An Examination of Operating System Support for Persistent Object Systems

Alan Dearle†, John Rosenberg∆, Frans Henskens¥,
Francis Vaughan† & Kevin Maciunas†

† Department of Computer Science
University of Adelaide
S.A., 5001, Australia

∆ Department of Computer Science
University of Sydney
N.S.W., 2006, Australia

¥ Department of Computer Science
University of Newcastle
N.S.W., 2308, Australia

Abstract

In this paper, operating system support for persistent systems that execute on conventional hardware architectures is examined. The focus of the paper is to examine the inadequacies of traditional operating systems as vehicles for the construction of persistent systems. We concentrate on four major areas, namely: addressing, stability and resilience, process management and protection. We examine the consequences of making the operating system kernel itself persistent. We conclude by outlining the requirements which must be met by future operating systems designed to support orthogonal persistence.

1. Introduction

Over the past ten years much research effort has been expended in attempting to build systems which provide orthogonal persistence [1, 3, 5, 6, 7, 8, 16, 27]. The idea behind persistence [12] is simple: all data in a system should be able to persist (survive) for as long as that data is required. Orthogonal persistence means that *all* data may be persistent and that data may be manipulated in a uniform manner regardless of the length of time it persists. In this sense persistent systems provide a uniform abstraction over storage. Orthogonal persistence is not found in contemporary operating systems, nor in most programming languages or database systems. In these systems, long lived data is treated in a fundamentally different manner from short lived data. Traditionally, long term data is maintained in a database or file system and short term data is managed by a programming language.

A number of experimental systems supporting orthogonal persistence have been constructed [4, 10, 20, 31]. These usually provide a large store within which concurrent processes may manipulate persistent data. In some of these systems, the stores contain all data including procedures, graphics objects, processes and their associated state. A common feature of persistent systems is that the persistent stores supporting them are both resilient and stable.

Stability is the ability of a system to be consistently checkpointed on a secure medium so that computation may resume from that point at some future time. If a system is resilient then it can safely resume computation after an unexpected system crash such as a power failure. It is possible to have stability without resilience. For example, with some programming systems, state may be preserved (in a file) using a "save" command but consistency may be lost in the event of a system crash. After the system has stabilised, all data is guaranteed to be consistent and reside on stable storage. Resiliency is usually achieved by making store updates non-destructive; this is generally implemented using logging [11] or some shadowing technique [26].

If persistent systems are to be anything other than research vehicles, they must be both stable and resilient. The persistent systems which exhibit these properties and that have been constructed to date, with a few exceptions [19, 33], have been constructed on top of traditional operating systems. Existing operating systems do not provide an ideal platform for the development of persistent systems. This is not surprising since this was never part of their design goals. Indeed, most operating systems have files as their only abstraction over long term memory.

Tanenbaum [38] has listed the four major components of an operating system as being memory management, file system, input-output and process management. The nature of these four components is different in persistent systems. In a persistent system, the functionality of the file system and memory management are replaced by the persistent store. In many operating systems, input-output is presented using the same abstractions as the file system; clearly this is not appropriate in a persistent environment. Some persistent systems require that the state of a process persists; this is not easily supported using conventional operating systems. It is therefore to be expected that an operating system designed to support persistence will have a different structure from a conventional operating system and will provide a different set of facilities.

From *Proceedings of the 25th Hawaii International Conference on System Sciences*, vol1, ed V. Milutinovic and B. D. Shriver, IEEE Computer Society Press, Hawaii, U. S. A., pp. 779-789, 1992.

We can summarise the principal requirements of such an operating system as follows:

i. The major requirement is support for persistent objects as the basic abstraction. Persistent objects consist of data and relationships with other persistent objects; the system must therefore provide a mechanism for supporting the creation and maintenance of these objects and relationships. This mechanism should be based upon a uniform addressing scheme used by all processes to access objects. That is, all processes share a single logical address space. This is essential for orthogonal persistence.

ii. A further requirement is that these objects must be both stable and resilient. The system must reliably manage the transition between long and short term memory transparently to the programmer.

iii. Processes must be integrated with the object space in such a way that process state is itself contained within persistent objects. The importance of this is that processes themselves become resilient.

iv. Although the persistent store is uniform, there is still a requirement to be able to restrict access to objects for the same reasons that file systems contain access control mechanisms. Any operating system supporting persistence must therefore provide some protection mechanism.

We term an operating system that provides these facilities a *persistent operating system*. The aim of this paper is to set the groundwork for the design of such an operating system. The major constraint that we wish to place upon ourselves is that the operating system should run on conventional architectures such as Sun workstations. Such a hardware configuration has many advantages as a platform:

- The performance of these systems is increasing dramatically every year due to the massive investment of the hardware vendors.
- These architectures are highly available. It is therefore easy to disseminate research results by providing copies of the system to interested parties.
- Should commercialisation become a possibility in the future, a totally software platform is easier to market than a solution including specialised hardware as has been found with the REKURSIV [22].

The effects of this constraint are that on most current architectures addresses are a maximum of 32 bits long, there is no hardware support for object protection and the only memory management hardware available is based on fixed sized pages.

In the following sections, we examine some existing persistent systems with respect to the four requirements listed above and discuss some of the approaches taken in supporting these facilities on top of conventional operating systems. This will provide a basis for the design of a persistent operating system. Finally we examine the problems associated with the transience of kernel structures and examine the consequences of making the kernel itself persistent.

2. Support for persistent objects and relationships

Conventional operating systems expect to support processes which only access short term data in directly addressable physical or virtual memory. Long term data is held on backing store and cannot be directly addressed. In contrast, systems which provide orthogonal persistence treat all data identically as persistent objects. This leads to a requirement that objects must be addressed uniformly and moved between long and short term storage in a manner that is transparent to the application programmer.

Several approaches have been taken to solve this problem. In Brown's stable store [15] two address spaces are managed: a local process address space in which objects may be directly accessed by machine instructions and a software supported persistent address space; objects are transparently moved from one to the other on demand. This requires software address translation between local address spaces and the persistent address space; this software address translation can never be made as efficient as hardware address translation. However, the impact of the cost of address translation in persistent systems is not clear due to the lack of sufficient measurement.

Another approach is to utilise paging mechanisms which are more efficient because they make use of hardware address translation. This approach, however, has two problems. Firstly, no operating system constructed to date provides sufficiently flexible mechanisms to exploit the hardware facilities to their full potential. Mach [9] and Chorus [36], for example, do provide considerable flexibility in managing virtual memory but, as we will show later, do not deliver all the required functionality. Secondly, addresses supported by the conventional hardware to which we have chosen to constrain ourselves are not large enough for extremely large stores. We do not intend to directly address the latter problem in this paper, instead the reader is referred to Rosenberg et al [34].

2.1. Software address translation

The first object systems to be called persistent [14, 18] did not rely on any support from the hardware or operating system other than the provision of a file system. In this section, for illustration purposes, we will concentrate on one of these persistent object management systems, the CPOMS [16]. The CPOMS is the persistent object management system used to support the Unix PS-algol [2] implementations.

The persistent store implemented by the CPOMS is a large heap with objects being addressed using persistent identifiers known as PIDs. Normal pointers are traditionally referred to as local object numbers or LONs.

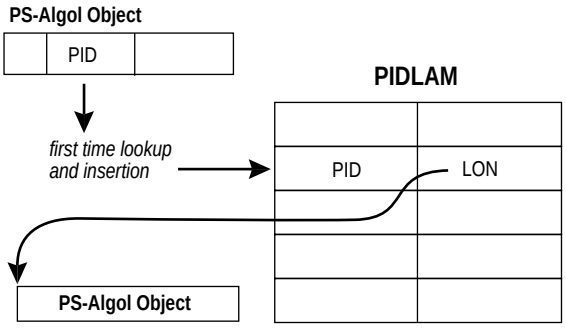

Figure 1: Looking up a PID in the PIDLAM

Since PIDs are pointers to objects outside of the program's address space, the objects to which they refer cannot be directly addressed by a PS-algol program. PIDs are identical in size to the normal pointers used by the PS-algol run time system but are distinguished by having their most significant bit set. Attempts to dereference PIDs are trapped and invoke the software translation mechanisms, whereby the object is fetched into memory and the PID is replaced by the appropriate LON; this activity has become known as *pointer swizzling*.

In order to prevent more than one copy being made of an object, a data structure called the PID to Local Address Map (PIDLAM) is kept. When a PID is first used and the object to which it refers is copied, the PID is entered into the PIDLAM along with the LON of the copy. Therefore, if the PID is used again, the LON of the copy can be found from the PIDLAM and used in its place as shown in Figures 1 and 2.

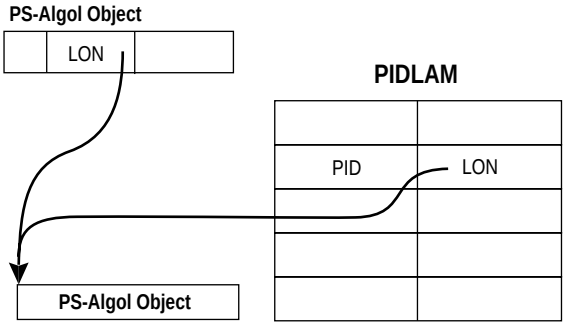

Figure 2: Overwriting a PID by a LON

The CPOMS address translation mechanism is relatively simple: a PID is divided into two parts: a segment identifier and an object number within that segment. Each PS-algol database is implemented using two Unix files: a data file and an index file. For each PS-algol database, the objects are stored in the data file which is composed of one or more segments. The index file maps database object numbers to addresses within the corresponding data file. Thus when a PID is encountered, the appropriate segment identifier is mapped to a particular database. Next, the database object number is calculated using the segment identifier and the object number within the PID. This number is used to address the database index file which yields the address of the desired object in the database data file.

Although relatively simple, this mechanism tends to be slow for two reasons: firstly all the address translation is performed in software, and secondly persistent object accesses involve several file seek and read operations. The second of these two problems may be addressed using memory mapping techniques as discussed in the following section.

2.2. Memory mapping techniques

2.2.1 Using SunOS memory mapped files

Support for persistence in the Napier88 system [28] has, to date, been provided using two different techniques [15, 16]. The first uses a much simplified CPOMS approach, the other uses SunOS memory mapped files. We will discuss this second implementation here.

The Stable Store which implements a resilient persistent address space as discussed in section 3.2.2 is maintained in a single, fixed length Unix file as shown in Figure 3. Using the SunOS system call *mmap*, the whole file is memory-mapped to a single virtual memory address range at an address *map_start*. The useable address space begins at *data_start* and extends to the end of the file. Reading a persistent address *x* constitutes accessing the contents of address *map_start* + *x* in the virtual memory of the executing process. The fetching of the appropriate page from disk is transparently handled by the operating system mapping mechanism.

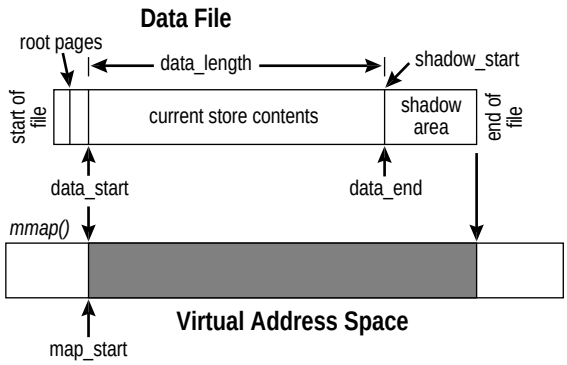

Figure 3: The memory mapped Napier88 store

This technique allows the store to be coded at a higher level by utilising the existing operating system support for virtual memory to abstract over disk access. This approach, however, gives less control over the time at which pages are actually written to disk.

2.2.2 Mach and Chorus

Mach [9] and Chorus [36] provide greater control over virtual memory in that both provide support for programmable page fault handling. In both systems, the user is permitted to provide a process which services page faults. This process is known as an *external pager* in Mach and as a *mapper* in Chorus. Since the Mach and Chorus communities use different terminology for what is essentially the same abstraction we will arbitrarily use the Mach terminology henceforth.

Both systems support an abstraction over memory called a *memory object*. Memory objects may be mapped into the virtual address space at any address. Both systems provide default mechanisms for the management of memory objects, for example a default pager which implements classical demand paging to a swap region. A user, however, may provide a manager for a memory object using an external pager. External pagers must conform to an interface specified by the kernel and are responsible for handling memory management requests from the kernel. For example, in response to an action by the swapper, the kernel may remove a page from physical memory and notify the external pager of its action. In this case the kernel will call the external pager interface function *memory_object_data_write* with suitable parameters. The external pager must deal with this page in some suitable way. The external pager interface provides enough functionality to allow page level access control over memory objects.

This mechanism may be exploited to implement stable stores. For example, at the University of Adelaide, the Mach external pager mechanism has been used to support a coherent distributed persistent address space [24, 41]. In this system, a centralised store may be accessed by client processes distributed around a local area network in a manner similar to the systems described by Li [25] and Henskens [23].

2.3. Conclusions

In the previous sections we have attempted to show a progression of persistent address translation techniques ranging from purely software approaches through to the utilisation of hardware made possible using systems such as Mach and Chorus. The trend in operating system support has been to make more of the functionality provided by the hardware available to user applications. Indeed, much of the contribution of operating systems like Mach and Chorus could be considered to be finding suitable abstractions over the base functionality provided by the hardware.

The software approaches tend to be slow but permit the implementation of address spaces which are larger than the virtual address space size supported by the underlying hardware. The need for large address spaces has lead some designers to adopt hybrid approaches which use both pointer swizzling and hardware address translation, for example [29, 42].

The memory mapping abstraction provided by SunOS permits regions of the file space to be mapped into virtual memory and accessed transparently. However, this mechanism was not designed to support large address spaces such as those found in persistent systems. This leads to problems, for example the SunOS memory mapped file implementation of Napier88 suffers from the problem of slow start up times due to the operating system's eager construction of large virtual address maps.

We have indicated how Mach and Chorus both provide external pager mechanisms which permit the implementation of an application specific pager. Once again, this mechanism was not designed to support large persistent address spaces. Both these systems are deficient in that they do not permit total control over the virtual address space. In particular, in both systems it is the kernel that selects pages to remove from the process' address space when physical address space becomes short. This functionality causes many problems for the persistent system implementor who may be using memory in complex ways unknown to the kernel.

The systems described in this section demonstrate that it is possible to support a persistent system using standard paged address translation hardware. Mach and Chorus have shown that it is possible to provide useful abstractions over this hardware and this is seen as their main contribution to this field.

The developers of persistent systems do not want to be burdened with the task of managing the complexity of external pagers. Instead, a persistent operating system should provide the higher level abstraction of a resilient persistent address space which can be implemented within the operating system directly utilising the address translation hardware.

3. Stability and resilience mechanisms

The requirement for resilience is not peculiar to persistent systems; the same problem occurs with conventional file systems. For example, most operating systems provide limited recovery features such as *fsck* in Unix. However, the problem is perhaps more acute with persistent systems. In a conventional file system each file is essentially an independent object. Therefore, the loss of a single file following a crash does not threaten the integrity of the overall system. In a persistent system there may be arbitrary cross references between objects and thus the loss of a single object can result in total system failure. In this sense the problem of recovery within a persistent store is much more closely related to recovery in database systems [11].

Early persistent stores such as POMS [18] and the CPOMS [16] were constructed using conventional file systems with no special features. Since the underlying file systems offered no explicit support for persistence, and for stability in particular, techniques similar to those developed for database systems were used. The persistent store was implemented as a series of databases against

which a program could apply transactions. As described above, such databases corresponded to individual files as provided by the operating system. A record of transactions was maintained in a transaction log, which allowed them to be either *commited*, thus changing the state of the database, or *rolled back*, thus returning the database to some previous stable state. To apply modifications to the database a commit operation had to be invoked by the program. The persistent store was thus partitioned in an unnatural way to provide correspondence with files, and details of this partitioning remained visible to the user.

The advent of memory mapped files allowed the use of virtual memory technology in accessing the filestore. This facilitated the use of *shadow paging* [26] as a means of ensuring store integrity. Many of the later systems are based on this technique [15, 35, 39, 40]. It is therefore instructive to examine this approach in order to establish the requirements of a persistent operating system.

3.1 Shadow paging

Resiliency requires that the persistent store evolves from one consistent state to another atomically. That is, in the event of a system failure, all the changes are either recorded or the system recovers to the previous stable state. A number of techniques have been developed for achieving stability, particularly in the context of database management systems [13, 16, 22, 26, 35, 39, 40]. The techniques differ in their efficiency with regard to the particular application area. However, there are two basic requirements. They are:

- the ability to perform an atomic update operation, and
- the ability to identify the old data and new data prior to the stabilise operation.

In order to explain how atomic update may be implemented we will assume the following:

i. There is a mapping table from virtual persistent store addresses to physical disk addresses (such an address map is required in systems where the virtual address space is not mapped in 1-to-1 correspondence with the physical address space). All the data in the system can be found using this mapping table.

ii. On system start up and after each stabilise operation a new copy of the mapping table and the data is made. Updates are made to these copies. That is, the old data is never overwritten. Such a system is unrealistic since the copy operation is too expensive but it will serve as a model for explaining atomic update. We examine some actual implementations and it will be seen that an efficient implementation is possible.

Prior to a stabilise operation there are two sets of mapping tables and data - the new updated one and the one representing the state of the system at the previous checkpoint. Challis' algorithm [17] uses two fixed blocks with known disk addresses that usually record the two previous stabilised states of the system. These are known as the root blocks. The root blocks contain information that allows the system to find the mapping table for a stabilised state. The root blocks record the two previous stabilised states.

Each root block also contains a version number that enables the system to determine which contains the most recent state. This version number is written twice as the first and last word of the block. The atomic update operation entails overwriting the root with the oldest version number, and a pointer to the new updated mapping table. The space occupied by the old stabilised state may now be reused.

Challis' algorithm depends upon two critical points for safety. Firstly an error in an atomic update can only occur if the root block is written incorrectly. It is expected that if a disk write operation fails during the atomic update it will inform the system which can then take appropriate action immediately. If, however, the failure is more serious, the technique depends upon the version numbers at the start and end of the root block being different in order to detect failure.

On system startup the root blocks are inspected. If the version numbers are consistent within the root blocks, the most up-to-date version of the system can be found. If not, only one root block may have different version numbers at any one time unless a catastrophic failure has occurred, which in any case would have other implications for the integrity of the data. Thus, subject to the above proviso, the correct stable data can be identified.

Assuming that an atomic update can be performed by this mechanism we return to the question of making efficient copies of the data. There are a number of different techniques that have been implemented and we now examine a selection of these.

3.2. Implementations using shadow paging

3.2.1 Thatte's recoverable virtual memory

Thatte [39] has proposed a *recoverable virtual memory* as the basis of a uniform memory abstraction for object-oriented databases. With Thatte's scheme, each page on the disk is in one of two forms, called *singleton* and *sibling*. Singleton form is used for pages unlikely to be modified and is represented by a single copy of the page on disk. In the sibling form two disk blocks are allocated to the page. When a page is written to disk a timestamp is written with the page, either in the page header or in the page table. The timestamps are derived from a reliable continuously running counter.

The essence of the scheme is that at any time a checkpoint operation may be initiated. This saves the current state of the system in a consistent manner. The time of the last checkpoint is recorded in a root page which is written in a secure manner. When a write fault

on a singleton page occurs, if the timestamp of the disk copy is earlier than the last checkpoint then it must be converted to sibling form and the modified page is written to a new disk block. Otherwise the singleton page is not part of the previous checkpoint and may be freely modified. A write fault for a sibling page results in the most recent of the two disk blocks which is still part of the checkpoint state being retained and the other page overwritten. In addition, at any time a sibling may be converted to a singleton (in order to conserve disk space) provided the timestamps of both disk blocks are before the last checkpoint. In this case the older of the two blocks is discarded. On a checkpoint, a transient root is created in the store. This contains the current state of the processor registers so that, following a failure, they may be restored as at the last checkpoint.

The above rules guarantee that for any page there will always be a copy of that page on disk as at the last checkpoint (if it existed). These checkpoint pages are not overwritten until a new checkpoint has been established. This new checkpoint will become the current checkpoint only when the root page is updated. This may be performed in an atomic operation using Challis' algorithm as described above. Following a system crash the system is rolled back to the checkpointed state by restoring the processor registers from the saved transient root object.

Thatte does not describe the operation of the page tables in such a system. It is not clear where they reside (in the store or outside of the store) or how free space is managed. This is a key issue, since following a crash we must ensure that no disk pages are lost. The only mention of the page tables is the suggestion that the timestamps may well be held in the page tables themselves. This would seem to be quite expensive since the timestamp is 64 bits and thus the page tables would be very large. In addition, for sibling pages two such timestamps (as well as two disk addresses) are required, further increasing the size of the tables.

The major difficulties with Thatte's scheme seem to be the size of the page tables and the extra disk space required to maintain the sibling pages. For siblings where the timestamp of both pages is before the last checkpoint, the older of the two pages is wasted space. As Thatte points out it is possible to recover this space by converting the pages to singletons but there is some cost associated with such a task.

3.2.2. Brown's stable store

The current implementation of Brown's store divides the disk storage into two regions as shown in Figure 3. The first contains the current version of each page and the second, called the shadow area, contains a copy of the original version of each page modified since the last checkpoint. On the first modification of any page following a checkpoint, a copy of the page is made on a disk.

The system does not take advantage of virtual memory page protection to detect the first write to a page, instead a call to the store interface function *can-modify* precedes any attempt to modify an object in the persistent virtual address space. If this is the first such update to the corresponding page since the store was last in a stable state, a copy of the page (called a *before look*) is made in the shadow region before permission to modify is granted. In addition, a data structure called the *written bitmap* is altered to record the fact that a shadow copy of this page currently exists, preventing multiple shadows. When the new version of the page is eventually written back to the store, it is written back to its original location. In this way, the new state of the store is built by overwriting the contents of the old, but the old state can be completely restored from the shadow copies should a failure occur between checkpoints.

A checkpoint takes place by copying every modified page back to disk and then clearing the shadow region. Following a crash the last checkpoint state can be re-instated by using the pages held in the shadow region and the written bitmap to restore all modified pages to their original states.

It should be noted that a failure to correctly call *can-modify* on each write may result in incorrect operation of the checkpoint mechanism. In this sense Brown's scheme can only be used with trusted systems.

The major difference between Brown's scheme and the scheme implemented by Thatte is that Brown maintains a before look and uses this to recover the data following a crash; the other scheme creates an *after look* leaving the original data unmodified. The major advantage of a before look is that the original order of the data is preserved, thus maintaining any existing locality within the data and potentially improving sequential access and access to very large structures. The cost is the overhead of copying pages before they are modified and the maintenance of the disk-based written bitmap. Similar clustering can be achieved using an after look technique as suggested by Lorie in his original paper.

3.2.3. Adelaide's coherent persistent address space

The system built at the University of Adelaide uses the Mach external pager. Objects in the store are addressed using virtual memory addresses, meaning that address translation hardware may be exploited. The system uses an *after look* shadow paging scheme to provide store stability, with the first modification of a page being detected by the address translation hardware.

In this architecture, a number of clients execute against a shared stable store using a coherency protocol that guarantees data integrity. The frequency of checkpoints in any one client is reduced by maintaining a record of those clients which must be considered dependent upon one another due to the fact that they share modified pages.

Only clients which are considered to be dependent on one another in this fashion need be stabilised together.

A central stable store server maintains information regarding the distribution and modification status of pages held by the clients; among this information is a record of which clients are dependent on each other. Interdependent clients are termed *associates* and a set of mutually dependent clients is called an *association*. Each association has a corresponding *page list*, which identifies those pages modified by members of the association since their previous stabilisation; this information is used to incrementally build the associations.

A mapping is maintained that maps the address of a virtual page to its location in stable memory (i.e., disk). This mapping table is called the *Logical to Physical map* (L-P map). As shown in Figure 4, each entry in the L-P map contains three fields: the physical page location of the stable version of the page, the location of the shadow copy of the page (if one exists) and a single bit selecting which entry holds the address of the stable page. Since the L-P map must be robust, it is stored within the persistent store which it manages.

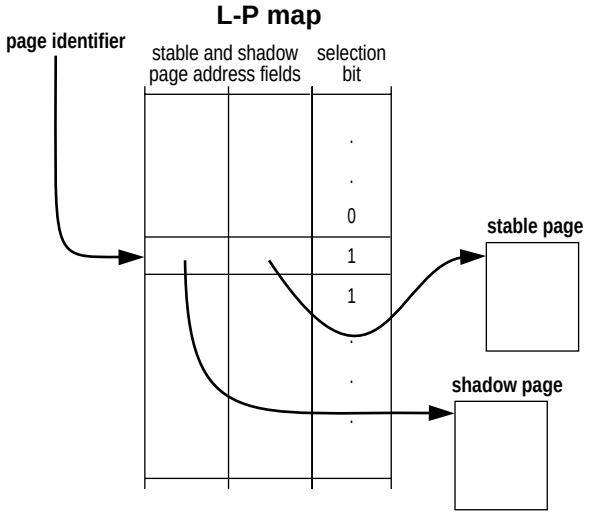

Figure 4: The L-P map.

The stabilisation protocol is as follows: the modified pages are first written from the persistent address space to their shadow location on disk. In the normal course of events, modified pages may also be delivered to the stable store by the coherency mechanism if there is insufficient space for them within a client's physical memory. These pages are also written to their shadow locations and are regarded as having been written back as part of this stabilisation. After all modified pages have been written to their shadow sites, the remainder of the stabilisation must be synchronised with any other stabilisations which have reached the same stage. The stabilising association's page list is then traversed (recall that this holds the list of all modified pages which are to be stabilised during the current stabilisation) and the selection bit is flipped for each of these pages to indicate that the shadow version is to be used as the current version once the stabilisation is complete. Next, the L-P map entries containing modified selection bits are written back to the appropriate shadow locations. Like the other systems described in this paper, in this system, the store is described by the contents of one of two header pages and Challis' algorithm is used to move from one stable state to another.

It is possible for several independent stabilisation operations to be in progress at any time since, by definition, a client can only ever belong to one association. Consequently, pages from more than one stabilisation may be written to the stable store concurrently. However, care must be taken to ensure that the stable store moves from one stable state to another in an atomic fashion. In practice, the final stages of stabilisation must therefore be serialised. In particular, the L-P map can only make one flip at a time.

3.3. Conclusions

The checkpoint operations used to move persistent stores from one stable state to the next have been a problem for their implementors. Checkpointing the entire store at once as required by Thatte and Brown's schemes may have a detrimental effect on system performance because no store accesses may occur during a stabilise operation. However, the duration of stabilise operations may be minimised by ensuring that they occur frequently and that few modifications have occurred since the previous checkpoint. Such a strategy also has implications on system performance because a stabilise operation involves disk accesses. Ideally it should be possible to stabilise parts of the store separately, while still allowing other parts of the store to be used. This approach has been described above using lists of dependent clients called associations. Desirable operating system features to support the partitioned checkpointing of distributed server persistent stores are unclear, and are the subject of current research.

Like the problems encountered in address translation, the majority of problems in using conventional operating systems as platforms to support stability and resilience revolve around the abstractions provided by the operating system. We have shown that memory mapped files provide convenient mechanisms for accessing objects within files. However, if the system is to be made resilient, objects cannot destructively overwrite their original versions on disk. Therefore the operating system mechanisms for flushing pages of a memory mapped file back to disk cannot be used. This results in much complexity being added to the code which supports resilience.

As we stated in section 2.3, a persistent operating system must provide a resilient persistent address space as a basic abstraction.

4. Process management and protection

Persistent systems provide a large persistent store in which all data resides. All processes execute against this store, meaning that it is necessary to ensure that a process may only access data for which it holds access permission. Failure to provide a protection mechanism could therefore result at best in the loss of privacy, and at worst in rogue or erroneous processes corrupting data owned by other users. Conventional systems such as Unix [30] implement two levels of protection of data; *file* level protection using three tiered control over read/write/execute access, and *process* level protection using control of access to process address spaces.

File level protection is of little use in a persistent system where all data is represented as objects directly addressable by processes. Therefore, all protection must be provided at the process level. Indeed, it is common to find data encapsulated within processes, modules, abstract data types and other language constructs in persistent languages. However these schemes rely on the security of trusted system components such as compilers.

Originally, operating systems for hardware platforms supporting paged virtual memory, provided heavyweight processes, each running in its own address space and associated with a particular user. Such processes communicate with each other using various IPC mechanisms ranging from stream based systems (pipes and sockets) to shared memory and signals. The heavyweight process paradigm clearly does not meet our requirements which state that the persistent address space should be shared by all processes.

More recent operating systems, have added support for lightweight processes called threads, which operate within a single shared address space. These are more appropriate to our model but are all associated with the same user and they share the same protection privileges. All threads have access to the entire address space and it is therefore possible for one thread to accidentally (or deliberately) corrupt the data of another. When a thread requests a new segment to be mapped into its address space it immediately becomes accessible by all other threads sharing that address space.

A second problem with the process model supported by existing operating systems is that, unlike the file system, there is no notion of persistence. The process state is maintained in transient memory and in the event of system shutdown or failure this state is lost. In order to implement any form of persistence the programmer must provide explicit code to checkpoint the state of the process. This problem of the non-persistence of processes extends itself to login/logout time and results in many operating systems having a proliferation of startup files (e.g. login.com, .login, autoexec.bat, etc.) which effectively rebuild the process environment each time the user logs on to the system. It should also be noted that they only rebuild a statically defined environment; they do not recreate the dynamic environment of the user at the time of the last logout.

The dichotomy between temporary and permanent data found in conventional operating systems also manifests itself in the synchronisation primitives provided. Two distinct sets of synchronisation mechanisms are usually provided. The mechanisms supported for transient data are low-level, providing little support for transactions, roll-back, and stability. Conversely, the synchronisation mechanisms which operate on permanent data are more sophisticated but are intimately bound to the file system model. Despite this sophistication there is usually little support for atomic update, making the development of alternative data models above the file system difficult.

4.1 Conclusions

We have examined the process models available in operating systems constructed to date. There appear to be two different problems associated with processes. The first of these is support for a single shared address space; this is solved by the light weight thread model. The second problem is support for protection at the process level; this is a more difficult problem and there are four different solutions:

i. The use of type secure languages and trusted system components such as compilers which strictly enforce protection rules. This restricts the class of systems which may be supported, for example persistent C++ systems could not be safely used.

ii. A second approach is to provide store level protection. For example, MONADS [32] provides protection based on *capabilities*. However, store-level protection requires some architectural support; this requirement is incompatible with our desire to implement a system on conventional hardware.

iii. Another approach is to implement multiple persistent address spaces. Adopting this approach gives a coarse grain of protection – processes either have total access to the address space or none at all. This may be acceptable where multiple independent persistent systems are desired. However, this approach is also incompatible with our requirements. It should be noted that with this approach it would be essential to have some global communication mechanism to allow processes operating in separate address spaces to cooperate. Such mechanisms have been described elsewhere [21].

iv. A final approach is to associate a page protection list with each thread; in such a scheme, the page protection map is changed on a context switch. In this manner, the threads would share a single

address space but may have their access restricted on a per page basis. This would result in performance penalties on machines with virtually addressed caches and would increase the overhead of a thread context switch. Although this mechanism does not provide access control at the object level, it does provide finer grain control over accesses than scheme *iii*.

It is apparent that none of these approaches fulfils the requirements stated in Section 1. In order to construct a persistent operating system on conventional hardware some compromises must be made. Approach *iii* permits the persistent operating system to support stores larger than the hardware supported address spaces. Approach *iv* has the advantage that it supports some level of protection within a persistent address space. A persistent operating system for stock hardware should support a combination of approaches *iii* and *iv*, permitting multiple address spaces with process based controlled access to pages.

5. Making the kernel persistent

Traditionally the operating system kernel is composed almost entirely of ephemeral data structures. These data structures have been regarded as structures for which it makes no sense to reason about their persistence. Kernel instances have been regarded as conceptually immortal and the bootstrap sequence has evolved to recover from those cases when this assumption is invalid. Most kernels maintain little or no state in stable storage; kernel and operating system subsystems are generally configured using static configuration files. Except by intervention of the system manager, these files do not change from one instance of the kernel to another and certainly do not change whilst the kernel is executing.

However, recent operating system developments such as network file systems have forced the operating systems to maintain persistent data structures. For example, the lock manager daemon, *lockd*, which is used to manage file locks under NFS [37], must keep a list of peers with which it must re-establish contact whenever it is activated. This is necessary so that the otherwise stateless NFS file system can recover state embodied in file locks. The benefits of persistence for applications programming are equally applicable to kernels. If the kernel is itself made persistent, then utilities like those mentioned above would require no special code in order to preserve their states between system invocations.

Traditionally, the kernel is the custodian of process state; this state resides in the saved register contents for a process and in other kernel structures such as scheduler queues, open file descriptors and network connection structures. The information is held within the kernel primarily for security reasons. If processes are to be persistent objects, it is essential that the execution state of a process reside within the persistent store. If the kernel is itself persistent, that is all the data structures maintained by the kernel are persistent, then the process state information may be held within the kernel and be persistent. Thus the provision of a persistent kernel has the advantage that its satisfies one of our requirements without compromising security.

Trap handling is one of the most basic functions provided by kernels and is pivotal to their operation. In conventional systems, traps are fielded by the kernel and subsequently delivered to user processes. Unix has provided such functionality with *signal*, VMS with *AST*'s, and Mach provides the ability to receive traps through a message from the kernel. However, none of the existing mechanisms provides any method by which reliable delivery may be achieved. Since stabilisation of a persistent space may take place asynchronously with respect to the execution of some user process, it is possible for a trap to be generated before stabilisation and yet be delivered after stabilisation. This creates a timing window in which the state information describing the trap only resides within the kernel. Therefore, this information would be lost in the event of a system failure unless stabilised with the total persistent system. With the provision of a persistent kernel this is not a problem.

6. Conclusions

Based on our experience in implementing persistent systems, we have demonstrated that current operating systems do not provide an appropriate platform for building persistent systems. This is not surprising since they are based on an abstraction which is completely foreign to the ideals of orthogonal persistence. This naturally leads to the conclusion that an operating system expressly designed to support orthogonal persistence is required.

Using current operating systems, the implementor of a persistent system must manage the address translation tasks. Systems constructed to date demonstrate that it is feasible to construct a persistent system on conventional architectures. A persistent operating system will provide an abstraction consistent with our requirements as a fundamental building block.

Intrinsically a persistent operating system would be capable of providing all the functionality of traditional operating systems. The research issues are those same issues which compromise the implementation of persistent systems when conventional operating systems are used as a platform, namely: addressing, resilience, process management and protection.

One of the requirements is that a persistent store should be both stable and resilient. Again such stores have been successfully constructed on conventional operating systems but only with considerable difficulty and some loss of efficiency due to lack of control over memory management. A persistent operating system must support intrinsically resilient persistent address spaces.

Our requirements for processes are largely met by the lightweight process model. In order to support some persistent languages, it is necessary to provide persistent processes; this can be achieved by maintaining all process states within the store itself.

We have identified four mechanisms which may be used to enforce protection in persistent systems. Two of these appear to be unacceptable because they are either overly-restrictive or require specialised hardware. None of these schemes fully satisfies our requirements. However, a combination of the other two schemes suggests a model combining multiple persistent address spaces with page level protection.

We have suggested that the benefits afforded by persistence are as equally beneficial to the kernel as to application software. In fact it may be observed that making the kernel itself persistent has benefits to the implementation of the overall operating system. The construction of such a reflective kernel raises interesting research issues in itself.

It is only by the careful examination and understanding of the requirements of persistent systems that it is possible to define the interface to a persistent operating system. We view this paper as a preliminary and essential step towards the construction of a persistent operating system.

Acknowledgments

We would like to thank Tracy Lo Basso for her useful comments on this paper. The persistence people at St Andrews must also be thanked for their savaging of earlier versions of this paper.

References

1. "Proceedings of the International Workshop on Database Programming Languages", Roskoff, France, 1987.
2. "The PS-algol reference manual", University of Glasgow and University of St Andrews, PPRR-12-87, 1987.
3. "Datatypes and Persistence", *Proceedings of Data Types and Persistence Workshop Aug. 1985*, Appin, Scotland, ed M. P. Atkinson, P. Bunerman and R. Morrison, Springer-Verlag, 1988.
4. "PS-algol Reference Manual - fourth edition", University of Glasgow and St Andrews, Persistent Programming Research Report 12/88, 1988.
5. "Persistent Object Systems", *Proceedings of the 3rd International Workshop on Persistent Object Systems*, Newcastle, Australia, ed J. Rosenberg and D. M. Koch, Springer-Verlag, 1989.
6. "Proceedings of the International Workshop on Database Programming Languages", Salishan, U.S.A., Morgan Kaufmann, 1989.
7. "Proceedings of the 4th International Conference on Persistent Object Systems", Martha's Vineyard, U.S.A., ed A. Dearle, G. Shaw and S. Zdonik, Morgan-Kauffman, 1990.
8. "Security and Persistence", *Proceedings of the International Workshop on Architectures to Support Security and Persistence of Information*, Bremen, Germany, ed J. Rosenberg and J. L. Keedy, Springer-Verlag, 1990.
9. Acceta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A. and Young, M. "Mach: A New Kernel Foundation for Unix Development", *Proceedings, Summer Usenix Conference*, USENIX, pp. 93-112, 1986.
10. Albano, A., Cardelli, L. and Orsini, R. "Galileo: A Strongly Typed, Interactive Conceptual Language", *ACM Transactions on Database Systems*, 10(2), pp. 230-260, 1985.
11. Astrahan, M. M. "System R: Relational Approach to Database Management", *ACM Transactions on Database Systems*, 1(2), pp. 97-137, 1976.
12. Atkinson, M. P., Bailey, P., Chisholm, K. J., Cockshott, W. P. and Morrison, R. "An Approach to Persistent Programming", *The Computer Journal*, 26, 4, Nov., pp. 360-365, 1983.
13. Atkinson, M. P., Bailey, P. J., Cockshott, W. P., Chisholm, K. J. and Morrison, R. "POMS: A Persistent Object Management System", *Software Practice and Experience*, 14(1), pp. 49-71, 1984.
14. Atkinson, M. P., Chisholm, K. J. and Cockshott, W. P. "CMS - A Chunk Management System", *Software Practice and Experience*, 13(3), pp. 259-272, 1983.
15. Brown, A. L. "Persistent Object Stores", Universities of St. Andrews and Glasgow, Persistent Programming Report 71, 1989.
16. Brown, A. L. and Cockshott, W. P. "The CPOMS Persistent Object Management System", Universities of Glasgow and St Andrews, PPRR-13, 1985.
17. Challis, M. F. "Database Consistency and Integrity in a Multi-user Environment", *Databases: Improving Useability and Responsiveness*, ed B. Scheiderman, Academic Press, pp. 245-270, 1978.
18. Cockshott, W. P., Atkinson, M. P., Chisholm, K. J., Bailey, P. J. and Morrison, R. "POMS: A Persistent Object Management System", *Software Practice and Experience*, 14(1), 1984.
19. Dasgupta, P., LeBlanc, R. J. and Appelbe, W. F. "The Clouds Distributed Operating System", *Proceedings, 8th International Conference on Distributed Computing Systems*, 1988.

20. Dearle, A., Connor, R. C. H., Brown, A. L. and Morrison, R. "Napier88 - A Database Programming Language?", *Proceedings Second International Workshop on Database Programming Languages*, Portland, Oregan, Morgan Kaufmann, pp. 179-195, 1989.

21. Dearle, A., Rosenberg, J. and Vaughan, F. "A Remote Execution Mechanism for Distributed Homogeneous Stable Stores", *Third International Workshop on Database Programming Languages*, Nafplion, Greece, Morgan Kauffman (to appear), 1991.

22. Harland, D. M. "REKURSIV: Object-oriented Computer Architecture", Ellis-Horwood Limited, 1988.

23. Henskens, F. A., Rosenberg, J. and Keedy, J. L. "A Capability-based Distributed Shared Memory", *Proceedings of the 14th Australian Computer Science Conference*, Sydney, Australia, pp. 29.1-29.12, 1991.

24. Koch, B., Schunke, T., Dearle, A., Vaughan, F., Marlin, C., Fazakerley, R. and Barter, C. "Cache Coherence and Storage Management in a Persistent Object System", *Proceedings, The Fourth International Workshop on Persistent Object Systems*, Martha's Vineyard, Massachusetts, U.S.A., pp. 99-109, 1990.

25. Li, K. "Shared Virtual Memory on Loosely Coupled Multiprocessors", Ph.D. Thesis, Yale University, 1986.

26. Lorie, R. A. "Physical Integrity in a Large Segmented Database", *ACM Transactions on Database Systems*, 2,1, pp. 91-104, 1977.

27. McLellan, P. and Chisholm, K. "Implementation of a POMS: Shadow Paging via VAX VMS Memory Mapping", Unpublished Report, 1982.

28. Morrison, R., Brown, A. L., Conner, R. C. H. and Dearle, A. "Napier88 Reference Manual", Universities of Glasgow and St. Andrews, Persistent Programming Research Report PPRR-77-89, 1989.

29. Pose, R. D. "Capability Based, Tightly Coupled Multiprocessor Hardware to Support a Persistent Global Virtual Memory", *Proceedings of the 22nd Annual Hawaii International Conference on System Sciences*, Hawaii, U.S.A., ed B. D. Shriver, pp. 36-45, 1989.

30. Ritchie, D. M. and Thompson, K. "The UNIX Time-Sharing System", *The Bell System Technical Journal*, 63(6), pp. 1905-1930, 1978.

31. Rosenberg, J. "The MONADS Architecture - A Layered View", *Proceedings of the 4th International Workshop on Persistent Object Systems*, Martha's Vineyard, U.S.A., Morgan-Kaufmann, 1990.

32. Rosenberg, J. and Abramson, D. A. "MONADS-PC: A Capability Based Workstation to Support Software Engineering", *Proc, 18th Hawaii International Conference on System Sciences*, pp. 515-522, 1985.

33. Rosenberg, J. and Keedy, J. L. "Object Management and Addressing in the MONADS Architecture", *Proceedings of the International Workshop on Persistent Object Systems*, Appin, Scotland, 1987.

34. Rosenberg, J., Koch, D. M. and Keedy, J. L. "A Massive Memory Supercomputer", *Proc. 22nd Hawaii International Conference on System Sciences*, vol 1, pp. 338-345, 1989.

35. Ross, D. M. "Virtual Files: A Framework for Experimental Design", University of Edinburgh, CST-26-83, 1983.

36. Rozier, M., Abrossimov, V., Armand, F., Boule, I., Gien, M., Guillemont, M., Herrmann, F., Kaiser, C., Langlois, S., Leonard, P. and Neuhauser, W. "CHORUS Distributed Operating Systems", *Computing Systems*, 1(4), pp. 305-367, 1988.

37. Sun Microsystems Inc. "Systems and Networks Administration", Part No: 800-1733-10, Revision A, 1988.

38. Tanenbaum, A. S. "Operating Systems: Design and Implementation", *International Editions*, Prentice Hall, 0-13-637331-3, 1987.

39. Thatte, S. M. "Persistent Memory: A Storage Architecture for Object Oriented Database Systems", *Proceedings of the ACM/IEEE International Workshop on Object-Oriented Database Systems*, Pacific Grove, California, pp. 148-159, 1986.

40. Traiger, I. L. "Virtual Memory Management for Database Systems", *Operating Systems Review*, 16(4), pp. 26-48, 1982.

41. Vaughan, F., Schunke, T., Koch, B., Dearle, A., Marlin, C. and Barter, C. "A Persistent Distributed Architecture Supported by the Mach Operating System", *Proceedings of the 1st USENIX Conference on the Mach Operating System*, Burlington, Vermont, pp. 123-140, 1990.

42. Wilson, P. "Pointer Swizzling at Page Fault Time: Efficiently Supporting Huge Address Spaces on Standard Hardware", *ACM Computer Architecture News*, June, pp. 6-13, 1991.