HEWLETT PACKARD

Brevix design 1.01

The Brevix design team (Martin Fouts, Tim Connors, Steve Hoyle, Bart Sears, Tim Sullivan, and John Wilkes)

HPL–OSR–93–22, **April 1, 1993**

Operating Systems
Research Department
Hewlett-Packard Laboratories

This document describes the design of Brevix—a framework for the nucleus of an operating system for a scalable, fault-tolerant computer system being developed at HP Laboratories. This design overview is intended to provide a concise survey of the principal functions that Brevix provides, and the approaches used in its design.

Contents

Figures

Tables

1 Introduction

Writing is nature's way of letting you know how sloppy your thinking is.

— Guindon

1.1 Document scope

Brevix is the framework for an operating system nucleus for distributed memory multicomputers. This document describes **what** Brevix is: the problems it solves, and the features it provides in doing so. It will be accompanied by separate design documents that describe **why**, and will be followed by a more detailed document that describes **how** Brevix is implemented.

This document is intended to be read by those who will implement the Brevix design, and those who seek an overview of what Brevix provides and the problems it solves.

The remainder of this chapter is organized as follows: the context in which Brevix is being designed, including our assumptions about the underlying hardware, is presented, after which we state the goals for Brevix. This is followed by an overview of the system structure and a summary of the contributions the Brevix effort makes and, finally, by a roadmap for the remainder of this document.

> Note: text displayed like this is ancillary supporting material that can be omitted on first reading. It is used for engineering notes and explanatory examples.

1.2 Context and goals

1.2.1 Context

The Computer Systems Laboratory of Hewlett-Packard Laboratories is developing a scalable multicomputer we call the *Scalable Concurrent System*, or SCS. The goal of the project is to develop software and hardware solutions to exploit the performance and functionality advantages of multicomputers.

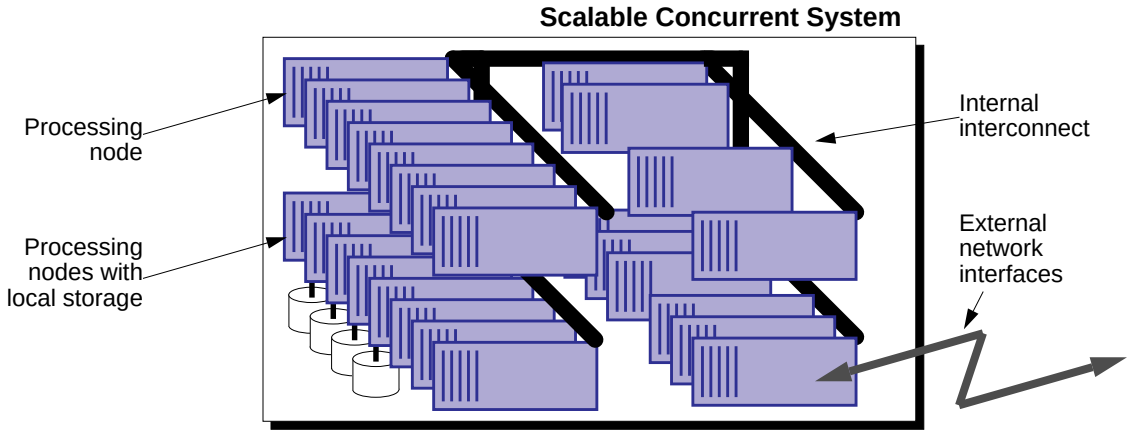

Figure 1: the SCS hardware model: nodes linked by a high-performance interconnect.

To complement the approach already being taken in clustered computing elsewhere in Hewlett-Packard, we have chosen scaling of commercial applications on a server platform as a demonstration goal for SCS. This requires that Brevix support scalable transaction processing for both on-line transaction processing (OLTP) and decision support applications. This also implies that we meet commercial fault tolerance needs. As time passes, and we extend the scope of the support tools and programming environments, it is our intent that Brevix come to be the nucleus of HP's full-function general-purpose operating system, supporting whatever it takes to be the best programming, development, and application executing engine of the next decade.

Hardware base. The SCS is a distributed-memory multicomputer (Figure 1). Each *node* consists of a processor, some local memory, and an interface to an optimized internal interconnect, called the *fabric*. This interconnect provides the routing, reliability, latency and bandwidth necessary to support from several to a few hundred nodes. In addition, each node may be configured to support storage devices such as disks and tape drives; and may be attached to external network interfaces such as Ethernet, FiberChannel, and FDDI. The SCS hardware provides no support for memory coherence between processor nodes.

The tentative hardware scaling goals for SCS are shown in Table 1. The *minimum* configuration represents the smallest supported system; the *maximum* represents the largest that could be physically assembled. The *design center* identifies the configuration for which the architecture, system design, policies, and implementation are optimized. Configurations in the region of the design center will achieve the highest cost-effectiveness. Secondary and tertiary storage capacities will be limited only by the connectivity provided by the I/O channels.

The I/O channels will initially be SCSI, perhaps FiberChannel later.

Table 1: SCS hardware scaling goals

	Min	*Design Center*	*Max*
Nodes/system	4	128	512
Processors/node	1	1	8
RAM/node	16MB	128MB	4GB
NVRAM/node	0	1MB	128MB
I/O channels/node	0	1	8
Disks/node	0	2	56
Total disk storage (2GB 3.5" disks)	2GB	512GB	50TB
LAN ports/system	2	16	512
Fabric hardware latency	—	2–5µs	10–20µs
Fabric bandwidth per node	16MB/s	128MB/s	1GB/s

The list of hardware scaling goals is subject to revision as we learn more; your inputs are solicited.

Fault tolerance. The list that follows specifies the fault tolerance design-center for Brevix (the terminology is defined in section 2.2 on page 15). These are intended as targets for the design to

optimize around, not exact system goals for any particular Brevix instantiation. Their achievement depends on many components, including the Brevix design, its implementation, and the fault rate and configuration of the underlying hardware and application software. A site may choose to ask for more or less than these requirements in order to meet business needs or economize on resources.

- Availability at 90% of max performance: 0.999 (all but 8.8 hours per year)
- Reliability at 90% of max performance over a year: 0.99
- Coverage: 0.999
- Integrity of data over a year: 0.9999
- Mean time to disaster: 100 years.

1.2.2 Assumptions about the underlying hardware

Brevix is designed to rely on small set of hardware facilities. The following list describes these facilities. Relaxing some of these requirements will not stop Brevix from running correctly, but may adversely impact its performance or robustness.

1. A reliable interconnect. The interconnect will "never" fail (see section 2.2 on page 15 for a description of "never"). While individual nodes, fabric connections to a node, and links within the interconnect may fail, none will cause the entire interconnect to fail or partition. This implies that services that are needed for the entire interconnect, such as power supplies, fabric components and external cabinets have sufficient reliability, possibly through redundancy.
2. An interconnect that scales up. The interconnect should provide nearly constant bandwidth per node as the size of the system increases. That is, aggregate bandwidth should scale, while per-node bandwidth need not. In addition, the latency between nodes should remain nearly independent of the distance between them, regardless of the scale of the system.
3. A trusted internal interconnect. In particular, Brevix assumes that the interconnect is not subject to message data capture and replay attacks, so no encryption is needed to build secure channels between nodes.
4. A low error rate on the interconnect. Messages that are delivered are never corrupt. Few messages are lost in the fabric.
5. A memory model that does not deteriorate in performance or robustness as the system scales up. Currently, we believe that a true sequentially-consistent global shared memory will not scale at any reasonable cost and thus Brevix does not assume or rely on a sequential global shared memory.
6. Hardware that, when it fails, fails cleanly. At a minimum we require the hardware to not fail in a Byzantine fashion with arbitrary data or message corruption. We would like the hardware to provide notification of failures, if possible.
7. Processor hardware that provides sufficient capabilities to support our memory-protection model (which is described in section 3.2.2 on page 27).
8. Processors that all use identical data formats and are instruction-set compatible.
9. Secure or securable paths to user terminals, if needed by the security model.
10. Support for *hot replug*, that is replacing components while the system is running. This allows on line repair, maintenance, and upgrade of the system.

11. Power supply and hardware configuration that is visible to software, so data and processing placement can be arranged to minimize the effects of loss of a power supply or hardware components.
12. Stable storage of some flavor. This could be either non-volatile memory (NVRAM) or an uninterruptible power supply (UPS).
13. An intrinsically low hardware fault rate: on the order of 50 software-visible hardware faults per year in a system the size of the SCS design center

    A complete high-level design for a hardware interface that supports high performance, low-overhead inter-processor communication across the interconnect is Hamlyn [Wilkes92]. Efficiency is achieved by using sender-controlled message placement at the recipient, and by providing a scheme that allows applications running in non-privileged mode to access the interconnect directly, without operating system intervention.

1.2.3 Goals

The goal of Brevix is to support *scale-up* in multicomputer systems: as more processing nodes are added to the system, it should be possible to solve larger problems in the same time. The size of the problem should scale linearly with the number of nodes in the system. We have concentrated on providing support for those applications which will exhibit medium to coarse grain parallelism when written to use the Brevix programming interface. Beyond the scaling requirement, Brevix will also provide the reliability, robustness and data integrity (security) necessary for commercial applications.

Brevix is intended to support multiprogramming rather than batch computing. As such, it does not provide facilities found on some traditional multicomputer aimed at scientific batch computing, such as the ability to partition a running system between competing jobs. However, it is designed to allow those facilities to be added at a later time.

A careful attempt has been made in Brevix to separate mechanism from policy and to allow the user to utilize multiple policies under administrative control.

1.2.4 Scope

The scope of the Brevix effort is restricted to development of the functions normally associated with an operating system kernel, even though Brevix is not structured as a traditional kernel. Since Brevix is capable of supporting multiple application programmer's interfaces (APIs) we will be able to leverage all of the existing commands, utilities and programming libraries from HP-UX.

The effort at HP Labs on Brevix is aimed at prototyping necessary critical component technologies and system structure. We anticipate separate efforts will be necessary to turn this into a full product offering, and plan to support these directly from our work.

1.3 Brevix system structure

Brevix is not a monolithic kernel. It is also not a micro-kernel. Of the currently available operating systems, it is closest to Chorus [Abrossimov89a, Herrmann88, Rozier88b] and Pilot [Redell80].

1.3.1 Conceptual organization

The Brevix structure is that of a *framework*, which defines a set of *system services* and provides the glue to bind them together. The services represent functions that can be performed, possibly in different ways by different implementations of the service. In some cases, there will only be one *instance* of a service. For example, there is no real reason to have multiple copies of the time-of-

day clock service on a node. In other cases, the same services may be made available to distinct sets of objects, such as a collection of physical memory pageframe, or a group of entities (storage elements). The objects in a set may be grouped together because they share a common implementation or policy, such as being optimized for a particular access pattern, or simply for administrative convenience. In either case, we say that each set has a separate *manager*.

Finally, all services and managers are implemented by *providers* (collections of code), possibly private data (which can be protected), and perhaps private worker threads to perform asynchronous processing. (Note: Threads are not precisely the same as processes. See section 4.1 on page 44). Providers can invoke other services. Services and managers are system-wide abstractions, and can span multiple nodes in the system. Each provider is local to a single node. This is represented diagrammatically in Figure 2.

> *Service Interface* as used in figure 2 refers to the signature of the interface, not to the specific instance of an interface. This distinction is clarified in section 4.2 on page 50. If the instance were referred to rather than the signature, there would be three distinct service interfaces shown on node 2, rather than one.

The services offered by providers are invoked by *customers* through instances of *interfaces*, which are capable of protecting service provider and customer from one another, if desired. One difference from the traditional client–server relationship is that it is the *same thread* that executes in the customer and inside the provider: all that happens at the boundary is a protection state change, possibly supported by argument and return value copying, not a switch to a new thread.

In Brevix, there is no fixed OS boundary. As more providers are added to the system, the system provides more functionality. There will be a standard Brevix application programming interface (or API) that will define the minimal functionality that a Brevix system will provide. In addition, it will be possible to write providers that will make other standard APIs available, such as POSIX, HP-UX, and NT.

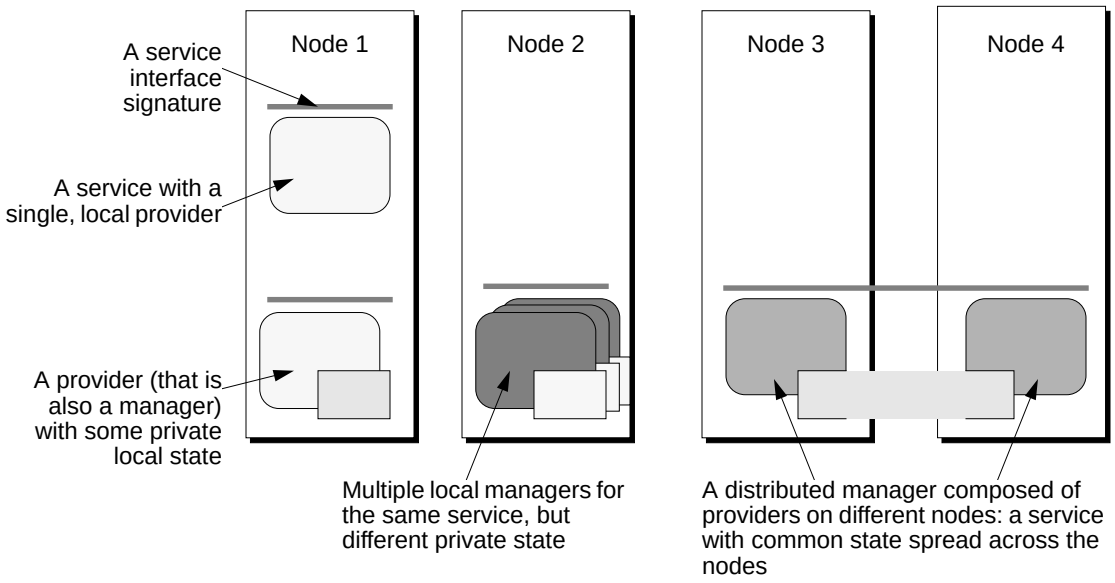

Figure 2: services, managers, and providers.

Brevix has been designed from the beginning for *scalability* and *fault tolerance*. Providers can be replicated for fault tolerance. There is as little global data as possible, to improve scalability.

> For example, structures like the UNIX system *proc* structure are broken up, with each provider containing the section of the *proc* structure that it logically owns. This makes it much easier for the code to scale.

Brevix minimizes the amount of privileged code needed to just a few small sections to simplify checking that the system meets its security goals. Only the low level interrupt handler, the interface crossing code, and the context switch code need be privileged.

1.3.2 Brevix abstractions

The evolution of operating systems has in some ways paralleled the evolution of programming languages, but, in the past, concern for efficient use of limited hardware has hindered attempts to provide facilities equivalent to the imperative programming-language model. In particular, the idea of programming as computation that transforms data has been obscured by the need to introduce I/O operations to support efficient uses of primary and secondary storage.

But computer hardware has evolved: faster, less expensive processors, larger main memories, and more available secondary storage are now coupled with more appropriate protection models and high performance local area networks. Many of the assumptions underlying previous choices in OS design are no longer valid. In addition, problem understanding has improved: many of the previously inefficient approaches to solving OS problems have now been replaced by more efficient algorithms. This has enabled the design of operating systems that provide abstractions more closely coupled to the programming language model. Brevix is such an operating system.

Brevix provides four principal abstractions; all other Brevix facilities exist to support these abstractions. The programmer's view of data is supported by the *entity*. The programmer's view of computation is supported by the *thread*. And the programmer's views of software code and encapsulation are supported by the *provider*, which is invoked through a Brevix *interface*.

Entities. Entities are containers for data. When the data are in primary storage (which is done by binding the entity into the virtual address space), they are supported by pageframe managers. When the data are in secondary storage, they are managed by the storage system, which is composed of entity, parcel, device, and hierarchy managers.

Access to entities is primarily through direct loads and stores, once the entities have been bound to the virtual address space. Appropriate copying takes place automatically between primary and secondary storage, although this can be controlled directly by the application if desired.

The Brevix distributed memory model is based on weak consistency, which gives better performance than strong consistency. As a result, explicit release operations are performed to ensure that changes are made visible to other threads.

Threads. Brevix threads are not traditional processes. They correspond to individual computations, but do not exist in a separate address space within their own environment as would be the case for a traditional process. Instead, threads shares a common virtual address space with all other threads in the system. That is, Brevix does not support the model of the process as the sole owner of a virtual machine.

At any moment in time, each thread has a collection of entities that it is allowed to access and possibly modify. Binding of entities to the common virtual address space is separate from

protection of access to entities by threads. This approach relies on the existence of a large virtual address space with a separate orthogonal mechanism for address space protection.

Concurrency is supported by having threads share access to entities, and by using a set of synchronization services. A thread can also register an interest in the termination of another. For example, this is used inside providers so that worker threads can perform cleanup operations.

Providers. Brevix provides traditional procedures and libraries. Additionally, Brevix provides a mechanism analogous to the programming language package, with the additional facility of support for controlled access to data. This is called a *provider*. Providers are invoked with simple procedure calls, with the result that the same customer thread that calls the provider executes the called code inside the provider.

A provider exports a collection of entry points, each representing some function that is exported by the provider, together with exception return information. In addition, a provider can keep private data between calls, and protect them from access by the provider's customers. It can also have private worker threads that operate even while no customer thread is executing inside the provider. Finally, a provider may be installed so that it executes at a different hardware protection level than its customers, to give it greater access to processor hardware.

Interfaces. A thread installs an interface in order to get access to the services of a provider. In doing so, it defines those portions of its own execution context that are to be accessible to the provider through this (instance of the) interface. This mechanism provides bidirectional control of the data that are shared between customer and provider.

Because the entire Brevix environment is visible across all of the nodes of the SCS, the Brevix remote procedure call mechanism and a local interface crossing are indistinguishable from the point of view of either the calling code or the called routine. If the provider resides on a different node, the thread is extended to that node via a remote procedure call. The called code need not know if the call was made from another node.

1.3.3 System structure overview

Brevix takes advantage of its own programming model by being implemented as a collection of related service providers. Brevix has neither a kernel nor a microkernel, only this suite of providers, some of which execute with greater privileges and access rights than others.

Figure 3 illustrates some of the relationships between the providers that make up a Brevix system. The solid lines in the figure show direct interactions between these service providers and managers. Not all of the interactions are shown, since all portions of the Brevix system depend on the interface manager, most depend on the synchronization service and many depend on the time services.

Entities are supported by pageframe managers and parcel managers. Pageframe managers rely on the local address translation service, parcel managers rely on device managers; the whole operation is coordinated by entity managers. Threads are supported by thread managers and the interface manager. Additional services include support for timing (both time of day and for precisely evaluating the length of an interval of time); thread synchronization; and a global namespace through a hierarchy of directory managers that resolve textual names into handles.

All of these rely on the interruption service, which translates hardware interruptions into exceptions or interrupts and dispatches them to the appropriate handlers, and the security service, which decides whether proposed operations should be permitted.

All Brevix providers interact with the security service. This interaction is discussed in more detail in section 2.1 on page 11. The security service is designed to separate security policy from system implementation, so that different instances of Brevix systems may support different kinds of security policies.

Only the interruption and address translation services, and the interface and device managers are concerned directly with manipulating protected state in the underlying physical machine. The remainder of Brevix uses the *abstract machine* provided by these facilities in the same manner as any application would. The abstract machine is the set of services provided by interruption, address translation, and the interface manager, which allow other parts of the system to manipulate the underlying hardware in a secure, portable fashion.

> Depending on the organization of the system, it may be possible for device managers to operate without privileged access.

Some of the services are shown with multiple boxes. Any of these may be supported by multiple providers on a single node to support variations that implement different policies, or to manage separate groups of items such as entities. Those shown with a single box may be supported by only one provider on the node, or they may be supported by an interface instance that accesses a service on another node. Address, interruption, device and interface services are always supported by a provider on the local node.

> Although the namespace service is shown with multiple boxes because the design allows for multiple possible namespaces, no decision has yet been made that requires multiple namespaces.

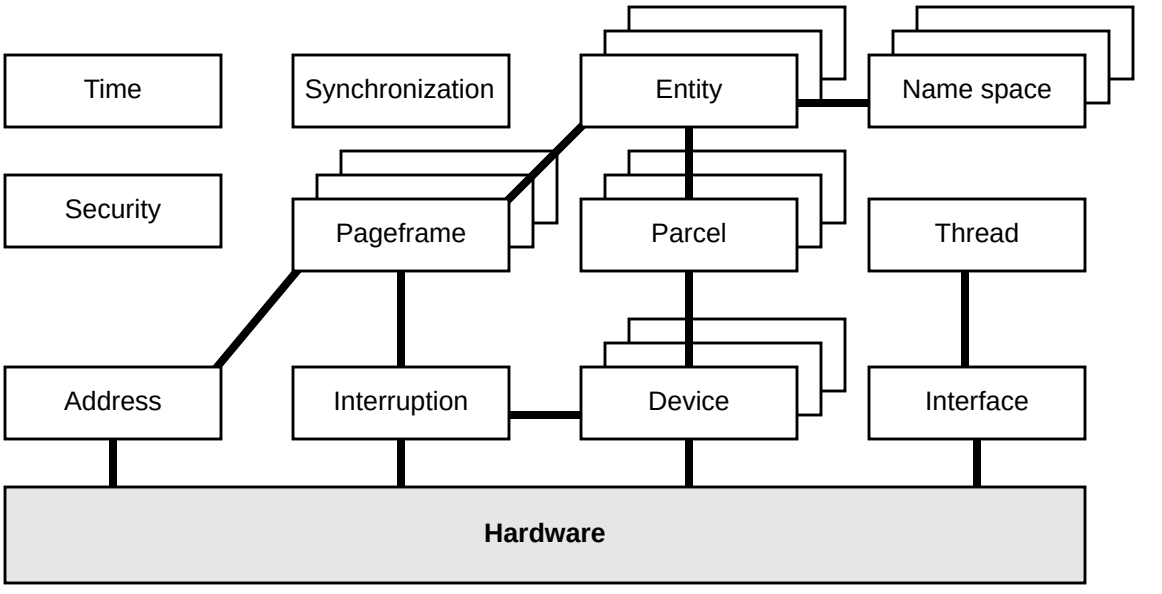

Figure 3: services in a Brevix system.

1.4 Brevix contributions

The Brevix design exploits and builds on the current state-of-the-art in operating system research, emphasizing solutions that provide scalable, fault tolerant computing. By taking this approach, the Brevix design makes several contributions. Some of these are to the state of OS technology, some to the state of the commercial exploitation of existing knowledge inside Hewlett-Packard. Here is a partial list of them:

> It is expected that each major contribution will result in one or more papers that describe the design trade-offs made and that support the value of the contribution. This list is merely a summary of those contributions.

- Full 64 bit address space support: the availability of a 64 bit address space is not merely supported in Brevix, but is fully integrated and readily utilized. Brevix supports a single system wide address space and uses that address space in ways that avoid problems resulting from virtual address aliases.
- The memory-management model: Brevix encapsulates a view of data that separates storage hierarchy management from virtual address space management, and separates both from data protection. Brevix introduces a consistent model for managing transient and persistent data.
- The storage-management model: Brevix provides a storage hierarchy. Explicit support for migration between levels of the hierarchy is integrated into this model.
- A unified procedure call model: Brevix replaces the client / server computing paradigm with the customer / provider paradigm in a way that allows application programs to ignore the distinction between local procedure calls, system service calls and remote procedure calls, and allows the implementation to optimize each kind of call.
- Secure services and their interfaces: Brevix provides a unique model for the interaction between customers and service providers, based on the concept of *interfaces*. Interfaces provide a programming abstraction that supports encapsulation, as well as an implementation mechanism which supports protection of data.
- System security: Brevix is designed to allow configurable security policies and to support a wide range of facilities necessary to implement those policies.
- Exception handling: Brevix provides a model of exception handling that allows providers to raise interface failure exceptions as an alternative to returning an error code.
- System structure: Brevix is structured to last for a long time. It is carefully partitioned to allow replacement of components and to allow extension. Considerable effort has been expended to ensure that the system framework is capable of supporting a wide range of target applications and policies, with room for more to be added later.

1.5 Road map

This section provides a road map to the rest of this design document. In addition, an overview of the features of Brevix may be obtained by reading the first section of each chapter. Section 2 on page 11 contains a discussion of topics that are common to all aspects of the Brevix design. These Brevix facilities include security, scalability, fault tolerance, and resource management. Entity management is the topic of section 3 on page 20, including an entity overview, and discussions of virtual memory and storage management in Brevix. Program execution, including threads, interfaces, exceptions and interruptions is discussed in section 4 on page 44. Brevix provides scalable, reliable name resolution, which is discussed in section 5 on page 61. Finally, section 6 on

page 74 presents a number of system support services, such as timers and communication. There is a bibliography at the end of the document in section 7 on page 78.

2 Pervasive concepts

This section describes ideas that pervade the entire rest of the Brevix design. These concepts are not individual subsystems, but are instead general “philosophies” that are followed throughout the Brevix design. The areas are: security, scalability, fault tolerance and resource management.

2.1 Security

Brevix is designed to provide good security features for a commercial environment. Brevix is not intended to be exactly an Orange Book B1 or higher system [TCS85]. In particular, we have no plans to do the formal verification needed for some of the higher levels, although we intend to structure the system to make it easy to do an informal verification and so that it may be possible sometime in the future to do a formal verification.

It is very likely that Brevix will have security features that are not called for even in an Orange Book A1 system and will be missing features required for a true B2 system. For example, the Orange Book is mainly concerned with military secrecy and deals with preventing the flow of information (covert channel analysis, etc.). Commercial systems are more concerned with authorization and logging data transformations than the distribution of data by authorized users. In a military system, it should not be possible for classified data to be disclosed by an authorized user to someone with a lower clearance, but that person can alter the data at will. In a commercial system, an authorized user may be able to read and disclose an account balance, but modification of that balance may require approval from another authorized user and may be logged. Another way of stating this difference is that military security is more concerned with data disclosure than data integrity, while in a commercial system, data integrity is of primary importance.

Finally, Brevix is designed to make it easy for a site to customize the security policies of a system to match local requirements.

Before proceeding, here are some definitions that will prove useful. A *principal* is an individual or their delegate (such as a thread). Principals are authorized to perform various operations on objects or state: we sometimes call principles *subjects*, and the things they act on *objects* (in the English, rather than C++ sense). A *right* is something that a principal is allowed to do: we speak of principals having, or wielding rights.

A general security model comprises several portions, each of which the model needs to state its policies for:

1. *Authentication*: verifying the identity of principals.
2. *Authorization*: granting rights to a principal, after doing all of the needed checks to ensure that the principal is entitled to the rights. This term also includes altering the rights needed to perform a specific operation on a piece of the system.
3. *Integrity*: checking to make sure that an object has not been altered without suitable authorization.
4. *Access checks*: verifying that a principal possesses the required rights necessary to perform an operation on an object in the system.

5. *Auditing*: automatically documenting and analyzing uses of the system that are of relevance to the security model (such as an audit trail in a bank).
6. *Denial of Service*: designing the system so that it is not possible for one principal of the system to deny another access to a service that they are entitled to use.

The two general philosophies of the Brevix security system are that as little code as possible should need to run in a privileged mode and that there should be a *security service* to implement the various *security policies* that determine whether a particular principal is authorized to perform a particular operation. All other providers of *secured services* (ones subject to security regulation) will check with the security service before allowing any restricted operations. Replacing security service modules will allow the security policies of the system to be tailored to a given site's requirements.

There is no traditional *super-user* in Brevix. Instead small-granularity rights (read any file, kill any thread, shutdown the system, etc.) are provided. While some users may be able to wield equivalent rights to those of the traditional super-user, this finer granularity allows for individual threads to have fewer rights (like only those required for backup) and makes it easier to verify the security of the system.

Figure 4 shows how a principal is authorized to acquire some rights (which involves an access check) and can then use them to pass an access check for some secure operation. This secure operation may be another authorization routine to gain yet more rights. Access checks can either be done by a software check (slow) or by hardware. For example, in the case of hardware, the possessed right would be access to a given page and the access check would be performed by the Memory Management Unit (MMU). While the hardware checks are very fast, once a principal holds a right which can be used for a hardware access check, it is in general difficult to log the use of this right. In contrast, software access checks are easier to log. Revocation of possessed rights is

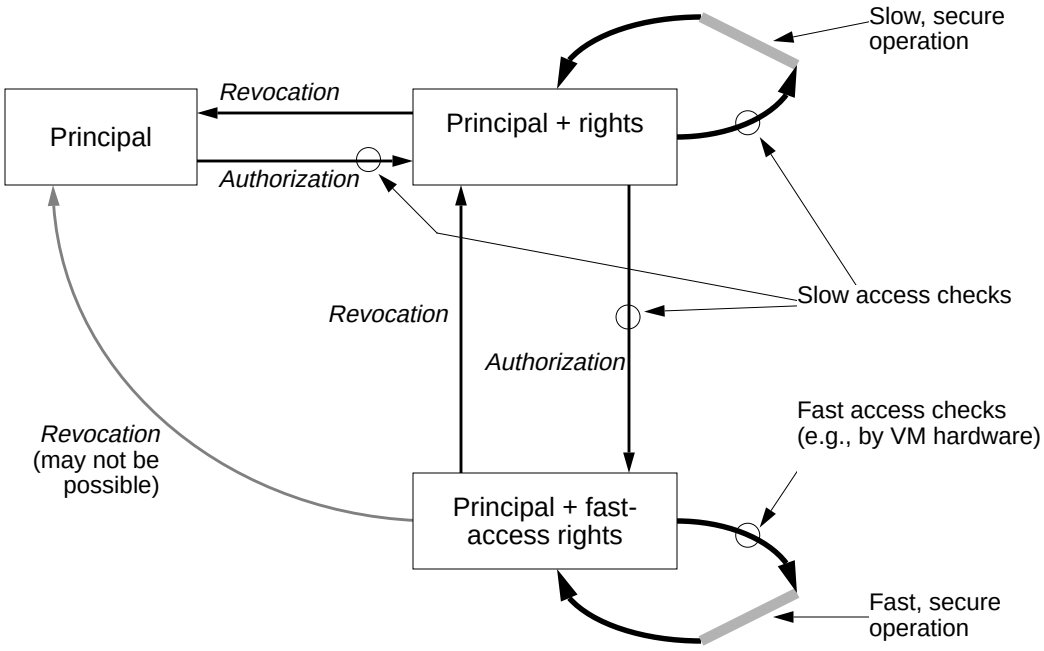

Figure 4: rights and access checks in Brevix.

in general difficult to do correctly and needs further design work. In Figure 4, a principal acquires a new right and then uses that right to acquire a "fast access" right. If the original right is revoked, the principal still has the fast access right. This problem exists any time a right is used to obtain another right. It makes no difference whether or not the second right is a fast access right or a "normal" right.

The six elements of security described above will now be discussed in somewhat greater detail.

Authentication. Authentication is the process by which the system verifies the identity of a principal. Authentication is usually only done once, when the person sits down at the terminal or the network connection from the foreign system is established. We are modelling our approach on the transitive trust model described in [Burrows90, Lampson91].

Authentication can be done in many different ways depending on the security requirements of the system. It involves both hardware (retina checks, secure "attention" keys on terminals, physical security of parts of the system, etc.) and software (the authentication service and any software encryption mechanisms that it uses).

Authorization. Authorization is the processes of granting rights (called *possessed rights*) to an active part of the system and doing all of the needed checks before granting these new rights. This may also include altering the rights (called *required rights*) needed to perform a specific operation on a piece of the system. Basically, pieces of the system (both principals and objects) have rights associated with them. Authorization is the process of altering these rights. Authorization is done by explicit routines. Some of these routines are familiar to traditional programers (like the UNIX chmod() system call). There are other routines that have not traditionally been thought of as security routines but are treated in Brevix as authorization routines. An example of this is the UNIX open() system call. In a security model, its purpose is to authorize the use of a file. Once the needed checks are done, a user acquires a new right (the open file descriptor) that allows it to read and / or write the file.

> The UNIX open() example highlights some important properties of authorization. First, while a thread perhaps should be able to read/write a given file, it does not actually have the rights until it calls the required authorization routine. In addition, once a thread acquires a right, it may not be possible to revoke the right. For example, once the file descriptor is open, no matter what is done to the file permissions (which are the required rights that were checked by the open() call before granting the file descriptor) the thread will still have access to the file until it closes the file. However, if there is a required right for the operation, it is always possible to change the required right and thus block the operation.

Integrity. Validation is the required checking of various messages to make sure that they have not been maliciously altered by anyone—that is, that their integrity is intact. Brevix has a secure interface between customers and providers so providers do not need to validate data to ensure that a third party has not altered the data on its way from a customer to a provider. It is still necessary to validate messages from external computers and possibly even validate data between a terminal and the system. (Note that we trust the internal interconnect to be secure.)

Access checks. When part of the system attempts a *secure* operation, its possessed rights are checked against the required rights needed to perform that operation on the specified piece of the system and the operation is only allowed if one of the required rights matches one of the possessed rights. This check is the *access check* and is done by the security service or its delegate (such as the processor's virtual memory hardware). Providers that export secure operations can make use of the various security modules. This both simplifies writing providers, makes the

system easier to verify, and makes it easy to change the security policies on a system by changing the various security modules.

> In the example of opening a file in the previous section, the “needed check” is the *access check*. In this example, the required rights for reading a file are a subset of: “anybody”, “a member of the group foo”, “the user bar”. For a particular file, the required rights may contain or lack any of these rights, depending on its file modes. At the time a UNIX process attempts to open a file it has the possessed rights of: “its user name”, “the list of groups to which it belongs”, and the property that it is an “anybody”. UNIX also imposes the additional requirement that all files contain a required right for the user “root”. The access check consists of verifying that one of the possessed rights matches one of the required rights. Once the process passes the access check, it acquires a file descriptor and is now authorized to read/write the file. For each read/write operation, an access check is made to verify that the process has a valid file descriptor. Note that the read/write operations only check for a valid file descriptor, they do not need to go back and re-verify the access check that was done for the open() call.
>
> In general it is difficult for an external process to revoke a possessed right once it has been granted. The given right may have been used to acquire more rights and while the original right could be revoked, extensive knowledge of the “use” of the granted right is needed to track down all the uses of that right. In addition, it is difficult to write programs that correctly deal with the unexpected loss of a right. In the open file example above, “the group foo” is a right which is used to acquire another right, the file descriptor to some file which has a group ID of “foo”. If “the group foo” is removed from the list of available groups for the process, this doesn’t invalidate the open file descriptor. In addition, in a UNIX system, there is no way to externally force a process to lose access to a file descriptor (other than killing the process). This is because it is much more difficult to write a program that behaves correctly when it randomly loses access to its open file descriptors. The decision to not require revokable rights greatly simplifies the design of various parts of the system. Note that even if revokable rights are not required, there is no reason a given manager can not implement revokable rights for a given set of rights.

Auditing. Auditing is the process of automatically logging and later analyzing uses of the systems that have a security component. Brevix’s model of a security service that can be involved on every authorization and access check makes it easy to add logging for all restricted operations. Other parts of the system may want to log additional activity (for example, for accounting), but this model is sufficient to verify and trace secure activity.

Denial of Service. Ensuring the availability of resources requires designing the system so that it is not possible for one user of the system to deny a service to another user. There is a trade-off between the amount of access a user has to system resources and the availability of that resource. In the UNIX system, disk quotas can be used to prevent denial of service attacks on disk space, but using quotas may mean that a given user will have limited access to the disk, even though there is free space on the disk. The Brevix design needs to account for denial of service requirements, but must be configurable so they can be set as required for a given installation.

2.1.1 Formal verification

We would like to informally verify the security of Brevix. While we do not intend to formally verify the system, we want to make it as easy as possible to do a formal verification at some time in the future. In order to demonstrate the security of Brevix, every operation that changes possessed or required rights (i.e., an authorization routine) must be identified and shown that it does not allow a thread to acquire or modify a right in a way not intended by the security policy. In addition, all of the trusted parts of the system must be identified and shown that they are

correctly implemented and that incorrect implementations of other parts of the system will not violate the security model.

2.1.2 Future design work

Beyond the basic model of security (the what) described above, there is the issue of identifying the operations that must be protected and the required and possessed rights for all the different parts of the system (the how). There is also an open issue on the transferal of rights from one thread to another. All of these will be attended to in future Brevix design work.

2.2 Fault tolerance

Fault tolerance or *fault resilience* is the general name for the ability of a computer system to continue doing useful work in the face of hardware or software faults or both. In order for a provider to be fault tolerant, it must both be able to ensure the integrity of its data and also be able to ensure that it makes forward progress. Brevix's fault tolerance requirements are driven by the requirements of *mission critical* commercial systems. These are systems which when down, cause significant monetary loss.

This section gives some general definitions needed to describe fault tolerance, discusses fault tolerant aspects of the OS, and gives a general description of how data integrity is provided. It is intended that Brevix will be able to satisfy fault tolerant requirements of mission critical commercial systems that have reasonable access for repair and that allow for some (even if very small) downtime. This would include ATM networks and telephone switching systems, but would not include the space shuttle (which requires zero downtime) or deep-space probes (which have no access for repairs).

2.2.1 Definitions, terminology, and the fault model

There is no single measure of fault tolerance; instead there is a set of measures that describe the behavior of the system under fault conditions. A standard reference on fault tolerance is [Siewiorek92]—as far as possible, the terminology we use here is consistent with that source.

> Two HP Labs reports relevant to this area are [Wilkes90] and [Wilkes93].

It is convenient to define here some terms that will prove useful in what follows. First, some basic terminology. In this, "the system" means an application, a Brevix system component, or the entire assembly of components, as appropriate.

- *Fault*: something goes wrong. This can be hardware or software.
- *Error*: a fault manifests itself. Note that not all faults result in errors.
- *Coverage*: the probability that the system can recover from a fault.
- *Disaster*: an error that can not be recovered.
- *Compliant*: the system is meeting a required performance level.

Next, some fault-tolerance metrics:

- *Availability*: the probability that the system is compliant at time *t*.
- *Reliability*: the probability that the system is continuously compliant between time 0 and time *t*.
- *Integrity*: the probability that there is no data loss or corruption between time 0 and time *t*.

Finally, in our model, a system goes through the following stages in experiencing and recovering from an error (Figure 5 displays this graphically):

1. a fault occurs; and manifests itself as an error;
2. the error is *detected* (its existence noticed) and *diagnosed* (its location or cause determined) by the software;
3. a *repair* is done (hardware is put back to original state);
4. and *reintegration* is done by the software. This process restores lost redundancy and / or recovers data or processing capacity that was temporarily unavailable (this step can either wait for the repair, or use standby or spare resources to complete)

We refer to the entire procedure as *fault recovery*. Once both reintegration and repair have occurred, the system is said to be *recovered*, or "back to normal".

> While *no single point of failure* is often listed as a feature of a fault-tolerant system, it is not listed in the above definitions because, depending on your definition of single, it can be very difficult to impossible to provide. It can also provide the illusion of determinacy, whereas most faults are by their very nature non-deterministic. Instead, we have chosen to define fault tolerance probabilistically. This allows the users and administrators to concentrate on requirements rather than solutions, and lets the system make appropriate implementation choices to meet the specified requirements.

If the performance level of an application or system component is above the minimum level required in the availability and reliability requirements, we say that the system is *compliant*. Otherwise, the system is *non-compliant* and the event and duration are logged (if the system is completely down, the non-compliance will be logged when the system comes back up); these count against the system when it is measured up against its non-compliance limits. Depending on the redundancy techniques used, and the performance specifications, a system may or may not become non-compliant during some or all of the fault recovery process.

2.2.2 Fault tolerant providers

In order for the Brevix OS to be fault tolerant, all the providers that compose Brevix must be fault tolerant. Describing a fault tolerant provider both describes how the OS is fault tolerant and how users can write fault tolerant applications.

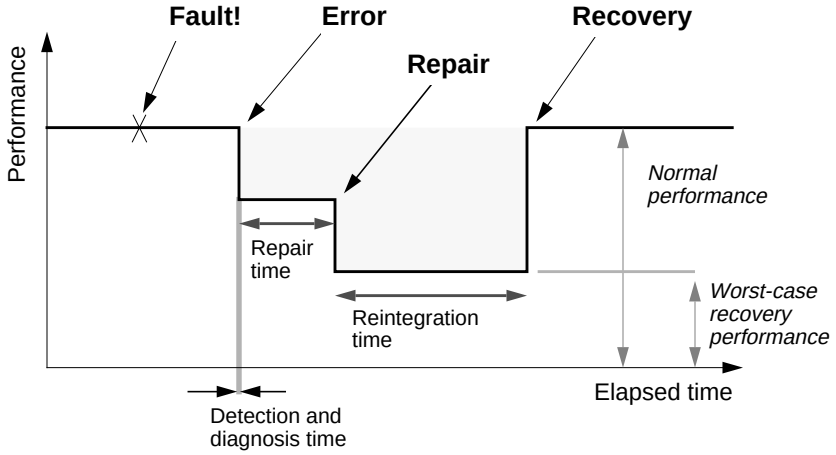

Figure 5: faults and the performance of the system during recovery.

There are two parts to a provider being fault tolerant. The first is that a provider must ensure that it makes forward progress in the face of errors. A thread can stop making forward progress at any time. This can be caused by getting stuck in an infinite loop, chasing a bad pointer and getting a bus error, or the failure of the node on which the thread is running. The thread may still be executing in the provider code that created it or it may be in some other provider—possibly on a remote node. The system may be able to report an error (exception or interface failure) or the thread may fail silently.

There are two basic Brevix mechanisms to ensure forward progress. One is the general policy that threads are not automatically terminated when an error is detected. Instead the failure is reported to an exception handler when it is detected, and this can choose to propagate the failure back through one or more interfaces (and their exception handlers), possibly even back to another node.

Exception handlers can take care of everything but death of a thread and *stuck* threads. The solution for both of these cases is to have a *watchdog* thread. The watchdog thread is notified if a thread dies and also periodically wakes up and checks on the status of other threads to make sure that they are still alive (each application must come up with its own measure of *life* so the watchdog thread can determine if the thread is still alive). In the case where a thread has died, the watchdog knows how to start a new thread to replace the failed thread. In the case of a stuck thread, the watchdog can cause the thread to take an interface failed exception. The watchdog thread may itself need to be made fault-tolerant, of course. Watchdog threads and their usage will be refined during the Brevix detailed design.

Fault tolerant applications *must* be written so that they can tolerate thread death at any time. This means that threads that manipulate data must always leave the data in a state in which it can be recovered. We hope that we can do this without having to add a full transaction system to the OS, but this is an area that is still under investigation.

The other half of setting up a fault tolerant provider is to ensure data integrity. Fortunately, this is much easier for the provider (although it requires a lot of work by Brevix). Brevix provides for fault tolerant entities. These will be described in detail in section 3.1.1 on page 20.

2.3 Scalability

One of the main features of SCS is its scalability. This section gives some scalability definitions, specifies the intended scalability of Brevix, and suggests minimum criteria for a Brevix provider to be scalable.

Definitions. *Scale-up* is the ability of the system to do more work in the same amount of time when it is given more resources. *Speed-up* is the ability of the system to do the same amount of work in less time when it is given more resources.

SCS is intended to provide for *scale-up* instead of *speed-up*. For problems that have more parallelism than currently available resources, there will be some observed speed-up, but this is actually a scale-up problem where the user cares about the total elapsed time. For example, for OLTP systems, SCS will process more transactions / second when given additional resources. In a decision support system where many activities need to be done before a query can be answered, the total time to satisfy a query will decrease as additional resources are added until the maximum parallelism of the query is reached. At this point, additional resources will not speed up the query.

The following description is summarized from [Wilkes92a]. No system will scale perfectly over an infinite range. With a small number of nodes, the cost of the interconnect will dominate the cost of the system. With a large number of nodes, interconnects that were designed for an intermediate number of nodes will stop scaling. We are interested in a metric for scalability that encapsulates how close to linear scale-up the system is capable of providing over what range. To do this, we have chosen the *pure scalability metric* from [Wilkes92a]. This is the range over which the scalability metric (performance per unit cost, where performance initially means OLTP transactions per second) is at least 50% of its best-case, or optimal, value; it is illustrated in Figure 6. The intended pure scalability of Brevix on SCS is roughly 4 to 512 nodes. We further assert that it is our goal to make Brevix provide equal or better performance / cost than a non-scalable system for at least some portion of its scalable range. This means that the *optimal* value must be at least as large as the *performance/cost* of a uni-processor of comparable performance.

Building scalable Brevix providers. The difficult question is "what makes something scale?" To begin with, the algorithm must be scalable. Since we are trying to scale throughput, it does not make sense to scale a problem for which we have already achieved the maximum throughput. Also, in order to scale, the application must have been written with scalability in mind and be able to take advantage of the SCS system. Applications will not just magically scale on their own.

Key features of a scalable Brevix provider include the ability to:

1. divide the work and the data effectively across many processors;
2. make the partitioning of work cost less than the work itself (in practice this means that the amount of work to be done is moderately large compared to the communication and setup costs);
3. avoid extensive use of non-scalable features that would bottleneck the provider (this includes overuse of "shared memory"—see section 3.2.5 on page 34);
4. avoid excessive synchronization.

Brevix vs. UNIX scalability. Standard UNIX systems are, in general, not scalable. Standard UNIX was never structured to be scalable, so making it conform to the guidelines identified above would be very hard. (For example, shared-memory multiprocessors typically need locks on important data structures. To get performance scalability, the software design must include a great deal of work to minimize the number of global locks.) Meanwhile, Brevix has been designed from the

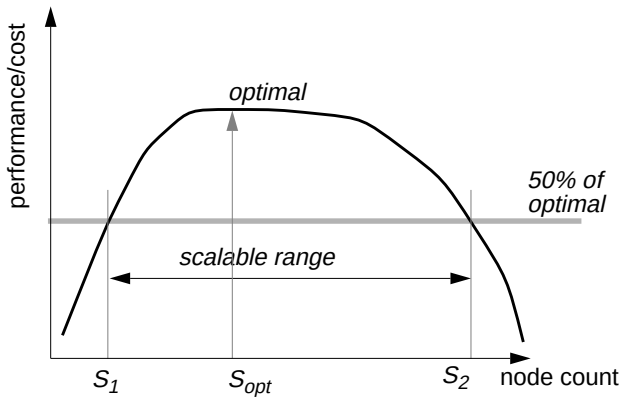

Figure 6: Brevix scalability model.

beginning to be scalable so it is much better structured, and has few of the scalability limitations identified above. Also, Brevix can provide a new API without the many scalability problems of the UNIX API, such as a single errno variable, etc.

2.4 Resource management

Resource management is done carefully in Brevix in order to assure scalability and fault tolerance. While each resource manager will be customized, there are some general philosophies that will help meet the scalability and fault tolerant goals. This section describes those philosophies for both reusable and ephemeral resources.

2.4.1 Reusable resources

Reusable resources are resources that can be used multiple times, such as memory pages and disk blocks. While it is advantageous to allow all of a reusable resource to be fully used, a manager is needed to handle the resource when all instances are allocated. The policy may be simple (like the denial of service when a standard UNIX file system is full) or may be very complex (paging of physical memory). When the manager tries to free up some space (for example, because the resource is full), it will want to free up the *best* space. The definition of best will be different for each resource, but in general a manager needs to keep track of usage. Some resources are purely local (disk space on a given disk and physical memory). Local resources are fairly easy to deal with for scalability. Since they are local, by definition, they scale and they are as fault tolerant as the node, which is all that is required for non-persistent objects. For persistent objects, the manager will need to insure the fault tolerance of data structures describing the persistent objects.

For global resources, there is usually a trade-off between scalability, fault tolerance and performance. In general, higher fault tolerance requirements mean less scalability and performance. The general Brevix model for these resources is to have a fully distributed manager of the global resource that allocates fairly large chunks of the resource for local management. The chunks should be big enough that the local managers can work within their chunks for a reasonable amount of time. When dividing up the global pool, each local manager will *bid* for a section of the pool. If a local manager is short of a resource, it is possible to bid for more of that resource. The global manager can always take back resources from a local manager (although the precise mechanisms for this are still a subject of research).

2.4.2 Ephemeral resources

Ephemeral resources are resources that are wasted if they are not used. Examples of these resources include processor cycles and bandwidth on a network channel. If a given time slice is not used, it is forever lost. Unused channel capacity is likewise lost if not used.

The standard Brevix implementation will manage all ephemeral resources locally—-that is, it will not provide gang scheduling. There is nothing to prevent this being implemented if desired, however.

> One common philosophy for managing ephemeral resources is for the manager to maintain a queue of customers of the resource. At any time, the consumer at the front of the queue gets access to the resource. The manager may reorganize the queue on occasion to maintain a degree of fairness. For example, for the CPU, there may be a *run queue* of threads that are ready to use the CPU. The run queue could be periodically reorganized by the thread manager as threads change priority. In addition, when a time quantum expires, the currently running thread could be moved to the back of the queue and the new top of queue thread given to the CPU.

3 Data management

In Brevix, data is stored in *entities*, and two portions of the Brevix framework provide the primary support for entities: *memory management*, which is concerned with how entities appear in primary memory and the address space of the machine, and *storage management*, which is concerned with how entities are kept on external storage devices. Entities, memory, and storage management together comprise the Brevix data management facilities. This section describes these three in greater detail. Section 3.1 provides an overview of entities. Section 3.2 describes the relationship between entities and the virtual and physical memory hierarchy. Section 3.3 describes the relationship between entities and the secondary and tertiary storage hierarchy. Some of the interactions involving both the memory and storage hierarchy are covered in section 3.2, others in section 3.3. This chapter concludes with a discussion of device drivers.

3.1 Entities

An entity is a *container of data*. Entities contain bytes, stored at various byte-offsets—strictly, an entity is a *container aggregate*, with the byte offsets as the individual containers.

This separation of contents, containers and container-aggregates is important (because much of the design here relies on a clear separation of these concepts), and will recur in what follows several times. The relationship between these elements is called their *container relationship*, and written in this fashion:

- byte $\Rightarrow$ byte-offset $\subset$ entity

which is to be read as “bytes (contents) fit into byte-offsets (containers) in entities (container-aggregates)”.

3.1.1 Properties of entities

Each entity has two explicit parts—a data portion and an attribute portion.

- The *data portion* holds the contents of the entity: a set of bytes that can be written and read by Brevix customers and service providers that have appropriate access permissions.
- The *attribute portion* holds additional information about the entity, in the form of <key:value> pairs. Examples of attributes include the entity length in bytes, last access and modification times, protection information, and resilience and performance constraints. Some of these are *mandatory attributes*, which are present in all entities (for example, the byte length, and the protection information for the attribute list itself); the others may or may not be present. Some entity managers will allow customers to specify their own attributes.

  Note: there are no intrinsic limitations on the size of an attribute list, although the standard implementations will optimize for “small” lists (about the size of a UNIX inode, for example). This means that the attribute facility could be (ab)used to support Macintosh-like “file forks”, if desired. (File forks in the Macintosh operating system allow a single object to hold multiple kinds of data. The most common use is to separate program code from other “resources” such as screen images.)

In addition, there is a set of *metadata* associated with each entity. This set of metadata is the internal control and data structures required for the storage system’s operations. It is comprised solely of

hidden internal state, such as mapping information. The metadata is not of interest or value to any customer of the entity, and it is not accessible outside the storage system; it is listed here solely for completeness.

All entities have a unique handle that identifies them and their entity manager (see section 5.3.1 on page 64).

When an entity is first created it is *transient*, which means that it will be destroyed once the protection group to which it is attached is deleted, or if the entity is explicitly unmapped. (Protection groups are described in section 3.2.2 on page 27. Typically, a protection group is itself kept alive by (indirectly) being bound to a thread, so this means that transient entities created by a thread go away when the thread does.)

An entity may be made *persistent*, in which case it will continue to exist for a while even when it is not attached to a protection group. It may also be made transient again (which will cause it to be destroyed if it is unattached). A persistent entity may be *reaped* if it is not accessed for a (longish) timeout period. (A Touch operation or other access resets the entity's timer.) Each entity manager has its own default time-out period, but this can be overridden by an attribute associated with the entity.

> The goal of this mechanism is to provide the system a way to garbage collect entities that have become "lost".

A persistent entity may also be given a *name* in the Brevix name space (naming is discussed in section 5 on page 61). An entity may be associated with many names. Only unnamed entities may be made transient. A persistent named entity will exist as long as the name does.

When an entity is created, it is given an initial size. This size may be changed explicitly later, although the size change operation may fail if a resource limit is exceeded (e.g., no more disk space), or if the entity is currently mapped into the virtual addresses space and the size increase would result in an overlap of addresses with another mapping. Any portion of an entity that has not been explicitly written appears as zeroes if an attempt is made to read it.

Entity performance guarantees. Each entity can have associated with it a set of *performance attributes*. These can be used to specify performance *goals* (levels of service) and provide *hints* (suggestions or additional information that may be used by the storage system to optimize resource usage). Table 2 shows some examples of performance attributes.

Two kinds of goals are supported:

- required, or *baseline* goals, which specify the minimum performance that the system has to *guarantee* to provide in order for an application to meet its own baseline performance goals;
- *nominal*, or best-case, goals, which define the level of performance that can usefully be taken advantage of. The system will try to structure its resource allocations to meet these nominal goals.

Goals represent requirements on the system; hints represent expected behavior of the application. Together they act like a contract between the two parties. Both baseline and nominal hints can be provided, but with the following difference: if an application violates its baseline hints, the system is not required to meet its baseline performance guarantees.

The baseline and nominal performance attributes are associated with the entity itself, and are called *permanent attributes*. In addition, *transient attributes* are used to describe the performance

Table 2: sample performance attributes.

	attribute	*metric*	*example*
goals	sustained bandwidth	bytes/s (and optional minimum duration)	2.2 MB/s 20 s
	max latency to first byte	seconds	500 ms
	max latency to more bytes	seconds	50 ms
	mean time to more bytes and max jitter	seconds	mean: 50 ms jitter: 1 ms
hints	R/W ratio	fraction of requests that are writes	0.1
	request transfer size	bytes	256 KB
	access pattern	(one of a list)	*append-only*

goals desired for a particular set of accesses, such as a specific time that an entity is bound to the virtual address space. Since the transient attributes are only goals, not requirements, they may exceed those specified in the permanent ones.

Attributes that are not specified will (be treated as if they are) given some suitable default value, which can be changed by the system administration facility; there may also be short-hand mechanisms to specify particularly popular combinations.

Attributes may be changed by an (authorized) thread making a request of the entity manager; in some cases, this may result in a "unable to meet guarantees" response.

Entity resilience guarantees. We define *resilience* to mean the combination of availability and reliability. *Availability* is the likelihood that the entity will be accessible at some particular time; *reliability* is the likelihood that it will have remained accessible between some initial moment and the sample time.

In addition to the performance attributes associated with each entity, resilience attributes can be specified as well. Most systems make the user or system administrator specify the mechanism by which resiliency is provided ("mirror this"). Unfortunately, making a good choice requires a great deal of knowledge about likely failure rates, system configuration, and system load. None of these are static, so keeping the system well configured is a continuous burden, which is not an efficient use of a person's time.

Instead, in Brevix, a user or system administrator defines the resiliency specifications in a goal-driven form ("this file must be unavailable no more than 30 s at a time"), and Brevix and its storage managers do the rest. They determine the necessary level of redundancy required to meet the stated resilience goals, taking into account system configuration knowledge and fault rate data that may not be readily available to the system administrator. By doing so, the burden of specifying the solution is removed from people, leaving them to concentrate on the problem specification. This also allows new resources to be taken advantage of automatically without administrative intervention, and for desired levels of reliability to be maintained even in the face of upgrades, failures, and other system events.

Brevix generalizes and extends the model of System Managed Storage [Gelb89]. The basis for the design described here was put forward in [Wilkes91][1]. This has been refined and simplified for use in Brevix.

The resilience attributes that can be specified for entities are described in Table 3.

Table 3: sample resilience attributes.

constraint	*meaning*	*metric*
baseline performance	minimum performance for the system to be compliant	set of performance goals
noncompliance	total duration and rate at which the system fails to meet its baseline performance goals	• seconds per year & • occurrences per year + mean time to violation
max data loss	bound on the amount of updates that can be lost from a fault	• amount of data (# bytes), or • time since update, or • "whole transaction", or … + mean time to violation
cost constraint	amount of resources (e.g., space) to use with respect to a non-redundant entity, and the metric to vary in order to reduce cost	• cost bound & • metric to relax

Two of the constraints (noncompliance and max data loss) are expressed probabilistically, with a mean-time-to-violation parameter. (You might like to think of these as similar to flood zone specifications: if a house resides in a "100 year flood zone" you can predict the expected rate at which the house will be flooded.)

The idea behind the max data loss constraint is to limit the amount of data that can be discarded once an update has occurred—for example, as a result of data remaining in volatile memory. For applications that require more control over entity resilience, Brevix provides a synchronous Force operation that propagates any changes to an entity far enough that the persistence and resilience guarantees associated with the entity can be met. (For example, it may cause a modification to be propagated to a piece of non-volatile RAM and a disk attached to a separate node before returning.) Optional arguments to the Force operation can express lower limits than those specified in the entity's permanent attributes. (For example, a particular Force operation on a highly-replicated entity could indicate that two-way redundancy was enough, and updates to the other copies could be begun in the background by a subsequent Force in a separate thread. These relaxed forms of resiliency are specified by providing an explicit data loss metric; a Force with no such argument is equivalent to a Force with the data loss metric defined in the entity's attributes.)

The cost constraint allows an application to specify that it wants the best resiliency that the system can provide within the limits of a bounded amount of storage space or cost (e.g., NVRAM may cost more to use than disk). The application defines "best" by specifying which other metric may be relaxed in order to meet the cost constraints.

Brevix will also provide support for transactional consistency and snapshots (checkpoints), but the details have not yet been determined. This is an area for further design and prototypes.

1. Also available as technical report HPL–CSP–90–6.

3.1.2 Accessing entities

Entities can be accessed and manipulated by a thread in two ways. The first one is to map them into portions of the address space. The second is to perform explicit Copy operations from one entity to another.

The map interface to entities. The content of an entity is made accessible to a thread once it *binds* the entity to the virtual address space. This binds the entity to a contiguous portion of the virtual address space for the duration of its binding. Such an entity may be used as the source for instruction or data fetches or as the target for stores, depending on the access rights associated with the entity when it was bound to the address space. Binding an entity to the address space also associates it with a protection group (section 3.2.2 on page 27)—these are logically separate operations, but occur together in the common case. Only threads that are allowed access to a protection group may access the entities currently associated with that protection group.

An entity that is currently bound to the virtual address space is *active*. When a persistent entity is unbound it becomes *passive*, while a detached transient entity is deleted.

The copy interface to entities. The copy interface allows a portion of one entity to be copied into another—even though the entities are not mapped. (Either or both may be, but they don't have to be.) A Copy operation takes as arguments a source entity, a start byte-offset within it, and a length, together with a target entity and a start byte-offset. After the operation completes, all future references to the portion of the entity that was the target of the Copy will see the contents of the source entity at the time of the operation. The effect of accessing the updated area or modifying the source range during the execution of the Copy operation is undefined.

Lazy evaluation of the Copy operation is allowed: the system behaves as if the copy is performed at the time of the call, but the actual copying may be deferred until a thread attempts to change either the source or target ranges, or access the target range.

Import/export of entities. It is possible to make a copy of an entity for transport outside of a Brevix system, by emitting it as a byte stream, a process known as *exporting* the object. An entity may be exported in Brevix external transfer format, which causes only actual data to be exported along with a map of the holes (portions that have never been written to) and the entity attributes. An entity may alternatively be *expanded*, which causes it to be copied as a byte stream with all of its holes replaced by actual zeros, and the atrtributes suppressed.

The Import operation restores entities from their exported format; if the external transfer format was used, the entity attributes may be restored if desired.

3.1.3 Entity managers

Each entity is controlled by a single *entity manager*, which controls the mapping of the entity's data onto external (secondary and tertiary) storage, in cooperation with other storage system managers.

Although all entities have the same basic structure and interface, different entity managers may provide additional services, such as reliability guarantees and performance optimizations. For example, one entity manager might be optimized for handling large numbers of small files; another might provide explicit support and optimizations for continuous media, with its strongly sequential access patterns and needs for guaranteed data rates.

The Brevix entity manager plays many of the roles traditionally associated with a "file system" in a conventional UNIX implementation: data layout, access mechanisms and control, and prefetching. There can be many different entity managers in any one Brevix system (indeed, at any one node). This allows (for example) each to be optimized for different access patterns, rather than having a single entity manager trying to support disparate access patterns: this can have substantial benefits in terms of reduced complexity and increased performance [Muller91].

A single entity manager may span multiple nodes—that is, it can be invoked from any of them. The details of how this is done are a matter for the implementation of the entity manager itself: at one extreme, a simple remote procedure call stub plus a centralized implementation may be used; at the other, identical copies of the entity manager's code may exist at all nodes, with the entity manager coordinating its own distributed buffer-coherency protocol.

The entity manager for an entity is selected when the entity is created—either by explicitly naming the manager (discouraged) or by specifying performance or resilience attributes or both to an *entity selection service*, from which Brevix will deduce an appropriate entity manager. The selection service returns the identity of the entity manager, and the creation call can then be made directly to that manager. If no attributes are provided, a system-default entity manager will be used. New entity managers can be introduced into the system at any time, and the default one changed at will; either change will only affect entities created after the change is made.

Ownership of an entity cannot be passed from one manager to another, because each manager will use its own private layout techniques for its entities' contents. However, there will be (relatively efficient) mechanisms by which an entity can be exported by one manager and imported by another. The mechanism will have copy semantics: the new entity will be different than the original.

> Ownership is not transferable to avoid the need for entity managers to understand the internal representations of entities used by other managers.
>
> *Move* semantics will be provided by copying, deleting the old copy, and associating any names attached to the old entity with the new entity.

Storage space can be explicitly reserved for entities ahead of time to guarantee that storage space will be available when it is needed. (Although can be reserved, particular portions of storage cannot.)

Entity managers strive to balance the conflicting needs imposed by resource constraints, performance and resilience goals, and the access patterns they are subjected to. They provide data on request to the memory management subsystem, and take advantage of the persistent storage services provided by the storage subsystem to preserve the contents of the entities that they manage. The memory and storage subsystems are the subject of the remainder of this chapter.

3.2 Memory management

Brevix memory management controls allocation and use of the virtual and physical memory address spaces provided by the processor, and uses these to provide an interface to entities. The virtual memory subsystem administers the virtual address space of a Brevix system and provides protection for portions of the virtual address space to prevent unauthorized access or updates.

When an entity is mapped into the virtual address space, its contents are associated with a set of virtual addresses. These in turn are divided by the hardware into ranges (which we call *virtual pageframes*). As these addresses are accessed by threads, the virtual memory mechanisms use the

underlying hardware mechanisms to allow appropriate accesses, and to migrate pages between the primary memory and the storage subsystem.

In addition, the virtual memory subsystem supports a system-wide consistency protocol (based on the release consistency model of Munin [Carter91]) for those cases where an active entity is being accessed from multiple nodes concurrently. Each of these are discussed in further detail in this section.

3.2.1 Virtual address space

Brevix provides a *single, system-wide virtual address space,* into which entities may be mapped. Every valid portion of the address space is backed by an entity that has been *bound* to the virtual address space. The processor hardware divides the (virtual) address space into a set of *(virtual) pageframes,* which are container aggregates, indexed by byte-offset:

- byte $\Rightarrow$ byte-offset $\subset$ pageframe.

Each pageframe is capable of holding a *page* of data. A page is a vector of bytes stored in a contiguous range of byte offsets. A page may occupy different pageframes during its lifetime.

- page $\Rightarrow$ virtual pageframe $\subset$ virtual address space
- page $\Rightarrow$ physical pageframe $\subset$ physical address space

Mapping an entity into the virtual address space causes the entity to be associated with a contiguous set of virtual pageframes. A virtual pageframe is either associated with an entity or unused. In some machines, multiple pageframe sizes are supported, and the virtual memory subsystem implementation may choose to take advantage of this by (re)mapping the virtual address space into different size pageframes on the fly.

We have yet to understand fully the ramifications of multiple (and dynamically variable) pageframe sizes of the kind supported by new processor architectures.

Binding entities to the virtual address space. By default, the first (lowest) virtual address assigned to the entity (referred to as its *binding point*) is chosen by the system each time the entity is bound. While an entity is bound, its binding point does not change. The expected maximum size that an entity will reach may be supplied at binding time to ensure that sufficient virtual address space is reserved for it. If no size is supplied, the reserved range defaults to the current size of the entity, rounded up to a virtual pageframe boundary.

Because the length of an entity need not be a multiple of the page size, mapping an entity into the virtual address space may result in a larger range of addresses being addressable than is needed for the entity. The result of attempting to access virtual addresses in this range is undefined, but no implementation is allowed to violate security requirements.

Once bound, an entity may later be unbound from the address space. If it is subsequently rebound, it may be bound to a different portion of the virtual address space.

Virtual address reservations. Portions of the virtual address space may explicitly be *reserved* by a thread. Address reservations have identical durability and persistent properties to entities: they are initially transient (and will be deleted when the thread exits), but can be made persistent, and even named. When an entity is bound to the address space, an address reservation can be supplied, as well as a binding point within the reservation. This binding will succeed only if the

reservation is still valid and the necessary virtual address range for mapping the entity is not already in use.

> Virtual address space reservations allow a thread to reserve space in a portion of the address space that it may want to use in the future, perhaps because it is trying to maintain some structure that is contiguous in the virtual address space, but whose size may increase in the future. Persistent virtual address reservations allow applications (such as databases) to use persistent pointers, by arranging that certain entities are always mapped into the same portions of the virtual address space.
>
> Note: applications that persistently reserve portions of the address space will need to develop solutions to the issues involved when two previously-separate Brevix systems are merged.

A mapping that does not refer to an existing reservation is treated as if an implicit reservation was made at the time of the mapping. (This is the mechanism by which the maximum expected size of a bound entity is represented.) When such a binding ends, the implicit reservation is discarded.

The assignment of virtual address space, and keeping track of address space reservations, is performed by the *address translation service* (see page 30), a provider for which resides on every node. These providers communicate amongst themselves to ensure that assignments are system-wide.

> Note that several techniques exist for doing this assignment without requiring communication between the address translation service providers on every invocation.

3.2.2 Protection groups

Binding an entity to the address space is not sufficient to allow a thread to access the entity. To do so, the thread must acquire access rights. This is done through protection groups.

A *protection group* is a set of access rights that can be attached to ranges of virtual addresses. There need be no virtual addresses for which the access rights are currently valid. Protection groups are global: they apply across all the nodes in a Brevix system (although they may use different hardware resources, such as PIDs, at each node).

A *protection domain* is a bag of protection groups (a bag is a set in which an item can occur more than once). Each thread has associated with it a single *active* protection domain: the thread is allowed some access to an entity bound to any of the protection groups in its active protection

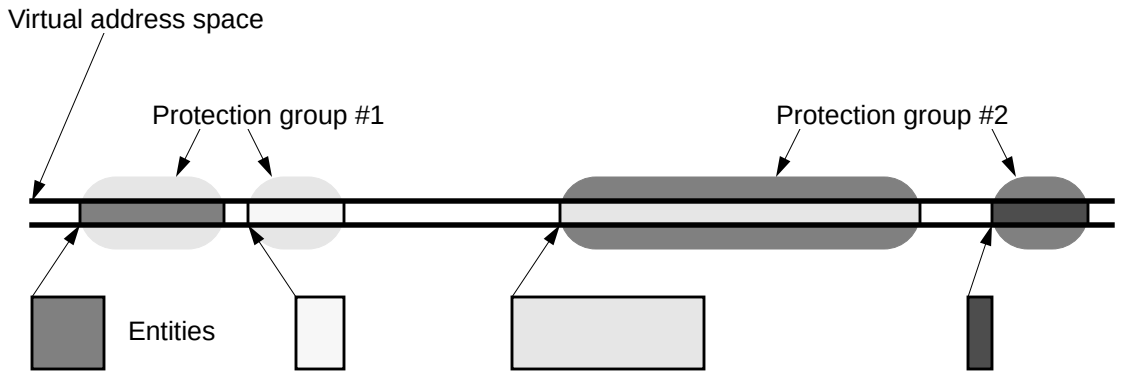

Figure 7: virtual memory management.

domain. (The access allowed to each of the protection groups in its bag is at most that allowed by the protection group itself, but may be limited to read-only, for example.)

The thread may also have one or more *dormant* protection domains that it may make active by returning across one or more interfaces. (More details of how the active protection domain is changed by interface crossings are found in section 4.2.1 on page 51.) A protection group can be deleted if it is no longer in any protection domain.

When an entity is bound to a portion of the virtual address space, the address space range is also attached to a protection group. The unit of this association is that of the reservation associated with a whole entity (rounded up to a pageframe boundary). The rights represented by the entity's attachment to a protection group may also be restricted at the time of the entity binding: for example, read-only access may be requested for a protection group that would otherwise permit read-write access. To access the entity, the thread must have the protection group in its active protection domain.

> Notice that limits to the full access provided through a protection group can be applied on both a per-thread and a per-entity basis; the security service is the final arbiter of what operations are allowed to which principals.
>
> We speak loosely of "attaching an entity to a protection group", meaning that the entity is bound into the virtual address space, and the virtual address space associated with the entity is then attached to the protection group.

When an entity is unbound from the address space, the protection group attachment is removed from the range of virtual addresses that had been mapped to the entity, any automatically-allocated address space reservation associated with the entity is also discarded, and the pages of the entity are made inaccessible.

Protection groups are assigned the access permissions they represent when they are created; these permissions can also be changed later. Brevix supports three kinds of access permissions:

- *executable*: a valid source for an instruction load;
- *readable*: a valid source for a data load;
- *writable*: a valid target of a data store.

Although these may be combined in eight different ways, Brevix imposes two constraints: (1) it assumes a Harvard memory model [Hennessy90], so that writes to an executable entity must be followed by an explicit operation before they are guaranteed to be visible to the execution (instruction) stream; and (2) writable permission also implies readable.

> The concrete machine implementation may choose to restrict access temporarily to portions of the virtual address space associated with a protection group in order to implement certain policies, such as copy-on-access, or copy-on-write. This will, however, be invisible at the abstract machine level.

3.2.3 Pages, pageframes, and physical memory management

In a similar fashion to its support for virtual addresses, the physical hardware of the machines on which Brevix runs subdivides the physical address space into *physical pageframes*. The content of a physical or virtual pageframe is called a *page*.

Physical memory in the nodes on which Brevix runs is managed by *pageframe managers* (Figure 8). Pageframe managers are in charge of handing out physical memory to other pageframe managers

and applications; establishing mappings between virtual and physical pageframes; and, together with the secondary storage subsystem, are responsible for deciding which pages to move between primary and secondary storage, and when this should be done (Figure 9).

There is a single distinguished *root pageframe manager* at each node that is given all of the physical memory on the node. This (and other) pageframe managers may have zero or more *children* on the same node. Pageframe managers can be introduced into the system at any time; they compete with each other for the attention of their parent pageframe manager for allocations of physical memory. The pageframe managers can request more memory at any time from their parent; they may or not be granted it. A pageframe manager can request return of physical memory that it has passed on to its children at any time; such requests must always be granted "immediately".

> Different pageframe managers might be used, for example, to support sequential data access, random access heaps, or database buffer pools. The pageframe managers can be provided by the system, or by the application code that operates on top of Brevix. This allows different applications to use different paging policies.
>
> The mechanisms by which pageframe managers compete are not yet well specified. We plan to investigate various forms of competitive bidding processes. The reclamation process of memory from the pageframe managers may require the addition of extra information interchanges between the pageframe managers, such as the relative costs of reclaiming different amounts of memory. The meaning of "immediately" may then be subject to negotiation for classes of pages that are expensive to reclaim.
>
> This model builds on that of Kieran Harty at Stanford [Harty92], streamlined to allow pageframe managers to nest, and for the root pageframe manager to not have a separate set of interfaces. This allows (for example) a node to use only single level of pageframe managers in certain dedicated applications, to get better performance, while also allowing considerable flexibility in the more general case.
>
> Having multiple pageframe managers means that different policies can be supported more readily, and we believe that the resulting flexibility and simplicity will provide better performance than a scheme that tries to accommodate multiple uses from a single pool of pages.

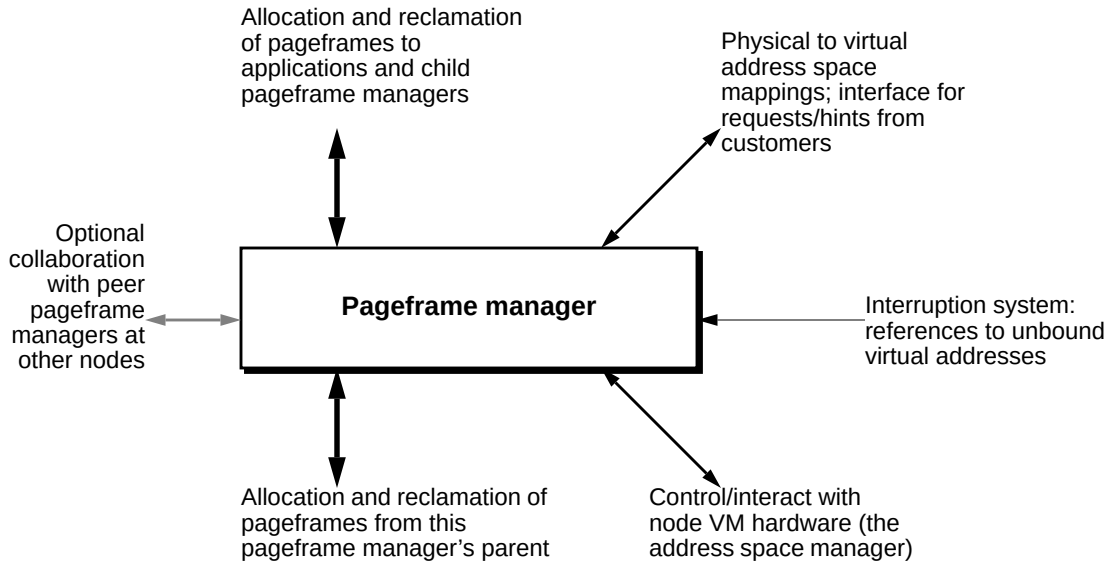

Figure 8: pageframe manager services and interfaces.

Each entity bound to the virtual address space is associated with a specific pageframe manager to take care of its primary memory needs, and with a specific entity manager to handle its secondary storage needs (see section 3.2.3). The association of entity to entity manager persists for the lifetime of the entity; the choice of pageframe manager is made each time the entity is bound to the virtual address space, using a similar goal-directed search technique to that used to select entity managers.

Address translation service. The address translation service is responsible for mapping the page protection and access status of pages into the physical memory protection mechanisms of the underlying hardware. (It also allocates virtual address space, and handles address space reservations.) There is precisely one provider for this service at each node.

> In PA-RISC, it's primary data structure is the (extended) PDIR, which it uses for recording how virtual addresses map to physically resident pageframes and access rights.

Pageframe managers may register interest in two page transitions with the address translation service. A resident page may be marked to generate an event the next time that the page is referenced by any operation. This is a *next-reference event.* A resident page may be marked to generate an event the next time the page is modified by a store. This is a *next-write event.*

When a next-reference or next-write event occurs, the thread that issued the operation is suspended before the operation is allowed to complete, and the pageframe manager receives the appropriate transition event. Once the pageframe manager has completed its operations, which may include changing the virtual to physical translation of the pageframe being referenced, the thread that caused the transition is resumed and the operation is restarted. If the pageframe manager converted the translation to an invalid translation, the operation may result in a page fault.

Pageframe managers may also request that the address translation manager change the apparent reference and dirty status of a page.

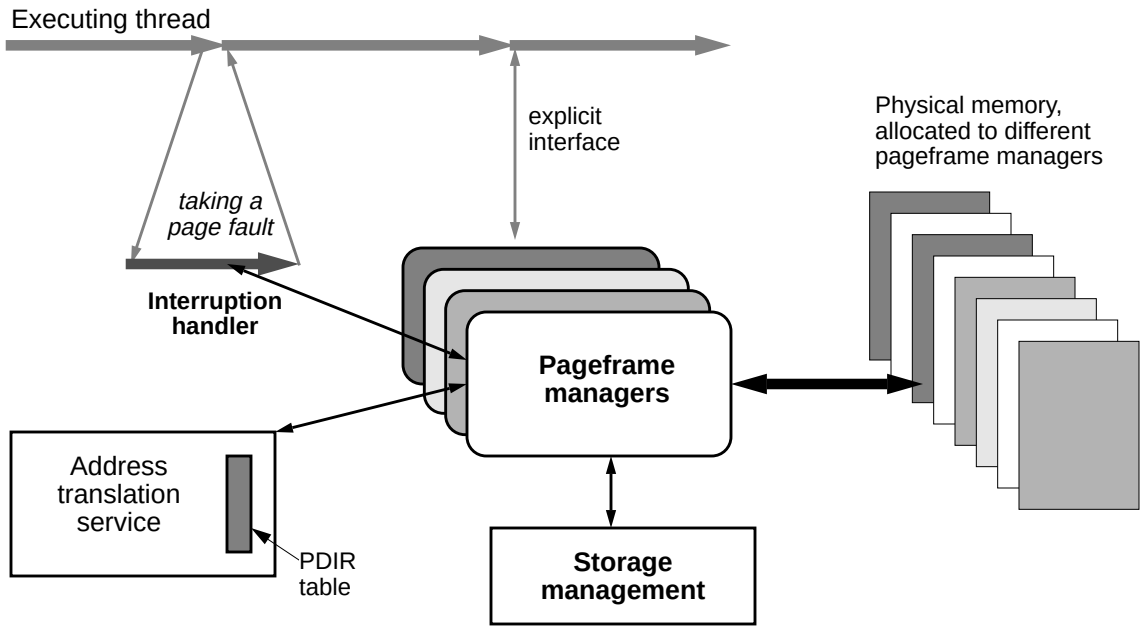

Figure 9: physical memory management.

There is an implementation trade-off related to the address manager. It is possible to require strict adherence to the architectural separation between the address manager and the pageframe managers, requiring explicit calls for each operation. This has the advantage of reducing the likelihood of damage on the part of a misbehaving pageframe manager, at the additional cost of an interface crossing for each such operation. The correct implementation choice awaits further prototypes and modeling.

3.2.4 State transitions seen by pages

This section introduces operations that the virtual memory system performs on the pages of bound entities that are accessed from a single node. We extend the coverage to include entities bound at multiple nodes in section 3.2.5 on page 34.

Entity, memory and storage managers cooperate to move pages between primary memory and external storage. In doing so, they need to keep track of whether the page has been modified or not, whether there is space allocated for it on external storage (and if so, whether there is a valid copy stored there).

The managers also keep track of access frequencies and patterns in order to help them make selections to maximize some performance metric. The selection of data to acquire and the uses to which it is put are part of the policies supported by the various managers. As such they are not further discussed in this section, which is about the framework in which the managers execute.

It is convenient to describe all these effects by means of a set of state-transition diagrams, but before doing so, we introduce some explicit operations that are performed on pages. We choose to distinguish between actions taken by the processor (Load and Store), space allocations in primary storage (Acquire / Relinquish) and external storage (Allocate / Free). In addition, pages can be copied from primary to external storage (Write) and back again (Read). Table 4 summarizes these operations.

Table 4: primitive operations on primary and secondary storage.

	"incoming"	*"outgoing"*
CPU hardware	**Load**	**Store**
physical pageframes	**Acquire**	**Relinquish**
secondary storage space	**Allocate**	**Free**
external ⇔ main memory	**Read**	**Write**

In all that follows, the state transition diagrams indicate every allowed operation. If an operation is not shown associated with a particular state, it is illegal, and the effects will be undefined. Note that threads cannot directly set or access these states: they can only cause pages to enter them. if an application needs more direct control over page state, it should supply its own pageframe manager.

Pageframe states. We begin with the transitions seen by a physical pageframe in main memory (Figure 10). Essentially the same diagram applies for pageframes in external storage.

Page states. The transitions for a page are rather more complicated: Figure 11 presents the resulting state transition diagram.

The first time a newly-created entity is bound to the virtual address space, it is bound to a range of virtual addresses, but no storage is allocated: all of its pages are *potential pages.* These pages are assumed to contain zeros and no storage is allocated for them until they are first accessed.

If the first reference to a potential page is a Load or an explicit Acquire, the page is installed in primary storage using a *demand-fill-zero* operation: a new physical pageframe is allocated and cleared to zero. Such a page is a *virgin page.* If the physical pageframe occupied by a virgin page is relinquished as a result of a Relinquish operation the page reverts to being a potential page.

If a Store operation is performed on a potential or virgin page, an *actual page* is created. If the page was a potential page, a demand-fill-zero operation occurs before the store is performed. While an entity is active, all of its actual pages always occupy either primary or external storage or both. Once a page has become actual, it never reverts to virgin or potential status, unless a thread explicitly performs a Relinquish operation on the page (and a Free if the page is also in external storage): this causes the current contents of the page to be discarded and the page to revert to the potential state. Any pageframes allocated to the page then become available for other uses.

The page can be the only extant up-to-date copy (in which case it is called *dirty*) or be one of several identical, up-to-date copies (*clean*). A dirty page should not be discarded without explicit instructions from the application; clean pages can be discarded at will by the system. Potential and virgin pages are always considered clean, even if they are the only copy.

A page can be *potential, resident* in main memory, or *external,* which means that it is stored on external storage. A resident page becomes dirty as the result of a Store operation or if an external copy is discarded.

In addition to using pageframes in primary storage, pages may also be allocated space in secondary storage. In particular, this allocation is necessary before a Write operation to secondary storage can be performed. Secondary storage space can be released by a Free operation at any time, which discards the contents of the secondary storage frame; a Free applied to a dirty page in secondary storage will return the page state to potential.

Composite operations. Various pageframe manager operations are composites, constructed from sequences of primitive operations that are performed only if needed (Table 4). For example,

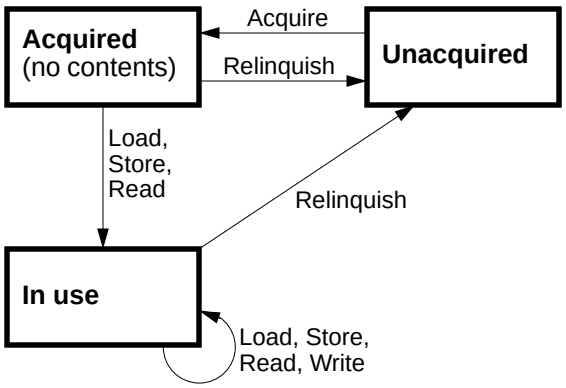

Figure 10: state transitions seen by pageframes.
Allocate and Free are valid in all cases, but cause no transitions visible in this diagram.

PageOut on a virgin page only does the Relinquish, since potential pages are never resident; the result is that the page reverts to potential.

Table 5: clumped operations on primary and secondary storage.
In each case, the primitive operations are only performed if needed.

operation	*expansion*
PageIn	Acquire, Read
PageOut	Allocate, Write, Relinquish
DeletePage	Relinquish, Free

The composite operations may also be explicitly invoked by a thread. These invocations are treated as hints by the pageframe manager. Paging out a dirty page first writes to external storage, thereby cleaning it. A PageOut on a clean page simply discards it, since it does not require a copy to secondary storage. This means that only actual pages are ever stored in secondary storage. Whether the page began clean, or became so, its primary storage is then relinquished.

When a persistent entity is detached from the virtual address space, the pageframe manager for that entity may eventually cause all of the resident pages from that object to be paged out.

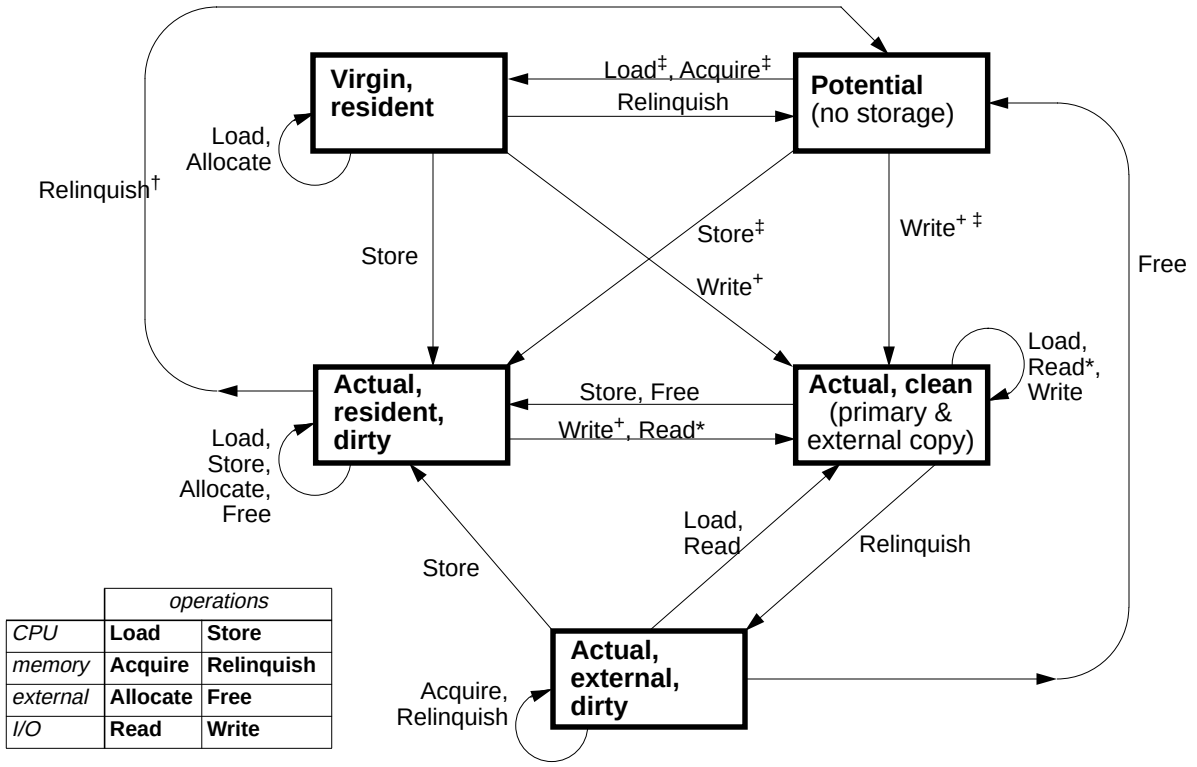

Figure 11: state transitions for pages accessed only at a single node.
Notes: *The Read refreshes the page by reading in an old copy from external storage; it is illegal if no Write has been performed. +The Write performs an implicit Allocate if needed. †The Relinquish performs an implicit Free. ‡Also zero-fills the page.

A pageframe manager may support *space reservation* in the primary storage it manages. A reservation guarantees that a certain amount of storage will be available when needed. Pageframe managers that support reservations will also provide a mechanism for releasing them.

3.2.5 Data shared between threads

An entity remains *private* as long as it is only accessed by a single thread. If a second thread accesses an entity concurrently, the entity becomes *shared.* A shared entity is *locally shared* if all of the threads that access it are executing on the same node. When an entity is accessed by threads running on two or more distinct nodes, the entity is *globally shared.*

There is no difference in operations between unshared and locally-shared entities, but consistency operations on globally shared entities are different. Brevix does not support the sharing of physical memory across nodes. Instead, it provides release consistency semantics for data that are globally shared.

> Release consistency was first described for hardware in [Gharachorloo90]. Its application to software consistency management first appeared in the Munin system [Carter91]; variants such as [Keleher92, Karp92] have since appeared, but all share the same basic philosophy.

If a page is resident on some node and a Load or Store operation on that page occurs from any other node, the page being referenced makes a transition from unshared or locally-shared to globally shared. Each node maintains its own copy of a globally shared page, and it is thus possible for these copies to differ as a result of a memory update (Store) operation at any node. When this happens, all copies of that page become *inconsistent*.

The primary feature of release consistency is a delayed update protocol that allows multiple copies of a memory page (on different nodes) to be in a mutually inconsistent state and to remain so until an explicit Release operation is invoked by (or on behalf of) one of the threads accessing the page. This operation signals the thread's desire to propagate all the updates it has made to other nodes.

> In practice, the hardware support for pageframes is used to bound the scope of the updates to a list of pages. If desired, even finer-grain bounds are then determined by comparing a previously-copied version of the page with the updated one, to determine exactly which data have been updated. Hardware support could be imagined to speed up the process of determining the areas where updates have been made on a per cache-line or per-word basis.

The release consistency mechanisms guarantee that if a thread makes one or more updates followed by a Release operation, followed by another update, no other thread will see the final update without also seeing all of the updates that preceded the Release.

Note that this is *all* that the release consistency guarantee provides. In practice, the final update is typically a lock-release, and other nodes perform a lock-claim before they attempt to read any data that might be globally shared. Since the lock-claim cannot complete until the lock-release update has been seen, this guarantees that the preceding updates before the Release will also be visible.

> Release consistency operates successfully only on correct programs that do not use out-of-band signals (such as timers) to communicate—i.e., ones that rely on correct application of locks to shared data structures. The effect on incorrect programs is undefined, but generally not what was intended.

Release consistency allows threads on one or more nodes to make changes to a memory page without propagating the changes until the synchronization call is made. Within this framework, different choices can be made for how and when updates are propagated. Brevix, like the Munin system, will allow different consistency protocols for each globally shared page.

The release consistency mechanism rounds up the sizes of all updates to some *detection granularity* for the purposes of determining what has been updated and propagating it. This granularity is settable at entity bind time to a power-of-two multiple of a byte, aligned on the power-of-two boundary. It may also be recorded as an attribute of the entity. If neither is specified, we will select a default granularity that takes advantage of any available hardware support, or a 32-bit word if there is none.

Deliberately, there are no promises of fairness, promptness, or performance in our specification of release consistency. Implementations of Brevix will endeavour to provide these, where appropriate. For example, some schemes treat Release as a synchronous "propagate to all nodes" operation; others merely enqueue a request to transmit the changes at a suitable future time; some approaches ensure consistency at release time; others at first access to a globally shared page after a release.

If it is necessary to take advantage of different release consistency policies, this will be done by using different pageframe managers, and by attaching each entity to the manager that implements the policy necessary for that entity. Notice that this means that the unit of allocation of release consistency policy is entire entities, and selected at the time they are mapped.

3.3 Storage management

The storage management subsystem provides access to the external storage devices in a Brevix system. The description here works its way up from physical devices towards the entity managers, which mediate the interactions between the storage subsystem and the memory subsystem. Figure 12 gives an overview of the entire storage system structure.

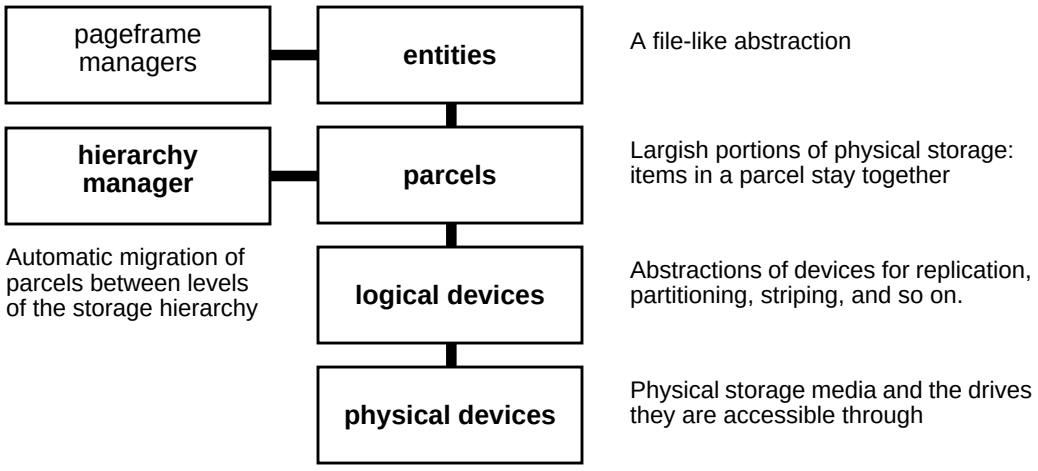

Figure 12: overview of the storage system.
Objects are shown on the right—each has an associated manager (not shown). Two extra managers that are also part of the storage system are shown on the left.

3.3.1 Devices: cartons and parcels

Ultimately, the storage resources used by the entity managers are provided by *physical storage devices*, such as disks, disk arrays, jukeboxes, tape robots, and so on. The term *device* refers to the entire storage element: that is, a device contains one or more *storage drives* into which *physical storage media* (e.g., tapes, disks) can be loaded (or are always loaded, in the case of Winchester disks). Data are ultimately stored on media. A medium has to be loaded into a drive to make it accessible. Thus:

- media $\Rightarrow$ drive $\subset$ device

Both the devices and their media exhibit a range of physical characteristics (e.g., capacity, access latency, bandwidth, reliability), and varying kinds of accessibility (e.g., tapes cannot easily be used as random-access devices, and some media cannot be written more than once). These differences are encapsulated in a set of *attributes* that are associated with each drive, device and medium. (In cases where media and drives are inseparable, like a Winchester disk drive, we will still preserve separate sets of attributes because this will simplify the code that has to handle removable media as well.)

Devices (and drives) are accessed through *device managers*, which handle the low-level details of communicating with the various pieces of hardware that represent the channels and the devices themselves. (Device managers are discussed further in Section 3.4.)

The Brevix abstraction of the storage capacity provided by a medium is known as a *carton*. A physical storage medium appears to the system as a vector of cartons (Figure 13), indexed from zero. Each carton is a contiguous area on the device into which bits can be stored; all cartons in a system are of the same (largish) size. This allows their contents to be moved between cartons with

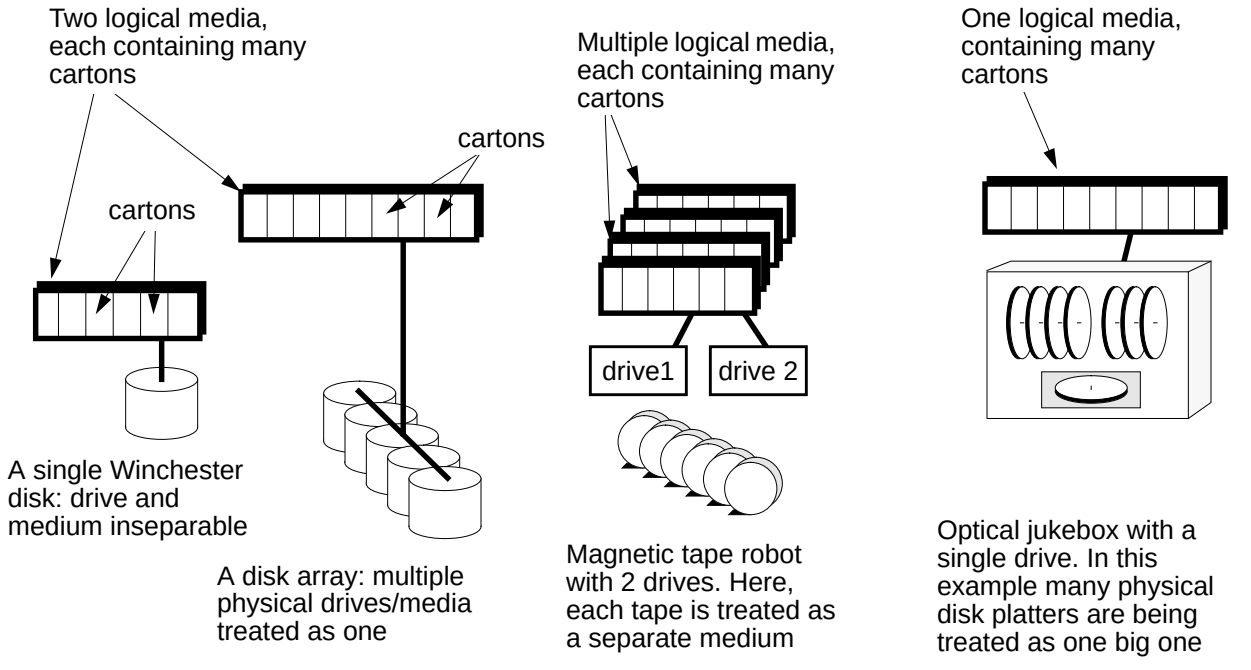

Figure 13: examples of media, devices and cartons.

ease. The static attributes associated with a carton are those of the medium the carton is part of; the dynamic ones are associated with the drive and device into which the medium is currently loaded. To a first approximation, cartons map onto storage media in a way that the "seek" time between cartons increases with increasing difference in their indices. It is expected that data layout algorithms will take advantage of this to improve access performance.

> The size of cartons is dictated by the following (competing) goals. The smaller the carton size, the less fragmentation there will be (e.g., when a storage medium is divided up into cartons, there may be some space left over). The larger the carton size, the fewer performance glitches there will be at carton-boundary crossings, and the less metadata is needed to locate the cartons in a system. In addition, we want the minimum size to be large enough to hide such boundaries even when the device is an aggregate, like a wide RAID array, where the effective carton size per device is the carton size divided by the number of disks. Altogether, this suggests carton sizes in the few-megabytes range.
>
> One of the notions behind cartons is that they represent a co-located set of storage, with similar performance attributes. You might like to think of them as an abstraction of (i.e. improvement upon) the 4.2BSD cylinder group concept, which is used to keep data and metadata together for better performance.

The contents of a carton is known as a *parcel*. Parcels can be moved between cartons; cartons are places into which parcels can be put, so they stay still.[1] A parcel acquires the media, drive and device attributes of the carton it is currently residing in. Access to a storage device can be done in smaller units than an entire parcel, down to the physical sector size of the underlying hardware device. Thus:

- parcel $\Rightarrow$ carton $\subset$ medium

The coarse-grained, fixed-size nature of parcels is chosen to simplify automatic storage migration, because there are fewer parcels to keep track of (by comparison to the number of bytes, or even 4KB-sized pages). As a result, storage hierarchy management in Brevix consists of deciding when to move parcels between cartons, and which cartons to move them to.

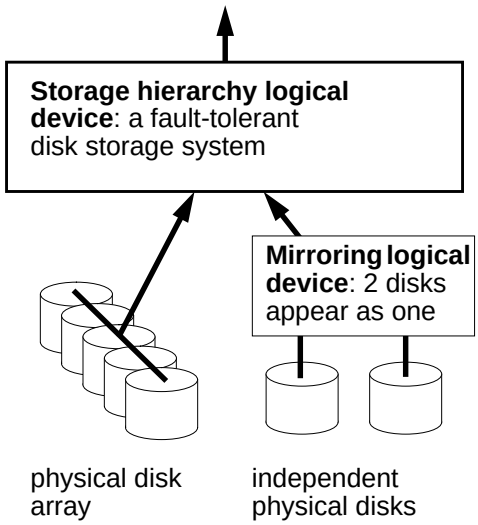

Figure 14: logical storage devices.

---

1. Actually, some of them are rotating at up to 5400RPM, but we'll ignore that for now.

In addition, raw device access will be provided to allow support of non-Brevix file system structures.

3.3.2 Logical storage devices

There are several ways that multiple storage devices can be used in unison—for example, to provide a software-implemented disk array, or to provide a hidden storage hierarchy across a set of disks and a tape robot—or even subdivided into smaller, more manageable elements, as in OSF's Logical Volume Manager.

In Brevix, these functions are provided by *logical storage devices.* A logical storage device sits on top of one or more other storage devices, and presents an upper-level interface that looks just like one or more physical storage devices—i.e., sets of cartons that have attributes and read / write operations. Since the properties of these cartons are determined by the code of the logical storage device, it may be that different cartons in a single logical device have different attributes, or even that the performance attributes of a carton will change over time.

Storage devices can be built into an arbitrarily deep tree, with the leaf nodes all being physical storage devices, and the branch nodes being logical storage devices. Some examples of the latter are as follows; Figure 14 illustrates two of them.

- *subset*: one that makes available only a portion of the cartons available to it;
- *striping*: the devices below it are treated as a single logical device with round-robin striping between (or even inside) cartons;
- *replication*: multiple copies are kept of a carton on the storage devices below this one;
- *storage hierarchy*: multiple device types can be managed as a single storage hierarchy by being made available to a device manager that will try to achieve performance of fastest at the cost-effectiveness of the cheapest;
- *partial data redundancy,* such as RAID 5 [Patterson88].

3.3.3 Parcel managers: slots and chunks

Each parcel has an associated manager, which allocates capacity within the parcel to entity managers, in units called *slots*. Each slot is capable of storing one *chunk* of data for an entity manager (Figure 15). Thus:

• chunk $\Rightarrow$ slot $\subset$ parcel

This two-level structure of chunks and parcels allows different optimizations to be applied at the different levels. There are far too many chunks of data to consider keeping an on-line index of them in a tertiary storage system that is handling terabytes of data—instead, the large units that are parcels can be used for this. On the other hand, it is desirable to have a tight handle on where active data is located, and the coarse granularity of parcels is insufficient for this. The combination of chunks and parcels provides the best of both worlds: a space-efficient representation for infrequently-accessed data, and a performance-optimal scheme for active data.

Just as with entities, the manager of a parcel does not change over time. Since parcel managers look after parcels, which may move between cartons, parcel managers are not associated with particular cartons or devices.

The simplest parcel manager exports a vector of fixed-size slots. The size chosen need not be that of a page—in particular, better performance may be obtained if slot boundaries coincide with physical media boundaries such as tracks, which are typically 30–40KB in size. Other parcel

managers will offer a range of slot sizes—possibly even arbitrarily variable—and manage the allocation problem for the entity managers, their customers.

> Note: having explicit parcel managers allows multiple entity managers to share a single parcel. In turn, this makes it easier to support different kinds of access pattern in a single parcel (e.g., both short control files and large sequential data files can be co-located for better performance), which will be especially important when tertiary storage support is included. If there were no explicit parcel managers, each entity manager would be required to do its own slot-in-parcel allocations. An undesirable side-effect would be that only the attributes supported by the single entity manager owning the parcel could be supported.

One reason for supporting variable-sized slots is to allow data compression as chunks are written into parcels. In this case, slot sizes will be determined by the size and compressibility of the contents of the associated chunk. Log-structured like techniques [Ousterhout89, Rosenblum92, Burrows92a] will have to be used to handle data overwrites, since a chunk with new data may be larger than the slot originally assigned to it.

3.3.4 Hierarchy managers

Parcels can be moved between cartons in order to implement a storage management policy. Typically, the goal of such a policy is to maximize performance while minimizing cost, and this is usually achieved by using several different kinds of storage devices—with higher performance typically associated with higher cost-per-stored-bit. In the Brevix storage model, a *hierarchy manager* is responsible for making decisions about which parcels live in which cartons.

To support the policy decisions of the hierarchy manager, the parcel managers keep track of performance-related attributes such as access frequencies and latencies.

> A hierarchy manager operates on a set of devices, and the parcels that they store. Since hierarchy managers are required to optimize access characteristics against physical device

---

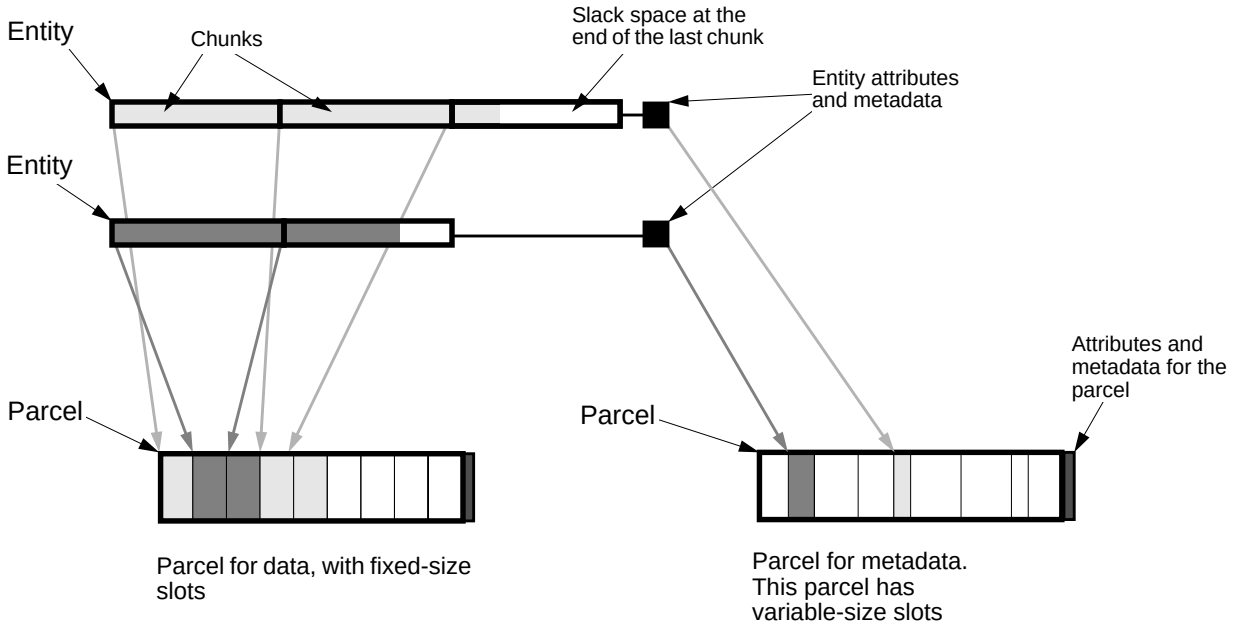

Figure 15: slots, chunks and parcels.

performance limitations, it is unlikely that the best performance will be achieved if a single physical device is given to more than one hierarchy manager.

3.3.5 Entity managers again

We can now revisit the role of the entity manager in the Brevix storage system framework. Each entity manager, while it is active, has available to it a set of parcels in which it can store data. (The parcels come ultimately from the logical devices made available to the entity manager, with the assistance of the hierarchy manager. As noted above, some parcels may also be in use by other entity managers.) The primary task of the entity manager is to map the entities they manage into chunks, and thence into parcels. Much of this work is concerned with maintaining a mapping of data to parcels that maximizes performance and / or resilience goals, possibly under explicit constraints expressed in the entities' attributes.

Typically, adjacent portions of an entity will be placed in near-adjacent portions of a parcel. Sophisticated entity managers may also take advantage of the different device attributes of different parcels to store different portions of their entities' data, attributes and metadata.

It is convenient to introduce the notion of a *chunk vector*, which is a set of chunks that are manipulated as a unit. Typically, entities are mapped one-to-one to chunk vectors. An entity manager optimized for small entities might place multiple small entities into a single chunk vector, or even a single chunk (cf. the block fragments of the BSD fast file system).

Some entity managers may shuffle chunks between or inside parcels in the background in order to improve performance once access patterns have become visible through use [Ruemmler91]. In the case of log-structured entity managers, background cleaning will also have the effect of moving chunks between parcels.

3.3.6 Storage subsystem review

Here's the story so far. Entities are divided up into chunks, which are placed in slots in parcels. The parcels are themselves fitted into cartons, which are portions of devices.

The entire set of container relationships can now be assembled (Table 6); and the overall framework structure displayed (Figure 16).

Table 6: mappings inside the storage subsystem.

item	*container*	*container-aggregate*	*item ⇒ container mapping done by*
byte	byte-offset	entity	application
byte	byte-offset	pageframe	application
chunk	slot	parcel	entity manager
parcel	carton	medium	hierarchy manager
medium	drive	device	device manager
page	virtual pageframe	virtual address space	application
page	physical pageframe	physical address space	pageframe manager

There are several functions that can be performed at multiple places in this framework. We need more experience before we can decide whether this is unnecessary duplication of opportunity—i.e. there is one “best place” for each function—or whether this diversity is indeed as fruitful as it seems to be. For example:

- *compression*: as chunks are put into parcels, and / or as parcels are put into cartons (e.g., at the device level)
- *replication*: at the entity level (on a per-entity basis as a function of resilience attributes), at the parcel level (some parcels are marked as “keep two copies”), and / or at the device level
- *partial data redundancy*: at the physical and logical device levels (e.g. hardware or software RAID arrays, the latter implemented as a logical device)

Examples of how other file systems fit into this framework. This section contains some examples of how different file systems might be mapped into this structure.

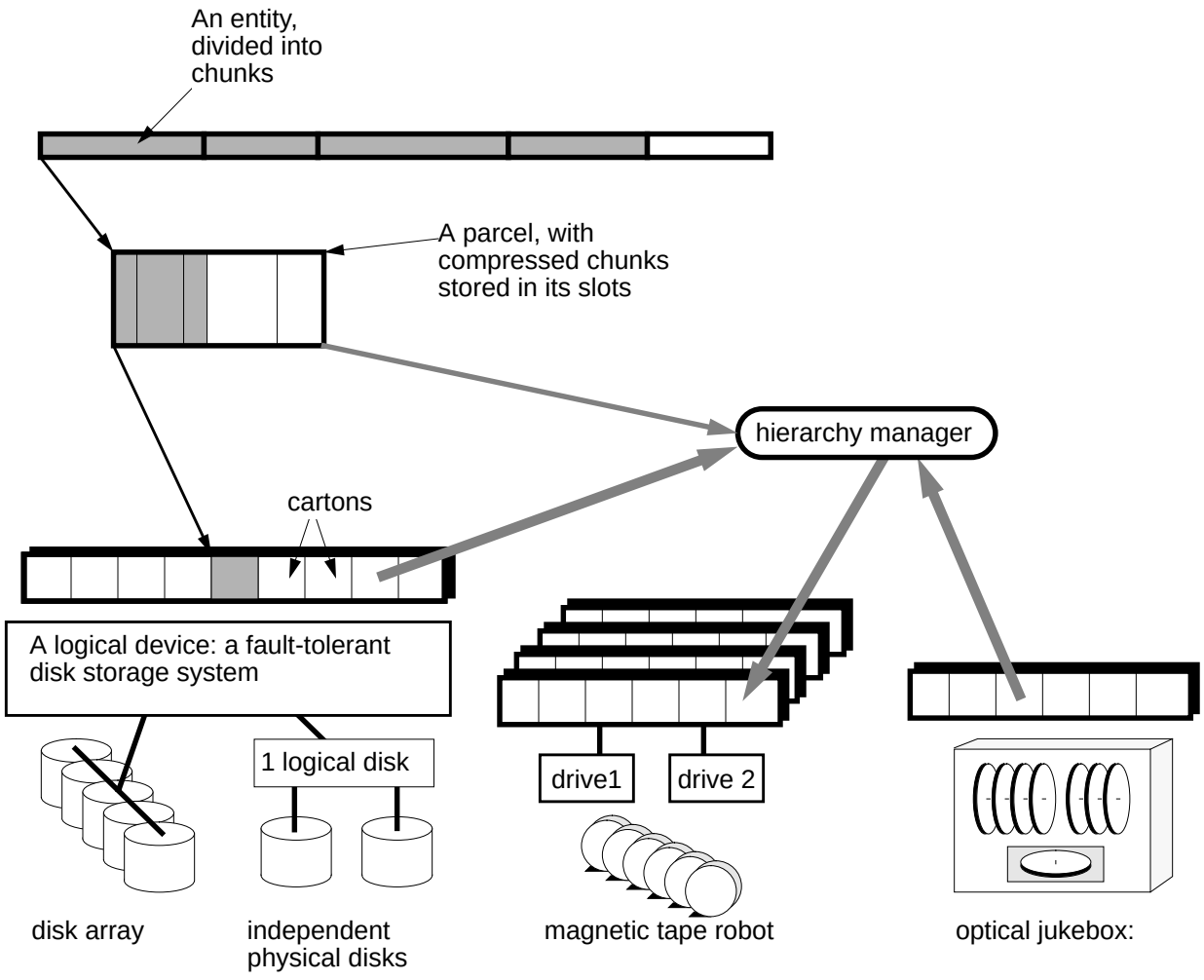

Figure 16: complete Brevix storage model.

- 4.2BSD [McKusick84]: only one entity manager; cylinder group = parcels; metadata for the data held in a parcel is kept in the same parcel (directories, cylinder bitmaps, inodes); no parcel movement between cartons, no tertiary storage.
- HP-UX [Clegg86]: logical volume manager and software striping = logical device, otherwise like 4.2BSD.
- Episode: one set of parcels is used for log-based metadata, the rest are used for in-place data update (this is very like Figure 13).
- LFS [Rosenblum92, Seltzer93]: an LFS segment = a parcel; interleaved data and metadata in a log-like fashion; parcel cleaning used for garbage collection.
- HighLight [Kohl93]: like LFS, except that parcels can migrate off to tape, and aggressive co-location cleaners are used to gather data and metadata into a parcel before it is migrated.
- Cray track-based file system: this used sector-sized allocation for small files, and track-sized allocations for large ones. In Brevix, the nearest to this would be a parcel manager that supported two allocation sizes for its slots: sectors and tracks.
- Multi-structured file system [Muller91]: different entity managers for different types of entities (file data, continuous media data, and control—i.e., metadata).

3.4 Device managers

Device managers are the Brevix abstraction for physical device drivers: there is a single device manager associated with each physical device (although one device manager may handle multiple devices). The device manager encapsulates the device's hardware-specific characteristics, and exports access to the device through an I/O interface. Device managers may

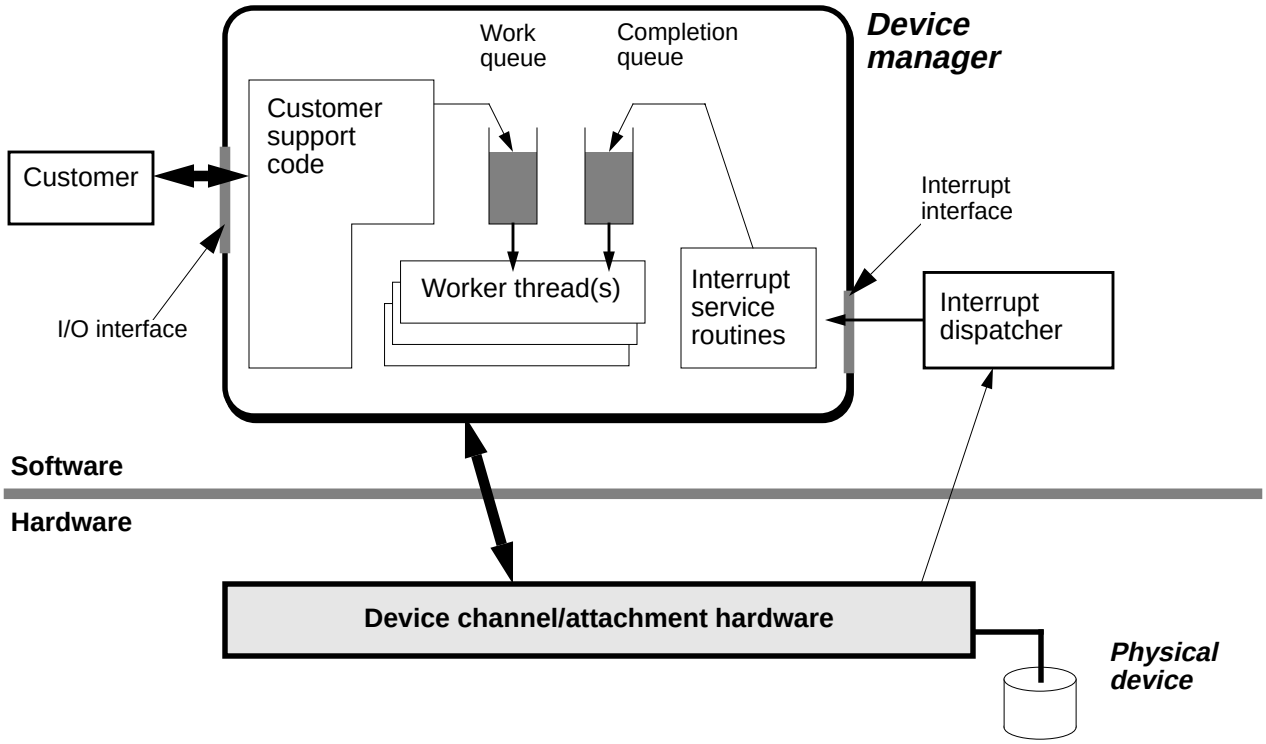

Figure 17: Brevix device manager model.

share access to a common physical path, or channel, to their devices (e.g., a set of disks on a SCSI bus). The organization of a typical device manager is shown in Figure 17. It has two interfaces: an interrupt interface that provides access to device-specific interrupt handlers (see section 4.3 on page 55), and an I/O interface, which is the one invoked by the customers of the device. At device manager installation time, the interrupt interface is bound to the interrupt dispatcher, which will dispatch a call to this on an interrupt from the associated device.

At I/O request time, the customer thread makes a call through the I/O interface. If the device is idle, the request is initiated; if it is busy, the request is added to a queue of work requests, and the calling thread blocked until the request completes. The work queue is emptied by one or more *worker threads*, private to the device manager.

Since the I/O interface is a regular Brevix interface, a remote attachment to it will provide cross-node access to the I/O devices supported by a device manager.

Interrupts are handled on the interrupt control stack, with as little code as possible being executed before the interrupt can be converted into an element on a (high-priority) completion queue for the worker threads to handle. An interrupt service routine will not be interrupted by a second device I/O interrupt for the same device, may not yield the processor for any reason, and may only Raise events, and (perhaps) Signal condition variables (section 4.1.5).

4 Program execution

In this section, we describe the Brevix environment in which programs execute. The three important concepts are threads, interfaces and interruptions.

The *thread* is the fundamental notion of computation in Brevix. Threads can be thought of as an instruction stream that is executed by an instruction set processor. All threads have access to a single, common virtual address space. Threads may cooperate though the use of various synchronization mechanisms. At any one time, a thread is *executing* (or active) on at most one processor, but may, during the course of its lifetime, visit different nodes via remote procedure calls. Threads are more fully described in section 4.1.

Encapsulation of operations and possibly data is supported in Brevix though the concept of *interface*. An interface is the boundary between a customer and a provider for some set of services. Binding can occur at runtime (and has to, for protected interfaces.) Code executing in the customer role will invoke the provider services using a subroutine call that appears quite ordinary, but which can result in the thread both acquiring and relinquishing access to selected protection groups. In this way, Brevix allows the manipulation of sensitive data to be made in a protected manner without recourse to expensive context switching. Interfaces are more fully described in section 4.2.

When a processor finds an anomaly, it undergoes a transition and begins to execute code at some predetermined location. In Brevix, these occurrences are called *interruptions*. They can occur for a number of reasons, many of which are processor-dependent. For this reason a complete description is impossible. Brevix does, however, provide a general architecture for interruption handling that will be followed on all implementations. One important feature of this architecture, called an *exception*, is a mechanism for some interruptions to be delivered to provider or application code where an action can be taken and the thread restarted. Interruptions are more fully described in section 4.3.

4.1 Threads

The lightweight thread is the mechanism that implements computation in Brevix. Multi-threading is accomplished through multiple processors and by multiplexing processors against a pool of threads. A thread executing on one node may visit (execute on) another node as a result of a *remote procedure call*. A thread has a single unique identity no matter where it is currently executing; in fact, we say that it exists on all the nodes that it has visited but not yet returned from. Termination of a thread is an event that is detectable by other threads.

There are four components to a Brevix thread. They are the *machine state* (program counter, data registers, etc.); a *protection domain* (a list of protection groups the thread has access to); some information regarding the thread's response to exceptional conditions; and various metadata (current run state, security information and thread id, etc.). On a thread switch, only the machine state and a pointer to the other items need be manipulated. Thus thread switch is a very fast operation. Note that an address space is not a property of a thread since Brevix provides a single system-wide address space. In traditional systems, much of the information about objects

associated with a process are stored with the process, for example open file tables. In Brevix, this type information typically resides with the manager of the object, e.g., the entity manager.

We start with a description of how processors are multiplexed between threads, and the various states of a thread. We then discuss Brevix facilities for basic thread management. The next two subsections describe the fundamental thread components of machine state and protection groups, followed by the thread synchronization primitives available.

4.1.1 Multi-threading

A thread with work to do is called *runnable*. The runnable threads on a node may outnumber the processors. In this case, processors are multiplexed among the runnable threads. A thread may choose to relinquish a processor by yielding or by making a call that blocks the thread. In some (most) installations, a thread's turn on a processor might also be preemptable. Reasons include the expiration of a time quantum or some activity within the system such as an I / O completion. When a runnable thread is assigned to a processor it is said to be *running*, otherwise it is said to be *ready*. Threads that have asked to wait for a specific synchronization occurrence are said to be *blocked* and are not runnable. A thread may also be put in another nonrunnable state referred to as *suspended*. Facilities detailed below are used to control this state. A thread may be both suspended and blocked. Figure 18 displays the Brevix thread states and their transitions.

When a processor is to switch threads, the Brevix *dispatcher* is invoked to disengage the current thread from the processor and to run another one. The dispatcher adds the current thread to the collection of ready threads known as the *run pool.* A thread is then selected from this pool and becomes the next thread to run. The dispatcher is not aware of any threads other than those it places in and removes from the run pool. It performs no other operations on the run pool. Each

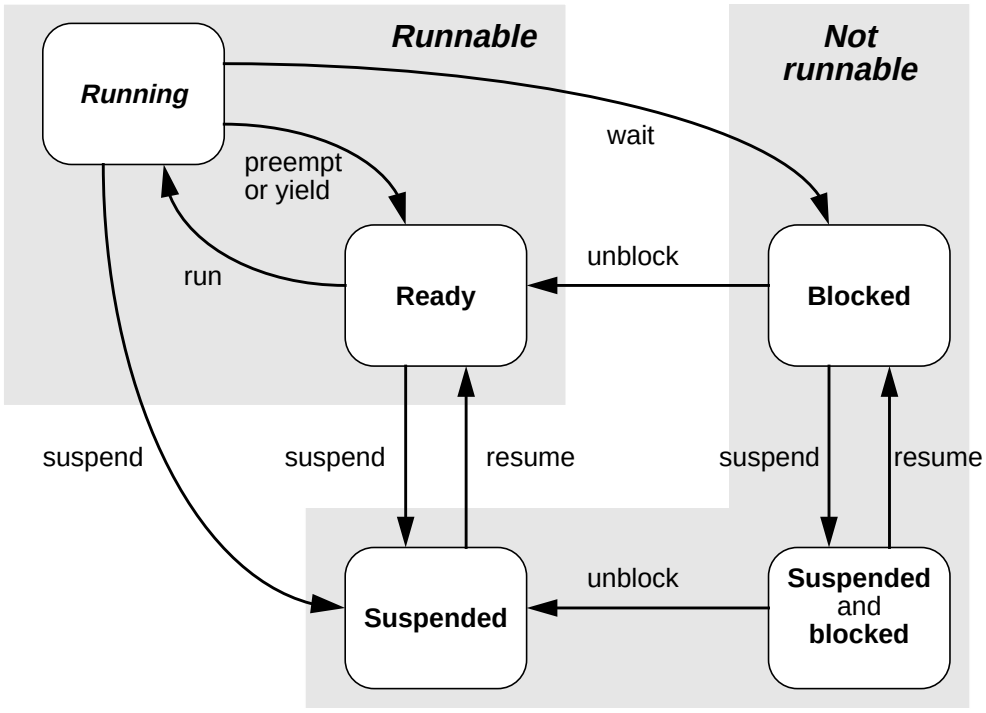

Figure 18: thread state diagram.

node has its own run pool, implying that processor multiplexing is a local activity. There is one dispatcher per node.

> Multiple priority levels can be supported by dividing the run pool into a run queue for each priority level, and making the dispatcher operate on the highest-priority run queue with any threads in it.

The organization and contents of the run pool are determined not by the dispatcher but by the *run pool.manager*. The run pool manager executes occasionally to add threads to and remove threads from the run pool and to reorganize its contents. The dispatcher need only know enough about the structure of the run pool to select an appropriate thread at switch time. Thus, it is the responsibility of the run pool manager to implement the *thread scheduling policy*. Brevix does not impose its own scheduling policy, but rather allows the selection of a policy from a set of choices, some or all of which may originate on site. For example, one may supply policies for priority queues or gang scheduling. Multiple simultaneous policies are allowed and will be handled by a scheme similar to POSIX.4. Brevix will come with at least one policy that can be used as a default. When a thread is blocked or suspended, it is not runnable and is removed from the run pool by the run pool manager. When circumstances are such that this thread should continue, it becomes runnable and is returned to the run pool, eventually to be run by (or dispatched to) a processor.

4.1.2 Thread management

Brevix provides facilities to create and manipulate threads as follows (see also Figure 18). Threads are global in scope and may be manipulated from any node. Thread_t and Node_t are opaque types whose values denote threads and nodes respectively.

Thread_t threadCreate();

> Creates a new thread and returns a reference to it. When a thread is first created, it is suspended, has a machine state that is undefined, has an empty protection domain and no exception handlers. These items must be set before the thread can be made runnable. The only metadata defined is the thread id and the run state.

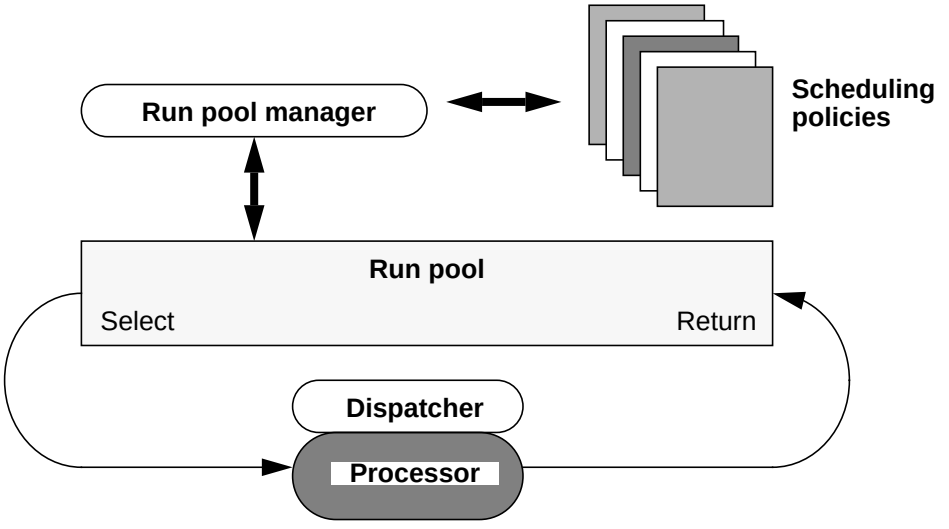

Figure 19: run pool manager and the dispatcher.

The mechanism by which a thread is created at a particular node is under discussion: the choices are to provide a node identifier in the threadCreate call, or to always perform the creation locally and rely on inter-node manager invocations to make the selection.

void threadResume(Thread_t t);

Attempts to make thread *t* runnable. If the thread is not suspended there is no effect. Resuming a suspended thread that is not blocked makes the thread runnable. Resuming a suspended thread that is blocked leaves the thread blocked but not suspended. This routine is used to start new threads and to restart suspended threads.

void threadSuspend(Thread_t *t*);

When this call returns, thread *t* will have been suspended. If the thread was ready, it will be removed from the run pool and becomes suspended. Running threads marked suspended are not runnable and, therefore, are not returned to the run pool on thread switch. If the thread was already suspended there is no effect. If the thread is blocked, it becomes suspended and blocked.

void threadKill(Thread_t *t*);

Terminates thread *t*. Given proper permission, thread termination requests cannot fail or hang. When a thread terminates by any means, all Brevix data structures for the thread are released and the thread is detached from all entities. In addition, a thread-death event is raised.

void threadYield(void);

Invokes the dispatcher to disengage the current thread from its processor and to run another thread.

4.1.3 Thread machine state blocks

The (potentially modifiable) machine state of a thread is represented by an abstract object called a *thread machine state block* or TMSB. It contains only the processor-specific state for a thread instruction stream. It does not include any other information associated with a thread. While a thread is running, this state is contained in the processor. While a thread is not running, this state is contained in Brevix memory. When a thread is released from a processor, the machine state values are saved from the processor to the thread's TMSB. When a thread is dispatched to a processor, the state values are restored from this memory. Saving and restoring machine state plus a few pointers to other thread information are the only operations that need occur on thread switch. This allows a very fast context switch.

The values of a thread's machine state block may be read or written as follows. (Tmsb is a type that can hold thread machine state information.)

void tmsbGet(Thread_t *t*, Tmsb* *tmsb*);

Returns a copy of thread *t* machine state in *tmsb*. These values are defined only if thread *t* is suspended.

void tmsbPut(Thread_t *t*, Tmsb* *tmsb*);

Replaces thread *t* machine state from *tmsb*. No privileged machine state will be changed. If thread *t* is not suspended, there is no effect. Therefore, a thread cannot change its current TMSB values with this call. Typical uses include initializing a new thread.

The TMSB values passed to an exception handler are a copy of those in effect at the point of exception. When called on a suspended exception handler, the routines tmsbGet() and tmsbPut() refer to the machine state of the handler and do not effect the "stacked" TMSB that is to be used to continue the thread

For each processor model, there will be mechanisms for reading and writing individual values within structures of type Tmsb. Since meanings for these values are machine dependent, the details will be described elsewhere.

4.1.4 Protection groups

The active protection domain of a thread represents the thread's rights to dereference memory addresses. Each valid memory location is protected by exactly one protection group. If a protection group belongs to the active protection domain of a thread, then the thread may dereference the location (load, store, or execute) in any way permitted by the protection group to which the location is bound. The protection groups that belong to the active protection domain for a thread can be changed (see section 4.2.1). Also, different protection domains will be made active for a given thread as a side effect of the thread crossing and returning back across a protected interface (section 4.2.2).

4.1.5 Synchronization

Thread synchronization is provided through three mechanisms: *mutexes*, *condition variables* and *events*.

- A *mutex* has two states, *held* and *available*. A mutex becomes available when some thread clears it. A thread may request that a mutex be *granted*, i.e., that the mutex transition from available to held. If the mutex is held, threads calling mutexSet() will be blocked. If the mutex is available and there are requesting threads, exactly one thread will be unblocked (if needed) and allowed to return. In addition, the mutex will be granted, i.e., become held.
- *Condition variables* are manipulated by three main operators: a wait operation causes the thread to block; a signal operation causes at most one of the blocked threads to become runnable; and a broadcast operation causes all waiting threads to become runnable.
- *Events* are manipulated by two main operators: a wait operation causes the thread to block until the event is raised; and a raise operation causes all waiting threads to become runnable.

  Even though events can be simulated with the other two primitives, we felt that a more efficient implementation would result if events were a first class synchronization primitive, and that events are common enough to justify this. Also events are a natural paradigm for providing some Brevix capabilities: for example, death-of-a-thread and interrupt handler interfaces.

Each mutex, condition variable and event is global in scope and may be manipulated by threads running on any node.

The Brevix synchronization routines are as follows. (Mutex_t, Condition_t and Event_t are opaque types whose values denote mutexes, condition variables and events respectively.)

Mutex_t mutexCreate(void);

Creates a new mutex and returns a reference.

void mutexSet(Mutex_t *m*);

When mutexSet() returns, the mutex will have been granted. If the mutex is currently held, the calling thread will block. Note that if a mutex is held, the system need not know by which thread.

void mutexClear(Mutex_t *m*);

Clears mutex *m*. If the mutex is available, there is no effect. If the mutex is held, the mutex becomes available. The important side effect of this case is that all threads blocked on the

mutex become eligible to unblock, have the mutex be granted, and return from their call to mutexSet(). Of course at most one such thread will do so.

Boolean mutexTest(Mutex_t *m*);

This is similar to mutexSet(). Requests that the mutex *m* be granted, but will not block. Returns false, if the mutex is held. If the mutex is available the mutex will be granted and this routine will return true.

void mutexFree(Mutex_t *m*);

Releases the mutex m. This is the opposite of mutexCreate(). If the mutex is available, it is destroyed and all associated data structures are reclaimed. If the mutex is held, the caller receives an interface failed exception.

The primitives for condition variables are as follows:

Condition_t conditionCreate(void);

Creates a new condition variable and returns a reference.

void conditionWait(Condition_t *c*, Mutex_t *m*);

Waits for the condition *c* to be signaled, i.e., until conditionSignal() or conditionBroadcast() is invoked on the condition. The current thread atomically clears mutex *m* and becomes blocked on the condition. When unblocked, but before returning the thread calls mutexSet(*m*). Upon return, the calling thread may assume that the condition has been signaled at least once sometime after the call was made. No additional guarantees are implied about the state of the system. If the condition or mutex are not defined, an interface failed exception occurs.

boolean conditionSignal(Condition_t c);

Signals condition *c*. If there are no threads blocked on the condition, there is no effect and false is returned. If there are threads blocked on the condition, then exactly one will be unblocked. The others remain blocked and the call returns true.

boolean conditionBroadcast(Condition_t c);

Signals condition *c* to all waiters. If there are no threads blocked on the condition, there is no effect and false is returned. If there are threads blocked on the condition, then all of them will be unblocked. In this case, the call will return true.

void conditionFree(Condition_t *c*);

Releases the condition *c*. This is the opposite of conditionCreate(). All threads blocked on the condition become runnable and receive an interface failed exception.

The primitives for events are as follows:

Event_t eventCreate(void);

Creates a new event and returns a reference. Events have two states, raised and not raised. When created an event is not raised.

void eventClear(Event_t *e*);

Sets event e to not raised.

Boolean eventTest(Event_t *e*);

Returns true if the event *e* is raised, false otherwise.

void eventWait(Event_t *e*);

Waits until the event *e* is raised. If the event is raised this call returns immediately. If the event is not raised the calling thread is blocked on the event.

void eventRaise(Event_t *e*);

Sets event e to raised and unblocks all waiters. Each thread that is blocked on this event is unblocked.

void eventFree(Event_t *e*);

Releases the event *e*. This is the opposite of eventCreate(). All threads blocked on the event become runnable and receive an interface failed exception.

4.2 Interfaces

Brevix provides client-server computing through the concept of customer / producer interaction. However, the Brevix implementation of client-server computing does not use the traditional RPC mechanism [Birrell83] of separate processes running in separate addres spaces, but rather uses a model in which a single thread moves between protection domains, similar to that of LRPC [Bershad89] which we have extended to include procedure calls within a protection domain and also use as a basis for our internode remote procedure call.

An *interface* in Brevix is the name we give to the contract that defines a set of services made available by a provider to its *customers*—threads executing code. In traditional usage, interface definitions include a list of procedures, their type signatures, global data, and exception information associated with the calls. Brevix interface definitions also include information about how protected arguments and results are shared between customer and provider. All interfaces define an exception called Interface-Failed, which is raised if no more specific information is available.

The code and data that implements the service behind an interface is called a provider. There can be many different provider implementations that all correspond to the same interface definition (Figure 20). In addition, one provider may export more than one service.

Providers can be instantiated in one of three ways, depending on whether they are copied into the executable image of the customer or not, and whether the provider and customer are protected from one another or not (Table 7). The remainder of this discussion concerns itself solely with providers that have been linked by reference.

Table 7: manifestations of providers.

code included by	*unprotected*	*protected*
copy	“statically-linked library”	(not supported)
reference	“dynamically-linked library”	protected provider

We have nothing to say in Brevix about how customer and provider code is assembled and stored: it may be one object module per entity, or several object modules may be collected into a library or archive file, or a language compiler may even support a complete code library scheme of its own, as with Algol68. These are compiler code-packaging problems.

Once a provider has been installed in the system, a thread can *install* one or more interfaces to it (Figure 20): we say that an *instance* of the interface is thereby created. In the case of dynamically-linked libraries, this instance is simply an indirection mechanism, but in the case of a protected provider, a piece of glue code is constructed to perform the protection domain and exception

handler changes, in collaboration with a system service called the *interface manager*. Once the interface has been installed, it may be invoked.

> In a PA-RISC system, this interface glue code is associated with a gateway page that temporarily raises the privilege level of the calling thread to allow protection state to be altered. In any case, interface crossing can be a lower-cost activity than a full context switch since only protection information needs to be altered.
>
> We leave as an implementation question whether there is a separate gateway page for each instance of an interface, or whether they can share code, at the cost of somewhat increased execution cost.

During installation, the interface may be bound to a name in the Brevix namespace (section 5 on page 61).

Brevix itself uses protected interfaces for many of the services that it offers. Non-Brevix providers (e.g., a database management system) are also expected to use protected interfaces whenever they offer concurrent service to two or more independent and unrelated customers and also keep internal, protected state.

The key factor in determining if a benefit will be derived from using a protected interface is whether a provider maintains state information that needs to be protected from direct customer access. Such information might be data that should be accessible to some of its customers some of the time (probably only while they are executing provider code), or it might be data that should never be accessible to any of its customers.

The overall job of the Brevix interface manager is to ensure that when a thread crosses a protected interface, it acquires access only to the entities that it has been authorized to access, and that such

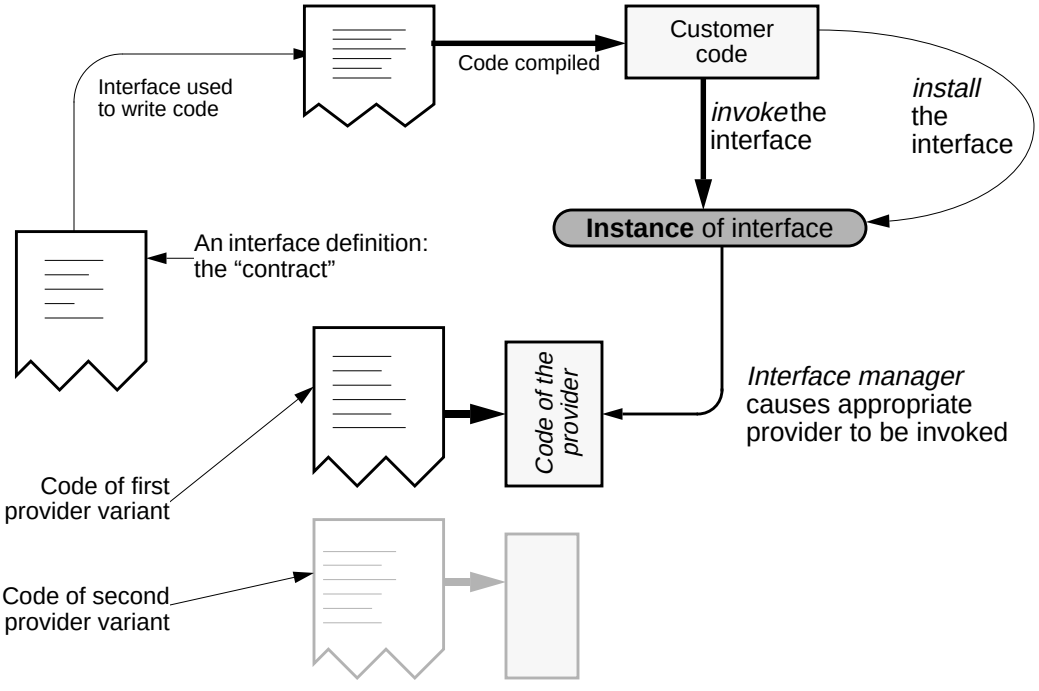

Figure 20: interfaces and providers.

access can only be exercised by executing code supplied by the provider. It also manages exception propagation across interfaces (this is discussed in section 4.3.4 on page 58).

4.2.1 Protected interface crossings

In the case of a protected provider, the interface manager uses the security service to help ensure:

- That a customer may enter a provider only by calling one of the entry points that have been exported by the provider.
- That only the threads that have successfully attached a protected interface may actually cross it.
- That the return from a protected interface crossing passes control back to customer code at the call site or through the exception mechanism (described in section 4.3.4 on page 58).

In addition, when a protected interface is crossed, the protection domain of the calling thread is changed to the protection domain associated with the interface instance the thread has crossed, extended by including the entities the thread has named as arguments to the call which caused the crossing. When a thread has finished executing provider code, it returns back across the interface and the original protection domain (the one in effect just prior to crossing the interface) again becomes active. (It is possible that the contents of the original protection domain will have been altered—in an authorized manner—by some other piece of the system. Such changes are made by explicit requests to the interface manager, and are described in section 4.2.4.)

4.2.2 Using Interfaces

Before a protected interface may be used, it must be installed and attached. The thread installing the interface informs the interface manager of the type signature of all of the procedures exported by the interface, the name of the exception handler for that interface, and a list of entities which will be made available to a thread which crosses that interface: the protection domain of the interface. It also informs the interface manager if instances of the interface may be shared, or if new instances must always be created when the interface is attached. If the installing thread has the appropriate rights, the interface is installed and becomes available for use.

When an interface is installed, it is attached to each interface on which it must rely, such as the interface to the shared version of the programming language runtime support library. Installing an interface causes the first instance of the installed interface to become available for attachment.

Before a thread can invoke an interface, it must *attach* an instance of the interface (the new interface) to the interface in which the thread is currently running (the current interface.) If the new interface is sharable, then the attaching thread may attach an existing instance of the new interface to the current interface, or it may ask for a new instance to be created. If the new interface is not sharable, a new instance is always created.

During attachment the interface manager will make certain that the thread is authorized to use the interface (by calling the Brevix security service,) and will create an *interface instance* for that attachment. The interface instance will contain a reference to the type signature of the procedures exported by the interface and the protection domainwhich will become available to the thread as a result of crossing the interface.

Interfaces that have been attached by a thread will implicitly be detached when the thread dies. Multiple attachments of a given interface by a given thread may be in effect at the same time.

When a thread is through with an interface it may detach from that interface, freeing the interface instance. All interfaces attached to a thread are detached when the thread dies.

4.2.3 Interface protection domains

An interface crossing has the same semantics as a procedure call. A protected interface crossing additionally establishes the set of entities which may be accessible to the called code while it is active, and reestablishes the (possibly modified) set of entities which are accessible to the calling code upon return from the interface crossing. The set of entities accessible to a thread at any one time is the *active protection domain* of that thread. (Protection domains are described in section 3.2.2.) When a thread crosses a protected interface as the result of a call, a new active protection domain is established for the thread. When the thread returns from the call, the previous (possibly modified) protection domain is restored.

Brevix utilizes a programming model which encourages encapsulation and data hiding. Interface crossings utilize the C call-by-value / value-return argument passing semantics, with the usual understanding that some of the values may be dereferenced as pointers. From the point of view of the programming language model of a procedure call, there are two kinds of entities which might occur in the active protection domain of a thread while it is executing within a provider. The provider requires entities to contain its code and data. It also needs access to a subset of the customer's entities, which contain the data made available to the provider as arguments to the procedure call which caused the interface crossing. The set of entities which are made active as a result of crossing the interface is described at the time the interface is installed in the system. The set of entities which are made available to the provider by the customer is described at interface crossing time.

Although Brevix encourages data hiding, it does not prohibit data sharing. The Brevix model of data sharing (other than procedure arguments) between customers and providers utilizes the security service: the customer arranges for the provider to have the necessary set of rights to independently add the shared entity to the set which will be active when the thread is executing the provider. By being in the active domain before and after an interface crossing, an entity becomes shared between the provider and the customer.

> For the purpose of the overview, we are ignoring two important issues. One is an implementation efficiency issue. The pure model is that the argument list is bundled up and handed across the interface: there are separate stacks on either side of the interface and the arguments are always copied. There is clearly a need for an efficiency enhancement to allow stack sharing. The other issue is exactly what mechanism will be used to enable provider access to dereferenced data. We do not need to describe the mechanism in the overview document, only mention that it will exist.

When a call is made, the new active protection domain is calculated from the entities associated with the instance of the interface and the entities made available as arguments to the procedure call. While the provider is active, it may modify the set of entities associated with the interface instance so that a different set will be made available the next time the thread calls across the interface. If the provider has the appropriate rights, it may also change the protection domain of its caller by adding or deleting entities from that domain. When the provider returns to the caller, either by a return or by raising an exception, the protection domain will be restored to the state it was in prior to the call, possibly modified by a sufficiently privileged provider.

4.2.4 Remote procedure call

A thread running on one node may wish to make a procedure call to a routine that best executes on another node. For this purpose, Brevix provides a remote procedure call facility. A remote procedure is a regular procedure accessed through a protected interface. When installing an interface to a remote provider, a local interface instance is constructed, and this may be called just like any other local interface instance. The interface manager transparently takes care of the node to node communication necessary to convey the call-by-value arguments to the provider and return the result.

The constructed glue code in the interface instance causes an invocation to turn into a remote procedure call: the interface manager first extends the execution state of the thread to the home node of the provider code, and then proceeds to act as if this was a regular protected domain crossing. Neither the customer nor the provider is required to know whether the call is local or remote.

The thread that instigates a remote procedure call is the same thread that executes the procedure body in the provider on the remote node. This implies that the thread ID will be communicated to the remote node by Brevix as part of the remote procedure call service. This also implies that remote procedure calls are synchronous. Asynchronous behavior can be obtained using multiple threads. Using the Brevix thread management and synchronization primitives makes this a simple matter. It is also efficient, since Brevix threads are lightweight.

4.3 Interruptions

An interruption in Brevix is defined to be any condition or occurrence that causes the CPU to interrupt its normal flow of instruction processing and to pass control to a special predetermined system routine called the interruption handler. Such occurrences generally fall into one of three distinct categories:

1. Events that are viewed as normal, but which require special processing and are unpredictable as to exactly when they will occur, such as memory page faults.
2. Hardware interrupts for device management, such as I/O completions and clock interrupts.
3. Hardware and software errors that prevent the processor from executing a given instruction, such as divide by zero errors.

The following section describes a generic paradigm for processing all interruptions in Brevix. (Hardware interrupts for device management are also discussed further in section 3.4.)

4.3.1 Interruption processing paradigm

When an interruption occurs, the processor will implicitly preserve enough of the current execution state so that if only a very short response by the interruption handler is required, the interrupted processing can be restarted as though nothing unusual had occurred (Figure 21). Before passing control to the interruption handler the hardware also enters a special predefined state that typically includes the following: further interruptions are disabled or at least masked, the highest privilege level takes effect, and virtual memory translation is turned off. Thus, when

the interruption handler begins executing, it has more or less unrestricted access to the bare machine.

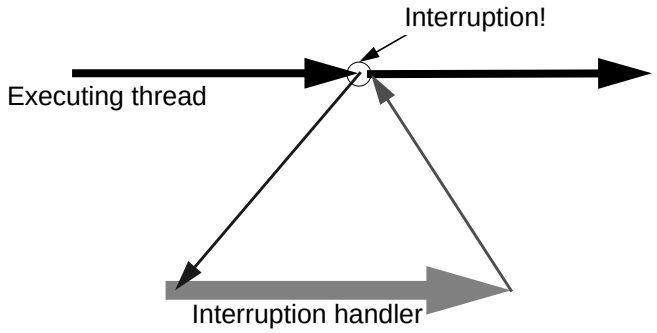

Figure 21: first-level interruption handling.

> Such access is often necessary to properly respond to an interruption that has occurred. On the other hand only a small portion of the actions to be taken actually require unrestricted access to the hardware. Moreover, only a minimal amount of processing should be allowed to occur before interruptions are again enabled. This is necessary to avoid delays in handling higher priority interruptions, to minimize latency in handling external interrupts, and more generally, to maintain equitable multitasking semantics.

Because of these considerations the generic interruption handling code should consist of a preliminary portion that is as short and fast as possible, and which is followed by a transition to a normal processing state. The portion of the interruption handler following this transition is called the system interruption code (Figure 22). During the transition most if not all interruptions should be re-enabled, virtual memory translation should be turned back on, and a complete copy of the execution state at the time of the interruption will have to be explicitly saved. More generally, the machine state after the transition should be capable of supporting code that has been written in a high level language and that needs to execute in a standard runtime environment. Most of the interruption processing should occur in this state.

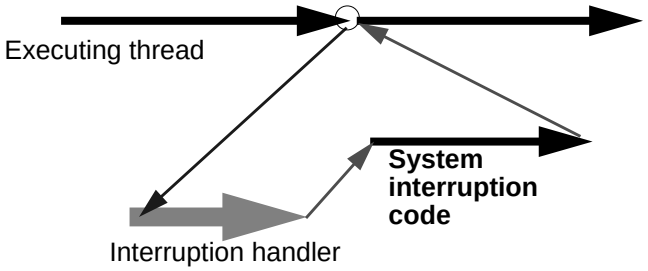

Figure 22: interruption handling with system interruption code.

The first job of the system interruption code is to determine as many details as necessary regarding the cause of the interruption so that it can execute an appropriate response. Some of the required details will be provided by the hardware when the interruption handler is called and should be made available to the system interruption code during the transition from the preliminary interruption code. In any case Brevix will provide a mechanism to install software for

responding to specific interruptions and a dispatcher interface for calling the installed code once the cause of an interruption has been determined.

It is assumed that a few interruptions will be completely serviced in the preliminary portion of the interruption handler. Such cases should always be viewed and justified only as optimizations to the generic interruption processing paradigm described in the preceding paragraph. The primary example of such an optimization is a TLB miss, which requires nothing more than reloading the TLB with the proper address translation. When this particular interruption occurs, its precise cause should be known immediately, and only a small number of instructions are required for repair before an attempt is made to retry the interrupted instruction. On the other hand a transition to system interruption code should be made whenever a TLB miss also leads to a page fault.

4.3.2 Synchronous processing

Any interruption directly related to and caused by the interrupted instruction is called a *synchronous interruption*. There are three possibilities for further action when the system interruption code completes its response to a synchronous interruption:

1. Resume the interrupted processing using the saved state as it existed at the time of interruption.
2. Resume the interrupted processing with some altered state.
3. Initiate thread death processing.

Generally speaking the first choice represents the final resolution of interruptions that are transparent to applications, and that are not in any way anomalous from the application's point of view, e.g. TLB miss handling, page faults, and explicit I / O requests. The second choice typically represents the final step of the system interruption code after resolving an error caused by some action being attempted by the interrupted instruction (Figure 23). Such errors are called *exceptions* in Brevix. The altered state may be constructed directly by the system interruption code or by an installed exception handler (see section 4.3.4 for more on exceptions.) The third choice occurs because an exception has been raised that cannot otherwise be resolved. (See section 4.1 for more on thread death.)

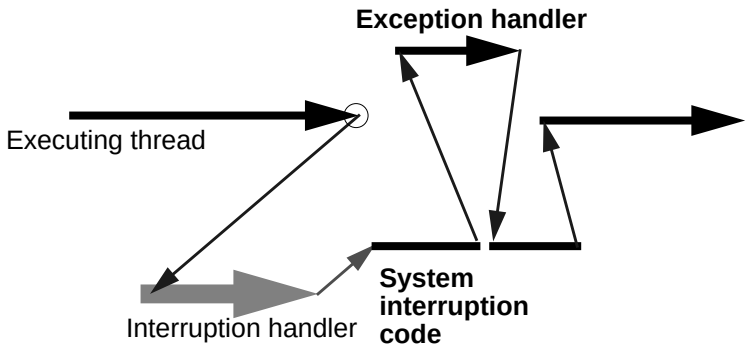

Figure 23: interruption handling with an exception handler.

The processing state of the system interruption code while it is executing a response to a synchronous interruption is not different from that of a typical thread in the system. All the

normal Brevix programmer abstractions and services are available to it and further interruptions may occur. In particular the system interruption code may be preempted by runnable threads of higher priority, or it may voluntarily yield the processor to wait for an I/O completion or for some other arbitrary system event to occur. When the system interruption code begins executing (following the transition from preliminary interruption code to a normal processing state), it assumes the identity and uses the TMSB of the interrupted thread. Thus, before executing the system interruption code, Brevix must acquire a fresh TMSB and use it to make a complete copy of the processor state as it existed at the time of interruption. Moreover, since interruptions may occur recursively, Brevix must store this TMSB in a stack-like data structure so that the original state of the machine can be recovered when the nested interruptions are eventually unwound.

4.3.3 Asynchronous processing

An interruption that is not directly caused by the executing instruction is called an *asynchronous interruption*. The most common examples are interrupts from external devices and the interval timer. Such interruptions are initially handled in an identical manner to synchronous interruptions, except that time counters (see section 6.1) tracking processor use by the thread should be stopped. As before, the preliminary interruption code should transition to a normal processing state as soon as possible, and the following system interruption code should assume the identity of the interrupted thread and use its TMSB while responding to the interruption. However, further processing done by the system interruption code should be minimal since the cause of the interruption is not likely to be of concern (or value) to the interrupted thread.

The goal of asynchronous interruption handling code should be simply to acknowledge the interrupt, and to preserve any available information regarding its cause on a work queue for consumption by a more appropriate thread through normal scheduling and dispatching activity at some later point in time. In practice the system interruption code will probably accomplish these tasks by calling an interrupt service routine provided by a device manager (for the case of an external interrupt), or by calling a routine provided by the time service (for the case of a time interval expiration). As soon as this has been done, time counters tracking processor use by the thread should be restarted and processing should resume with the original state saved at the time of the interruption.

4.3.4 Exceptions

An exception is said to be raised by Brevix whenever an error condition causes an interruption and cannot be repaired by standard system interruption handling code, or a program executing normally chooses to raise one explicitly. The first of these will be directly caused by an executing instruction and thus will trigger a synchronous interruption. When this occurs it may be possible to recognize the error condition immediately as in the case of a divide by zero error. In other cases the error condition may not be recognized until after further processing by the system interruption code. For example, a reference to an unmapped virtual memory address will probably be interpreted first as a TLB miss, then as a page fault, and then finally as an error.

Since the appropriate response to an exception may vary from one thread to the next, each interface instance has an associated entry point called an *exception handler* that will be called whenever exceptions are raised while the thread is running.

> Although the run time environment of the source code programming language may provide the exception handler for a thread, and may or may not provide the facilities for application code to change it, Brevix will allow any thread to dynamically change its exception handler

binding, and will automatically change exception handler bindings on interface crossings and returns (section 4.2).

Handling exceptions. Figure 24 shows the general flow of control across an interface. In the normal case, code executing on one side of the interface makes a *call* across the interface to cause code on the other side to execute. If no exceptional condition occurs, the called code executes a *return* to return to calling code. An exception may occur as the result of a hardware interruption, or as the result of a routine explicitly *raising* the exception. We shall refer to either of these as raising an exception. When an exception is raised, the *exception manager* causes an exception handler to run. Exception handlers do not execute in a special state. Exception handlers may be entered from the current provider as a result of a raised exception, or they may be entered through the punt mechanism described below. The identity of the thread executing a handler will be the same as the one for which the exception was raised, and interruptions will be enabled. This means, in particular, that a nested set of exceptions may occur.

When an exception is raised, the Brevix exception manager will make a copy of the machine state at the point where the exception was raised. It will then copy this state into a TMSB, and call the current exception handler with a pointer to the TMSB copy, and perhaps further arguments regarding the source and nature of the exception. The current exception handler is the one that was installed with the interface most recently crossed prior to the point where the exception is raised. It will either have been supplied by the provider associated with the interface or it will be a generic handler supplied by Brevix. When the exception handler is finished, it returns control to the exception manager with an indication of whether or not the exception was successfully resolved, together with the (possibly altered) TMSB.

If the exception handler indicates that it was unable to successfully resolve the exception, shown in Figure 24 as *punt*, the exception manager will discard the returned TMSB, and propagate the exception through the most recent interface crossing. To do this, it uses a new TMSB that gives the

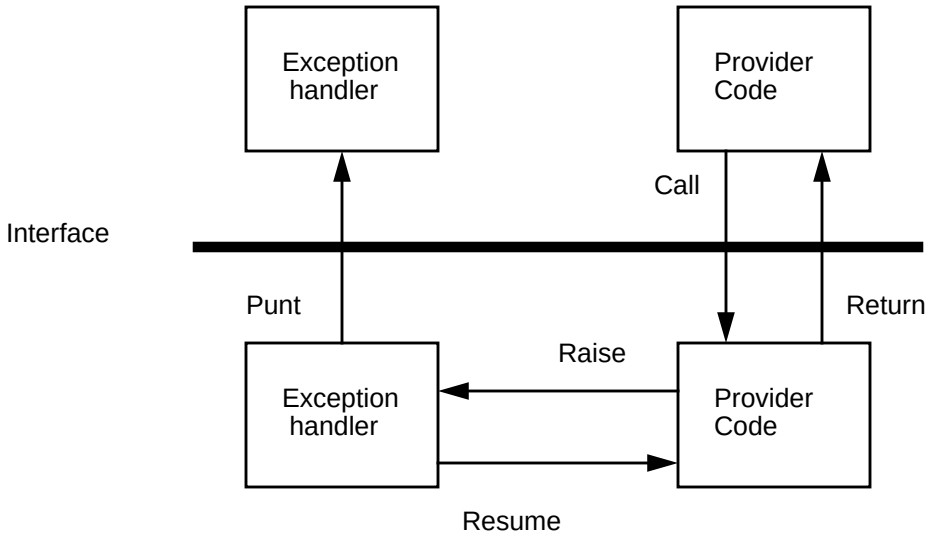

Figure 24: exception handling

appearance that the exception occurred at the instruction that called across the interface. If there is no interface crossing outstanding, the thread will be terminated.

Brevix will supply a mechanism that allows an exception handler that punts to specify which exception should be passed back across the interface. Although this mechanism must cooperate with the security service to respect security policies that choose to restrict the flow of such information, the generic exception interface-failed may always be raised across an interface. Interface-failed is an exception that indicates that a service could not be provided without indicating why.

When a successful resolution is indicated by a handler, shown in Figure 24 as *resume*, the exception manager will merge the portions of the returned TMSB that the thread has updated with the TMSB stored on the top of its internal stack, and resume the thread with whatever machine state exists in the result.

5 Naming

In this section, we describe the Brevix naming system. We begin by describing the requirements for the naming system. We then discuss related work, outline our design, and present the design in greater detail.

We begin by presenting the requirements for the naming service of a multicomputer, and follow this by an overview of the design and some more details on selected points.

> When computer scientists gather to discuss their work, it is common for discussions to end, or at least change direction radically, with the statement “That’s a naming problem.” This is not simply a ploy to avoid difficult topics; naming is fundamentally important in the design of computer systems. It would, in fact, be fair to characterize operating systems, and distributed operating systems in particular, as “naming problems.” Studies of UNIX systems have shown that, for the particular systems measured, 40 percent of system call overhead is due to file-name resolution [Leffler84], name mapping operations account for over 50 percent of file system calls [Mogul86a], and 20 percent of all commands issued require user-program access to directories [Sheltzer86].

5.1 Requirements

The requirements for the Brevix naming system can be divided into the following categories: scalability, fault tolerance, location-independence, flexibility, and efficiency. The subsections below describe each category of requirement in more detail.

5.1.1 Scalability and fault tolerance

The most important requirement of the Brevix naming system is that it allow Brevix to meet its goal of being a scalable, fault-tolerant operating system. To make this more concrete, we require:

- *Scalability 1*: as more nodes are added to a system, the processing required to translate the name of a given object remains nearly constant.
- *Scalability 2*: as more named objects are added to a system, the processing required to translate the name of a given object remains nearly constant.
- *Fault tolerance 1*: the name system can be configured to allow naming operations to continue despite node failures, link failures, and failures of managers that manage portions of the name space.
- *Fault tolerance 2*: the name system can be configured to provide different fault-tolerance guarantees for different named objects.

5.1.2 Location independence

Brevix is intended to run on a multicomputer that uses a collection of processors to implement a powerful, centralized computational resource. Services provided by providers that happen to be executing on a certain node of the system should be accessible to any client, regardless of the node on which the client is executing. A large part of the burden of providing this functionality, which we will call *location independence*, falls on the naming system. We specify this requirement as:

- Location independence: the Brevix naming system provides a *global name space* for text names and for handles: that is, both fully-qualified text names and handles refer to the same objects regardless of the node of the system on which they are used.

Notice that this requires the functionality, but not the performance, of naming operations to be location-independent.

5.1.3 Flexibility

It is also a goal of Brevix to allow the construction of providers that support APIs of other operating systems. This requires:

- *Flexibility 1*: the name system should be flexible enough to support the naming models of other important operating systems efficiently.

The design of Brevix entity management places another requirement on the naming system. Each entity is managed by an entity manager. The entity manager for an entity is specified when the entity is created, and is unrelated to any text name that might be associated with the entity. The requirement for the naming system is therefore:

- *Flexibility 2*: the manager of a named object and the text name of the object are unrelated.

To show that this is not a trivial requirement, let's consider the UNIX naming system. In UNIX, the "manager" of a file is the file system in which it resides, and file systems correspond directly to portions of the name space, as illustrated in Figure 25. In the figure, the directory movie1 contains the script, video, audio, and index for a movie. Since the files are stored in the same directory, they are in the same file system, and are managed with the same policies. The files, however, contain data of very different types, with different access requirements.

There are several ways to configure a UNIX system to allow the files to be managed by different file systems. First, the name space can reflect the file management structure, as shown in Figure 26. This allows specialized file systems (managers) to be used to manage the different file types.

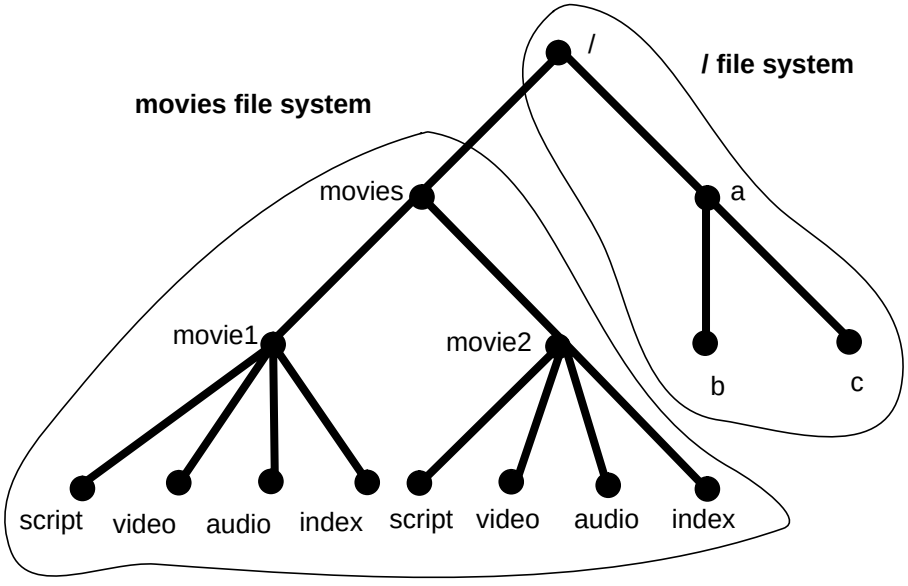

Figure 25: UNIX name space with two file systems.

The problem with this, however, is that related files, in this example all the files for a certain movie, must be stored in separate portions of the name space. A second alternative is to use the structure shown in Figure 26, but also to maintain a name structure as shown in Figure 25 by using symbolic links. This solution allows files to be named independently of their file system (manager), but is difficult to administer and keep consistent. We want to design the Brevix naming system to allow the text name of the object to be independent of the manager that manages the object, and to avoid the problems that result from using ad hoc solutions.

Naming is a service that is needed by many parts of an operating system, not only the data management subsystem. For example, some type of naming mechanism is required to implement RPC binding and named virtual address reservations. To avoid the need for each provider to implement a naming mechanism, we place another requirement on the naming system:

- *Flexibility 3*: the naming system should be flexible enough to be used by any provider that requires location-independent, fault-tolerant, scalable name service.

Certainly some naming requirements will be too low-level or too provider-specific to be handled by the naming system. But for naming problems that require the attributes listed above, the name service should be the solution mechanism of choice.

5.1.4 Efficiency

Although the requirement of efficiency is typically assumed, we will state it explicitly:

- *Efficiency*: the naming system should be as efficient as possible, in terms of processing and communication required, while still meeting the other requirements at a specified level.

We chose to make this requirement explicit mainly because we recognize that a requirement of absolute efficiency conflicts with the other requirements. A system that is maximally fault-tolerant, or maximally flexible, is probably not extremely efficient. The naming system should be structured so that objects that place few flexibility or fault-tolerance requirements on the system

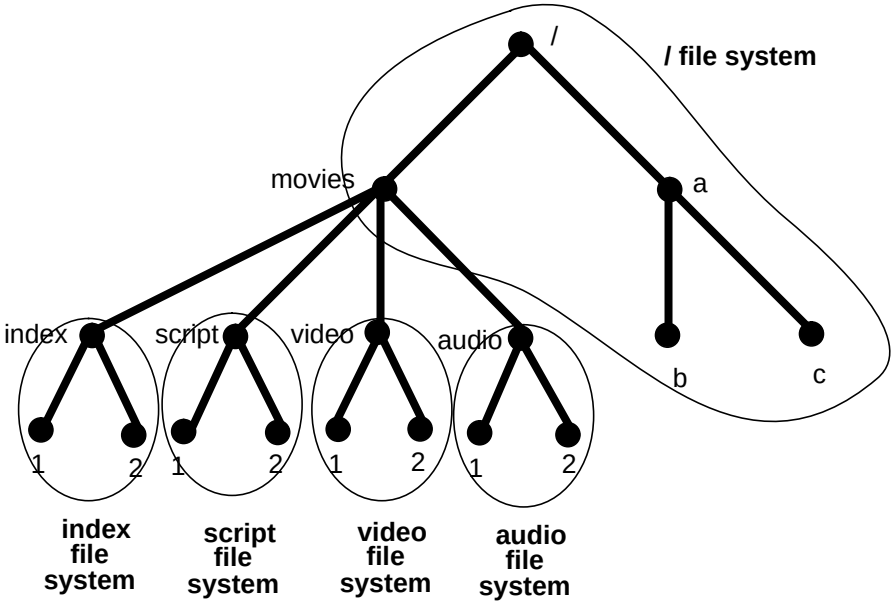

Figure 26: UNIX name space with a file system for each file type.

are very efficient, while those that require more service are less efficient, but as efficient as possible given the requirements.

5.2 Related work

A great deal of work has been done on naming systems for distributed systems. Much of the work, however, including Grapevine [Schroeder84] and the Domain Name service [Mockapetris87], concentrates on naming in systems that are more loosely connected, and for which issues of administrative autonomy and wide-area scalability are paramount. The systems that come closest to satisfying the goals we have set for our name service are the name services of V [Cheriton89] and Galaxy [Jia90, Sinha91, Sinha91a].

The managerial level of the V naming system supplies the type of scalability and efficiency characteristics we require in Brevix, but its design supports only the very narrow UNIX file system model of object management shown in Figure 25. The Galaxy name service is also scalable, and is much more flexible in terms of object management: too flexible, in fact, for the requirements of Brevix, at the expense of lost efficiency. Our goal is to design a system that takes the middle ground between these two systems, retaining as much efficiency as possible while allowing the name-space flexibility that other Brevix services require.

5.3 Design outline

This section outlines the design of the Brevix naming system. The details of the design are presented in section 5.4.

5.3.1 Naming model

The Brevix naming system uses two types of names: *external names* and *handles*.

External names. External names are human-readable text names. Several organizations for the external name space are possible:

- A flat name space, in which all names are in one logical space.
- A hierarchical name space with a fixed number of levels. The MPE name space is organized in this way.
- A hierarchical name space with an arbitrary number of levels. The UNIX name space is organized in this way.

Brevix provides a hierarchical external name space with an arbitrary number of levels. This is the best choice for two reasons. First, we require the Brevix naming system to be flexible enough to allow other APIs to be built on top of it, and an arbitrarily deep hierarchical name space is the most flexible of the three alternatives. Second, this organization offers the possibility of efficient name resolution due to locality of reference within the name space.

Brevix external names are very similar to UNIX external names: external names comprise a "/", a (possibly empty) list of / terminated *directory names*, and an *object basename*. For example, /users/osr/design/info is an external name; users, osr and design are directory names, and info is the object basename. We will also refer to a proper external name as a *fully-qualified* external name.

> The choice of / as the separator character in Brevix external names does not mean that Brevix cannot support APIs with hierarchical name spaces and different separators; a simple API mapping layer can translate external names in the syntax of the API into Brevix external names.

Context-relative external names are external names that do not begin with a /. A context-relative external name can be converted to a fully-qualified external name by executing a *context resolution function*. A *context* is a directory, together with additional state (associated with the invoker) that is used to disambiguate between multiple entries with the same name in the directory; a *context name* is a slash-rooted, slash-terminated (possibly empty) list of directory names. An example of a context-relative external name is design/info. Using the context name /users/osr/ and a context resolution function that concatenates the context name and the context-relative name, the context-relative name is equivalent to the fully-qualified external name /users/osr/design/info. A context name is said to be a *prefix* of an external name if the external name begins with that context name.

This external naming scheme is quite flexible. An object can have many external names. Different objects can have the same external name, presumably, though not necessarily, with different attributes (see Section 5.4.5) associated with each. Although the naming system does not impose any requirements on how the external name space is organized, widely followed naming conventions are very useful, and we expect that there will be a document that describes the conventions for using the Brevix name space.

Handles. Handles are bit-string names that, when used within a given Brevix system, uniquely identify an object. Brevix handles consist of a *manager identifier* or *MID*, and an *object identifier* or *OID*. We require that named Brevix objects, including directories, belong for their lifetime to a single manager, which is uniquely identified by its MID. An MID is a unique identifier, or UID; UIDs are described in detail in Section 5.4.1. An OID uniquely identifies the named object to the object's manager, and is chosen by the manager (although its size is the same for all managers). The manager may choose to use a UID as the OID, or, for convenience or efficiency, may chose to assign OIDs in some other fashion.

A two-level handle name space was chosen because it is much easier to implement efficiently and scalably than a single-level name space, and because we believe that most uses of the naming system are, in fact, structured in this way. For example: entity managers (the managers) manage files (the objects), interface managers manage interfaces, and so on. It would be possible to use a single-level name space for objects (Galaxy does so), but this would require a complex, and somewhat inefficient, object-location mechanism. In our opinion, the efficiency gained by choosing a two-level namespace is more important than the flexibility lost.

As far as the naming system is concerned, a handle refers to one specific object. An object manager may allow different handles to refer to the same object, or a handle to refer to more than one object, but this is invisible to the naming system. An object may have two handles that indicate that it is managed by two managers only if the managers cooperate to provide this appearance; this is also invisible to the naming system

It should be noted that not all Brevix managers need to use handles to refer to the objects they manage. Managers that do not manage objects that are referred to by some name in the namespace do not need to use handles, OIDs, or UIDs. Because UID generation and manipulation routines will be available, managers may wish to use them if they have a need for unique identifiers.

5.3.2 Name resolution mechanism

The primary function of the naming system is *name resolution*: converting external names to handles. The name resolution process is illustrated in Figure 27, and Figure 28 depicts the mechanism by which this is done in Brevix.

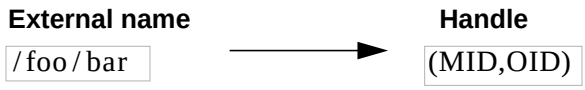

Figure 27: name resolution.

When a request to resolve an external name is presented to the naming system, a name cache is first consulted to determine if the handle for the fully-qualified name, or a handle for a prefix of the name, is already known. Brevix guarantees that at least the handle of the context-name " / " will be found in the cache.

> There can be more than one name cache, corresponding to multiple independently-rooted name spaces. This flexibility can be used at the discretion of the administrator who configures a Brevix system.

If the handle for the object is found in the cache, it is returned. Otherwise, the MID is extracted from the handle for the longest prefix found, and the directory manager table is consulted to find an interface reference for the manager of the context (which is a directory). The name resolution function of the manager is then invoked by using the interface reference; the OID of the directory and the context-relative name that is still unresolved are passed as parameters. The manager is then responsible for resolving as much of the context-relative name as it can. This name resolution process will have one of three results. First, the manager may be able to resolve the name completely; if so, it returns the handle of the object to the client. Second, the manager may determine that the name is unbound; if it is, a name unbound exception is raised to indicate this to the client. Third, the manager may be able to partially resolve the name to a directory object managed by another manager, and a context-relative name; in that case, it invokes the manager by looking up its interface reference in the directory manager table and invoking the name resolution function of the interface. This process is repeated until the handle for the object is found or the name is determined to be unbound.

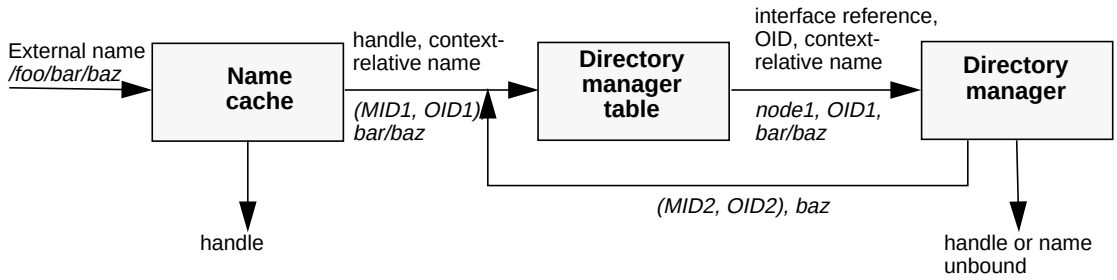

Figure 28: name resolution mechanism.

5.3.3 Object access mechanism

The result of the name resolution mechanism is the handle of the object that corresponds to the name that was resolved. The object access mechanism allows operations to be performed on this object. The object access mechanism is implemented by the interface manager (Section 4.2), but it is described here to show how handles are used to access the objects to which they refer.

The object access mechanism is shown in Figure 29. The AttachToInterface function of the interface manager is called to gain access to the manager specified by the MID of the handle, if it is not already attached. Functions in the interface of the object manager may then be invoked, and the OID of the object to be operated on can be passed as a parameter.

> For convenience or efficiency, an object manager may choose to return an identifier other than the OID that the client can use to invoke future operations on the object. The use of such identifiers is independent of the naming system.

5.3.4 Summary

This is only a general overview of the design. Many optimizations and refinements will be described in Section 5.4. But even at this level of detail we can see how the design satisfies some of the requirements we placed on the naming system.

- **Scalability.** First, the name cache allows many external names, or prefixes, to be resolved without network communication. Second, since we expect that the set of managers in a system will be relatively small and static, the directory manager table (section 5.4.4) can be replicated on each node and maintained without impacting scalability. And third, management of the name space can be partitioned between a large number of cooperating managers, each of which is responsible for managing a portion of the name space.
- **Fault tolerance**. Managers can be replicated to allow name resolution to proceed in spite of hardware failures.
- **Location independence**. The name space is global: any external name may be resolved on any node in the system. This is supported mainly by the directory manager table, which is (at least logically, if not actually) replicated on all nodes of the system.
- **Flexibility.** The hierarchical external name space is flexible enough to allow multiple naming models to be built on top of it. In addition, since the structure of the external name of the object does not dictate the manager of the object; the problems associated with the strict "file system" model of UNIX (see Section 5.1.3) are avoided.
- **Efficiency.** If a high degree of fault tolerance is required, managers for a portion of the name space will be replicated, and naming operations that update that portion of the name space will be more expensive; portions of the name space that require less fault tolerance will be more efficient. Similarly, if the full flexibility of the name space is used, an RPC might be required to resolve each directory in the external name. If, on the other hand, a manager wishes to manage a portion of the name space in a manner similar to a UNIX file system, it can handle all name resolution for its portion of the name space without any external

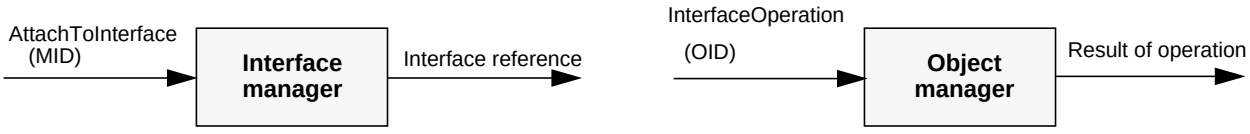

Figure 29: object access mechanism.

communication. Finally, the name cache allows network communication to be avoided altogether in many cases.

5.4 Design details

In this section we present the design of the naming system in more detail. The section is divided into subsections that describe unique identifiers, handles, the name cache, the directory manager table, and the managers involved in the naming system.

5.4.1 Unique identifiers

A unique *identifier*, or *UID*, is a 96-bit number that is guaranteed to be unique throughout all space, time, and machines. Brevix UIDs are modelled after singular identifiers in the RX system [Peck88]. Services are provided to create a UID and to check if two UIDs are equal. Users of UIDs must not depend on the internal format of UIDs in any way.

The internal format of UIDs is: 32-bit processor serial number, 32-bit seconds since 1/1/1970, 16-bit sequence number, 16-bits reserved for future use (for example clock rollover in 2106, or an increase in the size of the processor serial number). We call this *UID-time*, and it is guaranteed to never go backward. UID-time may advance at a rate greater than or less than the time-of-day clock; an attempt is made to keep UID-time close to the time-of-day clock while maintaining the constraint that UID-time never go backward. UID-time may also advance at a rate greater than the time-of-day clock if more than 2^{16}-1 UIDs must be generated in one second. (The time used in UIDs is not the same as that provided by the time-of-day facility described in section 6.1 on page 74.) Timestamps are used in UIDs as a debugging aid: given a UID, we know the approximate time at which the UID was created. If the implementation complexity of this format outweighs the benefit it offers, we might choose to use a 48-bit sequence number in place of the timestamp and sequence number.

5.4.2 Handles

A handle is an ordered pair of 96-bit numbers. The first number, called the MID, is a UID that identifies a manager, and the second number, the OID, identifies an object managed by that manager. The manager of an object assigns the OID to the object. The manager may chose to use a UID as the OID, or it may assign OIDs in another manner for convenience or efficiency, as long as it never reuses an OID. A manager that is very concerned with efficiency and that manages a static set of objects might choose, for example, to use array indices as OIDs for the objects it manages.

As far as the naming system is concerned, a handle uniquely identifies an object; if the manager internally replicates the object, it is invisible to the naming system.

5.4.3 Name cache

The name cache is designed to take advantage of locality of reference within the name space. The cache maps fully-qualified external names or prefixes to handles, as depicted in Table 9.

Table 9: logical function of the name cache.

cache entry	*handle*
/	MID1, OID1
/a/b	MID2, OID1
/a/b/c/d/e	MID1, OID2
/d	MID3, OID1
/e/f/g/h	MID2, OID2

This table is only a logical depiction of the function of the cache. The cache may be implemented in a variety of ways. For example, one cache might be maintained for each thread [Cheriton89], one for each node, or a combination of the two. The cache might be organized in one level, or hierarchically to match the name space [Sinha91a]. We currently believe that a per-node, hierarchical cache is the best choice for Brevix.

Security. Security requirements must be respected by the implementation of the cache: if a thread would not be able to resolve a name by the normal name resolution mechanism (for which the security policy is enforced by the managers), it must not be able to resolve the name using the cache. This could be enforced by including a list of required rights in each entry in the cache, or by maintaining a separate cache for each thread.

Maintenance policy. Most aspects of the cache maintenance policy, such as cache size and replacement strategy, are implementation choices. Any policy chosen, however, must keep the caches in a consistent state. As in Galaxy [Sinha91a], there are two cases in which cache entries can become stale:

a. When an object has been deleted from the system but its name mapping is still present in the name cache.

b. When a directory has been deleted or renamed, or an object has been renamed, but an entry containing the old name is still present in the name cache.

In case (a), on-use consistency control may be used. As an example, assume that the cache is in the state shown in Table 9 when the object /a/b is deleted. Some time later, a thread attempts to resolve the name /a/b, and the stale cache entry is used. The manager identified by MID2 receives a request to perform an operation on OID1, detects that OID1 is unbound, and returns an error. When this error notification arrives on the client node, the stale cache entry is deleted.

The deletion of the stale entry could be implemented as follows: the handle unbound error is delivered to the manager of the name cache (a namespace manager; see Section 5.4.5), which then searches the cache to find any entries for that handle and deletes them. This could be an exhaustive search, or data structures to make this search more efficient could be maintained.

Whenever possible, retries are performed within the name service.

There are two cases: the fact that the object was deleted can be detected during the name resolution process when the leaf directory is consulted, or, if the object is deleted after the

name has been resolved, it can be detected during the object access process. If it is detected during the name resolution process, we are still within the naming system, and the naming system can perform retries. If the fact that the object is deleted is detected during the object-access process, we are outside of the naming system, and the naming system cannot perform retries. The object manager should raise an Object Does Not Exist exception. The client may then retry the name resolution if it cares to; this time the fact that the cached data is stale will be detected, and retries done. If the named object really does not exist, it will be determined at this point.

For case (b), the approach that is semantically simplest is to immediately broadcast all such updates to the name caches, not allowing the update to complete until the caches are all in a consistent state. Clearly, however, this approach is not scalable: the rate of forward progress is slowed as more nodes are added to the system, and an unreasonable amount of network bandwidth is used if such updates are frequent. We choose to use gradual invalidation to inform all of the caches of the update: some time after the update is performed, broadcast or recursive multicast is used to inform all the caches of the update. Several updates may be bundled into each notification, decreasing the load imposed by the algorithm on the network. Semantically, this algorithm introduces a delay between when an update occurs, and when its effect is visible on all the nodes of the system.

If reliable broadcast is supported by the network, it is easy to bound the length of the delay introduced with a small variance between nodes; if it is not, the variance of the delay seen by different nodes will be larger, but the bound will still be reasonable. The semantics of this algorithm are discussed in more detail in [Sinha91a], but, in summary, the delay should present no serious problems to users, and is required if we want to ensure that the naming system is scalable.

Only the name service keeps name caches, so directory managers can simply inform the correct local instance of the namespace manager (i.e. the one corresponding to the namespace being accessed) to have the invalidations forwarded to other instances.

For cases in which applications must avoid any inconsistency between the caches, functions that ignore the information in the caches and that force the caches to be synchronously updated are also provided.

5.4.4 Directory manager table

The *directory manager table* maps from MIDs of directory managers to *interface references*. Interface references are returned by the interface manager when an interface is attached, and allow the invocation of the functions of the interface.

The organization of the directory manager table is shown in Table 10. Notice that an MID may appear only once in the table. For managers that provide fault tolerance by managing their portion of the namespace using several cooperating *manager instances*, the interface manager keeps track of which manager instances implement the manager, and the node on which each manager instance is running. All manager instances of a given manager have the same MID, but different interface references. From the point of view of the naming system, communicating with a manager via any of the interfaces registered by it with the interface manager is semantically equivalent; any necessary coordination must be implemented by the separate manager instances.

The directory manager table on every node contains an interface reference for each manager that has been instantiated. The directory manager tables on different nodes, however, can contain

different interface references for the same manager; this allows the naming system to communicate directly with local manager instances whenever they exist.

If a specific interface reference becomes invalid (perhaps because the node on which the manager instance was running goes down), the interface manager informs the namespace manager (via an Interface Unavailable exception) that the interface it is attempting to use is no longer valid; the namespace manager may then re-execute AttachToInterface to obtain an interface reference to an instance of the interface (a manager instance) that is still accessible. If no instance of the interface is accessible, the AttachToInterface call will fail by raising an Interface Unavailable exception, and the naming system will raise a Manager Unavailable exception to indicate to the client that some directory manager required to resolve the name is unavailable.

An attempt to look up an interface reference for a manager that has not been instantiated on any node causes the Manager Unavailable exception to be raised. However, it is also possible to place an entry in the directory manager table that instantiates a manager the first time a client attempts to use it.

Table 10: format of the directory manager table.

Manager identifier	*Interface reference*
MID1	IFC_REF_1
MID2	IFC_REF_2
MID3	IFC_REF_3
MID4	IFC_REF_4
MID5	IFC_REF_5

We believe that the number of directory managers in a Brevix system will be relatively small, and that the set of managers will change slowly. If this is the case, the directory manager table on each node can contain an entry for every directory manager in the system without hurting the scalability of the system and without placing unreasonable memory requirements on each node. If, in the future, we change our use of system services so that these assumptions are no longer true, a more complex protocol would need to be used to manage the table, such as caching.

5.4.5 Managers

The services involved in the naming system fall into three classes: namespace managers, directory managers and object managers.

Namespace managers. Namespace managers are responsible for managing a distributed name cache and directory manager table.There is a single namespace manager instance (provider) per node for each independently-rooted namespace. (We expect most Brevix installations to use only a single namespace.) Like all Brevix managers, namespace managers are logical entities that comprise some or all of: code accessed via an interface and executed by client threads; dedicated threads with private code; and library-resident code executed by client threads.

Directory managers. A *directory manager* is responsible for the management of one or more directories. Directories can be pictured as shown in Table 11. The table contains one entry for each object in the directory. Each entry contains the *name component* (either a directory name or an object

basename), the handle corresponding to that name component, and a list of attribute-value pairs. Included in the attribute-value information is security information, and an indication of whether the object referred to is another directory or another type of object. The attribute-value information is used to disambiguate multiple entries in the directory that have the same name component.

> We could use this to provide many things, such as some of the features of the Plan 9 [Pike90] or the Ansa Trader [Sventek88, Sventek89]. We need to do more design work on this facility.

Again, this table represents only a logical view of directories. For example, if some attribute-value pairs are present in all entries, it would not be necessary to store the attribute name (or an identifier of it) in the directory.

Table 11: logical directory format.

Name component	*Handle*	*Attribute 1*	*Value 1*	*Attribute 2*	*Value 2*	*...*
a	MID1, OID99	Owner	Barney	Type	directory	
fred	MID6, OID12	Owner	Fred	Modified	1/1/92	
foobar	MID1, OID23	Owner	Wilma	Type	object	

Directory managers export an interface that contains the functions defined in Table 12.

> Directory managers also must be able to raise an event when a directory changes. We have not yet decided if this facility will be implemented by a more general mechanism on Brevix interfaces or by the directory managers directly.

Object managers. Object managers manage a collection of objects, and support the execution of a set of operations on those objects. Section 3.1.3 presents a detailed description of one type of object manager, an entity manager. Another type of object manager is the interface manager, which manages Interfaces to providers.

In order to be able to participate in the naming system, object managers must assign an OID to each object they manage, and must be able to perform operations on an object specified by its OID.

Combined managers. A combined manager is a manager that supports the functions of both a directory manager and an object manager. This configuration is very efficient in terms of message traffic when the additional flexibility offered by separate managers is not needed. In fact, a combined manager that manages all of the files in its portion of the name space is nearly equivalent to the manager of an administrative directory (UNIX file system) in the V naming system. Furthermore, even in this configuration our naming system allows names in the namespace to be managed by other object managers with a small additional overhead.

Table 12: directory manager operations.

Operation	*In parameters*	*Out parameters*	*Description*
insert	*directory_OID name_component object_handle attributes*	*OID*	Insert an entry in the directory identified by *directoy_OID* under the name *name_component* that refers to the object specified by *object_handle* and has the specified *attributes* associated with it.
remove	*directory_OID name_component attribute_spec*		Remove the entries from the directory identified by *directory_OID* that has name *name_component* and matches the given *attribute_spec*. If no attribute specification is given, delete all entries with name *name_component*.
rename	*directory_OID old_name_comp. new_name_comp. attribute_spec*		In the directory identified by *directory_OID*, change the names of the entries for *old_name_comp* that satisfy *attribute_spec* to be *new_name_comp*.
resolve	*directory_OID context_dep_name attribute_spec*	*object_hndl*	Starting from the directory specified by *directory_OID*, resolve the given context-dependant name using the given attribute specification. This may require forwarding the resolve call to another directory manager. If the name is unbound, the Name Unbound exception is raised. If more that one resolution of the name is possible, a Not Unique exception is raised.
update	*directory_oid name_component attribute_spec attributes*		In the directory identified by *directory_OID*, update the attribute lists for the entries for *name_component* that satisfy *attribute_spec* to include *attributes*.
list	directory_oid (name_component)	list	Returns a list of the of the contents of the specified directories (optionally only for the specified name component) in a format that is logically equivalent to that shown in Table 11. The details of this format are not yet defined.

6 Miscellaneous functions

Previous sections of this document have described the major components of the Brevix system design. In this chapter, we present a few remaining odds and ends that did not fit in conveniently elsewhere, such as the time services, an overview of internal communication paradigms, bootstrapping, and creating and launching an application.

6.1 Time services

Brevix provides three time facilities; a *time-of-day clock*, *accurate elapsed time counters* and *time notification*. The time-of-day clock is used to report the current time, and to provide timestamp values associated with data. Accurate elapsed time counters provide detailed information on ephemeral resource usage and elapsed time. Time notification allows an application to be notified after a specified period of time has passed.

Time specification. Two types of time specifications are used in Brevix: *absolute time,* and *time increment*.

- Absolute time in Brevix is specified in a struct tm as defined in the ANSI C standard [ANSI89]. Brevix extends this struct to include the tm_tick field that gives the number of ticks since the start of the second.
- Time increments are specified in seconds and *ticks,* where a tick is one tenth of a nanosecond. Time increments are stored in a struct TimeIncrement.

  The Brevix value of the ANSI C manifest CLOCKS_PER_SEC is always in the range 1,000,000 to 10,000,000,000.

All Brevix implementations *represent* time values to the resolution of one tick. This is the *storage resolution*.The units in which time *advances* may be larger than the storage resolution of the time representation. This is the *measurement resolution.* The time-of-day clock and the elapsed time counters may utilize different measurement resolutions in a Brevix implementation. The measurement resolution shall not be larger than one microsecond. The values can be determined from the time service. Although measurement resolutions may vary from one installation to another, they are expected to remain constant on a running system.

Time-of-day clock. Each node in a Brevix system maintains a separate time-of-day clock. These clocks are kept synchronized to within a tolerance ε, whose value may be obtained on request. The tolerance is fixed for a given Brevix system and is of the same order as the round-trip time for an inter-node rpc call. The inter-nodal synchronization between clocks is maintained by a protocol that is to be determined.

> Note: the existence of this small variance between local values of the time-of-day clock means that time comparisons should consider two times to be equal if they differ by less than 2ε. This introduces the same uncertainty into comparison of absolute times as exists in comparison of other floating point quantities, but we are unaware of an alternative.

Brevix provides functions to read and set the time-of-day clock A desirable property of time is that it always appears to flow forward. This implies that a later attempt to determine the current time should return a greater value than an previous attempt. To maintain this property, time-of-day is

not usually set directly, but instead is changed by increasing or decreasing the rate of time-of-day advance. In exceptional cases, such as when the time-of-day clock has accidentally been set far into the future or past, its value may be updated directly. This may result in the appearance of time going backwards.

If the function that adjusts the clock has never been called, then the clock is incremented at the measurement resolution, which can be determined. While the clock is being adjusted it is incremented at a modified frequency. It is possible to determine if the clock is being adjusted.

Time counters. *Time counters* are tick-resolution counters that track elapsed time. Time counters are implemented independent of the time-of-day clock to isolate their counting from variations resulting from changing the value of the time-of-day clock.

Time counters are used in a fashion similar to stop watches. A time counter is created with its time increment set to zero. The counter may be started, at which time it begins to accumulate elapsed time. The measurement resolution of the accumulated time is implementation dependent and may be determined. The counter may be stopped and then restarted. While it is stopped, its time increment may be read or written.

Time counters are used by the Brevix accounting subsystem to track processor use; by language systems to provided detailed profile information; and by applications to measure the processor or elapsed time for code segments of interest.

Time counters do not interact with the time-of-day clock. It is possible to determine the frequency at which they are incremented.

Time notification. Applications may request to be notified after a specified period of time has elapsed, or when a specific absolute time has passed. If an absolute time is specified, notification occurs only once; if a time interval is specified, notification may be either once only or recurring with period equal to the specified time interval.

Notifications are accomplished via event variables (described in section 4.1.5 on page 48) and notifiers. Brevix guarantees that the event will be raised no sooner than the requested time. For periodic time notifications, the application should clear the event between notifications.

Notifiers may be created, deleted, armed and disarmed. It is possible to determine the amount of time remaining until a notify raises the event associated with it.

6.2 Communication services

This section outlines external and internal communication services.

6.2.1 External communication

Brevix will communicate with external systems using standard network protocols over standard networks. There may be multiple network connections in an SCS and Brevix will automatically use the appropriate network connection. To an external system, an SCS will appear as one system regardless of which network connection is used to communicate with it. It will appear multihomed if more than one interface to the network is provided. External communications will use an interface derived from one of the well known interfaces such as sockets or streams. This choice is to be determined.

6.2.2 Internal communication

6.2.3 Inter-thread communication models

The two primary communication models inside Brevix are shared memory and the procedure call (not the "remote procedure call").

Shared memory is supported through the single address space and the entities mapped into it. To provide for better performance, we use a weak consistency model for ensuring that updates are seen in a defined manner by other threads. (This is described in section 3.2.5 on page 34).

Procedure calls have call by value-result argument passing semantics (at least for C). This model applies across interfaces or not, locally or remotely.

> Refinements to this model exist to allow efficiency gains in three special cases:
>
> *batched calls*, which are sequences of calls that need no response: if the interface is instantiated as a remote one, the interface definition can allow Brevix to collect up one or more remote procedure call argument vectors and only send them for execution when the next call requiring a result is issued. This can reduce the amount of communication between the two nodes, but preserves the existing procedure call model in all but failure cases.
>
> *parallel calls*, in which multiple invocations are allowed to occur simultaneously. The semantics of this are not yet determined.
>
> *coroutines*, in the style of [Richards80].

The other primary communication model is explicit *bulk data transfer*, which is designed to support streaming large amounts of data from a customer to a provider.

> Finally, there will be some support available for direct inter-node message passing for those applications that can demonstrate that the above facilities fail to meet their functional needs. The use of this model will be discouraged: it is similar in many ways to assembly-level programming, and has similar pitfalls, especially in a system that is designed to support aggressive fault-tolerance goals.

6.3 Bootstrapping Brevix

The boot process for Brevix has not yet been fully specified, but at a minimum, it must install the interruption, pageframe, interface, time and address providers on each node. Doing so establishes the abstract machine which provides support to all of the other services and managers, as well as applications. To support data, the default entity, pageframe and parcel managers for the node, and the device managers used by the default parcel manager must also be installed at boot time. Doing so requires installation of the interface manager, which must be available to install any other service or manager. To support computation, the thread and synchronization managers are installed. Once this *magic installation* occurs, the boot process transforms itself into the primal thread, from which all Brevix threads are derived.

6.4 Running an application

An *application* is the executable code and initial data of a computer program. In a traditional system, the application is distinguished as a single file, often referred to as an *executable* or *binary image*. The use of shared libraries has blurred this definition somewhat, since not all of the executable code is stored in the binary image on such a system, but it is a useful approximation for the following discussion.

There are three distinct activities which are often taken together as *running an application*, but which are separate activities in Brevix. These are binding the binary image to virtual memory, creating a new thread of control to run the application, and causing the thread of control to begin executing at the starting point of the application.

As does Unix, Brevix distinguishes the operation of creating a new thread from the operation of executing a new image. However, the Brevix fork is more like that of Mach: the new thread is not a clone of the parent thread, and is created in a suspended state. Once a Brevix thread exists, it's machine state must be manipulated to prepare it to begin executing at the desired address. Once this is done, the thread is resumed.

In Brevix, an application may be thought of as a special kind of provider, which might exist in the system for a short duration and have a small number of customers, although it is also possible to envision a long running application, or one which is invoked by many customers. The same steps that are necessary to install an interface are also necessary to bind an application: the entities that make it up must be bound to the address space and its entry point must be made available to its caller(s). Thus, Brevix does not have an exec like service, but rather uses the convention that an application is merely a provider that exports an interface to a function called main. To run an application, a thread installs and attaches to the application's interface and then calls the application's main function. To run an application in another thread, the new thread is created, the interface to the application is installed, and the new thread is resumed at a call to main.

> Actually, the point of first call is really the programming language specific start up code for the program, but the discussion above gives the correct flavor.

It is also possible in Brevix to create a thread that is attached to the same set of resources as another thread, thereby achieving concurrent execution. Such a thread would take advantage of the synchronization and memory consistency protocols to cooperate with other threads.

7 References

[Abrossimov89a] Vadim Abrossimov, Marc Rozier, and Marc Shapiro. Generic virtual memory management for operating system kernels. *Proceedings of 12th ACM Symposium on Operating Systems Principles* (Litchfield Park, AZ, 3–6 December 1989). Published as *Operating Systems Review*, **23**(5):123–36, December 1989.

[Black90] David L. Black. Scheduling support for concurrency and parallelism in the Mach operating system. *IEEE Computer*, **23**(5):35–43, May 1990.

[Burrows90] Michael Burrows, MartínAbadi, and Roger Needham. A logic of authentication. Published as *ACM Transactions on Computer Systems*, **8**(1):18–36, February 1990.

[Burrows92a] Michael Burrows, Charles Jerian, Butler Lampson, and Timothy Mann. On-line data compression in a log-structured file system. *Proceedings of 5th International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, MA, 12–15 October 1992). Published as *Computer Architecture News*, **20** (special issue):2–9, October 1992.

[Carter91] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of Munin. *Proceedings of 13th ACM Symposium on Operating Systems Principles* (Asilomar, Pacific Grove, CA). Published as *Operating Systems Review*, **25**(5):152–64, 13–16 October 1991.

[Cheriton89] David Cheriton and Timothy P. Mann. Decentralizing a global naming service for improved performance. *ACM Transactions on Computer Systems*, **7**(2):147–83, May 1989.

[Clegg86] Frederick W. Clegg, Gary Shiu-Fan Ho, Steven R. Kusmer, and John R. Sontag. The HP-UX operating system on HP Precision Architecture computers. *Hewlett-Packard Journal*, **37**(12):4–22, December 1986.

[Gelb89] J. P. Gelb. System managed storage. *IBM Systems Journal*, **28**(1):77–103, 1989.

[Gharachorloo90] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. *Proceedings of 17th Annual International Symposium on Computer Architecture*. Published as *Computer Architecture News*, May 1990.

[Harty92] Kieran Harty and David R. Cheriton. Application-controlled physical memory using external page-cache management. *Proceedings of 5th International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, MA, 12–15 October 1992). Published as *Computer Architecture News*, **20**(special issue):187–97, October 1992.

[Hennessy90] John L. Hennessy and David A. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers, Incorporated, San Mateo, CA, 1990.

[Herrmann88] Frédéric Herrmann, François Armand, Marc Rozier, Vadim Abrossimov, Ivan Boule, Michel Gien, Marc Guillemont, Pierre Léonard, Sylvain Langlois, and Will Neuhauser. CHORUS, a new technology for building UNIX systems. *Proceedings of EUUG Autumn'88 Conference* (Cascais, Portugal), pages 1–18. EUUG, 3–7 October 1988.

[Jia90] Xiaohua Jia, Hirohiko Nakano, Kentaro Shimizu, and Mamoru Maekawa. Highly concurrent directory management in the Galaxy distributed system. *Proceedings of 10th International Conference on Distributed Computing Systems* (Paris, France, 28 May –1 June 1990), pages 416–23. IEEE, 1990.

[Karp92] Alan H. Karp and Vivek Sarkar. Data merging for shared memory multiprocessors. Technical report HPL–92–138. Hewlett-Packard Laboratories, November 1992.

[Keleher92] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. *Proceedings of 19th International Symposium on Computer Architecture* (Gold Coast, Australia), pages 13–21, 19–21 May 1992.

[Kohl93] John T. Kohl, Carl Staelin, and Michael Stonebraker. HighLight: using a log-structured file system for tertiary storage management. *Proceedings of Winter 1993 USENIX* (San Diego, CA, 25–29 January 1993), pages 435–47, January 1993.

[Lampson91] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: theory and practice. *Proceedings of 13th ACM Symposium on Operating Systems Principles* (Asilomar, Pacific Grove, CA). Published as *Operating Systems Review*, **25**(5):165–82, 13–16 October 1991.

[Leffler84] Sam Leffler, Mike Karels, and M. Kirk McKusick. Measuring and improving the performance of 4.2BSD. *Proceedings of Summer USENIX*, pages 237–52, June 1984.

[Massalin89] Henry Massalin and Calton Pu. Threads and input/output in the Synthesis kernel. *Proceedings of 12th ACM Symposium on Operating Systems Principles* (Litchfield Park, AZ, 3–6 December 1989). Published as *Operating Systems Review*, **23**(5):191–201, December 1989.

[McKusick84] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, **2**(3):181–97, August 1984.

[Mockapetris87] P. Mockapetris. Domain names – concepts and facilities. Request for comments 1034. ARPA Network Working Group, November 1987.

[Mogul86a] Jeffrey Clifford Mogul. *Representing information about files*. PhD thesis, published as Technical report STAN–CS–86–1103. Stanford University, March 1986.

[Muller91] Keith Muller and Joseph Pasquale. A high performance multi-structured file system design. *Proceedings of 13th ACM Symposium on Operating Systems Principles* (Asilomar, Pacific Grove, CA). Published as *Operating Systems Review*, **25**(5):56–67, 13–16 October 1991.

[Ousterhout89] John Ousterhout and Fred Douglis. Beating the I/O bottleneck: a case for log-structured file systems. *Operating Systems Review*, **23**(1):11–27, January 1989.

[Patterson88] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). *Proceedings of SIGMOD* (Chicago, Illinois), 1–3 June 1988.

[Peck88] Geoffrey G. Peck. Architecture of the RX operating system. Hewlett-Packard Laboratories, Draft of 17 March 1988. *HP company confidential*.

[Pike90] Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey. Plan 9 from Bell Labs. *UKUUG Summer 1990* (London), pages 1–9. United Kingdom UNIX systems User Group, Buntingford, Herts, 9–13 July 1990.

[Pu88b] Calton Pu, Henry Massalin, and John Ioannidis. The Synthesis kernel. *Computing Systems*, **1**(1):11–31, Winter 1988.

[Redell80] David D. Redell, Yogen K. Dalal, Thomas R. Horsley, Hugh C. Lauer, William C. Lynch, Paul R. McJones, Hal G. Murray, and Stephen C. Purcell. Pilot: an operating system for a personal computer. *Communications of the ACM*, **23**(2):81–92, February 1980.

[Richards80] M. Richards and J. K. M. Moody. A coroutine mechanism for BCPL. *Software—Practice and Experience*, **10**, October 1980.

[Rosenblum92] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, **10**(1):26–52, February 1992.

[Rozier88b] Marc Rozier, Vadim Abrossimov, François Armand, Ivan Boule, Michel Gien, Marc Guillemont, Frédéric Herrmann, Claude Kaiser, Sylvain Langlois, Pierre Léonard, and Will Neuhauser. CHORUS distributed operating systems. *Computing Systems* , **1**(4):305–70. USENIX, December 1988(?).

[Ruemmler91] Chris Ruemmler and John Wilkes. Disk shuffling. Technical report HPL–91–156. Hewlett-Packard Laboratories, October 1991.

[Schroeder84] Michael D. Schroeder, Andrew D. Birrell, and Roger M. Needham. Experience with Grapevine: the growth of a distributed system. *ACM Transactions on Computer Systems*, **2**(1):3–23, February 1984.

[Seltzer93] Margo Seltzer, Keith Bostic, Marshall Kirk McKusick, and Carl Staelin. An implementation of a log-structured file system for UNIX. *Proceedings of Winter 1993 USENIX* (San Diego, CA, 25–29 January 1993), pages 307–26, January 1993.

[Sheltzer86] Alan B. Sheltzer, Robert Lindell, and Gerald J. Popek. Name service locality and cache design in a distributed operating system. *Proceedings of 6th International Conference on Distributed Computing Systems* (Cambridge, Mass), pages 515–22. IEEE Computer Society Press, Catalog number 86CH22293-9, May 1986.

[Siewiorek92] Daniel P. Siewiorek and Robert S. Swarz. *Reliable computer systems: design and evaluation*. Digital Press, Second edition, 1992.

[Sinha91] Pradeep K. Sinha, Mamoru Maekawa, Kentaro Shimizu, Xiaohua Jia, Hyo Ashihara, Naoki Utsunomiya, Kyu S. Park, and Hirohiko Nakano. The Galaxy distributed operating system. *IEEE Computer*, **24**(8):34–41, 1991.

[Sinha91a] Pradeep K. Sinha, Kentaro Shimizu, Naoki Utsunomiya, Hirohiko Nakano, and Mamoru Maekawa. Network-transparent object naming and locating in the Galaxy distributed operating system. *Journal of Information Processing*, **14**(3):310–24, 1991.

[Sventek88] Joseph Sventek. *Constraint processing in the ANSA Trader.* Advanced Networked Systems Architecture, Cambridge, UK, 23 September 1988.

[Sventek89] J. S. Sventek, A. J. Herbert, D. J. Otway, and D. M. Eyre. *Introduction to the ANSA architecture.* Architecture Projects management Ltd. 24 Hills Road, Cambridge, UK, Draft of August 1989.

[TCS85] *Department of Defense trusted computer system evaluation criteria.* National Computer Security Center, Fort Meade, MD, DOD 5200.28-STD, supercedes CSC–STD–001–83 (15 Aug. 1983), December 1985. Department of Defense standard.

[Wilkes90] John Wilkes and Raymie Stata. *Specifying data availability in multi-device file systems.* Technical Report HPL–CSP–90–6. Concurrent Systems Project, Hewlett-Packard Laboratories, 1 April 1990.

[Wilkes91] John Wilkes and Raymie Stata. Specifying data availability in multi-device file systems. *Position paper for 4th* ACM-SIGOPS *European Workshop* (Bologna, 3–5 September 1990). Published as *Operating Systems Review*, **25**(1):56–9, January 1991.

[Wilkes92] John Wilkes. *Hamlyn—an interface for sender-based communications.* Technical Report HPL–OSR–92–13. Operating Systems Research Department, Hewlett-Packard Laboratories, 30 November 1992. HP Company Confidential.

[Wilkes92a] John Wilkes. *Specifying scalability*. Technical Report HPL–OSR–92–16. Operating Systems Research Department, Hewlett-Packard Laboratories, 10 December 1992.

[Wilkes93] John Wilkes. *Fault tolerance, SCS, and the price of fish.* Technical Report HPL–OSR–93–1. Operating Systems Research Department, Hewlett-Packard Laboratories, 7 January 1993.