Persistence in the Grasshopper Kernel

[†]Anders Lindström, [†]Rex di Bona, [*]Alan Dearle, [†]Stephen Norris,
[†]John Rosenberg, [*]Francis Vaughan

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

{anders, rex, srn, johnr}@cs.su.oz.au

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

{al, francis}@cs.adelaide.edu.au

Abstract

The Grasshopper operating system provides explicit support for orthogonal persistence. A consequence of this is that the kernel itself must, in part, be persistent. To conform to the model of persistence in Grasshopper, the kernel persistent store must provide a means to stabilise entities independently of each other and must also be able to maintain an arbitrary number of versions for each entity. The design of the kernel persistent store is constrained by the need to be very efficient and to intrude as little as possible on the code using the store. Entities in the store reside at fixed, unique virtual addresses by which they are identified. This allows standard demand paging techniques are used making the store efficient and unobtrusive. Rather than provide for the independent stabilisation of individual data structures, the store provides regions, which are variable-size sets of possibly noncontiguous virtual pages, that may be stabilised independent of each other and that may have many versions. These regions, called persistent arenas, are used by higher-level software as pools for the allocation of smaller data structures that must logically be stabilised together.

1 Introduction

The Grasshopper operating system is explicitly designed to support orthogonal persistence. It unifies the traditional abstractions of virtual memory and files to provide a single abstraction of data. This approach is motivated by the observation that data should be accessible in a uniform manner regardless of its creator, longevity or type[3]. The chief advantage of orthogonal persistence is that it significantly eases the programmer's task when sharing of arbitrary data structures between different programs (or different invocations of the same program) is required. An important implication of this is that failure should be as transparent to users as possible. To be truly persistent, an operating system must provide the following[11]:

- persistent objects as a basic user-level abstraction
- resilience of persistent objects – object state must survive system crashes
- persistent processes – running programs need not be explicitly restarted after a crash; they should resume from a previous state
- an object protection mechanism

In Grasshopper, many aspects of persistence are implemented by user-level software. Nevertheless, there is a small amount of data that must remain in the kernel in order to limit the damage caused by malicious or malfunctioning programs. For example, protection information, the queue of running computations and various memory management structures must all reside in the kernel. Given the goals listed above it is therefore necessary that the kernel itself, at least in part, be persistent. Conventional operating systems also require some amount of persistent kernel data; for example, meta-data for the file system. By contrast, persistence pervades *all* aspects of the Grasshopper system, requiring that the kernel's persistent store be sufficiently general to accommodate many object types and sufficiently efficient to avoid having an adverse effect on performance.

This paper describes the design and implementation of the kernel persistent store in Grasshopper. In particular it focuses on how the goals of high efficiency, generality and ease of programming have guided the design process. In addition, it shows that having knowledge of the persistent data structures that will be used leads to simplifications unavailable in a general-purpose system.

2 Overview of Grasshopper

Grasshopper provides three basic abstractions: *containers*, *loci*, and *capabilities*. Containers are the single abstraction of data storage and access. They are passive, persistent entities that provide a means for storing and accessing data. There is no inherent need to limit the size of containers; in particular, they may be larger than the virtual address space of the underlying hardware. For practical reasons though, the current implementation limits containers to 2^{64} bytes.

Loci (from locus of execution) are an abstraction of sequential execution. Each locus is basically a set of registers and some other system-related data such as priority and resource usage. A locus' addressing environment is defined by its *host container*, which provides both the code and data that it needs to carry out computation.

Grasshopper is a *procedure-oriented*[15] or *object-thread-based* system[7]. This means that a locus need not be tied to its host container for its entire lifetime. Instead, it may move to another container by *invoking* it. Invoking a container is very similar to calling a procedure; parameters may be passed and the locus resumes from the point immediately after the invocation when it returns. The point at which a locus begins execution after invoking a container is determined by the container's *invocation point*[1]. Invocation differs from calling a procedure in that the invoked container becomes the locus' host container and so provides a new addressing environment. This means that the locus can no longer access the contents of the container from which the invocation occurred. To track invocations, the kernel maintains a call-chain, analogous to a stack in Algol-like languages. It is comprised of *invocation records* that are used to pass parameters to the invokee and restore the locus state when it returns.

The invocation mechanism is extremely important in Grasshopper since it can be used to provide hardware-protected abstract data types(ADTs). Each instance of an ADT is implemented as a single container whose operations are accessed by invocation. This has the important implication that the kernel itself can be presented as a number of ADT instances whose services are accessed by invocation. This means that kernel-level and user-level services are totally indistinguishable. This approach has also been adopted in the Monads system[18].

Capabilities are the sole means of referencing and protecting entities in Grasshopper[9]. Whenever the code executed by a locus calls for the manipulation of another object, for example the invocation of a container, a capability must be presented. Grasshopper uses segregated capabilities – they are system-protected entities that may only be accessed through system calls. In contrast to password capability systems[2, 17], systems using segregated capabilities can keep a record of extant references to a particular entity, thus allowing deletion of unreferenced entities and any kernel data structures associated with them.

[1] *Each container can have only one invocation point; if it offers many services, one of the invocation parameters can be used to choose between them.*

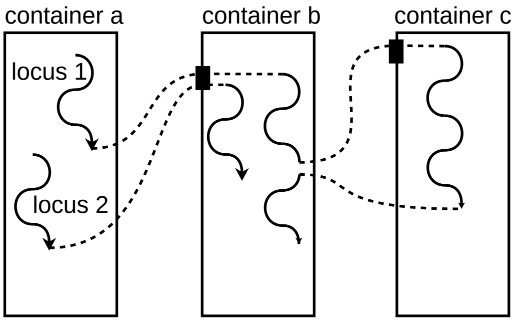

Figure 1. An example of loci and containers

Figure 1 illustrates an example of the basic abstractions. There are three containers, whose invocation points are represented by black squares, and two loci. Locus 1 starts executing in container *a*. Subsequently, it invokes *b*, after presenting the appropriate capability, and moves to *c* prior to returning to *b*. Concurrently, locus 2 starts in container *a* and then invokes *b*. Note that the loci are not confined to a single container and that many loci may execute concurrently in each container.

In addition to the three basic abstractions, Grasshopper provides a facility, called *mapping*, to share data and code extremely efficiently by allowing parts of one container to become directly accessible in another container. Mapping allows, for example, instances of abstract data types to share code rather than having their own separate copy.

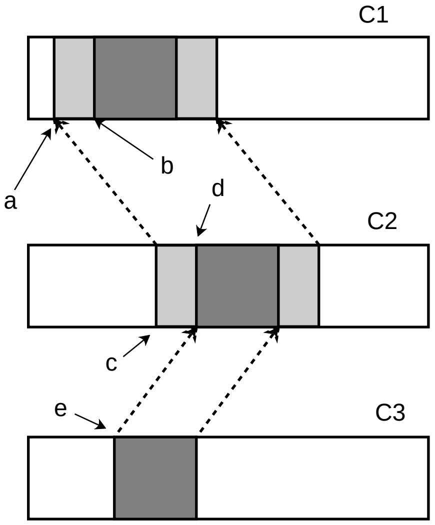

Figure 2. Mappings are not restricted to one level. Reading *b* in C1 returns the contents of *e* in C3.

Mapping in Grasshopper differs from similar features in other systems such as Mach[1] and Chorus[19] in a number of important ways. First, mappings may overlap each other – the last mapping made at a particular address overrides any others. Second, mappings are not restricted to one level. For example, in Figure 2 address *b* in C1 corresponds to address *d* in C2, which corresponds to address *e* in C3. This means that reading *b* in C1 will in fact return the contents of *e* in C3. Third, in addition to the usual case where a mapping is perceived by all loci in a container, a mapping may be made private to one particular locus. Such mappings are called *locus-private mappings* and take precedence over any global mappings. This allows, for example, each locus in a container to have its stack occupying the same address range as the stacks of other loci in that container. This technique both simplifies multi-threaded programming and provides a useful security mechanism unavailable in conventional systems.

3 Data Management

The implementation of containers includes the following tasks:

- the management of physical memory allocated to containers
- the management of the virtual address space within a container
- the management of stable storage (disks etc.)
- the implementation of stability algorithms

Not all these tasks are performed by the kernel. Rather, the kernel provides mechanisms by which user-level entities, called *container managers*, can implement any policy they choose. A container manager may be responsible for any number of containers and may coexist with other container managers. The advantage of this approach is that it provides a flexible system structuring facility, enabling research to be conducted with minimum impact to the rest of the system. In addition, it removes much complexity from the kernel making it both easier to write and maintain.

A container manager is implemented as a user-level container that presents a common system-defined interface[10]. When the kernel requires some aspect of data management to be performed on a container it invokes that container's manager. For example, if a page fault occurs in a container, the kernel invokes its manager with parameters that include the address of the fault. It is the manager's responsibility to service this fault in an appropriate way; for example, by retrieving the page from disk. This facility was inspired by experience with the Mach external pager facility[21, 22] but extends the model to include responsibility for stability and resilience.

4 The Model of Persistence

One of the perceived problems with persistent systems is that they incur an unreasonable overhead due to checkpointing. The reason for this is that stabilisation in most persistent systems involves suspending all current activity while modifications are saved on stable storage. This approach is sometimes called the 'stop the world' method. The advantage of such an approach, used by many single-user systems such as Napier88[8], is that it is relatively simple and automatically ensures a consistent recoverable state.

If the 'stop the world' method was used in Grasshopper, stabilising a single container would result in all loci being suspended, including those unrelated to the container. Clearly, this would lead to unacceptable response. The alternative is to stabilise containers independently. If this scheme is used, then after a crash it is generally *not* the case that the most recent stable states of all the containers form a consistent global state. Therefore, it is the system's responsibility to find such a state, a so-called *consistent cut*[5], from among the various checkpoints. There are a number of algorithms for achieving this [12, 13, 14, 20] which differ in the degree of asynchrony and the amount of recovery time and meta information required.

While this paper does not discuss the details of the algorithm used by Grasshopper, it is important to note three salient features. First, each entity may have an *arbitrary* number of stable states created independently from other entities. This is not to say that each container *must* have an arbitrary number of stable states, but the more there are the easier it is, in general, to find a recent consistent cut. Second, recovery involves finding the most recent *consistent* global state from among the set of entities. In general, this state will not include the most recent stable states of all, or even any, of the individual entities. This means that previous versions of entities need to be reestablished before the system can continue. This process is called *rolling back* an entity. Third, an important property of the algorithm is that stable states of individual entities which occurred before a consistent global state are guaranteed to be never used again. This means that the resources they use, such as disk space, can be reclaimed.

5 The Need for a Kernel Persistent Store

In common with most multi-user systems, Grasshopper prohibits user-level programs from indiscriminately tampering with the state of other programs and data. This means that some vital data structures must be held in the kernel. This section identifies these data structures and shows that, to achieve the goal of orthogonal persistence, they must themselves be persistent.

The state of a container can be divided into two parts: user-level state and kernel-level state. User-level state comprises any data or meta-data that cannot be used to gain unauthorised access to other entities. This includes the actual data in the container and any meta-information that the container's manager may keep in order to provide and maintain that data. The kernel-level state includes the data structures representing both mappings into the container and the capabilities associated with it. These data structures must be kept in the kernel, otherwise it would be possible for user-level programs to manufacture capabilities for arbitrary entities or to perform mappings from other containers, so gaining direct access to their contents. The state information of a locus also resides in the kernel. If it did not, user-level programs would be able to affect other loci in an uncontrolled way. The state of a locus consists of the following:

- current host container
- an invocation call-chain
- locus private mappings
- capabilities associated with the locus

Even though loci are not bound to remain in any particular container, each invocation record of the locus' call chain is inextricably linked to a certain container, as are the locus-private mappings. Thus, rather than thinking of the state of a locus as being totally separate from containers, it is better to think of it as being *distributed* among the containers in its call-chain.

Clearly, given the goals of Grasshopper, these kernel-level data structures must be persistent. If this were not the case, user-level code would perceive differences before and after a crash, a situation that is the antithesis of orthogonal persistence. For example, if mappings between containers were not persistent, user-level code would have to redo the mapping after a crash in order to be consistent. Further, if loci were not persistent, user-level code would have to decide from where to resume computation after a crash, a task that, at best, would significantly complicate the code and, at worst, may not even be possible.

6 The Kernel Persistent Store

6.1 Design Issues

From the above discussion, it is clear that there is a need for a persistent store in which to hold kernel data. To conform to the model of persistence in Grasshopper, described in section 4, the store must fulfill two requirements: it should be able to stabilise the kernel-state of an entity *independently* from that of other entities and it should be able to maintain an *arbitrary number* of stable states for each entity. Secondary goals are that it should be efficient, since it has the ability to affect the performance of the entire system, and that it should be not intrude on code that uses it; writing an operating system kernel is already difficult enough.

Keeping these goals in mind, the design of the store must address the following questions:

1. How are objects in the store identified?
2. What is the object loading strategy?
3. What is the granularity of objects?

There are two possible answers to the first question: software identifiers or virtual addresses. In the first approach, used in systems such as POMS[6], entities are uniquely identified by some software naming scheme that is independent of the virtual address space. This means that the data space can be of arbitrary size for which the virtual address space serves as a cache. The main problem with this approach is that software identifiers, by definition, cannot simply be presented to the hardware to access an object. Instead, software identifiers must be translated by a layer of software. Most systems that use this approach have a language built on top of them that makes this translation transparent to programmers. The Grasshopper kernel is implemented in C which does not provide the necessary language support. Therefore, a function or macro would need to be called to perform the translation on every access to an object. This violates the secondary goal of not intruding on the code.

In the second approach, which is adopted by Grasshopper, each object resides in a fixed and unique region of the kernel's address space until it is destroyed and can therefore be identified by its virtual address. The main advantage of this approach is that virtual addresses can be presented directly to the hardware without the need for software translation. The disadvantage is that the size of the store is limited by the virtual address range of the underlying hardware. We do not expect this to be a problem since the store only contains *kernel* data; user-level data is kept in containers which are of arbitrary size.

The second question also has two possible answers: explicit loading or implicit loading. An explicit loading scheme is one in which the store provides a command to retrieve an object. This method is clumsy without language support. It is also inefficient since a check must be made on every access to see if the object is in physical memory. An implicit loading scheme uses memory management hardware to detect accesses to non-resident pages and does not intrude on code using the store.

The disadvantage of using implicit loading is that it implies that stabilisation must be performed on page-sized units since memory management hardware can only detect modification of pages rather than individual entities. In other systems this is not a problem since the entire store is stabilised simultaneously, making page boundaries irrelevant. The Grasshopper kernel store, by contrast, is required to support independent stabilisation of entities requiring it to place them on separate pages.

Placing individual data structures on separate pages would lead to unacceptable levels of internal fragmentation – the typical data structure in the kernel is just a C structure that contains a small number of scalars and pointers. While at first this seems to be an intractable problem, knowledge of the relationships between these data structures allows us to make the key observation that it is not necessary or even sensible to independently stabilise *individual* data structures. Rather, data structures form logical groups and it is these *groups* that should be independently stabilised. For example, the state of a container includes the capabilities and mappings associated with it, the data structures for which are comprised of many smaller parts, for example, elements of a linked list. Stabilising a container then, involves saving a whole group of smaller data structures, called a *stability group,* which may be clustered together rather than lie on separate pages.

If stability groups are sufficiently large, internal fragmentation within pages can be alleviated making an implicit loading scheme feasible. This conclusion, together with the fact that explicit loading schemes are intrusive and inefficient, convinced us that an implicit loading scheme based on virtual addresses was the best choice for the kernel persistent store.

6.2 Persistent Arenas

Rather than support the persistence of individual data structures, the kernel persistent store provides for the creation of medium to large-size regions that may have any number of versions created independently of each other. These regions are called *persistent arenas* and are used by higher levels of software as pools for the allocation of individual data structures in a stability group. For example, each container has its own persistent arena from which the many small data structures that comprise the kernel's representation of that container are allocated. Stabilising the container's arena results in the atomic stabilisation of all these data structures.

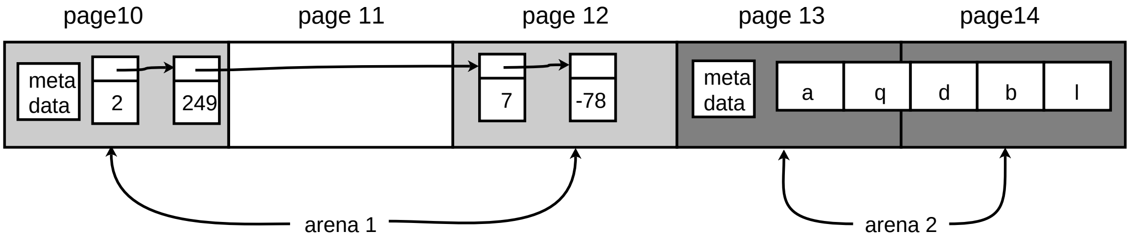

Figure 3. Two arenas.

Figure 3 shows an example of two arenas. The figure illustrates two important aspects of persistent arenas. First, the pages comprising the arena do not need to be contiguous in virtual memory. This is because, in general, individual data structures within stability groups are smaller than a page; for example, the list elements in arena 1. Of course, if a particular data structure is larger than a page, contiguous virtual pages will be needed. The array in arena 2 in figure 3 is such a data structure. In practice though, the occurrence of such large data structures in the kernel is rare. Allowing non-contiguous pages makes increasing the size of arenas fairly easy since any free virtual page can be used. This has the additional benefit of minimising external fragmentation of the virtual address space. The second point to note from figure 3 is that persistent arenas are self-describing; that is, all meta-data describing a persistent arena is embedded in the arena itself. This approach simplifies the provision of multiple, independent versions since stabilising the data of a version automatically saves the meta-data for that version independently of other arenas and other version of the same arena.

All versions of an arena have a distinguished page, the *root page*, that contains enough meta information to reconstruct the whole arena. Figure 4 shows the structure of the root page. The first two parts of the meta-data, the root page flag and the number of pages in a run, are used for recovery and are explained in section 6.4. The rest of the meta-data includes the head of the *arena page table*, a version number and the disk address of the root page of the previous version of the arena. Any space not used by meta data can be used for data. It is therefore possible for small arenas to fit entirely on a single page.

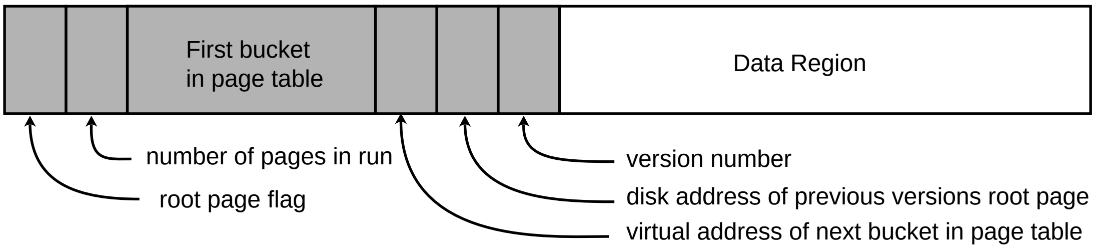
First bucket in page table
Data Region
number of pages in run
root page flag
version number
disk address of previous versions root page
virtual address of next bucket in page table

Figure 4. Meta-data in an arena's root page

An arena's page table serves two purposes. First, it records which virtual pages belong to a version of the arena. Second, it records which disk pages are used to store the data of these pages. Since arenas are of arbitrary size, the page table may not entirely fit into the root page. Consequently, the page table is implemented as a series of fixed sized buckets which reside on different pages of the arena. Each bucket contains a pointer to the next one so that, by traversing through the list of buckets, all page table entries can be found.

Having to linearly search the list of buckets initially appears to be inefficient but in practice an arena rarely grows to a size that warrants more that one bucket. In addition, the page table is usually searched in response to a page fault on a non-resident page. The cost of traversing the list of buckets is therefore completely dominated by the cost of actually retrieving the data from disk. This of course ignores the fact that traversing the list may itself cause page-faults but this would be true of any scheme that allows for arbitrary sized arenas since some of the page table will eventually spill over to other pages.

In addition to the first bucket of the page table, the root page contains a version number for the arena. This number serves as an ordinal between different versions of the arena and is used by the recovery algorithm described below. The root page also contains the disk address of the root page of the previous version of the arena. Thus, the root pages of the different versions of an arena form a list of disk pages that may be traversed to find any one of those versions. Again, this facility is used by the recovery algorithm discussed in section 6.4.

The process of stabilising a container includes stabilising its kernel-level state. This involves identifying all pages in the container's arena that have been modified since the last stabilisation and saving them to *new* disk pages so that previous versions are not lost. Page modification is detected by marking all pages in an arena read only – modification of a page will be signalled by an access violation. Since this is handled in the kernel, the overhead is very low. The latency of saving modified pages to disk, on the other hand, has the potential to be very high. Therefore, a lazy technique based on [16] is employed. Briefly, modified pages are marked read-only while they are queued to be saved to disk. This allows reads of the data to proceed before the disk writes complete. If a page is modified, an exception is raised. At this time the page is copied to a new physical page frame before allowing the modification to complete.

6.3 The Active Region Descriptor

The kernel address space is divided into two regions: the transient region and the persistent region. The transient region is used for data structures that do not have to survive system shutdown such as I/O buffers, the list of free physical pages etc. The pages in the persistent region are allocated to persistent arenas as they are needed. The persistent region is further divided into two subregions: the active region and the inactive region. The active region consists of all pages that are either currently allocated to arenas or have been allocated and subsequently freed. All pages in the inactive region are free.

virtual page number	0	1	2	3	4	5	6	7	8
type	R	F	D	F	D	R	D	F	D
page number	56	7	0	-1	5	289	5	3	5

Figure 5. The Active Region Descriptor

The kernel persistent store maintains a data structure, the active region descriptor (ARD), that has an entry for every page in the active region. An entry is 8 bytes long and has two fields: the type field and the page number field. The type field indicates whether the page is a root page, a data page or a free page. The page number field of a root page entry contains the *disk* page number used to store the root page of the current version of an arena. For example, in figure 5, pages 0 and 5 are root pages and are saved on disk pages 56 and 289 respectively. The page number field of a data page is the *virtual* page number of the root page of the arena to which the page belongs. For instance, page 2 belongs to the arena whose root page is page 0 while pages 4, 6 and 8 belong to the arena whose root page is page 5. The page number field of a free page is the virtual page number of the next page that is free in the active region. Thus, the free page entries form a list that is used for allocation. For efficiency, the ARD is implemented as a dense array – an entry is found simply by using its virtual page number to index directly into the array.

Having described the ARD and the meta-data embedded in arenas, the algorithm used to handle page faults can now be described:

```
On a page fault at va:

if va is in inactive region then
        an error has occurred so raise an exception
entry = ARD[page number of va]
```

```
case entry.type of
FreePage:
    error
RootPage:
    read the disk page indicated by entry.page_number into memory
    update memory management hardware
DataPage:
    use entry.page_number to find the virtual address of the appropriate root page
    find the disk address of the data page using the arena's page table
    read the disk page into memory
    update memory management hardware
end case
```

When looking up the page table the list of buckets may need to be traversed. This may cause page faults on the pages containing these buckets. So that these page faults may be correctly serviced, each page that contains a page table bucket must have an entry in a previous bucket in the list. The root page is a special case since the bucket it contains has no predecessor. This special case is handled by the above algorithm – the disk address of the root page is held in the ARD which is always memory resident.

When an arena is stabilised, its page table must be modified to reflect the new disk pages used to store the data. This will eventually result in the root page being modified and it must also be saved to a new disk page. Having done this, the ARD entry of the root page is now out of date since it must always contain the disk address of the current version. Therefore, the last step in stabilisation is to update the ARD.

One of the advantages of having the meta-data of arenas embedded in the arenas themselves is that the ARD does not need to stabilised after being updated. This is because the ARD does not hold any information that cannot be recovered from the meta-data of the arenas, it is just an efficient way of accessing it. Therefore, the ARD resides in the temporary region of the kernel's address space. Unfortunately, the process of reconstructing the ARD from the meta-data in the arenas, described later, can be time-consuming. Therefore, if shutdown is orderly, the ARD is saved to a special area on disk so the time taken to restart the system is reduced. This process must indicate whether the save succeeded or not. This can be achieved using a time stamp method similar to Challis' algorithm[4]. In contrast to Challis' algorithm though, the ARD is not shadowed. If it is not saved before a crash, it can be reconstructed from the meta-data in arenas.

When an arena needs to grow, free virtual pages must be allocated to it. To minimise the size of the ARD, the store always tries to allocate from the free list threaded through the ARD itself. If this is unsuccessful, the active region is expanded into the inactive region which allocates new pages to the active region but necessitates increasing the size of the ARD.

6.4 Recovery

When the system is restarted, whether after a crash or an orderly shutdown, the kernel persistent store need only provide access to the most recent stabilised versions of arenas. It does not have to guarantee consistency between the data of these versions; it is up to higher-level software to embed enough information in each version so that a consistent global state may be found by inspecting the latest versions and rolling back where appropriate.

When shutdown is orderly, the store saves the entire ARD to disk. On restart, the ARD is simply read into memory – arenas will be brought in by the usual mechanism of demand paging.

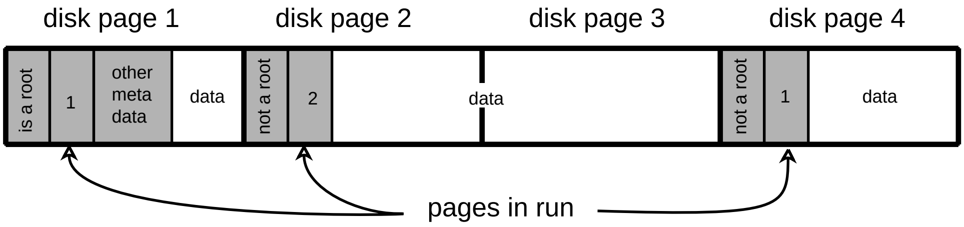

Figure 6. Recovery tags.

When shutdown is not orderly, the recovery process is complicated by the fact that the ARD will not be on disk. It must therefore be reconstructed from the meta-data contained in the root pages of the most recent versions of the arenas. The first step in this process is to actually find the most recent root pages. The is achieved by having a tag, called a recovery tag, at the beginning of each page indicating whether it is a root page or not. The entire store is then scanned to find the root pages; version numbers are used to find the most recent ones. The need to scan the entire store seems, at first, to be excessive. It must be remembered though that the store only contains *kernel* data and is therefore relatively small. In addition, the scan is performed on sequential disk blocks which means that average seek time for each page is low. Once the ARD is reconstructed, the most recent versions will be available but the store does not guarantee that these versions form a consistent global state. It is up to higher-level software to embed enough information in each version so that such a state may be found by reinstating previous versions.

This approach is complicated by data structures that span more than one virtual page since, in this case, the tag cannot be embedded at the beginning of intermediate pages. The current implementation solves this problem by saving spanned pages in a contiguous run of disk pages and including the number of pages in the run in the first page. When scanning the disk, the recovery algorithm uses this tag to determine which disk page to look at next. For example, in figure 6, the second and third pages contain data that spans a page boundary. This is indicated by the recovery tag at the beginning of page 2. Therefore, the recovery algorithm would ignore page 3 and skip straight to page 4. The concern with this scheme is that it requires the allocation of contiguous disk blocks but, as stated above, such large data structures are rare and therefore do not cause problems in practice.

6.5 Virtual Memory Allocation

Another property of the store is that it shouldn't, through its operation, cause dependencies between arenas where such dependencies are not apparent to higher-level software. Without this guarantee, the consistency algorithm, which is implemented above the store, will not take these dependencies into account which could result in an incorrect system state after recovery. For example, in figure 7, arena A is allocated virtual page 8 before a stabilisation point. After arena A has been stabilised virtual page 8 is no longer needed (maybe because the arena became smaller) and is therefore freed. After this time arena B is allocated virtual page 8 and is thereafter stabilised. If the system crashed at time T, the system would resume with both A and B having virtual page 8 allocated to them.

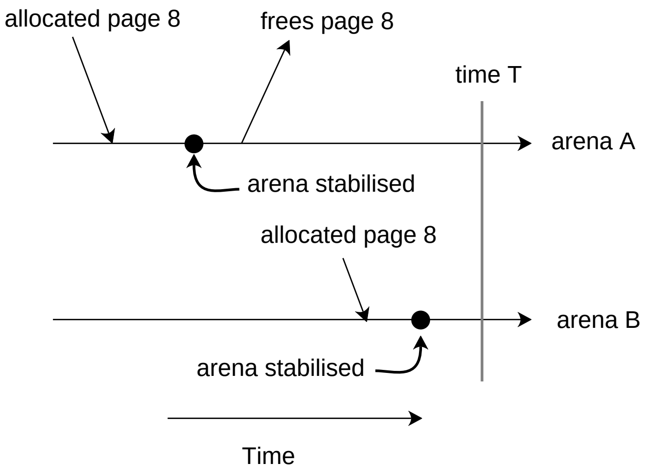

Figure 7. A crash at time T leads to an inconsistent state: both arena's are allocated virtual page 8.

To avoid this problem each version of an arena includes a list of virtual pages freed from the arena since the last consistent state. Such pages are called *locally* free pages. When a particular version becomes part of a consistent state, all locally free pages can be moved to the global pool to become *globally* free. This is guaranteed to avoid inconsistency since the consistency algorithm ensures that no entity will be rolled back beyond the last consistent state.

To allocate free pages to an arena, its list of locally free pages is inspected first. If it is empty, a globally free page is allocated either from the list threaded through the ARD or by extending the active region into the inactive region.

6.6 Disk Allocation

On recovery it is possible that past versions, back to and including the version contained in the last consistent state, will need to be reinstated. Therefore, the disk pages recording these versions must not be reused until a new consistent state is found that includes a later version of the arena. To speed this process, each arena maintains a list of disk pages used by previous versions of the arena. When a version becomes part of a consistent cut, members of this list may be freed, since there is no longer a possibility that they will be used.

When an arena is rolled back to a previous state by the consistency algorithm any disk pages allocated to the more recent versions may be freed since they will never be needed again. Therefore, the list of disk pages must also include the version number of the state in which the pages were allocated. On rollback, any disk pages allocated after the consistent version can be freed.

6.7 Memory Allocation for Meta Data

Each arena's meta data, including the lists in section 6.6 and 6.5, is embedded within itself. This means that as more meta-data is required, the memory will have to be allocated from the arena itself. For example, the list of locally free virtual pages and the list of allocated disk blocks are both dynamic structures of unpredictable size. In most cases, allocating memory for these structures is no different from allocating memory for normal data and so the same mechanism can be used. This significantly eases the implementation of arenas and means that arbitrary size limits, based on statically sized meta-data, need not be imposed.

In one particular case, increasing the size of the page table, care must be taken. As mentioned previously, the page table is implemented as a list of fixed sized buckets that must be searched linearly to find a particular entry. As buckets fill, new buckets must be allocated from the arena to extend the page table. The problem is that waiting until the last bucket is full to allocate a new bucket may not succeed since the last page allocated to the arena, the one in the last entry of the bucket, may already be full. If this occurred, a new page would have to be allocated to the arena, which would require a new page table entry. Since the original task was to create more page table entries, this process must fail.

The solution is to allocate a new bucket *before* the last entry in the last bucket is used to expand the arena. This requires the allocation of a new virtual page to the arena to accommodate the new bucket. Unfortunately, the rest of this page may never be used, wasting both virtual memory and disk space. Again, we rely on the fact that arenas are sufficiently small on average that they rarely require more than one bucket.

7 Conclusion

In order to meet the goals of orthogonal persistence, the kernel of the Grasshopper operating system needs a persistent store that supports multiple versions and independent stabilisation of entities and that is efficient and unobtrusive.

An important limitation while designing the store is that the kernel is implemented in C and therefore sophisticated language support is not available. A significant consequence of this is that, to be unobtrusive, the store must be based on virtual addresses rather than software identifiers. The drawbacks of this approach are that the size of the store is limited by the underlying hardware and that stabilisation must be performed on pages, complicating the need for multiple versions and independent stabilisation of entities.

These problems are alleviated by the fact that the store is not general purpose; it is used for kernel data structures only. This means that the general form of data structures and the relationships between them are known in advance. In particular, individual data structures are small, from tens to hundreds of bytes, and form natural groupings that should be stabilised together.

These groupings are exploited by the store. Rather than provide for the independent stabilisation of individual data structures, it provides for the independent stabilisation of regions of the virtual address space, called persistent arenas. These regions are used as memory pools for stability groups by higher levels of software, which are also responsible for the consistency between them.

Acknowledgments

The work described in this paper is supported by Australian Research Council grant A49130439 and by an equipment grant under the Alpha Innovators program from Digital Equipment Corporation. We would like to thank the Flamingo people at the Mill for the their help in getting us started and the OSF group from Nashua for answering many questions during kernel development. The authors would also like to thank the other members of the Grasshopper group – Matty Farrow, Frans Henskens and David Hulse – for their assistance in developing the ideas presented in this paper.

References

[1] Acceta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Trevanian, A. and Young, M. "Mach: A New Kernel Foundation for Unix Development", *Proceedings, Summer Usenix Conference*, pp. 93-112, 1986.

[2] Anderson, M., Pose, R. and Wallace, C. S. "A Password-Capability System", *The Computer Journal*, vol 29, 1, pp. 1-8, 1986.

[3] 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.

[4] Challis, M. F. "Database Consistency and Integrity in a Multi-User Environment", *Databases: Improving Usability and Responsiveness*, pp. 245-270, 1978.

[5] Chandy, K. and Lamport, L. "Distributed Snapshots: Determing Global States of Distributed Systems", *Transactions on Computer Systems*, vol 3, 1, pp. 63-75, 1985.

[6] 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.

[7] Dasgupta, P., LeBlanc, R. J. and Appelbe, W. F. “The Clouds Distributed Operating System”, *Proceedings, 8th International Conference on Distributed Computing Systems*, 1988.

[8] 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*, Morgan Kaufmann, pp. 179-195, 1989.

[9] Dearle, A., di Bona, R., Farrow, J., Henskens, F., Hulse, D., Lindström, A., Norris, S., Rosenberg, J. and Vaughan, F. “Protection in the Grasshopper Operating System”, Universities of Sydney and Adelaide, GH-04, 1993.

[10] Dearle, A., di Bona, R., Lindström, A., Rosenberg, J. and Vaughan, F. “User-level Management of Persistent Data in the Grasshopper Operating System”, Universities of Adelaide and Sydney, Technical Report GH-08, 1994.

[11] Dearle, A., Rosenberg, J., Henskens, F. A., Vaughan, F. and Maciunas, K. “An Examination of Operating System Support for Persistent Object Systems”, *Proceedings of the 25th Hawaii International Conference on System Sciences*, vol 1, Hawaii, U. S. A., ed V. Milutinovic and B. D. Shriver, IEEE Computer Society Press, pp. 779-789, 1992.

[12] Johnson, D. “Efficient Transparent Optimistic Recovery for Distributed Application Systems”, Carnegie Mellon University, CMU-CS-93-127, 1993.

[13] Johnson, D. B. and Zwaenepoel, W. “Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing”, *7th Symposium on Principles of Distributed Computing*, ACM, pp. 171-181, 1988.

[14] Koo, R. and Toueg, S. “Checkpointing and Rollback Recovery for Distributed Systems”, *IEEE Transactions on Software Engineering*, vol 13, 1, pp. 23-31, 1987.

[15] Lauer, H. C. and Needham, R. M. “On the Duality of Operating System Structures”, *Operating Systems Review*, vol 12, 2, pp. 3-19, 1979.

[16] Li, K. and Naughton, J. “Concurrent Real-Time Checkpoint for Parallel Programs”, *2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 79-88, 1990.

[17] Mullender, S. J., van Rossum, G., Tanenbaum, A. S., van Renesse, R. and van Staveren, H. “Amoeba: A Distributed Operating System for the 1990s”, *IEEE Computer*, 23(5), pp. 44-53, 1990.

[18] 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.

[19] 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.

[20] Strom, R. and Yemini, S. “Optimistic Recovery in Distributed Systems”, *Transactions on Computer Systems*, vol 3, 3, pp. 204-226, 1985.

[21] Vaughan, F., Lo Basso, T., Dearle, A., Marlin, C. and Barter, C. “Casper: a Cached Architecture Supporting Persistence”, *Computing Systems*, vol 5, 3, pp. 337-359, 1992.

[22] Young, M. W. “Exporting a User Interface to Memory Managment from a Communication-Oriented Operating System”, Ph.D. Thesis, Carnegie Mellon University, 1989.