VINO: The 1994 Fall Harvest

Yasuhiro Endo
James Gwertzman
Margo Seltzer
Christopher Small
Keith A. Smith
and
Diane Tang

TR-34-94

December 1994

Center for Research in Computing Technology
Harvard University
Cambridge, Massachusetts

VINO: The 1994 Fall Harvest

Yasuhiro Endo
James Gwertzman
Margo Seltzer
Christopher Small
Keith A. Smith
Diane Tang

An Introduction to the Architecture of the VINO Kernel

Margo Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith
Harvard University
{margo,yaz,chris,keith}@das.harvard.edu

Abstract

Current operating systems are designed to provide least-common-denominator service to a variety of applications. They export few internal kernel facilities, and those which are exported have irregular interfaces. As a result, resource intensive applications such as database management systems and multimedia applications, are often poorly served by the operating system. These applications often go to great lengths to bypass normal kernel mechanisms to achieve acceptable performance.

We describe a new kernel architecture, the VINO kernel, which addresses the limitations of conventional operating systems. The VINO design is driven by three principles:

- *Application Directed Policy:* the operating system provides a collection of mechanisms, but applications dictate the policies applied to those mechanisms.
- *Kernel as Toolbox:* applications can reuse the kernel's primitives.
- *Universal Resource Access:* all resources are accessed through a single, common interface.

VINO's power and flexibility make it an ideal platform for research in operating systems and resource intensive applications.

1 Introduction

Conventional operating systems provide a fixed interface with a predefined set of policies implementing that interface. The policies provided are the least common denominator of those required by applications. When the policy provided by the operating system is inappropriate for a particular application, there are two alternatives: leave the application to suffer with an inappropriate policy or reimplement the kernel mechanism in user space. The first approach leads to degraded performance; the second leads to redundant implementations and competition for resources between the operating system and the application.

Inappropriate policy is only one reason that applications reimplement kernel functionality. Kernel functionality is often unavailable to applications. For example, the kernel uses efficient synchronization primitives based on fast test and set instructions, which are not available to applications [AT&T]. Modern file systems use logging to provide improved performance and fast recovery, but these logging mechanisms are not available for application use [CHANG90, CHUT92, KAZAR90, VXFS].

Today's applications are unable to realize the potential of today's hardware [OUST90]. Database management systems are the classic example of competition between applications and the operating systems [STON81]. However, they are only one example; real-time systems, high-speed networking applications, distributed applications, and embedded systems all face similar problems. We call these applications *resource intensive*, as they place heavy demands on the allocation of resources, by virtue of the size of the resources (e.g. images and video) or the timing requirements of the resources (e.g. audio-video synchronization and quality of services guarantees). Today's systems address these problems with piecemeal solutions.

The VINO kernel focuses on three key ideas:

- *Applications direct policy:* the kernel controls *allocation* of resources, but leaves the *management* of those resources to applications. For example, the kernel is responsible for determining the allocation of physical page frames to processes, but each process then determines which virtual memory pages will be mapped into physical memory.
- *Kernel as a set of reusable tools:* rather than hide kernel mechanisms from applications, they are exported for application reuse. Many filesystems use a database-style logging of metadata operations to improve performance and simplify recovery; many applications do as well. Normally, an application needs to reimplement the logging facility in user space. In VINO, this facility is available for application use.
- *All resources share a common, extensible interface:* to simplify reuse of kernel services, we support a simple hierarchical type model for resources; any facility that works with a general resource will work with a

more specific one. A locking subsystem, written to work with the generic resource type, will work with any resource, be it a file, VM page, or serial port.

The VINO architecture consists of an inner kernel and a set of *application resources*. The inner kernel can not be modified by application code, but a process can override the behavior of its application resources. The inner kernel controls *allocation* and *security* decisions – it mediates requests for shared resources, and ensures that a process does not illegally gain access to resources.

2 Resource Types

Each VINO *resource* is described by its *resource type*. The resource type definition includes *operations* (function members, message handlers) and *properties* (data members, slots). A resource type includes a default implementation for each operation (a piece of code that is run when the operation is invoked).

The standard set of resource types include resources such as files, directories, threads, transactions, physical memory pages, virtual memory pages, and queues.

Resource types are arranged in a hierarchy. A subtype inherits the interface of its supertype, and can reuse or override its supertype's implementation. The subtype can add new operations and properties, extending its supertype's interface. It can not remove properties or operations from the inherited interface. For example, a QuickTimeFile resource type is defined in terms of the VideoFile resource type (which is defined in terms of the File resource type). It has the same basic interface, but includes different implementations for the encode and decode operations.

A new resource type is added to VINO by compiling it into the kernel. If the new resource type has the same interface as its supertype, or the new interface does not need to be called from existing kernel code, the new type can be added to the kernel dynamically, as new device drivers can be linked into Unix[1] on-the-fly. If the new interface needs to be called from existing kernel code, the kernel will need to be relinked.

3 Grafting

Application control of policy is accomplished by overriding the default implementation for an operation on a resource. In VINO, this is called *grafting* a new implementation into the kernel. For example, the PageTable resource allocated to a process uses an LRU algorithm for its eviction strategy. If an application wants to use a different algorithm, it *grafts* its own implementation for the Evict operation onto the PageTable resource for the process.

[1]Unix is a trademark of X/Open.

These resource-specific implementations are modules that are dynamically installed in the kernel. We install this code in the kernel because the cost of frequent cross-domain calls is too high, especially on performance-critical paths such as when policy decisions are made [BERS89].

Unlike other extensible systems, we have not undertaken the task of defining a new typesafe language [BERS94, ENGL94, LISK93]. It is outside the scope of our project to specify, implement,and support a new language, and widespread acceptance of new languages in the community, irrespective of their elegance and power, is very low. Extensions to VINO are written in C or C++.

Techniques to ensure the safety of object code are well-known. Each module is assigned a range of memory for its code and data segments. Instructions are inserted into the code to perform a base-and-bounds check on each memory reference. This type of check detects faults in code; an alternative technique, *sandboxing* [WAHBE93], masks and prevents faults, with an overhead lower than that of base-and-bounds checking.

Our plan is to a trusted compiler that generates code with either bounds checking or sandboxing to ensure code safety. Code generated by our compiler will be marked with a *fingerprint* [RABIN81] (a type of digital signature). A fingerprint is computationally infeasible to forge; it ensures (with a very high degree of certainty) that all code installed in the kernel comes from our trusted compiler.

Our compiler, based on compiler back-end work underway at Harvard, ensures that code grafted onto the operating system does not read or write outside its bounds, includes no instructions that mask interrupts, and does not modify itself.

Even with these assurances, user-installed code may not terminate in a timely fashion. The VINO kernel supports limited multithreading, and grafted code can be descheduled by timing out. The grafted code may be ill-behaved and never return to the application, but only the application itself suffers; no other process is prevented from making progress.

We must also guard against grafted code obtaining a critical system lock and not releasing it in a reasonable amount of time. To handle this, we attach a *time-out* to critical locks, and kill a process that does not release the lock before the time-out. Each piece of grafted code runs in the context of a lightweight transaction that keeps track of its allocated resources. If the process is aborted, the corresponding transaction is aborted, and the system is returned to a consistent state.

Unlike the external servers of Mach [ACET86], grafting allows small, *incremental* changes in kernel functionality. If the page eviction strategy of the system is inappropriate, it can be replaced without writing a new external pager [MCNAM90].

4 Resource Managers and Names

A *name service* maps a name to a *(resource manager, storage-id)* pair. The resource manager can then be asked to map the storage-id to a *file resource*. A file resource implements the expected read, write, and seek interface.

Because the name service is decoupled from the storage system, we can put files next to each other in the namespace that are stored in different places. For example, in `/home/chris` you find the entries `time`, `vino-arch.tex`, and `to-do`. The first is be handled by a time resource manager; when read, it responds with the current time. The second is a "regular" file, stored in a local filesystem. The third, when read, sends a query to the calendar database, and return the contents of the reader's to-do list.

This facility is similar to one offered by Plan 9 [PRES90], although because VINO separates name management from storage management, it gains the flexibility of allowing services to be located in the namespace where it makes most sense to the user. We also stray from the idea of using the filesystem namespace as the single unifying abstraction; not all resources can be easily modeled as files, or need to be present in the file namespace.

A resource manager is an instance of the resource type *manager*. A manager provides operations to create and delete entries, and control access to its stored data. It also implements the management of the underlying storage (read and write operations) for its files. Subtypes of *manager* include one implementing an FFS-style [MCKU84] file system, a journaling file system, an NFS file system, and a memory-based file system. Defining another subtype of manager (e.g. one that handles FTP requests) is straightforward.

Local disk storage is controlled by a *volume manager* which owns the physical disk; other storage managers request cylinders and tracks from it. By using a volume manager we are able to dynamically partition the amount of space allocated to different managers.

Stackable or layered file systems are implemented by building on top of an existing resource manager. If an encrypted file system is needed, a new manager is created with *read* and *write* operations that encrypt and decrypt data, and then delegate storage to an already existing resource manager.

5 Fairness

One of the primary jobs of an operating system is to arbitrate and abstract resource access. Some devices, such as physical memory, are shared and preemptable; others, such as disk space or a serial port, are not.

Some applications require service guarantees, e.g. an application displaying real-time video using a double-buffered display needs to be scheduled thirty-two times a second and have physical memory large enough to hold two copies of the displayed image. A query processor can tune its join algorithm to the amount of physical memory available for its use, if it can assume that the memory, once allocated, will not be taken away. Such applications can make *hard resource requests*, where no less than the minimum requested resources will be allocated, and once allocated, they will not be preempted. If a new *hard request* will exceed the physical resources of the system, VINO will not grant the request. A hard request can be thought of as application-specified entrance criteria; if the resource can not be allocated, the application can choose to not proceed. In order to ensure fairness of allocation, an application must be privileged in order to make hard requests. Applications with less stringent requirements make *soft requests*, specifying a preferred minimum resource allocation. If the sum of the soft requests exceeds the system resources, VINO will arbitrate between the requesters, sharing the resources available.

6 Kernel Tools

Operating systems are built around synchronization, transactions, recovery, and resource sharing. The code that implements this functionality is rarely exported to user applications. The VINO design is based on the idea that kernel tools should be exported to and used by applications.

6.1 Synchronization Primitives

The operating system's synchronization primitives are typically much simpler and more efficient than those provided to user-level applications. For example, the semaphores offered to applications by System V incur a large number of system calls and context switches while simple spin-locks are virtually free [SELT92]. VINO provides a kernel lock manager, accessible for application use.

In its simplest form, the lock manager provides spin-lock synchronization on memory locations, requiring kernel intervention only in the case of a contested lock. This interface is available both to the kernel and to applications.

As the resources being locked become more complex, so does the locking paradigm. The VINO lock manager supports *general-purpose hierarchical locking* [GRAY76]. For example, the file system typically requires locking on block, file, directory and file system levels. In most kernels, this hierarchy is enforced by convention. In VINO, it is enforced by design.

We call the levels at which locking may be needed the *containment hierarchy*. When a lock is requested from the lock manager, the manager examines the resource's containment hierarchy to determine if the lock may be granted. Applications using the lock manager can define alternate containment hierarchies, and make them available to the lock manager (e.g. a DBMS might create a logical containment hierarchy of database, relation, tuple, and field). Additionally, applications can create new instances of a lock manager that enforces alternate locking protocols (e.g. alternate deadlock handling, blocking vs. non-blocking, or new locking modes).

Finally, by integrating the kernel and user level locking systems, concurrency can be increased. For example, a DBMS running on a conventional Unix file system may implement its own lock manager and issue multiple I/O requests to the same file. Unfortunately, the Unix file system exclusively locks the file during each I/O operation so that no concurrency is achieved even though the DBMS is already ensuring the integrity of the operation. In VINO, since the same lock manager handles both DBMS requests and I/O requests, locks held by the DBMS are strong enough to perform I/O and no additional locking is required by the file system.

6.2 Log Management

VINO provides a simple log management facility that is used by the file system and the transaction system, and accessible to applications as well.

A log resides on one or more physical devices. It can be created on a single device, or extended onto a second device (not necessarily of the same type as the first). For example, a DBMS might request a log that spans both magnetic disk and archive media (e.g. tape or optical disk). The kernel requests a volatile, in-memory log to support transactions on ephemeral data, such as process structures and buffer cache metadata. The file system might request a log that spans non-volatile RAM (NVRAM) and disk; file system log records would be written first to NVRAM and later written to disk in large, efficient transfers.

The key interface to the log facility is the read/write interface which provides the essential information for write-ahead logging (i.e. a *write_log* function that returns a log sequence number, and a *read_log* function that returns records in log sequence number). It also supports a *synch_wal* operation for write-ahead log synchronization and a *checkpoint* operation for log reclamation and archiving. As logs can expand and contract, the partitioning between log resources and other data resources is not static, but can change as system demands fluctuate.

6.3 Transaction Management

The VINO kernel uses transactions to maintain consistency during updates to multiple related resources (e.g. a directory and its contents). For example, the carefully ordered writes of FFS can be reimplemented as a simpler series of unordered writes, encapsulated in a transaction.

The transaction interface supports the standard *transaction-begin*, *transaction-commit*, and *transaction-abort* operations. It accepts references to appropriate log and lock manager instances to use for each transaction. At transaction begin, a new *transaction resource* is created. This resource references the appropriate log and lock managers and is referenced by each protected update. Most kernel transactions are protected using a simple shadow-resource scheme with a log residing in main-memory (either volatile or non-volatile depending on the resources being protected). The mixing and matching of logging and locking components enables VINO to support arbitrarily complex transaction protocols.

Because the implementation of the transaction manager can be incrementally modified, different transaction semantics (e.g. as outlined in [BILI94]) can be implemented as needed by applications.

6.4 Memory Management

The VINO memory management system is based on the ideas of the Mach VM architecture, although its implementation differs considerably.

A *MemoryResource* is a collection of pages. As in Mach, it is backed by a file mapped into memory. It includes operations to read pages from and write pages to that backing store. When a page fault takes place, VINO determines which MemoryResource (if any) is assigned to the virtual memory page containing the faulting address. A request is made of the MemoryResource, with the address at which to write the requested page.

Unlike a Mach pager, the MemoryResource is entirely in the kernel; when a page fault occurs (which causes a trap into the operating system), it is not necessary to go back across the protection boundary from the kernel to the user level. Also, the Mach architecture requires that a new external pager be written for each kind of behavior needed. VINO allows each MemoryResource to use as much or as little of the standard implementation as is appropriate; the application need only override or augment the operations it wants to change.

The *AddressMapResource* is patterned after the Mach object of the same. Each address space has an associated AddressMapResource, which contains a mapping between physical pages and virtual memory pages. When VINO determines that a mapping needs to be removed (either because of a virtual memory fault, or because the number of physical pages assigned to the address space is being decreased), it invokes the ChooseVictim operation defined on AddressMapResource. By default, ChooseVictim selects the least recently used page, although an application can replace the implementation of ChooseVictim with the algorithm of its choice.

Note that, as in Cao's work [CAO94], VINO retains control over the *number* of mappings allocated to an address space, but not the mappings themselves. The former behavior is not under the control of an application (in order to ensure fairness of allocation); the management of the mappings is delegated to each application.

7 Related Work

Many systems have addressed the need for flexibility. Mach [ACET86] allowed the addition of external servers, factoring the kernel into replaceable servers. Chorus [ROZI88] worked to overcome the performance problems of external servers by allowing them to be developed outside the kernel, and then moved into the kernel as a build option.

Newer systems such as Aegis [ENGL94] and SPIN [BERS94] address the granularity problems of the original microkernel architecture by allowing small, incremental changes to be made by loading user code into the server. They have also addressed the issue of safety through the use of compilation techniques.

Object-oriented toolkits are composed of a set of reusable components (e.g. NextStep [NEXT93]) that can be combined and specialized as needed.

Work has been done to address policy control on a topic-by-topic basis. Scheduler Activations [ANDE91] are a method for sharing scheduling policy between kernel and user; Cao's work on application-controlled file caching [CAO94] addresses buffer cache management. The Berkeley Fast Filesystem [MCKU84] allows file layout to be controlled by the setting of the `rotdelay`, `maxcontig`, and `maxbpg` parameters.

System V Release 4 provides for multiple classes of scheduling algorithms, corresponding to time-sharing scheduling, real-time scheduling, and kernel process scheduling. It is not possible to add new policies without completely reconfiguring and relinking the operating system, and then only if the desired scheduling algorithm fits SVR4's (limited) model of how a scheduler should behave [OLEA92].

8 Status

We are targeting the x86 and HP-PA architectures as our initial platforms. The architectural outline is complete, and we have begun prototyping the resource types, grafting technology, and compilation tools at the user level. The inner kernel (boot code and device support) is based on 4.4BSD.

As part of our development, we plan to implement a POSIX compatibility [IEEE93] library on top of VINO. We are also looking into supporting BSD binaries directly, but have not committed to it.

It is our goal that applications that do not take advantage of VINO's extensibility should run at roughly the same speed as on a 4.xBSD system.

9 Conclusion

The VINO architecture is a simple and regular, and meets our goals of application direction of kernel policy, reusable kernel tools, and a common interface to all resources.

It is not necessary to define a new language in order to safely extend kernel behavior; conventional languages can be used, when combined with a trusted compiler and software protection techniques.

We believe that by concentrating on the key ideas of extensibility and reusability, we will be able to accomplish our goals with a minimal level of distraction.

References

[AT&T] AT&T, "System V Interface Definition, Third Edition," Volumes 1–3, 1989.

[ACET86] Acetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., and Young, M., "Mach: A New Kernel Foundation for UNIX Development", Proceedings of the Summer Usenix Conference (July 1986).

[ANDE91] Anderson, T., Bershad, B., Lazowska, E., Levy, H., "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism," Proceedings of the Thirteenth ACM Symposium on Operating System Principles, Monterey CA, October 1991, 95-109.

[BERS89] Bershad, B., Anderson, T., Lazowska, E., Levy, H., "Lightweight Remote Procedure Call", *Proceedings of the Twelfth ACM Symposium on Operating System Principles,* (1989).

[BERS94] Bershad, D., Chambers, C., Eggers, S., Maeda, C., McNamee, D., Pardyak, P., Savage, S., Gun Sirer, E., "SPIN – An Extensible Microkernel for Application-specific Operating System Services," Technical Report 94-03-03, Department of Computer Science and Engineering, University of Washington, Seattle (1994).

[BILI94] Biliris, S., Dar, S., Gehani, N., Jagadish, H. V., and Ramamritham, K., "ASSET: A System for Supporting Extended Transactions", *Proceedings of SIGMOD 94*, Minneapolis, MN (May 1994).

[CAO94] Cao, P., Felten, E., and Li, K., "Application-Controlled File Caching Policies", *Proceedings of the*

1994 Winter Usenix Conference, pp. 171-182 (June 1994).

[CHANG90] Chang, A., Mergen, M., Rader, R., Roberts, J., Porter, S., “Evolution of storage facilities in AIX Version 3 for RISC System/6000 processors,” IBM Journal of Research and Development 34, 1, January 1990.

[CHUT92] Chutani, S., Anderson, O., Kazar, M., Leverett, B., Mason, W., Sidebotham, R., “The Episode File System,” Proceedings of the 1992 Winter Usenix Conference, San Francisco, CA, January 1992.

[ENGL94] Engler, D., M. F. Kaashoek, and J. O’Toole, “The Operating System Kernel as a Secure Programmable Machine”, *Proceedings of the Sixth SIGOPS European Workshop* (September 1994).

[GRAY76] Gray, J., Lorie, R., Putzolu, F., and Traiger, I., “Granularity of Locks and Degrees of Consistency in a Large Shared Database,” in *Modeling in Data Base Management Systems*, Elsevier North Holland, New York, pp. 365-394 (1976).

[IEEE93] IEEE, “Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API) [C Language]”, IEEE Standard 1003.1b, September 1993.

[KAZAR90] Kazar, M., Leverett, B., Anderson, O., Vasilis, A., Bottos, B., Chutani, S., Everhart, C., Mason, A., Tu, S., Zayas, E., “DECorum File System Architectural Overview,” Proceedings of the 1990 Sum- mer Usenix, Anaheim, CA, June 1990, 151-164.

[LISK93] Liskov, B., Day, M., and Shrira, M., “Distributed Object Management in Thor”, in *Distributed Object Management*, Morgan Kaufmann, San Mateo, California (1993).

[MCNAM90] McNamee, D., and Armstrong, K., “Extending the Mach External Pager Interface to Accommodate User-Level Page Replacement Policies,” *Proceedings of the 1990 Usenix Mach Workshop*, Burlington, VT (1990).

[MCKU84] McKusick, M., Joy, W., Leffler, S., Fabry, R., “A Fast File System for UNIX,” *Transactions on Computer Systems*, v. 2 n. 3, pp. 181-197 (August 1984).

[NEXT93] “NextStep 3.0 Users Manual”, Next Computer (1993).

[OLEA92] O’Leary, K., Wood, M., Advanced System Administration, UNIX Press, Englewood Cliffs, NJ, 1992, Chapter 8.

[OUST90] Ousterhout, J., “Why Aren’t Operating Systems Getting Faster as Fast as Hardware?” Proceedings of the 1990 Summer Usenix Technical Conference, Anaheim, CA, June 1990, 247-256.

[PRES90] Presotto, D., Pike, R., Trickey, H., and Thompson, K., “Plan 9, A Distributed System”, *Proceedings of the Spring 1991 EurOpen Conference* (May 1991).

[RABIN81] Rabin, M., “Fingerprinting by Random Polynomials”, Harvard University Center for Research in Computing Technology TR-15-81 (1981).

[ROZI88] Rozier, M., Abbrossimov, V., Armand, F., Boule, I., Giend, M., Guillemont, M., Herrmann, F., Leonard, P., Langlois, S., Neuhauser, W., “The Chorus Distributed Operating System,” Computing Systems v. 1, n. 4 (1988).

[SELT92] Seltzer, M., Olson, M., “LIBTP: Portable, Modular Transactions for UNIX”, *Proceedings 1992 Winter Usenix Conference*, San Francisco, CA, pp. 9-26 (January 1992).

[STON81] Stonebraker, M., “Operating System Support for Database Management,” Communications of the ACM, 7, July 1981, 412-418.

[VXFS] Unix System Laboratories, “The vxfs File System Type,” from Advanced System Administration for UNIX SVR4.2, 1992.

[WAHBE93] Wahbe, R., Lucco, S., Anderson, T., and Graham, S., “Efficient Software-Based Fault Isolation”, *Proceedings of the 14th SOSP*, Asheville, NC (December 1993).

The Case for Geographical Push-Caching

James Gwertzman, Margo Seltzer
Harvard University
{gwertzma, margo}@das.harvard.edu

Abstract

Most existing wide-area caching schemes are *client initiated*. Decisions on when and where to cache information are made without the benefit of the server's global knowledge of the situation. We believe that the server should play a role in making these caching decisions, and we propose *geographical push-caching* as a way of bringing the server back into the loop. The World Wide Web is an excellent example of a wide-area system that will benefit from geographical push-caching, and we present an architecture that allows a Web server to autonomously replicate HTML pages.

1 Introduction

The World-Wide Web [1] operates for the most part as a cache-less distributed system. When two neighboring clients retrieve a document from the same server, the document is sent twice. This is inefficient, especially considering the ease with which Web browsers allow users to transfer large multimedia documents.

To combat this problem, some Web browsers have begun to add local client caches. These prevent the same client from transferring the same document twice. Some networks are also beginning to add Web proxies [6, 7] that prevent two clients on the same campus network from transferring the same document twice.

The problem with both these schemes is that they are myopic. A client cache does not help a neighboring computer, and a campus proxy does not help a neighboring campus. Furthermore, these caches are usually limited in size. Disk might be cheap, but as the size of multimedia files increases it will be impossible to cache everything. As a result these caches will only be able to store the most popular items even though there is still some demand for other, less popular items.

The solution is for clients to share each other's cache space. The degree to which a file is replicated should be proportional to that file's global popularity, and clients should retrieve files from the the nearest cache to minimize network traffic. The server can satisfy both goals by deciding when and where to cache files. Furthermore, when the server decides where to cache a file, it can make this decision using its knowledge of network topology and the file's access history for even greater network bandwidth savings.

2 Motivation

Preliminary analysis of Web access logs show that a few files on each server are responsible for most traffic from that server. Servers can therefore save a great deal of bandwidth by only worrying about those few files. This fact makes server caching feasible since replicating and distributing files puts a slight load on the server. Analysis also shows that the access pattern for each file is not geographically uniform. File requests are often clustered geographically which implies that judicious selection of cache sites can provide excellent bandwidth savings.

3 Architecture

There are two components to our proposed Web system: a modified HTTP server and a replication service. The modified server is responsible for tracking geographical access information for its files and for accepting and offering cached replicas of other files. This can be done by modifying a proxy server, such as the CERN proxy server [6], to accept files for replication using a modified POST request.

The replication service keeps track of modified HTTP servers that are willing to serve replicated files, the amount of available free space on each server, and each server's average load. The replication service works with the modified server to decide where a given file should be cached.

We must minimize the amount of state that each Web server stores for its files, or else we will face scalability problems. We therefore track geographical access information in a coarse manner. We are currently using states and countries since these can easily be obtained from network addresses.

When the demand for a file exceeds a replication threshold, the server replicates it. We are using trace-driven

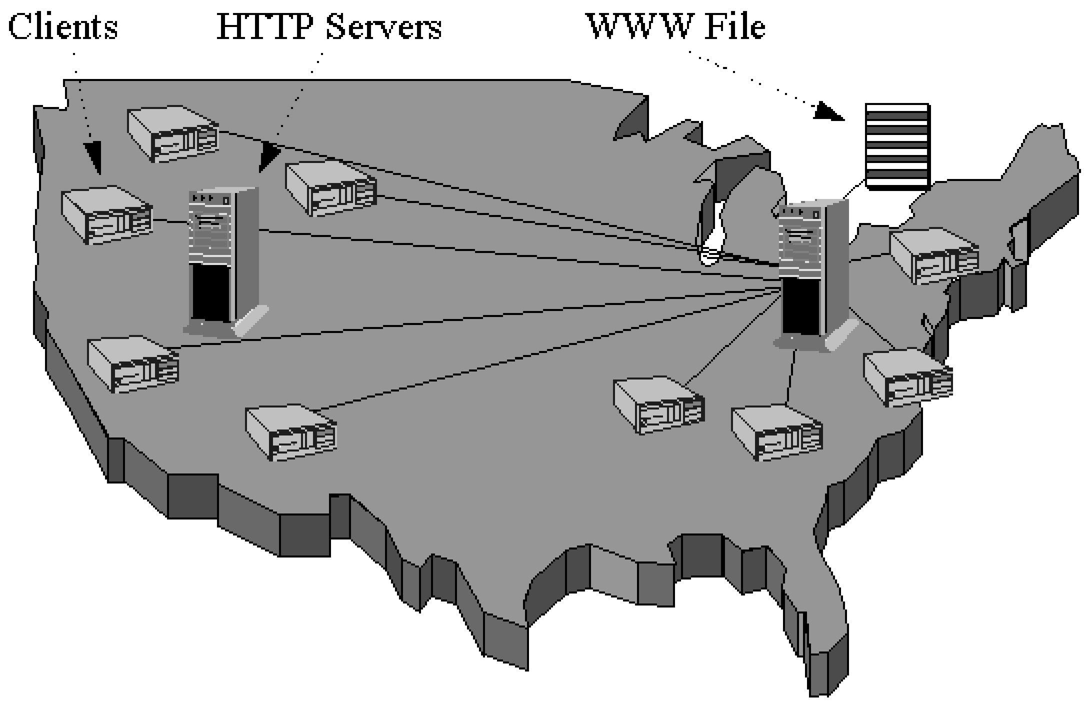

Figure 1: Before file replication takes place: several clients accessing a World Wide Web file on the east coast.

simulation to determine reasonable values for the replication threshold, and we expect it to be dynamic.

The replication service decides where to replicate the file given its access history. The goal is to pick a server to cache the file that will minimize the amount of bandwidth used in the future. We predict this by using the file's history.

Figure 1 and figure 2 illustrate the replication process in action. Several clients from across the United States are accessing a file on an east coast server. The east coast server replicates the file such that network bandwidth is minimized, and the file ends up on a west coast server.

The replication service maintains a list of all HTTP servers willing to replicate files; it must choose one of them to cache the file. We are considering several algorithms to determine the optimal cache location. If the service knows the Internet's topology it can solve the problem by finding a good solution to the corresponding graph partitioning problem or max-cut min-flow problem.

If only coarse grained information is available about the Internet, such as average latency between servers (available from `traceroute`) a better solution is to iterate over a representative sample of the available servers, calculating bandwidth savings for each. The service would then replicate the file on the server that would have reduced network bandwidth the most.

Once the primary server gives a file to another server for caching, the primary server forgets about the other server. The primary server's load will drop as clients begin to access the file from the new server. Should the primary server's load climb high enough that it must replicate the file again, the primary server will choose a different server to cache it on since the access patterns will have changed. Likewise, if the new server's load climbs high enough such that it must replicate the file, it will be cached in yet another place, because the access patterns for the new server will be very different than those for the old server.

There are two issues that must still be addressed for this scheme: file consistency and resource discovery. A server may determine that its copy of a file is out of date by using the *get-if-modified-since* HTTP request. This is an efficient way to both check consistency and to request the new file in the event it has been modified, but it is too expensive to use every time a file is requested.

Since weak-consistency should be acceptable for the Web, we are using a scheme developed for the Alex [4] file system. With the exception of dynamic pages (these will be addressed separately) we expect the Web to obey the same principle as FTP: the older a file is, the less likely it is to be modified. Therefore, the older the file that an HTTP server is caching, the less frequently the HTTP server must poll to check if its copy is still up-to-date. This is very efficient compared to checking for every request, and the client will be able to force a poll if it is essential to use the latest file.

As for resource location, there are several groups work-

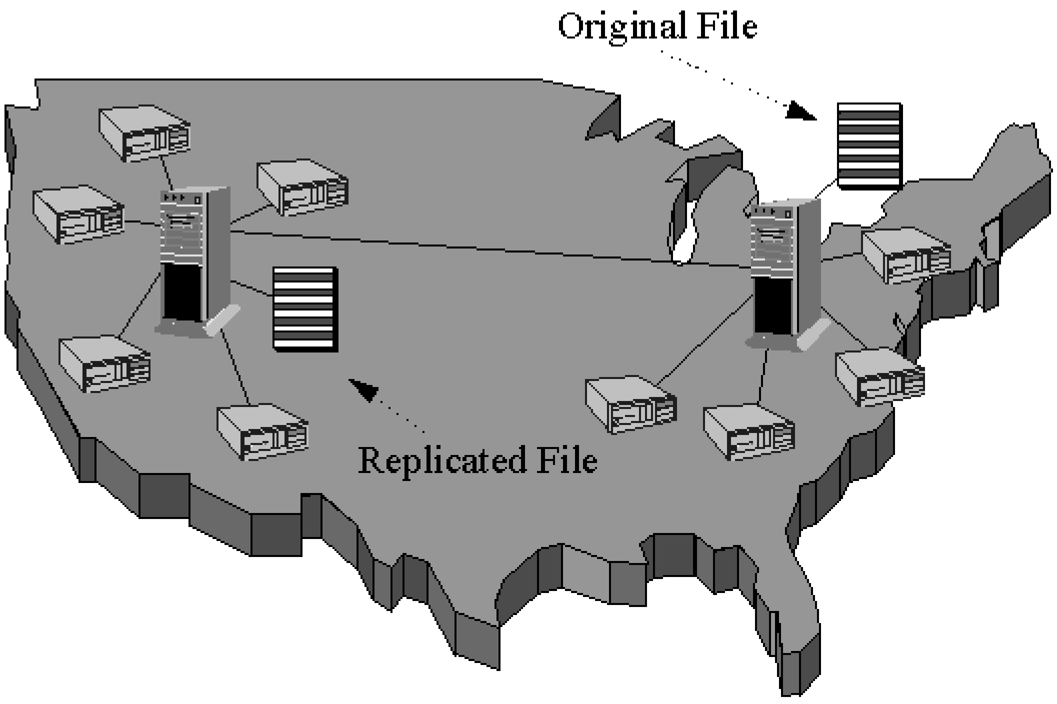

Figure 2: After file replication has taken place: the file has been replicated onto a west coast server so as to minimize network bandwidth.

ing on this problem. Until this problem is solved we are using a technique proposed by Blaze [2], which we call the "1-800 technique". Clients "call" the primary server to ask for the "server nearest you." This is not elegant, but it works because latency is currently more critical than bandwidth. The expense of querying a distant server once is amortized over the many local requests that are thereby made possible. Eventually there will need to be a cleaner solution so that all dependence on the original server can be removed. Otherwise we will face scalability and reliability problems.

4 Other Applications

Throughout this paper we have referred to HTTP servers and files. This was for the purposes of clarity, as well as to provide a focus for our research. We expect our results to be applicable to any wide-area distributed system, however; not just the World Wide Web. One application for geographical push-caching that we have in mind is to replicate not only data files but also services themselves.

A good example would be Archie [5], whose load problems are notorious. If Archie were to be written in a machine-independent network-service scripting language (e.g. Tcl [8]), its code could be replicated and cached just like a Web file. This might also be the answer to how to cache dynamic pages, such as those generated by cgi-bin scripts that are used to create Web pages on the fly.

5 Related Work

There is little work on caching in large-scale distributed systems outside of distributed file systems, since only in the past few years has the attention of the distributed systems community turned toward globally distributed systems such as the World-Wide Web and FTP. Several groups are working on similar problems, but none that we know of are working on server-initiated caching. The Harvest system [3] in particular incorporates an *object caching subsystem* that provides a hierarchically organized means for efficiently retrieving Internet objects such as FTP and HTML files.

Blaze [2] has addressed caching in a large-scale system. His research focused on distributed file systems, but can be applied to FTP or the Web. Finally, the Alex system [4] was designed to provide a means of caching FTP files. Of these three systems, Blaze's design comes closest to our own since it supports replication when demand becomes too high, and because it lets clients use any nearby cache. It does not, however, provide the server with control over where replicas are placed.

6 Conclusion

We do not believe that geographical push-caching should replace client-initiated caching. These two techniques address the same problem on two different time-scales, and therefore are complimentary. Client-initiated caching responds quickly to local changes in a file's popularity, but can not alleviate a global rise in demand. Likewise, server-initiated caching can not cope very well with sudden, localized jumps in popularity, but is best suited to handling long-term file request trends.

We close with this reminder of why server-initiated caching is necessary, taken from the Web home page of the WebLouvre [9].

> Note: Starting end of October 1994, we are currently experiencing severe network problems on our 256 Kb school Internet connection. Please be understanding! I am still looking for a site willing to mirror the WebLouvre exhibit (30 Mb in all), preferably in the USA.

References

[1] T. Berners-Lee, R. Cailliau, J-F. Groff, and B. Pollermann. World-wide web: The information universe. *Electronic Networking Research, Applications and Policy*, 2(1):52–58, 1992.

[2] Matthew A. Blaze. Caching in large-scale distributed file systems. Technical Report TR-397-92, Princeton University, January 1993.

[3] C. Mic Bowman, Peter B. Danzig, Darren R. Hardy, Udi Manber, and Mich ael F. Schwartz. Harvest: A scalable, customizable discovery and access system. Technical Report CU-CS-732-94, University of Colorado, Boulder, 1994.

[4] Vincent Cate. Alex - A global filesystem. In *USENIX File Systems Workshop Proceedings*, pages 1–12, Ann Arbor, MI, May 21 - 22 1992. USENIX.

[5] Alan Emtage and Peter Deutsch. Archie - an electronic directory service for the internet. In *Proceedings of the USENIX Winter Conference*. USENIX, January 1992.

[6] Ari Luotonen and Kevin Altis. World-wide web proxies. In *Computer Networks and ISDN systems*. First International Conference on the World-Wide Web, Elsevier Science BV, 1994. available from 'http://www.cern.ch/ PapersWWW94/ luotonen.ps'.

[7] Mosaic-x@ncsa.uiuc.edu. Using proxy gateways. World-Wide Web. available from 'http://www.ncsa.uiuc.edu/ SDG/Software/ Mosaic/ Docs/ proxy-gateways.html'.

[8] John K. Ousterhout. Tcl: An embeddable command language. In *USENIX Conference Proceedings*, pages 133–146, Washington, D.C., January 22-26 1990. USENIX.

[9] Nicolas Pioch. Le weblouvre. World-Wide Web. http: //mistral.enst.fr/ pioch/louvre/louvre.shtml.

Structuring the Kernel as a Collection of Reusable Components

Christopher Small
Harvard University
chris@das.harvard.edu

Abstract

Conventional operating systems provide high-level, black-box services to application programs. If the services do not fit the needs of an application, the application is out of luck. Applications whose needs might be met by the operating system need to reimplement facilities from scratch.

Instead of providing black-box services, an operating system should be decomposed into a set of *reusable, incrementally extensible* components. These components are used by the kernel to implement its services and can be reused by applications. If only part of a service is needed by an application (e.g. the buffer cache from the filesystem), it is available as a separate module. This approach is taken by the VINO kernel, under development at Harvard University.

1 Introduction

Conventional operating systems provide services such as file storage, name management, and caching. As a side-effect of implementing these services the kernel typically implements, but does not export, fast synchronization, simple transaction management, and logging. The interface provided by the kernel allows applications to use the exported services as-is; if the service is not appropriate, the application must implement a replacement from scratch.

At the application level, object-oriented development methodologies are all the rage. Toolkits for developing applications are available for user interface development, database access, document management, and general data structure manipulation [NEXT93]. These toolkits are composed of reusable, extensible, and cooperative modules.

The operating system should construct its services as such a toolkit. In order for a system structured this way to be useful, it needs to be appropriately decomposed and easily extended. This paper describes the service decomposition and extension mechanism found in the VINO kernel, under development at Harvard University.

2 Related Work

Others have examined the need for customizing operating systems. Kiczales et al. argue that the black-box model for operating systems hides not only crucial implementation details but crucial policy issues as well [KICZ93]; these policy decisions are not appropriate for all applications.

Anderson argues that the code in the operating systems should be stripped to a minimum [ANDE92]. The functionality of the operating system would be moved into the application; the kernel would only be responsible for arbitrating resource requests. Applications would have control over policy decisions because the decisions would be made in application code; services such as file systems would be developed as application libraries, and could be reused or circumvented by applications. The Aegis exo-kernel [ENGL94] follows this approach. In contrast, VINO leaves services in the kernel, but allows applications to reuse them in-place.

The SPIN system [BERS94] is an extensible microkernel that allows applications to add code to the kernel on-the-fly as *spindles*. The spindle mechanism allows services to be constructed that are tailored for a particular application. SPIN focuses on *adaptability* rather than *reuse*; although application code can be placed in the kernel (for a performance gain), the SPIN architecture is essentially that of a conventional microkernel.

3 Decomposition

An operating system is composed of a set of services. Some are normally exported, others hidden. For example, the Fast Filesystem [MCKU84] is composed of physical storage, a buffer cache, a name service, metadata synchronization and management, and a recovery tool. In VINO, where a larger service can be decomposed into potentially useful subservices, it is; for example, VINO offers each of the components of its filesystem as a separate service to application programs.

3.1 File Manager

The *file manager* provides the standard Posix-style file operations to its clients: read, write, seek, append, and truncate. A file system normally builds this abstraction on top of a physical disk using the disk; VINO interposes a volume manager between the file manager and the raw device. The volume manager arbitrates requests for disk space to subsystems that require persistent storage. The use of the volume manager allows the disk to be dynamically partitioned between its clients.

Alternatively, a filesystem can be created and directed to use a different manager for its storage, e.g. a virtual memory based volume (for building a memory-based filesystem), or a different file manager (delegating persistent storage to another filesystem). The latter technique can be used to build a *layered* filesystem; a compressed filesystem would override the default read and write operations and store the compressed data on a file system that writes its data to a disk.

3.2 Cache Manager

There are several caches in the typical system. The *buffer cache* is a cache of recently-used disk blocks; physical memory holds recently used virtual memory pages. A cache consists of a function that maps references to cache entries, a backing store interface, and a replacement policy. By allowing clients to define the implementation of each of these interfaces, the standard cache manager can be used by VINO for both for the buffer cache and the VM cache, or by applications managing their own data caches. For example, a database management system can manage its client cache by replacing the backing store interface with remote requests to the database server.

3.3 Name Manager

A filesystem normally includes a subsystem that maps a name to file reference (e.g. an i-node number, NFS file handle, or vnode pointer). The same code can be used for other applications that use a hierarchical namespace, such as a database naming subsystem, the Domain Name Service, or X11's resource database.

3.4 Locking

The operating system's synchronization primitives are typically much simpler and more efficient than those provided to user-level applications. For example, obtaining a semaphore in System V [AT&T] incurs the cost of a system call. This overhead is not usually necessary; if a lock is not contested, a user-level test-and-set instruction can be used to obtain the lock cheaply. If it is already held, a system call would then be made to enqueue the lock request [SELT92].

Single level locks are not always sufficient; a lock manager needs to provide *general-purpose hierarchical locking* [GRAY76]. For example, the file system typically allows locking at the block, file, directory and file system levels; a relational database management system lock hierarchy consists of field, tuple, relation, and database. When a lock is requested, the lock manager must verify that no conflicting lock is held on any other element in the hierarchy.

In most kernels, the file system locking hierarchy is implicit, buried in the code; a general lock manager must be able to work with any user-specified hierarchy. VINO accomplishes this by allowing an application to define a *containment hierarchy* for the resources being locked. When a lock request is made, the lock manager examines the currently allocated locks and the client-specified containment hierarchy to determine if the new request can be granted.

Clients need to be able to specify how the lock manager behaves in the face of lock contention, e.g. deadlock detection and resolution, blocking or non-blocking requests, and lock types (read vs. write locks) and lock compatibility (multiple concurrent readers vs. single writer).

Note that by integrating the kernel and user level locking systems, concurrency can be increased. For example, a DBMS running on a conventional UNIX[1] file system may implement its own lock manager to synchronize database access, and issue multiple I/O requests to the same file. Unfortunately, the UNIX file system exclusively locks the entire file during each I/O operation – no concurrency is achieved even though the DBMS is already ensuring the integrity of the operation. With shared lock management, because the same lock manager handles both DBMS requests and I/O requests, locks held by the DBMS are sufficient to perform I/O; no additional locking is required by the file system.

3.5 Log Manager and Recovery Manager

Several new file systems use database-style logging for improved performance and fast recovery [CHANG90, CHUT92, KAZAR90, VXFS], but this facility is not exported to applications. Obviously, database management systems use logging, but many other applications need recovery systems as well. For example, FrameMaker[2], `vi`, news readers such as *rn*, and email front-end tools all attempt to retain and recover their state in the face of runtime failure. Recovery code is notoriously complex, and is often the subsystem responsible for the largest number of system failures [SULL91]. Supporting multiple recovery systems can only reduce total system robustness.

In VINO, a *log* resides on one or more physical devices.

[1]UNIX is a trademark of X/Open.

[2]FrameMaker is a registered trademark of Frame Technology Corporation.

It can be created on a single device, or extended onto a second device (not necessarily of the same type as the first). A DBMS would request a log that spans both magnetic disk and archive media (e.g. tape or optical disk). The kernel would request a volatile, in-memory log to support transactions on ephemeral data, such as process structures and buffer cache metadata. The file system would request a log that spans non-volatile RAM and disk; file system log records would be written first to non-volatile RAM and later written to disk in large, efficient transfers.

The key interface to the log facility is the read/write interface that supports write-ahead logging: a *write-log* function that returns a unique identifier (a log sequence number), and a *read-log* function that returns records in log sequence order. It also supports a *synch-WAL* operation to synchronize the log with the data being logged, and a *checkpoint* operation for log reclamation and archiving.

3.6 Transaction Manager

The kernel uses transactions to maintain consistency during updates to multiple related resources (e.g. a directory and its contents). For example, when the Fast Filesystem updates metadata, it carefully orders disk writes to ensure the recoverability of the filesystem. In the context of a transaction, the order of these writes would be unimportant; the transaction would commit or abort atomically.

The VINO transaction manager supports the standard *transaction-begin*, *transaction-commit*, and *transaction-abort* operations. It accepts references to appropriate log and lock manager instances to use for each transaction. At transaction begin, a new *transaction resource* is created. This resource references the appropriate log and lock managers and is referenced by each protected update. Most kernel transactions are protected using a simple shadow-resource scheme with a log residing in main-memory (either volatile or non-volatile, depending on the resources being protected). The mixing and matching of logging and locking components enables VINO to support arbitrarily complex transaction protocols.

The transaction manager includes facilities for constructing *extended transactions* [BILI94], allowing applications to take advantage of alternative models such as nested and split-join transactions.

4 Extensibility and Reuse

Exporting a service to applications is only half the battle; we also need a mechanism for allowing the service to be specialized or extended.

VINO implements each of the managers described above as a *resource type*. A resource type consists of a group of operations and properties. The operations can be overridden by an application by *grafting* a new implementation into the kernel. The grafting process uses *sandboxing* [WAHBE93] or a similar software fault isolation technique to ensure that user code does not compromise the safety of the kernel. Code is written in a conventional programming language; unlike other extensible systems (e.g. SPIN [BERS94], Aegis [ENGL94] and Thor [LISK93]), we have not undertaken the task of defining a new typesafe language. It is outside the scope of our project to specify, implement,and support a new language, and widespread acceptance of new languages in the community, irrespective of their elegance and power, is very low.

Even with these assurances, user-installed code may not terminate in a timely fashion. The VINO kernel is multithreaded, and grafted code that runs too long times out. The grafted code may be ill-behaved and never return to the application, but only the application itself suffers; no other process is prevented from making progress.

We must also guard against grafted code obtaining a critical system lock and not releasing it in a reasonable amount of time. To handle this, we attach a time-out to critical locks, and kill a process that does not release the lock before the time-out. Each piece of grafted code runs in the context of a lightweight transaction that keeps track of its allocated resources. If the process terminates, the corresponding transaction is aborted, and the system is returned to a consistent state.

Unlike the external servers of Mach [ACET86], grafting allows small, *incremental* changes in kernel functionality. If the page eviction strategy of the system is inappropriate, it can be replaced without writing a new external pager [MCNAM90].

5 Conclusions

Conventional operating systems provide services as black boxes; where the services do not fit the needs of an application, the application is out of luck. Instead of offering monolithic services, the kernel should be structured to provide a collection of smaller, reusable, incrementally extensible tools for application reuse.

References

[ACET86] Acetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., and Young, M., "Mach: A New Kernel Foundation for UNIX Development", Proceedings of the Summer Usenix Conference (July 1986).

[ANDE92] Anderson, T., "The Case for Application-Specific Operating Systems", Proceedings of the

Third Workshop on Workstation Operating Systems, 1992.

[AT&T] AT&T, "System V Interface Definition, Third Edition," Volumes 1–3, 1989.

[BERS94] Bershad, D., Chambers, C., Eggers, S., Maeda, C., McNamee, D., Pardyak, P., Savage, S., Gun Sirer, E., "SPIN – An Extensible Microkernel for Application-specific Operating System Services," Technical Report 94-03-03, Department of Computer Science and Engineering, University of Washington, Seattle (1994).

[BILI94] Biliris, S., Dar, S., Gehani, N., Jagadish, H. V., and Ramamritham, K., "ASSET: A System for Supporting Extended Transactions", *Proceedings of SIGMOD 94*, Minneapolis, MN (May 1994).

[CHANG90] Chang, A., Mergen, M., Rader, R., Roberts, J., Porter, S., "Evolution of storage facilities in AIX Version 3 for RISC System/6000 processors," IBM Journal of Research and Development 34, 1, January 1990.

[CHUT92] Chutani, S., Anderson, O., Kazar, M., Leverett, B., Mason, W., Sidebotham, R., "The Episode File System," Proceedings of the 1992 Winter Usenix Conference, San Francisco, CA, January 1992.

[ENGL94] Engler, D., M. F. Kaashoek, and J. O'Toole, "The Operating System Kernel as a Secure Programmable Machine", *Proceedings of the Sixth SIGOPS European Workshop* (September 1994).

[GRAY76] Gray, J., Lorie, R., Putzolu, F., and Traiger, I., "Granularity of Locks and Degrees of Consistency in a Large Shared Database," in *Modeling in Data Base Management Systems*, Elsevier North Holland, New York, pp. 365-394 (1976).

[KAZAR90] Kazar, M., Leverett, B., Anderson, O., Vasilis, A., Bottos, B., Chutani, S., Everhart, C., Mason, A., Tu, S., Zayas, E., "DECorum File System Architectural Overview," Proceedings of the 1990 Sum- mer Usenix, Anaheim, CA, June 1990, 151-164.

[KICZ93] 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.

[LISK93] Liskov, B., Day, M., and Shrira, M., "Distributed Object Management in Thor", in *Distributed Object Management*, Morgan Kaufmann, San Mateo, California (1993).

[MCKU84] McKusick, M., Joy, W., Leffler, S., Fabry, R., "A Fast File System for UNIX," *Transactions on Computer Systems*, v. 2 n. 3, pp. 181-197 (August 1984).

[MCNAM90] McNamee, D., and Armstrong, K., "Extending the Mach External Pager Interface to Accommodate User-Level Page Replacement Policies," *Proceedings of the 1990 Usenix Mach Workshop*, Burlington, VT (1990).

[NEXT93] "NextStep 3.0 Users Manual", Next Computer (1993).

[SELT92] Seltzer, M., Olson, M., "LIBTP: Portable, Modular Transactions for UNIX", *Proceedings 1992 Winter Usenix Conference*, San Francisco, CA, pp. 9-26 (January 1992).

[SULL91] 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).

[VXFS] Unix System Laboratories, "The vxfs File System Type," from Advanced System Administration for UNIX SVR4.2, 1992.

[WAHBE93] Wahbe, R., Lucco, S., Anderson, T., and Graham, S., "Efficient Software-Based Fault Isolation", *Proceedings of the 14th SOSP*, Asheville, NC (December 1993).

Lies, Damned Lies, and File System Benchmarks

Diane Tang, Margo Seltzer
Harvard University, Division of Applied Sciences
{dtang, margo}@das.harvard.edu

Abstract

File system design, implementation, and performance is a hot topic in operating systems research, but nearly all the research in the area revolves around performance numbers derived from inadequate benchmarks. File system benchmarks suffer from lack of scalability, sensitivity to operating system behavior other than the file system, and fundamental misconceptions about what is being measured. If file system research is to move forward, the community needs robust, scalable, and informative file system benchmarks. This paper presents some of the flaws in today's file system benchmarks, proposes a set of guidelines for the development of good file system benchmarks, and discusses approaches to the creation of a compliant benchmark.

1 Introduction

Assuming that the number of publications in an area is an indication of research interest, file systems and distributed shared memory are among the hottest topics in operating systems research. The 1991 SOSP conference boasted seven file systems papers out of eighteen, 1993 SOSP boasted three of twenty-one, the 1994 Summer Usenix twelve out of twenty-seven, and the 1995 Usenix ten out of twenty-seven.

These papers focus primarily on three issues: distribution, improved performance, and scalability. In order to argue any of these points, researchers must demonstrate that the file system under investigation functions correctly, provides adequate (or exceptional) performance, and satisfies the novel claims made. Performance, in particular, is a key challenge for file systems as processor speeds climb exponentially while I/O speed grows at a linear rate [9]. Unfortunately, the technology for describing file system performance is woefully inadequate.

This paper critiques the most oft-cited file system benchmarks and proposes a set of criteria for the establishment of successful file system benchmarks.

2 Current Benchmarks are a Disgrace

Benchmarks commonly used to measure file system performance today suffer from several problems: lack of scalability, use of a single number as a final result, measurement of I/O performance rather than file system performance, and myopia (an emphasis on incidental implementation effects that rarely determine typical user performance).

We have examined most of the benchmarks used in recent research papers and will discuss Bonnie, the Andrew Benchmark, IOStone, and LADDIS (formerly known as NFSSTONE and NHFSSTONE). We use the weaknesses of these benchmarks to determine criteria for file system benchmarks.

Bonnie consists of six micro-benchmarks designed to measure bottlenecks in the file system [1]. Bonnie measures the disk read/write throughput and random seek time. The main shortcoming of Bonnie is that it is not really a file system benchmark, but rather a disk benchmark, and I/O performance cannot be equated with file system performance. For example, Bonnie gives no indication as to how fast a file system can perform a pathname lookup; it only tells the user how fast the system can transfer data. Because of these limitations, Bonnie does not indicate how well a real application will perform on the system. Bonnie yields the directive: *"Thou shalt measure the file system if thou art reporting file system performance."*

The Andrew Benchmark, originally developed at CMU to compare AFS to other file systems, uses existing Unix[1] utilities to create a directory hierarchy, copy files to that hierarchy, examine the files, and then compile them [3]. At the time Andrew was developed, it might have stressed many file systems. However, Andrew has not scaled with time: Andrew uses a fixed-size data set, which is too small. On most systems today, the entire data set will fit in the buffer cache, which means that after the initial create and copy, all data requests can be satisfied from the cache. Furthermore, Andrew's running time is dominated by the compile phase, which means that Andrew is almost

[1]Unix is a trademark of X/Open.

entirely user CPU bound, rather than either I/O bound or system CPU bound. As a result, it is unclear what Andrew measures today. Andrew yields the directive: *"Thou shalt make file system benchmarks scalable."*

IOStone, developed in 1990 at UC-Davis, is designed to measure file system performance on a workload based on Unix file system traces [7, 4] and IBM mainframe traces [11, 12]. IOStone has three phases: create a file system hierarchy, read and write the hierarchy, and delete the hierarchy [8]. Only the read/write phase is measured to produce one final result in IOStones per second. IOStone has many shortcomings. Its file system hierarchy model is flawed: IOstone claims to emulate the workload on a typical UNIX workstation by creating a model flat file system hierarchy, but real file system hierarchies are rarely flat. Furthermore, in an attempt to remove cache effects, it reads large spacer files before the read/write phase of the benchmark. Unfortunately, the spacer files are fixed-size (4 MB), independent of the cache size and therefore inadequate to flush large caches. The data set is also small enough to fit in almost any buffer cache, which means that, like Andrew, IOStone is not particularly I/O bound. The final shortcoming of IOStone is that it only produces a single result in IOStones per second. This result can provide comparative performance information, but it does not help the user determine what aspects of the system need to be improved or how to improve them. IOstone yields the directive: *"Thou shalt make benchmark results descriptive."*

The last benchmark we examine here is LADDIS, which is still under development. LADDIS is based on NHFSSTONE, which is based on NFSSTONE, and is designed to measure the performance of NFS servers [10, 5, 6]. LADDIS has the potential to be a wonderful benchmark - for NFS servers. It is scalable in the number of clients and in the load per client, its results must be presented graphically (showing how the performance of a server varies with load), and it measures the performance of the server. However, it is limited to NFS, and it does not give a clear indication of how to improve system performance since the only parameter it varies is load on the client. LADDIS yields the directive: *"Thou shalt make benchmarks prescriptive."*

What we can see is that many of the existing benchmarks have severe limitations: they do not scale, and they do not measure the file system. As a result, they are not particularly useful and they do not assist researchers in understanding file system performance.

3 The Call for a New File System Benchmark Metric

In order to design a good file system benchmark, we need to define a measure of goodness. Chen stated several goodness criteria for I/O benchmarks [2]. Namely, an I/O benchmark should be:

- Prescriptive: it should point system designers to possible areas of improvement.
- I/O bound.
- Scalable with advancing technology.
- Comparable between different systems.
- General: applicable to a wide variety of workloads.
- Tightly specified: no loopholes and clarity in what needs to be reported.

These measures are exactly those we derive for file system benchmarks. In fact, with the exception of I/O-boundedness, these criteria should probably be applied to most benchmarking methodologies.

In the case of file system benchmarking, we want to understand the behavior of each component, e.g., disks, caches, file system code. If we can isolate the performance of each component of the file system, then we can identify areas for improvement. Furthermore, if we can characterize an application in terms of these components, we can determine how well a particular file system will satisfy a particular application's needs. For example, suppose the task at hand is to optimize performance of a database system that stores every record in a separate file. This database will undoubtedly perform poorly on file systems with poor lookup performance.

It is our hope that the methodology that enables us to characterize a file system by its component performance can be extended to characterize an operating system by its component performance as well. Such a benchmarking system would allow us to accurately compare subsystems residing in different operating systems since we can accurately attribute differences to the correct components.

4 A Better Benchmark

The self-scaling I/O benchmark developed by Chen [2] fulfills the goals stated in the previous section. This benchmark has five parameters: the size of the overall data set, the number of processes running concurrently, the average size of an I/O request (to the nearest block), percentage of operations that are reads, and percentage of operations that are sequential (as opposed to random). The benchmark has two phases. It first finds the focal vector, which is the set of five values (one for each parameter) that are as far as possible from any drastic performance changes in throughput as a function of that parameter. Intuitively, the focal points are representative of "typical values," applicable over a wide range of workloads. Once the focal vector has been identified, the benchmark generates five graphs: plotting throughput as a function of each parameter with the remaining parameters at their focal point value. Using these graphs, Chen introduces the idea of predictive performance. He claims that since the focal vector is generally applicable, the shape of a graph should be applicable even when the parameters are not at their focal values. If a workload can be characterized in terms of these five parameters, then the workload performance is predicted by scaling between the five graphs.

This benchmark is scalable, tightly specified, reproducible, descriptive, and prescriptive - for the I/O system, which does overlap with the file system. Its only shortcoming for our purposes is that it is I/O system specific, rather than file system specific. It does not provide feedback on any file system component other than disk performance and buffer cache size, or what parts of the file system need to be improved.

We are searching for a file system benchmark that fulfills the metrics stated above. We are considering two possible approaches. One approach is to extend Chen's ideas to file systems, i.e., to find parameters that are indicative of file system performance, rather than I/O system performance. The main difficulty with this approach is in finding a set of parameters that are as plausibly independent from one another as those Chen uses to model the I/O subsystem. Without parameter independence, we lose the ability to predict the performance for a specific workload, and therefore the ability to make cross-platform comparisons as well.

A second approach is to augment Chen's idea with a trace-based benchmark. If we can gather file system traces for specific applications and successfully parameterize them in terms of file system operations, we can construct a file system benchmark that meets the criteria proposed above. For example, assume that we can gather file system traces of a target application set (e.g., a compiler, word processor, and database system). We process the traces and derive a parameterized workload, expressed in terms of the mix of file system operations, their dependence upon one another, and their interarrival times. This parameterized workload is then used to drive the benchmark. The traces are more informative than simply running the applications because they provide the ability to obtain timing information for specific calls. This benchmark can be scaled by increasing the number of processes, but can also be scaled to different processors and disks by altering the average interarrival time appropriately.

Using the combination of these two different parts of the benchmark, we fulfill all the metrics stated in the previous section. The adaptation of Chen's ideas gives us a prescriptive benchmark that measures specific aspects of the file system, while the trace-based benchmark yields cross-platform comparisons and applicability to a wide variety of workloads. The trace-based approach is insufficient by itself due to the difficulty in separating out the effects of the separate components in the file system, and thus is not prescriptive. Both methods can be made scalable and tightly specified.

5 Conclusion

Existing file system benchmarks are inherently flawed in that they are not very good at measuring the performance of a file system. We need a new benchmark that not only gives accurate performance numbers for a file system, but is also helpful, scalable, and able to be widely used.

References

[1] T. Bray, Bonnie source code, NetNews posting, 1990.

[2] P. M. Chen, D. A. Patterson. "A New Approach to I/O Benchmarks - Adaptive Evaluation, Predicted Performance", *UCB/Computer Science Dept. 92/679*, University of California at Berkeley, March 1992.

[3] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, M. J. West. "Scale and Performance in a Distributed File System", *ACM Transactions on Computer Systems 6*, 1 (February 1988), 51-81.

[4] I. Hu. "Measuring File Access Patterns in UNIX", Performance Evaluation Review 14, 2 (1986), 15-20. *ACM SIGMETRICS* (1986).

[5] M. K. Molloy. "Anatomy of the NHFSSTONES Benchmark", *Performance Evaluation Review 19*, 4 (1992).

[6] B. Nelson, B. Lyon, M. Wittle, B. Keith, "LADDIS - A Multi-Vendor and Vendor-Neutral NFS Benchmark", *UniForum Conference*, (January 1992).

[7] J. K. Ousterhout, J. DaCosta, et al. "A Trace-Driven Analysis of the UNIX 4.2 BSD File System", *Operating Systems Review 19*, 5 (December 1985), 15-24. Proceedings of the 10th Symposium on Operating Systems Principles.

[8] A. Park, J. C. Becker. "IOStone: A Synthetic File System Benchmark", *Computer Architecture News 18*, 2 (June 1990), 45-52.

[9] D. A. Patterson, G. Gibson, R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)", *International Conference on Management of Data (SIGMOD)*, (June 1988) 109-116.

[10] Shein, M. Callahan, P. Woodbuy. "NFSStone - A Network File Server Performance Benchmark", *Proceedings of the USENIX Summer Technical Conference* (1989) 269-275.

[11] A. J. Smith. "Sequentiality and Prefetching in Database Systems", *ACM Transactions on Database Systems 3*, 3 (1978), 223-247.

[12] A. J. Smith. "Analysis of Long Term File Reference Patterns for Application to File Migration Algorithms", *IEEE Transactions on Software Engineering SE-7*, No. 4 (1981), 403-417.

The Case for In-Kernel Tracing

Yasuhiro Endo
Christopher Small
Harvard University
{yaz,chris}@das.harvard.edu

Abstract

Operating systems are often criticized for providing fixed, lowest-common-denominator policies that are inappropriate for many classes of applications [6]. VINO, a new operating system under development at Harvard University, is like other extensible operating systems [1][2] in that it allows application programs to direct in-kernel policies through a mechanism called *grafting* [5]. However, to take advantage of this service and select or implement effective kernel policies, application developers must perform the non-trivial task of evaluating each new policy. This typically requires accurate simulation and/or tracing. There has been little research on how to aid programmers in this task. Many of the proposed solutions [3] rely on specially instrumented operating system kernels and/or simulation modules constructed to analyze a single decision. In this paper, we propose an in-kernel tracing and simulation mechanism designed to simplify the evaluation of application specified policies.

1 Introduction

Extensible operating systems offer an attractive alternative to applications whose functionality or performance are thwarted by the rigid kernel policies of conventional operating systems. However, extensible systems also burden the application with the need for increased decision making. It is possible for applications to exhibit worse behavior if application specified policies are not carefully selected or implemented. VINO simplifies the analysis process by providing in-kernel tracing and simulation tools. It does so by exploiting the VINO claim that the operating system kernel is a collection of reusable tools.

We structure the kernel to allow users to capture or introduce request streams between two kernel modules, and reuse existing kernel code to run simulations. Simulation modules are instances of regular kernel modules except that they do not affect any state seen by the rest of the kernel. For example, the buffer cache simulation module shares much of the code with the real buffer cache module including the grafted policy code. Application programmers can quickly and easily evaluate the suitability of different policies using the tracing and simulation tools that VINO provides.

2 Traces

The kernel is a set of modules with input ports and output ports connected together. Modules such as the file system, buffer cache, and device drivers have their ports connected together, and requests are transported through these connections. This architecture is similar to the use of streams in UNIX[1]. In UNIX, streams facilitate the adaption of network modules, while in VINO, our architecture supports the adaptation of nearly any module in the system [4]. For example, the input ports of the physically addressed buffer cache module are connected to the output port of the file system modules and the output ports are connected to the disk driver modules. The buffer cache module accepts requests for a block on a particular disk and either satisfies the request internally using the cached disk blocks or generates requests to the appropriate disk driver modules.

VINO provides users with simple methods to alter the flow of requests between modules. With this facility, users can easily generate a log of requests made by a particular module by redirecting the flow of requests to both its original destination and to a file. These logs can then be used to reproduce a stream of requests that can be fed into real or simulated kernel modules. In the remainder of this paper, we refer to the captured input request stream as a *trace* and the captured output request stream as a *log*.

This tracing facility allows application programmers a quick and simple way to evaluate policies. For example, we can evaluate different buffer cache management policies by replaying the same trace through the buffer cache module augmented with different policy algorithms, and counting the number of requests in the buffer cache log (i.e. counting the number of requests in the output stream). Because the tracing facility allows the replay of traces, we can recreate identical workloads for different

[1]UNIX is a trademark of X/Open.

test runs.

3 Simulation

Some modules inside the VINO kernel can be instantiated as simulation modules freeing programmers from the tasks of building separate simulators. Simulation modules are identical to real modules except that they do not modify the global state. Therefore, simulations can run without affecting the rest of the system. Since the simulators and real modules share much of the code, we do not increase code size substantially.

Modules that support simulation consist of two logical set of states: the first is writable by both the real and simulation instances of the module and is duplicated for each instance of such modules, and the other is writable only by the real instance of the module because the states are shared system-wide. In case of the buffer cache module, information such as which buffer cache page contains a particular block of a physical disk falls into the first category while the cached data itself falls into the second.

The simulation modules run without affecting the rest of the system nor are they affected by other activities in the system. This allows the simulators to be run under many different situations. The user can evaluate application specified policies by playing back a trace into the simulation module and logging the output to a file. Unlike using the real modules to evaluate policies, requests that simulation modules generate are not transmitted to the rest of the system. The simulations consume less system resources. This makes it feasible for application programmers to develop applications that dynamically alter the policies that they specify. Such programs may collect traces as they execute and periodically apply the information gained from tracing to select policies that may perform better.

4 Sample Uses

4.1 Buffer Cache Management

Using VINO's tracing and simulation facilities, application programmers can evaluate different buffer cache replacement policies using three different methods.

The first method is to run the target application in a controlled environment and collect logs from the output port of the buffer cache module. Given that the user is successful in executing the application in a controlled environment, the results obtained using this method are the most accurate of the three, since we are only using the tracing facility to record what is happening in the system: the requests that buffer cache management module receives are generated by a real program and handled by real kernel modules. This method is not subject to the timing problems associated with the second and third methods.

The second method is useful when the target applications' behaviors are difficult to reproduce. Many interactive programs fall into this category. For this class of applications, traces collected from the input port of the buffer cache module can be used to drive the experiment. The user must first run the application interactively to generate the trace, then while the programmer logs the requests sent from the output port, the trace can be played back into the input port of the buffer cache module as many times as needed to evaluate different policies. Results obtained using this method are not as accurate as those obtained using the first method. In order to properly reproduce the effects of events that are not directly triggered by the arrival of the requests, such as the flushing of dirty cache blocks, the trace must be played back with the exact timing. However, the timing of the request arrivals are often dependent on the behavior of the buffer cache. A miss in the buffer cache will delay the arrival of the next request. Therefore, if the policy being evaluated and the policy used when the trace was collected are drastically different in the efficiency, the result may be inaccurate.

The third method utilizes the simulation module as well as the tracing facility to provide a convenient way for programmers to perform dynamic evaluation of policies. An application may periodically collect traces from the input port of the buffer cache module and use the trace and the simulator to decide which policy is most appropriate for the tasks that the application is currently performing. Because a pre-recorded trace is used to drive the simulator, this method is also subject to the problems with accuracy of method two. In addition, imperfections in the simulator implementation may introduce additional errors.

4.2 Disk Layout Scheme Evaluation

Disks operate most efficiently when accessed sequentially. Therefore, an ideal disk layout scheme should maximize the sequential access of the disk and minimize the number and the distance of seeks that is needed to satisfy a given set of requests. We can use the tracing facility to examine the effectiveness of different disk layout scheme.

One of the difficulties in evaluating disk layout scheme arises from the fact that the disk layout scheme has long-term effect. Unlike the buffer cache management policy, which can be evaluated by running a short program, it is impossible to perform a meaningful evaluation of a disk layout scheme without aging the file system using the layout policy being examined. The aging process involves subjecting the algorithm to thousands of, if not millions of file creation, deletion, expansion, and contraction. One can create programs that artificially age the file system, but these programs usually fails to capture what really happens in the system under normal use.

To overcome these problems, the user can log all the relevant requests that the file system receives over a long period of time. The trace is then used to age the file system. The user must make sure that the file system is in a known and easily reproducible state (e.g. empty) before the aging process begins.

It is possible to analyze the aged file system statically to determine how much of the data is laid out sequentially, but the user should collect traces that reflect common file reference patterns to perform the evaluation. The use of these traces helps reflect the fact that the effectiveness of the layout policy depends heavily on which files are frequently accessed and how those files are accessed. The log of disk requests is collected from the output port of the buffer cache, and the user can use this log to analyze the effectiveness of the layout policy in maximizing the sequential access and minimizing the seek.

5 Conclusion

We have identified that in order for application programmers to take advantage of the kernel extensibility, there must be mechanisms to allow programmers to quickly and easily evaluate different policies. In-kernel tracing and simulation is a simple and general solution to this problem and can be implemented without a significant increase in the code size by reusing the code that is already in the kernel. We do not claim that this new facility will completely do away with the need for specialized tracing and simulation tools, nor do we advocate turning the operating system into a simulator, but rather, we present this as one example of how we can take advantage of existing kernel code to create a useful tool.

References

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

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

[3] Krueger, K., D. Loftesness, A. Vahdat, T. Anderson, "Tools for the Development of Application-Specific Virtual Memory. Management" In *Proceedings of OOPSLA '93*, volume 28, pages 48-64

[4] Richie, D. M., "A Stream Input-Output System", *AT&T Bell Laboratories Technical Journal* (Octover 1984).

[5] Small, C., "Structuring the Kernel as a Collection of Reusable Components", Submitted to HOTOS V.

[6] Stonebraker, M., "Operating System Support for Database Management", *Communications of the ACM*, 7, July 1981, 412-418.

Your Operating System is a Database

Keith A. Smith and Margo Seltzer
Harvard University
{keith,margo}@cs.harvard.edu

Abstract

The fundamental responsibilities of an operating system are the arbitration of access to shared hardware resources such as the processor, main memory, and I/O devices and the provision of a clean layer of abstraction atop potentially complicated devices. Database management systems provide the same functionality with respect to data—they arbitrate access to shared data and they provide simple abstractions to facilitate the manipulation of this data. Both operating systems and database systems address issues of synchronization, concurrency control, buffer management, distribution, and recovery. Despite this overlap of form and function, databases and operating systems have historically been implemented and researched completely separately, and often compete, rather than cooperate, for resources. This approach leads to redundancy in implementation and worse performance than is possible if a more integrated design and research approach is taken.

Operating systems consist of many components that perform essentially the same task as a database, but each of these pieces has its own idiosyncratic interface and implementation. The design and structure of operating systems can be simplified by using database structuring concepts to implement a uniform interface for resource management. In this paper, we present the VINO Universal Resource Interface, a general interface for structuring operating systems, and provide examples of its use.

1 Introduction

In simplest terms, both operating systems and database systems arbitrate access to shared resources. In the case of an operating system, these resources are typically hardware components: disks, network interfaces, serial lines, main memory or processors. In the case of database systems, the resources are typically data: files, records, or objects. Both types of systems perform similar tasks in managing these resources (e.g., buffer management, synchronization, disk allocation, recovery), yet they rarely share code to do so. The database community has been complaining about the lack of database support in operating systems for over a decade [Stone81], yet little seems to have changed.

Although database systems have been using logging to provide atomic updates and fast recovery for decades [Gray78], it is only in the last five years that we have seen the file system community accept logging as an approach for high performance and fast recovery [Ouster89, Kazar90, Chutani92]. Similarly, database management systems have long addressed the problems of distributed access to shared data [Bern81]. Operating systems have faced the same problem in providing support for distributed file systems and distributed shared memory. Unfortunately, much of the operating system research in these fields has ignored solutions used by database systems, leading to wasted effort rediscovering the same solutions (and the same dead ends). While operating systems have begun adopting some database techniques, they have done little to address the needs of database systems.

We believe that the time for operating systems researchers and database researchers to work together has long since passed. If either field is to move forward, both must acknowledge the commonality between the two and learn from the past. We go so far as to argue for a tighter integration: where database and operating system functionality overlap, they should endeavor to use a common code base. More fundamentally, as researchers, we must examine the conventional architectures of each and distinguish the gems from the flaws.

The rest of this paper is organized as follows. The next section provides a brief overview of database functionality. Section 3 describes a uniform interface for resource management in operating systems. In section 4 we show how a file system might be implemented using this structuring technique. Section 5 provides a brief survey of related work, and we summarize in section 6.

2 What Do Databases Do?

A database management system (DBMS) is a repository of data. A database used by a bank might include information about all of the accounts at that bank, the names and addresses of the account holders, and the amount of money in the accounts. Clients of a database make requests to the database when they wish to read or modify data (modifications include adding new data, deleting data, or editing existing data). A database may have many clients simultaneously issuing requests. In some cases, these clients may be distributed across a network.

In most cases, modifications to the state stored by a data base take place in the context of *transactions.* Transactions provide four important properties, atomicity, consistency, isolation, and durability. Atomicity means that that the modifications in each transaction are applied as a single unit. Either they all are applied to the database, or none of them are. The consistency property insures that data in the database is always in a consistent state; transactions are required to take the database from one consistent state to another. Isolation requires that result of concurrent transactions be indistinguishable from the result of applying the same transactions in some sequential order. The last property, durability, insures that once a transaction has been committed, its results are preserved across system failures [Gray93].

The properties of isolation and consistency require that some form of locking or concurrency control be applied to the data residing in a database. Traditionally, this capability has been provided through the use of locks, with read locks supporting multiple simultaneous accesses and write locks prohibiting concurrent access [Gray76].

To speed access to data, most databases index the stored data items by one or more keys. Given the key, the database can quickly retrieve the corresponding data object by performing an associative lookup. The exact data structures and lookup mechanisms used by a database are transparent to the user, residing behind the abstraction of *records* and *keys.* Frequently, the selection of indexing structures is tuned to the application and type of data being accessed. Balanced trees (B-trees) and hash tables are two of the most common indexing mechanisms.

Another optimization performed by nearly all database management systems is the buffering of frequently-used data in memory to exploit spatial and temporal locality in the stream of data references. This is similar to the buffering implemented in a traditional file system buffer cache except that databases typically implement more sophisticated page replacement algorithms than the simple LRU algorithm used by most file system caches [Chou85].

3 A Universal Resource Interface

An operating system controls many resources that are managed in the same manner as the data resources managed by a DBMS. Some of these resources are visible to the users of the operating system (e.g., directories, files), and some are not (e.g., page tables, scheduling queues). Each of these resource types can be viewed as a set of similar objects. New objects can be added to the set, and existing objects can be deleted from it. The operating system performs lookup operations to retrieve objects from the set, and the operating system may modify existing objects in the set. This model of resource management is the same as that used by database systems to manage data objects.

A page table, for example, is simply a collection of physical memory pages indexed by their virtual addresses in a process address space. When new pages are brought into memory, new entries are added to the page table. Similarly, when pages are evicted from memory, entries are deleted from the page table. Whenever the process references memory, the virtual address of the reference is used to perform an associative lookup in the page table, returning a physical memory page where the desired virtual address is located. In most architectures, this lookup is expedited by dedicated hardware, such as TLBs.

On a uniprocessor, the database notion of shared versus exclusive access seems to have no analog with respect to page tables. With only one processor, only one page can be accessed at a time, and therefore all page accesses are exclusive. In a parallel or distributed architecture, however, multiple processors may be using the same page table and attempting to access the same pages concurrently. In such a system, memory consistency becomes an important issue, and the processors must be coordinated with respect to which ones read or write a given memory page at what point. This problem is exactly analogous to the problem faced by a DBMS in handling concurrent requests to read and/or write the same data objects.

Although many different operating system facilities require this database model of resource management, each one is typically implemented independently, resulting in multiple implementations and interfaces for arbitrating access to resources. This lack of a common resource management interface limits the opportunities for code re-use, and, where only a subset of the database resource management interface is implemented, limits operating system functionality.

VINO [Small94], a new operating system under development at Harvard University, exploits this common ground between operating systems and database systems. In VINO, all resources are managed through a *Universal Resource Interface* (URI). This interface is explicitly modeled after the resource management techniques used by database management systems. The VINO URI is implemented by a set of kernel modules called *resource managers* each of which manages access to a collection of objects via a set of interface routines. This interface, which is standardized by the URI, has functions to add or delete objects, to retrieve objects via an associative lookup on one or more keys, and to return modified objects to the resource manager.

The VINO URI also includes commands that can be used by the transaction service when beginning, committing, and rolling back transactions. For example, rather than using careful write ordering [Ganger94] or a separate logging facility [Chutani92, Kazar90] to maintain the integrity of file system meta-data operations, VINO uses a general purpose transaction mechanism. The degree of durability conferred by the transaction mechanism may be tailored to the requirements of the resource manager, in a manner similar to that used by QuickSilver [Schmuck91].

4 Implementing a File System

A traditional file system exports an interface similar to that of the universal resource interface. Thus, the file system makes an excellent introductory example for understanding the VINO URI.

From the user's perspective, all files are objects managed by a resource manager called "the file system." The *creat* and *unlink* system calls correspond to adding and deleting (respectively) a file object. Reading a file is analogous to retrieving a file (or a part of it) from the file system for read access. A write operation acquires write access to the file and updates the file object appropriately.

Filenames serve as keys for referencing files. A file rename operation performs a transaction in which the old file is deleted, and a new file (with the same contents but a different name) is created. Since this happens in the context of a transaction, if the system fails during the rename operation, the file will either retain its old name, or have the new name, but it will be impossible to wind up in a state where the file has both names, or neither. While today's file systems also provide this functionality, they do so using fairly complicated, special-purpose code.

Because transactions are a fundamental part of the VINO model of resource management, the transaction facility is available to user processes for insuring the integrity file data. This allows a uniform recovery mechanism to be used by applications concerned about data integrity, such as word processors and source code control systems.

Although the file system has the outward appearance of a single resource manager, it is actually implemented as several distinct but cooperating resource managers. The file system namespace is implemented by the *name manager*, a resource manager that manages a collection of files. File names are used to index the files. File creation and deletion correspond to requesting an addition to or deletion from the name manager. The *open* system call is implemented by requesting a lookup from the name manager and returning the corresponding file object, with read and/or write permission, as requested in the *open* call.

Each file is implemented by a *file manager*, a resource manager that manages the set of disk blocks where the file resides. The file offset is used as a key for retrieving these blocks. Note that only one file manager need be implemented; individual files are treated as separate instances of this manager, reusing its code with the appropriate file-specific data. Reading a file is implemented by retrieving the blocks at the requested offsets with read permission. Writing a file is implemented, by modifying the file blocks at the appropriate offsets.

When a file is extended, new blocks must be allocated to it. These blocks are added to the file's file manager. The new blocks are allocated from a *storage manager*, a resource manager representing a disk partition. When a file needs to allocate new blocks, it requests the blocks from the storage manager where the file resides. The blocks are not returned to the storage manager until the file is truncated or deleted, preventing blocks from being simultaneously allocated to more than one file.

The URI model also allows us to specify a protocol for deciding when to grant shared or exclusive access to a file (i.e., when to allow read and/or write system calls to overlap). The UNIX Fast File System [McKusick84] does not permit shared access to files; no more than one read or write request to a file is serviced at a time. Clearly a one writer, multiple reader policy would permit greater file throughput. This more complicated form of locking is rarely implemented for file systems, however, since the gains in concurrency are not enough to warrant the effort of implementing it. The general-purpose resource management provided by the VINO URI allows code to be easily shared between resource managers. Thus, it is only necessary to implement one solution to the readers and writers problem, which can then be used for managing any resource where this model of shared access is appropriate.

The division of the file system into separate resource managers also allows a variety of locking and concurrency management schemes. The name manager can be used to prevent conflicting read and write operations on the granularity of entire files. This would prevent an application from reading any part of a file that was concurrently being written to. If decisions about shared access are left to the file manager, concurrency can be handled on the granularity of individual disk blocks.

5 Related Work

VINO borrows from both the database and operating system communities for its design. The Plan 9 operating system [Pres90] has a common interface for accessing most services. This interface is based on the file system interface found in other operating systems, and is not explicitly designed to provide resource management.

QuickSilver [Schmuck91] is a microkernel operating system that uses transactions as a fundamental primitive, however they do not extend the use of this transaction mechanism to applications.

There are a number of file systems [Chutani92, Chang90] that have incorporated logging to insure the integrity of their meta-data. Unfortunately these file systems do not export this transaction mechanism to user applications, or to other parts of the operating system kernel, which are therefore forced to implement their own recovery mechanisms.

Inversion [Olson93] is a file system built on top of the POSTGRES [Stone87] database management system. Inversion provides a transaction-based recovery mechanism for file data as well as for file system meta-data. It also allows users to define new files types and functions for operating on them. Users of Inversion can use the underlying database to issue queries on the file system's contents and meta-data.

VINO is not unique in separating naming from the file storage service. The Amoeba operating system [Mullen90] also divides traditional file system functionality along these lines, providing a directory service that maps names to capabilities for file objects. The file objects are managed by a separate service called the *bullet* service.

6 Conclusion

Traditional databases and operating systems use a variety of similar techniques for solving resource management problems. Over time, databases have evolved a uniform model for manipulating the range of resources they need to manage. Operating systems, in contrast, still use a variety of ad hoc mechanisms and interfaces for resource management. We feel that the access method model for resource management developed for use in databases is also appropriate for operating system resource management.

The VINO operating system is designed to explore this common ground between operating systems and databases. The VINO Universal Resource Interface provides a uniform interface for resource management modeled after the access methods used by databases.

References

[Bern81] Philip A. Bernstein, Nathan Goodman, "Concurrency Control in Distributed Database Systems," *ACM Computing Surveys*, Vol. 13, No. 2, June 1981, pp. 185–221.

[Chang90] Chang, A., Mergen, M., Rader, R., Roberts, J., Porter, S., "Evolution of storage facilities in AIX Version 3 for RISC System/6000 processors," *IBM Journal of Research and Development,* Vol. 34, No. 1, January 1990.

[Chou85] Chou, Hong-Tai, DeWitt, David, "An Evaluation of Buffer Management Strategies for Relational Database Systems," *Proceedings of the Eleventh International Conference on Very Large Database*, August 1985, pp. 127–141.

[Chutani92] Chutani, S., Anderson, O., Kazar, M., Leverett, B., Mason, W., Sidebotham, R., "The Episode File System," *Proceedings of the 1992 Winter Usenix Conference,* San Francisco, CA, January 1992.

[Ganger94] Ganger, G., Patt, Y., "Metadata Update Performance in File Systems," *Proceedings of the First Usenix Symposium on Operating System Design and Implementation*, Monterey, CA, November, 1994, pp. 49–60.

[Gray76] Gray, J., Lorie, R., Putzolu, F., and Traiger, I., "Granularity of locks and degrees of consistency in a large shared data base," *Modeling in Data Base Management Systems*, Elsevier North Holland, New York, 365–394.

[Gray78] Gray, J., "Notes on Database Operating Systems—An Advanced Course," Springer Verlag Lecture Notes in Computer Science, Volumne 60 1978.

[Gray93] Gray, J., Reuter, A., *Transaction Processing: Concepts and Techniques,* Morgan Kaufmann Publishers, San Francisco CA, 1993.

[Kazar90] Kazar, M., Leverett, B., Anderson, O., Vasilis, A., Bottos, B., Chutani, S., Everhart, C., Mason, A., Tu, S., Zayas, E., "DECorum File System Architectural Overview," *Proceedings of the 1990 Summer Usenix*, Anaheim, CA, June 1990, pp. 151–164.

[McKusick84] Marshall Kirk McKusick, William Joy, Sam Leffler, and R. S. Fabry, "A Fast File System for UNIX," *ACM Transactions on Computer Systems,* Vol. 2, No. 3, August 1984, pp. 181–197.

[Mullen90] Sape J. Mullender, Guido van Rossum, Andrew S. Tannenbaum, Robbert van Renesse, and Hans van Staveren, "Amoeba—A Distributed Operating System for the 1990s," *IEEE Computer,* May 1990, pp. 44–53.

[Olson93] Michael A. Olson, "The Design and Implementation of the Inversion File System," *Proceedings of the 1993 Winter Usenix Conference,* San Diego, CA, January 1993, pp. 205–217.

[Ouster89] Ousterhout, J., Douglis, F., "Beating the I/O Bottleneck: A Case for Log-structured File Systems," *Operating Systems Review 23*, 1, January 1989, 11–27.

[Pres90] Presotto, D., Pike, R., Trickey, H., and Thompson, K., "Plan 9, a Distributed System," *Proceedings of the Spring 1991 EurOpen Conference,* Many 1991.

[Schmuck91] Frank Schmuck, Jim Wyllie, "Experiences with Transactions in QuickSilver," *Proceedings of the 13th Symposium on Operating System Principles*, Pacific Grove, CA, October 1991.

[Small94] Chris Small, Margo Seltzer, "VINO: An Integrated Platform for Operating System and Database Research," Harvard Computer Science Technical Report TR-30-94, 1994.

[Stone81] Michael Stonebraker, "Operating System Support for Database Management," *Communications of the ACM,* Vol. 24, No. 7, July 1981, pp 412–418.

[Stone87] Michael Stonebraker, "The Design of the POSTGRES Storage System," *Proceedings 13th International Conference on Very Large Data Bases,* Brighton, England, September 1987, pp. 289–300.