A Model For User-Level Memory Management in a Distributed, Persistent Environment

[†]Anders Lindström, [*]Alan Dearle, [†]Rex di Bona,
[†]J. Matthew Farrow, [†]Frans Henskens,
[†]John Rosenberg, [*]Francis Vaughan.

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

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

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

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

Abstract

The Grasshopper operating system provides a flexible environment for conducting research into orthogonal persistence. In particular, it allows user-level software to perform memory management so that new techniques may be investigated without having to modify or even reboot the kernel. We describe the facilities provided to support this and show how they are used by both the kernel and user-level software.

Introduction

In a conventional operating system two abstractions of data access and storage are provided: virtual memory and files. Data in virtual memory may be directly accessed but does not outlast the computation that created it. On the other hand, data in files must be accessed using system calls but may last for an arbitrary length of time.

A system that supports orthogonal persistence unifies virtual memory and the file system to provide a single abstraction of data storage and access. This is motivated by the realisation that data should be accessible in a uniform manner regardless of its creator, longevity or type [5] . The chief advantage of 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.

For example, consider a graph structure that must be accessed by programs other than its creator. In a conventional system, this sharing is achieved by the transforming the graph into a 'flat' form suitable for the file system or a database. Any other program wishing to use it must then access the file or database and reconstruct the graph from the flattened form. In a persistent system, this transformation is unnecessary. The graph exists in its original form for as long as it is needed. Thus, both the creator and other programs access it with equal ease.

A number of early systems [7, 22, 28] provided an abstraction of data storage and access called the *single–level store*. While these systems did not provide true orthogonal persistence, they did remove the distinction between long and short term data. This idea was both simple and powerful but did not find its way into the mainstream due to weaknesses in other parts of the system or overdependence on custom hardware.

A true persistent system [2, 6, 11, 24] extends the notion of a single-level store by providing data stability, which is the ability to restore data to a *consistent* state after the system crashes or is shutdown [25] . Single-level stores could recover data but they did not guarantee consistency. Conventional systems can also recover data, even sometimes to a consistent state, but only if it is in a file or a database. In an orthogonally persistent system a uniform abstraction is provided and so *all* data is stabilised, not just that data that is in a file or a database. Stability does not mean that all *current* changes to data can be recovered. Rather, it means that the system maintains some consistent state to which it can revert in the face of failure. When required, a checkpoint occurs that forces the saved consistent state to reflect the current state of the system ensuring that not too much is lost if the system crashes. This is normally achieved using logging [4] or some shadowing technique [19] .

Most persistent systems are implemented either on top of a conventional operating system or employ specialised hardware. The disadvantages of the first approach are inefficiency and poor operating system support for user–level memory management [14] . The second approach, while promising, limits the possibility of distributing the system to interested parties. A third alternative is to implement a system that runs directly on conventional hardware.

There are a number of systems that take this approach [8, 9, 10, 21] . All of these systems are, at least originally, research oriented and tend to emphasise different areas. Clouds, for example, was constructed to investigate distributed computation while Alpha emphasises real-time applications. The Grasshopper system, being developed jointly by the Universities of Adelaide and Sydney, is primarily a platform for conducting research into issues directly related to persistence. These include how persistence is provided in a distributed environment, how causal dependencies are tracked to ensure data consistency, and how persistence may be exploited for both constructing applications and improving efficiency.

One of the most important issues in a persistent system is how the persistent store is managed. Persistent data management encompasses two tasks. The first is to move data between backing store, which provides resilience, and main memory, which provides efficient access. The second is to maintain the consistency of the persistent store. If the abstraction is to remain truly uniform, these tasks must be carried out in a way that is totally transparent to applications using the persistent store.

The importance of data management in persistent systems makes it an active area of research in much the same way as file systems are an active area of research in conventional operating systems. To allow for such research to be conducted, Grasshopper provides mechanisms that allow persistent data management to be conducted by user-level software. This means that new ideas may be tested without having to alter the kernel. Similar facilities have been provided in a number of conventional systems but in the context of virtual memory management [1, 15, 29] and file systems [20, 23] .

The aim of this paper is to describe data management in Grasshopper [13] In particular, it describes the facilities provided for performing data management in user mode and gives some examples of how they might be used.

Overview of Grasshopper

Hardware. Grasshopper is targeted at a hardware base that consists of a number of conventional workstations connected by a local area network. Each workstation, or node, is assumed to consist of one or more processors, some physical memory, and zero or more disks. A processor may only directly access the memory of the node to which it belongs. If a processor wishes to access data in remote memory, a copy must be made into local memory by passing messages over the network.

Hardware support for memory management is assumed to be based on fixed sized pages. For the purposes of this paper it is assumed that a software-filled translation lookaside buffer(TLB) is provided. One of the properties of TLBs is that they can only hold a finite number of entries. When an access is made to a page for which no entry exists an exception, called a TLB miss, is raised. When this occurs, the system must determine whether the page is memory resident or not. If it is, the TLB can be updated and the access can continue. If it is not, a copy of the page must be placed in main memory. The detection of an access to non memory-resident page is called a page fault.

In common with other distributed systems, Grasshopper has a distributed kernel. Each node in the network has a copy of the kernel, called that node's *local kernel,* which co-operates with the local kernels of other nodes to provide a single, logical system. The basic abstractions provided to users are *containers*, *loci*, *invocation*, *capabilities* and *mapping*.

Containers. Containers are an abstraction of data storage and access. They are passive, persistent entities that provide a means for storing data. Further, they provide an environment for the execution of any number (from zero up) of concurrent activities. A typical container contains data and code to manipulate that data.

Figure 1: Containers and Loci

Loci. 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.

Invocation. Grasshopper is a *procedure-oriented* or *object-thread-based* system [10, 17] . 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 same spot when it returns from the invocation. It differs from calling a procedure in that the invoked container becomes the locus' host container and so provides a whole new addressing environment. This means that the locus can no longer access the contents of the container from which the invocation occurred.

When a locus invokes a container, the point at which it begins executing is determined by the container's *invocation point*. 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 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 in container *a*. It then moves to *b* and thence *c* by invocation and then returns 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 each container may have many loci executing in it concurrently.

The invocation mechanism is extremely important in Grasshopper since it is used to provide hardware-protected abstract data types(ADTs). Each instance of an ADT is a single container whose operations are accessed by invocation. For example, the traditional abstraction of a file could be implemented in Grasshopper as a container with five operations: *open*, *close*, *read*, *write* and *seek*. When a locus needed to perform an operation on a file, it would simply invoke it specifying the appropriate operation.

Languages that provide abstract data types rely on the compiler to enforce protection. By contrast, Grasshopper enforces protection at the hardware level. 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 the distinction between kernel- and user-level services is totally transparent.

Invocation is also important because it is ideally suited to distributed computing. If a locus invokes a container that resides on a remote node, the kernel may transparently migrate the locus to that node or it may bring the container to the local node. The decision as to which option is taken is based on factors such as the load on each node and the expected cost of migrating the container. Since invocation is a network transparent operation it is immaterial to the locus which of these the kernel chooses.

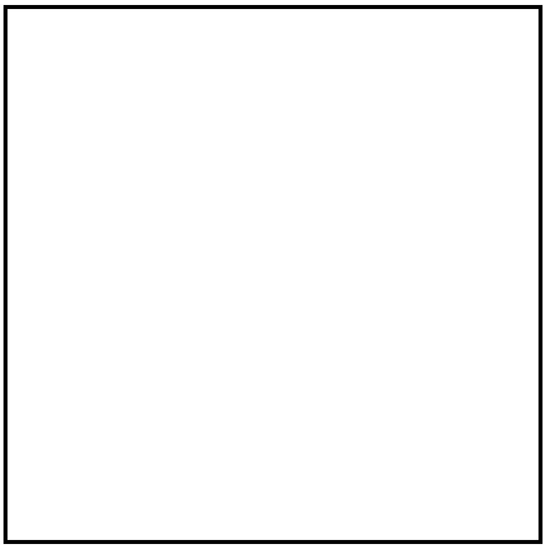

Figure 2: Mapping

Capabilities. Capabilities are the sole means of referencing and protecting entities in Grasshopper [12] . Whenever the code executed by a locus calls for the manipulation of another object, a capability must be presented. A capability is a system-protected object that serves to identify an entity and to define what operations the holder may perform on it. Of main interest to this paper is the fact that capabilities provide a location independent way of referencing entities. Since the kernel maintains the correspondence between capabilities and entities, the user need not be aware of an entity's location.

Mapping. One of the chief advantages of persistent systems is that they allow sharing of arbitrary data structures between different programs. In Grasshopper, sharing can be achieved in two ways. The first is by modelling the data structure as an abstract data type and putting it and the code to manipulate it in a container that may be accessed by invocation. It is expected that this method will be used when protection is a priority.

The second method of sharing data is by *mapping*, which provides direct access to the contents of a container by making parts of it visible in another container. Mapping allows, for example, instances of abstract data types to share code rather than having to have a their own separate copy.

Mapping in Grasshopper differs from similar features in other systems in two major 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.

All loci in a container perceive the same address space including any mappings that have been made. In some cases it is desirable to provide a locus with its own private data which no other locus can perceive. To satisfy this need, Grasshopper provides *invocation mappings*, which take precedence over any mappings in the host container and are only perceived by the locus for which they are performed. This allows, for example, each locus to have its own stack space with the stacks of all loci occupying the same address range within the host container. This technique both simplifies multi-threaded programming and provides a useful security mechanism unavailable in conventional systems.

Large Containers. For some applications a 32 bit address space, which is typical of conventional hardware, is not large enough to hold all the required data. To accommodate such large data sets, the design of Grasshopper does not restrict container addresses to be the same size as the hardware-supported virtual addresses. For example, the Grasshopper prototype is implemented on Sun 3s, which are based on the Motorola 68020 32-bit processor but the kernel defines container addresses to be 64 bits.

Having containers that are too large to be addressed directly by the hardware immediately poses a problem. One solution is to use pointer swizzling techniques [27] . Another is to use mapping. Figure 3 illustrates the latter method, which is to map inaccessible portions of one container into an accessible portion of another container. Thus, the hardware supported virtual address space acts as a window on the larger container address space.

Figure 3: Accessing a Large Container

Data Management in Grasshopper

One of the most important issues in the design of Grasshopper is how the container abstraction is implemented. The two key issues are how to provide loci with direct, instruction-level access to container data and how to maintain the resilience and consistency of that data.

The techniques used to address these issues are an active area of research. So that research into these techniques may be carried out in a flexible manner, the kernel does not wholly implement the container abstraction. Rather, it co-operates with user-level entities called *container managers* that provide essential services related to a container's data such as moving data from disk to physical memory and maintaining data stability. A container manager is simply a user-level container that conforms to a common interface defined by the kernel. When the kernel requires some aspect of data management to be performed on a container it invokes that container's manager.

When a container is created a manager must be specified. It is possible to change a container's manager at a later stage, as discussed below, and so this initial manager is called its *default manager*.

Figure 4 illustrates an example of user-level data management. The locus in container *a* accesses a page that is not resident in memory. This causes a page fault that is handled by the kernel's exception handler. The kernel then finds the container's manager, container *b*, and invokes it requesting that it service the page fault. The manager uses the facilities described below to obtain a copy of the data from backing store and place it in physical memory so that the access may continue.

Figure 4: Handling a Page Fault

Note that when an exception is caused by a locus the kernel can simply use it to invoke the manager. Such a kernel-initiated invocation is called a *forced invoke* since it is not a result of an explicit invoke operation and is not controlled by the locus. The advantage of performing a forced invoke is that the exception is handled at the priority of the faulting locus.

There are number of advantages gained by having data management relegated to user-level. The first and most important for Grasshopper is that new techniques may be implemented and evaluated without having to modify or even reboot the kernel. This means that experiments may be conducted with almost no impact on the rest of the system. The second advantage is that much complexity is removed from the kernel making it both easier to write and maintain.

Some interesting possibilities for user-level data management are container managers that compress and/or encrypt data before it is moved to disk, data managers that dynamically adapt page replacement and prefetching policies to data access patterns and data managers that perform concurrent garbage collection. Clearly, without the ability to implement these policies at user-level, the kernel would soon become a monolith that would be difficult to maintain, extend and debug.

For container managers to perform the services required of them they must have a certain degree of control over physical memory, backing store, and address translation information. This goal is constrained by the need to protect managers and the data of the containers they manage from other managers. Thus, it is not satisfactory to let managers indiscriminately alter any part of physical memory or disk. Instead, they must be constrained to use only those parts which they have been allocated.

The approach taken by Grasshopper is to provide a number of kernel-implemented abstract data types that model the underlying hardware. Instances of these abstract data types are simply containers which are accessed by invocation. The three main abstract data types used specifically by managers are *disks*, *physical page sets* and *local container descriptors(LCDs).* Disks are an abstraction that allow managers to access physical disks in a controlled manner, physical page sets are used by managers to manipulate physical memory, and LCDs are used to store address translation information such as the correspondence between virtual and physical addresses.

The following sections describe the responsibilities of container managers, the responsibilities of the kernel, and the facilities provided to managers to perform data management. Following this, an example is given to illustrate the use of these facilities.

User-Level Data Management.

All aspects of the data in a container are the responsibility of that container's *manager*. A manager is simply a user-level container that is invoked by the kernel whenever it requires the following data management services to be performed on its behalf:

- handling page faults
- handling access violations
- performing page discard
- performing stabilisation

A page fault occurs when a locus attempts to access data that is not resident in the local memory of the node on which the locus executes. Such an access causes an exception to be raised by the node's memory management hardware. In response to a page fault, the kernel invokes the container's manager specifying at what address, in which container and on what node the fault occurred. It is the managers responsibility to retrieve a copy of the data from backing store and place it in the local memory of the faulting node.

Access violations occur when a locus attempts, for example, to write to a container address for which writing is disallowed. When an access violation occurs the container's manager is invoked by the kernel with parameters specifying the address and the type of the attempted access. The manager may handle the violation in any way that it sees fit. There are a variety of applications for which detection of access violations are important [3] .

At some point in time the local memory of a node will be filled with copies of pages from various containers. When this occurs, servicing of page faults cannot proceed until some of the memory is freed. When the kernel detects that this situation has arisen it invokes one or more managers requesting that they discard a number of pages on its behalf. The choice of *which* pages are discarded is left to the individual managers allowing them to implement any algorithm they choose. This is particularly useful when the traditional least-recently-used algorithm is inappropriate [15, 26] .

The kernel is responsible for maintaining system-wide stability. When checkpoints occur it determines which containers need to be stabilised to ensure consistency then invokes the appropriate managers. It is up to the manager to use any policy it chooses to fulfil such requests. The techniques by which stability is achieved are beyond the scope of this paper and the reader is referred to [25] .

Kernel-Level Data Management

The kernel is responsible for dispatching data management requests to container managers and providing the abstractions that managers use to service these requests.

To dispatch requests, for example when a locus causes a page fault or an access violation, the kernel must determine which container manager to invoke. This task is complicated by the presence of mappings. For example, in Figure 2 address *b* in C1 corresponds to address *e* in C3. This means that a page fault at *b* in C1 would have to be handled by C3's manager. The container to which an address eventually maps is called its *base container*. The address within the base container to which it corresponds is called its *base address*. Thus, *b*'s base container is C3 and its base address is *e*.

The kernel encapsulates all mapping information in a container called the *mapping graph*. The mapping graph is an abstract data type that may be manipulated by invoking the following operations:

- **cont_map**(cap *dest*, cont_offset *dest_offset,*
  cap *src,* cont_offset *src_addr,*
  cont_offset *len*)
- **cont_unmap**(cap *cont*, cont_offset *addr*)
- **cont_invocation_map**(cap *locus*, cap *dest*,
  cont_offset *dest_addr*, cap *src*,
  cont_offset *src_addr*, cont_offset *len*)
- **cont_invocation_unmap**(cap *locus*,
  cap *cont*, cont_offset *addr*)
- **cont_translate**(cap *cont*, cap *locus*,
  cont_offset *addr*) returns (cap *base_cont*,
  cont_offset *base_addr*)

cont_map maps the region from *dest_addr* to *dest_addr*+*len*-1 in the container referred to by the capability *dest* to the region *src_addr* to *src_addr* +*len*-1 in container referred to by *src*. Any data that was visible in this region is now hidden.

cont_unmap undoes the topmost mapping in the container referred to by *cont* that includes the address *addr*. Any data that was previously obscured by the mapping becomes visible.

cont_invocation_map performs a map in the same way as **cont_map** with the exception that only the locus specified by the capability *locus* perceives the mapping.

cont_invocation_unmap removes any invocation mapping made for the specified locus at the given address.

cont_translate traverses the mapping graph from address *addr* in the container referred to by *cont* taking in to account any invocation mappings made for the specified locus. It returns the base address and a reference to the base container.

The mapping graph is mainly used by the kernel to determine base containers and addresses. For example, in Figure 2 if a page fault occurred at address *b* in C1, the kernel would invoke **cont_translate** and find that the base container is C3 and the base address is *e*. It would then invoke C3's manager to service the page fault.

Note that all operations on the mapping graph are performed subject to the permissions embodied in the capabilities presented. This prevents arbitrary loci from altering the mapping graph unless they been given permission to do so.

Kernel-Provided Facilities for User-Level Data Management

Physical Page Sets. When a page fault occurs on a node it the responsibility of the manager to place a copy of the data in the physical memory of the faulting node. To achieve this it must have some degree of control over memory. Management of physical memory by user-level code is complicated by the need to ensure that it cannot access physical memory allocated to other managers. In Grasshopper, this is achieved with an abstraction of physical memory called a *physical page set*.

Figure 5: Two Physical Page Sets

A physical page set models a number of equally sized pages from a node's physical memory. Each page in the set is given a unique index by which it is identified. Figure 5 shows two examples of physical page sets on a node. Indices of pages within a page set do not, in general, map to the same page number in the underlying physical memory. Also, as shown by PS2, the size of pages in a page set may be different from the underlying page size. Note, however, that they must be a multiple of the underlying page size and that the physical memory for a single page must be contiguous.

The operations on physical page sets are as follows:

- **ps_create**(cap *mem_manager*, cap *node*, int *npages*, page_type *type*) returns (cap *page_set*)
- **ps_grow**(cap *page_set*, int *npages*) returns (int_list *new_pages*)
- **ps_remove**(cap *page_set*, int_list *old_pages*)
- **ps_copy**(cap *dest_page_set*, int *dest_page*, cap *src_page_set*, int *src_page*)
- **ps_transfer**(cap *dest_page_set*, int *dest_page*, cap *src_page_set*, int *src_page*)
- **ps_get_stats**(cap *page_set*, int *page*) returns (page_stats *stats*)
- **ps_set_stats**(cap *page_set*, int *page*, page_stats *stats*)

ps_create requests that a new page set be created on the node referred to by *node* with *npages* pages each of *page_size* bytes. If the required memory is not available, the operation fails. When a page set is created its type must be specified to tell the kernel what the memory will be used for so that it can conform to any hardware constraints. For example, memory used by device drivers for direct memory access(DMA) is a special case. The only two types at the moment are *ordinary*, for general use, and *DMA*, for direct memory access by device drivers. The operation returns a reference to the newly created page set.

ps_grow increases the amount of physical memory associated with a page set by *npages* pages. If the required memory is not available, the operation fails. The indices of the new pages are specified by *new_pages*.

ps_remove removes the pages specified by *old_pages* from the page set. The memory backing these pages is freed for use by other page sets.

ps_copy copies the data from a page in one page set to a page in another page set.

ps_transfer moves a page from one page set to another. *dest_page* must have already been allocated before the transfer. This avoids the possibility of trying to transfer a page to a remote node where no physical pages are available. After the transfer, the contents of *dest_page* are the same as *src_page*, and *src_page* is no longer valid in *src_page_set*.

ps_get_stats returns the access flags of the specified page. The flags indicate whether the page has been accessed and whether it has been modified.

ps_set_stats sets the access flags for the specified page. This will usually be used by a manager to clear the flags after which it can use **ps_get_stats** to determine whether the page has been accessed or modified. This facility is essential for implementing certain page discard policies such as the popular clock algorithm and its variants.

Local Container Descriptors. When data in a container is accessed, a copy of it must be moved to physical memory by the container's manager. If many loci, each executing on a different node, access the same page of data in a container, a copy will have be made in the local memory of each of those nodes. This situation may arise when a container on one node is mapped into container on other nodes. For example, in Figure 2, if C1, C2 and C3 were all on different nodes and address *b* were accessed in C3 and address *d* were accessed in C2 then two copies of the data from C3 would exist: one on C1 and one on C2. When this situation arises a consistency protocol [16, 18] . must be implemented by the manager so that only one logical copy of the data exists

In general, the set of memory-resident pages for a container on one node will differ from the set of memory-resident pages for the same container on a different node. To keep track of these sets, a data structure called a *local container descriptor*(LCD) is maintained on every node which has memory resident pages of a container in its local memory. At any point in time the LCD describes which pages of a container exist in the local memory of a node and at what physical address they exist by referring to a page in a page set.

Figure 6: Two Local Container Descriptors

Figure 6 shows two LCDs for container *a*(not shown). LCD(a, node1) shows which pages exist in the local memory of node 1. From the illustration it can be seen that pages 0 and 3 exist on node 1 in the physical memory represented by pages 0 and 2 of the page set PS1. LCD(a, node2) shows that pages 3 and 4 of container a exist on node 2 in the physical memory represented by page 0 of PS2. Note that the page sets on each node are of different size.

In addition to memory resident pages, the LCD maintains a reference to the container's manager so that the local kernel knows which container to invoke when it requires data management to be performed on the container represented by that LCD.

Each LCD is an instance of an ADT that presents the following interface:

- **lcd_bind**(cap *lcd*, cont_offset *addr*,
  cap *page_set*, int *page*, prot_flags *prot*)
- **lcd_unbind**(cap *lcd*, cont_offset *addr*)
- **lcd_protect**(cap *lcd*, cont_offset *addr*,
  cont_offset *len*, prot_flags *prot_flags*)
- **lcd_set_manager**(cap *lcd*, cap *manager*)

lcd_bind informs the system that the data in the region from *addr* to *addr*+*len*-1 in a container, whose LCD is *lcd*, is memory resident. The data is in the physical memory described by *page* and page *page_set*. The page has its protection flags set to *prot_flags* which are a combination of *read* and *write*. Any access that violates a page's protection will be delivered to the manager.

lcd_unbind undoes a previous **lcd_bind** at *addr* in the LCD referred to by *lcd*. This does not free the memory however. This is achieved by removing the page from the physical page set to which is belongs.

lcd_protect changes the protection flags of the region *addr* to *addr* + *len* -1 in the LCD referred to by *lcd* to *prot_flags*. This region must be page aligned.

lcd_set_manager changes the container that is invoked for memory management services pertaining to the specified LCD. Since it is possible to set a container's manager on a per node basis, it is possible that many managers for the one container exist at the same time. If this is the case then these managers must cooperate to perform the responsibilities of a container manager. Note that changing the manager for an LCD does not change the default manager of the container.

Disk drives. Backing store in Grasshopper is provided by ADTs called *disks*, which is an abstraction of actual secondary storage devices. Usually, a real disk will be partitioned into many smaller abstract disks each used by a separate manager. For the purposes of this paper a disk is simply an ADT with the following interface:

- **disk_read**(cap *disk*, disk_offset *block*,
  cap *page_set*, int *dest_page*)
- **disk_write**(cap *disk*, disk_offset *block*,
  cap *page_set*, int *source_page*)

disk_read copies data from secondary storage to physical memory. The caller specifies the disk to read from, the block index to start reading from, and a page within a page set to put the data.

disk_write copies data from physical to secondary storage. The caller specifies a page in a page set from which to get the data, the disk to write to and the block on that disk from which the write should begin.

Figure 8: a Container Manager Servicing a Page Fault

Figure 7: a Locus Causes a Page Fault

Kernel Handling of TLB Misses

When a TLB miss occurs it is the kernel that must initially handle the exception. This section describes the sequence of events that occurs.

Exception handling. When a TLB miss occurs, an exception is raised by the hardware. The exception is handled by forcing the locus that caused the miss to invoke the kernel's exception handler. See Figure 4.

A*ddress resolution.* The kernel's exception handler invokes the **cont_translate** of the mapping graph, as explained in a previous section, to resolve the base address and base container of the faulting address.

LCD probe 1. The kernel checks whether it has an LCD for the base container found in the previous stage. If it does not, it creates one and sets the manager to be the base container's default manager.

LCD probe 2. By this stage an LCD exists for the base container found in the address resolution stage. The local kernel uses it to determine whether the page containing the base address is memory-resident. If it is, it proceeds straight to TLB refill.

Page fault service. The page is not resident in the local memory so one must be obtained. The local kernel invokes the manager specified by the LCD. It specifies the base container, base address and node on which the fault occurred. It is up to the manager to supply the data and update the LCD appropriately. An example of how this is achieved is provided in the next section.

TLB refill. By this stage a copy of the page does exist and its physical page set identifier and index within that page set are stored in the LCD. The local kernel uses this information to derive a real physical address and puts an entry in the TLB accordingly.

Exception return. The TLB miss has now been serviced. The faulting locus returns from the forced invocation of the kernel and resumes execution by retrying the access that caused the fault.

User-Level Servicing of Page Faults

In the previous section, a description of the facilities provided to managers for data management was given. In addition, the manner in which TLB misses are handled by the kernel was outlined. The aim of this section is to provide an example of how Grasshopper's data management facilities are used by data managers; in particular, to service page faults.

Figure 7 illustrates the initial situation:

- there are three nodes: **a**, **b** and **c**.
- there are three containers: C, a simple container with no mappings; M, which is the default manager of C and a disk D, which M uses for backing store.
- there is a locus, L, executing in C on node **a**.

- the local container descriptor for C, LCD1 on **a** has already been created, presumably as a response to earlier page faults.
- there are two page sets PS1 and PS2. PS1 has been created by M to service page faults on node **a** and is of type *ordinary*. PS2 has been created by D as a local cache for disk reads and is of type *DMA*.

When L tries to access address *q* in C, the following sequence of events, illustrated by figure 8, occurs:

1. L attempts to access *q* in C and a TLB miss occurs on **a**.
2. The local kernel on **a** follows the procedure for TLB misses and finds that a copy of the required page does not exist in its local memory. It invokes C's manager, M, requesting it to service the page fault.
3. While in M, L determines that the required data is on the secondary storage represented by D so invokes it to read the data into page 1 of PS1.
4. While in D, L reads the data from disk into PS2 and calls **ps_transfer** to move the data from PS2 to PS1. Since the two page sets are not on the same node, a copy of the transferred page is sent from **c** to **a** transparently to the disk driver.
5. The disk driver has completed its task so L returns to M.
6. Back in M, L calls **lct_bind** to map the appropriate page of C to the data just copied to PS1).
7. The manager has completed its task so L returns to the local kernel of **a**.
8. The local kernel on **a** uses LCD1 to service the TLB miss by translating the physical page set and index into a real physical address and updating the TLB.
9. The TLB miss has been serviced successfully so the locus resumes execution.

Two important points are illustrated by this example. The first is that the container in which the fault occurred, the manager of the container and the disk driver used for backing store may all be on different nodes. Since Grasshopper provides location transparency, the machine boundaries are invisible; therefore, it makes no difference where individual components exist. The second point is that data copying is kept to a minimum. Only one copy, that between the two nodes, was needed.

Further Work

The Grasshopper system as presented in this paper is still under development. It is initially being prototyped on a network of Sun 3/60 workstations and will soon be ported to the DEC Alphas donated to the group by Digital Equipment Corporation.

The implementation of the basic memory management facilities is nearing completion after which a number of container managers will be implemented. Initially our goal is to implement three managers. First, a manager that implements basic demand paged virtual memory will be constructed. This will be used to compare the performance of Grasshopper with that of other systems providing user-level memory management. Second, a manager employing shadowing techniques to provide stability will be constructed and evaluated. Following this, it is planned to implement a container that provides stability in a distributed environment.

Related Work

Grasshopper is not the first system to provide user level programs with facilities to perform memory management. Other notable systems include Mach [29] , Chorus [1] , V++ [15] and some versions of UNIX.

Experience with a number of these systems [14] has shown that they are inappropriate for conducting research into persistence. This is not surprising considering they were not designed with persistence in mind. The major reasons these systems are inappropriate are that the level of control they provide is insufficient and that they do not support large enough address spaces. Nevertheless, they have shown that with suitable abstractions user-level memory management is feasible.

Conclusions

Based on previous experience in implementing persistent systems, Grasshopper has been designed to provide a flexible platform for conducting research into persistence.

One of the major areas of research is the management of the persistent store in an efficient manner. To allow research in this area to be conducted, Grasshopper allows user-level software to perform most aspects of memory management, a task normally performed by the kernel. Experimentation and evaluation of new techniques can thus be performed without having to complicate the kernel or disrupt other parts of the system.

The way in which this is achieved is by providing a number of kernel-implemented abstract data types that allow user-level software to manipulate local and remote hardware in a constrained, secure way.

We believe that the ideas presented in this paper improve on work done in other systems that provide similar facilities. These improvements are based on first-hand experience and will allow future research to be conducted in a very flexible and yet secure manner.

Acknowledgements

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.

Thanks go to both Stephen Norris and David Hulse, who have recently joined the Grasshopper project, for their comments on this paper.

References

1. Abrossimov, V., Rozier, M. and Gien, M. "Virtual Memory Management in Chorus", *Progress in Distributed Operating Systems and Distributed Systems Management*, Springer Verlag, Berlin, 1989.
2. 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.
3. Appel, A. W. and Li, K. "Virtual Memory Primitives for User Programs", *ASPLOS IV*, ACM, Santa Clara, California, pp. 96-107, 1991.
4. Astrahan, M. M. "System R: Relational Approach to Database Management", *ACM Transactions on Database Systems*, 1(2), pp. 97-137, 1976.
5. 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.
6. Atkinson, M. P., Chisholm, K. J. and Cockshott, W. P. "PS-algol: An Algol with a Persistent Heap", *ACM SIGPLAN Notices*, 17(7), pp. 24-31, 1981.
7. Bertis, V., Truxal, C. D. and Ranweiler, J. G. "System/38 Addressing and Authorisation", *I.B.M. System/38 Technical Developments*, pp. 51-54, 1978.
8. Campbell, R. H., Johnston, G. M. and Russo, V. F. "Choices (Class Hierarchical Open Interface for Custom Embedded Systems", *ACM Operating Systems Review*, 21(3), pp. 9-17, 1987.
9. Chase, J. S., Levy, H. M., Baker-Harvey, M. and Lazowska, E. D. "Opal: A Single Address Space System for 64-Bit Architectures", *Third IEEE Workshop on Workstation Operating Systems*, IEEE, 1992.
10. Dasgupta, P., LeBlanc, R. J. and Appelbe, W. F. "The Clouds Distributed Operating System", *Proceedings, 8th International Conference on Distributed Computing Systems*, 1988.
11. 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.
12. 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", *(Submitted to) Proceedings, Third International Workshop on Object Orientation in Operating Systems*, Ashville, Noprth Carolina, IEEE, 1993.
13. Dearle, A., di Bona, R., Farrow, J., Henskens, F., Lindström, A., Rosenberg, J. and Vaughan, F. "Grasshopper: An Orthogonally Persistent Operating System", *Technical Report, Departments of Computer Science, Universities of Adelaide and Sydney*, 1993.
14. 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.
15. Harty, K. and Cheriton, D. R. "Application-Controlled Physical Memory using External Page-Cache Management", *ASPLOS V*, ACM, Boston, 1992.
16. Henskens, F. A., Rosenberg, J. and Keedy, J. L. "A Capability-based Distributed Shared Memory", *Australian Computer Science Communications*, 13(1), pp. 29.1-29.12, 1991.
17. Lauer, H. C. and Needham, R. M. "On the Duality of Operating System Structures", *Operating Systems Review*, vol 12, 2, pp. 3-19, 1979.
18. Li, K. "Shared Virtual Memory on Loosely Coupled Multiprocessors", Ph.D. Thesis, Yale University, 1986.
19. Lorie, R. A. "Physical Integrity in a Large Segmented Database", *ACM Transactions on Database Systems*, 2,1, pp. 91-104, 1977.
20. Mullender, S. J., van Rossum, G., Tanenbaum, A. S., van Renesse, R. and van Staveren, H. "Amoeba: A Distributed Operating System for the 1990s", *Computer*, 23(5), pp. 44-53, 1990.
21. Northcutt, J. D. "Mechanisms for Reliable Distributed Real-Time Operating Systems—The Alpha Kernel", Academic Press, 1987.
22. Organick, E. I. "The Multics System: An Examination of its Structure", MIT Press, Cambridge, Mass., 1972.
23. Pike, R., Presotto, D., Thompson, K. and Trickey, H. "Plan 9 from Bell Labs", *Summer 1990 UKUUG Conference*, London, pp. 1-9, 1990.
24. Rosenberg, J. "The MONADS Architecture - A Layered View", *Proceedings of the 4th International Workshop on Persistent Object Systems*, Morgan-Kaufmann, 1990.
25. Rosenberg, J., Henskens, F. A., Brown, A. L., Morrison, R. and Munro, D. "Stability in a Persistent Store Based on a Large Virtual Memory", *Proceedings of the International Workshop on Architectural Support for*

Security and Persistence of Information, ed J. Rosenberg and J. L. Keedy, Springer-Verlag and British Computer Society, pp. 229-245, 1990.

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

27. Wilson, P. R. "Pointer Swizzling at Page Fault Time: Efficiently Supporting Huge Address Spaces on Standard Hardware", *Computer Architecture News*, 19(4), ACM, pp. 6-13, 1991.

28. Wulf, W. A., Levin, R. and Harbison, S. P. "HYDRA/C.mmp: An Experimental Computer System", McGraw-Hill, New York, 1981.

29. Young, M. W. "Exporting a User Interface to Memory Managment from a Communication-Oriented Operating System", Ph.D. Thesis, Carnegie Mellon University, 1989.