Grasshopper - A Persistent Operating System for Conventional Hardware

†Alan Dearle, *Rex di Bona, *James Farrow, *Frans Henskens,
*Anders Lindström, *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
{rex,matty,frans,anders,johnr}@cs.su.oz.au

Abstract

This paper describes Grasshopper, an operating system designed to provide generic mechanisms capable of being tailored to support a wide range of persistence paradigms. A constraint placed on this design is that the system must be implementable on conventional architectures which support paged virtual memory.

In this paper the basic system abstractions relating to addressing environments, processes, and protection are described. It is shown that these provide explicit support for distributed persistent objects and processes, stability, and access control. At the same time the system provides the flexibility to allow user implementation of alternate object management techniques.

1. Introduction

Most persistent systems developed to date have been constructed above conventional operating systems. The fact that these operating systems do not provide an ideal platform for such construction is not surprising since support for the principles of orthogonal persistence was not among their design goals. In [21], for example, Tanenbaum lists the four major components of an operating system as being memory management, file system, input-output and process management. In persistent systems the functionality of the file system and memory management is replaced by the persistent store. Many conventional operating systems extend the file abstraction to encompass input and output, an approach which is obviously inappropriate for persistent systems. Some persistent systems require that the state of a process persists. This is not easily supported using conventional operating systems.

It is 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. The principal requirements of such an operating system may be summarised as follows [7]:

i. *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.

ii. *Stability and resilience for objects:* The system must reliably manage the transition between long and short term memory transparently to the programmer.

iii. *Integration of processes into the object space:* Process state should itself be contained within persistent objects. The importance of this is that processes themselves become persistent and resilient.

iv. *Control over access to objects:* Although the persistent store is uniform, there is still a requirement for restricted 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*.

In this position paper we describe the motivation for and initial design of such an operating system. A major constraint placed on the design is that it must be implementable on conventional architectures (for example Sun workstations and Intel 80386/486 based PCs). This decision was made for the following reasons:

- 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.

Published in: *Proceedings, Second International Workshop on Object Orientation in Operating Systems*, Dourdan, France, ed L.-F. Cabrera and E. Jul, IEEE, pp. 81-86, 1992.

- Should commercialisation become a possibility in the future, a totally software platform is easier to market than a solution including specialised hardware.

The effects of this constraint are that:

- Addresses are typically a maximum of 32 bits long.
- There is no hardware support for object addressing and protection.
- The only memory management hardware available is based on fixed sized pages.

In the next section we place the Grasshopper project into perspective in relation to other projects and indicate the major aims in terms of flexibility and experimentation. This is followed by a description of the computational model and the basic abstractions over memory, computation and naming supported by the system. We conclude with a discussion of the issues still to be addressed.

2. Related work

Attempts to build persistent systems above conventional operating systems have encountered difficulties, caused in part by the fact that these operating systems were not designed with persistence in mind. For example, the developers of Napier88 [16] have experienced unpredictable performance degradation apparently caused by bottlenecks in Unix. In-depth knowledge of the internal structure of the operating system is required to make any substantial performance improvements.

Even researchers using 'open' operating systems such as Mach [1] have experienced difficulties brought about by the inability to control certain operating system functionality. For example, the distributed Napier group [23] had problems with lack of control over page discard.

Other projects such as Monads [19] have taken the opposite approach and have developed purpose-built hardware and a new operating system which operates above it. This approach shows much promise in that appropriate architectures can be provided thus avoiding the constraints mentioned above. However, there are major disadvantages, such as the cost of development of new hardware, the time this development takes, and the inability to make the operating system and persistent applications readily available to other research groups.

An intermediate approach is to develop a new operating system with provides support for persistence, but which operates on top of *conventional* hardware. Clouds [6] and to a lesser extent (since persistence was added later) Choices [3] are examples of such systems. The major problem with existing systems which implement this approach (in particular Clouds) is that they force a narrow model of persistence on the applications which run above them. This precludes their use as a basis for experimentation with more general persistence paradigms.

3. Aims and goals of this project

In this project we aim to design and implement an operating system kernel which provides generic mechanisms capable of being tailored to support a wide variety of persistence paradigms,

In this sense Grasshopper has similar aims to the micro-kernel projects [1, 4], but with a specific orientation towards support for persistence.

4 The Basic abstractions

An operating system is largely responsible for the control and management of two entities: *objects*, which contain data and *processes*, the active elements which manipulate these objects. One of the most important considerations in the design of an operating system is the model of interaction between these entities. There are two principal models of computation which we call the *out-of-process* and the *in-process* models.

The out-of-process model aligns processes with objects. Communication between processes aligned with different objects is achieved through the use of messages. The in-process model provides for processes which move between objects. Processes access objects by *invoking* them. The invoke operation effectively causes a procedure call to code within the object. By executing this code the process may access other data stored within the object. It is interesting to note that, whilst the out-of-process model cannot be used to efficiently simulate any other computational models, the in-process model is more flexible. For instance, it can easily simulate the out-of-process model by associating a message passing process with every object. The in-process model can also be used to implement other addressing paradigms, for instance distributed memory [10].

Protection of data is more complex with the use of the in-process model. When a process invokes an object its addressing environment is defined by that object. Thus the addressing environment of a process may change throughout its life. As a result the host machine's addressing environment must be changed, not only on each process switch, but also on each invoke operation. This is in contrast to the out-of-process model, in which the addressing environment of a process is constant. Whilst we prefer the flexibility inherent in adoption of the in-process model for this project, we recognise that provision of strong protection mechanisms is another important consideration in developing our design.

Objects in the Grasshopper system are called *containers*. Persistence of containers is by reachability, that is a container persists whilst it is referenced by some other persistent entity. Our concept of process is called a

locus. Loci move between containers in accordance with the in-process model. Control over access to containers is achieved through the use of *capabilities* [8]. A locus must present a valid capability in order to invoke a container.

4.1 Containers

Containers are persistent entities which abstract over storage and thus provide the environment in which loci execute. A container may consist entirely of its own data, or may (recursively) have one or more regions of other containers mapped onto parts of it. Information about such mappings is maintained by the kernel.

A locus executing within a container accesses the data stored in it using *container addresses*. The container address of a word of data is its offset relative to the start of the container in which it is accessed. Every container is paged and may have an associated *manager* which is responsible for:

- provision of the pages of data stored in the container,
- implementation of a stability algorithm for the container [2, 12, 13, 15, 18], and
- maintenance of coherence in the case of distributed access to the container [9, 14, 17, 20].

Containers consisting entirely of mapped regions need not have a manager.

Managers are ordinary programs which reside and execute within their own containers. A manager is invoked whenever the kernel detects a memory access fault on an access to data stored in the container it manages. It should be noted that the manager for a mapped region is the manager for the container which contains the region *with no mappings*. This is shown in figure 1. In this figure, Container 1 has the shaded region from Container 2 (which itself has no mappings) mapped onto it. Access faults for the shaded pages are handled by Manager 2, while those for the other (non-shaded and unmapped) pages are handled by Manager 1.

The mappings described above are defined on a global basis, and as such are visible to all loci executing in the mapped container. Grasshopper also provides *per locus* mappings. Per locus mappings are only visible to one locus. This feature allows multiple loci to concurrently map regions such as their individual stacks in the same range of container addresses.

Whilst default managers are provided with the system, experimentation is supported by allowing users to develop their own managers. Since managers have full control over the mapping between container addresses and the backing store it is possible to provide a modified view of the container data. For example, a manager could implement large persistent addresses using a pointer swizzling technique such as those described in [2, 5, 22, 24].

4.2 Loci

Loci are the active entities in the Grasshopper system. Each locus must at any instant be executing within a container, but may meander through the containers in the store, moving from one container to the next by invocation. It is possible for multiple loci to simultaneously execute within the same container. The invoke operation is in effect a procedure call to a well known location in another container. The kernel optionally maintains a history of invoke operations so that a corresponding return may take place.

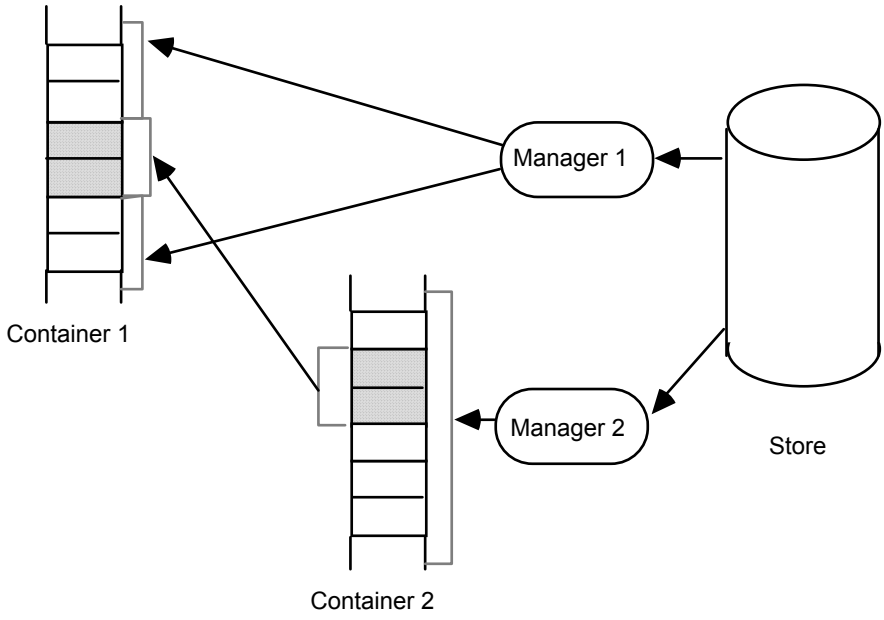

Figure 1. Provision of Pages for Mapped Containers.

Support for persistent processes [11] is provided by allowing the state of a locus to be captured within a persistent object.

4.3 Protection

Access to any entity in the Grasshopper system is obtained by the presentation of a key called a capability [8]. The capabilities indicate the type of access permitted. This varies depending on the kind of entity. For example, access rights in a container capability include the right to invoke the container, the right to map the container into a another container, the right to map other containers into a container, etc. Access rights in a locus capability include the right to suspend the locus, the right to activate the locus, the right to delete the locus, etc. Capabilities are themselves protected through their storage in segregated sections of entities.

5 Summary

The Grasshopper system provides support for three basic abstractions, namely containers, loci and capabilities. The storage abstraction provided by containers both defines an addressing environment for loci and simplifies sharing of data between loci through the use of container mappings. By associating a manager with a container, the system facilitates the implementation of container-specific stability protocols, distributed coherence protocols, and unconventional addressing techniques such as those involving pointer swizzling.

The in-process model of computation adopted by the project allows a locus to move naturally between containers without the overhead of accompanying process switches. Use of the in-process model is also expected to simplify the implementation of distributed access to containers.

The basic abstractions and associated operations supported by the system provide sufficient flexibility to allow for experimentation with a variety of persistence paradigms. Issues still to be addressed include flexible support for synchronisation, distribution, exception handling and debugging. It is hoped to report on progress in these areas in the near future.

References

1. 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*, pp. 93-112, 1986.
2. Brown, A. L. "Persistent Object Stores", Universities of St. Andrews and Glasgow, Persistent Programming Report 71, 1989.
3. 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.
4. Chorus Systemes "Overview of the CHORUS© Distributed Operating Systems", CS/TR-90-25.1, 1991.
5. 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.

6. Dasgupta, P., LeBlanc, R. J. and Appelbe, W. F. "The Clouds Distributed Operating System", *Proceedings, 8th International Conference on Distributed Computing Systems*, 1988.

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

8. Fabry, R. S. "Capability-Based Addressing", *Communications of the A.C.M.*, 17(7), pp. 403-412, 1974.

9. Henskens, F. A. "A Capability-based Persistent Distributed Shared Memory", PhD Thesis, University of Newcastle, N.S.W., Australia, 1991.

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

11. Keedy, J. L. and Vosseberg, K. "Persistent Protected Modules and Persistent Processes as the Basis for a More Secure Operating System", *Proceedings of the 25th Hawaii International Conference on Systems Sciences*, vol 1, IEEE, Hawaii, USA, pp. 747-756, 1992.

12. 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*, pp. 99-109, 1990.

13. Lampson, B. "Distributed Systems - Architectures and Implementation", *Lecture Notes in Computer Science*, vol 105, Springer-Verlag, pp. 250-265.

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

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

16. 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.

17. Philipson, L., Nilsson, B. and Breidegard, B. "A Communication Structure for a Multiprocessor Computer with Distributed Global Memory", *10th International Symposium on Computer Architecture*, vol 11(3), Stockholm, Sweden, ACM, pp. 334-340, 1983.

18. 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.

19. Rosenberg, J. and Keedy, J. L. "Object Management and Addressing in the MONADS Architecture", *Proceedings of the 2nd International Workshop on Persistent Object Systems*, Appin, Scotland, 1987.

20. Tam, M., Smith, J. M. and Farber, D. J. "A Taxonomy-based Comparison of Several Distributed Shared Memory Systems", *Operating Systems Review*, 24(3), pp. 40-67, 1990.

21. Tanenbaum, A. S. "Operating Systems: Design and Implementation", *International Editions*, Prentice Hall, 1987.

22. Vaughan, F. and Dearle, A. "Supporting Large Persistent Stores Using Conventional Hardware", *Proceedings of the 5th International Workshop on Persistent Object Systems*, San Mineato, Italy, Springer-Verlag (to appear), 1992.

23. 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*, pp. 123-140, 1990.

24. 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.