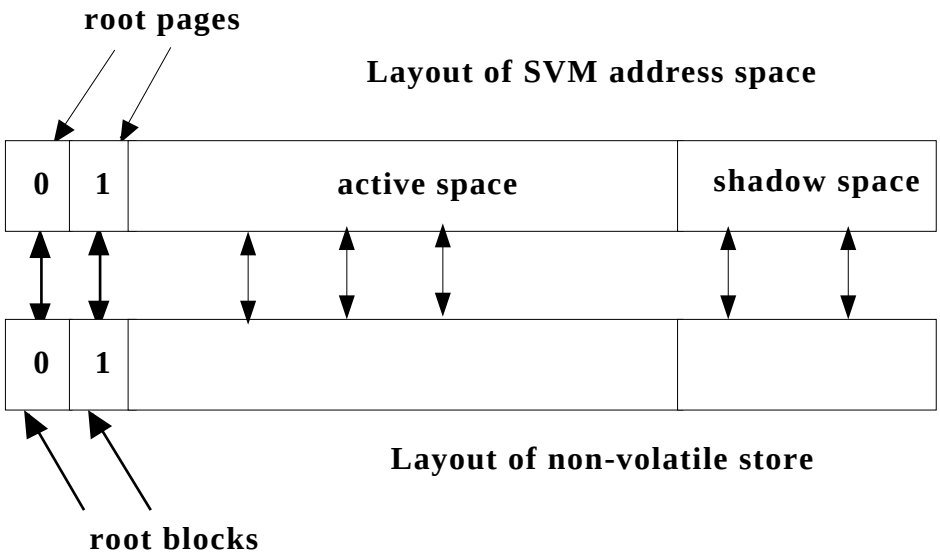


Figure 3.12: Layout of Brown's stable store

There are two main differences between Brown's store and the after-look store described above. The first is the layout of the SVM and non-volatile address space and the second is the before-look shadow paging mechanism.

The layout of this store is given in figure 3.12. The root pages in this implementation are used in the same fashion as in the after-look store. The remainder of the SVM address space is divided between *active* and *shadow*

space. The active space defines the range of the address space that is used by the stable heap and the shadow space is used as an area for shadow copies of pages. In this implementation the entire physical resource needed to support this SVM is pre-allocated. When the stable store is created the user must specify the size of the active area and the size of the shadow area. The Unix file, which is used as non-volatile storage, allocates disk blocks not only for the two root pages but also for each page of the active and shadow areas. The stable store file is thus identical in size to the SVM address space and there is therefore a 1-1 correspondence between pages of the SVM address space and disk blocks of the non-volatile storage. In this implementation the stable store file does not grow in size and hence the active and shadow size specified by the user determines the maximum size of the stable store. The position of the static division between the active and shadow areas is recorded in the root pages. On store startup the entire stable store file is memory-mapped into the virtual address space. Because there is a 1-1 correspondence between the pages of the SVM and the disk blocks of the non-volatile storage there is no need for the stable store to maintain its own disk page tables.

With before-look shadow paging the original page is copied to another location and updates to the page are done in place. The before-look mechanism has to ensure before the update in place is allowed to proceed there must be a copy of the original page on non-volatile storage. Furthermore this copy of the original must be recoverable from the non-volatile storage in the event of a soft failure. The shadow area of the SVM is used to store the shadow copies of pages that are about to be modified. This area effectively forms a sequential log of before-images of pages that have been modified. The root page maintains a corresponding array, the *copied pages* array, that records the address of each page that has been shadowed. So the *i*th entry of this array gives the address of the page that has been copied to the *i*th page of the shadow area.

On checkpoint a new consistent state is saved by flushing all the modified pages of the active area to disk. The copied pages array of the current root page is set to zero and the root page is also flushed to disk. This signifies that the before-images in the shadow area are no longer required since a new consistent state has been saved on non-volatile store. The same conditions for the atomic update of the root page that were described in the after-look store are used here.

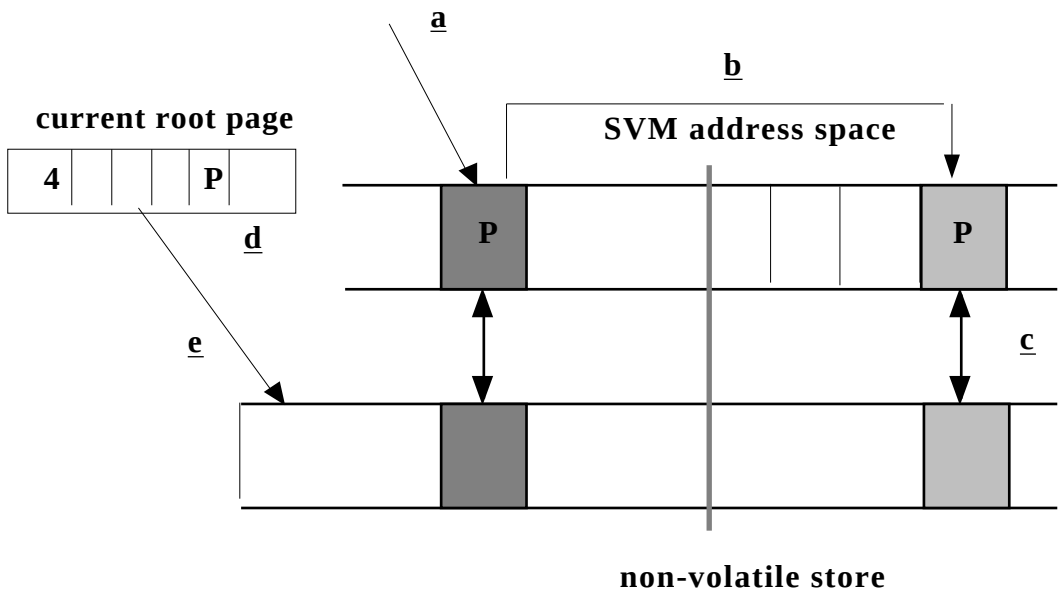


Figure 3.13 : Before-look shadow-page mechanism

Figure 3.13 illustrates the sequence of events, lettered a to e, that are involved in the before-look shadow paging mechanism. The area to the right of the broken vertical line represents the shadow area.

a) A request is made to modify page P. If this is the first modification to P then a shadow copy of the page must be taken.

b) The page P is copied to the SVM shadow area. The offset in the shadow area that the page is copied to is found in the current root page (offset 3 in this case).

c) This shadow copy is forced onto non-volatile store by writing the page onto its corresponding disk block. This ensures that a copy of the original is on non-volatile storage before the update takes place.

d) The current root page records that another page has been shadowed and saves its address in copied pages array. In this example the root page records that the fourth page that has been shadowed is page P.

e) Finally the current root page is written back to non-volatile storage. After this has completed the update to page P can proceed.

This final step of writing the root page back to disk is necessary so that after a soft failure the state of the SVM can be reconstructed from non-volatile store. The before-look mechanism effects this reconstruction after a crash using a undo/no-redo algorithm. On store startup the most recent root page is copied into local memory from the stable store file. If there are any entries in the copied pages array then it is assumed that there was a soft failure before a new consistent state was established. The undo mechanism reconstructs the last consistent state by overwriting the pages in the active area specified in the copied pages array with the originals from the shadow area.

3.3.5 Comments

Brown’s store has a number of possible advantages over the after-look store :-

- All the physical resource required to support the store is available before the store is accessed. This has enabled a number of run-time checks and error handling mechanisms to be avoided.

- There is no requirement to maintain separate disk page tables because there is a 1-1 correspondence between the pages of the SVM and disk blocks of the non-volatile storage despite having no access to the operating system's page tables. This store may expect some performance gains since all the memory-

mapping of disk blocks to pages is performed at store startup. These mappings are not altered during store access.

- The before-look mechanism performs update in place and hence preserves clustering. The contiguity of large objects or objects which cross page boundaries will be preserved.

However there are a number of possible drawbacks with this implementation :-

- The size of stable store is permanently fixed at creation time with no facility for expanding. The user must determine at store creation time the maximum size of the store. If the store is found to be too small for its intended use then the only available option is to throw it away and allocate a new larger store. If store is created too large then it suffers from internal fragmentation.

- Statically defining the active and shadow areas at creation time determines the maximum number of changed pages that can be shadowed between checkpoints.

- The before-look shadow-paging mechanism used effectively involves a checkpoint of the log of shadowed pages. When a range of pages is being shadowed a disk write is required for each page and a disk write for the root page.

The design of the after-look store tries to address some of the possible shortcomings of Brown’s store implementation. The main differences are that :-

- it separates the store address space from the non-volatile address space. The user is not required to statically define the maximum size of the store when it is created. The SVM uses as large an address space as can be memory-mapped by the operating system. The Unix file created to support the non-volatile storage is created to a minimal size and is grown on demand.

- The after-look shadow paging mechanism ensures that a modified page is written back to a different disk block. The state of the last checkpoint is therefore always on non-volatile storage and hence recoverable after a soft failure. This implementation thus employs a no undo/no redo recovery algorithm. Shadowing a page then just requires one disk write to copy the page to its new location.

One optimisation that has not been explored in either implementation is pinning. In the after-look store pinning could be used to implement commit batching by holding a page in main memory until a checkpoint then flushing these pages out in a sequential write. The SunOS memory-mapping features allow pages to be locked in main memory. However this facility is restricted to privileged users.

3.4 Conclusions

Traditional demand-paged virtual memory systems maintain a main memory page table to record the mapping of pages to main store page frames. The pages of the virtual memory reside on contiguous disk blocks on backing store and do not move during the lifetime of the process. In an after-look shadow paging system modified pages are never written back to backing store to the same place they were read from and hence the one-to-one correspondence of pages to disk blocks is broken. As a consequence such systems require an additional table, the disk page table, to record the mappings between the pages of the virtual memory and the disk blocks of non-volatile store. The main reason for a shadow paged system is to ensure that a consistent version of the address space can always be recovered, even after a soft failure. This implies that the shadow-paging system maintains a stable disk page table on non-volatile store which records the mappings from the last consistent state and a transient disk page table which reflects the current state of the mappings. The shadow paging scheme must provide a mechanism that allows the atomic update of the stable disk page table.

The implementation strategy of a shadow paged system built on a typical virtual memory system relies on the extent to which control and access to the paging mechanisms are afforded by the target operating system. Aspects of an implementation are thus chiefly concerned with how it makes use of the available operating system facilities and how it compensates for the lack of others.

This chapter has documented a particular implementation of an after-look shadow paged virtual memory system under SunOS. The SunOS operating system offers the user a degree of control over the paging process through the memory-mapping functions. These allow the user to effectively control the mappings between the pages of virtual memory and the disk blocks on non-volatile store. Of interest in this implementation are the *mmap*, *msync*, *mprotect* and *munmap* functions. The *mmap* function establishes mappings between the blocks of a file and pages of the process virtual address space. The *msync* function flushes all modified pages in the specified range from main store to their permanent storage locations. The *mprotect* function changes the access protections for a given address range to the specified protection. Protection options are read, write, execute or none. The *munmap* system call removes the mappings.

The SunOS stable virtual memory implementation uses these facilities for performance gain since the memory-mapping functions use the operating systems page-replacement mechanism and memory management hardware to effect address translations. Non-volatile store is implemented through a Unix file where the blocks of this file are mapped to the pages of the stable virtual memory. When a page is about to be modified for the first time the implementation finds an unused disk block in this file and changes the mappings so that the modified page will be written out to a different block thereby effecting an after-look scheme. The SVM maintains a disk page table which

records the disk block to pages mappings. On a checkpoint all modified pages are flushed to their shadows using the *msync* call and a new consistent state is established by atomically writing the page table to non-volatile store.

4 Concurrency

4.1 Introduction

Many models of concurrency have been designed and implemented [Dav73, Dav78, EGL+76, Mos81, NZ92, Hoa74, Hoa75, Bri75, Mi80] yet it is not clear which model, if any, is best suited in a persistent context or how concurrency should be incorporated into the persistent architecture framework. Most of the previous work in this area has concentrated on particular models [MBC+88, Wai88, KB92] which may only be useful for a particular set of problems. One exception is CPS-algol [Kra87] which allows the user to specify a number of models in the language. The incorporation of concurrency into the Napier88 system presented here has some similarities to the CPS-algol approach.

The interpretation of concurrency presented here is as a spectrum of understandability (figure 4.1) where points on this spectrum denote the extent to which the user can perceive and manipulate concurrent activities. The points on this scale can be thought of as levels of abstraction over the exposition and control of concurrent operation.

Figure 4.1: Spectrum of understandability

At one extreme is *isolation* where the concurrent activity is hidden from the user and concurrent activities work without knowledge of or interaction with each other. At the other end is *co-operation* where the conceptual interaction of concurrent activities is under complete user control. Specific models of concurrency lie on points within this spectrum. For example the atomic transaction model [EGL+76] could be thought of as lying towards the isolation

end of the spectrum whilst models based on semaphores lie towards the co-operation end (figure 4.2).

A different interpretation of this spectrum can be formed by making the observation that concurrency models residing towards the isolation extreme are typically found in the database paradigm whilst the co-operating models tend to belong in the programming languages world. One of the principal aims of the persistence model is to avoid the *impedance mismatch* between these two worlds by providing the user with a uniform view of data. The problem of integrating concurrency and persistence can then be seen as one of incorporating this spectrum into the persistence framework.

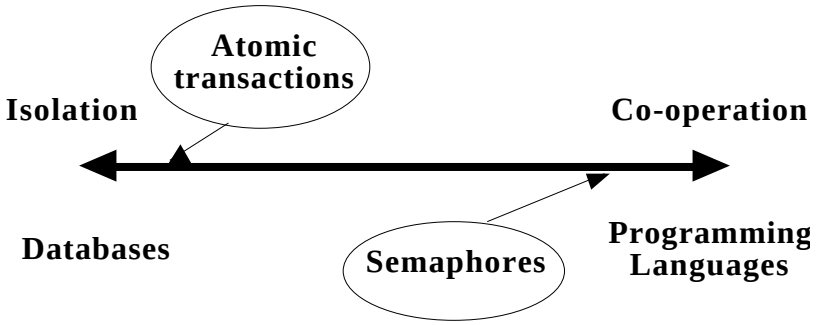


Figure 4.2: Alternate view of the spectrum

Incorporating the complete concurrency spectrum in a persistent system then enables any particular model to be built. Since it is not yet clear which level of concurrency abstraction, if any, is best suited in a persistent context this approach provides a flexible basis for experimentation and usage.

This chapter begins with a discussion of some background issues of concurrency and broadly categorises styles of concurrency in terms of the bounds of cohesion. The second section presents a persistent architecture that has been designed to provide all styles of concurrency by capturing this spectrum into Napier88. This is achieved by the provision of threads and semaphores at the language level to enable the expression of concurrent activity and a supporting architecture which is a marriage between an extended form of shadow paging and the CACS

concurrency control specification system [SM92]. Concurrency schemes, designed using CACS, are directly mapped into these threads. The interaction of the threads on data is then controlled by the dictates of the specification and maintained by the architecture.

4.2 Concurrent Architecture

The basic properties of any model of concurrency can be described in terms of :-

- *Concurrent Activity* - The system provides support for the expression and management of logically separate activities executing at the same time.

- *Shared Information* - A fundamental requirement of any concurrency model is a specification of how information is communicated among activities.

- *Understandability* - It is essential that the programmer can reason about the behaviour of the model's execution. This means that the system supporting the model must maintain the model in a state that is explainable to the user.

Implementation strategies for concurrency models broadly speaking take the following approach to realising these intrinsics.

- Concurrent activity is usually represented by processes or threads of control [HR73, Ras86]. The term *process* will be used in the rest of this chapter to denote a separate activity.

- Information sharing is achieved using either a shared memory mechanism or message passing [Dij65, Bri70].

- Understandability is derived from process synchronisation techniques, melding and stability mechanisms [Hoa74, Lor77].

A particular model of concurrency can be defined in terms of the specification and interaction of these mechanisms. The model's position on the spectrum of

understandability is therefore set by the extent to which the user can determine and control the mechanisms and their interaction. A persistent architecture that incorporates the entire concurrency spectrum must provide :-

- the user with a method of specifying concurrency models.
- a method of executing concurrency models.
- a number of concurrency primitives.
- interfaces to these primitives that allow the system to relinquish and retain control of their interaction as determined by the user's model.

To shed some light on how such an architecture might be constructed the next three sections provide some background by describing three different styles of concurrency that represent different levels of concurrent abstraction and emphasise the organisational support they require from a concurrent system. They are :-

- Co-operating concurrency
- Conflict concurrency
- Designer concurrency

4.2.1 Co-operating Concurrency

At the co-operating end of the spectrum, the programmer is responsible for maintaining global cohesion. Models in this category have complete freedom to organise the concurrent activities and their interaction. The constraints on patterns of behaviour are imposed by co-operation and the preservation of understandability is by agreement.

Typically processes within this framework organise themselves by communication and synchronisation to achieve understandability.

Synchronisation is provided through a system primitive such as a semaphore [Dij65] or critical region [Hoa72]. The important point of this organisation is that the various concurrent activities are completely defined and controlled by agreement between processes.

The major variations with models in this category centre around how processes communicate and how they synchronise. In essence there are two approaches. The shared variable model where processes share a common address space through which they communicate or by message passing.

4.2.2 Conflict Concurrency

In contrast to co-operating concurrency, global understandability may be enforced. The underlying system may hide much of the concurrent operation and present the user with a very limited view of concurrent activity by imposing an organisation that completely controls the interaction of processes. This type of concurrency is needed in situations where processes work in isolation and whose interactions with the system may interfere with other processes. The system provides a framework that ensures that the separate activities do not conflict.

The system is responsible for ensuring the understandability of process execution and, in executing the separate activities concurrently, must take appropriate action to ensure a *serializable* schedule of component parts from a number of processes. The system must also ensure that when a process completes its effects are made permanent. One widely documented model that fits well with this scheme is the atomic transaction model [EGL+76].

The required components necessary to construct a system with imposed organisation are :-

- A concurrency control mechanism is required to ensure that processes do not interfere with each other. More precisely the concurrency control mechanism

determines which processes have access to data and the type of access. Common concurrency control mechanisms include locking, timestamp ordering and optimistic concurrency.

- A mechanism that implements atomicity. In particular it must ensure that the effects of an aborted process or one that failed to complete due to system failure must be undone.

- A mechanism that provides permanence of effect. The changes made by a committed process must be persistent and recoverable after system failure. A detailed discussion of recovery techniques is given in chapter 2.

4.2.3 Designer Concurrency

Strictly speaking any concurrency scheme where the operation is not completely system controlled can be considered to be co-operative. If the system does not impose all the constraints outlined in the previous section then an activity that causes inconsistency can always be constructed. However there is a need to identify a middle ground and recognise that there are a growing number of concurrency models which require a level of global cohesion in which conflicts can occur but that the imposition of total system control is too restrictive.

The reason for the rise in the number of such models is that for a large class of applications the constraints imposed by the atomic transaction model are too heavy. For many of these applications the issue is purely one of performance where the restrictions enforced by atomic transactions can often stifle potential concurrency and hence overall throughput. This is particularly true when transactions are long lived. Techniques such as semantic-based concurrency [Gar83] and the Sagas model [GS87] have been proposed as methods of alleviating this problem.

Many applications that have sprung from work on CAD databases, multi-media, interactive systems and software engineering find that, in addition to stifling performance, the conflict concurrency is unsuitable from a modelling perspective. Tasks in these applications often split naturally into a hierarchy of highly inter-dependent subtasks that are required to interact with each other in a structured way. Total co-operation may not be suitable since groups of subtasks may require to work in isolation from other groups. These so-called *design* transactions are often open-ended in that they are interactive and iterative in nature and cannot always be completely specified when they start.

A variety of concurrency models [FZ89, NSZ91, NZ92, Sut91, EG90] that relax the serializability constraint have come from these application areas. In these models the separate activities do not work in total isolation or in complete agreement with each other. Conflict is either avoided or the effects compensated by following the convention of the model. The processes agree to follow the conventions of the model but not necessarily with each other. In this sense then the system abdicates responsibility to the model for maintaining correct concurrent operation. The level of concurrent abstraction in the designer concurrency range varies from one model to the next. Such models often employ ad-hoc methods to provide a level of understandability. These models are not generally applicable but are only workable in situations where the problem can be expressed within the limitations of the model.

The architectural requirements for designer concurrent systems are difficult to pin down since the models are so diverse. The architecture of each of these systems tends to be specifically tailored to supporting its particular concurrency control mechanism.

4.3 Concurrency in Napier88

4.3.1 Introduction

The integration of concurrency into Napier88 was directed by a number of design aims :-

- The model must be able to support the range of concurrency models.
- Concurrency should be incorporated without changes to the language model.
- The existing layered architecture should be retained and the integration should build in as few primitives as possible to the system.

The crux of the approach taken is to define understandability in terms of data visibility between concurrent activities. This is reflected in the design of a conceptual concurrent layered architecture in which visibility is defined and controlled by the movement of data between a hierarchy of conceptual address spaces. The architecture provides global cohesion at one end of the spectrum by constraining data accessed by one activity to a separate address space from all others. Concurrency models using this architecture define their point on the spectrum in terms of data visibility through the sharing of address spaces and the control of movement between these spaces.

The motivation for this design comes from Stemple and Morrison's CACS system [SM92]. The CACS system provides a framework in which concurrency control schemes can be specified. The CACS system is a generic utility for providing concurrency control for other applications. The system does not actually manipulate any objects, but instead maintains information about their pattern of usage. In order for the system to collate this information, the attached applications must issue signals regarding their intended use of their objects. In

return the CACS system replies indicating whether or not the operation would violate the concurrency rules.

A particular concurrency control scheme is defined by giving a set of rules for the behaviour of the CACS abstract machine. The CACS language is a formal design language supporting mechanical theorem proving. The conceptual framework of CACS may be explained in terms of the components used to specify concurrency control schemes. These are actions, objects, events, and visibility control.

- **Actions**

  An action is a sequence of operations on shared data that has some sense of cohesion; it is a unit of computation that needs some isolation from concurrent users of shared data.

  Actions perform their computations by executing programs which are algorithms annotated by markers. The algorithms specify the manipulations of the data whereas the markers, called events in CACS, specify the points at which the actions must interact with the control system to operate correctly over the shared data.

- **Objects**

  The data consists of uniquely identified objects.

- **Events**

  In CACS there are two categories of events: action events and object/visibility events. All events are requests by an action to the control system to proceed and any particular concurrency control scheme will include all the events necessary to control the use of shared data.

Action events include action initiation and termination. Other events, such as spawning subtransactions in the nested transaction model are action events in particular concurrency control schemes. Action events and their semantics are defined as part of concurrency control schemes. The semantics of these events is defined by rules that specify the effects of actions on the visibility and dependencies.

Object/visibility events include object operations and object commits.

- **Visibility**

  The model of computation that forms the CACS abstraction is designed to focus on the visibility of data from different actions. The semantics of CACS control over visibility is expressed in terms of the database, which comprises the globally visible data, and conceptual stores called access sets. Each action is associated with a local access set and may use other shared access sets in order to effect communication between actions without using the database. When using CACS to specify a concurrency control scheme and when explaining the semantics of a particular control scheme the access sets may be thought of simply as stores holding data. When actions operate on shared data this is modelled in CACS by the effects of the operations being kept in shared access sets. Movement of data from access sets to the database, which is the semantics of object commit, is the way changes to visibility are made global in CACS. Movement among local and shared access sets occurs explicitly.

The basic components of the architecture of CACS are shown in figure 4.3.

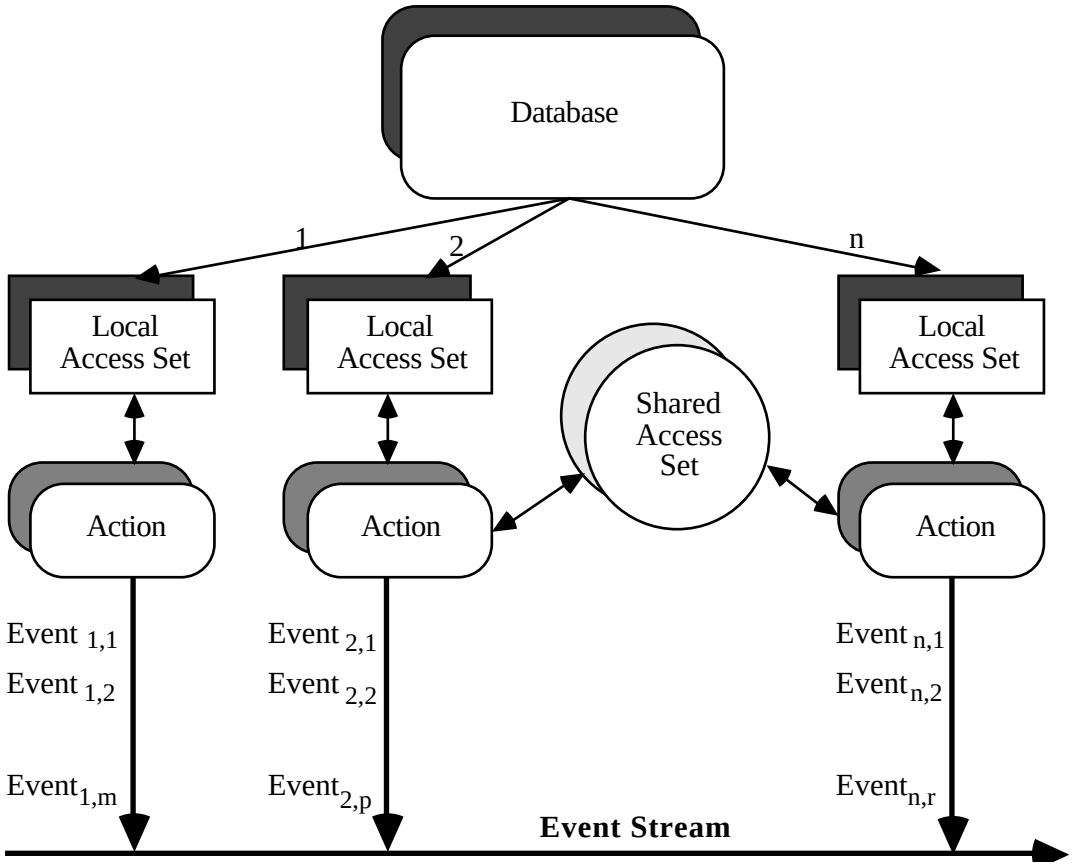


Figure 4.3: The CACS Basic Components

CACS then presents a model that is based on data-centred invariance. This is in contrast to the approach taken by formal models such as CCS [Mi80] and π-calculus [Mi92] which build abstractions from a co-operative concurrency base. In these models the programmer must be aware of the total code body of the system because of the local cohesiveness of the synchronisation primitives. This can add considerable complexity in defining and building concurrency models especially in an evolving system of some scale.

From the conceptual layered design a concurrent persistent architecture has been constructed that supports the CACS model in the Napier88 system. The persistent store is used as the model for the CACS database with a concurrent shadow-paging scheme corresponding to the access sets of CACS. Actions in this system are defined as collections of lightweight threads and semaphores which are used as a way of expressing concurrent activity. A multithreading and

semaphore package has been added to Napier88 and its language design and interface are described in appendix A.

The advantages of dovetailing CACS and Napier88 with a concurrent shadow-paged persistent store are seen as :-

- a system which is capable of supporting any model of concurrency with CACS controlling the movement of data between access sets.

- alliance of access sets with shadow paging. It was argued in chapter 2 that the coarse granularity of shadow paging was seen as an advantage in systems which exhibit locality. The shadow paging scheme does not employ page-level locking.

- The resulting architecture enhances the existing Napier88 layered architecture rather than re-defining it.

The rest of this chapter is concerned with the details of the conceptual layered architecture, the concurrent shadow-paged store derived from it and the incorporation of CACS and the Napier88 architecture. The chapter concludes with an example of how one concurrency model, the atomic transaction model, can be designed and built in this system.

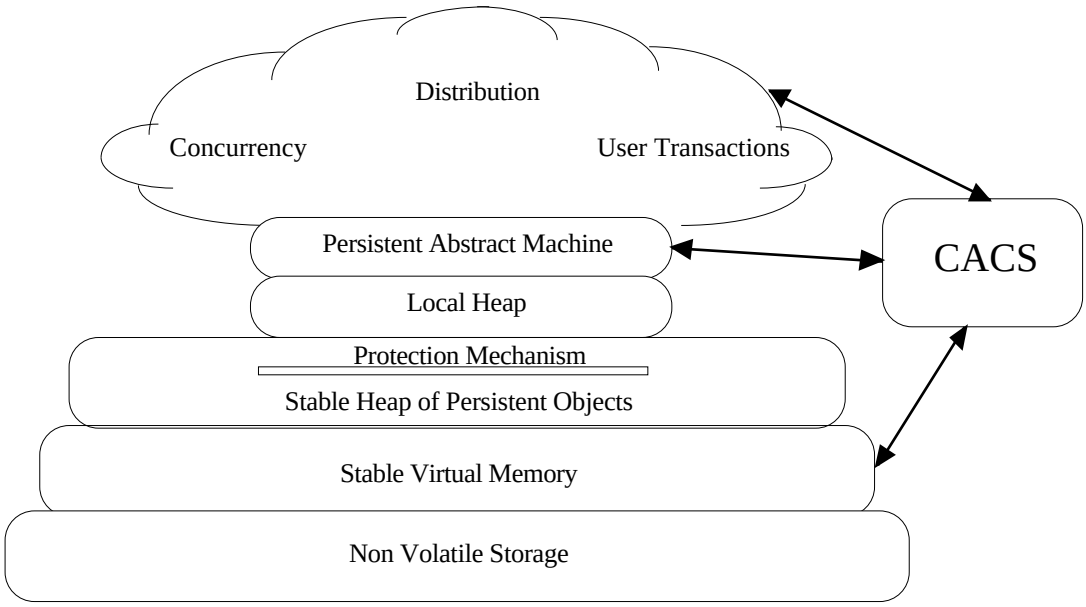


Figure 4.4: The Napier88 Concurrent Persistent Layered Architecture

The implementation of CACS can be thought of as a whiteboard architecture [Cut92] that enables communication from different levels of the Napier88 system (figure 4.4). At the top level the CACS system regards Napier88 programs as event generators. A Napier88 program will observe a concurrency control protocol when written and present itself to the CACS protocol generator. This transforms the program with the correct CACS communication built in.

For example in an atomic transaction model the protocol will communicate with CACS when a significant event occurs such as starting a transaction, committing or aborting a transaction and reading and writing data. CACS then can maintain information that enables it to check for read or write conflicts when a transaction commits. On a commit CACS will inform the paging system to meld thereby establishing a new consistent state on non-volatile store. Once the changes made by the transaction become permanent they must be made visible to the other transactions. The meld involves writing the pages to disk changed by the transaction and atomically modifying the stable disk page table so that it reflects these changes thereby creating a new consistent recoverable global state. The differences between the shadowed pages modified by the transaction are then

propagated to any other transaction working on a copy of the same original pages.

4.3.2 Conceptual Concurrent Layered Architecture

The conceptual concurrent architecture is layered in such a way that it separates the concurrency control mechanism, the atomicity and the persistence. These layers are described in terms of conceptual address spaces together with a protocol that controls the movement of data between these address spaces. By unbundling and distributing the concurrency intrinsics in this hierarchy the architecture provides a generic layered store capable of supporting a number of different models of concurrency.

Figure 4.5 gives a diagram of the architectural hierarchy shown as a layer of conceptual address spaces. Each layer implements a separate and independent property of concurrency. This permits any particular concurrency model the flexibility to choose a desired combination of intrinsics.

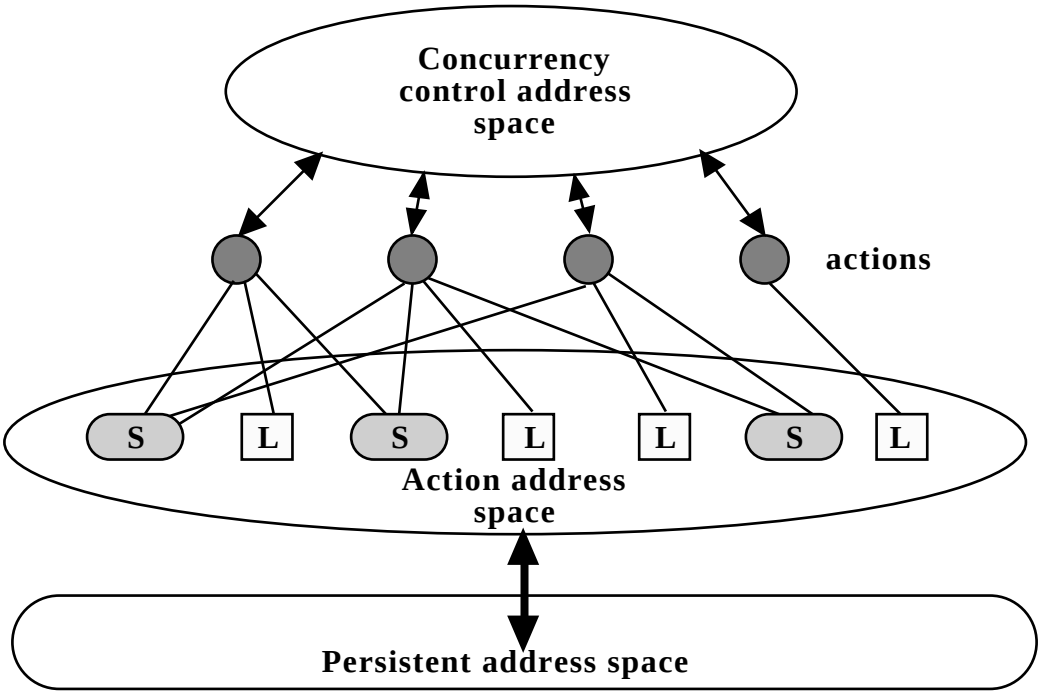


Figure 4.5: Conceptual Concurrent Architecture

At the top level is a logical address space that contains a full definition of the concurrency control mechanism. No assumptions are made by the lower layers about the concurrency control and hence this leaves the implementor freedom to choose any desired scheme. For example this may be realised in an implementation as a heap of objects incorporating two-phase locking or alternatively by CACS.

An action, in this architecture, is an isolated thread of control that communicates with the concurrency control address space and whose state is defined by the action address space. The action address space layer is a set of private and group address spaces. Each action has a private address space (marked “L” in figure 4.5) which is analogous with the local access set of CACS. In addition the action address space may house a number of group address spaces that are shared by a combination of actions (marked “S” in figure 4.5) which reflects the CACS shared access sets. Group address spaces can be hierarchic. The CACS database is modelled by the persistent address space.

The operations available at the interface allow the concurrency control scheme to create actions, abort actions and move data between address spaces. There is no in-built communication, synchronisation or serializability between actions and hence the concurrency control must define the movement of data between the group and local address spaces. The architecture ensures that all data movement is atomic. Movement of data from an action's private address space or a group of actions' group address space to the persistent address space is through a meld operation. The architecture supports the atomic update of the persistent address space so that its data is permanent, recoverable and consistent. The meld operation, under the control of the concurrency control, makes sure that data movement to the persistent address space becomes visible to all other actions. The effect of an action abort is to release the action's private address space. The relationship between action abort and group address space is determined by the concurrency control.

This architecture framework enables the separation of concurrency intrinsics into logical address spaces. Concurrency control is defined and contained within the concurrency control address space. Isolated and group actions operate within the action address space. Atomicity is provided by the action address space layer and permanence is handled by the persistent address space layer. This then provides the versatility to support a complete range of concurrency models. For example:-

- Support for atomic transactions can be provided in this architecture through a concurrency control specification that constrains each action's updates to their private address space thereby isolating their effects. Transaction commit involves the atomic update of the transient and persistent address space making the transaction's changes permanent and globally visible. Transaction abort is a trivial matter of discarding the private address space.

- Co-operative concurrency is viewed in this architecture as a single action since the interaction of co-operative activities do not require isolation control.

- Designer transactions models can be accommodated by a combination of private and group address spaces and a concurrency control specification that defines their creation, interaction and movement. Thus the effects of operations may be shared among actions without their objects being committed to the persistent address space.

4.3.2.1 Concurrent shadow paged store

It may be difficult to produce an efficient implementation of an architecture, especially on stock hardware, that involves the maintenance of a range of separate address spaces and the control of data movement between them. One possible solution arises from the observation that each layer need only maintain copies of the portions of the underlying layer's address space that it has changed along with a table which provides a mapping between the original and the copy. Applying this strategy down the layers, the resulting architecture collapses into one flat address space with a hierarchy of mappings. This address space can then be implemented as a recoverable paged virtual address space using an extended form of after-look shadow paging.

The shadow paging scheme works much as described before in section 2.5.1 whereby the virtual address space is mapped to non-volatile storage through a disk page table. Modified pages are shadowed and the transient disk page table reflects the current global state of the address space. In addition a separate disk page table is created for each private and group action address space used. Each action has its own private address space and so a disk-page table is created for each action. Similarly a disk page table is created for each group address space required by the model. Entries to the disk page tables are added for each page modified by the action. When an action first modifies a page a shadow copy is made and the action works on the copy. The concurrency control specification dictates whether the update is a private or group one. Hence the changes made by an action to its private address space are totally isolated from other actions'

private address spaces. Also the group address spaces are isolated from each other and from the private address spaces.

The architecture is illustrated in figure 4.6 and shows that the transient disk page table and the action disk page tables are accessible from the root page. As in the single threaded case the transient disk page table and the stable disk page table maintain a page table entry for each page of the address space. When a page, modified by an action, is written out to non-volatile store it is written to its shadow and the mapping recorded in the per-action page table. The action page table only has entries for pages modified by that action and not the complete address space. The illustration in figure 4.6 shows that there are five pages in the virtual address space and that there are currently two actions A and B. The disk page table for action A shows that A has modified pages 0 and 2 in its private address space and that action B has modified pages 0, 1 and 3 in its private address space. Note that page 0 has been modified by both actions but that the shadow page mechanism isolates their modifications. The third disk page table reflects a group address space that shows action A and B are working on a shared copy of page 2 and 4. CACS will disambiguate action A’s access to page 2.

The scheduler for this concurrent system must ensure that the correct mappings are established on a page fault. For example when action A accesses a page that results in a page fault, the system must search A’s disk page table for the page (or a group that A is currently in). If there is no entry for the page in A’s disk page table then the transient disk page table is searched.

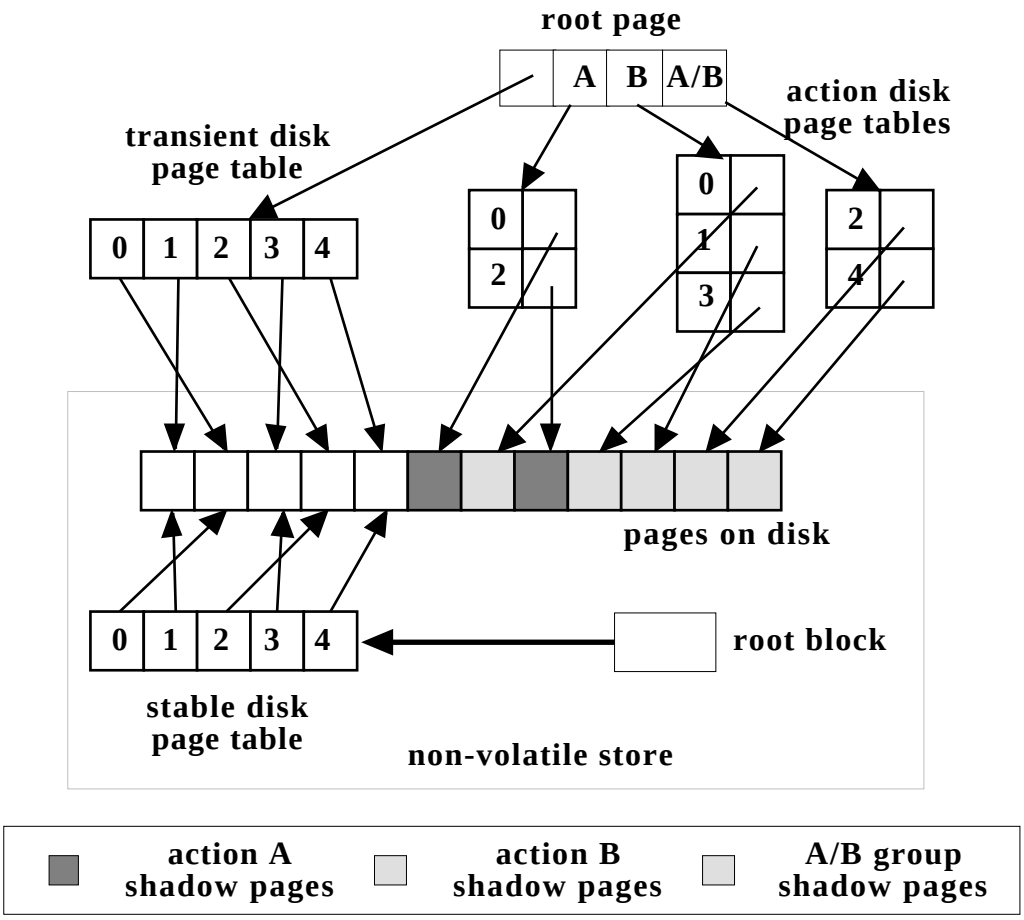


Figure 4.6: Concurrent shadow-paged architecture

A meld mechanism is provided on a per-action basis so that changes made by an action can become part of a new recoverable, consistent state and then these changes are made visible to other actions. It is possible for a number of actions to be working on shadow copies of the same page and the meld propagates the differences between the shadowed pages modified by the action and the originals through to any other action working on copies of the same original pages.

To establish a new consistent state all pages modified by the action are written to their shadows. Then the entries for these pages in the transient disk page table are updated to record the same mappings. For example, if a melding action had modified page P and its shadow page was disk block D then the transient disk page table entry for P must also record that it is mapped to D. To ensure atomicity of the meld the updating of the transient disk page table will involve

shadowing of the page encompassing the transient disk page table entry. Once the transient disk page table reflects the new consistent state it atomically replaces the stable disk page table.

4.3.2.2 Concurrency control and per-action melding

The algorithm that meld uses to propagate changes is dependent on the particular concurrency model in operation. For example suppose that two actions A and B share a page but modify different objects on that page. Because of the isolation of the concurrent shadow paging mechanism A can meld without affecting B. For B to meld it must retain the changes that A made. And so a mechanism is required for B to ingest the changes made by A. The approach taken is to link data access and the melding process in with the concurrency control through an implementation of the CACS system.

There are a number of ways this propagation could be implemented. For each page in its page table the action could record the address ranges that it has modified within that page as the changes are made. This could then be used to copy the modifications to other transactions holding a copy of the same page. An alternate method is *page diffing* as suggested by Wilson [SKW92] whereby a byte-by-byte comparison of a page is made with the original to determine the changes.

Alternatively logical operations can be used to propagate the changes. Suppose two actions A and B have changed different objects on the same page P and action A melds. The changes made by A to page P can be calculated by xor P onto the original page. These changes can now be xor'd onto action B's copy of page P. Thus B's version of page P now includes the changes made by A. This will only work provided that two transactions have not modified the same object. Provided that there is a mechanism elsewhere that prevents or detects such object

conflicts then this approach may be reasonably efficient and is the one that is adopted in the first implementation.

4.3.3 Atomic transactions in Napier88

The concurrent shadow paged store just described together with the per-action melding, threads, semaphores and the CACS system provide sufficient flexibility to enable a number of concurrency models to be constructed. The Napier88 system and CACS communicate with each other at three levels. At the language level the Napier88 program contains annotations which signal events, such as starting an action or committing an action, to the CACS system. The abstract machine level communicates with CACS to register the object reads and writes and CACS communicates with the store level to perform the meld. As an illustration an atomic transaction package has been built in this system.

The package uses a serializable schedule that is based on an optimistic version of the readers/writers protocol and observes the following :-

- An object O is always read by a transaction T_i before it is written.
- An object O modified by T_i and read by any T_j where $i \neq j$ will cause T_j to be aborted.

The protocol is optimistic in the sense that a transaction performs updates and only checks for conflicts with other transactions when it commits. If there are conflicts then the other transactions that have contributed to the conflict are aborted. This approach is possible because of the passive nature of the CACS system and the per-action shadow paging which isolates the changes made by running transactions (figure 4.4).

```
type transactionPack is abstype [tid] (
                        createTransaction : proc (-> tid);
                        beginTransaction : proc (tid, proc ());
                        commitTransaction : proc (tid);
                        abortTransaction : proc (tid);
                        readPid : proc (tid, pId);
                        writePid : proc (tid, pId))
                        )
```

Figure 4.7: The transaction package definition

The transaction package generates five events that are significant to the CACS system. These are begin transaction, commit transaction, abort transaction and read and write. These events are reflected in the transaction package definition shown in figure 4.7. The transaction package is declared as an abstract data type parameterised by the transaction identifier so that its structure cannot be discovered or impersonated. The *createTransaction* procedure registers a new transaction and returns a unique transaction identifier which the user then provides on calls to the other transaction procedures.

The *beginTransaction* procedure is also parameterised by a void procedure which is the code the transaction executes. The *beginTransaction* will signal the event to CACS which will communicate with the stable virtual memory system to create a new per-action shadow page table. Similarly the *abortTransaction* procedure will cause CACS to inform the system to discard the shadow page table.

The only shared data in this model is persistent data and sharing is done at the object level. Persistent objects in the stable heap are addressed through their persistent identifiers, or pids. The Napier88 architecture as shown in figure 4.4 uses a local heap as a cache of persistent objects. Both the local heap and the abstract machine work directly on virtual memory addresses and hence there are no explicit read and write procedures to move data between the local heap and the abstract machine. However, a convention is required so that the abstract

machine can distinguish local heap addresses from pids. This may arise for example if the abstract machine was to de-reference a field of a structure in the local heap that pointed to an object in the stable heap. Such a dereference causes the object to be copied from the stable heap to the local heap and effects an address translation or *swizzle* that overwrites its pid with the local heap address. When objects are written out from the local heap cache they are de-swizzled and have their local heap addresses replaced with pids and hence local heap objects carry their pid (figure 4.8).

Figure 4.8: Format of local heap object

The abstract machine can tell when a swizzle has occurred or when an object has been modified and can signal these events to the CACS system through the *readPid* and *writePid* procedures. CACS maintains a table of transactions and pids which can then be used to detect conflicts. When *commitTransaction* is communicated to CACS this table is checked for conflicts and any offending transaction aborted. CACS can then signal the stable virtual memory to perform a meld and propagate the changes to the other transactions using the double xor method described above.

Capturing and reporting the shared object accesses from the abstract machine highlights the most significant difference between this approach and the CPS-algol model [Kra87]. In CPS-algol the shared objects must be declared when the program is constructed and are wrapped up in a package of procedures that are used to monitor access to these objects. In the Napier88 atomic transaction system the user need not be concerned by shared objects and can access and manipulate objects normally. The user interface to the transaction package thus only has procedures for creating, starting, aborting and committing transactions as shown in figure 4.9.

```
type userTransactions is structure (
        createTransaction : proc (-> any);
        beginTransaction : proc (any, proc (any -> proc ()));
        abortTransaction, commitTransaction : proc (any))
```

Figure 4.9: User interface to the transaction package

The *createTransaction* procedure returns a value of type **any** which acts as a handle or key into the transaction package for that particular transaction. The structure of the **any** cannot be discovered by the user. This key is a value of the dynamic witness type for the *transactionPack* abstract data type injected into an infinite union. This reflects the way in which Napier88 enables values of witness types to escape their scope [Cut92].

4.4 Conclusions

The concept of concurrency can be viewed as a spectrum of understandability where points on the spectrum define levels of abstraction over the exposition of concurrent operation. The integration between concurrency and persistence is seen as one of incorporating this spectrum into the persistence model. This requires a persistent architecture that has the flexibility to support all styles of concurrency.

In an effort to understand how such an architecture might be constructed three different styles of concurrency that represent different points on the spectrum were discussed. The styles were categorised by how they specify and interpret the concurrency intrinsics of separate activity, sharing and understandability and how these interpretations inter-relate.

The approach to integrating concurrency into a persistent system presented here is to view understandability in terms of data visibility between separate actions. A conceptual concurrent layered architecture was described in which the

intrinsics of concurrency were separated into address spaces. In the model the visibility of data was equated to the sharing of address spaces and the movement of data between them. The motivation for this architecture arose from the specification system CACS.

From this design a persistent architecture for Napier88 that can support all styles of concurrency has been constructed. The system incorporates CACS with the persistent store acting as the database and uses a correspondence between concurrent shadow paging and the CACS access sets.

As an example of how the features of this new architecture can be combined the construction of an atomic transaction package was presented.

5 Implementation of Concurrency

5.1 Introduction

The issue of integrating concurrency into a persistent framework was seen as one of the linguistic provision and machine support for a range of styles of concurrency. In the previous chapter an architecture was presented which has the flexibility to enable any concurrency model to be built. As examples of this, two concurrency models in Napier88 which lie on opposite ends of the concurrency spectrum have been implemented :-

- A co-operating concurrency model based on lightweight threads and semaphores for synchronisation. The language level interfaces for these packages together with an example program is given in appendix A.

- A competitive concurrency model based on an atomic transaction package. This package, constructed in Napier88, builds transactions from the threads and semaphores and relies on a new concurrent shadow paged store to ensure isolation. The package employed the CACS system to monitor conflicts and to communicate with the stable store. Appendix B gives an annotated listing of the Napier88 code for this package.

This chapter is concerned with details of the implementation changes to Napier88 system to realise an architecture that can support all styles of concurrency and how these contrasting models were accommodated. The presentation of the implementation details is divided into two main sections. The first section deals exclusively with how the Napier88 system was adapted to provide the multithreading facilities. Because the threads are used as a basis for co-operating concurrency then there is no requirement to isolate the changes made by one executing thread from another. Hence the provision of

multithreading can be accommodated entirely within the language and abstract machine.

The second section is concerned with the implementation of the atomic transaction package and in particular with the concurrent shadow paged store. At present a full implementation of the CACS system is still being designed. In particular the area of the primitives and protocols required for CACS to communicate with the Napier88 system at various architectural levels outlined in section 4.3.3.4 are yet to be finalised. Towards this goal a new concurrent shadow paged store that directly supports the atomic transaction package has been designed and built. The implementation essentially hardwires in the handling of the significant events that would be passed to CACS. Such an approach allows for experimentation with the language level atomic transaction model and hopefully helps to make it clearer how a CACS system might be constructed.

5.2 Multithreading Implementation

5.2.1 Introduction

The implementation strategy in incorporating multithreading into the Napier88 system involves changes to the Persistent Abstract Machine (PAM) [CBC+90]. The Persistent Abstract Machine is primarily designed to support the Napier programming language. It is closely based on the PS-algol abstract machine [BCC+88], which in turn evolved from the S-algol abstract machine [BMM80]. As such the PAM is inherently single threaded.

The issues involved in adding multithreading functionality to PAM centre around :-

- the definition of a thread context.
- the creation and deletion of thread contexts.

- the scheduling of contexts.
- the abstract machine's interpretation of user control over threads.
- making threads persist.

The strategy for incorporating multithreading presented here is constrained to implementations of the abstract machine that operate on single processor machines. At any one time there is only ever one thread executing. This greatly simplifies the implementation. The decision to take this approach allowed for greater experimentation and testing of the language model. With the benefit of experience of persistent threads and a better feel for the desirable features of a thread context, future implementations may be designed to fit multiprocessor machines.

5.2.2 Semaphore Implementation

For experimentation purposes it was decided to implement the semaphore package in Napier88 in the standard environment rather than implementing a semaphore abstract machine primitive. This allowed for greater flexibility in that it is easier to modify its implementation. The package presents a general semaphore and its implementation is given in figure 5.1. The implementation uses the Napier88 primitive function *modlock* to achieve atomic update to the semaphore value. Details of this function and its implementation are given in section 5.2.6.

```
use threadPackage as X[ Thread ] in
begin

        rec type SemaphoreQ is variant( entry : SemaphoreStruct; empty : null )
        &
        SemaphoreStruct is structure( thread : Thread ; next : SemaphoreQ )

        let semaphoreGen = proc( initialValue : int -> SemaphorePack )
        begin
                let count := initialValue
                let Q := SemaphoreQ( empty : nil )

                let wait = proc( )
                begin
                        let dontCare = modlock( 1 )        ! Atomic update
                        count := count - 1
                        if count < 0 then
                        begin
                                let thisProc = X( getThreadId )()
                                ! Get id of this thread
                                ! Add this thread to the end of the SemaphoreQ
                                …
                                ! Suspend this thread release modlock atomically
                                let dontCare = modlock( -1 )
                                X( suspend )( thisProc )
                        end else { let dontCare = modlock( -1 ) }
                end

                let signal = proc( )
                begin
                        let dontCare = modlock( 1 )       ! Atomic update
                        count := count + 1
                        if count <= 0 then
                        begin
                                ! Now take a thread off front of the SemaphoreQ
                                …
                                ! Release lock and restart this thread
                                let dontCare = modlock( -1 )
                                X( restart )( thisProc )
                        end else { let dontCare = modlock( -1 ) }
                end

                SemaphorePack( wait,signal )
        end
end
```

Figure 5.1: Semaphore implementation

5.2.3 Persistent Abstract Machine

To understand how multithreading is built into the Napier88 system requires some background detail of the Persistent Abstract Machine (PAM) [CBC+90].

The machine is an integral part of the Napier88 layered architecture and interfaces cleanly with the persistent store. The abstract machine has the following distinguishing features :-

- a uniform representation of heap objects
- a block retention system
- a low-level type system
- a heap-based storage architecture
- a small number of machine registers

The PAM uses one object format for all heap items. This allows the utility programs such as garbage collectors and persistent object managers to be built in a manner that is independent of the programming language types. Heap objects have the following format :-

word 0	object header (includes the number of pointer fields)
word 1	the size in words of the object
word 2..n	the pointer fields
word n+1..	the non pointer fields

The block retention mechanism is required to support higher-order functions. The Napier language supports first-class procedures with free variables. To achieve the desired semantics, the locations of these variables may have to be preserved after their names are out of scope.

A primitive two-level type system within the machine contains enough information to allow machine instructions whose behaviour depends on the dynamic type of their operands. In conjunction with the block retention

architecture, the type system is used to provide an implementation of polymorphic procedures, abstract data types, and bounded universal quantification [MDC+91].

The abstract machine is built entirely upon a heap-based storage architecture. Although the machine was primarily designed to support a block-structured language, for which a stack implementation might be the obvious choice, the heap-based architecture was chosen as a convenient way of supporting the block retention. The PAM does not maintain its own store using the stable heap interface to the persistent store instead. This means that there is only one storage mechanism and one possible way of exhausting it.

Stacks are still used conceptually, and each stack frame is modelled as an individual data object. Stack frames represent the piece of stack required to implement each block or procedure execution of the source language. To aid garbage collection, a stack frame contains two separate stacks, one for pointers and one for non-pointers. The size of each frame can be determined statically.

The PAM uses five registers :-

- ROP - abstract machine root object pointer

  The special object, known as the root object for the abstract machine, is pointed to by the ROP register. The object contains, within its closure, all the housekeeping information required by the abstract machine, including the current state of any active programs, and a pointer field that is used as the root of persistence for user data.

- LFB - local frame base

  The persistent abstract machine implements a stack using a separate heap object for each stack frame, described below. A stack frame is created whenever a procedure is called or a block is executed. The LFB register is

used to point to the stack frame for the currently executing procedure or block (the local frame) and must be updated on every procedure call, procedure return, block entry and block exit. All local data may be accessed by indexing the LFB.

- LMSP and LPSP - local frame main and pointer stack tops

  In order to conform to a single object format, each object representing a stack frame actually contains two distinct stacks [Mor79b]. The pointer stack contains pointers and the main stack contains non-pointers. Within the local frame, pointed to by the LFB register, the LMSP register points to the top of the main stack and the LPSP register points to the top of the pointer stack. In fact the LMSP and LPSP registers point to the word following the last word on the appropriate stack. It should be noted that the LMSP and LPSP directly address the contents of a heap object. However, these registers are never stored in the persistent heap and are always recalculated whenever the LFB register is updated.

- CP - code pointer

  The next abstract machine instruction to be executed is directly addressed by the CP register. The CP register is similar to the LMSP and LPSP registers in that it is never stored in the persistent heap. Its contents are always recalculated whenever the object containing the abstract machine code is changed or moved.

A stack frame contains a pointer stack, a main stack, the relative positions of the stack tops with respect to the start of the frame and the relative position of the next instruction with respect to the start of the code vector. The relative positions are used to calculate the values of LMSP, LPSP and CP when a frame becomes the local frame. Similarly, the relative positions are recalculated whenever a frame ceases to be the local frame or a store operation is performed that may

move the local frame or the code vector. The format of a stack frame is shown in figure 5.2.

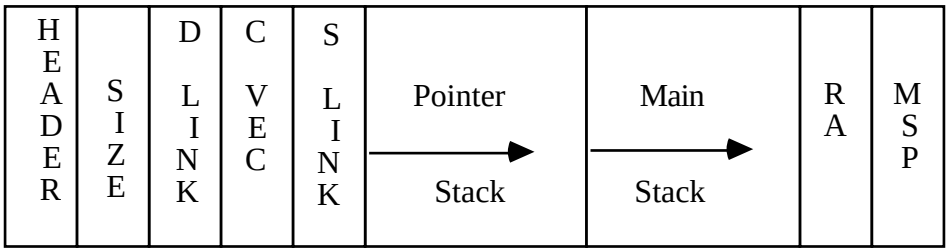


Figure 5.2: Stack frame

word 0,1 object header and size

word 2 the dynamic link (DLINK)

word 3 a pointer to the code vector for the frame's procedure (CVEC)

word 4 the static link for the frame's procedure (SLINK)

word 5..m the pointer stack for the frame's procedure

word m+1..n the main stack for the frame's procedure

word n+1 the resume address for the frame's procedure (RA), the saved offset (in bytes) of CP from the start of the procedure's code vector

word n+2 the saved offset (in words) of the LMSP from the LFB (MSP)

5.2.4 Definition of Thread Contexts

The inspiration for the multithreading implementation strategy comes from the observation that, on a checkpoint, the abstract machine saves the context of the currently executing procedure in the local frame and a pointer to that frame is saved in the PAM root object. On system startup the PAM root object and the local frame are fetched from the store. From the local frame, the machine registers are re-established and the procedure continues from exactly the point it

was at before the checkpoint. The multithreading implementation essentially generalises this method by the introduction of thread objects, or *throbs,* in PAM which capture the context of the executing procedure associated with each thread.

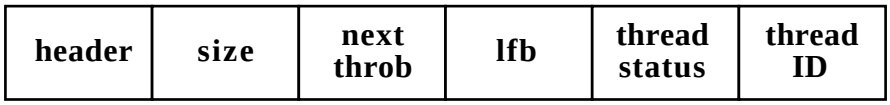


Figure 5.3: Thread context block

The format of the thread context block is given in figure 5.3 and shows that the throbs conform to the heap object format.

word 0,1 object header and size

word 2 pointer to next thread context block

word 3 a pointer to local frame base containing the current context for this thread's procedure

word 4 current status of the thread. E.g., suspend, runnable etc.

word 5 integer thread identification

In the initial implementation thread context blocks are held as a single linked list where the head of the list is stored in a field of the PAM root object. This was done purely for simplicity to ease the implementation and allow for experimentation.

5.2.5 Thread Context Block Creation

A new thread context block is created every time the user starts a new thread. The user interface is through an abstract data type, *ThreadPack*, as shown in figure A.1. The interface allows a user to start a new thread with an associated

void procedure, kill, suspend and resume threads and find the id of the current thread.

The *ThreadPack* abstract data type is defined in the Napier88 standard environment where it is specialised to integer. Each of the procedures of the *ThreadPack* calls a primitive Napier88 function which executes an appropriate PAM instruction. Rather than add a new PAM instruction for each procedure of the thread pack the implementation introduces one new instruction, *threadOp*, which is parameterised to distinguish each thread operation.

When a new thread is started, a new stack frame object and a new throb is allocated from the heap. The static link and code vector for the thread's procedure are taken from the stack and placed in the corresponding fields of the new frame. The dynamic link field is set to nil so that the thread can exist independently of its caller. A pointer to this frame is saved in the new throb and the thread status is marked as ready to run. The identification number of the thread is obtained from the root object. The root object then increments this number.

5.2.6 Context Switching

The multithreading implementation requires a scheduler to control the execution of separate threads. One possibility would have been to execute the threads as SunOS lightweight threads and let the target machine perform its own scheduling. It was felt that initially it would more beneficial for experimentation to retain full control over scheduling threads by writing a scheduler within the abstract machine. This avoided problems that might have arisen in the interplay between the execution of PAM and the SunOS scheduler. For example the scheduler can easily ensure that a context switch does not happen in the middle of a PAM instruction.

By incorporating a scheduler into the abstract machine the atomicity of PAM instructions can be preserved. There are however two critical sections in Napier88 where it is necessary for a sequence of PAM instructions to be executed indivisibly with respect to context switching :-

- Update of a semaphore value must be done atomically. The semaphore implementation is written in Napier88, as shown in section 5.2.2, and hence the semaphore value update may involve the execution of a number of PAM instructions. Context switching in the middle of this sequence of instructions must be prevented.

- Similarly the implementation of environments is also written in Napier88 and hence environment updates need to be atomic with respect to context switching.

Such critical sections are identified by enclosing them in calls to the primitive function *modlock*. This function sets a global lock that the scheduler tests before context switching. The lock is maintained as an integer count rather than a lock bit and its value is incremented when current executing thread enters a critical section and decremented when it leaves. The scheduler will not context switch when the lock count is positive. A lock count is used so that the currently executing thread can nest through critical sections. It is recognised that the use of a global lock for the semaphore value update may cause an unnecessary bottleneck in the store. This approach was taken for simplicity of implementation to quickly produce a system that could be used to test the language model. Future implementations will consider alternatives such as using a Unix system semaphore.

It should be pointed out that the *modlock* operation is strictly dependent on a single processor machine. Because there is only one currently executing thread it will always “get” the lock and never be halted waiting for it. The lock count is

held in the PAM root object so that its value will be automatically saved and restored on a checkpoint.

The scheduler forms part of the PAM instruction decode loop. The decode loop extracts the next current instruction from the code pointer, increments the code pointer and calls a procedure to handle that particular instruction. After each instruction has been executed the decode loop checks to see if an asynchronous event that needs attention has happened during the execution of the last instruction, such as input from the keyboard. These events are dealt with before the next instruction is extracted and dispatched. At this point the scheduler determines if a context switch should happen. The scheduler can be forced to context switch or it may decide itself to switch. A forced context switch is flagged by the last executed instruction setting a thread signal. This will happen for example when the user suspends the current thread. The scheduler will undertake a context switch based on time-slicing and provided the lock count is zero. Thread execution is time-sliced by the number of PAM instructions it has executed. Currently the number of instructions, determining the time quantum, is fixed but experiments and future implementations may make this a persistent self-modifying variable dependent on execution analysis.

The context switch involves first searching the list of threads for one that is not suspended. The scheduler employs a round robin algorithm by placing the current thread on the end of the list and making the first waiting thread found that is ready to run the current thread. Before placing the current thread on the end of the queue its context, the code pointer and stack pointers, is saved in the local frame base and the pointer to this frame is stored in the local frame base field of the current throb. To complete the context switch the PAM machine registers are then loaded from the new current thread context block. The decode loop will now execute the instructions from the code for the procedure associated with this new thread from the point where it was suspended.

When a thread's procedure completes the thread context block is automatically removed from the list of threads. This can be detected by the interpreter at the point where a return instruction is executed and there is no dynamic link. The scheduler will then switch to another thread. If there are no more threads or if all existing ones are suspended and not ready to run then the state of the machine, including the suspended threads, is checkpointed and the current session terminates.

5.2.7 User-control of Threads

As explained above the user can control the threads through suspend, kill and restart procedures. Calls to these result in the execution of the *threadOp* instruction parameterised to indicate which procedure is being called.

A suspend or restart *threadOp* instruction results in the status of the specified thread recording the new state. If the specified thread is the currently executing thread then the instruction sets a flag to inform the scheduler to context switch immediately.

A kill thread instruction simply removes the specified thread's context block from the list of throbs. If the currently executing thread is being killed then the suicide is reported to the scheduler which then forces a context switch.

At present no attempt is made to report errors in control of the threads back to the user. Examples of the kind of errors that can arise include trying to kill a thread that was already killed or restarting a process that was not suspended. It is not clear whether it is vital to report such situations or to take the approach here and just ignore the errors. One suggestion that may help is for each thread to have the functionality to provide its own error reporting. At present the root object contains a pointer to a vector of event handling procedures and a pointer to a structure of error handling procedures. It would not be difficult to provide these on a per-thread basis.

5.2.8 Persistent Threads

When a checkpoint is initiated the state of the machine must be saved onto non-volatile store and restored on startup. By including a pointer to the list of threads in the root object the checkpoint will automatically save their state since it forms part of the closure of the root object. On startup the list of throbs must be pulled from the store for the scheduler to resume execution.

It is rather easy for the user to construct either deliberately or inadvertently a runaway thread for which the user has no handle or to create suspended threads that cannot be awoken through mismatch of wait and signals on a semaphore. These threads however will persist since they are still in the closure of the root object. One possible solution is to make provision for a “super-user” or system control that supplies a handle on all threads hanging off the root thus allowing unwanted threads to be killed. This implies that such threads can be identified and that it may also be useful to record in the throb statistical information such as date and time started, number of context switches and number of instructions executed. All this of course tends to fatten the lightweight thread.

5.2.9 Threads and I/O

Textbook lightweight threads usually operate with minimum context and independently of their parents. It is often in the area of inherited features from parents that distinguishes heavyweight from lightweight threads. One particular area concerns the problem of whether files or devices opened by a parent can be accessed by a child thread. There are essentially two options; either a thread inherits its parent’s open file descriptors or it doesn’t. The problem in this Napier88 thread implementation is that control over access to such file descriptors is determined by the closure of the procedure the thread is executing. The program fragment Figure 5.4 illustrates the problem. A file is opened in an outer scope of a procedure which will executed by a thread. The thread can

correctly reference the file descriptor since it is in its closure and hence can close the file. If the semantics define that a thread inherits the parent's open file descriptors then the success of the read procedure call is dependent on whether it is executed before or after the close call in the *threadProc* procedure.

If the semantics are the opposite so that a thread does not inherit the open file descriptors of its parent then there is a dilemma. The close of the file descriptor in the *threadProc* will cause an exception since that thread never opened the file yet the program is still perfectly valid Napier88. If *threadProc* were called as a procedure rather than executed as a separate thread then there would be no exception.

```
let f = open( "AFile", 0 )
let threadProc = proc()
begin
        …
        close( f )
        …
end
…
aThread( start )( threadProc )
let noOfBytes = read( f, … )
```

Figure 5.4: Threads and file I/O

5.2.10 Comments

Future implementations will be based on a thread object format shown in figure 5.5. This format has been formed in collaboration with Casper project at the University of Adelaide [KSD+91]. The Casper project is a system which provides a shared persistent store in a distributed environment where client processes execute separate threads against the shared store. The system is described in more detail in chapter 6. Thread context blocks will be kept in a vector structure with a pointer to the vector from the root object. The threads will

have the capacity to define their own event handler and error handler procedures but will default to the system-provided procedures.

```
Throb =
begin
        eventProc       ! per process pointer to event handler routines
        errorProc       ! per process pointer to error handler routines
        openFiles       ! per process pointer to open files
        saveLFB         ! save LFB on context switch
        semaphore       ! pointer to semaphore the process is currently blocked on
        copyout         ! Casper use, pointer to Casper special data structs
        throbId         ! thread Id
        lastIO          ! last I/O error
        numberObj       ! Casper use, number of objects in heap
        lheapStart      ! Casper use, VM addr of start of local heap
        lheapEnd        ! Casper use, VM addr of end of local heap
        lheapP          ! Casper use, VM addr of heap alloc pointer
        remSetBase      ! Casper use, VM addr of end of remembered set
        remSetP         ! Casper use, VM addr of alloc pointer of rem set.
end
```

Figure 5.5: New thread object format

5.3 Concurrent Persistent Object Store Implementation

5.3.1 Introduction

One of the principal aims in implementing the concurrently accessible persistent object store was to preserve as much of the existing Napier88 store technology as possible. This is evident in that the same architectural abstractions of a stable heap of objects operating on top of a stable virtual memory (SVM) are preserved as are the majority of the interface functions. The general model of the architecture was shown figure 4.4 where the Napier88 layered architecture communicated with CACS in three different levels. At the language level the annotated Napier88 programs were seen as CACS event generators. The abstract machine informed CACS of read/write operations on objects. The store provided the CACS visibility structures with the persistent store acting as the CACS

shared database and a collection of shadow pages representing the CACS access sets. Figure 4.6 illustrated the resulting architecture.

Ideally the store implementation would have been built with a flexible paging strategy and meld mechanism that communicated with an implementation of the CACS system for support. However in the absence of a full CACS implementation, the store has been constructed with explicit support for the transaction model discussed in section 4.3.3. The transaction package is written in Napier88 (appendix B) which implements a version of conflict serializability as concurrency control. The abstract machine traps reads on pid translation and writes on local heap updates. The store has a built-in melding mechanism based on the double xor function described in 4.3.2.2. The transactions operate over a shared shadow paged stable virtual memory address space. At the language level the transactions are constructed using the Napier88 threads and thus the interleaving of transaction execution is controlled by the same scheduler as the one described in section 5.2.5.

The implementation described here is one that was produced under the SunOS operating system and makes extensive use of the memory-mapping facilities detailed in chapter 3. Hence it is subject to the same benefits and drawbacks of the operating system's memory-mapping facilities as the store implementation described in chapter 3.

5.3.2 Overview

The concurrent after-look shadow paging scheme works on a per-transaction basis. For each page that is shared by a number of transactions there may be, at any time, a number of shadows of that page residing in different disk blocks. Figure 5.6 shows three shadow pages mapped to a page P in the SVM. Transactions T1 and T2 have modified the page and hence have their own shadows and there is also a version of the page on non-volatile store that reflects

the last consistent state of the page. For each running transaction a shadow page is created when the transaction modifies a page and each transaction maintains its own mapping table (the per-transaction mapping table) that describes which pages the transaction has modified and where these pages are in the backing store.

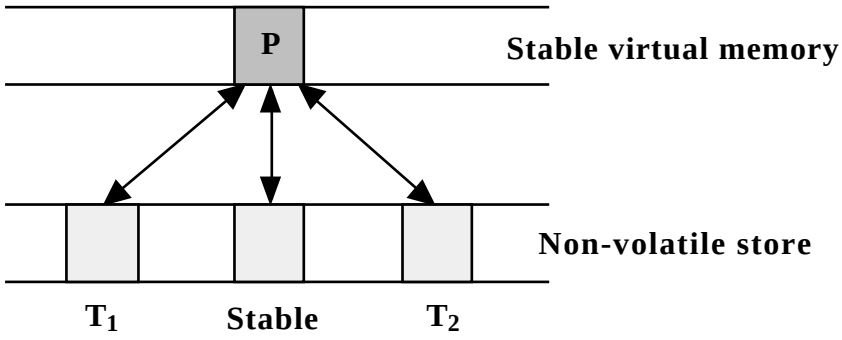


Figure 5.6: Shared page mapped to different shadows

When the scheduler is context switched to a thread of a different transaction, i.e., a transaction context switch, the PAM informs the store which then uses the mapping table for the incoming transaction to map the modified pages of this transaction back from the shadow pages into the SVM. In the SunOS implementation it is necessary to write back all pages of the outgoing transaction that were modified during a time slice to their shadows at a context switch. This is because the memory-mapping does not provide access to the page-replacement mechanism.

The per-transaction shadowing thus provides a method for isolating the changes made by transactions without the necessity to resort to page-level locking. Transaction abort is a trivial matter in this scheme because of the isolation of changes. Transaction commit first establishes a new consistent state through a transaction-based meld and then ensures that the changes made by the transaction are propagated to other active transactions. Establishing a new consistent state effectively involves the atomic update of the stable disk page table to reflect the changes made by the committing transaction. For example if

transaction T1 in figure 5.6 committed then the stable disk page table would be modified to reflect that page P was now mapped to T1's shadow disk block. Because the atomic transaction implementation is hardwired into this store the changes made by the committing transaction are propagated to other transaction using the double xor mechanism outlined in section 4.3.3.2. It is also assumed that conflicts arising where two transactions have modified the same object on a page are resolved by CACS conflict serializability.

The motivation for considering a concurrent shadow paged store is based on the belief that shadow paging is a better method than the alternatives for persistent systems that are expected to exhibit a high degree of locality and regard code as data. The design and implementation of this particular store was derived from the single-threaded store design presented in chapter 3. One of the features of that implementation was the introduction of disk page tables to record the mappings between the SVM and the non-volatile store. These mappings were created on demand and hence the order of pages in the SVM address space is independent of the order of disk blocks on the non-volatile storage. The concurrent implementation essentially generalises this approach by maintaining a mapping table per transaction.

5.3.3 Concurrent Shadow-paged Stable Virtual Memory

Much of the SVM layout is similar to the single-threaded store. Figure 5.7 illustrates the layout. The root pages, disk page table, data pages and block list all perform the same functions as before. The stable store file which is used to provide the disk blocks of non-volatile store is again one that grows on demand.

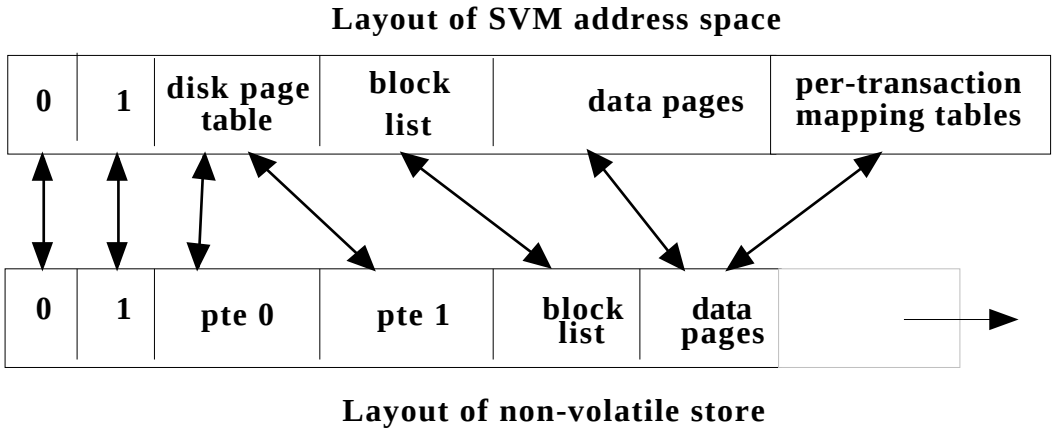


Figure 5.7: Layout of the stable store

At the end of the SVM, space is reserved for the per-transaction mapping tables. The SVM is informed through its interface when a new transaction is created by the user. The SVM allocates a page in this area which is then used as a table for the mappings of the shadows of pages modified by the transaction. This page is thus an array of per-transaction mapping entries. Each entry is a two word structure where the first word records the SVM address of the modified page and the second word records the disk block where the page is written out to. The entry also records status information about the page and its shadow in the flags field (figure 5.8).

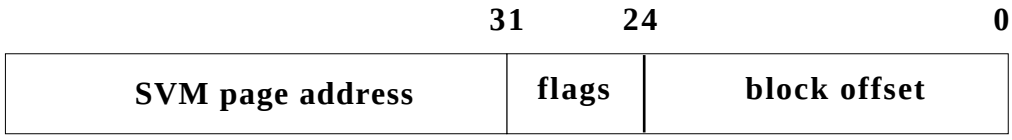


Figure 5.8: Per-transaction mapping table entry

In addition to the fields used in the single-threaded store the root page records the next transaction identification number and the current transaction id and maintains a free list for the pages of the per-transaction mapping tables. It also houses the disk page table entries for the pages of the per-transaction mapping tables. The format of the new fields of the root page and the disk page tables for the pages of the per-transaction mapping tables pages is shown in figure 5.9.

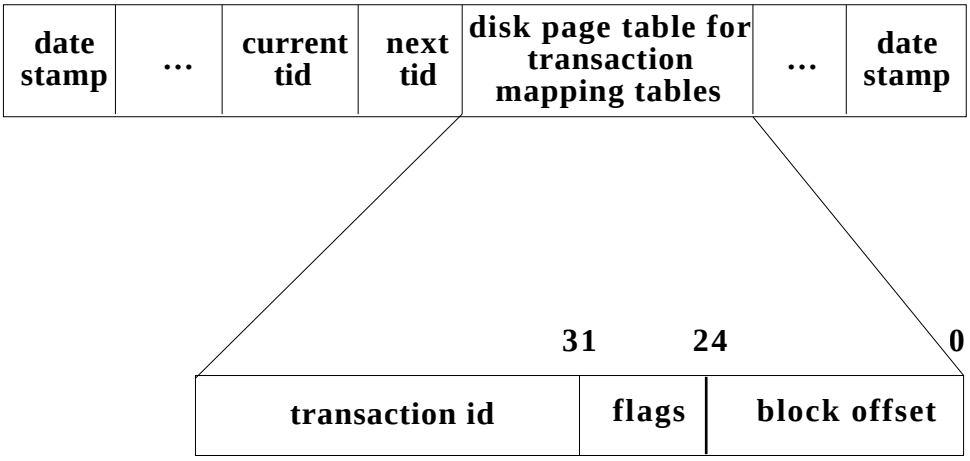


Figure 5.9: Root page

The page size and data area size is the same as the single-threaded store. With the new additions the root page has sufficient space to accommodate a page table for 960 pages of per-transaction mapping tables.

5.3.3.1 Store access

Unlike the previous implementation the mappings between the disk blocks of non-volatile store and the pages of the SVM are established on demand. This allows for a much cheaper context switching as explained in section 5.3.3.2.

The SVM traps a signal from the operating system when an address is accessed for a page that is not mapped to a disk block. To resolve such a signal the SVM page-fault handler must first check the per-transaction mapping table of the currently executing transaction to determine if that address lies in a page that has been modified by the transaction. If an entry is found then the table is used to establish a mapping between the disk block which contains the transaction's copy of the page and the SVM page. If there is no entry for that page in the per-transaction mapping table the mapping from the transient disk page table is used instead.

The first time a page is modified a shadow page is found from an unallocated disk block in the stable store file. If none are found then the store file is

extended. The page is then copied to its shadow page and the shadow page memory-mapped to the page. The per-transaction mapping table for this transaction then records the mapping in a new table entry. The flags field of this mapping table entry records that the page has been modified and shadowed. The shadow flag determines if a page has been modified since the start of the transaction whereas the modified flag indicates if the page has been modified during the current time slice. The address of the page is added to the block list which records the block order mappings for pages modified by the transaction.

5.3.3.2 Transaction context switch

The scheduler in the abstract machine informs the store when a transaction context switch is about to happen. Because the SunOS memory-mapping does not allow access to the page-replacement mechanism then all the pages modified by the transaction during its time slice are written back to their shadows and the mapping table entries for these pages are then marked as unmodified. These pages are easily found since the block list records the modified pages in block order. To complete the context switch the block list is cleared and the pages that were memory-mapped in during the time slice have their mappings disestablished.

When the next transaction starts to execute the store will have no mappings established for any of the data pages and the SVM page fault handler will map these in on demand.

5.3.3.3 Transaction abort

The SVM traverses the aborting transaction’s disk mapping table entries. The disk blocks in non-volatile store used for the transaction’s shadows are marked as unallocated. Finally the page containing the mapping table entries for the transaction is marked as available.

5.3.3.4 Transaction commit

The commit mechanism is performed in two stages. The first stage establishes a new consistent, recoverable state on non-volatile store and the second stage handles the propagation of changes made by the committing transaction to other transactions.

To establish a new consistent state the SVM first writes out all pages modified by the transaction to their shadows. Then the entries for these pages in the transient disk page table are updated to record the same mappings. For example, if the committing transaction had modified page P and its shadow page was disk block D then the transient disk page table entry for P must also record that it is mapped to D. To ensure atomicity of the commit the updating of the transient disk page table will involve shadowing of the page encompassing the transient disk page table entry. This shadow is recorded in the secondary page table in the root page. Once this has been done for all pages modified by the transaction a new consistent state is established by atomically writing back the root page.

Now that a consistent state is safely on non-volatile store the original disk blocks, that were mapped to the pages which were shadowed by the transaction, would usually be marked for re-use. However each of these disk blocks contains an image of the pages as they were before the transaction modified them. These are needed to propagate the changes to other transactions that are sharing pages. With the after-look mechanism there are always two recoverable consistent states on the non-volatile store immediately after a meld. From the most recent root block the new consistent state can be traversed and the previous consistent state can be found from a traversal of the other root block.

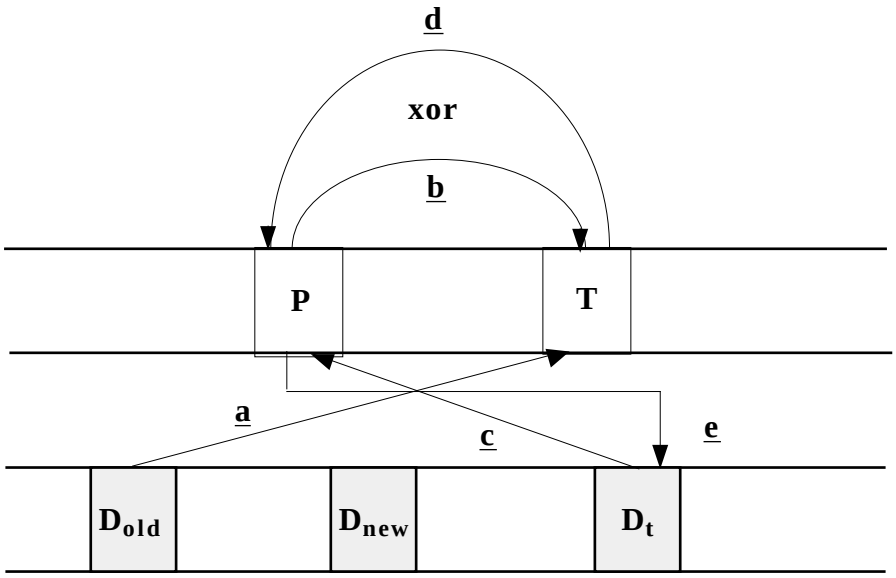


Figure 5.10: Propagating the changes

The sequence of events that are used to propagate are illustrated in figure 5.10. Page P has been modified by the committing transaction. D_{old} is the disk block that holds the previous consistent state of P and D_{new} holds the new consistent state of P. For each page P modified by the transaction the propagation proceeds as follows :-

a) D_{old} is found by traversing the secondary and primary page from the old root block. D_{old} is then mapped into an unused page T in the SVM.

b) Page P is then xor'd onto T. T now contains the differences to page P made by the transaction.

For each transaction that has also modified P :-

c) Its shadow, D_t, is found from the per-transaction mapping table and mapped into P.

d) T is then xor'd onto P. Page P now has incorporated the changes made by the committing transaction.

e) Page P is written back to its shadow, D_t.

5.3.4 Stable Heap Implementation

In making the persistent store concurrently accessible, changes have been made to the stable heap layer. The stable heap layer maintains a heap of persistent objects that is visible to the programming language level. The interface provides a number of persistent object management functions that enable the programming language access to the persistent store. These functions include the ability to create and delete objects, a checkpointing procedure to stabilise the persistent store and a procedure to invoke a garbage collector.

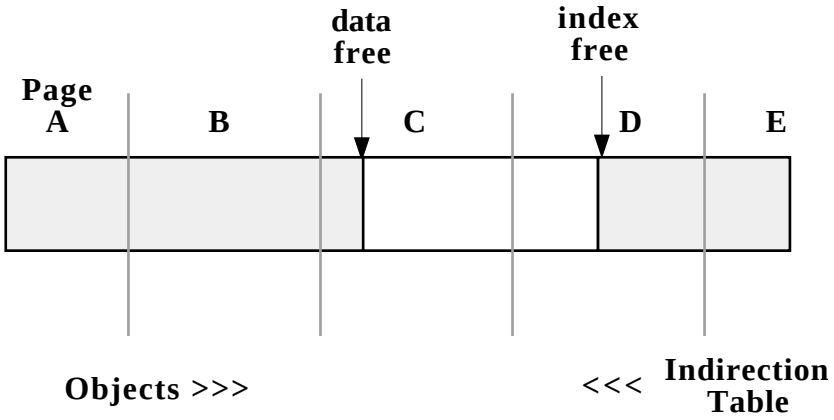


Figure 5.11: The stable heap layout.

The address range of the stable heap is defined by the data area of the SVM and its layout is shown in figure 5.11. The stable heap is split into two distinct areas - an object area and an indirection table. The object area, which contains heap objects, starts at the low address end of the heap and grows towards the high address end. For each object in the heap there is a corresponding two word entry in the indirection table. Entries in this table start at the high address end of the heap and grow towards the low address end.

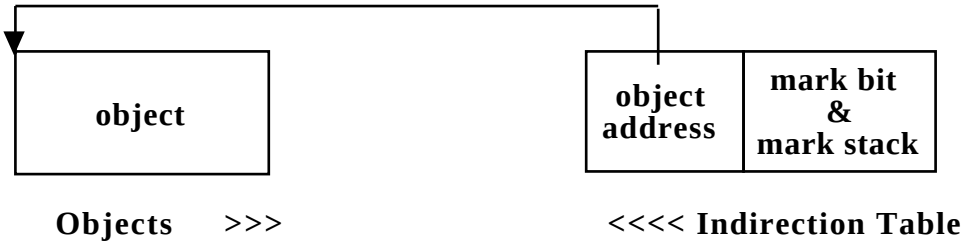


Figure 5.12: Object allocation

The heap is implemented using indirect addressing. When a new object is created it is allocated space from the object area along with two words from the indirection table. The first word of the entry is set to point to the object as shown in figure 5.12. The logical address of this entry is called the object's **key** and is the address used by the abstract machine to refer to the object. Indirect addressing simplifies the compacting garbage collection since all references to an object are indirect. The second word of each entry in the indirection table is used during the marking phases of the garbage collection.

The main problem in adding concurrency to the stable heap concerns the strategy for allocating new objects. The stable heap occupies the data area of the shared paged SVM. With the context switching mechanism described above the state of the stable heap, at any one time, reflects the view of the currently executing transaction. The *data_free* and *index_free* pointers are held on a global basis and new objects are allocated from the ends of these variables. This avoids introducing conflicts where the allocation of new objects might lead to transactions modifying the same areas on the same page. This implies that from a transaction point of view the allocation of new objects is not contiguous. This leads to a situation as shown in figure 5.13 where from the point of view of the current transaction there are a number of “holes “which, in fact, are objects on the page that have been allocated to other transactions.

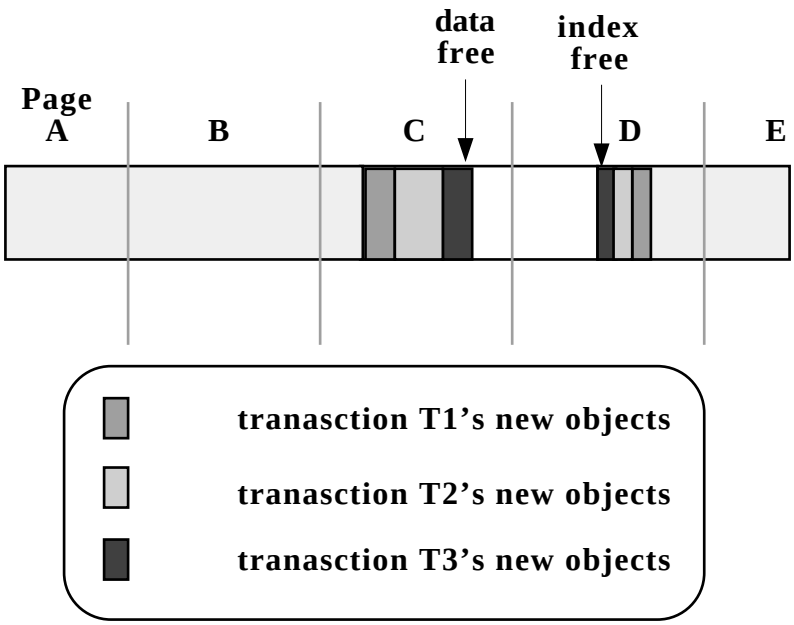


Figure 5.13: Allocation of new objects

Allocating new objects using this mechanism leads to a difficulty in designing a compacting garbage collector. At any one time the heap contains the state for the current transaction. This implies that marking all reachable objects, not just those of the current transaction, and then compacting can only be performed at the heap level if it takes into account all running transactions. A suitable collector is still being designed and the present implementation uses the compacting collector that existed in the single-threaded store but only performs this off-line when there are no outstanding transactions.

5.3.5 Conflict Resolution

The atomic transaction model outlined in section 4.3.4 used an optimistic concurrency control based on conflict serializability. A transaction could commit first and then check for conflicts with other transactions. Any transaction found that conflicted was aborted. In the model the only shared data would be persistent data The abstract machine detected when a transaction read an object in from the stable heap by trapping when the object was swizzled to a local heap address. The abstract machine trapped a transaction write of a persistent object

by detecting when an object with a persistent id (pid) was modified on the local heap. These events were reported to CACS which could be queried on a transaction commit for conflicts. For the present this package can be considered as being CACS and hence the detection and resolution of conflicts must be implemented as part of the package.

The PAM operates over a local heap that has two main purposes. The first is to gain some efficiency by providing a cache of persistent objects. The second is to provide an area of storage where new objects can be created. Hence one of the local heap's principal functions is to control the movement of data to and from the stable heap. All new objects are created in the local heap. These objects only migrate to the stable heap after a meld has determined that they are reachable from the root of persistence. This then prevents the overhead of unnecessarily allocating whatever resources are required in creating objects in the stable heap. The local heap is constructed in such a way that it can be garbage collected independently of the stable heap and since a great many new objects are transient they can be efficiently collected on the local heap.

The local heap is divided into two areas. Objects are allocated space contiguously from the low address end growing to the high end address. A mapping table is used to map the addresses of stable heap objects, known as pids or *keys*, that have been cached to their local heap addresses. The entries in this mapping table start at the high address and grow towards the low end. An entry is added to this table for each object cached in the local heap. This table is called the key to RAM address table or KRT.

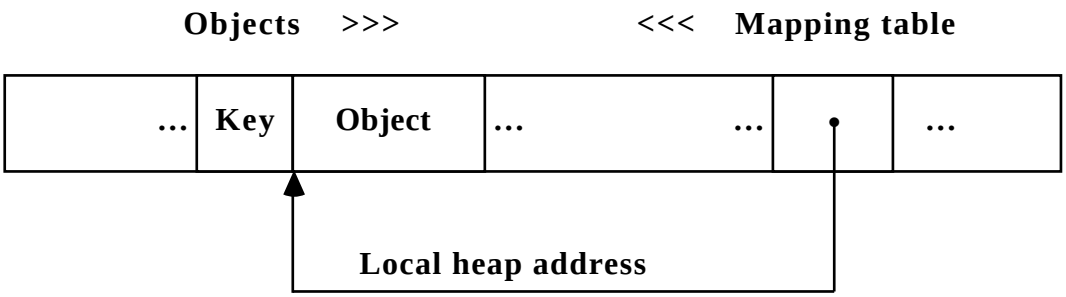


Figure 5.14: The layout of the local heap

The layout of the local heap is shown in figure 5.14. When an object is brought into the local heap from the stable heap an extra word is allocated. This extra word prefixes the object and is used to hold the object's key. This provides the reverse mappings from local heap addresses to keys. This is used when objects are copied back to the stable heap.

With the introduction of concurrency the transactions work over a shared local heap and thus may cache objects with the same key. There is therefore a need to isolate the changes made by one transaction on an object in the local heap from the changes made by another. One approach is to cache the objects in on a per-transaction basis hence potentially duplicating objects in the local heap. This effectively involves parameterising the KRT entries by transaction id. Figure 5.15 illustrates the idea.

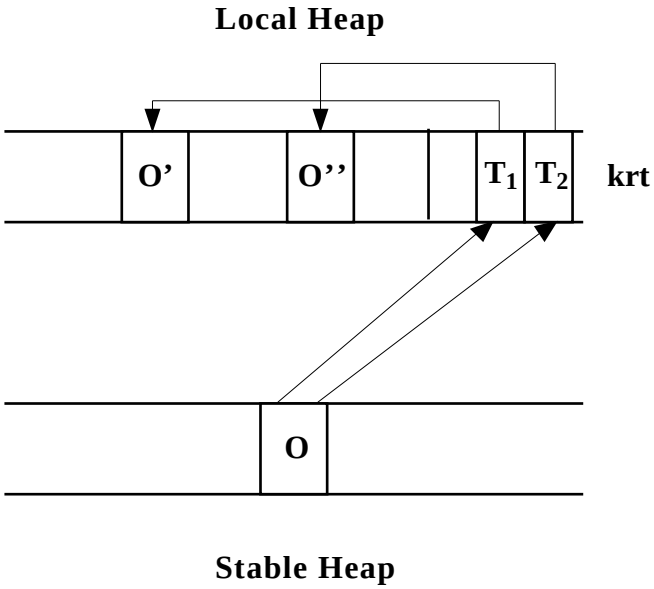


Figure 5.15: Per-transaction caching

The atomic transaction implementation at the language level maintains it own record of objects that have been read and written to and from the stable heap. The current set of transactions is kept as a cons list and within each transaction two binary trees are constructed that record the persistent objects that have been read or written. Entries in the binary tree are indexed by the object's key (figure 5.16).

```
type pId is int

rec type pidIndex is variant( node : Node ; tip : null )
& Node is structure( key : pId ; left,right : pidIndex )

type tId is int

type transaction is structure( tid : tId ; thread : any ;
                               readPids,writePids : pidIndex )

rec type transactionList is variant( cons : Cons; tip : null )
& Cons is structure( hd : transaction ; tl : transactionList )
```

Figure 5.16: Transaction data structure

Entries to the transaction's trees of persistent object accesses are through two procedures *readPid* and *writePid* which use a mutual exclusion semaphore to

ensure an atomic update to the binary trees (figure 5.17). These procedures must be called from the abstract machine when it detects a swizzle or write of a persistent object. In the current implementation this is done by adding these two procedures to the events vector in the events environment.

```
let readPid = proc( tid,pid : pId )
begin
        wait()
        let this = getTransaction( transactionsList,tid )
        if this ≠ emptyTransactionList do
            this'cons ( hd,readPids ) := pidEnter( pid,this'cons( hd,readPids ) )
        signal()
end

let writePid = proc( tid,pid : pId )
begin
        wait()
        let this = getTransaction( transactionsList,tid )
        if this ≠ emptyTransactionList do
            this'cons ( hd,writePids ) := pidEnter( pid,this'cons( hd,writePids ) )
        signal()
end
```

Figure 5.17: Recording persistent object accesses

After a commit the *pidIndex* binary trees of each transaction can be interrogated to look for read and write conflicts.

Appendix B gives a complete listing of the transaction package.

5.3.6 Comments

One of the drawbacks of the transaction implementation just described is that it involves object duplication in the local heap and page duplication in the stable store. This seems a little excessive. The problem arises since the transactions and PAM operate over a shared local heap and local heap is a cache for a shared stable heap. The movement of data between these heaps is also of significance to the atomic transaction model. There are two possible ways of avoiding this duplication :-

- The first is to use a single-threaded store like the one described in chapter 3 and to devise a competitive concurrency scheme that exists only in the local heap. This could be achieved provided data only moved from the local heap to the stable heap when a transaction committed. This would restrict the computation and data of transactions to the size of the local heap. This approach would also not take advantage of the perceived benefits of using shadow paging for a concurrent orthogonally persistent system and could not employ the fast xor mechanism for propagating changes.

  This would alter the relationship between the local heap and stable heap. The local heap would no longer act as merely a cache and an area for new data. The persistent object store architecture has always provided a stable heap that can operate independently of a local heap. Many of the languages that use the existing store such as Staple, Galileo and Quest do not use a local heap and so would gain nothing from this approach.

- The second solution is to abandon Napier88's use of the local heap and to operate the PAM directly on top of the stable heap. This requires a significant rewrite of the abstract machine since it expects to manipulate direct virtual memory addresses and not object keys. Much of the performance of the PS-algol/CPOMS system was believed to be derived from the address translation mechanisms that minimises the number of checks required.

  However this approach is worth pursuing - even if to provide some basis for comparing two approaches. It is difficult to quantify the benefit of using a local heap. The advantage of an area to create new objects that can be independently garbage collected may be offset by the cost of memory-to-memory copies from one heap to another.

5.4 Conclusions

The incorporation of two different models of concurrency into the Napier88 system has required a number of changes to the abstract machine and object store. The support for a co-operating concurrency model based on threads and semaphores was provided through the creation and manipulation of thread context blocks. Each context block can be thought of as an abstract machine root object for the procedure the thread is executing. The thread context block is then a small structure that contains a pointer to the stack frame for the thread's procedure and a unique identifier for the thread. These threads are autonomous and can be nested to any depth.

The scheduling of threads is handled within the decode loop of the abstract machine and is constrained to executing only one thread at a time. This allowed for an implementation to be fairly readily constructed and enables experimentation with the threads and semaphores language model.

Providing support for the atomic transaction model has meant a significant change to the operation of the object store. In essence the transactions operate over an after-look shared stable virtual memory. Each transaction has its own mapping table which is used to keep shadows of pages it has modified. Thus the effects of one transaction are isolated from another. On a commit the changes are propagated to other transactions sharing a page using bitwise logical operations on the page. The implementation presented here uses the same scheduler as the threads package and is subject to the constraint of only being able to execute one transaction at any one time.

The design and implementation of the threads and concurrent shadow pages store has shown up a number of possible enhancements. With persistent threads there is a need to be able to identify and remove runaway and permanently suspended threads. This may be provided by allowing some sort of privileged access to the

thread queue structures. The concurrent shadow page store and its relationship to the local heap has meant a duplication of effort to ensure isolation of transactions. One proposed solution is to implement a Napier88 system that works directly on top of the stable heap.

6 Distribution

6.1 Introduction

To maintain the illusion of an unbounded data space, persistent stores must eventually be distributed. However, there is a tension between the conceptual ideals of orthogonal persistence and the technological realities of distribution that make their integration difficult. This gives rise to a spectrum of possible solutions that balance the ease of user programming and conceptual modelling with the ease of implementation of the underlying system.

At one end of the spectrum, distribution is introduced to enhance the performance of the overall system and the ideal behaviour, as far as ease of programming is concerned, is where the distributed system may be programmed as if it were non-distributed. All other models of distribution expose some aspect of the underlying distribution to the user. To this extent distributed systems can be categorised by the manner in which they hide the underlying distribution mechanisms from the user. This concept is called *transparency* and has been shown to have a number of dimensions.

In [ANS89] the dimensions of transparency are given in relation to object-oriented systems. The situation is somewhat different for persistent systems; the dimensions of distribution transparency may be refined to the following:-

- *operation transparency* means that there is a uniform mechanism for invoking operations of both local and remote values, concealing any ensuing network related communications;

- *location transparency* means that the user cannot tell the location of a value in the network from its name;

- *migration transparency* means that an object may be moved from node to node by the system while maintaining its identity;

- *replication transparency* means that, wherever an object is replicated by the system for greater availability or efficiency, the intricacies of replica consistency maintenance are concealed;

- *recovery transparency* means that the semantics of any recovery mechanism is independent of the way the data it governs may be distributed.

In addition to the above and in order that the distributed stores be considered as part of one system there must be some underlying mechanism to allow them to work in unison. This is provided by the concurrency control mechanism which may be co-operative and controlled by synchronisation, competitive and controlled by atomic transactions or in between as in designer transactions. Thus the concurrency control ensures both synchronisation and isolation across the network. Implementation of this is non-trivial and may involve system wide semaphores and two phase commit protocols.

Where the distribution mechanism is completely transparent the user is presented with a single large persistent space, the *one-world* model. This approach fits in well with the concept of orthogonal persistence since all the physical properties of the data are hidden from the user including the placement of data, replication of data and the failure of nodes. The system is free to move, copy and replicate data to optimise its utility and is responsible for abstracting over any failure due to distribution. Applications need no modification to operate in different distributed environments.

While the one-world model is conceptually simple, there are a number of technological issues that make it difficult to deliver in scale. The management of very large stores involves problems which are well known to cause implementation difficulties. This chapter provides some background on these

problems and the effects of exposing levels of distribution transparency. The chapter discusses :-

- Mechanisms for providing transparency and the problems of totally transparent distribution.
- Alternate models of distribution that relax some of the transparencies.
- Particular models built for persistent systems.
- A distribution model that has been constructed in the Napier88 system as part of this thesis.
- an extension to the model that effects a two-phase commit of Napier88 atomic transactions over a number of nodes.
- an example application that uses the two-phase commit in a networked software distribution scheme.

6.2 Distribution Models

6.2.1 Transparency Provision

A number of mechanisms have been used as solutions or partial solutions to the problems of transparency provision :-

- *operation* transparency may be provided by an implementation of distributed shared virtual memory or alternatively through a mixture of procedure call and remote procedure call (RPC) [Nel81] depending of the locality of the called procedure. This is the basis of the Newcastle Connection [BMR82] and Amoeba [MRT+90] systems.
- *Location* transparency requires uniform naming and mapping tables from logical names to physical addresses. These mapping tables may also be used in the implementation of migration transparency.

- *Replication* transparency requires a more complex set of mapping tables and a coherency mechanism.

- *Recovery* transparency should ensure that the system can recover after a failure due to distribution.

The provision of a universal address space with total transparency on anything other than a small number of nodes is beset by technical problems. These are outlined by Dearle [DRV91] as involving :-

- the generation of unique addresses. In a flat shared address space objects are typically designated a unique address. When a collection of nodes are sharing this space then the address space must be large enough to ensure the unique identification of any object. Research into support for persistent stores over a large virtual address space [KR90, Coc89, Coc90] has focused on the development of specialist hardware which is not particularly widespread.

  Moss [Mos89] examines the cost of providing a large flat address space. He suggests that the provision of wide addresses could have an adverse price-performance effect on the CPU, cache and main store and backing store. Instead he favours a contextual naming scheme such as that designed in Mneme [MS88] where the address space is separated into distinct localities. Each locality has considerable autonomy in the management of free space, clustering strategies, object formats, recovery methods and concurrency control. Typically the locality of data is such that programs usually need only manipulate short addresses. The problem, however, with contextual addressing is that there is little or no hardware support and it requires considerable overhead in managing the localities. In addition, the availability of larger address space architectures is rapidly becoming more widespread and a number of Moss's objections to large address spaces have been

diminished by recent research in addressing and address translation techniques [SKW92, VD92].

- free space management. The structures used by a system to keep track of free space can be large. This overhead may be exacerbated by an increased address space. Coupled with this is the cost of allocation and freeing of secondary storage. A further consideration is distributed garbage collection. Garbage collection mechanisms that are time-independent of the size of the address space [Kol92, ELA88] can be complex. The complexity is increased when the address space is distributed over several nodes. A review of distributed collectors in [AMR92] highlights the problems and indicates that very few fault-tolerant collectors exist.

- distributed stability and recovery. The issues of distributed stability are similar to the problems of distributed garbage collection. In a one-world store pointers can “leak” across nodes and hence an interdependence of nodes is constructed so that they must be stabilised together. Recovery transparency may have to capture the entire state of the persistent stores at a checkpoint for co-operative concurrency models, and the partial but interrelated states for transaction models.

  Detecting such causal relationships in distributed systems has been the subject of much research [CL85, Jef85, SY85]. The algorithms for achieving distributed synchronisation and recovery are non-trivial and may result in cascade rollbacks. Rollback propagation can happen when a node X has become dependent on another node Y because they share a copy of the same modified object. If node Y were to crash and be restored to its last checkpoint state then, in order that the store on Y does not appear to travel backwards in time with respect to X, node X is also rolled back to it last consistent state. This may start a domino effect since the rollback of X may cause other nodes, including Y, to also rollback.

Algorithms for such stability often involve some form of two-phase commit [Gra78]. By using two phase commit protocols over a high bandwidth local area network, a modest number of reliable machines may be stabilised together. However, it is unlikely that such protocols would be successful when applied to large numbers of machines in geographically distributed locations.

6.2.2 Non-transparency

In light of these issues it is clear that persistent systems that wish to communicate over a wider-area distribution must consider alternative models. For scalability it may be prudent to relax the ideal of complete transparency by partitioning the persistent store into regions and making these regions visible at the language level. This relaxes location, migration and replication transparencies. The movement of data is now explicit and objects may be moved and replicated by the user. The potential advantages are for enhancing the performance of the system by user controlled parallelism and replication. This comes at some programming cost since, as the placement is no longer transparent, then programs may not work for all configurations of the system. In such models there is a need to define the semantics of failure so that the user can understand and react to it.

The relaxation of any of the transparencies is governed by a set of design decisions that gives the user more control and flexibility at the cost of complexity in the model. Where the system is not totally transparent there are a number of degrees of relaxation. A transparency may be available but in a restricted number of operations ($\checkmark^R$), it may be visible (**V**), visible but with a restricted number of operations ($\mathbf{V}^R$) or not available at all (**X**). An example of the last is where a distributed system does not allow migration of objects. Two visibilities are introduced in order to categorise non-transparent systems. They are :-

- *side effect visibility*. This means that operations may cause visible side effects in remote stores.

- *structure visibility*. This means that the structure of data and its internal sharing will not be preserved over operation invocation, migration and replication.

There are two broad models of distributed systems that allow different degrees of visibility. With the *federated* model, the persistent stores are known to be independent but obey some laws (transparencies) of the federation whereas the *confederated* model is a loose association of non-interfering stores only acting in unison by ad hoc agreement and disallowing side effects.

In a federated model some operations are made available to the user that explicitly refer to remote stores. Typically these operations deal with remote execution or remote data manipulation, such as RPC, and as such require the user to have knowledge of data location. This then makes location, migration and replication visible. The model has an overall protocol that governs data movement so that the referential integrity is maintained. The movement of data across store boundaries that preserves sharing can lead to pointer “leaking” where the closure of an object may reside on a number of stores. To preserve integrity then these stores become dependent on each other and hence must be synchronised together.

In the confederated model a restricted set of operations is provided, the stores do not make any attempt to act together and stabilising the stores is performed independently. For example the Stacos store, described below, is confederated and provides the user with a restricted store interface to interrogate a remote store and copy objects from it. Failure may occur between or during interrogations. The advantage of the confederated approach is that there are no remote pointers or inter-store dependency through side-effect. The stores can

then stabilise and perform garbage collection without regard to one another. The stores may however take part in a distributed commit by a two phase protocol and may synchronise within transactions by convention.

A major disadvantage of the confederated approach is the potential loss of referential integrity. This refers to a situation where two roots of a graph are independently moved or replicated in a store and these refer directly or indirectly to a common sub-graph. In a system which maintains referential integrity, only one copy of the common sub-graph is moved or replicated. Confederation by definition does not allow pointers to span stores therefore copies of data structures must be propagated between stores. Such copying may (and often does) violate referential integrity.

Figure 6.1 summarises the transparency/distribution model matrix using the notation described above.

Transparency	**One-world**	**Federated**	**Confederated**
Operation	✓	✓^{R}	✓^{R}
Location	✓	**V**	**V**^{R}
Migration	✓	**V**	**X**
Replication	✓	**V**	**X**
Recovery	✓	✓	**V**^{R}
Visibility			
Side-effect	**N/A**	**V**	**X**
Structure	**N/A**	✓	**V**

Figure 6.1: Transparency/Model matrix

6.2.3 One-world models

One approach to complete distribution transparency has been through the provision of distributed shared virtual memory [Li86, LH89, WF90]. The architecture typically comprises of a number of loosely connected computers each with their own local memory, no shared physical memory and a protocol that presents a uniform shared virtual address space to all the nodes. Each node maps local memory into the shared virtual address space with pages moving not only between main store and disk but also between the physical memories of the nodes. Several nodes may contain copies of the same page and hence the mechanism employs a coherency protocol to preserve data integrity.

The versatility of the coherency mechanism is usually the distinguishing feature between implementations. Wu and Fuchs [WF90] present a scheme that allows the virtual memory to be recoverable without cascade rollbacks. Their approach involves automatic node checkpointing and recovery that limits the rollback propagation. The solution proposed by Wu and Fuchs limits the rollback propagation by insisting that a process always checkpoints before sending a modified page to another node.

Two different distributed persistent systems, the Casper model [KSD+91] and the Monads system [HR91] have been designed and built using the distributed shared memory approach and are discussed below.

6.2.3.1 Casper

Casper (Cached Architecture Supporting Persistence) is of particular interest in this thesis as it is a distributed architecture designed to support Napier88 programs. The architecture, as shown in figure 6.2, is based on a client-server model where a client is a Napier88 thread operating on a page cache of objects from a single stable virtual memory. The central heap manager in the server allocates unused pages to clients requesting free space. The backing store for the

shared memory is provided by the server which employs a shadow paging mechanism to ensure that a consistent state is always recoverable.

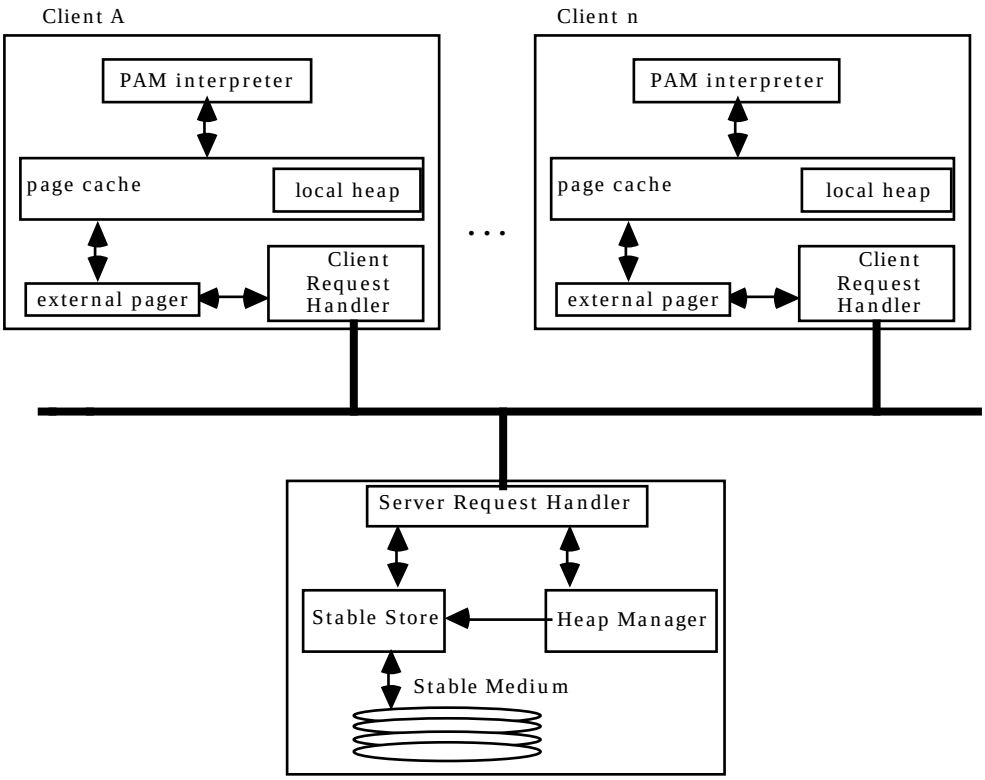


Figure 6.2: The Casper architecture

A coherency protocol is used that guarantees data integrity. The protocol allows multiple clients to read the most up-to-date copy of a page using a single writer with multiple readers mechanism.

- All read/write requests are channelled through the server. If the server does not hold a copy of the most recently modified state of a page the request is forwarded to the client that does.

- Clients become dependent on each other because they have seen the same modified page with respect to the server. The set of mutually dependent clients is called an *association*. Any client initiating a meld requires all other clients in its association to meld also. This allows several independent melds to be in progress at the same time.

- The server maintains the list of associations. If a client fails then all members of the client's association must return to their previous consistent state.

The Casper system has been implemented using the multi-threading and external pager features of the Mach operating system.

6.2.3.2 Monads

The Monads project has produced a new computer architecture that was designed to support :-

- a large single-level persistent store through a stable, paged virtual address space based on shadow paging.

- a uniform and secure protection scheme based on capabilities.

- separate address spaces within the virtual address space that can be used as independent information-hiding modules.

The resulting research led to the construction of a new microprocessor, the Monads-PC, which implemented these ideas. Further work on the project [HR91] resulted in the extension of the virtual memory address space to a distributed shared virtual memory across a network of Monads-PCs. Unlike the Casper model there is no central server and each node provides it own backing store for a portion of the address space. This complicates the coherency protocol in that the mechanisms such as stability and free-space management are then necessarily de-centralised.

The virtual addresses of the distributed shared virtual memory are unique network-wide and are never re-used. Figure 6.3 illustrates the partitioning of a Monads virtual address.

Figure 6.3: A Monads virtual memory address

- The node number uniquely identifies a host. All addresses therefore have an explicit owner node.
- The volume number corresponds to a disk or logical disk partition on the host.
- The address space number corresponds to an in-volume address space that typically contains related sets of data such as a program, an information-hiding module, or process stack. Address spaces are divided into segments and accessed by segment capabilities.

Each node maintains an exported pages table (XPT) and an imported pages table (IPT) which are used in the single writer/multiple readers coherency scheme. When a page-fault occurs the node number of the address is checked to see if the page is local or remote. A page-fault for a local page must first check the exported page table to ensure that no other node has a writeable version of the page. If there is such a node, it is requested to return the page and mark the page as read-only in its imported page table. The page-fault is then resolved from the network rather than the local disk. On a page fault of a remote page, a copy of the page is requested from the owner node. Pages are always exported read-only. The remote node must explicitly request promotion to write status. Unlike the Casper model if there is a single writer then no other writers or readers have a copy of that page. In other words nodes do not form associations. However, if a node fails and is restarted from the point of its last consistent state then any node that has write access to a page from the crashed node must also rollback to its previous state.

6.2.4 Federated Models

6.2.4.1 DPS-algol

DPS-algol [Wai88] is a persistent system derived from PS-algol that supports a model of distribution through the provision of a universal address space that spans several nodes. Other extensions to the language include a facility for expressing and manipulating separate concurrent activities and a remote procedure call mechanism that enables inter-process communication and synchronisation.

Through the provision of a universal address space DPS-algol presents a one-world model where all the transparency dimensions are supported. However an assertion is made that it may not always be desirable for efficiency reasons to completely abstract over locality. DPS-algol provides a facility that allows the explicit naming of remote nodes and the discovery of object locality. Remote nodes are named as values of a new type *locality* and are spaces where processes may be executed. Two language constructs *transcopy* and *assign* enable the atomic data movement between localities. Figure 6.4 illustrates the ideas.

```
let presto = locality selectCuts
! presto is a handle on the locality (node) where the object selectCuts resides

let prestoBrekkies := transcopy brekkies to presto
! This transfers a copy of the object brekkies to the presto locality

assign newImprovedBrekkies to prestoBrekkies
! The value of the variable prestoBrekkies is re-assigned to the value
! of the object newImprovedBrekkies
! The assign will effectively transcopy the object to prestoBrekkies locality
```

Figure 6.4: Using localities in DPS-algol

The semantics of the copying are dependent on the type of the object being copied and effectively involves a top-level copy of the object's closure rather

than a deep copy. The idea is to reduce the quantity of data transferred between localities and avoid the inadvertent copying of the complete store without the programmer's knowledge. As a consequence of this method an object graph can be spread across several nodes.

With this notion of locality a process can be executed on a remote store or a remote procedure called via the RPC mechanism. The example in figure 6.5 shows how a process can be executed on a remote machine. The second expression of the **start** construct, *shopping*, determines the name that the process is registered with on the remote store.

```
let shopping = "shoppingProcess"
let goShopping = process
begin
…
end
let processHandle = start goShopping as shopping at presto
```

Figure 6.5: Starting a process on a different locality

The illusion of a universal address space is provided through a uniform treatment of object reference. This is supported by the DPS-algol abstract machine which distinguishes pointers to local and remote data. References to remote objects are through heap objects called *remote pointers* which use contextual addressing to uniquely define the object. An instance of the abstract machine running over a local store generates a remote pointer when there is an external reference to an object in that store. Each process maintains an export table of remote pointers it has exported. The coherency of the distributed address space is thus dependent on the consistency between the export table on a store and the use of remote pointers by a process.

The essence of DPS-algol is that it provides, by default, a one-world model where a user can manipulate data and execute threads without knowledge of

locality. The programmer can make use of constructs that expose locality allowing the explicit movement of data or placement of process execution.

6.2.4.2 Argus

The Argus [Lis84] system is an archetypal example of a federated system that allows a programmer to construct programs into a collection of modules that are executed on different nodes. A *guardian* is the Argus abstraction for an stable store and encapsulates data and a set of processes that operate on the data. A data object in Argus wholly resides within one guardian and the sharing of objects between guardians is not permitted. Instead guardians communicate through *handlers* which are defined in the body of a guardian definition.

An Argus program is structured as an atomic action that is recoverable, serializable and total. As a program execution progresses it accesses and modifies data at several other guardians through handler calls. The system employs a two-phase commit protocol that ensures that when an action completes it either commits all the changes made at all the visited guardians or aborts at every guardian. The commits are to stable storage and are hence recoverable.

Argus uses a remote procedure call mechanism to pass data between guardians. Pointer leaking between guardians is avoided by (deep) copying objects between the sender and receiver. These copies are considered to be separate and hence referential integrity is lost. The Argus model thus exhibits structure visibility.

Figure 6.6 summarises the models described. The DPS-algol can function as a one-world model but has the transcopy and assign operations that control the visibility.

Transparency	Casper/Monads	DPS-algol	Argus
Operation	✓	✓R	✓R
Location	✓	✓R	**V**
Migration	✓	✓R	**X**
Replication	✓	✓R	**X**
Recovery	✓	✓	✓
Visibility			
Side-effect	**N/A**	**V**	**V**
Structure	**N/A**	✓	**V**

Figure 6.6: Classification of models

6.3 Stacos

6.3.1 Introduction

Stacos (St Andrews Confederated Object Store) is a confederated model that extends the Napier88 name space to include other global address spaces. The model is confederated in that the distribution is constrained to services that do not side-effect these other spaces. Two stores then can communicate with each other but at no time become inter-dependent and hence can meld and garbage collect independently. Figure 6.7 shows where the Stacos model lies within the transparency classifications.

Transparency	Stacos
Operation	✓R
Location	**V**R
Migration	**X**
Replication	**X**
Recovery	**V**R
Visibility	
Side-effect	**X**
Structure	**V**

Figure 6.7: Stacos classification

Whilst this is a fairly restrictive form of distribution the architecture design to support confederate distribution can be used as a framework for building other models. The features of Stacos presented in this section are :-

- a base model that incorporates a simple extension to the Napier88 environment that allows a program to browse remote object stores and specify and deep copy objects from these stores in a type-safe manner.

- an implementation level communication based on a protocol layered on TCP/IP sockets. This abstraction was chosen instead of a higher-level mechanism, such as Sun's RPC implementation, principally for the widespread availability on a range of platforms and world-wide connectivity. This enables any Napier88 persistent object store residing on a machine on the internet to be accessible.

- an extension to the model that enables user-written services in addition to the scan/copy functionality. These services can define there own store-to-store protocol with the restriction that they do not side-effect the remote store.

- the provision of support in the model for two-phase commit of Napier88 atomic transactions across stores.

6.3.2 Base Model

In the base model remote Napier88 stores can be named and communication established that allows the remote stores to be browsed and have objects copied from them. A Napier88 program still has its own stable store with one distinguished persistent root but has the ability to import objects from other stores. Once an object has been copied it then behaves and can be manipulated like any other Napier object.

Figure 6.8 illustrates the architecture with the distribution at the Napier88 level. The user converses with another store using a package of communications procedures that provides clean failure semantics so that the user can understand and react to failure.

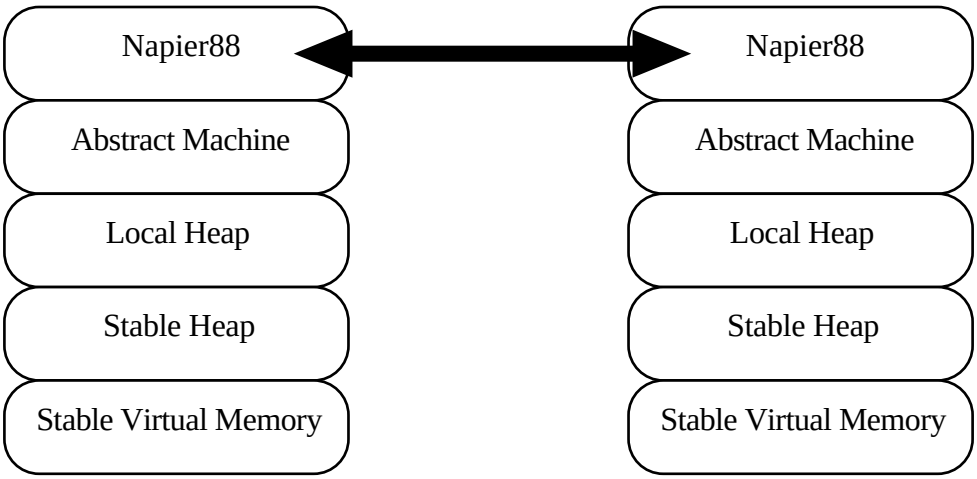


Figure 6.8: Stacos architecture

The model is inherently confederated since browsing and copying do not cause side-effects on the remote store. Objects can only be copied in one direction - i.e., 'pulled' from a remote store and because the objects are copied by transitive closure there is no possibility of “pointer leaking” across stores. Note that this is in stark contrast to the *transcopy* construct of DPS-algol. However there is a

consequential loss of referential integrity since the object identities are forfeited and hence a loss of replication transparency. If two objects on the remote store that point to the same object are copied to the local store then they will no longer share the object in common. Each will have a copy of the object.

Remote objects are specified by pathname through environments. For each connection to a remote store an instance of a browser procedure and a copy procedure is supplied that allows remote environments to be scanned and objects copied. Each scan returns a list of name-type tuples for the specified pathname. Using this information the calling program can traverse the remote store's hierarchy. The copy function names an object in a specified environment that is to be deep copied to the local store.

6.3.2.1 Language interface

An addressing convention in this model uniquely identifies a remote store by a combination of the host's internet domain name or number and the full pathname of the stable store file. The user references a remote store by assigning a symbolic name to the remote store's address. A database is provided that allows the user to add, list and delete these names. The interface functions to this database are given in figure 6.9.

```
type RemoteStore is structure( name,host,storeDir : string )

addEntry : proc( string, string, string → string )
listEntries : proc( → List[ RemoteStore ] )
removeEntry : proc( string → string )
```

Figure 6.9: Remote store database interface functions

- addEntry : **proc**(**string**, **string**, **string** → **string**)

This procedure adds a new entry to the remote stores database. The parameters are the entry name, the name of the remote host and the pathname of the store at the remote host. The host name may be specified as a local machine name, a full Internet host name or an Internet number. If an entry with the given name already exists the addition fails and an error message is returned, otherwise the null string is returned.

- listEntries : **proc**(→ List[RemoteStore])

  This procedure returns a list of the entries in the remote stores database. Each element of the list is a structure containing an entry name, a host name and a store pathname.

- removeEntry : **proc**(**string** → **string**)

  This procedure takes the name of an entry in the remote stores database and removes the entry from the database. If an entry with the given name does not exist an error message is returned, otherwise the null string is returned.

The symbolic name for a remote store is used in the *openRemoteStore* procedure call to establish a new remote connection. Figure 6.10 shows the definition of the procedure and its associated types.

```
type EnvEntry is structure( entryName : string ; entryType : TypeRep )

type ScanResult is variant( successful : List[ EnvEntry ] ; error : string )

type CopyResult is variant( successful : any ; error : string )

type RemotePack is structure(
        scan :   proc( string → ScanResult ) ;
        remoteCopy :  proc( string,string → CopyResult ) ;
        closeRemotePack :     proc() )

type ConnectionResult is variant( successful : RemotePack ; error : string )

openRemoteStore : proc( string → ConnectionResult )
```

Figure 6.10: Remote store communication language interface

- openRemoteStore : **proc**(**string** → ConnectionResult)

  This procedure takes the name of an entry in the remote stores database and attempts to open the remote store for scanning and copying. If the open operation is unsuccessful the result is a string giving an explanatory error message, otherwise it is a structure containing the following operations on the remote store:-

- scan : **proc**(**string** → ScanResult)

  This procedure takes a string pathname describing an environment in the remote store and attempts to scan it. A pathname is relative to the persistent root and should consist of an initial slash followed by environment names separated by slashes, for example: "/Library/Distribution".

  If the pathname is not well formed or if a communication error has occurred during the scan the result is a string describing the error. Otherwise the result is a list of structures containing one element for each of the bindings present in the remote environment at the time of the scan. Each element contains the name of the binding as a string and a representation of the type of the binding.

- remoteCopy : **proc**(**string**, **string** → CopyResult)

  This procedure takes the name of a binding in a remote environment and a string pathname describing the location of the remote environment, and attempts to make a deep copy of the binding. The pathname is described in the same way as for *scan* above. If the pathname is not well formed or no binding with the given name is present, or a communications error occurred the result is a string describing the error. Otherwise the result is a copy of the remote binding injected into *any*.

- closeRemotePack : **proc**()

  This procedure closes the connection to the remote store. Further calls to the procedure have no effect. Subsequent calls to *scan* and *remoteCopy* result in messages stating that the connection is closed.

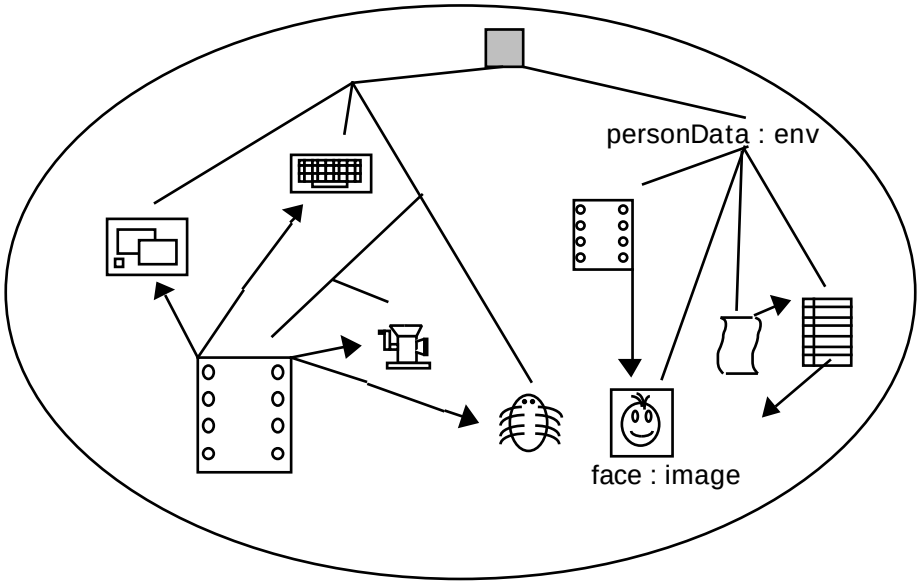


Figure 6.11: Example remote store

For example consider the remote store illustration in figure 6.11 and suppose that a Napier88 program running on a local store wants a copy of the object *face* in the remote store. The code to initiate the connection and scan the path to the environment containing *face* and copy the object is shown in figure 6.12.

```
let connect = openRemoteStore( myNameforRemoteStore )
if connect isnt successful then writeString( connect'error ) else
begin
        let scan = connect'RemotePack( scan )
        let remoteCopy = connect'RemotePack( remoteCopy )
        let closeConnection = connect'RemotePack( closeRemotePack )
        ! Now scan the root environment
        let currentEnv := "/"
        let scanResult := scan( currentEnv )
        if scanResult isnt successful then writeString( scanResult'error ) else
        begin
                ! Look through the return list of name-type tuples for personData
                …
                currentEnv := currentEnv ++ "personData"
                scanResult := scan( currentEnv )
                if scanResult isnt successful then … else
                begin
                ! Look through the return list of name-type tuples for face
                …
                let copyResult = remoteCopy( currentEnv, "face" )
                if copyResult isnt successful then …else
                project copyResult'successful as X onto
                image : writeString( "Successful copy" )
                default : writeString( "Oops" )
        end
end
```

Figure 6.12: Program to scan and copy remote object

To simplify accessing remote stores a graphical interface was developed and incorporated into the Napier88 programming environment. This is described in Appendix C.

6.3.3 Implementation

The implementation strategy for supporting Stacos involved the following :-

- incorporation of an interface to the Unix socket abstraction in Napier88.
- definition of communication establishment protocol and store-to-store copy/scan protocol.
- a mechanism for transferring the transitive closure of an object across a virtual circuit.

The socket-based connect/accept primitives were incorporated into the Napier88 primitiveIO environment. The procedures in this environment provide a mapping between the I/O facilities of Unix and Napier88. The socket-to-socket communication is seen at the language level as file or device I/O. Once a connection is established the user can then perform I/O in the same fashion as reading and writing disk files.

6.3.3.1 Connection establishment

Connection establishment between two stores using sockets is complicated by the internet address binding mechanisms. Communicating processes between sockets on the internet are bound by an *association*. An association is composed of local and foreign addresses (hosts) and local and foreign ports. Associations are always unique and no duplicate <protocol, local port, local address, foreign port, foreign address> tuples exist network-wide. On a connection set-up the sender is allocated the local port and local address dynamically by the operating system and uses the internet number of the receiving host to form the foreign address The problem for the sender is knowing what the foreign port number is. Typically a TCP/IP Unix application such as telnet uses a fixed port number that is agreed by convention. The internet daemon process on Unix listens on that port on behalf of the application for incoming requests. For each connect received the daemon accepts the call on behalf of the application and forks a process that executes the application binary. The application binary is started with its standard input and standard output descriptors bound to the socket.

There are a number of reasons why this approach does not blend well with the Napier88 to Napier88 communications model :-

- The application binary in the Napier88 case is an instance of the abstract machine interpreter running Napier88 programs against a store. This implies that an instance of the interpreter would be executed for each incoming

connect request and also that each interpreter would be limited to one connection. This then places a heavy restriction on the functionality of the Stacos model.

- Having an instance of the interpreter that is bound to the standard input and standard output will again restrict the functionality of the model.

- Having a number of Napier88 interpreters, spawned by a connection, running against a single store does not fit the concurrency models outlined in chapter 4.

The implemented solution was to create a front-end program, outwith the Napier88 system, that acted as an interface between the connection establishment procedures of Unix and the Napier88 requirements. The front-end program is called by the internet daemon every time a connect request for access to a Napier88 store arrives at the host. The solution effectively defines a protocol between any running Napier88 interpreters on a host and an incoming connect and a protocol between the program and the abstract machine of the caller. The sequence of events is portrayed in figure 6.13. When a Napier88 session begins a Napier88 thread is executed that allocates a free port number from the operating system using a primitive I/O socket function. The thread then registers its port number and the pathname of the store that this interpreter is running against with the front-end program. The front-end program maintains a table of <store pathname, port number> tuples for each running Napier88 system on that host. When the client program issues an *openRemoteStore* the abstract machine issues a connect request on a known port number to the server host. The internet daemon sees the connect and accepts the call on behalf of the front-end program and forks a process which then executes the front-end program.

At this point the interpreter of the client and the front-end program enter into a dialogue. The client will have received the accept of the call by the server's

internet daemon and so sends back a string containing the pathname of the remote store that it wishes to access. The front-end program matches that string against its table entries and if one is found the port number of the entry is returned to the client. If no match is found, indicating that the requested store either does not exist or is not currently available for access, the front-end program closes the connection.

The client then closes the first connection and makes a new connection call using the returned port number. At this point the thread of the Napier88 program running on the server is resumed. The thread first removes its entry from the listening table, accepts the call and starts a new listening thread. The Napier88-to-Napier88 communication has now been established.

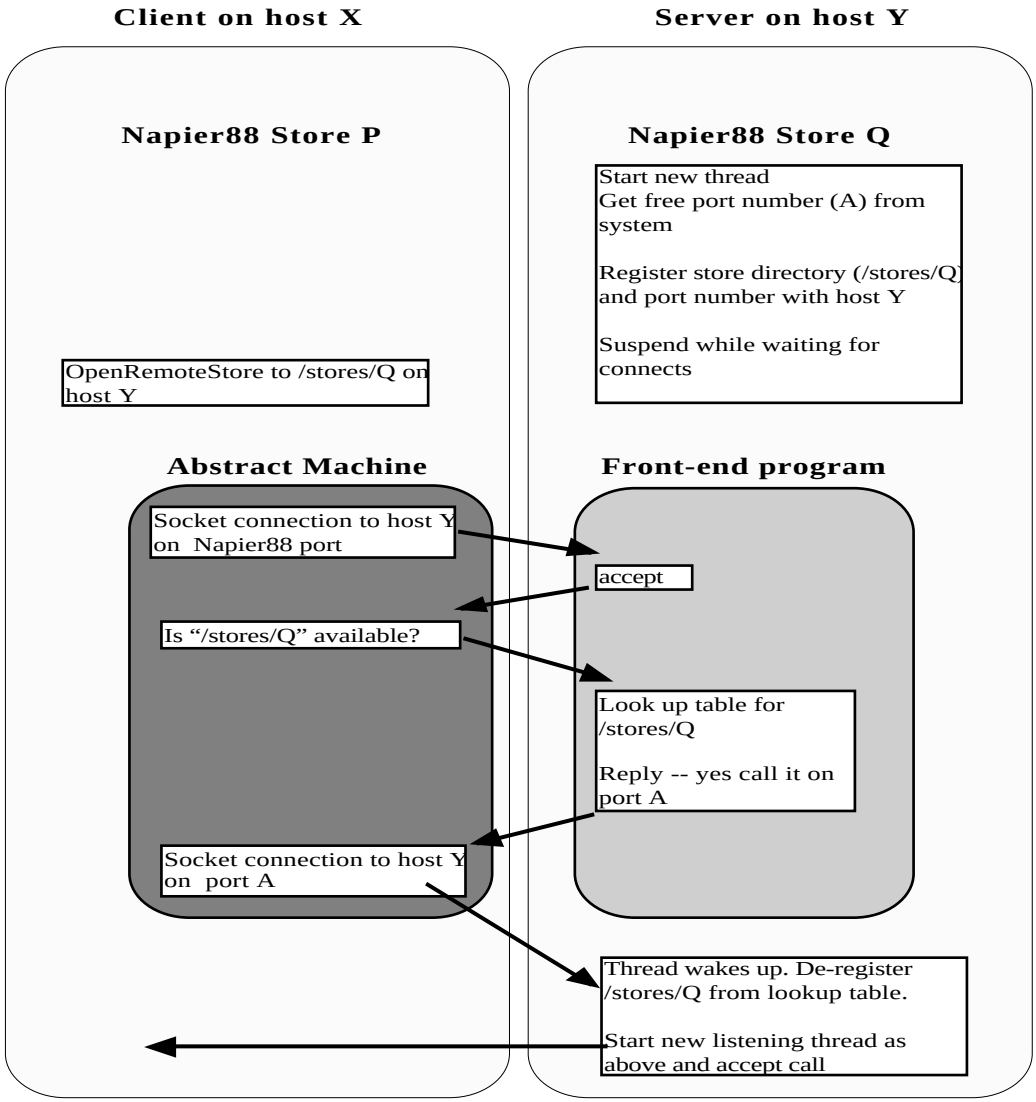


Figure 6.13: Connection establishment

6.3.3.2 Remote object copying

Once a connection is established then communication between the stores is controlled through the Napier88 programs at either end of the connection following a simple protocol. The client sends scan/copy requests to the server by constructing a string with the desired command (scan or copy) and the parameters as quoted strings. This string is then written to the socket. The client then waits for a reply.

The server is a thread which runs a loop looking for scan/copy commands from the client. On a scan request the parameters are read from the network and a list

of the bindings in the specified environment are constructed using the scan procedure from the standard Napier88 user environment. The list is then transferred across the connection using a mechanism described below. A copy request similarly checks to see if the requested object does in fact reside in the specified environment. The object is then injected into an **any** and copied across the network.

The server thread is suspended by the scheduler if it is blocked waiting for a command from the client. The scheduler keeps track of all threads that are blocked waiting on an external event and periodically checks for data pending on any open file descriptors.

It should be noted that the server copy and scan functions are written in Napier88 and compiled using the bootstrap compiler. As such they have access to primitive functions not generally available to the user. For example one of primitive functions allows a value of type **env** to be decomposed into its internal structure which includes its name as a string. This function is used by the scan/copy functions to match the string pathname parameters passed by the user with environments in the store.

Two new primitive functions were constructed to enable the transfer of Napier88 objects across a socket. One to flatten an object and write it to the socket and another to read from the socket and reconstruct the object. Two functions, namely *importAny* and *exportAny*, already perform such a function and are used for code planting to the file system and reconstituting code from a Napier88 object file. However the implementation of exportAny performs a recursive traversal of an object ‘s closure writing out each object’s header as it traverses and then back patching the pointer fields with file offsets using file seeks. These procedures keeps a list of objects already checked to handle circular references. Seek operations cannot be performed on a socket stream and so two similar functions, *exportAnyStream* and *importAnyStream*, were written to perform the

same job but without seeks. The *exportAnyStream* procedure recursively traverses an object's closure and assigns each object with a unique object number which is written out along with the object header. As an object is traversed its pointer fields are written to the socket with either the object number assigned to the object pointed at if it has been written out already or a new object number. The *importAnyStream* function reads the data from the input stream socket and builds the objects as they are read. From the object numbers in the pointer fields a table of pending objects, that is objects that are still to be read in, is constructed. For each entry in the pending table a list of objects is kept that "point" to the pending object. When a pending object is read from the socket this construct is used to resolve the pointers and the object's entry is removed from the table.

6.3.4 Transaction Processing and Two-phase Commit

Whilst the scan/copy service is limited in its functionality it can provide a valuable service as a passive data server to a number of Napier88 clients. The functionality of the model can be enhanced by developing a more complicated underlying protocol. The communications framework of the virtual circuit, the object copying facility and the connection establishment procedure can be used as a base for developing different services. The connection establishment protocol discussed in section 6.3.3.1 described how a server thread displays its willingness to accept calls by registering the port number and store pathname with the front-end program. By adding an extra parameter to the handshaking protocol a number of threads can be started by the server, each of which listen on a different port number and perform a different service. For example, in addition to the scan/copy facility, a listener thread may be written that provides a network "talk" service between two users using different stores.

Any service can be developed provided it does not rely on the availability of a connection or side effect the data on the server. These services can be thought of

as the equivalent of the guardians of Argus. By developing services that communicate with other services a controlled two-way communication can be created. For example node X may call a service of node Y that asks Y to copy a procedure from X and execute it. This then could form the basis of a RPC mechanism or a remote evaluation model [SG90]. In addition a model could be constructed where a node transfers source to another node where it is then compiled and executed. In particular the source may be the code for a new service and by using the callable compiler of Napier88 one node could be asked to transfer the code.

Building more complex models from these basic services inevitably leads to a position where a store may want to synchronise changes with a number of remote stores. By encompassing the facility in the transaction package described in chapter 4 a two-phase transactional commit can be constructed to provide distributed synchronisation.

In a two-phase commit protocol one of the transactions acts as a co-ordinator and the other transactions involved act as participants. Both the co-ordinator and the participants have a preparing phase and a committing phase. Since the participants are running as Napier88 transactions then no permanent change will have been made to the store until a meld is executed. It is the synchronisation of each participant's meld that is the subject of this model's two-phase commit. Figure 6.14 shows the amended language interface with a prepare and result procedure added to the *RemotePack*. Note also that the *RemotePack* has been altered to cater for a number of different services.

```
type ScanCopyPack is structure(
        scan :   proc( string → ScanResult ) ;
        remoteCopy :  proc( string,string → CopyResult ) )

type OtherPack is structure(…)

type ServicePack is variant( scanCopy : ScanCopyPack ;
                             otherServices: OtherPack …

type RemotePack is structure(
        servicePack : ServicePack ;
        prepare : proc( → bool ) ;
        ! true reply means "prepared" ; false is "abort"
        result : proc( bool → bool ) ;
        ! send true means that all the participants are prepared
        ! and should now commit
        ! Reply boolean should always be true except to indicate network errors
        closeRemotePack :     proc() )

type ConnectionResult is variant( successful : RemotePack ; error : string )

openRemoteStore : proc( string,string → ConnectionResult )
```

Figure 6.14: Amended Remote store communication language interface

- prepare : **proc**(→ **bool**)

  The co-ordinator calls this procedure for each remote connection to signal to the participants that they should prepare to commit. If a participant replies that it has successfully prepared then a **true** value is returned. All other responses result in a **false** being returned.

- result : **proc**(**bool** → **bool**)

  This is issued by the co-ordinator to each open connection to inform them of the outcome of the "vote" on the prepare call. If all the participants involved acknowledged that they had prepared the result procedure is parameterised by **true** telling them to commit now. Otherwise an abort is signalled using **false** as the parameter value.

  If the participants were instructed to commit then when this has been done the *result* procedure returns **true**. This should always be the return value of such

a call since the protocol demands that a co-ordinator, once it has sent a commit request to each participant, must continue sending messages to the participant until it gets a reply.

To support two-phase commit requires a change to the concurrent store to accommodate a “prepared” transaction state that is recoverable. This is described in the sequence of actions of the two-phase commit below :-

- **Co-ordinator Actions**

  Prepare phase. Each participant is sent a prepare-to-commit message. If each participant replies “prepared” the co-ordinator enters the completion phase. If any participant replies “abort” the co-ordinator sends abort to each participant.

  Completion phase. The co-ordinator transaction executes a meld. The meld signifies that the transaction has committed and that that state cannot be revoked even in the face of network or participant failures. The program state of transaction forms part of the meld so in the event of a subsequent crash of the co-ordinator node the transaction would automatically be restored and continue from that point. As a consequence it is essential that the meld includes sufficient information to allow communications to be re-established on system restart. This is achieved by retaining the remote hosts and remote store pathname information as part of the open files vector which is in the PAM root object written back on a meld. This can be interrogated on start-up and a socket connection for each store re-connected.

  Committing messages are now sent to each participant and the transaction waits for committed replies from each participant. Once it has received these it again melds and the transaction is complete.

- **Participant Actions**

  Prepare phase. This requires that a transaction saves its state in the persistent store so that it is recoverable but without committing changes. To accommodate this a “prepare” function has been added to the transaction interface. This function writes back all pages modified by the transaction to their shadow pages. The stable disk page table is not updated at this point as would happen in a commit. Instead the current root page records that the transaction is in a prepared state by setting a flag in the secondary transaction disk-page table for that transaction. The root page is then written back atomically and a prepared message returned to the co-ordinator. The participant enters the completion phase.

  Should the system crash at this point then on restart any prepared but incomplete transactions can be found from an inspection of the root page. The running state of the transaction will not have been saved with the crash since it did not form part of the last meld and so the system spawns special purpose-built transactions that listen for a committing or abort messages from the co-ordinator.

  Completion phase. The participant waits for either a commit or abort message from the co-ordinator. On a commit the transaction melds, sends a completed message to the co-ordinator and completes. On an abort the transaction aborts and releases its shadowed pages.

6.3.5 Software Distribution with Two-phase Commit

As an example of the kind of functionality a confederated model can provide, a software distribution scheme has been built into the Stacos model. The model involves a Napier88 program informing a number of remote stores that it has a new version of a piece of software and where it is stored. The remote stores can

then copy the software to their own stores. The distribution scheme is constructed as a set of transactions that communicate across nodes and then synchronise using the two-phase commit.

In this software distribution scheme two listening transactions are spawned when a Napier88 system is started. One is for the copy/scan communication as before except that the other transaction listens on a different port for any news of new software from any other Napier88 system. This transaction registers with the front-end program giving a port number, store pathname and a service name of “software distribution”. As part of the TCP/IP socket abstraction a receiver of a connect can always ascertain from the incoming information the remote address of the caller. The handshaking protocol and the Napier88 accept primitives have been amended so that when a listener thread receives a connection it is provided with the internet address and store pathname of the caller.

When a machine wishes to connect to a remote store it must now also specify the service it wants to talk to. The changes to the connection interface are given in figure 6.15. The *RemotePack* returned by a connect uses the variant *ServicePack* which contains the pack of procedures specific to the particular service connection. As new services are added to this system their interface functions can be added to the *ServicePack.*

```
type EnvEntry is structure( entryName : string ; entryType : TypeRep )
type ScanResult is variant( successful : List[ EnvEntry ] ; error : string )
type CopyResult is variant( successful : any ; error : string )
type SoftDistResult is variant( successful : null ; error : string )

type ScanCopyPack is structure(
        scan :   proc( string → ScanResult ) ;
        remoteCopy :  proc( string,string → CopyResult ) )

type SoftDistPack is structure(
        newObjects :   proc( *string → SoftDistResult ) )

type ServicePack is variant( scanCopy : ScanCopyPack ;
                             softDist : SoftDistPack )

type RemotePack is structure(
        servicePack : ServicePack ;
        prepare : proc( → bool ) ;
        result : proc( bool → bool ) ;
        closeRemotePack :     proc() )

type ConnectionResult is variant( successful : RemotePack ; error : string )

openRemoteStore : proc( string,string → ConnectionResult )
```

Figure 6.15: New Remote store communication language interface

The software distribution protocol begins with a Napier88 system sending a remote Napier88 system a connect request for the software distribution service. If this is accepted then the caller receives an instance of the *SoftDistPack* structure with the *newObjects* procedure. The caller then uses this procedure to tell the remote store where the new objects are. This is specified as a vector of strings where each string is a pathname for the objects the caller wishes to be copied. The caller then waits for a reply. A successful reply indicates that the remote store has successfully copied objects. Otherwise a string with an explanatory message is returned.

At the remote end a transaction listens for software distribution service connection requests. When such a request arrives the transaction reads the vector of pathnames from the socket after accepting the call and then calls the caller's copy/scan service. Using the pathnames it was provided with the remote store

pulls the objects over to its store and then closes the connection. The caller's *newObjects* procedure returns successfully.

Since the participants are running as transactions then no permanent change will have been made to the store until a meld is executed.

The co-ordinator in this model is the transaction that initiates the distribution. Once each of the participants has replied that the software updates have been successfully transferred then the two-phase commit can begin. The co-ordinator will issue a "prepare" to each participant and wait for replies. If all the replies are successful then the co-ordinator melds and requests each participant to do likewise. The co-ordinator then waits for the acknowledgement from the participants. On receiving a prepare message a participant transaction enters its prepare phase which effectively terminates the transaction execution. The Napier88 system writes all pages modified by the transaction to its shadow blocks and records the transaction prepared state in the root page. The "prepared" message is then sent to the co-ordinator and the system will either effect a meld or an abort dependent on the outcome of the commit.

6.4 Conclusions

The one-world model where the distribution is completely transparent to the user is undoubtedly the ideal model for orthogonally persistent systems. This approach preserves the persistence abstraction of hiding all the physical properties of data from the user. A number of distributed persistent systems have been designed and built using such a model. These systems effect a one-world model through the provision of a uniform address space across a number of nodes. Such schemes require a coherency protocol to guarantee data integrity.

The problems of maintaining the universal address space illusion were discussed. One particular problem that affects distributed systems is the detection of causal relationships between nodes that derives from the absence of a global clock. In a

distributed persistent system this can be seen in recovery and garbage collection techniques across nodes. Fault-tolerant distributed collectors and network-wide recovery procedures can suffer from a cascade of rollback propagation. The technical solutions to providing complete distribution transparency suffer from a lack of scalability that constrains such systems to only a handful of nodes.

Distribution models that relax any of the transparencies can be categorised as being either federated or confederated. In these models the user is made aware of other address spaces and is provided with a mechanism for communicating with them. A federated model is one where each store operates on a logically separate address space but the stores conform to a global convention. In a confederated model there is no convention and interaction between stores is limited to functions that do not side-effect across store boundaries.

In view of the scale limitation of the one-world model and the plethora of non-transparently distributed alternatives, a persistent system that has the flexibility and infrastructure to support a range of models may be the optimum strategy.

The Stacos store is a typical example of a confederated model. The model allows a remote store to be scanned and objects copied from it. It was shown how this model could be enhanced to provide a synchronised update service to a number of participants. This then can be viewed as providing a base on which models of distribution can be constructed and integrated into Napier88. There are a number of issues of the Stacos store which warrant further investigation. For example techniques that can reduce the amount of data transferred in a deep copy could be improved. It is hoped that the advent of hyperworlds [Kir92] will help reduce the size of an object's transitive closure. The export/import functions used to transfer the data across the virtual circuit may benefit from some compression/de-compression filters.

7 Conclusions

The motivation for the work presented in this thesis is the integration of the concepts of concurrency and distribution with the persistent abstraction. The principal contribution presented is the development of a flexible persistent architecture for Napier88 in which any model of concurrency and distribution can be constructed and supported. The key areas in this study are :-

- Integration of concurrency with persistence.
- Integration of distribution with persistence.
- Design and implementation strategy for the construction of the supporting architecture.

7.1 Integrating Concurrency

The concept of concurrency has been viewed here as a spectrum of understandability. Concurrency models lie on points on the spectrum and are distinguished by the extent to which the programmer or the system is responsible for the maintenance of global cohesion. By capturing this spectrum into the Napier88 persistent architecture a framework is provided in which all models of concurrency can be built and supported.

A conceptual layered architecture was developed that separated the intrinsics of concurrency into separate address spaces. The concurrency control address space specifies and controls the interaction of the separate actions and directs the movement of data between the address spaces. Each action has a private address space in the action address space layer and may also belong to a group address space shared with other actions without the data being globally visible. The concurrency control can commit actions through a meld operation by transferring the local or group address spaces to the persistent address space. Data in the

persistent address space is visible to all actions. The movement of all data is atomic.

Concurrency in this architecture is defined in terms of the sharing of address spaces and the movement of data between them rather than by a description of the behaviour and interaction of processes. The motivation for this approach is derived from the CACS specification system. The address spaces reflect the CACS visibility structures with the persistent address space corresponding to CACS database and the action address spaces equated with the access sets.

The conceptual architecture led to the design of a concurrent shadow-paged store which conforms to the Napier88 layered architecture design and supports the CACS view of data visibility. In this store the stable virtual memory acts as the CACS shared database with a form of shadow paging being used to support the access sets.

The implementation of this concurrent store provides a basis for supporting CACS in the Napier88 system. The incorporation of CACS into Napier88 is completed with the provision of communication paths between the two systems at strategic points in the architecture. These channels which enable CACS to control and get feedback from the execution of any concurrency model involve:-

- At the language level annotations in the model's code to inform CACS of the significant events such as action begin or action commit.

- Object events such as read and write communicated to CACS from the abstract machine's detection of swizzling and object update.

- CACS communicating events to the store to perform melding, aborting and the per-action shadow paging.

A number of issues concerning the complete integration of CACS into the Napier88 system still need investigation and are outlined in section 7.4. As a step

towards this, an instance of the concurrent architecture was constructed that directly supports an atomic transaction package written in Napier88. The atomic transaction package effectively acts as CACS and handles the transaction events and detects and resolves conflicts. A transaction commit results in an action meld where the propagation procedure for making changed objects globally visible is hardwired into the store using the double xor algorithm. From this implementation, and future implementations of different concurrency models, it is hoped that a clearer idea how a CACS system might be constructed will emerge.

7.2 Integrating Distribution

Maintaining the illusion of a uniform persistent address space transparently distributed across a number of nodes is beset with technological difficulties. The approach here is to tackle distribution from the opposite end of the transparency spectrum and provide a basic communication primitive in which objects can move between stores. The primitive can then be used as a building block on which any model can be constructed and so provides a high-level testbed in which hybrid schemes can be developed.

At the base level the distribution primitive makes all the transparency dimensions visible. The programmer can construct abstractions over this primitive to form desired transparencies for a particular model. At one extreme the architecture may be used to create a one-world model, admittedly at some cost, which could then be used for experimenting with the problems of a large scale fully-transparent distribution.

At the machine level the implementation is based on the widely-available TCP/IP protocol and thus offers an infrastructure that can be used to unite Napier88 stores across the internet.

Two example models of the concept were constructed :-

- A client/server style model that enables a user to browse remote stores and copy objects from these stores in a type-safe manner. The copy involves the transfer of the object's transitive closure to prevent pointer leaking.

- A model that provides a two-phase commit protocol that enables the Napier88 atomic transactions to perform a distributed synchronise across a number of nodes.

7.3 Building the Architecture

Engineering the integration of concurrency and distribution into the Napier88 architecture has required significant changes at all levels to the vanilla system. One of the important achievements in this work is that the implementation strategy used to realise the integration :-

- did so without the introduction of new language constructs.

- preserved the overall layering of the standard Napier88 architecture by augmenting the interfaces rather than altering them. This is really a testament to the genericity of the original layered design.

- probably, most importantly, protected the investment in Napier88 by ensuring that the large body of existing applications, software tools, compilers etc. all work as before.

Producing an integrated system has resulted in a number of innovative implementation components :-

- an after-look shadow paging scheme that is designed to get performance benefits within the limitations of the memory-mapping features of the SunOS system.

- an extended after-look scheme to handle concurrent access.

- incorporation of persistent threads and semaphores together with a process scheduler in the abstract machine.

- an atomic transaction package that dovetails into the architecture at the language level, the abstract machine for pid translation to capture object reads and writes and the store level to effect the per-action shadow paging and the double xor melding function.

- a socket abstraction that fits into Napier88's device I/O interface together with a mechanism for flattening and expanding an object's closure for transferring across a stream interface.

- a connection establishment protocol that interfaces between the Unix convention and the requirements of Napier88.

- enhancements to the atomic transaction package to enable a distributed two-phase commit between nodes.

7.4 Future Work

The provision of this architecture should really only be regarded as an initial result which forms a sound basis on which models of concurrency and distribution can be constructed. The specific research areas and experiments that follow from this work are itemised below.

7.4.1 Concurrency

CACS provides a system in which models of concurrency can be specified in an abstract, operational way that aids their implementation. The ultimate goal is to provide a system in which different concurrency schemes can be specified, constructed and compared in a meaningful way. There are still a number of open issues regarding the completion of CACS within the Napier88 architecture that need to be researched and developed. These include :-

- A delivery vehicle for turning CACS specifications into Napier88. This involves firstly the provision of a process to automatically (or semi-automatically) generate the annotations of a Napier88 program with the CACS events through program transformation. Secondly, using the Napier88 reflection technology [Kir92] to turn the annotations into code. For example these may be provided as procedure calls to CACS or alternatively as in-line code.

- A persistent architecture that can support the completed CACS. The work presented here provides an initial framework but the support for a complete CACS as a whiteboard architecture remains an issue. In particular :-

  At the abstract machine level the mechanism for reporting object reads and writes to CACS needs generalising. One possible approach may be to extend the thread context block to enable CACS-specific event handler procedures to be dynamically assigned to an action.

  At the store level the provision of a generic melding mechanism for propagating the changes made by a committing action to other actions. The double xor of the shadow pages only works in the cases where no two actions have modified the same object.

Specific experiments to evaluate the reference model will be carried out. These will produce hardwired instances of the architecture to support specific concurrency models. Building architectures to support models such as Sagas, transaction groups and nested transaction will give a fuller examination of issues of attaining a generic architecture to support CACS.

In the concurrent store the operation of using a local heap as a cache for the stable heap led to a duplication of effort to ensure the isolation of actions and detracted somewhat from the expected benefits of using shadow-paging. One future experiment will be to redesign the PAM so that it runs directly from the

stable heap. This will involve identifying the points where PAM assumes its working on virtual memory addresses rather than checking for persistent addresses. This change will also require a alternate method to swizzling for trapping when an action has been read or written. With all objects, new and transient as well as stable objects, being created on the stable heap an efficient stable heap garbage collector is required. This will lead investigations into generation and incremental collectors with the work of Kolodner [Kol92] on the concurrent tracking of newly stable objects of some interest. With two functionally equivalent Napier88 systems, one with a local heap and one without, a good basis would be available for measurement experiments.

7.4.2 Distribution

The provision of the distribution primitive provides a building block for constructing any models. Investigations will be undertaken into designing models for examining the problems of a distributed universal address space. These models may implement specific transparencies or combinations of transparencies to isolate a particular issue such as recovery or migration visibility. One interesting idea would be to develop a CACS equivalent for distribution in which models could be formally specified and constructed.

Specific implementations of known models such as RPC on top of the primitive will eventually lead to the provision of a large-scale fully-transparent distribution model.

Work on increasing node-to-node throughput will include investigations into minimising the depth of an objects closure and exploring the usefulness of employing compression to the object flattening.

7.4.3 Reliability

The provision of shadow-paging enables an atomic update to the store and ensures recoverability after a system crash. Future work in reliability will include investigations into making the object store resilient from media failure through mirroring or RAID technology.

It has been argued here that shadow paging may suit persistent systems which exhibit locality. It is worth investigating the validity of this claim through the provision of a framework in which shadow-paging implementations may be measured against logging alternatives. This will involve the design of a platform for building and comparing models of shadow-paging and logging models.

7.4.4 Measurement

Much of the work that has been implemented as part of this thesis has not yet been sufficiently analysed and measured. Future work will include research into how concurrency models and distribution models can be quantitatively compared. Similarly methods of comparing and measuring shadow paging and logging will be investigated.

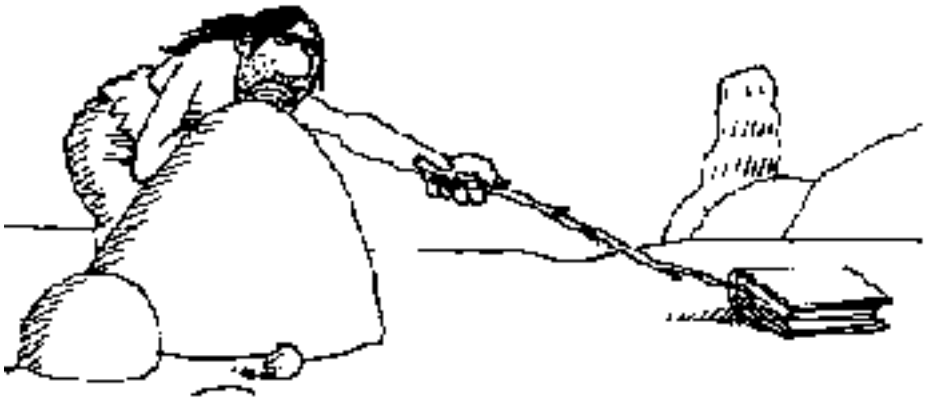
It is not clear exactly what metrics would be used but past experiments, such as the Predator project [KGC85], provide valuable pointers. In addition to measuring performance based on transaction throughput and mean response times, the Predator project also compared the cost of building recovery managers and measured the crash recovery time. Their results were derived from a purpose-built testbed.

The work of Atkinson [ABJ+93] in the measurement of persistent systems should be valuable in these experiments. Atkinson has developed a range of benchmarks in Napier88 that have been run under the single-threaded Napier88 system and offers some insights into the interpretation of measurement results.

7.5 Finale

This thesis has presented an attempt to integrate the notions of concurrency and distribution into a persistent framework in a flexible manner. The resulting architecture enables models of concurrency and distribution to be designed, constructed and executed in a persistent system. Providing concurrency and distribution as an add-on facility instead of building it into the system challenges the convention of related work. Whether or not this approach is superior is highly subjective but is founded on a conviction that a high-level solution delivers increased expressive power, safety and simplicity in the production of complex models.

It is hoped that providing this functionality has increased the expressive and modelling potential of a persistent system and that the work may broaden the appeal of persistence to a wider audience. However what metrics can one use to determine if such an approach is successful? Even if the work here becomes adopted, adapted and widely used, an argument based on popularity must by the same token decide if Cobol and C and systems such as Unix and MS-DOS should be considered successful. Whatever the outcome, the effort will not have been fruitless since the personal rewards gained from undertaking this research and the production of this thesis have finally convinced the author that there is a more interesting life beyond the System Manager's Guide [Mor85-93]. A case of WTFM superseding RTFM.



Appendix A Multithreading in Napier88

The provision of co-operating concurrency in Napier88 requires a way of expressing concurrent activity at the language level, a synchronisation primitive and a scheduler to control concurrent operation. An important point to note is that because the concurrent activities in this model interact by agreement rather than in conflict there is no need to isolate the effects of one action on the store from another. This means that the introduction of co-operating concurrency into the Napier88 system has no bearing on the object store architecture and can be fully implemented within the language and abstract machine.

Concurrent expression in this model is provided through an abstract data type that provides a package of procedures that allow the creation and manipulation of separate threads of control. The thread package is not built into the language but is obtained through the standard environment. The standard environment is a special Napier88 environment that contains many packages of standard functions. The specification of the thread package is given in figure A.1.

```
type    ThreadPack is abstype[ Thread ] (
                start     proc( proc() → Thread ) ;
                getThreadId    proc( → Thread ) ;
                kill,
                restart,
                suspend :       proc( Thread )
                )
```

Figure A.1: Thread Package

This abstract data type contains procedures to operate on threads. For some witness type *Thread* the operations are :-

- start : **proc**(**proc**() → Thread)

This procedure creates a new thread to execute the given void procedure, adds the thread to the list of threads, marks the thread as runnable, and returns an identifier for the thread. The thread completes when the given procedure completes.

- getThreadId : **proc**(→ Thread)

  This procedure returns the identifier of the currently executing thread.

- kill : **proc**(Thread)

  This procedure removes the thread denoted by the given identifier from the list of threads. If the thread is currently executing it is terminated.

- restart : **proc**(Thread)

  This procedure marks the thread denoted by the given identifier as runnable. If the thread is currently executing the procedure has no effect.

- suspend : **proc**(Thread)

  This procedure marks the thread denoted by the given identifier as suspended. If the thread is currently executing it is immediately suspended.

The thread package enables any number of void procedures to be executed concurrently without a change to the Napier88 language model. Threads can be nested to any depth and a thread will execute in the same environment as its parent. However there is no implicit dependency between a parent and child process; suspension or termination of one does not affect the other. The thread package is not dissimilar to the dynamic processes used in CPS-algol [Kra87]. One novelty of this approach is that in using an abstract data type the witness type cannot be discovered and hence thread ids are unforgable.

Synchronisation of Co-operating Threads

The facility for synchronisation of the threads is provided through a semaphore package shown in figure A.2.

```
type Semaphore is structure( wait,signal : proc() )
semaphoreGen : proc( int → Semaphore )
```

Figure A.2: Semaphore Package

- semaphoreGen : **proc**(**int** → Semaphore)

  This procedure takes an initial value for the semaphore and returns a structure containing procedures to operate on the semaphore.

- wait : **proc**()

  The value of the semaphore is decremented. If the new value is less than zero then the current thread is suspended and its dependency on the semaphore is recorded.

- signal : **proc**()

  The value of the semaphore is incremented. If the new value is less than or equal to zero, one of the threads suspended on the semaphore is selected and re-activated.

Dining Philosophers in Napier88 threads

As an example of how the threads are programmed the following listing gives a solution to the dining philosophers problem.

```
type ThreadPack is abstype[ Thread ] (
        start : proc( proc() → Thread ) ;
```

```
        getThreadId : proc( → Thread ) ;
        kill,restart,suspend : proc( Thread )
        )

type SemaphorePack is structure( wait,signal : proc() )
type message is bool

use PS() with Library : env in
use Library with Concurrency : env in
use Concurrency with threadPackage : ThreadPack ;
semaphoreGen : proc( int → SemaphorePack ) in
begin
        let enter = true ; let exit = false
        let pickup = true ; let putdown = false

        use threadPackage as X[ Thread ] in
        begin
                let Room = semaphoreGen( 4 )

                let room = proc( message : message )
                if message = exit then Room( signal )() else Room( wait )()

                let forkSemaphore = proc( i : int → SemaphorePack )
                semaphoreGen( 1 )

                let Forks = vector 0 to 4 using forkSemaphore

                let forks = proc( i : int ; message : message )
                if message = pickup then
                Forks( i )( wait )() else Forks( i )( signal )()

                let philosopherGenerator = proc( i : int → Thread )
                begin
                        let philosopher = proc()
                        while true do
                        begin
                                ! Think
                                room( enter ) ; ! Enter the room
                                forks( i,pickup ) ; ! Get one fork
                                forks( ( i + 1 ) rem 5,pickup ) ; ! Get two forks

                                ! Eat
                                forks( i,putdown ) ; ! Put down one fork
                                forks ( ( i + 1 ) rem 5,putdown )
                                ! Put down second fork

                                room( exit ) ! Leave the room
                        end
                        ! A new philosopher is born
                        X( start )( philosopher )
                end

                let philosophers = vector 0 to 4 using philosopherGenerator
        end
end
```

Appendix B Atomic Transaction Package

The listing below gives a the full description of the atomic transaction package hardwired into the system. The listing is split into 5 sections with the following interpretation :-

- a section which maintains a binary tree of objects read and written on a per-transaction basis.
- a section which maintains the transaction data structures.
- a section which has code to handle the transaction events. This includes the code that is called from the abstract machine when an object is read or written. The code for the conflict resolution on a commit uses the conflict serializability method described in chapter 4.
- a section which describes the language level interface to the package.
- an example program of transactions on a simple bank account.

```
! Here are the index (binary tree) types for the pids within a transaction
type pId is int

rec type pidIndex is variant (node : Node; tip : null)
& Node is structure (key : pId ; left, right : pidIndex)

let nilPidIndex = pidIndex (tip : nil)
rec let pidEnter = proc (k : pId; i : pidIndex -> pidIndex)
!Enter the value into the binary tree indexed by key
if i is tip then pidIndex (node : Node (k, nilPidIndex, nilPidIndex)) else
case true of
        k < i'node (key)        : { i'node (left) := pidEnter (k, i'node (left)) ; i }
        k > i'node (key)        : { i'node (right) := pidEnter (k, i'node (right)) ; i }
default                         : i

let pidLookup = proc (k : pId; i : pidIndex -> bool)
!lookup the value in the binary tree
begin
        let head := i
        while head is node and k ≠ head'node (key) do
        head := if k < head'node (key)          then head'node (left)
```

```
                                        else head'node (right)
        head is node
end
```

```
! Here are the list types for keeping the transactions
type tId is int
type transaction is structure (tid : tId ; thread : any ; readPids, writePids :
pidIndex)
rec type transactionList is variant (cons : Cons; tip : null)
& Cons is structure (hd : transaction ; tl : transactionList)

let mkTransactionList = proc (element : transaction;
                        tList : transactionList -> transactionList)
                        transactionList (cons : Cons (element, tList))

let removeTransaction = proc (tList : transactionList ;
                                element : transaction -> transactionList)
if tList is tip then mkTransactionList (element, tList) else
begin
        let eqTransaction = proc(a, b : transaction -> bool) ; a (tid) = b (tid)
        if eqTransaction (tList'cons (hd), element) then tList'cons (tl) else
        begin
                let done := false ; let this := tList
                while this'cons (tl) isnt tip and ~done do
                        if eqTransaction (this'cons (tl)'cons (hd), element) then
                        begin
                                this'cons (tl) := this'cons (tl)'cons (tl)
                                done := true
                        end else this := this'cons (tl)
                tList
        end
end

let getTransaction = proc (tList : transactionList; tid : tId -> transactionList)
begin
        let done := false
        while tList is cons and ~done do
                if tList 'cons (hd, tid) = tid then done := true
                else tList := tList 'cons (tl)
        tList
end

let addTransaction = proc (tList : transactionList;
        element : transaction -> transactionList)
begin
        let geTransaction = proc (a, b : transaction -> bool) ; a (tid) > b (tid)
        if tList is tip or getTransaction (tList'cons (hd), element)
        then mkTransactionList (element, tList) else
        begin
                let done := false ; let this := tList
                while this'cons (tl) isnt tip and ~done do
                        if geTransaction (this'cons (tl)'cons (hd), element) then
                        begin
                                this'cons (tl) := mkTransactionList (element,
                                                this'cons (tl))
```

```
                                done := true
                        end else this := this'cons (tl)
                if ~done do
                this'cons (tl) := mkTransactionList (element, this'cons (tl))
                tList
        end
end
```

```
! Code to handle transaction events
type threadPack is abstype [thread] (start : proc (proc () -> thread) ;
                                        kill, restart, suspend : proc (thread))

type SemaphorePack is structure (wait,signal : proc ())

type transactionPack is abstype [tid] (
                        createTransaction : proc (-> tid);
                        beginTransaction : proc (tid, proc ());
                        commitTransaction : proc (tid);
                        abortTransaction : proc (tid);
                        readPid : proc (tid, pId);
                        writePid : proc (tid, pId))

let createTransactionPack = proc (-> transactionPack)
use PS() with Library : env in
use Library with Concurrency, Transactions: env ; meld : proc () in
use Concurrency with threadPackage : threadPack ;
                        semaphoreGen : proc (int -> SemaphorePack) in
use Transactions with pidEnter : proc (pId, pidIndex -> pidIndex);
        pidLookup : proc (pId, pidIndex -> bool);
        addTransaction : proc (transactionList, transaction -> transactionList);
        removeTransaction : proc (transactionList,
                        transaction -> transactionList);
        getTransaction : proc (transactionList, tId -> transactionList) in
begin
        let emptyTransactionList = transactionList (tip : nil)
        let transactionsList := emptyTransactionList
        let tid := 0
        let mutex = semaphoreGen (1)
        let wait = proc () ; mutex (wait) ()
        let signal = proc () ; mutex (signal) ()
        let startThread = proc (a : proc () -> any)
        begin
                let this := any (0)
                use threadPackage as X [thread] in
                begin
                        let thisOne = X (start) (a)
                        this := any (thisOne)
                end
                this
        end

        let killThread = proc (a : any)
        use threadPackage as X [thread] in
        project a as Y onto
                thread : X (kill) (Y)
```

```
        default : {}

        let emptyAny = any (0)

        let createTransaction = proc (-> tId)
        begin
                wait ()
                tid := tid + 1
                transactionsList := addTransaction (transactionsList,
                transactionType (tid, emptyAny, nilPidIndex, nilPidIndex))
                let x = tid
                signal ()
                x
        end

        let beginTransaction = proc (tid : tId ; prog : proc ())
        begin
                wait ()
                let this = getTransaction (transactionsList, tid)
                if this ≠ emptyTransactionList do
                this'cons (hd) (thread) := startThread (prog)
                signal ()
        end

        let stopThis = proc (tid : tId)
        begin
                let this = getTransaction (transactionsList, tid)
                if this ≠ emptyTransactionList do
                begin
                        killThread (this'cons (hd) (thread))
                        transactionsList := removeTransaction (transactionsList,
                                                this'cons (hd))
                end
        end

        let abortTransaction = proc (tid : tId)
        begin
                wait ()
                stopThis (tid) ! Let go of all shadows
                signal ()
        end

        rec let overlap = proc (a, b : pidIndex -> bool)
        a isnt tip and (overlap (a'node (left), b) or
                                pidLookup (a'node (key), b) or
                                overlap (a'node (right), b))

        let findConflictsandAbort = proc (tid : tId)
        begin
                let theseWrites = getTransaction (transactionsList, tid)
                if theseWrites ≠ emptyTransactionList do
                begin
                        let writes = theseWrites'cons (hd) (writePids)
                        let this := transactionsList
                        while this isnt tip and this'cons (hd, tid) ≠ tid do
                        begin
                                if overlap (writes, this'cons (hd, readPids)) do
```

```
                                                    abortTransaction(tid)
                            this := this'cons (tl)
                        end
                end
        end

        let commitTransaction = proc (tid : tId)
        begin
                wait ()
                meld ()                         ! Assume successful
                findConflictsandAbort (tid)
                stopThis (tid)
                signal ()
        end

        let readPid = proc (tid, pid : pId) ! Called from PAM
        begin
                wait ()
                let this = getTransaction (transactionsList, tid)
                if this ≠ emptyTransactionList do
                        this'cons (hd, readPids) := pidEnter (pid,
                                                this'cons (hd, readPids))
                signal ()
        end

        let writePid = proc (tid, pid : pId)  ! Called from PAM
        begin
                wait ()
                let this = getTransaction (transactionsList, tid)
                if this ≠ emptyTransactionList do
                        this'cons (hd, writePids) := pidEnter (pid,
                                        this'cons (hd, writePids))
                signal ()
        end

        transactionPack [int] (createTransaction, beginTransaction,
                                abortTransaction,
                                commitTransaction, readPid, writePid)
end
```

```
! User interface to transaction pack

type userTransactions is structure (createTransaction : proc (-> any),;
                        beginTransaction : proc (any, proc (any -> proc ()));
                        abortTransaction, commitTransaction : proc (any))

let userTransactionsPack = proc (-> userTransactions)
begin
        let transactionPackage = createTransactionPack ()

        let createTransaction = proc (-> any)
        begin
                let this := any (0)
                use transactionPackage as X [TID] in
                        this := any (X (createTransaction) ())
```

```
                this
        end

        let beginTransaction = proc (tid : any ; a : proc (any -> proc ()))
        use transactionPackage as X [TID] in
        project tid as Y onto
                TID: X (beginTransaction) (Y, a (tid))
        default : {}

        let abortTransaction = proc (tid : any)
        use transactionPackage as X [TID] in
        project tid as Y onto
                TID: X (abortTransaction) (Y)
        default : {}

        let commitTransaction = proc (tid : any)
        use transactionPackage as X [TID] in
        project tid as Y onto
                TID: X (commitTransaction) (Y)
        default : {}

        userTransactions(createTransaction,beginTransaction
                abortTransaction,commitTransaction)
end
```

```
! Example Use
let thisTransaction = userTransactionsPack ()
let createTransaction = thisTransaction (createTransaction)
let beginTransaction = thisTransaction (beginTransaction)
let abortTransaction = thisTransaction (abortTransaction)
let commitTransaction = thisTransaction (commitTransaction)

type account is structure (bal, Limit : int)

let withdraw = proc (tid : any -> proc ())
! Withdraw debit pounds from ac.
proc ()
use PS() with Library : env; accounts: *account; upb : proc [t] (*t -> int);
                readInt : proc (-> int); writeString : proc (string) in
begin
        let ac = accounts (upb [account] (accounts))
        writeString ("Please input account requested : 'n")
        let debit = readInt ()
        let result = ac (bal) - debit
        if result > ac (Limit) and debit > 0 then
        begin
                ac (bal) := result
                commitTransaction (tid)
        end else abortTransaction (tid)
end
! loop forever preparing transactions in this style
let tid = createTransaction ()
beginTransaction (tid, withdraw)
```

Appendix C Stacos user interface

To simplify the interface to accessing remote stores a graphical communications tool was developed and incorporated into the Napier88 programming environment [Kir92]. The programming environment is an application written entirely in Napier88 that supports the interactive development of Napier88 programs. It comprises of a number of tools including a persistent window manager, an editor, callable compiler and object browser. The programming environment allows source programs to be constructed, compiled and linked into the store all within the one environment. The communications tool enables the connection and remote scan and copy of other stores to be directed by user gesture.

To explain fully the functionality of the communications tool and its incorporation into the programming environment requires some background description of some of the existing tools and in particular the function of the *Local Values* window.

The programming environment provides editing tools that can be used to develop program source. An *evaluate* button in an editor window compiles the selected text. Any compilation errors are reported in an *Output* window. If the compilation is successful then the resulting code is executed. If the code is non-void, then the user is prompted for a name for the result and that name is entered in the *Local Values* window shown in figure C.1. The names and corresponding values in the Local Values window are automatically brought into scope in subsequent compilations.

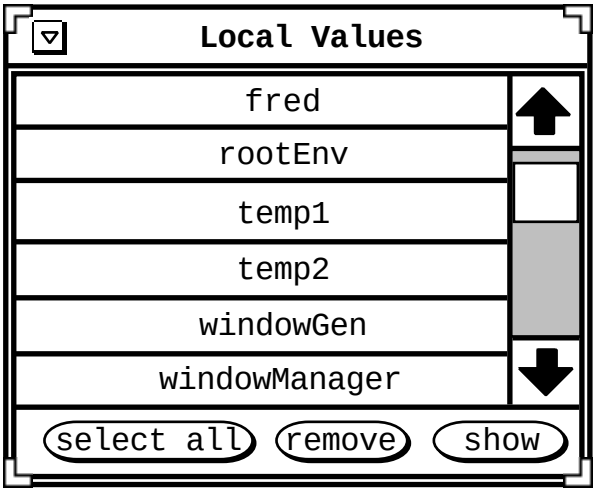


Figure C.1: Local Values window

The communications tool allows the symbolic naming of remote object stores to be done using the *Remote Stores* window shown in figure C.2. When adding a new entry the user is prompted via a dialogue box for three parameters corresponding to the fields of the *RemoteStore* structure, namely the symbolic name for the remote store, the remote host name and the pathname of the store. The symbolic name appears in the *Remote Stores* window.

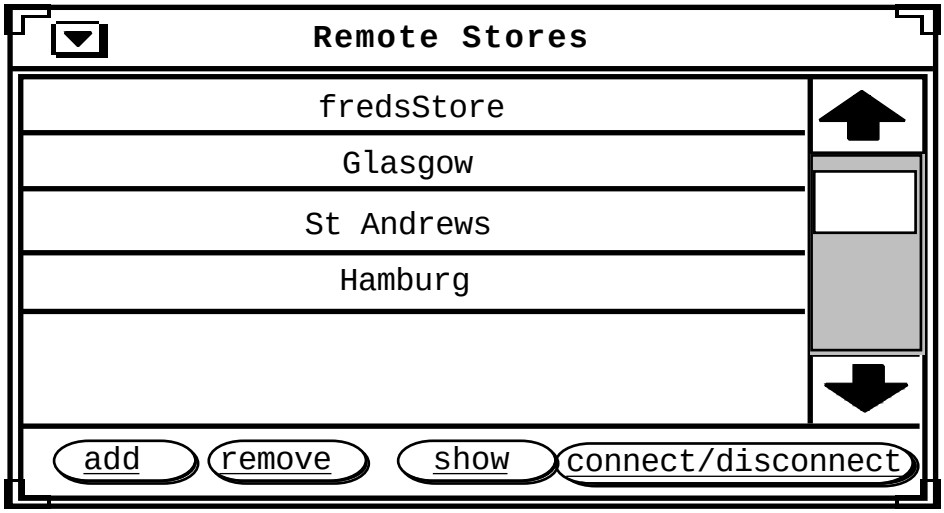


Figure C.2: Remote Stores window

To access a remote store the user selects an entry in the window and then presses the *connect/disconnect* button. If the connection is successful then the root environment of the remote store is browsed and a new window displayed

showing the bindings of the remote root environment. This is illustrated in figure C.3. An unsuccessful connection results in the error string being displayed in the Output window.

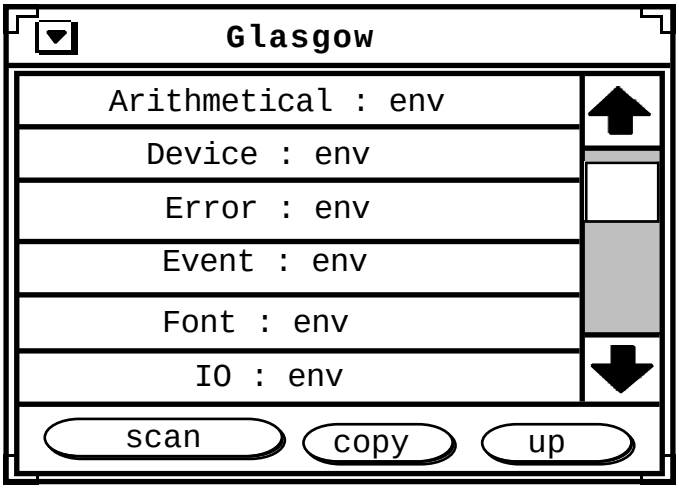


Figure C.3: A remote connection window

The window title displays the symbolic name used to refer to the remote store. The *scan* button is used to move down the remote object store graph, displaying the bindings of the selected entry. Each environment that is scanned results in the window being updated to display the bindings for that remote environment. Whilst scanning down in this manner, the output window displays the "pathname" of the environment currently being scanned. The *up* button has the same effect as the *scan* button but is used to move back up the object graph. The *copy* button copies the selected object from the remote store to the local store and makes an entry in the *Local Values* window where it can then be browsed or used in subsequent compilations.

Because the remote object store is being used concurrently then each invocation of the *scan*, *copy* or *up* button must check that the path to the selected environment or object is still valid and has not been removed since the last invocation. These functions cope with this type of failure by displaying a message in the *Output Window*.

Finally the communication to a remote store can be disconnected by selecting the *connect/disconnect* button.

References

[AB87] Atkinson, M.P. & Buneman, O.P. “Types and Persistence in Database Programming Languages”. ACM Computing Surveys 19, 2 (1987) pp 105-190.

[ABB+86] Acceta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A. & Young, M. “Mach: A New Kernel Foundation for Unix Development”. USENIX (1986) pp 93-112.

[ABC+83] Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R. “An Approach to Persistent Programming”. Computer Journal 26, 4 (1983) pp 360-365.

[ABC+84] Atkinson, M.P., Bailey, P.J., Cockshott, W.P., Chisholm, K.J. & Morrison, R. “Progress with Persistent Programming”. Universities of Glasgow and St Andrews Technical Report PPRR-8-84 (1984).

[ABJ+92] Atkinson, M.P., Birnie, A., Jackson, N. & Philbrow, P.C. “Measuring Persistent Object Systems” In Proc. 5th International Workshop on Persistent Object Systems, San Miniato, Italy (1992). In Persistent Object Systems (Eds. A.Albano & R.Morrison). Springer-Verlag pp 63-85.

[ACC81] Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P. “PS-algol: An Algol with a Persistent Heap”. ACM SIGPLAN Notices 17, 7 (1981) pp 24-31.

[ACC83] Atkinson M.P., Chisholm K.J. & Cockshott W.P. CMS - A Chunk Management System. Software Practice and Experience, vol. 13, no. 3, 1983 pp 259-272.

[AD85a] Agrawal, R. & DeWitt, D. “Recovery Architectures for Multiprocessor Database Machines”. In SIGMOD International Conference on Management of Data, (1985) pp 131-147.

[AD85b] Agrawal, R. & DeWitt, D. “Integrated Concurrency Control and Recovery Mechansims: Design and Performance Evaluation”. ACM Transactions on Database Systems, 10,4 (1985) pp 529-564.

[AM85] Atkinson, M.P. & Morrison, R. “Procedures as Persistent Data Objects”. ACM Transactions on Programming Languages and Systems 7, 4 (1985) pp 539-559.

[AM88] Atkinson, M.P. & Morrison, R. “Types, Bindings and Parameters in a Persistent Environment”. In Data Types and Persistence, Atkinson, M.P., Buneman, O.P. & Morrison, R. (ed), Springer-Verlag (1988) pp 3-20.

[AMP86] Atkinson, M.P., Morrison, R. & Pratten, G.D. “A Persistent Information Space Architecture”. In Proc. 9th Australian Computing Science Conference, Australia (1986).

[AMR92] Abdullahi, S.E., Miranda, E. & Ringwood, G.A. “Collection Schemes for Distributed Garbage”. In Bekkers, Y. & Cohen, J. (ed), International Workshop on Memroy Management, LNCS 637, Springer-Verlag, 1992 pp 43-81.

[ANS89] “ANSA Reference Manual Volume A”. ANSA, Poseidon House, Castle Park, Cambridge, UK (1989).

[Atk78] Atkinson, M.P. “Programming Languages and Databases”. Proc. 4th IEEE International Conference on Very Large Databases, 1978 pp 408-419.

[Atk92] Atkinson, M.P. “Persistent Foundations for Scalable Multi-Paradigmal Systems” ESPRIT BRA Project 6309 FIDE Technical Report FIDE/92/51 (1992).

[Bac92] Bacon, J. “Concurrent Systems: An Integrated Approach to Operating Systems, Database and Distributed Systems”. Addison-Wesley, Wokingham, 1992.

[BC85] Brown, A.L. & Cockshott, W.P. “The CPOMS Persistent Object Management System”. Universities of Glasgow and St.Andrews PPRR-13, Scotland, 1985.

[BCC+88] Brown A.L., Carrick R., Connor R.C.H., Dearle A. & Morrison R. The Persistent Abstract Machine. Universities of Glasgow and St.Andrews PPRR-59, Scotland, 1988.

[BDM+90] Brown, A.L., Dearle, A., Morrison, R., Munro, D.S., Rosenberg, J. "A Layered Persistent Architecture for Napier88". International Workshop on Computer Architectures to Support Security and Persistence of Information, Universität Bremen, West Germany, (May 1990). In Security and Persistence. (Eds. J.Rosenberg & L.Keedy). Springer-Verlag, 155-172.

[BF89] Brossler, P. & Freisleben B. “Transactions on Persistent Objects” 3rd International Workshop on Persistent Object Systems, Newcastle, N.S.W., (1989). In Persistent Object Systems.(Eds. J.Rosenberg & D.Koch). Springer-Verlag pp 341-350.

[BMM80] Bailey, P.J., Maritz, P. & Morrison, R. “The S-algol abstract machine”. Technical Report CS-80-2 (1980), University of St Andrews.

[BMR82] Brownbridge, D.R., Marshall, L.F. & Randell, B. “The Newcastle Connection or Unixes of the World Unite!”. Software Practice and Experience, 12 (1982) pp 1147-1162.

[BR91] Brown, A.L. & Rosenberg, J. “Persistent Object Stores: An Implementation Technique”. In Dearle, Shaw, Zdonik (eds.), Implementing Persistent Object Bases, Principles and Practice, Morgan Kaufmann, 1991 pp 199-212.

[Bri70] Brinch Hansen, P. “The Nucleus of a Multiprogramming System”. CACM 13,4 (1970) pp 238-241.

[Bri75] Brinch Hansen, P. “The Programming Language Concurrent Pascal”. IEEE Transactions on Software Engineering, SE-1,2 (1975) pp 199-207.

[Bro89] Brown, A.L. “Persistent Object Stores”. Ph.D. Thesis, University of St Andrews (1989).

[BT85] Bates, K. & TeGrotenhuis, M. “Shadowing Boosts System Reliability”. Computer Design, 1985.

[CAB+81] Chamberlin, D.D., Astrahan, M.M., Blasgen, M.W., Gray, J.N., King, W.F., Lindsay, B.G., Lorie, R.A., Mehl, J.W., Price, T.G., Selinger, P.G., Schkolnick, M., Slutz, D.R., Traiger, I.L., Wade, B.W. & Yost, R.A. “A History and Evaluation of System R” CACM 24,10 (1981) pp 632-646.

[CAB+84] Cockshott, W.P., Atkinson, M.P., Bailey, P.J., Chisholm, K.J. & Morrison, R. "A persistent object management system". Software, Practice and Experience 14, 1 (January 1984) pp 49-71.

[CBC+89] Connor, R.C.H., Brown, A.L., Carrick, R., Dearle, A. & Morrison, R. "The Persistent Abstract Machine". 3rd International Workshop on Persistent Object Systems, Newcastle, N.S.W., (January 1989), 80-95. In Persistent Object Systems (Eds. J.Rosenberg & D.Koch). Springer-Verlag, 353-366.

[CH74] Campbell, R.H. & Habermann, A.N. “TheSpecification of Process Synchronisation by Path Expressions”. In Operating Systems, Springer-Verlag, Berlin (1974).

[Cha78] Challis, M.P. “Data Consistency and Integrity in a Multi-User Environment”. Databases: Improving Usability and Responsiveness, Academic Press, 1978.

[CL85] Chandy, K.M. & Lamport, L. “Distributed Snapshots: Determining Global States of Distributed Systems”. ACS TOCS 3,1 (1985) pp 63-75.

[Coc89] Cockshott, P. “Design of POMP - A Persistent Object Management Processor”. 3rd International Workshop on Persistent Object Systems, Newcastle, N.S.W., (January 1989), 80-95. In Persistent Object Systems (Eds. J.Rosenberg & D.Koch). Springer-Verlag pp 367-376.

[Coc90] Cockshott, P. “Implementing 128 Bit Persistent Addresses on 80x80 Processors”. In Security and Persistence, Rosenberg, J. & Keedy, J.L. (ed), Springer-Verlag (1990) pp 123-136.

[Con90] Connor, R.C.H. “Types and Polymorphism in Persistent Programming Systems”. Ph.D. Thesis, University of St Andrews (1990).

[CRW91] Cooper, R., Roberts, A. & Wai, F. “An Implementation of a Co-opertaive Locking Scheme for a Persistent Programming Language”. In Dearle, Shaw, Zdonik (eds.), Implementing Persistent Object Bases, Principles and Practice, Morgan Kaufmann, 1991 pp 319-328.

[Cut92] Cutts, Q.I. “Delivering the Benefits of Persistence to System Construction and Execution”. Ph.D. Thesis, University of St Andrews (1992).

[Dav73] Davies, C.T. “Recovery semantics for a DB/DC System”, In Proc. ACM Annual Conference, (1973) pp 136-141.

[Dav78] Davies, C.T. “Data Processing Spheres of Control”, IBM Systems Journal 17,2 (1978) pp 179-198.

[Dea87] Dearle, A. “Constructing Compilers in a Persistent Environment”. In Proc. 2nd International Workshop on Persistent Object Systems, Appin, Scotland (1987).

[Dea88] Dearle, A. “On the Construction of Persistent Programming Environments”. Ph.D. Thesis, University of St Andrews (1988).

[Dec78] Digital Equipment Corporation. “VAX/VMS Summary Description”. DEC 1978.

[Dij65] Dijkstra, E.W. “Co-operating Sequential Processes”. Technical Report EWD-123, Technological University, Eindhoven (1965).

[DRV91] Dearle, A., Rosenberg, J. & Vaughan, F. “A Remote Execution Mechansim for Distributed Homogenous Stable Stores”. In Database Programming Languages: Bulk Types & Persistent Data (eds P.Kanellakis & J.W.Schmidt). Morgan Kaufmann Publishers Inc., Palo Alto, Ca, USA, (1992) pp 125-138.

[DVM+92] De Francesco, N., Vaglini, G., Mancicn, L.V. & Pereira Paz, A. “Specification of Concurrency Control in Persistent Programming Languages”. In Proc. 5th International Workshop on Persistent Object Systems, San Miniato, Italy (1992). In Persistent Object Systems (Eds. A.Albano & R.Morrison). Springer-Verlag pp 126-143.

[EG90] Ellis, C.A. & Gibbs, S.J. “Concurrency Control in Groupware Systems”. In Proc. SIGMOD International Conference on Management of Data. (1990) pp 399-407.

[EGL+76] Eswaran, K.P., Gray, J.N., Lorie, R.A. & Traiger, I.L. “The Notions of Consistency and Predicate Locks in a Database System”. CACM 19,11 (1976) pp 624-633.

[ELA88] Ellis, J.R., Li, K. & Appel, A.W. “Real-time Concurrent Collection on Stock Multiprocessors”. Technical Report 25, Digital Equipment Corporation (1988).

[FZ89] Fernandez, M.F. & Zdonik, S. B. “Transaction Groups: A Model for Controlling Co-operative Transactions”. 3rd International Workshop on Persistent Object Systems, Newcastle, N.S.W., (1989). In Persistent Object Systems.(Eds. J.Rosenberg & D.Koch). Springer-Verlag pp 341-350.

[GAD+92] Gruber, 0., Amsaleg, L., Daynes L. & Valduriez, P. “ Eos: An Environment for Object-Based Systems”. In Proc. 25th Hawaii Conference on System Sciences 1,1 (1992) pp 757-768.

[Gar83] Garcia-Molina, H. “Using Semantic Knowledge for Transaction Processing in a Distributed Database”. ACM Transactions on Database Systems, 8,2 (1983) pp 186-213.

[GMB+81] Gray, J., McJones, P., Blasgen, M., Lindsay, B., Lorie, R., Price, T., Putzolu, F. & Traiger, I. “The Recovery Manager of the System R Database Manager”. ACM Computing Surveys, vol. 13, no. 2, June 1981 pp 223-242.

[Gra78] Gray, J.N. “Notes on Database Operating Systems”. LNCS 60, Springer-Verlag (1978).

[GS87] Garcia-Molina, H. & Salem, K. “Sagas”. In Proc. SIGMOD International Conference on Management of Data. (1987) pp 249-259.

[Hag87] Hagmann, R.B. “Reimplementing the Cedar file system using logging and group commit”. In Proc. 11th Symposium on Operating Systems Principles, 1987 pp 155-162.

[Hoa72] Hoare, C.A.R. “Towards a Theory of Parallel Programming”. In Operating Systems Techniques, Academic Press, London pp 61-71.

[Hoa74] Hoare, C.A.R. “Monitors: An Operating Systems Structuring Concept”. CACM 17,10 (1974) pp 549-557.

[Hoa75] Hoare, C.A.R. “Recursive Data Structures” International Journal of Computer and Information Science 4,2 (1975) pp 105-132.

[Hoa78] Hoare, C.A.R. "Communicating sequential processes". CACM 21, 8 (1978) pp 666-677.

[HR73] Horning, J.J. & Randell, B. “Process structuring”. Computing Surveys 5,1 (1973) pp 427-430.

[HR91] Henskens, F.A. & Rosenberg, J. “A Capability-Based Distributed Shared Memory”. In Proc 4th Australian Computer Science Conference (1991) pp 29/1-29/12.

[HZ87] Hornick, M. & Zdonik, S. B. “A Shared Segmented Memory System for an Object-Oriented Database”. ACM Transactions on Office Information Systems 5,1 (1987) pp 70-95.

[Jef85] Jefferson, D.R. “Virtual Time”. ACM TOPLAS (1985) pp 404-425.

[Joh71] Johnston, J.B. “The contour model of block structure processes”. ACM SIGPLAN Notices 6,3 (1971) pp 56-82.

[KB92] Keedy, L. & Brossler, P. “Implmenting Databases in the Monads Virtual Memory”. In Proc. 5th International Workshop on Persistent Object Systems, San Miniato, Italy (1992). In Persistent Object Systems (Eds. A.Albano & R.Morrison). Springer-Verlag pp 318-338.

[KGC85] Kent, J., Garcia-Molina, H. & Chung, J. “An experimental evaluation of crash recovery mechanisms”. In Proc.4th ACM Symposium on Principles of Database Systems (1985) pp 113-122.

[Kir92] Kirby, G.N.C. “Reflection and Hyper-Programming in Persistent Programming Systems”. Ph.D. Thesis, University of St Andrews (1992).

[Kol87] Kolodner, E.K. “Recovery Using Virtual Memory”. M.Sc. Thesis, MIT (1987).

[Kol92] Kolodner, E.K. “Atomic Incremental Garbage Collection and Recovery for a Large Stable Heap”. Ph.D. Thesis, MIT (1992).

[KR90] Koch, D. & Rosenberg, J. “A Secure RISC-based Architecture Supporting Data Persistence”. In Security and Persistence, Rosenberg, J. & Keedy, J.L. (ed), Springer-Verlag (1990) pp 188-201.

[Kra87] Krablin, G.L. "Building Flexible Multilevel Transactions in a Distributed Persistent Environment". 2nd International Workshop on Persistent Object Systems, Appin, (August 1987) pp 213-234.

[KSD+91] Koch, B., Schunke, T., Dearle, A., Vaughan, F., Marlin, C., Fazakerley, R. & Barter C. “Cache Coherency and Storage Management in a Persistent Object System”. in Dearle, Shaw, Zdonik (eds.), Implementing Persistent Object Bases, Principles and Practice, Morgan Kaufmann, 1991 pp 103-113.

[LH89] Li, K & Hudak, P. “Memory Coherence in Shared Virtual Memory”. ACM TOCS 17,4 (1989) pp 321-359.

[Li86] Li, K. “Shared Virtual Memory on Loosely Coupled Microprocessors”, Ph.D. Thesis, Yale University (1986).

[Lis84] Liskov, B.H. “Refinement - From Specification to Implementation, The Argus Language and System”. Lecture Notes for the Advanced Course on Distributed Systems - Methods and Tools for Specification, Institute for Informatics, Technical University of Munich, 1984.

[Lor77] Lorie, A.L. Physical Integrity in a Large Segmented Database, ACM Transactions on Database Systems, 2,1 (1977) pp 91-104.

[MBC+87] Morrison, R., Brown, A.L., Connor, R.C.H. & Dearle, A. “Polymorphism, Persistence and Software Reuse in a Strongly Typed Object Oriented Environment”. Universities of Glasgow and St Andrews Technical Report PPRR-32-87 (1987).

[MBC+88] Morrison, R., Brown, A.L., Carrick, R., Connor, R.C.H. & Dearle, A."On the Integration of Object-Oriented and Process-Oriented Computation in Persistent Environments". Proc. 2nd International Workshop on Object-Oriented Database Systems, West Germany (1988). In Lecture Notes in Computer Science (Ed. K.R.Dittrich), 334. Springer-Verlag, (September 1988) pp 334-339.

[MBC+89] Morrison, R., Brown, A.L., Connor, R.C.H. & Dearle, A. “The Napier88 Reference Manual”. University of St Andrews Technical Report PPRR-77-89 (1989).

[MBC+90] Morrison, R., Brown, A.L., Connor, R.C.H., Cutts, Q.I., Kirby, G.N.C., Dearle, A., Rosenberg, J. & Stemple, D. “Protection in Persistent Object Systems”. In Security and Persistence, Rosenberg, J. & Keedy, J.L. (ed), Springer-Verlag (1990) pp 48-66.

[MDC+91] Morrison R., Dearle A, Connor R.C.H. & Brown A.L. "An ad hoc Approach to the Implementation of Polymorphism". ACM.TOPLAS. (July 1991) pp 342-371.

[Mi80] Milner, R. “A Calculus of Communicating Systems”. LNCS 92, Springer-Verlag, 1980.

[Mi92] Milner, R. “Functions as Processes”. Mathematical Structures for Computer Science, 2 (1992) pp 119-141.

[Mor79] Morrison, R. “On the Development of Algol”. Ph.D. Thesis, University of St Andrews (1979).

[Mor79b] Morrison, R. “S-algol reference manual”. Technical Report CS-79-1 (1979), University of St Andrews.

[Mor85-93] Morrison, R. “There is more to life than mucking about with Unix”. Private Communication.

[Mos81] Moss, J.E.B. “Nested Transactions: An Approch to Reliable Distributed Computing”. Ph.D. Thesis, MIT (1981).

[Mos89] Moss, J.E.B. “Addressing Large Distributed Collections of Persistent Objects: The Mneme Project’s Approach”, In Database Programming Languages. (Eds. R.Hull, R.Morrison & D.Stemple). Morgan Kaufmann Publishers Inc., Palo Alto, Ca, USA pp 358-374.

[MRT+90] Mullender, S.J., Rossum, G. Van, Tanenbaum, A.S., Renesse, R. Van & Staveren, H. Van. “Amoeba: A Distributed Operating System for the 1990s”. IEEE Computer 23,5 (1990) pp 44-53.

[MS88] Moss, J.E.B. & Sinofsky, S. “Managing persistent data with Mneme: Designing a reliable shared object interface”. In Dittrich, K.R. (ed.) Advances in Object-Oriented Database Systems: Second International Workshop on Object-Oriented Database Systems, LNCS 334, Springer-Verlag, 1988 pp 298-316.

[Nel81] Nelson, B.J. “Remote Procedure Call”. Ph.D. Thesis, Carnegie-Mellon University (1981).

[NH82] Needham, R.M. & Herbert, A.J. “The Cambridge Distributed Computing System”. Addison-Wesley, Wokingham (1982).

[NSZ91] Nodine, M.H., Skarra, A.H. & Zdonik, S. B. “Synchronisation and Recovery in Co-operative Transactions”. In Dearle, Shaw, Zdonik (eds.), Implementing Persistent Object Bases, Principles and Practice, Morgan Kaufmann, 1991 pp 329-342.

[NZ92] Nodine, M.H. & Zdonik, S. B. “Co-operative Transaction Hierarchies: Transaction Support for Design Applications”. VLDB Journal 1,1 (1992) pp 41-80.

[OLS85] Oki, B., Liskov, B. & Scheifler, R. “Reliable Object Storage to Support Atomic Actions”. In Proc 10th Symposium on Operating Systems Principles, 1985 pp 147-159.

[OS93] Orji, C.U & Solworth, J.A. “Doubly Distorted Mirrors”. In Proc. SIGMOD International Conference on Management of Data, Washington, D.C., (May 1993) pp 307-316.

[PB61] Peterson & Brown “Cyclic codes for error detection” Proceedings of the IRE Volume 49 1961.

[Ras86] Rashid, R. “Threads of a New System”. Unix Review 4 (Aug 1986) pp 37-49.

[RHB+90] Rosenberg J., Henskens F., Brown A.L., Morrison R. & Munro D.S. "Stability in a Persistent Store Based on a Large Virtual Memory.". International Workshop on Computer Architectures to Support Security and Persistence of Information, Universität Bremen, West Germany, (May 1990). In Security and Persistence. (Eds. J.Rosenberg & L.Keedy). Springer-Verlag pp 229-245.

[RO91] Rosenblum, M. & Ousterhout, J.K. “The design and implementation of a log-structured file system”. In Proc 13th Symposium on Operating Systems Principles, 1991 pp 1-15.

[Ros83] Ross, G.D.M. “Virtual Files: A Framework for Experimental Design”. Ph.D. Thesis, University of Edinburgh (1983).

[RR81] Rashid, R. & Robertson, G. “Accent; A Communication Oriented Network Operating System Kernel”. In Proc. 8th Symposium on Operating Systems Principles (1981) pp 64-75.

[SDP91] Shrivastava, S.K., Dixon, G.N. & Parrington, G.D. “An Overview of the Arjuna Distributed Programming System”. IEEE Software 8, 1 (1991) pp 66-73.

[SG90] Stamos, J.W. & Gifford, D.K. “Remote Evaluation”. ACM TOPLAS 12,4 (1990) pp 537-565.

[SGK+85] Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D. & Lyon, B. “Design and Implementation of the Sun Network Filesystem”. In Proc. USENIX Summer Conference (1985) pp 119-130.

[SKW92] Singhal, V., Kakkad, S.V. & Wilson, P.R. “Texas: an efficient, portable persistent store”. In Proc. 5th International Workshop on Persistent Object Systems, San Miniato, Italy (1992). In Persistent Object Systems (Eds. A.Albano & R.Morrison). Springer-Verlag pp 11-33.

[SM92] Stemple, D. & Morrison, R. "Specifying Flexible Concurrency Control Schemes: An Abstract Operational Approach". Australian Computer Science Conference 15, Tasmania (1992) pp 873-891.

[SO91] Solworth, J.A. & Orji, C.U. “Distorted Mirrors”. ACM Parallel and Distributed Information Systems, 1991 pp 10-17.

[Sun90] Sun Microsystems Inc. “SunOS Reference Manual”. Report 800-3827-10, 1990.

[Sut91] Sutton, S. “A Flexible Consistency Model for Persistent Data in Software-Process Programming”. In Dearle, Shaw, Zdonik (eds.), Implementing Persistent Object Bases, Principles and Practice, Morgan Kaufmann, 1991 pp 305-319.

[SY85] Strom, R.E. & Yemini, S. “Optimistic Recovery in Distributed Systems”. ACM TOCS 3,3 (1985) pp 204-226.

[Tha86] Thatte S.M. “Persistent Memory: A Storage Architecture for Object Oriented Database Systems”. Proc. ACM/IEEE 1986 International Workshop on Object Oriented Database Systems, Pacific Grove, CA, September 1986 pp 148-159.

[VD92] Vaughan, F. & Dearle, A. “Supporting Large Persistent Stores using Conventional Hardware”. In Proc. 5th International Workshop on Persistent Object Systems, San Miniato, Italy (1992). In Persistent Object Systems (Eds. A.Albano & R.Morrison). Springer-Verlag pp 34-53.

[VDD+91] Velez, F., Darnis, V., DeWitt, D., Futtersack, P., Harrus, G., Maier, D., and Raoux, M. “Implementing the O_2 object manager: some lessons” In In Dearle, Shaw, Zdonik (eds.), Implementing Persistent Object Bases, Principles and Practice, Morgan Kaufmann, 1991 pp 131-138.

[Wai87] Wai, F. “Distribution and Persistence”. In Proc. 2nd International Workshop on Persistent Object Systems, Appin, Scotland (1987) pp 207-225.

[Wai88] Wai, F.H.W. “Distributed Concurrent Persistent Languages: an Experimental Design and Implementation” Ph.D. Thesis, University of Glasgow (1988).

[WF90] Wu, K.L. & Fuchs, W.K. “Recoverable Shared Virtual Memory”. IEEE Transactions on Computers, 39,4 (1990) pp 460-469.