Structuring the Kernel as a Toolkit of Extensible, Reusable Components

Christopher Small and Margo Seltzer
{chris,margo}@eecs.harvard.edu
Division of Applied Sciences
Harvard University
Cambridge, MA 02138

Abstract

Applications often require functionality that is implemented in the kernel, but is not directly available to the user level. While extensible operating systems allow kernel functionality to be augmented, we believe that the emphasis on extensibility is misplaced. Applications should be able to reuse kernel code directly and the emphasis should be placed on designing a kernel with that reuse in mind. The advantage of structuring the kernel as a set of reusable, extensible tools is that applications can avoid re-implementing functionality that is already present in the kernel. This will lead to smaller applications, fewer lines of total code, and a more unified computing environment that will be easier to maintain and perform better.

1 Introduction

Many applications share functionality with the operating system kernel. For example, databases and file systems manage persistent data, arbitrate shared access, and sometimes implement transactions. File systems and many applications (e.g. text editors, word processors) restore data to a consistent state after failure. Multimedia applications perform scheduling and synchronization. We can simplify application development and avoid the duplication of effort that plagues software development by allowing applications to reuse and extend kernel components. The key capability that is required to provide this reuse is to support *fine-grained*, per-operation extensibility. This per-operation extensibility allows applications to override the kernel's policy decisions (i.e. how to use allocated resources), without affecting the kernel's ability to make global allocation decisions. Addition of new algorithms and support for new resource types can be accomplished by adding new components to the kernel.

2 Motivation and Model

The VINO kernel [13], under development at Harvard University, is structured as a toolkit of classes that implement core system behavior. Instances of these classes can be incrementally specialized for application-specific purposes by specifying policies for resource use, providing fine-grained extensibility. New classes can be derived from the toolkit classes to extend kernel functionality, providing coarse-grained extensibility. We find that there are three forms of kernel augmentation: replacement of resource policy decisions, subsystem reusability, and the addition of new functionality. Each of these forms implies a model of fine-grained modification of kernel operations.

First, implicit decisions concerning resource use are embedded in many system components, e.g. cache replacement strategy, scheduling decisions, and file read-ahead policy. Kiczales et al. [8] point out that abstracting the implementation of the kernel as a black box hides these policy decisions. Policy-making is often a zero sum game; a policy that benefits one application often harms some other application. When policy decisions are left to the kernel, it must select a least-common-denominator policy that harms all applications the least rather than one that benefits any particular application the most. A better solution is to allow each application to select its own policy and have the kernel mediate between the different policies. We are not proposing that applications control global *allocation* decisions; the kernel retains control over global allocation, and the application controls how the resources are *used*.

In the VINO kernel, all policy decisions are implemented as application-replaceable operations. This may seem unnecessarily complex, but many policy decisions are simply priority decisions: which page or buffer should be evicted, which process or thread should be scheduled next. These decisions require lit-

tle or no access to global kernel state and in many cases, the policy can be specified algorithmically (e.g. LRU, MRU, round-robin) or can be driven by data supplied by the user-level application via shared memory. For example, database systems often know what blocks they will be reading next, but the blocks do not necessarily follow an ordering that is apparent to the operating system. Rather than using the kernel's default read-ahead policy that attempts to detect sequential access and read the next sequential block, the database could place a list of disk block addresses in shared memory and let the kernel reference this list when it is ready to issue a read-ahead request.

The second form of kernel augmentation is subsystem reusability. Kernel subsystems such as caching, logging, persistent storage, and transaction management should be directly reusable by applications. Most systems implement file system recovery (e.g. replaying the log in a journaling file system), but applications must rewrite their own recovery system from scratch. Instead, the logging and recovery subsystem of the kernel should be exported so that it may be reused directly or augmented slightly and reused by applications. Recovery is a particularly good example where reuse is of paramount importance. Research indicates that recovery code is notoriously difficult to implement and debug [14]. Because it handles exception cases, it is exercised infrequently, yet in order to provide system stability, it is essential that it be correct. Rather than requiring that each application implement this crucial functionality from scratch, it should be implemented once, gotten right, and reused.

The third form of kernel augmentation is the ability to add new functionality at the right level of granularity. Current extensibility models (e.g. the external pagers of Mach, the VFS layer of 4.4BSD) offer only coarse-grained extensibility; a pager or filesystem must be replaced as a whole. Instead, the kernel should be decomposed into smaller, easily combined, easily extended classes. For example, most relational databases store relations and tuples in files provided by the file system or by implementing a storage system on a raw disk device. Instead, the kernel classes that implement directories and files should be structured in a way that would allow the kernel's file system to be augmented and reused to support the format of the database system. An RDBMS "filesystem" could then be created by reusing the standard buffer-cache code (with an application-specific eviction policy), the standard naming and directory code, application-specific files (with the read-ahead operation taking advantage of index information), and the system logging and transaction facilities.

3 Kernel Classes

The kernel should be decomposed into reusable classes that do not make assumptions about the data they manipulate. Indexing and caching classes need not know the details of the data they reference; file storage systems should remain ignorant of the implementation of their indexing structures. Policy decisions must be made explicit in such a way that they can be replaced without perturbing the rest of the implementation. The Kernel Classes provide for the basic functionality of VINO. They are designed with reuse in mind, so that they can be combined and extended to build new services.

- *Storage:* a Storage instance accepts a block of data for storage and retrieves it on demand. Storage can be backed by a disk, a virtual memory object, a File (which itself would be backed by a Storage instance), or a Log. The implementation of a disk-based Storage class is concerned with efficient layout; a Log requires periodic truncaton or archival.
- *Cache:* the interface to a generic cache consists of a fault operation, a writeback operation, and a select-victim operation. Interfaces for marking entries used, dirty, and invalid allow the client to pass information to the select-victim operation. This model supports using physical memory as a cache for virtual memory or building a buffer cache for a Storage object.
- *Name Directory:* a Name Directory maps a name to an object. To support hierarchical naming, the object can itself be another Name Directory. A general-purpose naming system is useful not only for naming files, but also elements in a database schema, environment variables, and X11 resources. Name Directory objects can be used to construct a tree, a directed acyclic graph, or a more general directed graph.
- *File:* a File consists of an indexing structure that maps file-relative addresses to disk blocks and write and read operations to move data to and from the store. The File class can be specialized by changing the indexing structure (to use an inode, B-tree, hash table, or extent-based index), the read and write operations (for automatic compression and decompression), or the open and close operations (for implementing access-control lists).
- *Log:* a Log can be volatile (backed by physical or virtual memory) or persistent (backed by disk or non-volatile RAM). A database management

system would create a Log that spans disk and archive media (e.g. tape), while the kernel would use an ephemeral log to support transactions on kernel data structures. A *log sequence number (LSN)* is assigned to each piece of data added to the log; sequence numbers are used to allow the log to be replayed in LSN order. The *checkpoint* operation flushes the log and reclaims unneeded records.

- *Transaction:* the general-purpose transaction class supports the standard `begin`, `commit`, and `abort` interface. A Transaction instance is provided with references to log entries and acquired locks; at commit or abort time, the transaction has the information needed to commit or abort changes. The implementation for ephemeral data uses a simple in-memory shadow copy scheme; an implementation that uses a persistent log and persistent copies can be used for persistent data. By subclassing Transaction, extended transaction models can be supported.
- *Schedule:* a Schedule is responsible for sharing CPU time among a group of processes. A simple schedule performs round-robin allocation; more complex feedback scheduling mechanisms are implemented using the standard schedule interface. By default, processes are added to the global Schedule, which uses a global priority scheme. A group of processes can control their relative priorities by creating a specialized Schedule, and then linking their Schedule to the global one. For example, when a client is blocked on a request to a server, it would prefer that the server get its CPU allocation, in order that the request complete more quickly. A client-server application would construct a Schedule that used a delegation algorithm, where the server would normally get little or no allocation; when a client makes a request of the server, the client's time quanta would be delegated to the server while the request is pending.

4 The VINO Augmentation Model: Grafting

VINO places extensions in the kernel, rather than leaving them in user space. Research has shown that the cost of crossing protection boundaries (between user and system, or user-system-user, as in the case of a Mach external server) is very high [3]. By placing extension code in the kernel, we decrease the number of protection boundary crossings, improving performance.

Placing extensions in the kernel can potentially compromise system integrity in three ways.

- Violating address space boundaries: a major advantage of running application code in a separate address space is that it can not read or modify kernel structures.
- CPU hogging: once invoked, a kernel extension might run forever, starving all other processes.
- Failure: errors can cause a kernel extension to leave kernel data structures in an inconsistent state.

There are at least three ways to address the potential integrity violations of kernel extensions. Some extensible systems (e.g. SPIN [4] and Thor [10]) use type safe languages for writing kernel extensions. Others (e.g. HiPEC [9]) use specialized, interpreted languages that are limited to the specific extension domain. VINO uses software fault isolation techniques such as *sandboxing* [15], which allow grafts to be written in conventional languages, such as C. Sandboxing forces all memory operations (read, write, and jump) to fall within the address space boundaries allocated to the grafted code. By refusing to download any code that has not been sandboxed, we can avoid address space violations.

We expect policy grafts to be fairly small; for short sequences of code, the runtime overhead of software fault isolation can be much lower than that of kernel-user protection-domain crossings. The overhead of sandboxing has been shown to be substantially lower than that of interpreted languages, which typically have a slowdown factor of one to two orders of magnitude [11].

We ensure that grafted code does not monopolize the CPU by preempting and time-slicing long-running grafts. A graft that is called in the context of an application (e.g. in read-ahead code) that runs for an extended period of time blocks the progress of only the grafting application. A graft that does not terminate in a reasonable amount of time is aborted.

To address failure, a graft is run in the context of a transaction, using the standard kernel transaction support. If grafted code fails or is aborted, the enclosing transaction is aborted, causing graft resources to be deallocated and modifications to kernel structures to be undone, using compensating operations. The kernel then kills the process that installed the graft, or resumes the operation, using its default implementation.

5 Related Work

Cao et al. [5] separate buffer cache allocation decisions from use decisions, showing that with only two

policy choices (LRU vs. MRU) application performance can be significantly improved. VINO's extension model provides this algorithmic selection as well as other, more complex, decision models.

Scheduler Activations [1] are lightweight threads for multiprocessors that leave global processor allocation decisions to the kernel, but move thread scheduling to application control. Threads are a user-level construct, and their scheduling is performed entirely at the user level. This is implemented in a kernel extension model by allowing the set of processes that implement the multi-threaded task to implement their own scheduling algorithm.

The Mach external pager interface allows pagers to control how backing store is used, but not which pages to evict. McNamee and Armstrong [12] extended the model to allow external pagers to control page eviction. The level of granularity is still coarse; implementing a different paging algorithm implies implementing a new pager. We argue for a fine-grained model that allows any particular algorithm to replaced in the context of an existing pager.

The SPIN extensible microkernel [4] allows client code to be dynamically added to the kernel, although the focus is on extensibility rather than reuse. Anderson's proposal [2] to implement extensibility by moving functionality out of the kernel and up to the application level, also seen in Lipto [6] and Aegis [7], allows for structuring system services as a library of reusable classes.

References

[1] Anderson, T., Bershad, B., Lazowska, E., Levy, H., "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism", *Proceedings of the 13th SOSP*, pp. 95–109 (1991).

[2] Anderson, T., "The Case for Application-Specific Operating Systems", *Proceedings of WWOS III* (1992).

[3] Bershad, B., Anderson, T., Lazowska, E., Levy, H., "Lightweight Remote Procedure Call", *Proceedings of the Twelfth ACM Symposium on Operating System Principles,* (1989).

[4] Bershad, B., Chambers, C., Eggers, S., Maeda, C., McNamee, D., Pardyak, P., Savage, S., Sirer, E., "SPIN - An Extensible Microkernel for Application-specific Operating System Services", University of Washington Technical Report 94-03-03 (February 1994).

[5] Cao, P., Felten, E., Li, K., "Implementation and Performance of Application-Controlled File Caching", *Proceedings of the First OSDI*, pp. 165–178 (November 1994).

[6] Drushel, P., Peterson, L., Hutchinson, N. C., "Beyond Microkernel Design: Decoupling Modularity and Protection in Lipto" *Proc. 12th Int. Conf. on Distributed Computing Systems*, pp. 512–520, Yokohama, Japan (June 1992).

[7] Engler, D., Kaashoek, M., O'Toole, J. Jr., "The Operating System Kernel and a Secure Programmable Machine", *Proceedings of the Sixth SIGOPS European Workshop* (September 1994).

[8] Kiczales, G., Lamping, J., Maeda, C., Keppel, D., McNamee, D. "The Need for Customizable Operating Systems", *Proceedings of the Fourth Workshop on Workstation Operating Systems*, Napa, CA (August 1993).

[9] Lee, C-H., Chen, M. C, Chang, R-C., "HiPEC: High Performance External Virtual Memory Caching", *Proceedings of the First OSDI*, pp. 153–164 (November 1994).

[10] Liskov, B., Day, M., and Shrira, M., "Distributed Object Management in Thor", in *Distributed Object Management*, Morgan Kaufmann, San Mateo, California (1994).

[11] May, C., "MIMIC: A Fast System/370 Simulator", *Proc. SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, published as *SIGPLAN Notices 22*, 7, St. Paul, MN (July 1987).

[12] McNamee, D., Armstrong, K. "Extending the Mach External Pager Interface to Accommodate User-Level Page Replacement Policies", *USENIX Mach Workshop*, pp. 17–29 (October 1990).

[13] Seltzer, M., Endo, Y., Small, C., Smith, K., "An Introduction to the VINO Architecture," In "VINO: The 1994 Fall Harvest," Harvard University Technical Report TR-34-94.

[14] Sullivan, M., and R. Chillarege, "Software Defects and Their Impact on System Availability – A Study of Field Failures in Operating Systems", *Digest 21st International Symposium on Fault Tolerant Computing* (June 1991).

[15] Wahbe, R., Lucco, S., Anderson, T., and Graham, S., "Efficient Software-Based Fault Isolation", *Proceedings of the 14th SOSP*, Asheville, NC (December 1993).