A Study of Fully Open Computing Systems

Yeshayahu Artsy (Landesman)

Computer Sciences Technical Report #769

1987

A STUDY OF FULLY OPEN COMPUTING SYSTEMS

by

YESHAYAHU ARTSY (LANDESMAN)

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy
(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN — MADISON

1987

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Full openness is desirable to allow applications to obtain their specific service needs and implement particular semantics. In a fully open system *any* application can choose, add, replace, and extend services as well as resources. However, in a multiuser environment protection considerations impose constraints on openness. Enforcing protection may reduce sharing, entail execution overhead, or increase programming complexity. This dissertation studies how a fully open computing system can be constructed for a multiuser environment and explores the aspects of openness.

We investigate these issues at three levels of abstraction: model, design, and implementation. The dissertation defines a model of a fully open multiprocessor computing system, describes a specific design based on the model, and examines various techniques by which that design can be implemented. Our study provides insights into the extent of openness and into the interplay between openness, protection, efficiency, and complexity.

The model, called the **FOCS model**, identifies a minimal set of mandatory services required by considerations of protection and sharing. The role of the operating system is then defined as providing only these services. All other services can be provided by any application. The model offers a novel view of resource management, based on the notion of *resource ownership*. Applications own private and shared resources and provide the policies as well as the mechanisms for using them. The protection mechanism of the model employs the concepts of *encapsulation*, *ownership*, and *light-weight capabilities*. The model introduces the notion of *activities*, which are *threads of control* that represent computations across multiple address spaces.

At the design level we explore the mechanisms needed to support protected openness and examine their efficiency and complexity. The design elaborates on the issues of processor and memory management, accounting, naming, and language support. Based on the examination of implementation techniques, we conclude that a fully open computing system can be implemented with contemporary technology. Our initial analysis suggests that the execution overhead of such a system would not be large and that programming complexity would be comparable to that in conventional systems.

ACKNOWLEDGEMENTS

I would like to express my gratitude to my advisor, Miron Livny, for his assistance during this research. I am indebted to him for his encouragement and countless suggestions. His critical reading and persistence on the details have greatly improved the thesis.

I feel grateful to my committee members, Marvin Solomon, Mike Carey, Jim Goodman, and Rafi Lazimi. Marv and my former advisor, Raphael Finkel, were endless sources of ideas and inspiration. Their astute thinking and clarity of communication are admirable. Special thanks go to Mike for being always willing to discuss databases and to contribute smart advice. My thanks to Jim and Rafi for participating in my committee.

I want to acknowledge many people that made my stay in the department pleasant, and in particular Sheryl who was always eager to give a hand or lend a smile; Lorene for her help and for enriching my son's stamp collection; Bob for taking care of the department; Paul, Scott and the "lab rats" for trying their best to make my workstation cooperative.

I enjoyed having many good friends during my five years in Madison, and all deserve special acknowledgements. In particular, my partner in several projects and offices Henry Chang; former officemates Hari, Prasun, Kishore, Erez, and Cui-Qing Yang, and to current officemates Phil, Henry, and Sam for occasionally lending a listening ear but always distracting my thesis work with enjoyable discussions (life beyond computers); P.B. Schecter and Dave Kamowitz who shared my Israeli jokes; Tony Rich for his enthusiastic effort to "Englishize" this dissertation; class mates Matt, Giri, Rajiv, Suhas, Bob, Mary and many more; Betty and Chet for being our hosts in the first year in Madison; Irene of the office for foreign students for helping foreigners feel at home; and the Chabad House friends for being the best family away from home—especially Jenny for trying to cheer me up by letting me win at Ping-Pong.

But most of all I thank my family for all your love and support: Paulette for your endless efforts to keep my little nest abroad intact Ariel and Serge for being such great buddies; my in-laws Claire, Claude, Lolo and Simon for your continuous encouragement; Tzippi for your immense moral support, and my parents, Masha and Yechiel Landesman, for instilling in me the desire to always do the best I can.

Finally, my children Avishay and Elinor deserve special thanks for being so considerate and understanding. Light the light back kids, I'm home on a full-time dad position. There are no words I can use to express my thanks to my wife, Chantal. Thanks for tolerating Eagle Heights for five l-o-n-g years, for assuming extra burdens at home, for putting up with all the hardship of poor-student's salary, and for being a great partner.

Chapter 1

Introduction

Imagine you have just completed the design of a next-generation, general-purpose operating system (**OS**). You have thoroughly examined the needs of its prospective applications and carefully crafted the services to support them. The inevitable question is that of how soon designers of databases, programming languages, and other applications will complain about missing features or inadequate facilities in the new OS. This phenomenon is a symptom of two problems designers of general-purpose OS's face: First, they cannot predict at design time the service needs of all potential applications. During the lifetime of an OS the needs of its users change as new applications emerge. Second, the designers cannot satisfy all needs. Needs may conflict, so satisfying the needs of one application may prevent satisfying those of another application, or impose unbearable overheads on the latter.

These problems were observed in many different environments [94, 113, 125]. As the following cases demonstrate, OS services considered "generally adequate" fail to support the needs of certain applications. Buffering services provided by the OS simplify programming, but are very inefficient for applications, such as a database management system (**DBMS**), that require particular allocation and replacement policies [36, 37, 125]. If the DBMS implements its own buffer management services, then it may suffer large overhead due to redundant buffering in the OS and the memory-replacement policy of the OS [94, 125]. Interprocess communication (**IPC**) services designed to support diverse communication paradigms [15] may impose undesirable semantics on a distributed programming language with particular features [112]. Another general IPC mechanism [107] has become inefficient and insufficient for an application of multiple, tightly-coupled processes [121, 122]. Because of the protection measures employed by the mechanism, it cannot support the specific semantics of memory sharing and the efficient address-domain crossing required by that application. Scheduling services in a time-sharing environment relieve applications from the complexity of CPU time allocation, but are inadequate for real-time applications. If a real-time application has to run in such an environment, these requirements necessitate adding redundant and complex mechanisms to the OS [57]. These mechanisms, in turn, incur execution overhead and may obstruct real-time scheduling requirements. Moreover, while an OS usually supports one model of communication, a real-time application may require that several models are efficiently accommodated, and that they can be chosen dynamically [111]. Resource-management services of the OS provide a virtual, higher-level view of the hardware, and thus simplify programming; however, they are inadequate for an extendible DBMS which requires direct access to disks for reasons of efficiency and correct placement of objects [24]. Furthermore, testing a new device driver above the service interface of an OS is a complex, costly, and rarely correct task [139]. Finally, a rigid file structure imposed by the OS can restrain the applicability of a user-friendly interface that requires a different file organization [117].

The failure of these general-purpose services to support specific needs is inherent to their generality — they cannot adjust to each application's needs. Hence they inflict excessive overhead or unwanted semantics on some applications. To overcome this problem, services should be adaptable. We believe that in order to support adaptability, the OS should be restructable and extensible. In addition, it should facilitate sharing of resources and services between applications. An *ideal* OS would support adaptability *dynamically*, in a *protected* way, and without incurring extra overheads. To support dynamic adaptability, an OS should be *open* to its users, letting them choose, add, replace, and extend OS services and resources. An open system will be ideal if protected openness is accommodated efficiently. One may argue that the failure of general-purpose services can be overcome by having multiple application-tailored OS's. This alternative is less plausible. It does not support sharing; providing a full-scale OS for each application is costly.

The open system approach has the promise to solve the design problems mentioned earlier: It lifts the burdens of foreseeing the needs of all potential applications, and of identifying an optimal set of services. Yet openness is constrained by considerations of protection and efficiency. In this dissertation we investigate the interplay between openness, protection, efficiency, and complexity in a multiuser environment. Through an elaborate description of a fully open computing system we expose the problems of openness and examine potential solutions. Our thesis is that such a system is amenable to applications' needs, it is viable, and it can be implemented by employing features of contemporary technology.

1.1. Open *vs.* Closed Systems

The open system approach can be better defined by comparing it with the *closed system* approach. In the latter approach the OS owns all physical resources and provides mandatory services for using them. The service interface of the OS cannot be changed or bypassed by applications. An application, for instance, cannot own a CPU or dictate its allocation. It cannot access a disk directly without using intermediary OS services. The service interface can be at a high or low level.

In a closed system with a high-level service interface, the OS provides ample functionally rich services. It offers applications the convenience of having a virtual view of resources. As shown in Figure 1a, applications are "small" (or "simple"), since the system provides most of the services they need. This approach, however, has several severe drawbacks. It inflicts on OS designers and implementors a heavy burden of continuously extending and modifying services to accommodate new needs. Some modifications are often impossible because they conflict with existing services. Moreover, in an all-if-anything service-provision style, each application pays a performance penalty for all features, including those it does not want. As we learned from the design of a flexible IPC mechanism [16, 53], adding features to enhance generality increases the OS complexity and execution overhead, and still falls short of satisfying all users.

In a closed system with a low-level service interface, the OS provides a reduced set of services, as depicted in Figure 1b. Since applications supplement most of the necessary services, they can presumably satisfy their needs. For instance, an application can efficiently access the physical resources. However, writing applications in such a system is more complex, and might be unbearable for many users. Application-level user interfaces (see Figures 1a and 1b) can mitigate only

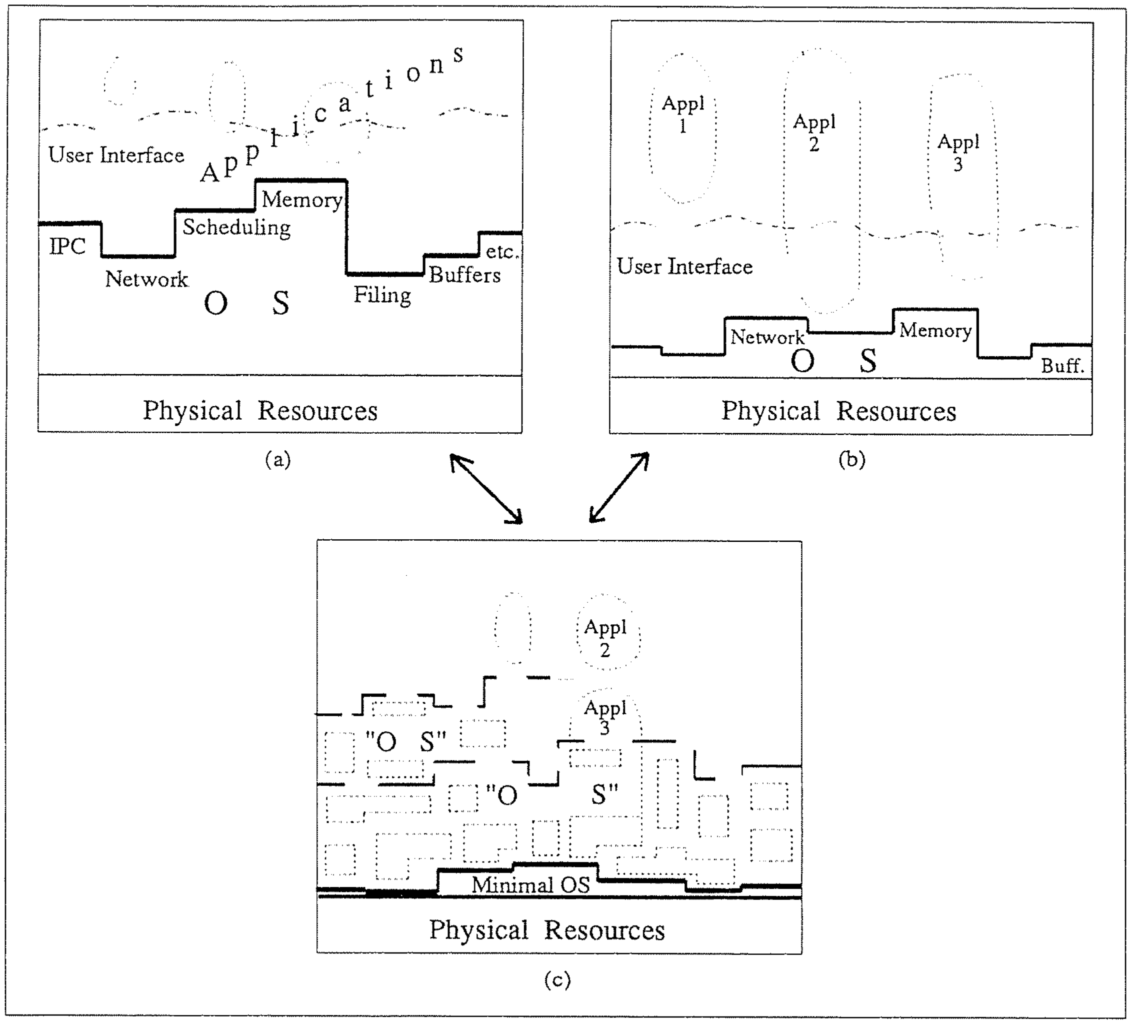

Figure 1: The Operating System in Various Approaches
(a) Closed OS with a high-level service interface; (b) Closed OS with a low-level service interface;
(c) Open OS with a low- to high-level service interface.

some of the programming complexity. Furthermore, since the OS does not support sharing of services between applications, this approach imposes a large programming overhead on each application writer.

An open OS tries to combine the benefits of these two variations of the closed-system approach, as shown in Figure 1c. It provides a low-level, mandatory service interface, which offers efficient access to resources and the ability to implement customized services. The OS allows applications to construct different, higher-level service interfaces. These interfaces are modular, and thus can be selected or bypassed by applications.

1.2. Full Openness

How open should a system be? In order to achieve maximal adaptability, a system should be open to *all* applications. Any application should be able to select services or resources in all domains, including the IPC, processor and memory management service domains. We refer to such a system as *fully open*.

Full openness raises the following intriguing and intricate problems.

(1) **How is openness achieved**? Specifically, how is the system structured, what are the constructs that represent resources and services, and what are the primitives that accommodate dynamic customization? One aspect of this problem is whether openness requires the support of a particular computer architecture. Another aspect is the extent to which service domains can be selected independently of each other while being shared by different applications. In most systems IPC, processor and memory management are tightly coupled, so selecting a service in one domain dictates using particular services in other domains.

(2) **How much does protection constrain openness**? A multiuser environment dictates that resources are protected from being improperly used, and that users are protected from each other. Therefore, some services *must* be retained in the secure domain of the OS, and some restrictions *must* be imposed on users. These services should also assist applications in implementing their own protection mechanisms. Previous research in operating systems has not established the desired minimal set of such services and restrictions.

(3) **To what extent does protected openness conflict with efficiency or simplicity**? Specifically, what are the tradeoffs between these aspects? Can efficiency be improved through architectural support, without requiring an overly complex architecture? Do protection restrictions imply a complex programming style?

1.3. Research Overview

The goal of this research is to investigate the interplay between openness and the constraining aspects of protection, efficiency, and complexity. The interplay is studied in a multiuser environment in which the protection of both shared and private resources has to be addressed. The aspects of efficiency that are examined include execution and resource allocation efficiency. Complexity includes the aspects of programming and computer architecture.

The research method chosen is to experiment with a fully open computing system (**FOCS**) defined at three levels of abstraction. We have developed a model of a fully open computing system called the **FOCS model.** The model defines the system components and their interrelationship. It has been used to design a fully open OS for a multiprocessor system. Following the design phase, we have examined techniques to implement a FOCS.

Each of these stages of refinement contributed to our understanding of the problems and provided a framework to evaluate solutions at different levels of detail. The model helped us study how to support the required functionality. The design stage and the examination of implementation techniques provided insights into the potential performance of a FOCS. We found that the modeling process was instrumental in exposing many obstacles to achieving full openness, and it shed light on dependencies between the components of a FOCS. The design was useful in experimenting with concrete solutions to these problems, and in evaluating their implied efficiency and complexity. The implementation study has shown the practicality of a FOCS. It has illuminated the architectural complexity and execution efficiency attained in such a system.

1.4. Lessons from Related Work

A new model was necessary since no other model or OS design supports openness to its ultimate extent. Several models, such as the object [68], client–server, and virtual machine models [59], as well as many systems motivated by them [14, 19, 41, 54, 85, 107, 115] open up high-level services or virtual resources to applications. Some support construction of virtual OS's. Others allow applications to partially direct the policies of the OS. However, none of them enables general users to manage shared physical resources or provide low-level services. Most services for IPC, processor and memory management in these systems are provided by irreplaceable OS components. A few single-user systems [80, 108, 129] open OS policies and mechanisms to the user, but they ignore aspects of protection and sharing that are important in a multiuser environment. Therefore, these models and systems cannot provide a framework for exploring the aspects of openness in a multiuser environment.

Moreover, we have not identified a simple way to extend any of these models to define a FOCS without altering its major features. In some cases, the desired notions, such as letting any application directly access physical resources and provide allocation policies, conflict with the philosophy of the model [63, 68]. Some systems isolate their entities from each other and impose protection barriers, which in turn imply high communication costs [66, 99, 140, 144]. As a result, opening low-level or frequent services is impractical.

The FOCS model borrows several concepts from related work, but upgrades them in the pursuit of full openness. Some concepts, such as decomposing a computing system into objects, are extended with features that support full, protected openness. Other concepts, such as using capabilities for addressing and access control, are simplified by removing features that obstruct openness.

1.5. Organization of the Dissertation

The rest of the dissertation is organized as follows. Chapter 2 surveys models, designs, and other conceptual frameworks that have introduced concepts relevant to system openness. This chapter points out the main characteristics of past work that either support or obstruct full openness.

The FOCS model is presented in Chapter 3. The chapter describes an abstract system and motivates the necessary restrictions on openness due to protection requirements. It discusses how the general principles apply to diverse resources, elaborating on CPU and memory management.

In Chapter 4 a system design based on the model is detailed. The chapter discusses services and techniques by which the abstract system can be realized. The discussion shows the aspects of efficiency and programming complexity in a FOCS.

The feasibility of an implementation of the design is discussed in Chapter 5. The discussion focuses on selected issues of service representation, CPU management, addressing, and memory management. The chapter examines the implied execution efficiency and architectural complexity of a FOCS.

Chapter 6 summarizes the lessons on system openness that stem from this research. It discusses bounds of openness, the interplay between the aspects of openness, dependencies between service domains, and hierarchies. Chapter 7 concludes the dissertation with a summary of the key ideas, contributions to Computer Science, and directions for future research.

Chapter 2

Related Concepts

Although no previous work has addressed full openness, many systems introduced concepts which are amenable to full openness. To put our work in perspective, this chapter surveys models and OS designs which set openness as a major goal, or which can be considered open to a certain extent. We present their major concepts, discuss their approach to system openness, and analyze their characteristics that either support or obstruct full openness. The diverse systems surveyed here are grouped into categories based on their central abstraction or their structure. Each category is preceded with a brief characterization of the systems in it and the common lessons they provide to system openness. We discuss also programming language concepts that support system openness.

2.1. Object–Oriented and Capability–Based Systems

An object-oriented system is structured as a collection of objects. An object, according to the *object model* as defined by Jones [68], encapsulates invariant properties and operations. The only way one object can determine or manipulate the state of another object is by invoking the latter's operations. An object has a *type*. The system supports type inheritance and the construction of composite objects out of basic-type objects. These characteristics together with the support of dynamic naming of objects provide a good foundation for an open system, since they accommodate easy selection and replacement of services. The model, however, cannot easily support memory manipulation or scheduling of one object by another one. In systems strictly following this model these services are reserved for a special object — the *OS kernel*. Moreover, the model encourages isolation of objects from each other due to protection requirements [69]. This isolation renders interobject communication costly, and thus effectively precludes the relegation of related OS functions to separate objects.

The object model and most object-oriented systems base protection and dynamic naming primarily on capabilities, although this property is not inherent in the concept of objects. A *capability* was initially proposed as a secure pointer that names an object and lists a set of operations or rights regarding that object [48]. It is also useful as a protected address [52]. A capability-based system, therefore, provides a good foundation for an open system, since it offers dynamic selection of services in a unified and protected way. In practice, most capability-based systems employ generic capabilities; capability management, i.e. inheritance, copying, and transfer of capabilities, is usually performed by the OS or by specialized hardware. Hence, it incurs extra execution overhead or architectural complexity. We elaborate below on these issues and discuss several object-oriented or capability-based systems.

The primary motivation for Hydra [143, 144] was to allow *any* application-level program to provide OS services to *any other* program. To protect programs from each other, Hydra's designers painstakingly defined an extensive protection mechanism based on elaborate capabilities. The mechanism can cope with most conceivable protection problems (*cf* Ch. 7 in [144], for example).

As the designers have admitted, this mechanism is too restrictive for most applications, which rarely need most of its features. An object in Hydra is viewed as an instance of an abstract type, modeled after a Simula class [46]. Each object has a capability list which can be dynamically inherited, amplified, or reduced. Hydra supports strong typing and performs object-type checking at execution time, at the expense of extendibility. Among the object types supported by the kernel are procedure, process, I/O device, file, and policy. Because the granularity of some objects is too small, and since object creation and capability verification require extensive work, every computation suffers a large execution overhead. The overhead, in turn, is a crucial barrier to effective OS openness.

As Hydra's designers realized, frequent decision-making functions, such as memory replacement and process dispatching, could not be practically implemented by programs other than the kernel. To facilitate OS openness, they defined the principle of *mechanism–policy separation*. The kernel implements mechanisms. It accepts policy specifications from higher level software, which can be supplemented by user programs. However, this principle has never been clearly applied. Some policy functions remain in the kernel; some kernel mechanisms allow only certain policies. For example, Hydra demonstrates that CPU scheduling policy can be specified by user-level programs (called *PM*'s). Since a PM is fully trusted by the kernel, it has to be *ordained* by some installation authority, and cannot be installed by *every* user. Moreover, because of the large communication overhead, the kernel–PM interface is inflexible and so a PM cannot easily react to changing scheduling needs. For similar reasons, most of the memory management functions are retained in the kernel. (This decision is in part due to the restrictive hardware for which Hydra was designed.) Unfortunately, memory management is bound to CPU management.For instance, memory allocation decisions are based on the scheduling state of processes. Hence, these two service domains could not be opened independently.

StarOS [70] mitigates some of these problems by supporting larger objects, such as a module, and simplifying the capabilities. StarOS and Medusa [100] have introduced the concept of a *task force*, which is a collection of small processes. A task force typically represents a single application or implements a utility. Its processes may share data and resources, and can be scheduled as a group. A task force could potentially replace certain OS utilities. In these systems, however, most of the OS functions are provided by the OS kernel and irreplaceable utilities.

CAP [96, 140] incorporates capability-based protection with a hierarchical process tree. To improve performance, the protection mechanism is directly supported by the architecture and by microcoded services. The goal of CAP was to let *any* process supervise the services of its offspring processes. Accordingly, a process is a *coordinator* of its subtree of processes for all services they need, including CPU scheduling and memory management. A process has a capability pointing back to its parent (that is, to a procedure or to a capability pointing back to the parent's parent) for each service that the process cannot or is not allowed to provide. The protected services are provided by the root of the hierarchy — the *master coordinator*. This structure simplifies service provision and control, and allows each application to be its own virtual OS. In addition, a process can obtain real CPU time to multiplex among its offspring processes. However, the rigid, multilevel hierarchy reduces sharing. The backward capability resolution mechanism, which is needed for service invocation, has resulted in performance degradation and in increased architectural complexity.

Consequentially, only two levels of the hierarchy were implemented [86]. To reduce invocation overhead, some OS services can be executed as *protected procedures* within the domain of each process. Nonetheless, these services cannot be replaced by any process.

The iAPX-432 computer architecture and its iMAX operating system [71] are an integrated, object-oriented system [99]. The system enhances the object model of Hydra and the architectural support of CAP. It provides OS openness to a large extent. iMAX defines interfaces through which applications can introduce new facilities or replace the existing ones. Several template packages allow an application to create a type manager (a process), which can control other processes. A Process_Manager (PM) can create new processes, control their progress, and delete them. Using a generic queueing and IPC mechanism [45], a PM directs the kernel to which queue to move a process. However, the direction is a *hint*, subject to the kernel's consent. And, a process or its "servers" cannot specify scheduling requirements to the process' PM. Any application can build its own IPC mechanism, using ports and the simplified *send* and *receive* operations provided by the architecture. Hierarchical IPC mechanisms are also possible, but the overhead incurred by object maintenance and by access control is very high. User-level memory managers can create objects of virtual segments and allocate them to physical memory, but replacement policies are retained in the kernel. Any user can create a new device implementation without altering the system code. Overall, the extended facilities of this system have required a complex architecture which evidently exhibits poor performance.

Eden [14] supports object management in a distributed environment. It differs from its predecessors, Hydra and iMAX, in three important aspects. First, an object can be active. It may contain multiple, light-weight processes, for whom the object controls memory management, scheduling, and IPC functions. However, the kernel alone controls these same services for objects. Second, objects communicate via invocations. It is the invoked object, not the kernel, that verifies access rights. Invocations are performed via a message-based, remote-procedure-call (RPC) mechanism [25], which is too slow to support frequent or urgent operations — even in a shared-memory environment. Last, Eden and its tailored language EPL support capabilities as a first-class type. A capability can be manipulated as easily as any datum; capability constants can be assigned at compile time. This property greatly reduces the overhead of capability management.

Cronus [110] offers further openness. It extends Eden by allowing processes to be the managers of one or more object types, and to perform access control and resource monitoring on them.

Argus [89] and Clouds [84, 120] take a language-based approach to construct fault-tolerant distributed systems. A *guardian* (Argus) and *object* (Clouds) can be thought of as a virtual, per application OS. It provides application-level control, and implements application-dependent semantics. A guardian defines handlers, executes them upon invocation, and guards various resources. In other aspects, these systems are closed according to our definition.

2.2. Client–Server, Message–Based Systems

The client–server model has never been formally defined; its *de facto* definition stems from many message–based systems that claim to follow such a model. This model focuses on the relationship between the components of a system. The OS is structured as a kernel and a collection of utilities that provide services to applications. Application-level servers may provide services to other applications. In practice, however, these services implement high-level functions only, and most of the OS utilities are irreplaceable. In fact, the execution cost of message passing is detrimental to opening low-level or frequently-used services to non-kernel processes. The OS in such systems supports a single, generic IPC mechanism. A customized IPC mechanism can be constructed only above the default one, and only if their semantics do not clash. A customized mechanism is rendered inefficient if its semantics are simpler than those supported by the default mechanism. The following systems further illustrate these issues.

Demos [19] has introduced the above system structure in a single machine environment. It has been extended by Arachne [119], Charlotte [16], and Demos/MP [91] to a distributed environment. The three systems address policy–mechanism separation by retaining mechanisms in the kernel and relegating policy functions to *utility processes*. These systems are open, but only to a limited extent, as demonstrated in three respects. First, some of the mechanisms restrict the policies that can be supported. Second, some of the utilities are distinguished processes whose functions cannot be performed by ordinary processes. For instance, the process and memory management utility is a single, distributed monitor, which cannot be replaced by users. Third, the utilities are responsible for higher-level functions, e.g. process placement, and do not control lower-level functions, e.g. process scheduling or message transfer.

Thoth [39] is an early single-machine predecessor of the distributed V kernel [40, 41]. These systems add several interesting aspects to openness. First, Thoth supports the notion of a *team*—a collection of processes sharing an address space. In a single-team environment, kernel functions can be linked with the application processes, saving context-switching time on system calls. In addition, a team can provide several kernel functions. Second, the process structure is a tree in which a parent process can control the progress of its child processes. However, because scheduling is not open, the parent cannot schedule them directly. Third, these systems support a reduced, efficient IPC mechanism, which thus is attractive to openness. However, application-specific, higher-level IPC mechanisms cannot be easily built above this mechanism. For example, it is hard to facilitate nonblocking send-receive or memory-mapped IPC.

RIG [83] extends Demos by relegating to user-level servers the time and network services which traditionally have been provided by OS kernels. Even I/O device drivers can run as user-level servers. Nonetheless, these servers cannot be replaced or bypassed by general users.

Accent [107], a descendant of RIG, provides uniform process-kernel interaction through the IPC mechanism. Memory management, CPU scheduling, and I/O operations are handled through messages. Theoretically, a process may choose its kernel by passing its *kernel port* to another process. The high-level, elaborate IPC mechanism is extensible but *closed* to applications. Accent integrates memory management with the IPC mechanism and the filing system [56], an extension to similar notions employed by Multics [23] and Tenex [30]. This integration allows relegating memory management functions to user-level servers. Accent opens memory management to some extent by letting applications create virtual memory objects and specify their backup policy. On a page fault, a message is sent to the *guardian* of the faulting process to decide what to do.

Nonetheless, memory replacement policies as well as process scheduling are solely decided by the kernel.

Mach [12] upgrades Accent and extends its memory management facility [134]. It lets application-level tasks specify sharing and inheritance rules. An external pager (a task) can communicate to the kernel requests to pin or unpin data objects in core, to initialize memory objects, and to initiate page replacement. Although this facility largely opens memory management, it cannot replace the kernel's role as *the* memory manager because of its execution overhead. Most notably, this facility does not suit real-time applications, or applications with frequently-changing mapping requirements. Mach recently introduced a new facility of multithreading [135]. Although tasks can create and destroy threads, they have little control over thread scheduling. For instance, a task cannot indicate execution dependencies between threads. Or, if a thread is blocked in a task because a non-sharable resource is occupied by another thread, the task cannot dictate that the latter thread be dispatched urgently or aborted.

SODA [74, 76] and Amoeba [95, 131] address openness by separating processes physically. They assume a multicomputer environment in which each computer runs a single process at a time. (In Amoeba, however, multiple threads can run at one process.) Hence, SODA and Amoeba simply ignore many of the traditional OS functions. They do not concern themselves with memory allocation policies. Process scheduling—apart from process placement—is virtually null. SODA and Amoeba offer a simple, connectionless IPC mechanism. It is possible to provide IPC services outside the kernel, e.g. to provide connection-based communication or message screening, but such a facility might be inefficient. Other services, in particular device and filing services, are fully open. Amoeba demonstrates the ability of using "cheap" capabilities, stored in a user's space and verified by specialized hardware. It bases openness on the notion that servers can bill customers for services and execution time. It is noteworthy that SODA has explored the ramifications of simplifying a system as far as possible, similar to our goal in exploring how far a system can be opened.

2.3. Layered and Hierarchical Systems

A layered system is divided functionally, assigning a different function or a service class to each layer. Models of open, layered systems were formally defined for network communication [132, 146], but no model has been formalized for operating systems.[1] Layering a system helps to "untangle knotty services" so that they can be handled separately. Once functions are separated, a layered system can be open by allowing applications to construct higher-layer facilities above lower-layer ones.

A hierarchical system, in our definition, supports different logical (sub)*systems*, each of which may be a layered system. The virtual machine model [59] defines a computing system organized in a tree structure of virtual machines, each of which possibly runs a different operating system. The physical resources are retained by the lower level of the hierarchy—the real machine and the real

[1]Notice, however, that these models define *openness* differently than we do, in the sense that they allow multiple high level protocols to be constructed above a lower-level protocol. Also, the description of the THE system [49] is viewed by some as an informal model definition of a layered OS.

OS—and are multiplexed among the higher levels. Although each user may install a full-fledged OS, the model does not allow users to dictate allocation policies or mechanisms for the physical resources. As shown below, the rigid structure of systems following the layered or hierarchical approach impedes openness, since it restricts sharing as well as dynamic replacement of resources and services.

THE [49] was an early layered system which focused mainly on the internal structure of the OS for reasons of improved resource utilization and ease of testing. THE divides OS functions into a rigid hierarchy of layers, each implementing one or more independent abstractions. All functions are performed by sequential processes — I/O drivers, a storage controller, and a scheduler, as well as user programs, are each a sequential process. Since the processes implementing OS functions completely trust each other, they cannot be replaced by user processes.

Like THE, Venus [88] was built as a hierarchy of levels of abstraction. It was designed for a small community of cooperating and mutually-trusted users. Therefore, Venus grants its users some ability to access the hardware features and to be involved in the OS mechanisms through shared procedures. Users are given *virtual devices* which they can manage. Resource management is primarily provided by independent processes so that applications could potentially manage physical resources. In practice, however, *all* physical resources are owned by the OS, which also retains all device management functions.

Swift [44] is a recent layered system that focuses on programming methodology, not on system openness. Like THE and Venus, it does not let applications supplement system functions. Each layer is implemented by a module; a logical function is carried out by a task that crosses several layers. Tasks execute in a single shared address space. Swift allows cyclic dependency between layers and reverse invocations. A lower layer can call higher layers synchronously. This feature offers efficient communication and a simple synchronization mechanism between different layers of abstraction. It is useful for an open system, where frequent operations, e.g. interrupt handling, may be carried out by separate components.

An unnamed system by Habermann *et al* [63] is a hybrid of a layered and a hierarchical system. The designers' goal was to develop families of operating systems, the software equivalent of computer architecture families such as PDP [21] and IBM Systems 360/370 [116]. The OS functionality is statically divided into a rigid hierarchy of *functional levels*. Each level can have several implementations, thus forming collectively a tree of OS's. A path in the tree is analogous to an OS. It is assumed—and checked at compile time—that a level knows the interfaces of the underlying levels and complies with them. These features support static openness, which suits system designers rather than user applications. Furthermore, special instructions allow cross-space invocations. However, sharing among the levels is limited. An additional drawback is that interlevel communication is highly time-consuming, since at each invocation a complete address space must be created.

VM [115] supports a rigid hierarchy of virtual machines running virtual OS's. It allows *any* user to introduce a full-fledged virtual OS above another virtual or real OS. In this respect, the system is fully open, including being open to device management. However, this approach poses several obstacles to dynamic, full openness. A user's virtual machine controls only virtual resources. Except for a few implementation short-cuts, a virtual machine has limited or no control over the policies employed by the underlying machines. Access services and resource management functions traverse several hierarchical levels, which decrease their efficiency. Intermachine communication is cumbersome: it requires static authorization, lengthy setup, and complex communication protocols [6, 66]. Resource sharing between OS's at various paths or at non-neighboring levels of the hierarchy is very constrained. Moreover, VM incurs programming complexity and space inefficiency, since even changing one feature in an existing OS requires creating a new, full-fledged OS. As a final remark, the implementation of VM shows that with an adequate computer architecture and with tuned mechanisms, the achieved performance is reasonably good.

2.4. Single–User Open Systems

An unnamed system by Lampson and Sproull [80], Pilot [108], and Cedar [129] represent three generations of single-user open systems. They let the user replace many OS functions, since they do not concern themselves with most of the aspects of sharing and protection. In this respect, these systems represent an extreme departure from the other systems discussed here.

The main design goal of the first of these systems was to allow an application program "to reject, accept, modify or extend" any of the OS facilities. Using the *Junta* mechanism, an application can literally remove from memory those OS services it does not need, or install its own ones. Moreover, it can read the system state from disk and take over the machine. Actually, the OS is activated only at entry-procedure calls, and hence an application can execute without preemption. This approach towards *full openness* is very simplistic due to the assumed environment. Protection has been considered to a very limited extent, and fairness is not an issue of concern. In addition, virtual memory has not been supported because of the lack of adequate hardware support. The IPC mechanism is almost null—only via disk files. Another characteristic of the designers' approach to openness is that some interfaces are rigorously defined (that is, standardized), but applications can implement them with different semantics.

Pilot is written in Mesa [7] and is tightly coupled to the language. It was designed as a powerful runtime support package that implements the semantics of Mesa, e.g. process creation, and relies on Mesa services, e.g. process scheduling. The kernel and the user processes share a single address space. Openness is supported by structuring the kernel and by defining standard interfaces. The kernel is "horizontally" structured as a collection of facilities, each implementing a service domain. A facility is "vertically" divided into kernel–manager layers, analogous to the policy–mechanism separation of Hydra. A facility is composed of interfaces and implementation modules. Most of the I/O devices can be directly accessed through low-level services. Some services, however, are closed. For instance, memory replacement policies are retained in the kernel. Like its predecessor, Pilot deliberately ignores many protection problems, such as *Trojan Horse* programs implanted in the system. Apart from using the type checking facility of Mesa to prevent errors, the sole protection measure of Pilot is to prevent chaos, e.g. by excluding concurrent access to system files.

Cedar is an extension of Pilot. It is similarly tightly-coupled with the Cedar language—itself an extension of Mesa—and with the Cedar machine. Its approach toward openness is similar to that of Pilot. Cedar extends the notion of abstract device interfaces, which let the user control devices without knowing the details of the physical resources. Cedar hides the management of physical memory in the microcode level and in a single memory management layer. All other system components, including device drivers, run in virtual memory; they cannot dictate its mapping to physical memory. In Cedar, as in Pilot, a program gets bound to the interfaces it needs through the language's import and export constructs.

2.5. Other System Designs

The major contribution of Multics [23, 98] to system openness is its support for dynamic selection and binding of services. The entire system can be viewed as a collection of named segments. A program can dynamically bind existing segments, including those of the supervisor, or introduce new ones. System services can be invoked efficiently, since all segments bound to a program run in a single address space. However, this approach has necessitated a complex memory architecture. Address translation requires an extra level of indirection, the "linkage area", which incurs overhead on every memory reference. Certain supervisor services can be activated only through cross-address-space calls. The supervisor executes on the caller's thread of control, and hence it preserves the caller's scheduling precedence. Employing the notion of memory-mapped files and devices, Multics unifies memory, file, and I/O management. The IPC mechanism [123], however, has not been integrated with memory management. Overall, Multics cannot be considered open, since most of the traditional OS functions are retained in the supervisor. Furthermore, its protection mechanism based on a hierarchy of 16 rings is too rigid for an open system, in which more dynamic transfer of control between servers and customers is needed.

Unix™ [109, 137] was designed as a small OS with only a few default options. Users, therefore, have the flexibility to define their preferred options. Through the *shell* interface [32], users customize their command language, terminal options, and a variety of execution environment parameters. Unix employs a device-independent I/O mechanism which allows users to add new devices and control them, albeit not dynamically. An application can access a device in a "raw mode," namely, to dictate the device's organization; it cannot, however, control access scheduling. Processes form a static tree structure. A parent process can stop, resume, signal, or kill its child processes. In all other respects, Unix is a closed system. The kernel alone implements CPU scheduling, IPC, memory management, filing, buffering, and network services. The IPC mechanism, based on *signals* and *pipes*, restricts the ability of applications to customize their IPC. As observed by others [29, 105, 135], the major obstacles to openness in Unix are the rigid structure of the file system, the tight coupling of different service domains, and the structure of the OS centered around a large, monolithic kernel.

2.6. Other Conceptual Frameworks

Various concepts introduced in the context of programming languages are attractive to support full openness.

Parnas has introduced techniques to modularize the software base of a system [101, 102]. A module can be thought of as an abstract service. Although the focus of his work is on the structure

™Unix is a trademark of Bell Laboratories.

of programs rather than on mechanisms to open a system, decomposition of a system is a necessary step to openness.

Hoare and Brinch Hansen have introduced monitors [33, 65] as the basic blocks for structuring operating systems. A monitor is an extension of a module that provides implicit synchronization. Brinch Hansen's Distributed Processes language [34] extends the notion of monitors to support active or passive monitors. A monitor may have one or more processes, running occasionally to perform housekeeping chores. A monitor is activated by other processes through procedure calls. A monitor can be conceptualized as a server in an open system, activated by customers to perform services.

Kieburtz and Silberschatz [77, 78] have introduced the notion of capability managers which help maintain capabilities outside the protected domain of the OS kernel. This property remedies some of the overhead encountered in traditional capability-based systems. The linguistic support they propose simplifies the construction of resource managers and the verification of resource access.

2.7. Conclusions

We have discussed many concepts which lay a good foundation for a fully open system. Capabilities facilitate a unified and protected approach to select resources and services. An object-based, server--customer-based, or a modular view of a system enables decomposition of functionality as well as encapsulation of resources. Policy–mechanism separation and decomposition of a system to layers encourage distribution of OS functions among different components. A unified view of devices or interfaces, together with the support of application-level device drivers, can accommodate private or customized physical resources.

However, as the many systems surveyed here illustrate, each concept bears potential perils to full openness. In particular, supporting a concept with protection barriers that are too high, or with a communication facility that is too slow, impedes openness of many OS functions. The following chapters exhibit how these concepts, and the lessons we learned about their drawbacks, have shaped our model and design of a fully open computing system.

Chapter 3

The Model

3.1. Introduction

In order to investigate the problems and possibilities of full openness, we need a system with which to experiment. The process of defining such a system is part of the experimentation, since during this process the problems are studied and potential solutions are evaluated. In defining a system, one has to deal separately with the questions of functionality and performance. First, one has to examine the questions of whether and how the required functionality can be achieved. Then one can study the questions of how efficient and complex the resulting system would be. In defining a fully open computing system, the first question is thus how to construct a system that provides the functionality of a conventional OS without sacrificing openness. A model of such a system is needed to examine this question. Developing a model exposes obstacles to full openness due to protection requirements or due to dependencies between service domains. Experimentation with the abstract system defined by a model allows us to evaluate potential ways to overcome these obstacles. A model provides a base to derive a specific, more concrete design, which in turn serves as a framework to study the aspects of efficiency and complexity. We discuss each of these issues in turn.

A model defines system components, their interrelationships, and the principles upon which resources and services can be added or selected. This definition lays the groundwork for deciding the system structure and the rules of interaction between its components at the design level. For example, a designer has to define rules for inter-application communication in order to allow applications to provide services to each other. These rules should specify the methods for naming, typing, addressing, and for inheritance of names, types, and invocation rights. Likewise, a design has to state how service providers and customers are protected from each other, and to address the subjects of buffering and synchronization. The difficulty in coping with these issues in an open system is that openness advocates conflicting objectives. On the one hand, one wants ***uniformity***, so that services can be invoked by different applications. On the other hand, one wants to ***minimize*** the set of mandatory services or features, so that applications can select the services they need without suffering excessive overhead or coerced semantics. A model defines principles to resolve this conflict at the abstract level, and to guide the definition of more concrete rules at the design level.

An important role of a model is to help us investigate how protection requirements constrain openness. Protection has a dual purpose. First, it provides a resource allocator the necessary means to implement its allocation policy—in particular, to enforce usage restrictions and to revoke previous allocations. Second, it assures a service provider or a resource owner that the service/resource cannot be used by unauthorized entities. A model is needed to identify a minimal set of mandatory services that accommodate protection requirements. In essence, the model establishes a lower bound on openness due to protection requirements; it qualifies *full* openness as limited only by those protection barriers necessary to accommodate higher-level, customized protection mechanisms.

The modeling process explores the requirements for openness and protection in different service domains and the interaction between those domains. For instance, the developer of a model examines how protected openness of CPU scheduling and memory management can be supported, and in what aspects these domains interact. The main contribution of this phase is to expose dependencies between service domains that restrict the ability of users to select services. The subject of controlling critical sections demonstrates this role. Consider the case where part of a service is a critical section. Since such a section must be executed *urgently* or *atomically*, the service provider desires that a computation entering the critical section completes it uninterrupted. One solution would be to disable CPU preemption during the critical section. However, not every service provider can be entrusted by the system to do so. Likewise, the scheduler of a computation that invokes this service cannot be permitted this right, since in a fully open system *any* application may include a private scheduler. In either case, allowing applications to disable CPU preemption may render the system chaotic. This example highlights the following four dependencies:

(1) The dependency between a service provider and the schedulers of computations which invoke its services. If the provider does the scheduling, then how is it told the precedence of computations, and how can it satisfy all precedence requirements? Otherwise, how does the provider tell different schedulers to increase the priority of a computation occupying a critical section, or to block one awaiting a critical section?

(2) The dependency between unrelated computations. A computation might be delayed because another computation occupies a mutually-exclusive resource. If the scheduler of the second computation does not schedule it frequently to release the resource, then how can the service provider schedule the computation or preempt the resource?

(3) The dependency between CPU scheduling and memory management. If a service is required by an urgent computation, then how can the provider guarantee that page faults do not occur during the service?

(4) The dependency between the service provider and whoever maintains a computation's state. If a computation is "private" to an application or to its scheduler, but invokes services of other applications, then where is the computation's state maintained? In particular, how can to guarantee that a computation cannot "jump into" the middle of a critical section, causing chaos to the service provider and its customers?

These problems were exposed by our experimentation with defining the model. The solutions adopted were generalized to define principles that apply to other service domains, such as buffer management and recovery.

3.2. Overview

Central to the model is the assumption that all computing requirements can be captured by two abstractions: a service and a resource. A **service** is an abstraction that uses **resources** to carry out a logical function. The model is based on the concepts of *resource ownership* and *service provision*. The computing system is modeled as a collection of **servers** that *own* resources and *provide* services. To provide a service, a server has to own the required resources or gain access to them via other servers. Full openness is achieved by enabling each server to dynamically select the services and resources it needs.

A computation is a sequence of services represented by an activity. An **activity** is a thread of control that executes services. Execution means materialization of a service by using a distinguished type of resource called a **processor**. A server is stored in a distinguished type of resource called **memory**. Services are invoked via bindings. A **binding** is a reference to a service; it is ***held*** by a server or an activity. Through services, a server obtains ownership of resources and accesses them. A server may access the resources it owns, transfer ownership of portions of them to other servers, and later reclaim them. A server can give **permits** (with specific access rights) to other servers to access the resources it owns.

In order to accommodate full openness, the model leaves the semantics of using resources and services to the mutual understanding of each provider and customer. However, as necessitated by protection requirements, a few restrictions apply to resource ownership and service provision. First, resources which all servers may share are controlled by generally-trusted servers. Likewise, the services required by servers to protect their own resources and services are also provided by generally-trusted servers. These servers constitute the **O**perating **S**ystem **B**ase (**OSB**). Second, a few conventions apply to the interfaces of services which a server might be obliged to invoke without knowing their invocation protocols. The conventions, called the **standard interface**, name such services and specify standard invocation protocols.

The OSB is the base above which customized services are defined. Resources and services not included in the OSB are open to any application. An **application** is defined as any collection of non-OSB servers. **Users** are represented in the system by servers and activities.

The physical environment assumed by the model is a multiprocessor, shared memory environment. The computing system includes one or more tightly-coupled and architecturally identical CPU's. There are one or more memory subsystems, each of which defines a physical address space. All memories conform to the same service interface. (A glossary of the terms introduced in this chapters appears at the end of the dissertation.)

3.3. Servers, Resources and Services

A **service** is an abstraction of a set of actions that carries out a logical function. Its functionality and its invocation protocol, such as the number and types of input arguments and of outputs, are decided by the service provider. A service can be performed synchronously or asynchronously with its invocation. When executed asynchronously, it may return multiple outputs.

Servers are self-contained entities that communicate via service invocations and shared resources. A server can be viewed as a dynamic representation of a program (composed of executable algorithms and data structures) that implements services. Any server can create a new server by acquiring memory space and placing the new server there. A creator does not need permission from the OSB or any other server to create a new server, nor does it have to register the new server. It is up to the creator of a server to decide which of its bindings, permits, and resource ownerships the new server inherits.

A **resource** is a physical component—such as a CPU, disk space, or communication bandwidth—or a logical entity—such as a semaphore, file, or virtual disk. A resource is encapsulated in one server, called its **host**. The host defines the semantics of accessing the resource. A resource is composed of units defined by its host. Each unit can be **owned** by several servers concurrently. The default owner of the entire resource is its host, which can grant ownership to other servers. Through the host's services, an owner can access the resource and transfer ownership of units it owns to another server. A transfer of ownership is called **allocation**. Depending on the resource, access and allocation of a unit can be exclusively granted to only one owner of it. For instance, access and allocation of a unit of physical memory would conceivably be granted to its *current* owner, which is the last server which was allocated that unit. However, any *former* owner of a unit may revoke it. **Revocation** of a unit U from an owner O means that O looses its ownership of U, and that the revoking server becomes the unit's current owner. It also implies that U is revoked from any server to whom U has been allocated by O. It should be noted that an owner of a resource can provide both the allocation policy of the resource and the mechanisms to use it. When an owner dies, ownership of its resources is returned by their hosts to the servers that have allocated these resources to the deceased server. A host may allow an owner to name other default inheritors. An owner can give permits to other servers to access its share of the resource. A **permit** references resource units and lists rights for accessing them.

Figure 3-1 illustrates the relationship between servers, resources, and services. It shows that multiple hierarchies of services and resources can be formed. The physical resources comprise the lowest level of the resource hierarchy. A logical resource is mapped by its host to several resources which the host owns or is permitted to access. The mapping of a resource or a service to other resources or services can be transparent to the customers of the resource/service. Figure 3-2 further illustrates the notion of resource ownership and access. Server R, for instance, accesses resource P through the host of P, not through Q, which allocated the resource to R.

A server can charge for resource and service usage. It can charge an activity directly while the activity executes its service, or charge another server that has furnished a permit for this operation. Charging can be used to reduce contention for scarce resources, as well as to account for resource usage.

The model view of naming and typing of servers, services, and resources is simplistic, which is the result of our desire to minimize the imposition of default semantics and overhead. Servers, services, and resources are identified by unique names. Types of services and resources have application-specific semantics in practice, hence there is no support in the model for formal definition and checking of types.[1] Hosts and service providers attach names to their resources and services as they like. The names are used in announcing a resource/service either statically, e.g. in user manuals, or dynamically through intermediary servers. We denote such servers as

[1]For the sake of convenience, we refer to various services or servers by their assumed functionality. For instance, a *disk server* is one that provides access to a disk. However, we do not imply that this functionality is rigorously defined by the model, nor that services named identically—such as two disk access services—are functionally identical.

matchmakers. A matchmaker allows servers to deposit bindings or permits, to locate required resources and services, and to obtain the necessary bindings. Servers can acquire bindings and permits both statically, e.g. at load time, and dynamically, e.g. as a result returned by another service.

There are two exceptions to this principle of naming and typing. First, two resource types are distinguished from other resources. A **processor** is the only resource strictly required to materialize every service. A **memory** is a resource necessary to store servers. Different processors and memories may exist in the system. We further distinguish between (1) special-purpose, non-shared processors, and (2) general-purpose, shared processors called **CPU**'s. For the sake of simplicity, the model deals only with architecturally identical CPU's. A host of a physical resource (denoted for brevity a *p-host*) may embed a special-purpose processor that is dedicated to perform access to the resource. For instance, a dedicated processor in a disk host performs disk accesses independently of any CPU.

Second, naming and typing conventions are defined for the interfaces of services which a server may be *obliged* to invoke without knowing their invocation protocols. Such a situation occurs when a server must invoke a *given* service to carry out a required function for a *given* activity or resource, without being able to (1) choose the service, or (2) learn the service's interface in advance. These conventions, called the **standard interface**, identify such functions. They specify

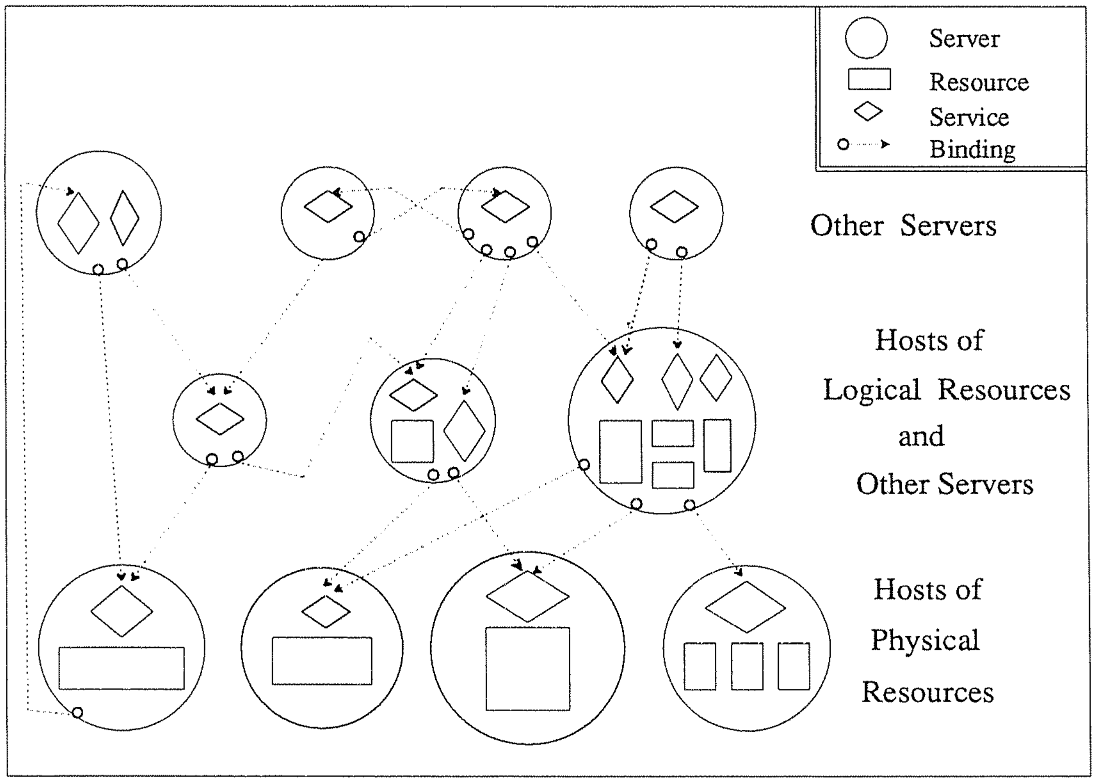
Server
Resource
Service
Binding
Other Servers
Hosts of Logical Resources and Other Servers
Hosts of Physical Resources

Figure 3-1: Servers, Resources, and Services

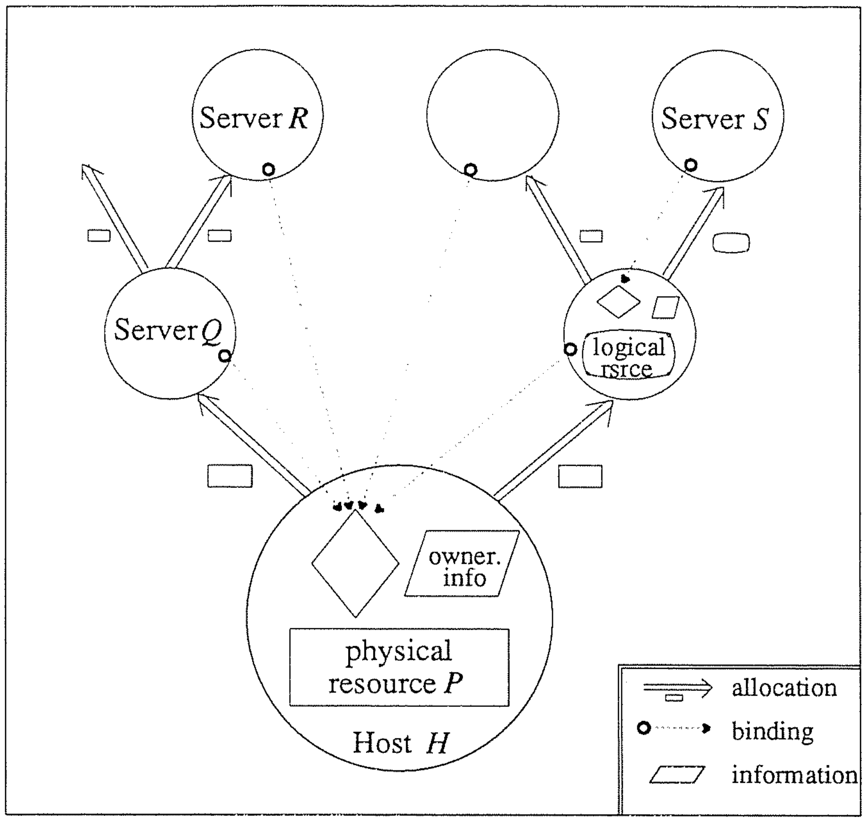

Figure 3-2: Resource Ownership

Allocation of a resource is illustrated as transferring units of it. To distinguish a logical resource from a physical one, they are depicted differently, although any difference can be hidden from the user of the resource, e.g. from Server *S*.

for each function how any service that carries it out should be named, the service's invocation protocol, and the way its binding is obtained. To clarify why a standard interface is needed, consider the following two examples. A server may have a permit for a customer's buffer which it wants to "pin" in core. The *pinning* service needs a standard interface because the service of a specific, possibly different, memory server must be invoked for each buffer. In contrast, *writing* a file does not necessarily require a standard interface, since the service can be performed by the server itself or be selected from some file server in advance.

The model does not define a particular standard interface. This task is relegated to the system designer.[2] The model's view is that the standard interface is minimal in the sense that it includes

[2]The standard interface is defined statically, for two reasons. First, if a server were to dynamically learn the invocation protocol of a service in the standard interface, then the system designer would still have to *statically* define the rules for specifying invocation protocols. Second, dynamic learning of protocols would be complex or inefficient.

only those services for which the above two conditions cannot be circumvented; the standard interface is not imposed, but rather suggested to any server that wants to provide such a service.

3.4. Activity Management

An **activity** represents a computation — an independently scheduled entity that consumes processor time to execute services. It is a thread of control that starts executing one service, and through service invocations can span multiple servers. At each invocation the activity starts to execute the invoked service. The invoking service is suspended until the invoked one returns. The sequence of invoked services which have not yet returned is denoted the activity's **dynamic chain**, or simply **chain**. (We interchangeably refer to such services and the servers that provide them as being in the activity's chain.) The last service (server) in a chain is denoted the **current service** (current server) of the activity.

Multiple activities can run concurrently at one server. They all share its data structures, resources, permits, and bindings. It is the server's responsibility to synchronize them. They can communicate through the server's data structures and resources. Figure 3-3 shows two activities, one currently blocked at Server *R* and the other running at Server *Q*. Activity *A*'s chain, for instance, is (*S*, *Q*, ..., *R*), and its current server is *R*. The major difference between an activity and the common abstractions *process* and *thread* supported by other systems [109, 135] is that an activity can run at multiple address spaces.

The model has a dual view of the activity–server relationship. On the one hand, a server is viewed as passive, letting any of its services be executed by the activity that invokes the service. In this view, the server's resources and bindings are used by activities. On the other hand, the activity can also be viewed as passive. It is owned by a server—its creator—and can be transferred to other

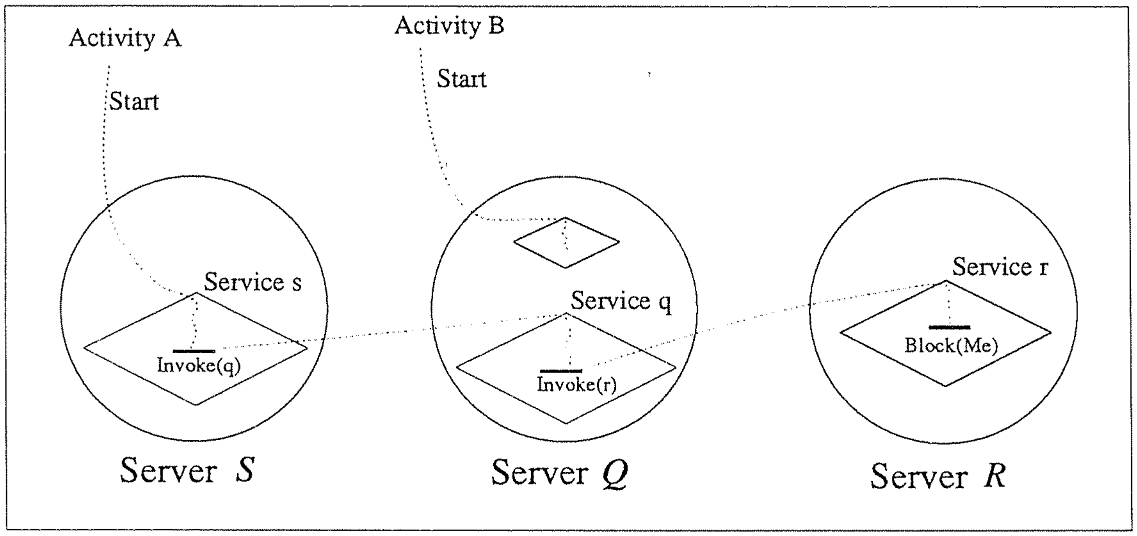

Figure 3-3: Activities — An example

servers. The owner *dispatch*es the activity, that is, lets the activity consume processor time. The owner can block the activity or terminate it. In this view, an activity merely transfers execution state between active servers. A service invocation thus can be viewed as transferring an access permit for the processor to the server whose service is invoked.[3] Moreover, an activity may hold bindings, which can be used by servers in its chain to invoke services on its behalf. A limited scheduling capability of an activity is granted to its current server by letting the server switch between activities running at the server. Such switching is useful to *push* an activity, for example to release a critical section or a scarce resource. For protection reasons, a server is not allowed to switch to an activity for which another server is the current server, because the latter might require that the activity be blocked awaiting a certain event.

In an asynchronous service provision, the service is performed by a different activity than the one that invoked it. The invoked service is not in the chain of the invoking activity. Access services of a p-host that has a private processor are examples of asynchronous services. However, the invocation of a p-host's service is more intricate because it requires transfer of control between two processors. The invocation is initiated on a CPU. Then, an internal mechanism of the p-host is used to schedule an activity to perform the access on its private processor. At access completion, a CPU is required by the p-host to perform various tasks such as telling a waiting customer about the event. Using a similar mechanism, the CPU host schedules a specified or a predefined activity to perform these tasks on a CPU.

The model also recognizes the need for asynchronous service *invocation* in addition to asynchronous service *provision*. In some situations a server may wish to invoke a service asynchronously, because suspending the invoking service could be undesirable. For instance, suppose server S invokes a service of server R, such as an I/O access, which is performed asynchronously; at access completion R invokes a "reply" service s of S to report the event. However, if s cannot be entrusted to return quickly, then R cannot invoke it synchronously because of the risk of suspending further accesses to R's resource. The model does not support asynchronous invocation directly as a "built-in" facility, since such an invocation can be accomplished by other means. For example, R can schedule another activity to invoke s later. Or an asynchronous service invocation can be provided as a *service* which itself is invoked synchronously and which returns after relegating the invocation of s to another activity.

Activities may have to communicate by means other than the data structures and resources of servers in their chains. For example, while activity A runs at the owner of activity B, the owner decides to terminate B. So the owner wants to tell the services in B's chain to properly clean up their state. Similarly, a service executed by activity A discovers an event which implies changing the course of execution of activity B, e.g. aborting the current service of B. For these purposes, activity A can *raise an exception* on B through a service provided by the OSB. As a result, when B resumes execution the exception is noticed by the CPU, which diverts B's control to execute a service predesignated to *handle* such an exception. For protection purposes, raising an exception is allowed

[3]Other permits can be transferred as parameters to the service.

only to B's owner and B's current server; another server R cannot raise an exception directly, even if R is in the activity's chain, since R might not know the current state of B. Notice, however, that R can notify B's owner of the event through the activity's bindings; it may notify other servers in B's chain through ordinary service invocations, if it knows which servers these are. The invocation protocols of exception-handling services are included in the standard interface because raising exceptions by an activity's owner is tantamount to invocations of services whose protocol the owner could not know in advance.

We distinguish between activities using solely a private processor of some p-host and others that use CPU's. The former activities are controlled by the p-host and are invisible to other servers, while the latter may span multiple servers. In the rest of the dissertation we discuss only the latter activities, except where stated otherwise.

Each activity has an associated execution state, which consists of scheduling information and the environments of the services in its chain. The scheduling information is maintained by the activity's owner. The environments are distributed among the servers in the activity's chain—for example, the environments are stored in the servers' local "stack frames." Some of these environments are stored in the CPU's state during execution. When an activity is preempted from a CPU, its state must be preserved until the activity is redispatched. The state is stored in locations accessible to the CPU host and protected from corruption by other servers. These locations are collectively called the Activity's Context Descriptor (**ACD**). Consequently, dispatching and halting an activity are services provided by the CPU host.

3.5. The OSB

3.5.1. The OSB and protection

On the one hand, openness implies that servers should be allowed to select the protection mechanisms that fit their security objectives. On the other hand, protection in a multiuser environment prescribes the following demands: (1) that servers are able to implement desired protection mechanisms, (2) that every server can access resources intended to be shared by all servers, and (3) that a server cannot cause unrelated servers to fail, namely, those that do not use its services and that have not permitted the server to access their resources. To resolve the conflict between these requirements, the FOCS model offers a low-level protection mechanism based on the notions of encapsulation, ownership, and light-weight capabilities. The OSB enforces only those protection measures that facilitate the protection of resources and services. We discuss the above issues in turn.

In order to let a server select its protection mechanism, it is left to the server to decide how to protect its resources and services, to discriminate among customers, to reject invocations, to check service inputs, and (if the server is a host) to verify ownership and access permission to its resource. It is assumed that a service is trusted by its customers to provide the expected function. Hence, a customer is not protected against a faulty service (a misservice), except that the service provider cannot use resources to which it has not been granted access.

To guarantee a server the ability to implement a customized protection mechanism, the OSB provides a minimal set of mandatory services. The set consists of services to verify a binding at service invocation and a return address at service return, to raise and to notice exceptions, to authenticate the id of a server or of an activity, and to maintain accounting. The verification service guarantees proper entry to and exit from services; the exception-related service protects activities, as discussed above; the authentication service allows servers to discriminate among customers; and the accounting service protects the accounts, for which the users are responsible to "pay the bills." Furthermore, the OSB is the default inheritor of a terminating activity and of an activity whose owner has terminated, in order to ensure the servers in the activity's chain the ability to recover their state.

This set of mandatory services excludes any service or restriction that servers can provide for themselves, such as further authentication of invokers. Most notably, the OSB does not maintain bindings or permits, which are viewed as light-weight capabilities. Each holder of a binding or a permit can duplicate and transfer it to other servers as an ordinary data structure. Nonetheless, a binding for a service is created and can be invalidated only by the service provider, and a permit for a resource—by the resource owner.

To guarantee access to the generally shared resources, called **system resources**, their hosts are included in the OSB. The definition of what constitutes the system resources is installation-dependent. The view of the FOCS model is that each such host provides a minimal set of mandatory services, required to access its resource and to control its allocation. These services implement rudimentary mechanisms to carry out access and allocation requests in a protected way, as well as a basic policy to allocate the resource to a few customers. Higher-level mechanisms and allocation policies are relegated to the resource owners. It is noteworthy that the power of the OSB to impose protection restrictions derives from being the hosts of certain resources, which the OSB annexes at system initialization.

Finally, the notions of resource ownership and of encapsulation of resources and services prevent a failing server from inducing failure in If a server crashes, its resources are reclaimed or inherited by former owners, and it cannot corrupt resources to which it has no access.

3.5.2. The OSB and applications

What servers do we consider appropriate for inclusion in the OSB? Foremost, the OSB includes the hosts of an installation-defined list of system resources. This list should include hosts of resources the OSB depends on, such as the host of the memory into which the OSB is loaded, the CPU host, and a clock host. The CPU host encapsulates all the CPU's in the system.[4] It provides services to allocate CPU's, to use them, to create and to terminate activities. A clock host is essential for proper reclamation of CPU's. The OSB includes also an *Accountant* and an *Initiator*. The Accountant provides services to manage accounts, to charge and to credit them. The Initiator is the

[4]The model does not allow additional CPU hosts outside the OSB because of the semantics of service provision and of protection. Since a service (other than an access to a physical resource) is performed by the CPU at which the service is invoked, an untrustworthy CPU host may violate protection requirements or maliciously modify the OSB, thus corrupting the entire system. The reason for a single CPU host is mainly of convenience. It relieves servers from being bound to many CPU hosts, and from negotiating CPU allocation with each host separately.

server that creates the OSB at system-initialization time.

Figure 3-4 depicts the computing system as composed of several collections of servers, one of which is the OSB, the others of which are applications. As mentioned earlier, an **application** is any logical collection of non-OSB servers. Customized OS's can be constructed by applications above the OSB. We imagine that at an installation one or more customized OS's (denoted **COS**) may exist. A COS provides system services, so that not *every* application must complement the minimal services of the OSB with all the services that it needs. For example, a COS may include (see Figure 3-4) (1) A *matchmaker*, (2) a *server manager*, which provides services to create and terminate servers, (3) a memory host and a *virtual store manager*, which provide services to allocate physical memory and to map servers to it, (4) a *user-interface* server that mediates between users and the

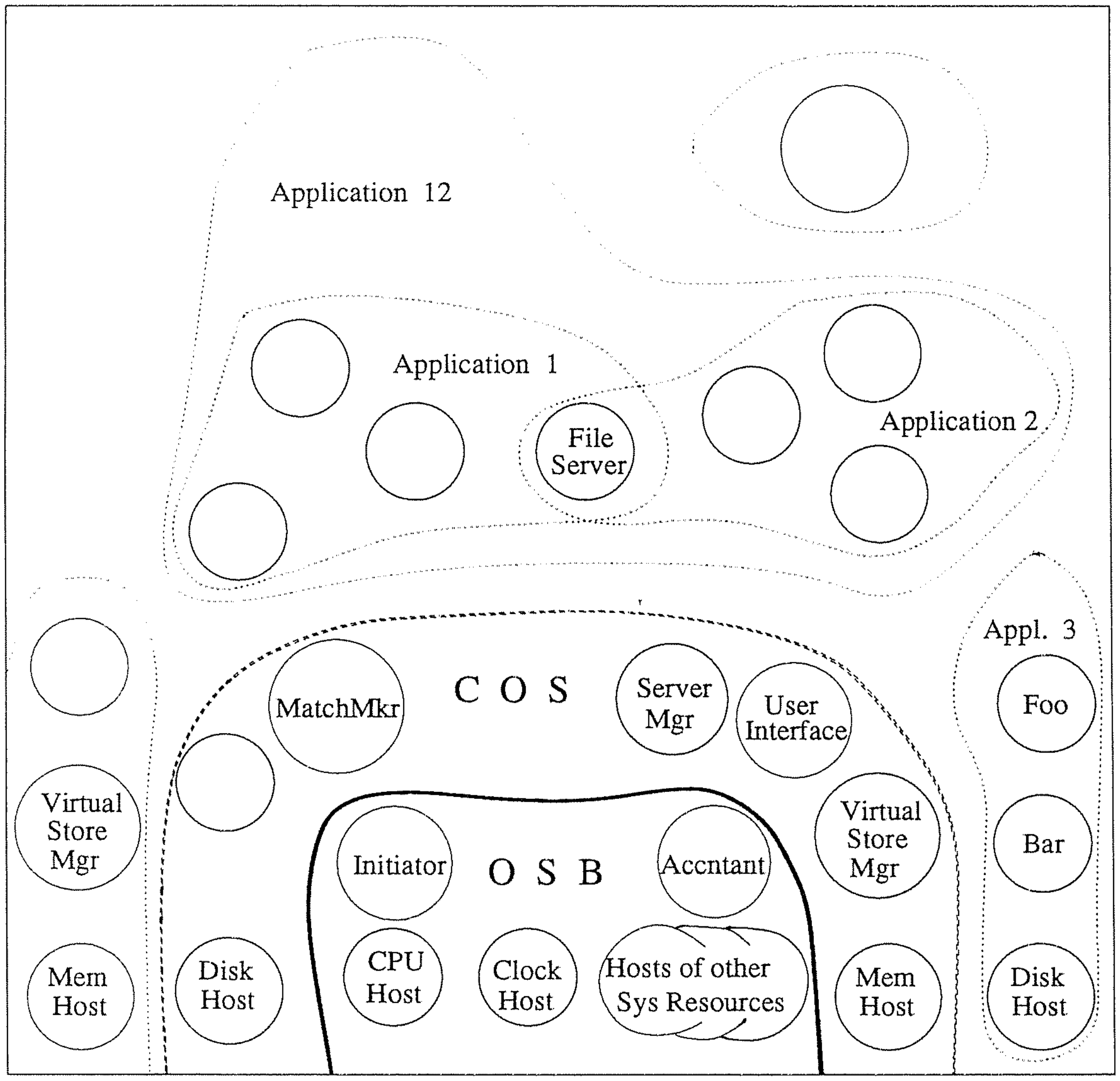

Figure 3-4: OSB and Applications

system, and (5) other hosts, language processors, and editors.

The model assumes that a distinguished user, called the **System Administrator**, is the primordial owner of all accounts, and its servers are the primordial owners of all system resources. Thus, the system administrator can dynamically manipulate allocation policies, ownerships, and accounting information without needing a special interface or a privileged mode.

3.6. Management of Specific Resources

In this section we discuss how the principles and primitives defined by the model apply to specific resources. This discussion is at the model level and does not introduce design decisions. The concepts introduced here, however, guide the design which will be presented in Chapter 4.

Resources are categorized as those that are allocated in units of *time*, and others that are allocated in units of *space*. A time-partitioned resource is owned *entirely* for *T* time units. During this period the owner can access the resource exclusively. Allocation is *consumed*, in the sense that the ownership *expires* once *T* time units have passed. Likewise, the allocator's ownership is reduced by the portion that has been allocated. On revocation, only the unused portion of a previous allocation can be reclaimed. Processors, tape drives, and network-communication channels are examples of resources that exhibit these characteristics.

A space-partitioned resource can be shared and accessed concurrently by multiple owners. Allocation of such a resource does not reduce the allocator's share. Revocation can reclaim a previous allocation entirely. Obviously, an allocator may set a limit on the duration of an allocation; however, in such a case time is used as a decision parameter, not a resource unit. Likewise, the resource's host may restrict sharing, for instance by forbidding a former owner to access the resource unless that owner first revokes the resource. Storage resources such as memories and disks (both physical and virtual) are examples of resources that exhibit these characteristics.

The discussion below focuses on the CPU and memory resources, for which openness sharply confronts protection requirements and efficiency objectives. As shown below, the concepts of resource ownership and service provision allow every application to implement its particular CPU scheduling or memory allocation policies. The aspects of efficiency and complexity in managing these resources will be discussed in the following chapters.

3.6.1. CPU Management

A major difficulty of CPU management is to accommodate CPU scheduling by any server in a protected and efficient way, with simple mechanisms that can support any scheduling policy. Here we show solutions that derive from the model principles. These solutions are elaborated in Chapter 4 with concrete services.

The CPU host encapsulates multiple CPU's. It routinely allocates them to its customers, and provides services to further allocate CPU's and to revoke them. Any server can own a CPU for a *slice* of time units. A p-host may want to own a CPU in order to schedule activities to initiate accesses to its resource, or to inform customers of access completion. A CPU owner is called a **scheduler**.

CPU allocation establishes a stack-like ordering of owners for each CPU. At each allocation, the allocator becomes the *current owner* of the allocated CPU. Control passes to this scheduler to dispatch activities or to further allocate the CPU. Control of the CPU is returned to the allocator when the server allocated a slice either consumes the slice or releases the CPU. Nonetheless, the allocator can revoke the slice before it is consumed, or allocate another slice to another server, who then becomes the CPU's current owner. The allocator can do so either when it is activated by a service invocation or while it is executed concurrently on another CPU. Of course, a scheduler may allocate *virtual* time slices to its customers. It is up to the scheduler to "map" these slices to allocations of real-time slices. Figure 3-5 depicts the CPU host, a dynamic hierarchy of schedulers, and the notion of CPU allocation.

The ordering of CPU ownership is maintained by the CPU host. A primordial ordering is defined at system initialization. It can be dynamically altered by the CPU host. This ordering represents relative *priorities* of servers for obtaining CPU's. When a server urgently needs a CPU, it revokes one from the CPU's later owners via a service called a *CPU interrupt*. An owner can prevent a CPU from being preempted during urgent work by asking the CPU host to increase its priority, namely, to promote its position in the stack of CPU owners.

A scheduler *dispatches* an activity for execution on a given CPU for a specified portion of the scheduler's slice, called a *quantum*. Dispatching is a service of the CPU host, which can be invoked directly by every scheduler. For instance, S_b in Figure 3-5 is allocated CPU's through S_a but invokes the dispatching service directly. We decided that dispatching an activity implies reclaiming the remaining quantum of the activity currently executing on the specified CPU, since quanta are of real time and since a CPU must be used exclusively. Hence, if the latter activity is the dispatching one, it is blocked at the CPU host until the dispatched activity consumes the quantum. Notice, however, that the scheduler may be executed by the latter or by another activity before the quantum expires, and thus it can modify its scheduling decisions "on the fly." Using a CPU is itself accomplished via the access services denoted as *machine instructions*.

Dispatching means granting a CPU permit to the activity's current server, as indicated in Figure 3-5. This permit is conveyed by the activity to other servers through service invocations and returns. Having such a permit, a server may switch execution to another activity as discussed earlier. This switching is performed by the CPU host, whose intervention is needed to preserve the former activity's CPU state and to restore the latter's state.

3.6.2. Memory management

We assume below that servers are stored in virtual memory which during execution must be mapped to physical memory. References in virtual memory must be translated to physical memory addresses. A main difficulty of memory management is to accommodate openness of memory allocation and mapping while ensuring efficient and protected translation and access. We show here a solution based on the model. Specific mechanisms that support this solution will be discussed in Chapters 4 and 5. The structure of physical and virtual memory are presented first, and then the issues of mapping and translation are discussed.

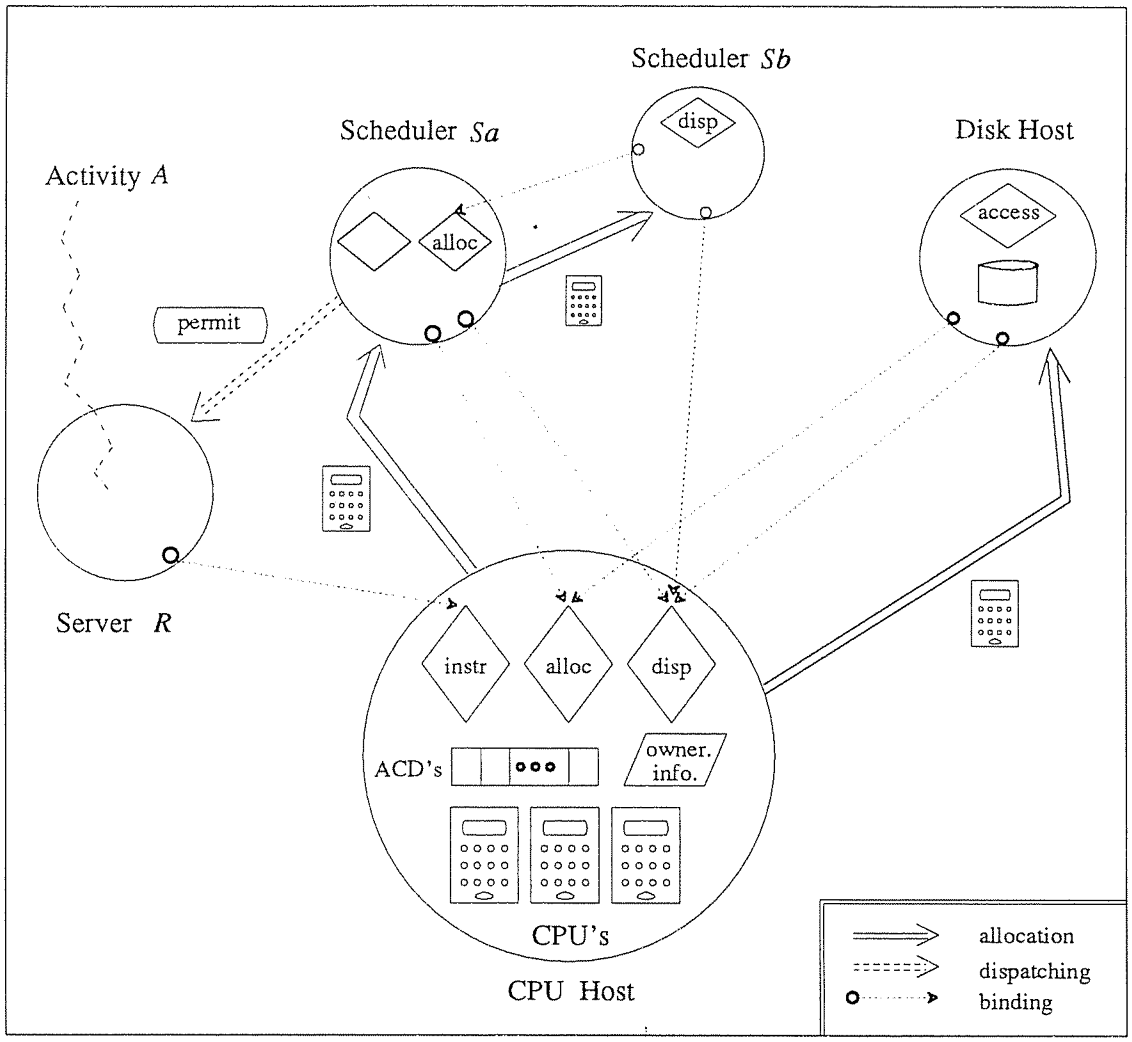

Figure 3-5: CPU Management
CPU allocation is illustrated as transferring smaller CPU's, although the allocation is of time units. Dispatching an activity *A* is illustrated as handing a permit to Server *R* at which *A* currently runs. Every server permitted access to a CPU can directly invoke the CPU host's services, denoted as machine instructions.

The components of physical and virtual memory are the following. The shared physical memory of the computing system consists of one or more physical address spaces. Each such space is encapsulated in a host, denoted **m-host**, and is composed of frames. A **frame** is a contiguous, host-dependent range of physical addresses. The frame is the unit of allocation of physical memory. Other p-hosts may have private memories, which are not discussed here.

The virtual memory of the system is composed of Universes. A **Universe** is a logical resource, managed by a server called the Universe **manager** (denoted U-mgr). A Universe is mapped by its manager into physical memory owned by the latter. A server that wants to directly control the mapping of its virtual space to physical memory, or to provide such a service to other servers, can obtain frames, create a new Universe and map it to these frames.

A Universe is divided into Spaces, which are the units of allocation of the Universe. A **Space** is a reference environment in which a server is stored, as shown in Figure 3-6. A server is by definition an owner of the Space it is stored in. The decomposition of a Universe to Spaces is motivated by protection and efficiency considerations: Once execution switches to a given server, there is no need to check the eligibility to reference code and data structures within its Space. References outside the Space need be validated via permits or bindings. It is assumed, therefore, that at a given time one Space is the ***current Space*** for each CPU, and that a CPU generates virtual addresses within that Space.[5] A Space switch occurs in a protected way at activity dispatching, at service invocation, at service return, and at any copying between Spaces that is not performed directly in physical memory. The owner of a Space may create permits to access regions of the Space, for instance to be used by other servers as buffers of service inputs and outputs. This structure supports sharing of address spaces through permits and through mapping of different Spaces to the same frames. It is left to the design level to accommodate sharing in an efficient way.

A U-mgr allocates a Space to any server that wants to create a new server and store it in the Space. Both servers own the Space and can access it concurrently, but the former server can revoke the Space from the new server. The U-mgr may allow each owner to dynamically increase or reduce the size of its Space. A Space owner can specify access restrictions regarding the Space, e.g. that a given portion of the Space is read only. Such restrictions are useful to avoid access errors or to prevent modification of some structures stored in the Space. The U-mgr verifies that each access complies with such restrictions.

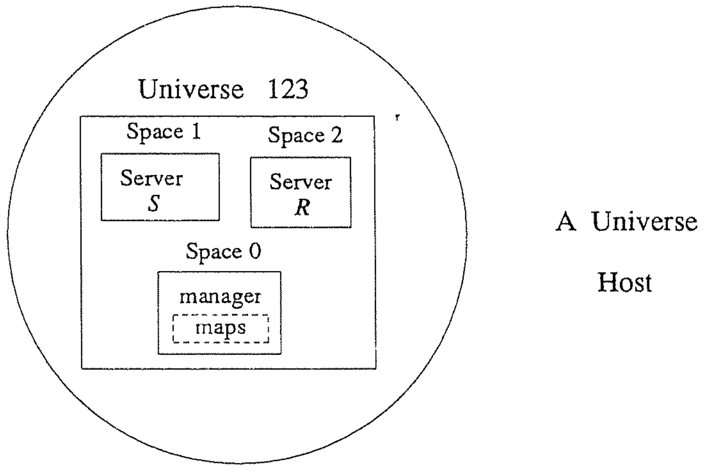

Figure 3-6: A Universe and its Host

[5]Actually, a virtual address is a permit to access a location in a Space, furnished by a server to the CPU host. The CPU host passes the permit to the m-host as a parameter to a memory access service.

We turn now to discuss the mapping and translation between virtual and physical memory (see Figure 3-7). At each memory access a virtual address needs to be translated into a physical address. Conceptually, the virtual address is passed to the U-mgr, which translates it and generates a request to access the resulting physical address. Since the U-mgr is stored in virtual space too, a recursive translation of several virtual addresses might be necessary. Before performing the access, the m-host verifies that each frame accessed during the translation (as well as the frame of the

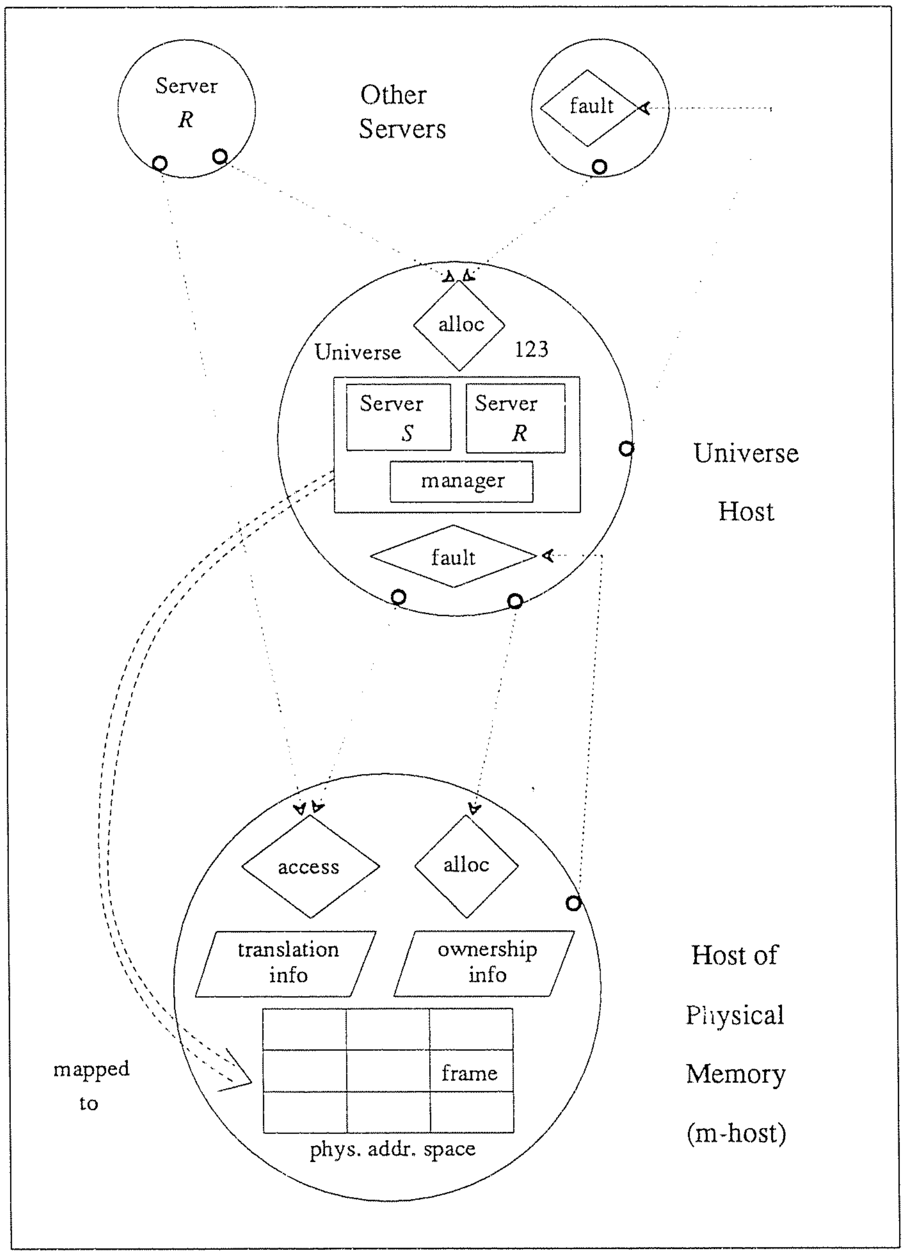

Figure 3-7: Virtual and Physical Memory

resulting physical address) is owned by that U-mgr.

This hierarchical address translation is straightforward but would be prohibitively inefficient. Therefore, a virtual address is passed to the m-host, which performs the translation and the access without involving the U-mgr. The mapping information of a Universe should be therefore visible and accessible to the m-host. This information is stored either at the m-host or at the Universe, depending on the mapping scheme supported by the m-host. In the former case, the m-host allows the U-mgr to manipulate this information directly, for efficiency reasons. In the later case, the mapping information is grouped in an m-host-dependent structure whose location is made known to the m-host by the U-mgr. In either case, each server accesses memory directly as shown in Figure 3-7. The m-host verifies the access as before.

It is left to the design level to make the interfaces for mapping and translation simple and efficient. One possibility is that the U-mgr is stored in one of the Universe's Spaces, as indicated in Figures 3-6 and 3-7. Consequently, the Universe's structural and mapping information are local data structures of the manager; they can be directly and efficiently manipulated by the manager. (Hence, a Universe is *autonomous* and *self-contained.*) Moreover, the U-mgr can control its own mapping to physical memory, and so it can fix volatile structures in core.

Upon a mapping fault or an access violation, the m-host either rejects the access service with an appropriate error indication, or invokes a designated service of the U-mgr (labeled as the "fault" service in Figure 3-7). A binding for this service is passed to the m-host by the manager at Universe creation. This service may further invoke a fault-handling service of the server that caused the fault, if such a service is known to the U-mgr.

3.7. Examples

This section illustrates how the model's primitives and principles lend themselves to support the coexistence of different applications and the sharing of various resources. It differs from the former section in that it presents specific examples, some of which include design-level solutions. An additional purpose of this section is to emphasize various aspects of the model and thus support the rationale discussed in §3.8. We discuss here a hypothetical fully open computing system, in which a customized OS (COS) and three major applications coexist: a Unix-like system, a VM-like system, and an extendible DBMS called ED (see Figure 3-8). The following examples focus on the inter-application interfaces.

3.7.1. A Shared Memory Subsystem

In order to directly control its memory mapping and replacement, the OSB, the COS and each of the three applications runs in its own Universe, as shown in Figure 3-9. There is a single memory host in the OSB, whose entire physical memory is statically divided between the COS and the OSB. The U-mgr of the COS allocates its share of the physical memory to its customers — the U-mgrs of the Unix-like and the VM-like systems. It also maps the servers of the COS to physical memory. The U-mgr of the ED DBMS is allocated frames through the Unix-like system's U-mgr.

The allocation policy of the U-mgr of the COS is the following. It allocates a portion of the memory on a static basis. Each application is allocated several memory regions, which are intended

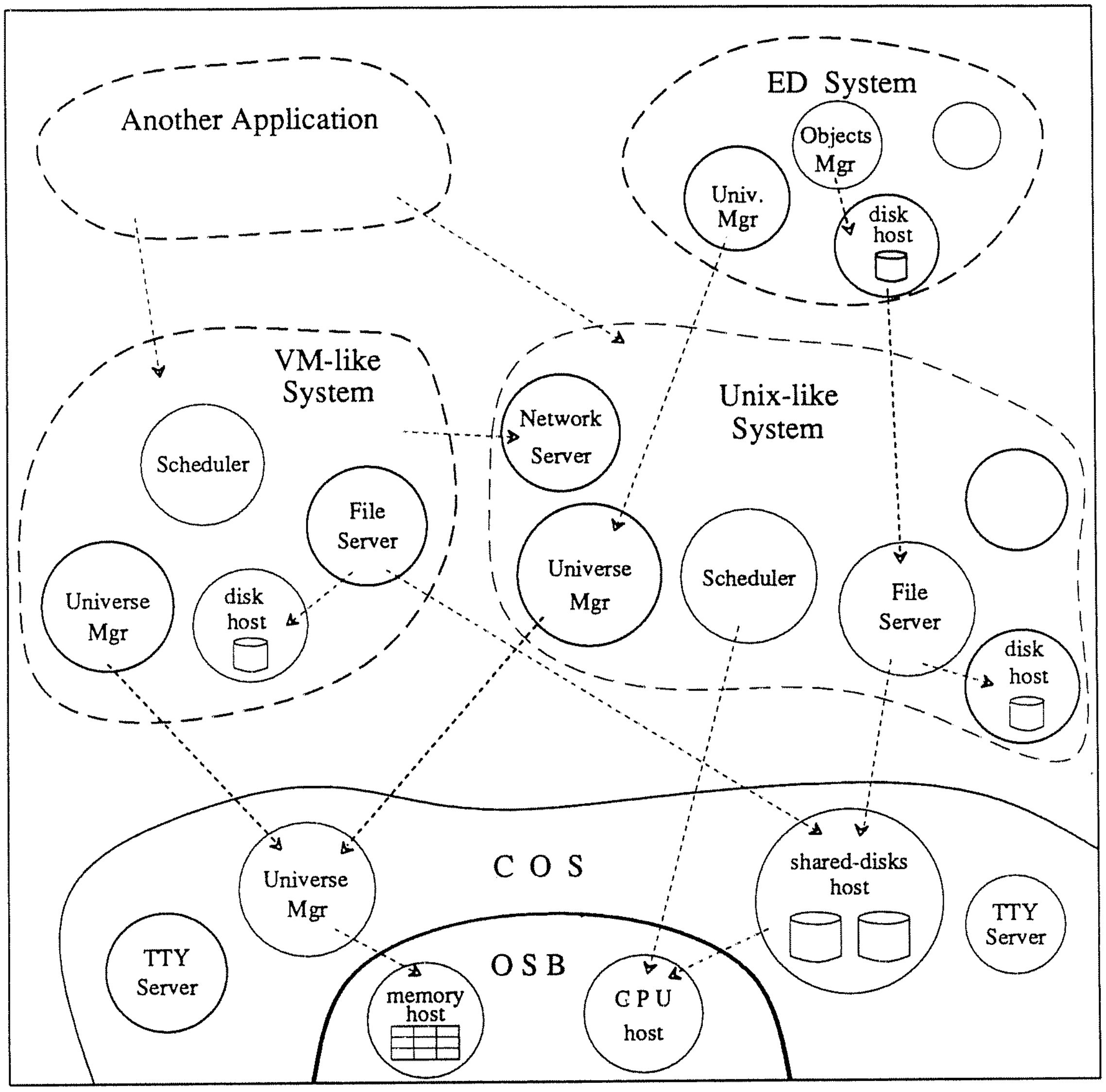

Figure 3-8: A fully open computing system

to support the application's normal demand (or a fraction of that if the contention for memory between the applications is high). Additional regions are allocated dynamically on a short-term basis, to accommodate fluctuations in the applications' demands.

The allocation strategies in such an environment can differ in sophistication. They can be based on different pricing schemes, bidding for memory, or trading memory for other resources. Examples of memory-allocation algorithms in a market-oriented FOCS are discussed elsewhere [51].

The U-mgr of the COS uses three methods to revoke frames from a customer *U*. Foremost, it extracts the required number of frames from a free-frame list of *U*. It uses a permit given by *U* to access this structure. If the list is empty, then the U-mgr invokes a service of *U* to release the frames. If the service fails to release the required number of frames, then the U-mgr—being the

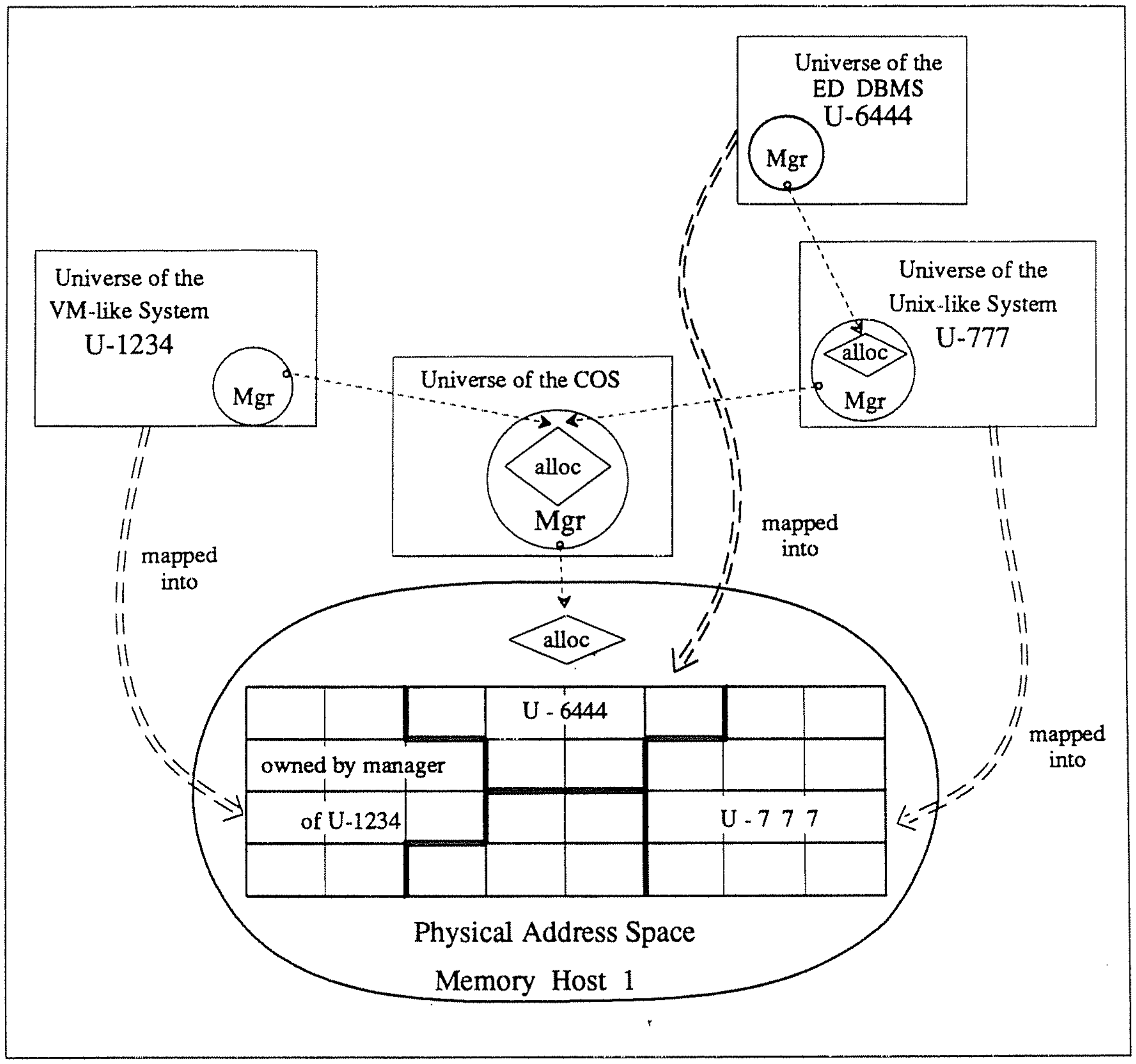

Figure 3-9: A shared memory subsystem

former owner of the frames—enforces the revocation.

3.7.2. Private and Shared Disks

Each of the three applications includes a p-host of a private disk, as shown in Figure 3-8. Thus, the application completely controls the organization of its disk and the scheduling of accesses to it. The host of a private disk is visible only to the application's servers and to the CPU host — as required for CPU allocation.

A host of the shared disks is included in the COS. The shared-disk space is used by the three systems for temporary files, or to support a sudden growth in demand for disk space. It is also used by applications constructed above the Unix-like and the VM-like systems which want to customize their file services. The host of the shared disks divides the disk space into *blocks*, which are the units of allocation. It provides low-level access services to physical locations, and is oblivious of the logical structures mapped by disk owners to these locations. It does not provide buffering — it reads and writes into each customer's buffers through access permits.

The shared-disks host employs the following simple allocation policy. A given portion of the disk space is partitioned among its customers for long-term usage. In a friendly, non-competitive environment this partitioning is based on customers' demands for disk space. In a competitive environment, the demands are adjusted according to criteria set by an administrative authority. The other portion of the disk space is reserved for temporary use. The host employs a pricing function which penalizes customers for holding disk blocks for long periods, pressing them to release unused blocks.

Multiple file systems are constructed above private and shared disks, as shown in Figure 3-8. Each File Server (FS) defines the rules for sharing files and physical locations by its customers. A file created by a customer of an FS becomes a resource owned by that customer. Accesses to the file and its allocation to other servers are performed through the FS. To reduce access overhead, the Unix-like FS allows a customer to obtain a permit for the physical locations of a specific file, then access that file in "raw" mode, that is, directly through the disk host. The VM-like system defines *virtual disks*, maps them to its share of disk space, and allocates them to its customers. The ED system uses the services of the Unix-like FS to maintain backups and the image files of its servers. It uses its own FS and disk host to maintain its database.

3.7.3. An Interserver Communication Server

There is a server that provides services for interserver communication (**ISC**). Its purpose is to help servers in different applications to communicate and synchronize when they cannot do so by invoking each other's services directly. This ISC server accommodates two mechanisms: (1) message-based communication over logical channels, and (2) event-based synchronization.

In the first mechanism, customers acquire and share logical channels, which are the units of allocation. A channel can be concurrently owned and accessed by multiple servers. An owner of a channel may transfer its ownership to another server. Suppose the communication primitives allow for synchronous or asynchronous *Send*, *Receive*, *Wait*—which blocks until a specified *Send-Receive* transaction completes, and a nonblocking *Inspect* to check the status of such a transaction. A customer invoking the *Send* or the *Receive* services supplies an access permit to a buffer in some Space. If the customer elects to *Wait* until the *Send/Receive* completes, the invoking activity is blocked through the owner of this activity. When a *Send–Receive* pair is matched, the ISC server copies the data from the sender's buffer to the receiver's buffer directly, that is, through their common U-mgr or otherwise through the memory host.

In the second mechanism, the ISC server provides synchronization services. A *Notify* service is used to indicate that a given event happened. A *WaitEvent* service blocks the invoking activity until all or any of a specified list of events occurs. When the awaited events are notified, the ISC server requests the owner of a waiting activity to unblock the activity. Note that the customer need not own any resource, nor present any permit.

3.7.4. Concurrent Interaction with Multiple Systems

Multiple terminal servers that control shared terminals in the computing system are included in the COS. A user logging in at a terminal interacts first with a terminal server who owns the terminal. This server is also a window manager. It facilitates creating windows, which are the units of allocation of the physical resource (the terminal). The user may choose to interact with a different user-interface (UI) server at each window, as shown in Figure 3-10. That UI server becomes the owner of the window until the user logs off or disconnects from that UI server. The allocation policy used by the host is simple — let the user decide the allocation. The terminal server provides "raw" *Read* and *Write* services from/to the window. These services are permitted to the current owner of the window, or to whoever the current owner has issued an appropriate permit.

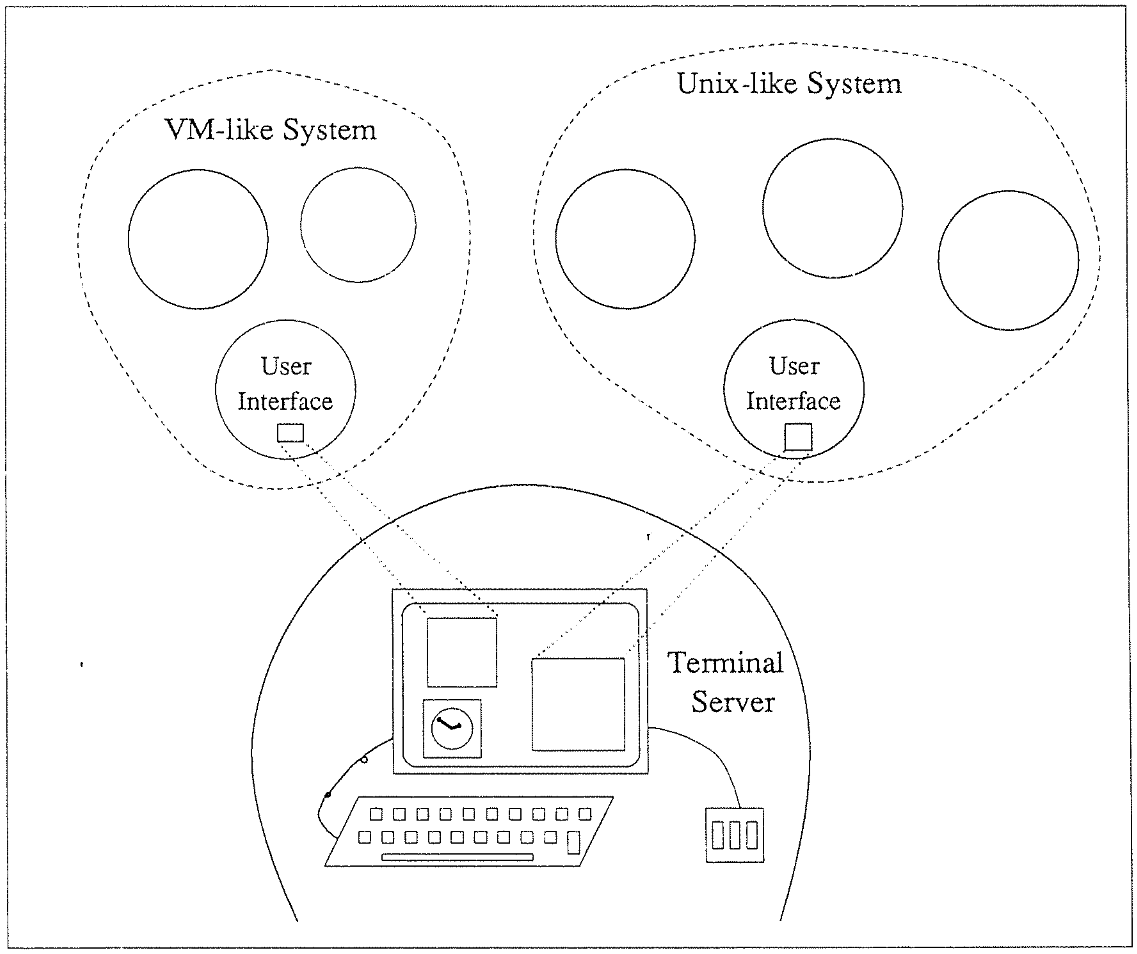

Figure 3-10: Interaction with Multiple Timesharing Systems

3.7.5. ED: An Extendible Database Management System

In this example we elaborate on the needs of an application-specific DBMS and show how those needs can be met in a FOCS. ED exemplifies the new class of *extendible* DBMS's [20]. To further illustrate the advantages gained in a fully open system, we assume that ED supports an "object-oriented" database, organized as a collection of typed objects. Actually, ED is modeled after two such systems, Exodus [36] and POSTGRES [128].

Of major importance to any DBMS is the ability to control: (1) The organization of the database on disk, so that accesses are efficient and the database can be reorganized as a result of growth or decline, (2) the scheduling of disk accesses, especially due to data dependencies as well as during recovery, (3) the management of buffers: in particular to decide memory and buffer replacement policies [94, 43]. These issues are more acute for an extendible DBMS, since its data types, structures, and access methods are application-dependent and may change, perhaps even dynamically [60, 90]. These issues become even more important when the database is extendible and object oriented, since more sophisticated algorithms for disk and buffer management are required. Specifically, we assume that in addition to the above requirements ED must be able (1) to dictate the placement of objects on disk, e.g. their page alignment and clustering, so that data and objects can be efficiently inserted into or deleted from an object, (2) to support sharing of disk blocks among object versions, (3) to support different buffer sizes, (4) to allocate buffers per object stored on disk rather than per disk block as do the general-purpose buffering facilities of operating systems, and (5) to ensure buffer alignment on a page boundary and buffer continuity in physical memory, in order to avoid multiple page faults on buffer access. Finally, as any DBMS may require, ED also needs (1) to control the scheduling of queries of high priority, (2) to implement a particular security policy, and (3) to be able to efficiently share buffers with its customers.

A configuration of ED that accommodates these requirements is depicted in Figure 3-11. The managers shown are layers of abstraction, not necessarily separate servers. As mentioned earlier, ED runs in its own Universe and controls memory mapping and replacement for its servers. In fact, the U-mgr of ED may elect to map several servers to a single Space, in order to reduce communication overhead. ED implements its own buffering mechanism. The U-mgr maps the buffer pool into a segment of virtual space, which is shared by the Spaces of ED servers. ED servers share buffers with customers through permits.

ED includes a private disk to maintain its database of objects. The disk host provides blocking and nonblocking disk access services to ED servers. It accepts access-scheduling directives from other ED components. The Objects & Types Manager is responsible for defining new objects and types, for access method selection for queries, and for other aspects of query processing. Other ED components are responsible for policies in other domains, in which ED has specific requirements: recovery, accounting, security, and transaction management—including concurrency control.

As illustrated in Figure 3-11, ED may acquire services from "outside" servers. The shared-disks host is used to retrieve load modules and to store backups. An outsider server schedules activities to perform urgent queries, recovery operations, and maintenance chores. This scheduler accepts directives from the Transaction Manager of ED.

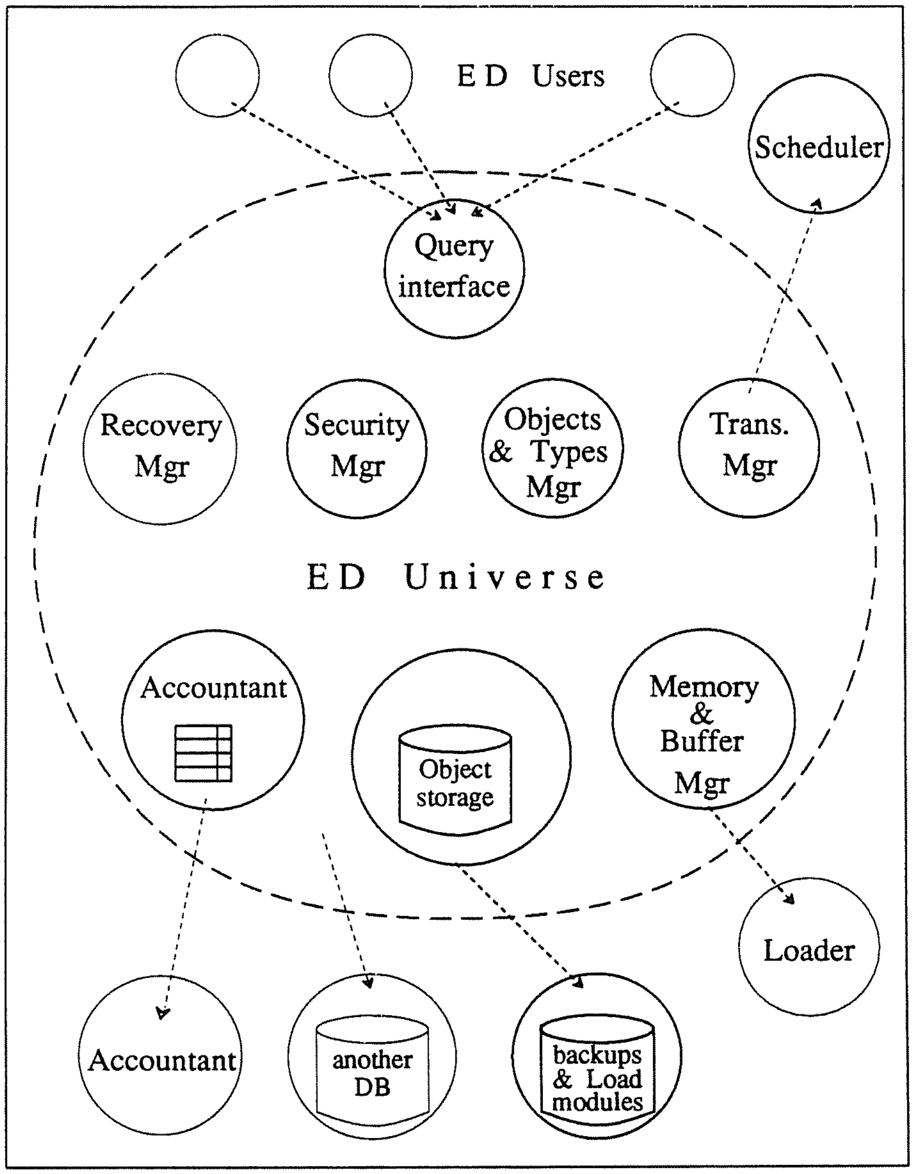

Figure 3-11: The ED DBMS

3.8. Rationale

The preceding sections described the model and illustrated its usefulness. This section justifies our selection of model features and semantics in light of the examples above and other potential applications. Some of the features of the FOCS model are unique; others borrow or combine features of systems described earlier.

Resource Ownership

Resource ownership enables applications to access physical resources directly, to control mapping of logical resources to other resources, and to transfer these abilities to customers. Other models do not support direct control of CPU scheduling or memory management by ordinary applications. The model's view of resource management supports policy–mechanism separation. In addition, it allows an application to supplement the mechanisms for using a resource, for instance when the existing mechanisms do not support the application's policy. Encapsulation of resources within their hosts enables every application to define private resources and protect them. Resource ownership, however, has a potential drawback: a resource is subdivided among applications, so it might be underutilized by one application while being overutilized by another one. The charging system, as well as other mechanisms at the design level that pressure owners to release unused resources, aim to cope with this problem.

It is noteworthy that our approach to full openness based on resource ownership has been recognized by other researchers [92] as an essential foundation for future systems based on marketing concepts.

Service Provision

The view of services as being encapsulated within servers supports hiding implementation details, as do concepts of other models such as modules, monitors, and objects. The unified view of services in the FOCS model extends this hiding to any service. The simple semantics of service invocation allows multiple models of communication to coexist. We chose synchronous service invocation over asynchronous or time-bounded service invocations, because the former is simpler and potentially more efficient. The rejected paradigms can be accomplished via scheduling services. Moreover, as other researchers have observed [27], programmers prefer to use synchronous communication primitives over asynchronous ones.

Service invocation is modeled to combine the benefits of two paradigms: local and remote procedure calls. With adequate optimization techniques at the design and implementation levels, a service invocation can be nearly as efficient as a procedure call. A service invocation is similar to a remote procedure call [25] or a remote invocation [28, 114], and allows simple cross-address-space communication and different modes of synchronization. However, service invocation avoids the time-consuming chores of parameter packing, unpacking, and message passing.

Activities

The decision that an activity spans multiple servers stems from the need to convey the precedence and the execution state of a computation to the services it uses. Otherwise, scheduling of the activity would be left solely to the discretion of each server in the activity's chain. This decision is based also on efficiency considerations: there is no need to create a new activity to carry out the service at each invocation. Otherwise, frequent communication among servers implementing different service layers [44, 121] could be unbearably costly.

An activity is analogous to a *light-weight process* in other systems [12, 81]. It is designed to combine the benefits of two approaches to light-weight processes. Like Mach threads, activities can be dispatched concurrently and hence applications can achieve a higher degree of parallelism. Like Mesa processes, a server has control over the activities that run at the server because it can switch among them. Moreover, the OSB associates only minimal state information with an activity. Thus, activities can be created and destroyed cheaply. Activities are appropriate for different levels of granularity of execution as required by real-time applications [111]. Each application can associate more state with its activities, and thus give them "heavier" weight.

Customized Protection and Security

The level of protection supported by default allows an application of mutually-trusted servers to communicate without protection overhead. The imposed verification of bindings is necessary, regardless of trust, in order to destroy obsolete bindings. This verification is similar to detecting dangling pointers in programs. Checking invocation parameters is not imposed. It can be easily achieved by a server's runtime package. The low-level protection mechanism fits systems that implement different security policies. For instance, DBMS's want the OS to accommodate orderly invocation of their services; once a customer "enters" the DBMS, the latter wants to use its own protection mechanisms to achieve more sophisticated and less inefficient security policies [94]. Higher-level protection mechanisms can be accomplished through service parameters (e.g. a password) or private capabilities. Further research is needed to determine whether a specific architecture can support such capabilities, so that their management is not entirely—and inefficiently—performed by software.

Bindings and Permits as Light-Weight Capabilities

Bindings and permits sharply contrast with generic, heavy-weight capabilities in other systems [87]. We selected them in order to allow each application to tailor them to its needs. Not all resources and services need equally complex capabilities [77]. For reasons of efficiency, we rejected the idea of combining bindings and permits to one form of capability. Although sometimes when a server passes a permit it should pass also a binding for a service to use the resource, it might be the case that the receiver of the permit already holds the appropriate binding. Likewise, passing a binding for a service that uses its "own" resources does not necessitate passing a permit.

Creation and Birth Inheritance

Creation of servers, resources and services is not controlled by the OSB, so that applications can choose the computation/communication models and the inheritance semantics as they see fit. Activities are an exception because their execution states need to be protected, but still their management is left to their owners. It is left to the design level to appropriately support unique naming and identification. Since bindings and permits are regular data structures stored in the space of any ordinary server, diverse inheritance schemes can be easily implemented.

Termination, Cleanup, and Inheritance after Death

If termination were controlled by the OSB, then the OSB would impose its own termination semantics. How would the OSB know death-inheritance rules for resources it does not control, or avoid unnecessary cleanup chores when an entire application terminates?

Although servers and activities can be terminated by any server, some means to guarantee orderly termination are needed. In the model's view, termination can be announced via service invocations or via exceptions (that is, prior to termination or by another server). Termination can be discovered through the OSB, which validates server and activity ids. When a server terminates, other servers are not automatically notified because such notification is rarely necessary. Notification is left to whoever terminates or discovers the termination of a server. The resources owned by a deceased server are inherited by other servers; each host supports inheritance semantics appropriate to its resource.

Exceptions

An exception raised by a current server of an activity is useful as a "self-addressed" notification, to be noticed when the server is re-executed by that activity. A server cannot achieve this task through its data structures, since the server would have to check for an exception when it is re-executed by that activity, namely, checking would be required at virtually every instruction. Exceptions can be used to notify an activity of a preempted resource and to trigger the activity to recover its state.

Data Abstraction and Hiding Hardware Dependencies

Modern languages and operating systems encourage hiding of implementation details from the user. So does the FOCS model. Implementation details can be hidden from a customer that prefers a higher view of resources and services. But a server that wants to directly access physical resources or manipulate virtual-to-physical resource mapping cannot do so efficiently through another agent. Therefore, that server can (and in fact must) be aware of the underlying architecture, hardware, access protocol, and any low-level mechanism. This argument is similar to what other researchers call the "don't hide power" principle [27, 82].

Upcalls and Trust

Some systems propose special mechanisms to allow backward procedure invocations between system layers [44]. We believe that structuring a system based on trust between components is more appropriate than defining rigid layers. With the semantics of service provision, "upcalls", "downcalls", or "sidecalls" are uniformly supported. Hypothetical layers are formed dynamically by passing bindings. Data transfer between layers in either direction is simplified with permits. An "upcall" invocation of an untrusted "layer" can be handled asynchronously, aborting the activity when its execution exceeds a time limit.

A binding is modeled as a capability that can be created on the fly, without particular rights, so that procedure parameters can be passed efficiently. Such parameters are needed, for instance, to announce service completion or, as in an extendible DBMS, to consult a higher layer (a customer) about different options in an access method [126, 127].

Charging for Resource Usage and Services

Charging is needed to help reduce contention for scarce resources, or support fair distribution of them. Even in a friendly environment charging is useful to detect erroneous or malicious monopolization of resources by some user or application [29]. Charges can also serve as a trace for usage statistics. The charging mechanism, however, could be unfair or overly inefficient — two problems which should be coped with at the design and implementation levels.

Chapter 4

A System Design

4.1. Overview

This chapter details a system design based on the model. The main purpose of the chapter is to illustrate the efficiency and complexity issues of a fully open system. Other purposes are to explore mechanisms that support protected openness, to derive possible policies of the OSB, and to examine the interfaces needed in certain service domains such as naming, CPU management and memory management.

We present services and techniques by which the abstract system outlined in the previous chapter can be realized. They are presented bottom-up starting from the mandatory services of the OSB and continuing through its optional services, conventions in an installation, and services of an example COS. In various cases where different services or interfaces can accomplish a given function, we describe the problems involved with accomplishing the function, illustrate a possible solution, and discuss alternate or extended solutions.

4.2. The OSB

4.2.1. Introduction

This section presents the OSB services, focusing on services to support CPU management, activity management, recovery, and accounting. A resource-allocation policy of the OSB and possible extensions of the OSB services are discussed. This discussion highlights the trade-offs between protection, efficiency, complexity, and fairness. The intent of this section is to show that the basic services are simple and can be efficiently implemented.

A system is started as follows. The primordial server is the *Initiator*. It creates the Universe into which the OSB is mapped. Then it creates and loads the OSB into memory. This stage includes taking inventory of system resources and creating their hosts. The actions of the *Initiator* are guided by a *configuration schema*, which is an initialization plan prescribed by the *System Administrator*. The schema also tells which bindings are required initially, defines the ordering of CPU ownership (i.e., CPU priorities) and of other system resources, and specifies accounting information. The OSB then creates the application-level servers which are required initially, such as the initiators of a COS and of other predefined applications.

The OSB imposes three requirements on naming and addressing that are necessary for the provision of its services. First, a virtual address which references another Space is interpreted as the pair (Space id, address within the Space). The Space id should contain the id of the memory host to which the Space is mapped, as shown in Figure 4-1. This requirement is necessary so that a CPU can submit references to the appropriate memory host. It implies that a memory host and any p-host that is stored in its private memory must be registered with the OSB to obtain a unique identifier.

The OSB is oblivious to the way that the memory host interprets the Space id and the address within the Space. Later we will discuss an addressing scheme supported by the example COS.

Second, a server is identified by the Space it is stored in. Servers may have other identifiers to which the OSB is oblivious. It is assumed (but *not* required) that the Space id is invalidated by the U-mgr when the server terminates and its Space is released. Failing to do so is considered a misservice of the U-mgr to its customers because obsolete references to that Space may confuse the state of a newer customer stored in the Space. For example, a return address from a previous invocation made by the deceased server is not detected as invalid.

Last, a binding for a service consists of two lists (see Figure 4-2). One list is held by a customer and is used for service invocations; it consists of the a key, a target address, and optional attributes. The other list is stored at the target address at the service provider; it consists of a lock value and a list of references to the operations that implement this service. As discussed shortly, a customer specifies at service invocation the binding and the operation chosen. The CPU verifies the binding by comparing the key against the lock, then transfers control to the requested operation. This structure of a binding enables servers to create, copy, and invalidate bindings efficiently. Later we will present additional, non-mandatory conventions about bindings, aimed to simplify communication between servers. For simplicity of presentation, a customer's list is henceforth referred to as the *binding*, and the server's list as the *binding's origin*.

Various OSB services obtain permits to access buffers in their customers' virtual memory. Each such permit is similarly assumed to be a data structure that contains a target address and optional attributes. The attributes are passed to the memory host of the target address for verification at access time.

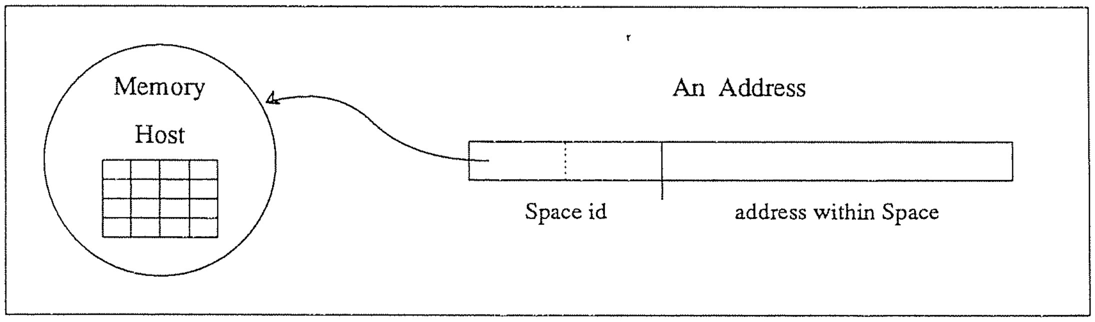

Figure 4-1: An address

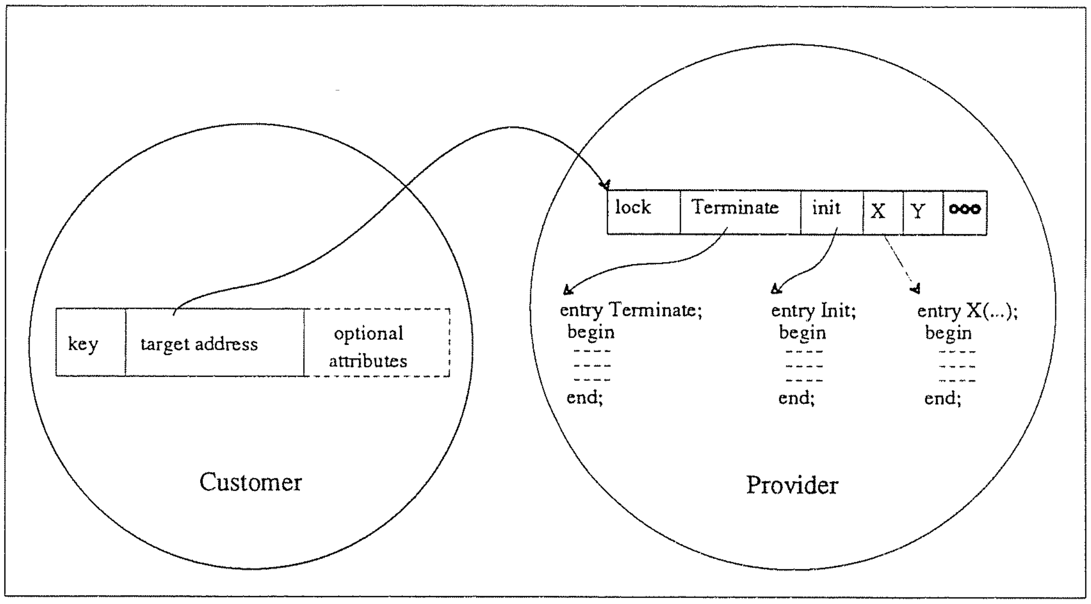

Figure 4-2: A binding

4.2.2. Services for Activity Management

Activities can be created transparently to the OSB. However, since the OSB protects the execution state of the activities that use the CPU's, such activities must be registered with the OSB in order to be allocated activity context descriptors (ACD). In this section we discuss the OSB services for activity creation, service invocation, service return, and handling exceptions.

Creation

A creator of an activity announces it to the OSB, via the *NewACD* service. (Appendix A.1 provides a summary of the interface of OSB services for activity management.) The invoker of *NewACD* becomes the owner of the allocated ACD and is considered also the owner of the activity. It can therefore initialize the state of the activity by manipulating the ACD, modify the ACD later, and dispose of it when the activity terminates. An owner can let other servers perform these tasks via permits to the ACD. The OSB guarantees unique ACD ids. From the point of view of the OSB services, an activity is identified by its ACD.

To initialize the activity's state, its creator has to specify the first service that the activity will start to execute, an address to return to from the service, and optional parameters to that service. The return address is conceivably of the creator's routine that terminates the activity. In addition, the creator has to supply an *accounting permit*, which enables servers in the activity's chain to charge the activity for using their resources or services. The owner is supposed to set the activity's bindings. These bindings (or pointers to them) are stored in the ACD, so that the OSB can easily locate them when a server in the activity's chain invokes a service through these bindings. For protection, the OSB verifies that the specified initial service and return address are either in the creator's Space or are referenced by valid bindings. For the same reason, the ACD owner is not allowed to modify the environments of services preserved in the ACD and their return addresses.

Service Invocation and Return

A service is invoked via the CPU host's service ***Invoke***.[1] The invoker indicates a binding for the service, which the invoker or a given activity holds, and specifies the selected operation. The binding's key, the target address, and the operation are verified. If they are invalid, then the invocation fails and returns a distinctive indication so that the invoker can recover appropriately. Otherwise, the CPU host preserves the return address in the activity's ACD and transfers control to that operation. The invoked server becomes the activity's current server. As a matter of efficiency, the invoker can specify that the invoked service should not return to it, but rather to a former service in the dynamic chain. This feature is useful in situations where the invoking service's role is merely to select an appropriate service for a given case. For instance, a Universe manager's fault service, invoked because of a fault caused by a server in the Universe, need not be returned to after invoking the server's fault-handling service.

How should parameters be passed to the invoked service and result parameters returned to the invoking service? We have decided that they are passed in absolutely-addressed memory protected by the OSB, which for simplicity is called here the *CPU registers file*. The registers are inaccessible to other activities; they are preserved in the ACD upon activity switch. This decision is based on considerations of protection, efficiency, and simplicity. If parameters were passed in a stack that resides in the invoker's Space or some other Space, then it would require more than one *current Space* per CPU at a time. This requirement increases the complexity and potentially the inefficiency of address translation. If alternatively the stack were copied at each invocation and return, then it would render invocations inefficient. Transferring the stack between the Spaces of the invoker and the invoked server was rejected for complexity reasons, especially since the Spaces might be in different Universes. Moreover, in the last two alternatives it might be complex or imprudent to maintain stack consistency when the activity's quantum expires in the midst of service invocation or return. In either case, vulnerable values stored in the stack are not protected — they can be erroneously corrupted by the U-mgr of the invoker or the invoked server. Finally, the alternative of relegating stack management to the OSB was also rejected. This alternative would put the complexity of stack allocation into the OSB. To avoid having a second current Space would necessitate an additional mechanism to access this stack, e.g. through special capabilities as proposed by Dennis and Van Horn [48]. As we will show in the next chapter, with adequate architectural support the registers file can be viewed as a preallocated stack in absolutely-addressed memory.

The types of arguments expected by the service and the registers in which they are passed are declared by the service provider. Likewise, it declares the types of the results and the registers in which they are returned.

[1]To avoid an infinite cycle, the *Invoke* service is not invoked through *Invoke*, and likewise does not require a binding.

The number of registers may be insufficient when a large amount of data is passed. A parameter, therefore, may be an access permit for a structure where additional parameters are supplied, for a large buffer, or for a parameter passed by reference. The invoked server/invoker then "pulls" the parameters whenever it wants. This method is reminiscent of the active *Accept* primitive of SODA [74] and the buffer copy by the receiver in V [40]. It is up to the invoker to save—prior to the invocation—the registers it does not want to be altered or inspected by the invoked service, and restore them thereafter. It is left to the invoked service to record which activities execute it and what internal resources they occupy. This information is useful for cleanup at service abortion, or when recovery chores are performed by another activity.

A service normally returns via the CPU host's service *Return*. For recovery purposes, *Return* can be forced by the activity's owner, as will be discussed in §4.2.4. What happens, however, if the returned-to server has terminated after the invocation, as shown in Figure 4-3? Upon return, the CPU is notified by the server's m-host that the return address is invalid. Consequently, the CPU host forges a return to a former invoker in the chain with a predefined result indication. A similar return is enforced if during execution or at activity dispatching the CPU finds out that the activity's current server has terminated. The intervention of the CPU host in these cases stems from protection considerations: to ensure orderly return to the former server in the chain.

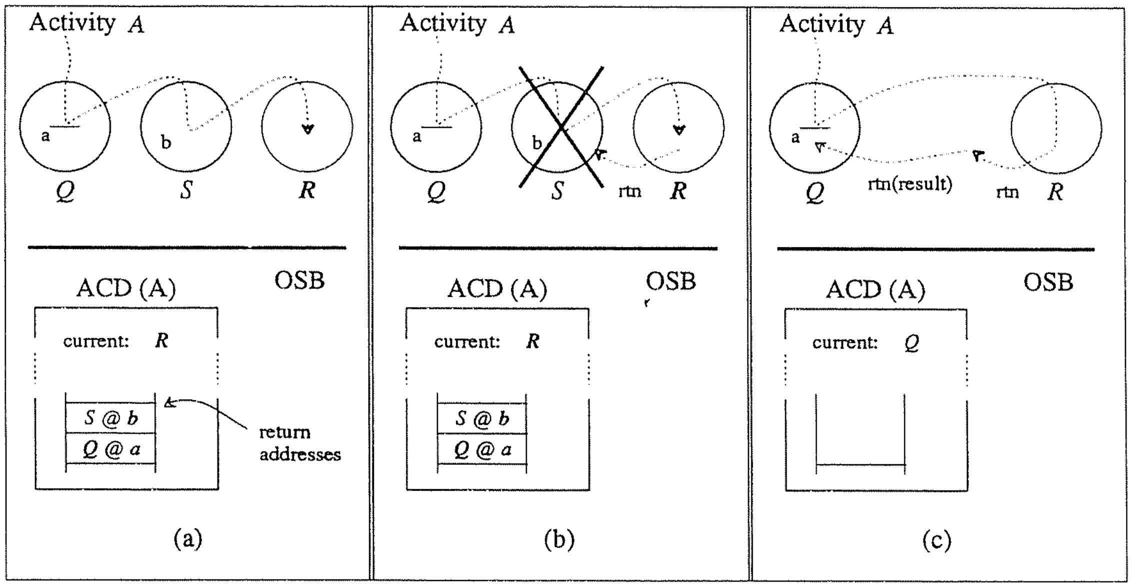

Figure 4-3: Service return to a deceased server

(a) The return addresses of services in Activity *A*'s chain are recorded in its ACD.

(b) Server *S* dies. Its death is discovered when *R*'s service returns.

(c) The CPU host forges return to *Q* with an indicative result.

Context Switch between Activities

The services for activity scheduling are *Dispatch*, *ChangeQntm*, and *Switch* (see Appendix A.2). Since they interact with the CPU management services, they are discussed in the next section.

Exceptional Events

Inter-activity exceptions (called for short **exceptions**) are raised via the OSB service *Raise*. The invoker of *Raise* specifies the target activity *A*, the exception type *T*, and a value to pass to the exception handler. As explained in the former chapter, raising an exception is allowed only to *A*'s owner and *A*'s current server. Also, the standard interface includes a protocol for invocations through exceptions, which defines some standard exception types. *T* can be one of the standard types or an application-specific type.

The exception is noticed immediately when *A* resumes execution. Execution is *diverted* by the CPU host to an appropriate handler of *A*'s current server *S*. Conceptually, this diversion occurs as follows. The CPU host notifies the m-host of *S* of the exception. The m-host subsequently invokes the U-mgr of *S*, which invokes a designated service of *S*. If there is no such service, the U-mgr may invoke a service preregistered for debugging *S*, or terminate *S* altogether.

For reasons of efficiency and simplicity, this sequence of invocations can be cut short as follows. The standard interface defines where exception handlers or their bindings are found. The CPU host, therefore, can simulate transfer of control—that is, a procedure call—to the appropriate handler of *S*. If such a handler does not exist, the CPU host then tries to invoke the appropriate handler of *S*'s U-mgr. If the latter is not found either, then the regular sequence is followed. In either case, since the exception may imply termination of *S*'s service, then *S*, its U-mgr, or its m-host can return to the former service in *A*'s chain.

An interesting question is how to pass parameters to an exception handler. Parameters cannot be passed in "ordinary" registers, since an exception occurs asynchronously, without letting the current server save the registers in use. To avoid having the CPU host manipulate the current server's or the U-mgr's local stacks (what if they don't have a stack, or what if the stack overflows?), we adopted another solution: parameters should be passed in CPU registers designated for this purpose. Similar to the ordinary registers, the exception registers are protected by the CPU and are preserved at activity switch. These requirements are likely to increase the complexity of the CPU architecture. We will refer to this issue in Chapter 5.

A related question is how to handle a subsequent exception that occurs before a former one has completed. The solution adopted is that when a second exception is raised and noticed, the CPU host saves the exception registers and transfers control to the handler of the second exception. For reasons similar to the above, the exception registers are saved in a stack associated with the activity and maintained by the CPU host. Consequently, at exception-handling completion the handler should indicate this fact to the CPU host by invoking *ReturnFromExcpt*, in order to restore the environment of the former exception.[2]

A minor extension of the mechanism of exception raising allows a former server in an activity's chain to ***Raise*** an exception that the server itself would handle when it is reexecuted by that activity. This feature is useful in a situation when a server S notices or generates an event that affects the course of execution of another activity A, of which S is *not* the current server. For example, S preempts a critical section from A, and so S wants this event to be noticed by A once it resumes running at S. The motivation for this feature is to simplify the programming of servers so that explicit checking for exceptions is not required at each service return. This kind of exception is similarly treated by the CPU host.

Another group of exceptions are **intra-activity exceptions**. In this group we distinguish between **language-based exceptions**, such as exceptions in Ada® [72], and **traps**. The former are raised invisibly to the CPU. They should be handled by language-provided tools (see [18], for example) transparently to other activities, servers, and the CPU host.

A **trap** is an exception condition raised in the course of execution by a CPU or an m-host. A trap occurs due to an erroneous machine instruction—e.g. invalid operator or operand—or due to an error caused by that machine instruction—e.g. integer overflow or floating-point underflow. Every trap is noticed and examined by the CPU host. The action taken is according to a predefined classification of trap types. If the trap is such that it does not require altering the course of execution, then an appropriate indication is returned—e.g., a flag is set in a register called R_{OVFL}. The trap can be in the handling domain of the CPU host, e.g. a trap indicating quantum expiration, and thus it is handled by the latter. In this particular example, the CPU host performs an activity switch. Otherwise, the trap type is defined to be handled by the current server, its U-mgr, or its m-host. Examples of such traps are division by zero, a mapping fault, or a reference to an invalid Universe. These traps trigger the CPU host to transfer control to an exception handler in a fashion similar to handling inter-activity exceptions.

Finally, an **interrupt** is a request to reclaim a CPU for urgent work, posted by a p-host or a former CPU owner. It results in preempting a CPU from a scheduler and suspending the activity using that CPU. The CPU host would later resume the activity on that CPU or another one. An interrupt, therefore, does not affect the course of execution of an activity, other than causing a delay.

Service Extensions

Postpone activity switch. A quantum expiration that occurs in the midst of a critical section or an urgent function is undesirable from the current server's point of view. Hence, the CPU host lets a server postpone time expiration for up to a predefined *short quantum*, after which activity

[2]It is conceivable that in practice only a few exception registers and a short stack of them are needed, for two reasons. First, only an exception type and none or a few parameters have to be passed to an exception handler. Second, exceptions are infrequently raised. Hence, the overhead imposed on the OSB to support exceptions would be minor.

®Ada is a registered trademark of the U.S. Department of Defense (Ada Joint Program Office).

switch is enforced. This feature guarantees that a *short* critical section or an urgent function can be performed without interruption, unless they exhibit traps that are handled for a lengthy period.[3]

Memory replacement of ACD's. If the OSB lacks sufficient physical memory to store all ACD's, some activities will suffer a page fault upon dispatching. Such an event is undesirable for an activity dispatched for urgent work or a computation with real-time constraints. Therefore, the interface with the OSB can be extended to allow an owner of activities to direct the ACD-replacement policy of the OSB. Specifically, each owner is allotted a quota of physical memory and is let decide how this quota is allocated to its activities. Alternatively, the OSB lets an owner dictate which ACD's to pin in core and charges a higher price for such a service.

Status. The OSB provides a service to check an activity's status, such as its remaining quantum, pending exceptions, and how long the activity is idle. This information can help a server decide whether to reclaim an internal resource from an activity. For instance, when a server finds that a critical section is occupied by another activity, it can consult this information to decide whether to block the current activity, to unroll the former activity from the critical section, or to go into a "busy-wait" loop until that activity releases the critical section. A further extension would be to let a server choose the order of handling the pending exceptions.

4.2.3. Services for CPU Management

The CPU host has two roles: to allocate CPU's and to verify the allocation and usage of CPU's by other servers. The first role includes a mechanism which is described below, and a policy which is discussed in §4.2.6. The second role includes services to transfer CPU ownership (both allocation and revocation) and to switch between activities. These services are intended to support any application-level mechanism or policy in a protected way. Below we describe these services, discuss problems raised by them, and present our solutions. We discuss the efficiency and complexity of the services and possible service extensions. The interfaces of these services are presented in Appendix A.2.

CPU Ownership

The CPU host allocates CPU's to a predefined set of applications. The customers specify *demands* of CPU time through the *GetCPU* service. The invoker of the service may specify an urgency level and the number of CPU's needed. These parameters are useful to accommodate the concurrency or responsiveness level of the requester's customers. The invoker may also indicate that the demand is a continuous one, namely, to be renewed when the allocated slice is consumed. This feature solves a potential race condition occurring when that server's slice expires — if the server has more work to schedule, but not enough CPU time to invoke *GetCPU* again.

[3]As mentioned earlier, longer critical sections can be dealt with by requesting preferred scheduling from the activity's scheduler through the activity's bindings. In any event, a server can *Switch* to an activity whose quantum has expired in a critical section, so the formation of a *convoy* [61] of activities can be avoided.

The CPU host allocates time in *real* Time Units. It satisfies a demand partially or gradually when the level of contention among its customers is high. Since all CPU's are identical, the CPU host can distribute an allocation among more (or fewer) CPU's than requested. To simplify the interface with the CPU host, a customer is allocated *logical* CPU numbers mapped dynamically by the CPU host to specific CPU's. The mapping is changed transparently to the CPU owner as a result of *interrupts*. A potential extension of *GetCPU* is to let a customer indicate a desired response level, rather than a concrete quantity of CPU's and time units. This extension could simplify the scheduling algorithm of a customer, at the expense of complicating the CPU host's algorithm.

A CPU owner can transfer ownership via the *Allocate* service. The allocator can specify default values such as *all* or *any* CPU. This feature reduces the schedulers' burden of keeping track of CPU ownership. In addition, since the allocator can name any recipient, a higher level of CPU sharing is attainable compared with schemes that support a static hierarchy of schedulers [115, 140].

Two related problems with CPU allocation are how to notify the recipient of a slice of that event, which activity should be dispatched to use the slice. The simplest solution the CPU host may adopt is to do nothing, letting the allocator decide on both questions. This solution seems appropriate, since *Allocate* might be invoked because the recipient has asked the allocator for a slice. So, the allocator can report how much has been allocated via a service return to the recipient. Likewise, the allocator may dispatch an activity of the latter server to report the event via a service invocation and/or use the slice. However, all these alternatives are susceptible to quantum expiration: If the quantum of the activity—by which the allocator tells the event—expires, then the latter server cannot make use of its slice.[4] Therefore, the above solution is not always desirable. Another solution is to let the CPU host dispatch a predefined activity associated with the recipient. This solution requires that that server preregisters such an activity with the CPU host; it also reduces the flexibility of the server to decide which activity is dispatched and when.

The solution adopted is to let the allocator choose among the alternatives. Specifically, the allocator may name the activity to be dispatched, which can be the invoking one. For protection reasons, this activity should be owned by the allocator or the recipient. If no activity is nominated, then a default activity associated with the latter server (denoted its *scheduling activity*) is dispatched by the CPU host. A server can register such an activity via the *Register* service. If there is no such activity, then *Allocate* fails. This solution allows servers to choose the appropriate balance between flexibility and complexity by adding a small complexity to the OSB.

A former owner of a slice can reclaim the unconsumed portion of it through *Revoke*. A current owner can voluntarily *Release* it. Each released slice is returned to the possession of its former owner, who is notified by the CPU host. (The notification mechanism will be described shortly.) An *interrupt* is a generic version of *Revoke*. Interrupts are conceivably more frequent and endure for a shorter time than regular revocations. Hence, for efficiency reasons, an interrupt does not actually cancel the CPU ownership of current owners. The owners are reallocated a CPU by the

[4]Notice that adding the entire slice to the current activity's quantum is not possible when the slice is of one CPU and the activity runs on another CPU. Doing so after the activity's quantum expires requires the CPU host to monitor the activity, hence this alternative adds complexity to the CPU host.

CPU host when one becomes available.

Notice that a CPU owner may get slices from several servers, and it might have allocated its slice to several servers. Hence, the CPU host has to decide which slice is reclaimed at *Revoke*, when this is not obvious from what the invoker of *Revoke* has specified. For the sake of consistency, the CPU host records the order of allocations per server and per CPU. It employs a simple rule of First-Allocated First-Used, which can be implemented simply as a stack. The overhead of this decision and additional bookkeeping adds minor complexity to the CPU host.

Context Switch between Activities

Dispatching an activity implies a context switch from the activity that is currently running on the specified CPU to the dispatched activity. Therefore, the *Dispatch* service suspends the dispatching activity if it runs on the same CPU; otherwise, *Dispatch* is analogous to an asynchronous service invocation. The remaining quantum of the suspended activity is returned to the possession of its scheduler. The scheduler can dynamically upgrade the activity's quantum via the *ChangeQntm* service.

Similar to the problems mentioned above with *Allocate*, there are two problems with transfer of control and notification. First, which activity runs after the dispatched one consumes its quantum? Notice that the dispatching activity might no longer be suspended at *Dispatch*, for instance because the dispatching was asynchronous, or because the dispatching activity has been itself dispatched afterwards on another CPU. Second, should the scheduler be told when the dispatched activity consumes its quantum?[5] It is not always possible to notify the scheduler through a result returned from *Dispatch* for the same reason as above. Notifying the scheduler by invoking its predesignated service, similar to notifying the completion of an asynchronous service, raises two additional problems: Should the CPU host allocate an extra slice to the scheduler to perform the service, in case the scheduler has exhausted its CPU ownership before or during that service? If so, how can it guarantee that this service completes promptly?

The solution adopted to the transfer-of-control problem is again based on the notion of a *scheduling activity*. If the server has registered such an activity, it is dispatched whenever a previously dispatched activity consumes or releases its quantum. Otherwise, control is transferred to the dispatching activity; if that activity concurrently runs on another CPU, then the transfer is delayed until the activity idles. The first alternative is simpler when a server dispatches many activities, because then the server does not have to synchronize scheduling decisions among different activities that perform its algorithm. Moreover, this alternative avoids the risk that an activity gets suspended within the scheduling algorithm when its quantum expires, and thus causes a delay of scheduling decisions until the scheduler can *Switch* to it, if at all. The other alternative is simple in the simpler cases, such as when a server infrequently alternates the use of a CPU between a few activities. It

[5]An activity does not consume its entire quantum if the CPU is revoked from its scheduler, or if the activity elects to pause or to be blocked. The first event is told to the scheduler by the CPU host; in the other events the activity tells the scheduler. Hence, no further intervention of the CPU host is necessary in these events.

does not require that a server preregisters an activity, but it might be less efficient since it delays a transfer of control. In either case, a server can choose the alternative, and again, it can decide the appropriate balance between complexity and flexibility.

The solution to the notification problem is combined with solutions to similar problems at CPU allocation and revocation. Specifically, the CPU host enqueues "notifications" that describe the events and passes them either at service return or via a shared buffer. The exact mechanism is described shortly.

A server can *Switch* to another activity of which the server is the current server. The server can restrict the switching by a specified time limit parameter. This feature is useful in a situation when a server wants to invoke a given service, and to ensure that the latter service returns within the time limit. For example, in the midst of an urgent operation a server may want to *Unblock* another activity or to notify a customer of an access completion — in both cases without suspending the current activity for too long. This simple time-bounded invocation does not abort the invoked service when the time limit expires. Hence, it is safe since it does not pose the risk of leaving the state of that service inconsistent.

The parameterless *Switch* is used to switch back. This option is important for fairness. For example, imagine that a server executed by activity A finds that a critical section is occupied by Activity B, and thus switches to be executed by B. When B leaves the critical section, the server can switch back to A to continue that computation.

Notice that a switched-to activity may switch to another activity, for instance when the activity running in one critical section must enter another one. The CPU host maintains a graph of switched activities that tells which activity to reactivate when the switched-to activity completes its subquantum or asks to switch back. This graph may become very complex since activities in the graph might be dispatched by their schedulers or *Switch* to other activities already in the graph. It is unclear whether the OSB can maintain correct dependency semantics between the activities in such situations. In practice, however, it is conceivable that *Switch* is mainly used for short-term transfer of control. Therefore, the semantics of *Switch* are defined to support the simple situations efficiently. Specifically, the graph is maintained until the quantum of the root of the graph—the dispatched activity—is consumed or released. It is left to the activities to *Switch* again when they are redispatched, if the conditions that formerly required the switching still holds. The major reason for this decision to abandon that graph is that until the "root" activity will be redispatched, other activities might be dispatched as well, and perhaps resolve the conditions that required the switching.

For a similar reason of simplicity we decided not to reject the dispatching of an activity already in the switched-to graph, nor maintain a complex graph. Instead, in such an event the former graph splits and the latter activity becomes the root of a new graph. For instance, suppose A is dispatched, then switches to B, which switches to C, which switches to D; the graph is A-B-C-D. Now C is dispatched; the graph splits to A-B and C-D. It is up to B to wait until the condition for which it has switched to C is resolved. The solution is based on the assumption that such scenarios will rarely occur, and thus they are not worth the extra complexity of maintaining more sophisticated graphs. This solution also resolves potential scheduling conflicts between the current server of an activity (which requested the *Switch*) and the activity's scheduler by favoring of the latter.

Notifications

As discussed above, a server has to be notified when (1) its slice is revoked, (2) a slice it has allocated to another server is consumed or released, or (3) an activity dispatched by the server consumes its quantum. It is not always possible to furnish a notification as a result returned from the service relevant to the event, and not prudent to invoke an acknowledgement service of the customer. In addition, when a server is asked to make scheduling decisions, it wants to be kept up-to-date with the results of its former scheduling requests. Hence, the CPU host has to queue the notifications. If the server has furnished a buffer, then the notifications are stored there in a list. In this way, the server can be up-to-date simply and efficiently. This sharing of a buffer raises three problems: (1) The concurrent write problem, in which both the CPU host and the server try to modify the list, (2) The concurrent read–write problem, in which the CPU tries to modify a list entry while the server reads it, and (3) a buffer overflow problem when the CPU host generates notifications faster than the rate at which the server reads them. To avoid the first problem, only the CPU host should modify the list. The server can only modify an indicator which tells how far it has inspected the list. A mutual-exclusion mechanism is employed to avoid the second problem, based on architectural features such as memory locks or a test-and-set machine instruction.[6] If the buffer overflows or if the server has not provided a buffer, then the CPU host queues the notifications in the OSB, up to some implementation-dependent limit. These notifications are passed to the server either upon a subsequent return from a service such as *Allocate* or *Dispatch*, or when the server explicitly requires the notifications by invoking a *GetNotices* service.

Handshaking with Schedulers

Two services let servers establish their interface with the CPU host. A server can use the *Register* service to declare which activity is its *scheduling activity* and to supply a buffer for notifications. The *YourCustomer* service is used by a server to become the CPU host's customer of CPU allocation.

Service Extensions

Status Information. The interface with servers can be extended to help them recover from lost notifications and to tell them global scheduling information. The former task is supported by letting a server check the "balance" of its CPU ownership, the status of a given activity, and the result of the last dispatching of the activity. To provide global information, the CPU host maintains *system load* statistics, such as the number of schedulers and activities, as well as averages of CPU demand and CPU consumption. The statistics can be reported either via a new service or by giving servers read-only access permits to the memory area where these statistics are accumulated. This information can improve schedulers' decisions with regard to CPU demand and allocation since, as explained shortly,

[6]Having the buffer in the server's address space raises a risk: a mapping fault may occur when the CPU host tries to put a notification there, and the server's U-mgr may does not resolve the fault in a short time, thus delaying the CPU host from serving other customers. To avoid this risk, the invocation of the U-mgr's fault handler is delayed until the above server is allocated CPU time, so that the server's U-mgr consumes its time.

the CPU host charges for CPU allocation as well as for CPU demand.

Detached Mode. There is an inherent inefficiency in activity dispatching: when a quantum expires, another activity of the scheduler is dispatched, in some cases only to immediately dispatch another activity. The interface with schedulers can be extended to allow a scheduler to be periodically *detached* from short-term scheduling operations. The scheduler, however, should be able to make short-term scheduling *decisions*, as well as decide when not to be detached. To support the extended interface, a scheduler should supply an access permit to a list of scheduling orders in its address space. The list has a predefined, CPU-host-dependent structure. When a *detached* scheduler owns a CPU, the CPU host dispatches activities from this list, and updates it. If the scheduler indicates that it is no longer in detached mode, an order is incorrect, or the list is empty, then the CPU host dispatches the server's scheduling activity. The scheduler can easily react to changing requirements by dynamically rearranging the list and changing quanta. It can also easily inspect the results of previous dispatches marked in the list, or switch dynamically between being detached or involved in short-term scheduling operations. Likewise, it can put an activity in the list which periodically makes decisions or performs housekeeping chores.

Summary

In summary, we note that most of the CPU management services are very simple. They require the OSB to consult simple data structures to validate ownership without deciding policy issues. Therefore, they can presumably be implemented efficiently. *GetCPU* is more complex because it requires the OSB to arbitrate among conflicting demands when the contention for CPU's exceeds availability. Some of the complexity involved in services for CPU allocation, revocation, and activity scheduling is inherent to the environment of multiple CPU's. We carefully crafted them to cope with concurrency problems as well as the problem of time expiration in critical situations. For example, *ChangeQntm* allows a scheduler to increase the quantum of an activity executing its service so that the service is not suspended before completion. *Switch* and the service to suspend quantum expiration allow any server to avoid time expiration in a critical section. The notification mechanism should be simple in cases when a server supplies a buffer, dispatches a few activities, or dispatches them sequentially.

The services above allow dynamic sharing of CPU resources and immediate revocation when needs change. Although the services are tailored to support any hierarchy of schedulers, it is conceivable that the dynamic hierarchy will be relatively flat and so that negotiations and allocation will not be very complex.

4.2.4. Support for Authentication and Recovery

An additional role of the OSB is to support authentication. Accordingly, the OSB guarantees unique identifiers for activities and memory hosts. The Accountant authenticates account permits, accounts, and the users associated with accounts (discussed later). This service allows servers to base their protection mechanism and allocation decisions on the identity of *user* customers, rather than server ids. The OSB does not guarantee server ids to be unique, since it may be reasonable for some U-mgr or server manager to associate a single id with multiple servers. The OSB, however, provides a service to validate a given server id. It is left to the interface between servers to further authenticate each other, e.g. by using passwords or logical names.

Two additional OSB services for authentication are intended to simplify other tasks of servers. First, a server can find out the id of an activity running at the server. This service helps the server to record allotments of internal resources per activity and easily reclaim them when the activity aborts. Second, a server can find out the (server) id of its invoker so that reauthentication at each invocation is not necessary. The id of an invoker is taken from the binding that is used to invoke its own service. For the sake of efficiency, these activity and server ids are stored in CPU registers readable by any server, denoted *CurrAct* and *PrevSrv*.

The service to check whether a given server id is still valid enables a server to reclaim resources from deceased servers. For example, if server *S* maintains a mutually-exclusive lock which is held by server *R* and for which other servers are waiting, it wants to free the lock once *R* terminates. However, it would be very inefficient if every server must continuously check whether the servers it is associated with are still alive. Putting the complexity of announcing servers' death into the OSB imposes the overhead on the OSB. Hence, it is left to intermediary servers to provide such checking services. For example, imagine the existence of a "watchdog" server, which periodically checks whether servers from a given list are still alive. *S* can ask that server to watch *R*, and hands a binding for a "reply" service. When the watchdog discovers *R*'s death, it reports the event to *S* through the reply service. (An analogous method is used in Swift [44] for a similar problem, but there the complexity and the overhead of distributing death notices are put in the kernel.)

The OSB helps servers recover from the termination of an activity or of its owner. An activity whose owner has terminated cannot be dispatched any more. Consequently, servers in the dynamic chain of such a *dangling* activity might be unable to recover their state, especially the activity's current server who has not had the opportunity to save its state. Therefore, for protection reasons, the OSB inherits the activity and schedules it for orderly termination. Specifically, the OSB raises an exception of a predefined type, and dispatches the activity. The exception triggers the handler of the activity's current server to perform cleanup and to report the exception to a former server in the chain, and so forth until the chain is "untangled". When the activity's original service returns, the OSB can safely dispose of the activity. This scenario resembles language-based recovery methods such as that of Mesa [81].

What if the untangling process takes longer than the scheduler allows, for instance because some service either requires a very long cleanup period or it loops forever? The OSB handles such a case by setting a predefined limit on the time allotted to each server in the chain. If a server does not return within that limit, then a return is forced by the OSB. The returned-to service is notified of the event with a predefined result indication, such as *ReturnForced_ActivityTerminates*. This time limit has to be decided at the implementation level. It should suffice *at least* for creating an activity and assigning to it the task of cleanup, e.g. by copying the state of the terminating activity to the new activity. In this way, a well-behaved server that has to perform lengthy cleanup chores is protected, while the overhead and complexity imposed on the OSB is minimal.

For analogous reasons, intervention of the OSB is necessary at activity termination. Suppose that a well-behaved activity owner raises an exception on the activity to notify the activity's current server of its intent to terminate the activity. But as in the scenario above, the untangling does not complete. (Recall that the owner may inspect the activity's chain and find out whether the untangling proceeds properly.) As above, the owner may invoke *Return* on behalf of the activity, and the OSB returns a similar indication to the returned-to service.

One might ask how the OSB can impose "good behavior" on the activity's owner in order to protect the servers in the activity's chain. After all, the owner itself might be terminating, or it might lack CPU time to dispatch the activity, or an impatient user might dictate an urgent termination of the computation. We adopted the following solution, combining it with the solution described above: an owner may force the activity to return to a preceding service in its chain by invoking *Return*(Activity id, # of hops). To protect the servers "hopped over", the OSB enables them to perform cleanup as before. Specifically, as illustrated in Figure 4-4, popping m chain-hops from activity A that has an n-hop chain ($n \geq m$), creates a new activity B with a chain of the last m hops of A. Activity A remains owned by its previous owner. Activity B, however, is inherited by the OSB, which schedules it to complete cleanup as described above. In order to prevent an activity owner from flooding the system with such dangling activities, the OSB may refuse that *Return* if it reaches a limit on the number of activities per system or per owner. Furthermore, Activity B is assigned the accounting permit of A, so it does not get resources or services for free. Notice that this mechanism allows servers to implement a time-bounded service invocation, while the invoked server is assured the opportunity to recover an aborted invocation in an orderly manner.

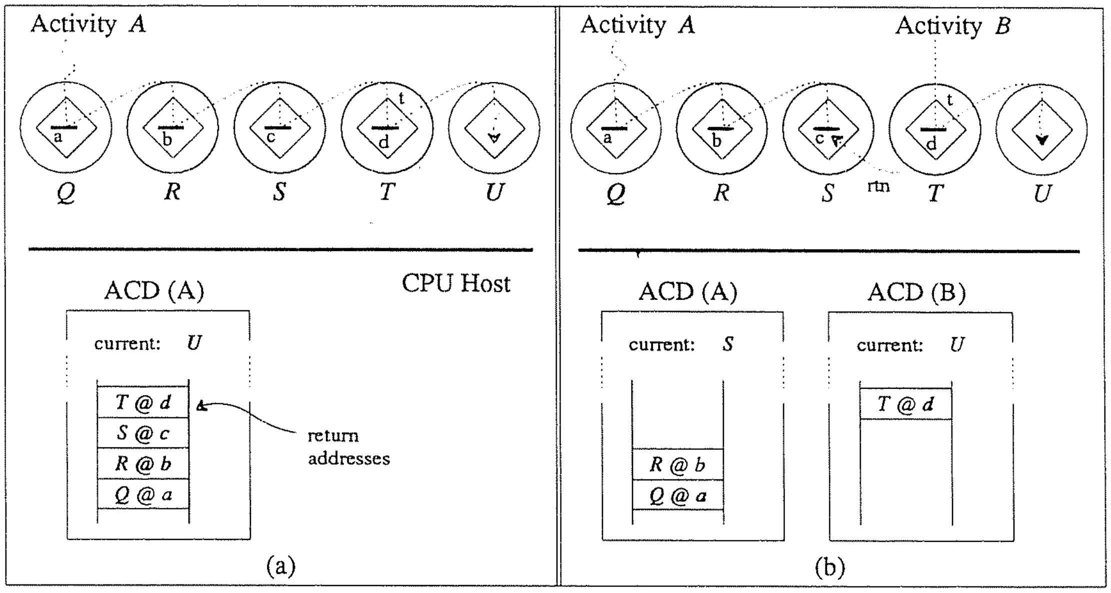

Figure 4-4: Splitting an Activity

(a) Activity A runs at Server U when its owner asks the OSB to remove two hops from A's chain. (b) Activity B is created and is assigned the chain (T, U); A retains the remaining chain.

4.2.5. The Accounting Services

This section presents mechanisms that allow servers to charge for their services and resources. We first draw a basic mechanism that allows whoever submits a permit for an account to debit that account. This mechanism does not allow an account owner to protect the account from being debited excessively or too frequently. Two extensions to support this protection are subsequently presented, and their added complexity and overhead are discussed.

Elements

Table 4-1 summarizes the elements of the accounting mechanism. For simplicity, a single currency system and a centralized banker—the *Accountant*—are assumed. Extensions to multiple currencies as in Amoeba [95] and multiple bankers as suggested by Miller *et al* [93]. are possible, but require more complex accounting mechanisms. Such extensions are not discussed here.

As in Dennis and Van Horn's work [48], principals are the users of the system. They are represented by servers and activities. A principal would conceivably control several applications in the system. It can assign an account to each application and a subaccount to each activity or server. The motivation for supporting subaccounts is to let several servers share the balance of one account—that is, to dynamically balance each other's credits and debits—while each server can issue distinct accounting permits, can have a distinct history of transactions, and can be restricted by a private balance. Figure 4-5 illustrates the accounting hierarchy supported by the Accountant. The

principal	A user or a group of users responsible for the "expenditures" accumulated in an account.
account	A logical resource owned by one or more servers. An account is associated with one principal, and is credited and debited by servers. It records the sequence of such credit and debit requests and the cumulative *balance* that results from these operations.
subaccount and **merged account**	An account A can be a merged account of accounts $A_1, ..., A_n$, in that credit and debit operations on each of $A_1, ..., A_n$ are reflected also in A's balance. Accounts $A_1, ..., A_n$ are called subaccounts of A.
accounting permit	An allowance to debit a specified account with an optional limit on the amount that can be debited. For brevity it is denoted a *cheque*.[7]
currency unit	The unit of accounting transactions (credits and debits) supported by the Accountant.
accountant	A server responsible for maintaining accounts and their balances. The *Accountant* is a generally-trusted server included in the OSB.

Table 4-1: The elements of the accounting subsystem

dashed lines represent optional levels. Each subaccount can be further divided into subaccounts. In practice, the depth of the hierarchy is restricted to some implementation-dependent number of levels. The Accountant's services that support these functions are summarized in Appendix A.3.

Paying the bill for accumulated expenditures in an account bears installation-dependent semantics. In one installation a bill can be paid with real-world currency, while in another installation bills are merely used to record resource utilization. It is left to each installation to decide whether accounts are allowed to accumulate unlimited debts, or whether they have "debit limits" which cannot be exceeded.

Principals and Accounts: Creation and Termination

A new principal is introduced to the Accountant by the System Administrator. A new account is then created to record all the principal's transactions in the system and is credited with the principal's initial budget. If there is a debit limit imposed on the principal, it is recorded in that account. A principal is identified by a secret key established then. Any server presenting this key can perform operations on behalf of that principal.

An account is identified by a public id and a private key. Any server presenting that key is considered the account's owner. All owners of an account have equal rights to it. An owner is eligible to create subaccounts under this account, to issue cheques, and to close the account or its subaccounts. This feature allows a server to close accounts that were spun off its account by "child" servers. An account owner can also change the account's private key. This operation invalidates all outstanding cheques. It may also take away ownership from other owners.

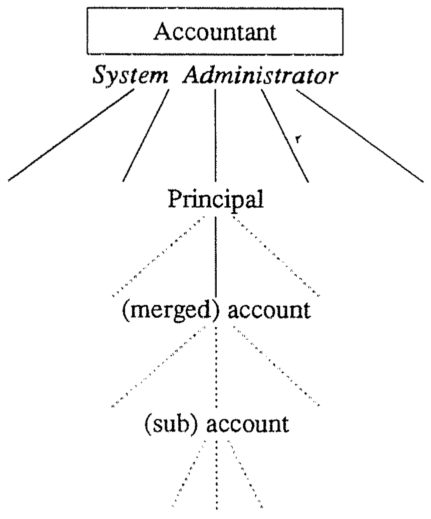

Figure 4-5: Accounting hierarchy

[7]We chose the spelling "cheque" over "check" in order to distinguish it from other uses of "check".

Charging with Cheques

The basic mechanism assumes a complete trust of customers in their service providers. Therefore, a cheque issued by a customer is an unrestricted authorization to charge its account. Any holder of a cheque can *Deposit* it to a specified account and thus charge the customer's account as many times as the holder wishes. It may freely pass the cheque to other service providers.

A cheque is the pair (*FromAccount, Signature*), where

Signature := **Sign** (FromAccount, Account's private key),

and *Sign* is a publicly known one-way function. *Sign* can be a library routine or a machine instruction, so that issuing a cheque is simple and efficient. Cheques are transferred between servers as regular data structures. The Accountant merely verifies the signature at a *Verify* or *Deposit* request by using the same *Sign* function.

Charging with Cheques: Extended Mechanism

The major drawback of the former mechanism is that the OSB does not allow servers to protect themselves. An account owner cannot restrict the amount that can be charged, prevent reusing a cheque, or cancel a cheque. Therefore, cheques are extended to include a maximal amount and a serial number. Specifically, a cheque is the quadruple

FromAccount	Serial #	Max amount	Signature

Signature := **Sign** (FromAccount, Serial #, Max amount, Account's private key),

and *Sign* is as before.

In this mechanism, *Deposit* can charge only a limited amount, and a cheque can be *Cancel*led. The Accountant records the serial numbers of cheques that have been cleared (that is, deposited or cancelled), and verifies the serial number of a cheque at *Deposit*. Appendix A.3 presents the algorithm used by the Accountant for recording and verification. If the issuer of a cheque does not need this protection, it can sign the cheque with *any* amount or a serial number of *none*.

This mechanism allows a cheque holder to charge the relevant account gradually via *Partial-Deposit*. For instance, a file server can charge file owners periodically instead of charging them once for the entire life span of their files. This service relieves a customer from furnishing multiple cheques, which is an inefficient or sometime impractical alternative. However, this service imposes the complexity of remembering the remaining amount of the cheque, in order to protect the cheque issuer from holders that overuse the cheque. To relieve the Accountant from recording such information for each cheque, the Accountant replaces the cheque by a new one with the remaining amount. In addition, the new cheque includes a reference to the original one, so that the issuer can cancel the cheques that spun off its cheque. Another problem is that the serial number of the new cheque should be different from any number assigned by the account owner, since otherwise one cheque will invalidate the other. For simplicity, a convention should be observed, for instance that account owners assign positive serial numbers and the Accountant negative ones.

It is desirable in some cases to let a cheque holder split the cheque between a number of servers. For example, a file server may want to credit its account by part of the amount of a customer's cheque; it may want to use the rest to pay for resources and services of other servers used to maintain the customer's files, such as disk space and communication services. Consequently, *SplitCheque* allows substituting two cheques for one cheque — one for a specified amount, the other for the remainder.

Further Extensions

There is still a protection gap in the former mechanism. The issuer of a cheque cannot enforce that a cheque is indeed gradually used, if that is its intended use. Nor can the issuer limit the amount charged at each *PartialDeposit* request, nor the time interval between consecutive charging requests. Such protection, for example, might be desired by a suspicious customer of the file server in the former example, to ensure that the file server does not charge more frequently than every *n* time units. A possible extension of the accounting mechanism is to define a cheque as the tuple

From Account	Serial #	Max amount	Max each time	Time of charge	Time interval	Signature

where *Signature* and *Sign* are analogous to the former ones. *Time of charge* is the earliest time the cheque can be deposited. *Max each time* is the maximal amount that can be charged at *Partial-Deposit*. *Time interval* is the frequency of charging. The time of charge is set by the Accountant at every *PartialDeposit* or *SplitCheque* of a cheque, again so that the Accountant does not have to record these values for every cheque.

The services presented above do not change except for additional protection, execution overhead, and complexity. The provider's algorithm of charging for resources and services becomes more complex too, since it has to deal with scheduling the deposition of cheques. Moreover, a customer now has to meet the requirements of resource/service providers regarding the amount and the frequency of charging. And since cheques are passed around, they have to meet the requirements of any potential holder. If a provider employs a dynamic pricing function, it would be difficult to produce adequate cheques. Hence, presumably more services and resources would be denied to customers because of inadequate cheques. Providers and users would have to negotiate frequently to agree on the appropriate charging schemes and/or replace refused cheques. Therefore, this mechanism imposes higher overhead and complexity.

Discussion

Each of the three mechanisms in turn improves the protection level of accounts at a cost of higher complexity and execution overhead. None of them, however, can offer the ultimate protection of requiring a provider of services or resources to charge reasonable prices, to distribute the charges fairly among its customers, or even to provide the service/resource for which it charges. Since it is assumed that a customer trusts its service provider with respect to the service, we may presume that the server can be trusted with the customer's resources as well, including its accounting budget. Thus, the basic mechanism should be sufficient for most users.

Since a server is allowed to credit any account, a more efficient and direct charging for resources is possible by eliminating hierarchical charging. A host of a given resource may charge the current owners or users of the resource rather than charging the initial owners, which then charge later owners, and so forth. In fact, the CPU host charges the current owner of each CPU and the dispatched activities directly. Moreover, this feature allows a host to easily refund a server from whom a resource is revoked.

The Accountant records the transactions of each account, up to some implementation-dependent limit of volume or rate of transactions. It is left to the implementation to guarantee that these logs do not produce impractically voluminous data. The implementation can either restrict the frequency of charging or record only the most recent transactions and summaries of older transactions. Since the Accountant does not remember the uncleared cheques forever, the implementation should also guarantee that valid cheques are not made obsolete too soon by space-efficiency considerations. Otherwise, servers would have to *PartialDeposit* cheques very often, or negotiate with the cheque issuers to replace rejected cheques — two outcomes that increase overhead and complexity.

In order to maintain usage statistics, *Deposit* and *PartialDeposit* can be extended to indicate the resource or the service provided. Another server would then provide statistics of resource usage by summarizing the logs produced by the Accountant. These statistics help in finding bottlenecks in the system, as well as in identifying users or applications that are too "greedy." The Accountant may support tracing the logs as well as undoing unreasonable charges. Likewise, the Accountant can let users ask about principals, accounts, and subaccounts, in order to check their balances and their proportional usage of resources. In an installation where balances are not restricted by debit limits nor paid by "hard" currency (e.g., the UW-Madison computer sciences department), such reports pressure users to avoid overly demanding or inconsiderate usage of shared resources.

4.2.6. Allocation Policy and Algorithms

This section presents a general policy and mechanism employed by the OSB for system resource allocation. It then discusses them in the context of CPU allocation, and draws several allocation algorithms. The resource allocation policy of the OSB is based on the following considerations:

(1) *Openness*: The OSB honors the requirements of its customers as expressed by their demands, and lets them implement their policies and mechanisms.

(2) *Priority*: When demands exceed availability, resources should be allocated to the most important demands, based on some estimation of the OSB.

(3) *Fairness*: Even the demands of the less-preferred customers are granted, according to some installation-dependent fairness criteria.

(4) *Simplicity*: The OSB is designed to function as a *coarse-grain allocator*, in the sense that it allocates resources to a *few* customers, in relatively *large* amounts, and for *lengthy* periods. It is left to these customers to deal with the needs of particular applications, computations, or

other allocators.

The allocation mechanism is as follows. Each customer poses a demand in *real* units, e.g. real CPU Time Units. The demand represents the accumulative resource requirements of its customers' work. Since in general the OSB is oblivious to the relative importance of its customers' work, they include importance indicators in their demands, such as an urgency level. The pricing system is used to deter customers from overdemanding. The OSB charges more for urgent demands. For some resources it can give "credits" to customers that abstain from demanding them. These credits can be later used to backup urgent demands. (A similar method based on microeconomics models was suggested to regulate demands in a slotted Aloha multiaccess communication channel [79], giving credits to "silent" stations.) An extension to this mechanism is to let customers *bid* for resources. Similar to some scheduling algorithms for real-time systems [106, 145], arbitration is based on bids. This mechanism assumes that more important customers have higher budgets (or rather that richer customers are more important).

The allocation mechanism can be parametrically tuned to support fairness considerations. The preferred degree of fairness is decided at each installation. To achieve a low degree of fairness such as simply avoiding starvation, old demands are promoted by getting credits or higher bids. A medium degree of fairness can be achieved by guaranteeing a minimal allocation to every customer. The minimal amount is defined as either an absolute value or as a fraction of a demand. To support a higher degree of fairness such as balanced allocation, each customer is allocated equal portion of the resources, normalized by the estimated number of its customers.

Since the above considerations are not conflict-free, there is no optimal policy that satisfies them all. It is left as an open problem to devise good allocation algorithms in a FOCS. We have designed a preliminary algorithm of CPU allocation that illustrates the issues discussed above. It is a multilevel feedback algorithm that accepts customer demands via the *GetCPU* service, and adjusts them by considerations of fairness and utilization efficiency. The algorithm is presented in stepwise approximations, each of which improves on these considerations at the expense of complexity and efficiency.

First Approximation:

The basic algorithm is a multi-priority, preempt–resume, which allocates CPU's based on the notion of urgency. It uses two parameters: a maximal allocation in order to avoid starvation of non-urgent demands, and a minimal allocation to prevent thrashing. These parameters are installation-dependent; they can be scalars or functions of the global demand.

The CPU host maintains a single queue of demands, ordered by their urgency level. The queue is scanned sequentially. Since there are multiple CPU's, several demands can be honored concurrently. A new demand preempts CPU's from demands of lower urgency that are currently granted CPU's. When the new demand consumes or releases its allocation, the preempted demands are granted back their CPU's, and the scanning of the queue resumes. If a demand is larger than the maximal parameter, only that much is allocated in a given scan of the queue, and the rest is deferred for the next scan.

Problem: A nonurgent demand may starve if more urgent demands arrive frequently.

Second Approximation:

The algorithm is extended to include multilevel queues of demands, each level being for a different range of urgency levels. For instance, there are three queues for *pressing, rush* and *normal* demands. The algorithm uses additional parameters for fairness. There is an upper limit on allocations per queue; demands are promoted to better positions in their queues or to higher queues if they are not satisfied within some time threshold.

The queues are scanned sequentially, level after level. Each of them is scanned sequentially and repetitively; demands in the queue are honored in a round-robin fashion. Newly-arrived urgent requests preempt less urgent ones, as before. A demand that is not fully satisfied in one scan, or one that stays in a lower queue for "long time" is promoted to a better position in its queue or in a higher queue. Likewise, frequent, urgent demands of one scheduler are demoted to lower queues. The parameters for promotion and demotion are also installation dependent.

Problem: Nonurgent demands may delay an urgent one for a long time, if the latter is not satisfied by its initial allocation. For instance, a demand larger than the maximal parameter of its urgency level receives an initial allocation and then waits until the scanning of all other queues completes and returns to its queue.

Third Approximation:

The algorithm is extended to include different groups of multilevel queues of demands. Each group is designated for a different magnitude of allocation so that urgent and non-urgent demands can be simultaneously satisfied. For instance, one group is of demands for "few" CPU's for "short time", one for few CPU's for "quite-long" time, and the other of any number of CPU's for "long" time. "Few", "short" and "long" are installation-dependent parameters.

The CPU host divides CPU's into several pools, one for each group. (In a single-CPU installation, the CPU will alternate among the groups.) The size of pools adjusts dynamically to the relative contention for them. Similar to the former approximation, there are several queues per pool based on urgency, which are scanned in a similar fashion as before. However, the number of queues, the maximal- and minimal-allocation parameters, and the method for promoting and demoting demands between queues can vary in each pool.

This algorithm allows concurrent scheduling of urgent and nonurgent work. Depending on the tuning of the parameters, short, urgent demands—such as to handle I/O interrupts—can be satisfied instantly without aborting normal work. The algorithm gives higher preference to a scheduler that controls an I/O-bound mix over one with a computation-bound mix, assuming the former requests fewer CPU's for shorter time. To simplify the execution of the algorithm, the CPU host devotes an activity per pool, which acts as the pool's allocator; another activity runs periodically to rearrange the pools.

Possible extensions:

The algorithm does not address all aspects of utilization efficiency and fairness. For instance, how can a selfish, utility-maximizing scheduler be truly prevented from overdemanding CPU's? On the other hand, how can a demand that *must* be satisfied indeed be satisfied, even if it is very large (e.g. an important simulation that requires *n* CPU's for few hours)? Several features can be used to support these requirements.

(1) The CPU host can charge for demands as well as for actual allocations. As mentioned above, higher prices can apply to more urgent requests. Charging for a demand should increase linearly with the amount requested, so that a scheduler that serves a large community of activities does not pay more per activity than a scheduler of a smaller community of activities.

(2) *GetCPU* can be augmented with a bidding parameter, to be used to upgrade the position of a demand in the queues.

(3) The CPU host can maintain periodic averages of CPU usage per customer give credits to customers with low averages. Credits can be used to factor a customer's demand and accordingly to position the demands in the queues. in the form that the customer's demand is factored by its relative CPU usage, and the demand positioned higher in the queues.

(4) Customers of the CPU host can be ranked according to their importance. Ranking can be altered dynamically by the System Administrator. Demands can be factored by the relative ranks of customers. So for example, the scheduler of the ED DBMS can be ranked highly with respect to getting one CPU, which implies that a CPU would be dedicated to that scheduler as long as it needs one.

4.2.7. The System Administrator

The System Administrator is the initial principal in the system. Its account is credited with an infinite budget. This fact makes the System Administrator a privileged user, but otherwise it does not need any privileged services from the Accountant. To create new principals, the System Administrator merely creates their accounts as the subaccounts of its own account. Therefore, it can always close these accounts. It can also credit any account or clear a balance by transferring funds from its own account.

The System Administrator is represented by a server which is the first owner of all system resources. Therefore, its right to dictate the primordial ordering of resource ownership derives from being their owner. It can dynamically change the ordering or dedicate resources to given servers by merely revoking the resources from their current owners and allocating them to the chosen servers. So, for instance, to let a given scheduler have a partition of *n* CPU's exclusively for *m* Time Units, the System Administrator's server would revoke *n* CPU's and allocate them to that scheduler through *Allocate*.

4.3. A Standard Interface

A standard interface is a set of conventions for services which a server might be obliged to use, as discussed in Chapter 3. A minimal standard interface is defined by the system designer. Each installation may set additional conventions for service interfaces, names, addresses, and for representations of entities such as bindings and permits. Such conventions simplify programming and improve the level of sharing between applications because they allow servers to select services without concerning themselves with the particular interfaces of the services. For example, a server that needs a "standard" window management service can choose anyone that conforms to the standard, without having to learn the service's particular interface in advance.

It should be emphasized that conventions are not mandatory: If a server provides a service of the standard interface which does not conform with the conventions, then the server may harm its customers. This situation is similar to providing a faulty service. For example, suppose the standard interface defines operation O to be provided by each owner of an activity. If the current server of activity A invokes O but the invocation fails, then the server can decide to refuse the service currently executed by A, and hence abort its computation. Likewise, if the owner provides O but with "nonstandard" parameters, then the owner may either perform the service wrongly or return an incorrect result, which again may cause the computation to abort in either case.

At the design level a standard interface contains three components:

(1) Names and parameters of services for specific service domains, e.g. for memory management.

(2) Names and parameters of services for exception handling.

(3) Specifications of where the bindings for these services are found.

At the implementation level, a standard interface specifies the data types of the parameters and in what registers they are passed.

We believe that a minimal standard interface should specify operations for four service domains:

(1) *Activity management*, to allow a server in the thread of an activity to pose scheduling requirements to the activity's scheduler.

(2) *Memory management*, so that a server can use a customer's buffer by invoking the customer's U-mgr or m-host.

(3) *Interactive I/O*, to allow different servers of an interactive computation to communicate with the user.

(4) *File I/O services*, so that multiple servers in the thread of an activity can share a common file to read service parameters and to write execution traces or error messages.

Appendix B presents an example standard interface which lists the operations for activity management and memory management. Only a few standard operations are defined for each service domain. The other two service domains require simple operations on byte streams, such as *Open, Close, Read,* and *Write*. For simplicity, a single binding per service domain references all the operations defined for that domain. The bindings are posted in the ACD of each activity. In this way, a server that wants to invoke *Block*, for example, simply uses binding #1 of the current activity. It is similarly simple for the *Invoke* service to locate the binding. Bindings for memory management services are not included in the ACD — they should be transferred together with buffer permits.

The example standard interface lists two additional operations. *TerminateBinding* is a generic operation to be included in every binding. It can be invoked in order to terminate a binding by the binding's holder or by whoever terminates the holder. This operation allows the service provider to properly clean up the state associated with the binding's holder.

ExcptHandler is a generic operation to be invoked to handle any exception, including traps. We have decided in favor of a single handler for all exceptions unlike Unix [109] or Charlotte [16], each of which uses a vector of handlers indexed by exception type. This decision is motivated to support virtually any exception type. The same handler is invoked to handle an exception related to a server controlled by the invokee, e.g. when the invokee is the U-mgr of a faulty server that does not have an exception handler. For simplicity, the address of a handler or a binding thereof is prescribed by every server in a well-defined location in its Space. It can be easily located by the CPU upon exception, and can be dynamically changed by the server. The exact location is defined at the implementation level.

4.4. System Provided Services: A Customized OS

This section discusses services provided by an example customized OS (COS). The services are intended to support the basic computational needs of applications. They are not mandatory and can be provided by other applications as well. The focus of this discussion is on services for memory management, services to match service providers with their customers, and on language support services.

4.4.1. Memory Management Services

In this section we assume that the COS includes a Universe manager, and that a physical m-host is included in the OSB or the COS. The following issues of memory management are presented:

- A structure of the Universe of the COS. This structure allows efficient and flexible sharing of both virtual and physical spaces among the servers stored in the Universe.
- An interface between the m-host and U-mgrs. This interface allows the m-host to perform address translation efficiently. It lets a U-mgr manipulate the structure and the mapping of its Universe dynamically and efficiently.
- Services provided by an m-host and a U-mgr to their customers.

Universe Structure

As discussed in Chapter 3, a Universe is composed of Spaces, and a server can be stored in each Space. In order to accommodate efficient sharing between servers, a Space is divided into segments. A **segment** is a virtual address space which is separately mapped to physical memory. By sharing a segment, servers in one application can share code or data structures global to the application more efficiently than doing so via permits. Segment sharing, however, raises a problem: a segment shared by several Spaces would have to be identically numbered in each Space, or else be addressed via an extra level of indirection as in Multics [23]. To avoid the complexity and the inefficiency that this problem incurs, the mapping information of a segment is shared, *not* the segment itself. Namely, segments in different Spaces may share the same memory map. (For simplicity of presentation, we continue to refer to these segments as *shared segments*.) This form of sharing does not solve the problem of a reference stored in a shared segment, since the reference needs to be interpreted differently for each Space sharing that segment. We will discuss in Chapter 5 simple solutions to this problem, e.g. using base registers for those segments which are referenced through shared segments.

It would be undesirable to require that segments will be mapped to a contiguous region of physical memory, for reasons of memory utilization and of allocation complexity. Therefore, a segment is divided into **pages**, each of which is mapped to a frame. Though each Universe in the system may support different page sizes, pages must be mapped to the m-host-dependent frame size(s). For protection reasons, access rights are associated with each segment, not with each page. Thus, different servers sharing code and structures via shared segments may have different rights to access them. To protect the other servers, a server cannot extend the access rights of a shared segment.

The specific mapping structure that supports sharing of segments and frames depends on the mapping scheme employed by the m-host. There are two basic mapping schemes: A direct-mapping, index-based scheme as in the VAX® [3], and an inverted-mapping scheme, as in the IBM RT/PC [38] or the HP Spectrum [136]. In the former scheme, a per-segment page table specifies the frame each page is mapped to. A frame can be shared by having different pages of different segments pointing at the frame. The U-mgr maintains a descriptor for each segment which includes a pointer to the segment's page table and its size. A virtual address space is thus shared by having multiple segment descriptors pointing to the same page table.

In an inverted-mapping scheme, a single frame table is maintained by the m-host. Each frame has a descriptor which specifies the pages mapped to that frame. To support sharing of a frame by multiple segments, a frame descriptor indicates two pairs of (group id, page number). The first pair allows multiple servers to access a given frame by each having a segment marked with that group id. (A segment's group id is specified in the segment's descriptor.) The second pair allows the segments of the U-mgr to share every frame it owns with its customers; moreover, the frames can be referenced with virtual addresses. The m-host allows a U-mgr to manipulate the descriptors of the frames it owns. In the rest of this section we discuss a direct-mapping, index-based scheme.

In order to allow efficient address translation by the m-host, the structural and mapping information of a Universe is grouped into several predefined, m-host-dependent structures. These structures are contained in a single segment of the Universe, called its **base segment**. Since these structures are also manipulated by the m-host, it is henceforth assumed for simplicity of design that a Universe is mapped to a single physical memory. The mapping of the base segment to physical memory should be known to the m-host during translation, so that virtual addresses relevant to the base segment (e.g. the address of a given structure) can be translated by the m-host too. Therefore, the location of the maps is provided by the manager at Universe-creation time. The base segment is an "ordinary" segment of a Space. Since a U-mgr can store itself in that Space, it can modify the structural and mapping information contained in the base segment via virtual addresses. Hence, there is no need for a privileged mode or for an extra translation mechanism.

The base segment includes a *Spaces Table*, which lists all the Spaces in a Universe (see Figure 4-6). An entry contains information about a Space, including a descriptor for each segment of

®VAX is a registered trademark of Digital Equipment Corporation.

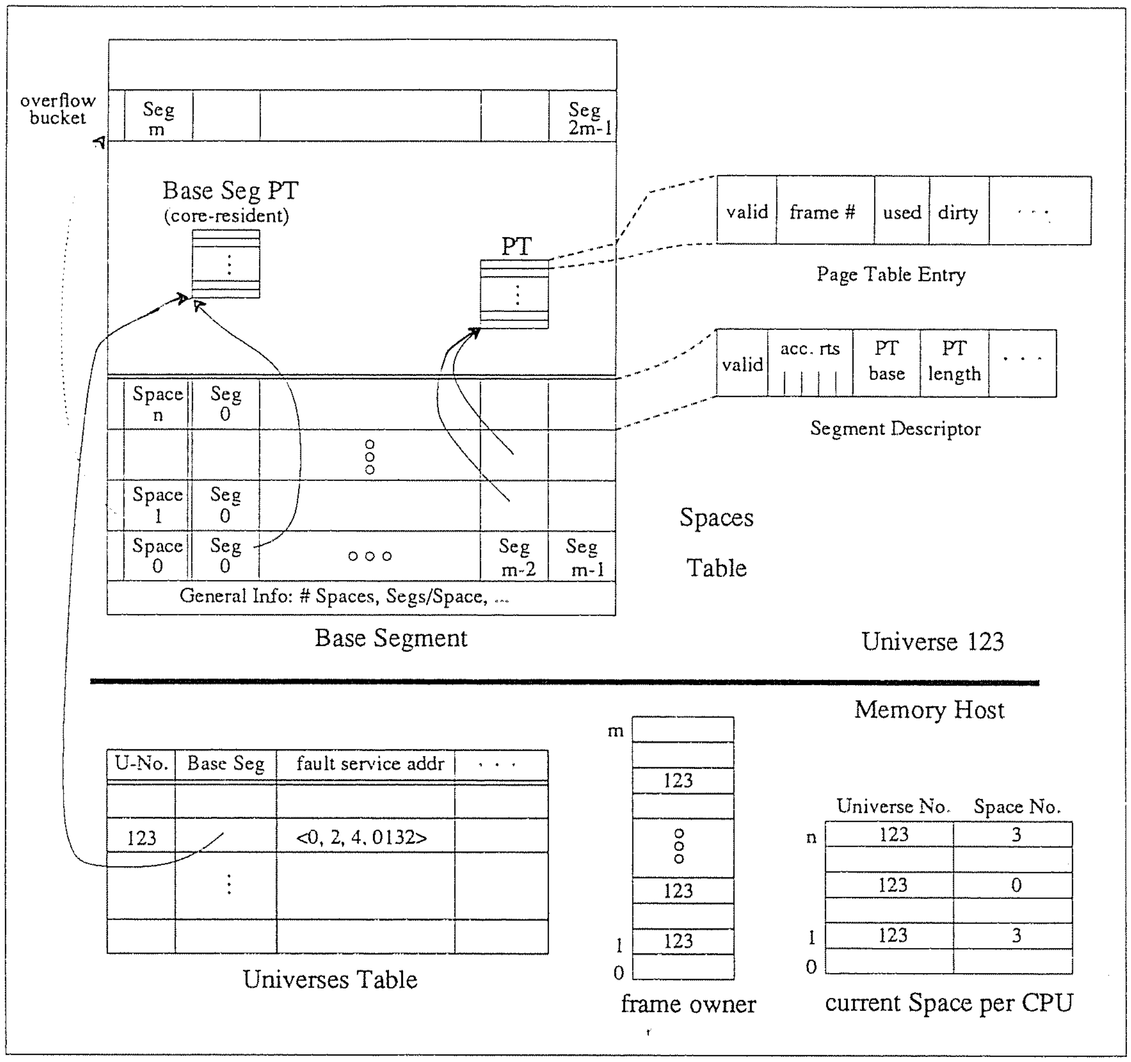

Figure 4-6: Universe Structure and Mapping in a Direct-Mapping Scheme

the Space. Since entries are of fixed size, the m-host can locate the necessary information of a referenced Space by using the Space number as an index in the table and the segment number as an index in the entry. Notice that this feature does not limit the number of segments per Space, because an entry can be extended with "overflow buckets" of segment descriptors. This extension, however, may incur translation delay if overflow buckets must be searcheds! In addition to the fields shown in Figure 4-6, a segment descriptor contains control information used by the U-mgr, such as sharing information or an indication to copy the segment when a shared segment is modified (that is, a copy-on-write indication).

The base segment also contains the page tables, including its own page table (denoted BSPT). The BSPT is core resident, in order to avoid cyclic page faults during address translation.

(Assuming a contiguous table, a manager therefore must own at least one *contiguous* physical memory area of the table's size.) A page table entry (PTE) is similar to its counterpart in a conventional architecture, except for the lack of access rights. The validity field of a PTE can indicate *Load-on-Demand*, which means that the page has yet to be loaded into memory for the first time. This indicator implies that upon a page fault the U-mgr should consult the particular loading service for the server stored in the relevant Space.

The m-host keeps a *Universes Table*, with an entry for each Universe mapped to this physical memory. An entry includes the physical address of a BSPT and the virtual address of a fault handler.

Address Translation

The following example illustrates how the m-host translates virtual-to-physical addresses. Suppose the current Space is <123, 3>, that is, Space 3 in Universe 123, and the referenced address is <4, 2, 32>. Translation takes the following steps.

(1) The m-host locates the BSPT of Universe 123.

(2) The virtual address of the entry of Space 3 in the Spaces Table is calculated; using the BSPT, this address is translated and the entry is fetched from physical memory.

(3) The descriptor of segment 4 is located.

(4) The entry of page 2 in the segment's page table is similarly calculated and fetched.

(5) Finally, 32 is added to the frame number specified in the PTE to yield the physical address.

Notice that steps (1) and (2) need not be repeated at every address translation. For efficiency, the m-host performs these steps when Space <123, 3> becomes the current Space of a given CPU, and keeps the structures in a cache. Similarly, step (3) can be omitted on repetitive references to the segment 4. Consequently, a U-mgr should notify the m-host whenever it changes any of these entries, so that they are removed from the cache. If the m-host maintains a cache of recent translations, then step (4) can also be omitted if the PTE is found in the cache.

During the translation process the m-host verifies the validity of the Space, segment and page numbers, as well as the validity of their entries in the respective tables. It verifies the validity of the access by checking that every frame accessed during the translation, as well as the referenced frame, is owned by the U-mgr. It also checks access eligibility, namely, that the intended access operation does not conflict with the segment's access rights. If any of these checks fails due to a bad pointer, to invalid information, or to a page fault, the U-mgr's handler is invoked by the m-host.

Universe Creation

The m-host accommodates the creation of new Universes. A new Universe is created by a server stored in an existing Universe. The following is a possible scenario of Universe creation. The creator obtains frames, at least as many as necessary to store the portion of the base segment that has to be core-resident. It then defines the structure of the new Universe in one of its segments, and invokes a service of the m-host to declare a new Universe. At that point the server named as the

Universe manager is allocated these frames and is copied by the m-host from its "current" Universe into the new Universe. The manager further defines the structure of the Universe and initializes the mapping information. Note that the creator does not disappear. It can serve for recovery of that Universe. For example, if the manager fails to handle a fault in its Universe because the fault-handling service itself is paged out, then a predesignated service of the creator may be invoked to recover that Universe.

Services of the Universe Manager

We turn now to discuss the interface of a U-mgr with its customers. (The service interface of the U-mgr of the COS is presented in Appendix C.1.) A new Space is allocated via the *New* service. The U-mgr assigns a unique Space number for the new Space, which consists of the Universe and the m-host numbers, an index into the Spaces Table, and a random identifier. These components allow the CPU to direct references to that m-host and allow the m-host to perform address translation. As mentioned earlier, for all aspects of memory management and addressing, a server is identified with the id of the Space in which the server is stored.

The creator of a new Space indicates the initial structure of the Space in terms of how many segments it includes, the size of each segment and its access rights. This information is needed by the U-mgr to set the respective tables appropriately. The structure can be later modified by the server stored in the Space or by the creator. A creator of a new Space specifies sharing information, that is, whether a new segment is a copied from another existing segment S_g or whether the segment shares the page table of S_g. For protection reasons, the creator should either own the Space in which S_g is, or else it should have an adequate permit to S_g. For the simplicity of the design, shared segments must be in the same Universe. The creator becomes the owner of the new Space, which enables the creator to load a new server and initialize its state. At Space creation, as well as at segment creation and at memory copy, a customer can specify that the copy operation is delayed until the source or the target is modified (*copy-on-write*). This feature accommodates fast Space/segment creation and efficient exchange of buffers between servers.

Loading a server into the new Space is the responsibility of the Space's creator. Suppose the U-mgr employs a pure demand-paging policy. Upon a page fault, the page needs to be copied from an auxiliary storage space into memory. However, the server's image might reside at various storage spaces, or be maintained by various file servers. Hence, the U-mgr needs to invoke some service which can load the page into memory. This service is provided by the creator or by some file server to which the creator is bound. The interface of this service is predefined by the U-mgr. The creator has to hand a handle for the service to the U-mgr; the handle includes a binding for that service and a permit for the server's image file. (An analogous method is used in the Accent [56] and Mach [134] systems.) For efficiency, the U-mgr maintains a *swap area*, into which it copies pages upon replacement. Hence, a subsequent fault on that page is handled by the U-mgr.

The U-mgr supports the *Pin* and *UnPin* operations of the standard interface. In addition, it lets a customer request that a given region of a Space that it owns be fixed in physical memory. This region then remains core-resident as long as contention for physical memory permits. *MemCopy* is used to copy data across Space boundaries. For protection reasons, the invoker of this service must own the Spaces from and into which the copying is performed, or must present valid permits to access those areas.

The permit formats that the U-mgr supports are depicted in Figure 4-7. Like a binding, a permit consists of two lists. One is presented by a server that wants to access the referenced area (Server S_u), and it consists of a target address and a key. The other list is stored at the target address, which is in the Space of the permit issuer (Server S_i); it consists of a lock and a description of the referenced area. That area can be in the issuer's Space or in a different Space S_b owned by the issuer. The U-mgr verifies that the key matches the lock, and that S_i owns S_b. We chose these formats because they allow a server to easily issue permits to Spaces it owns and to pass them to other servers without needing the U-mgr's help. The server can easily invalidate each permit separately. Furthermore, any permit issued by a given server is automatically invalidated when the server

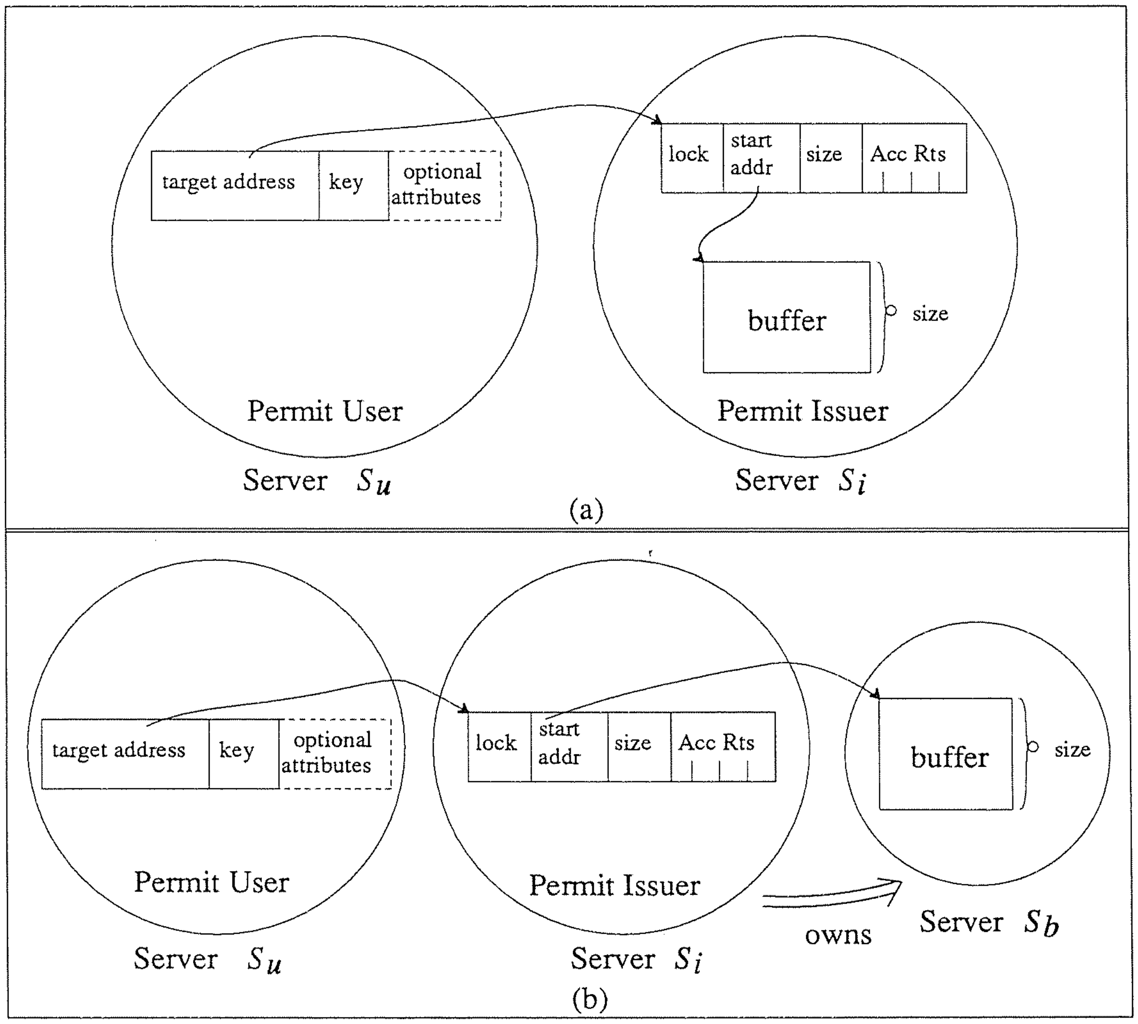

Figure 4-7: Permits to access virtual memory

terminates, or when it loses ownership of the Space referenced in the permit.

Access across different Spaces raises a problem: what if Space S_b is not in the same Universe as the others mentioned above? The solution adopted is that the U-mgr of that Space is invoked to verify the ownership of S_i over S_b. Likewise, if S_i and S_u are in different Universes, then the U-mgr of S_i is invoked to verify the permit. If the operation is a memory copy, then the invocation and the copy are performed via the m-host. The protocol of the invocations is a convention defined by the m-host. This protocol can be more efficient if (1) Space-ownership information is visible to the m-host, e.g. by indicating ownership in the Spaces table, and (2) all Universes mapped to this physical memory support standard permit formats. In such a case, the m-host can perform the verification and the copy without the intervention of the target U-mgr, a process which is analogous to address translation. In a more complicated scenario, the Universes are mapped to different physical memories. In this case, the intervention of several m-hosts is needed. Such cases require an elaborate protocol between various U-mgrs and m-hosts, which we do not detail here.

To support recovery and debugging, the U-mgr allows a Space owner to *Freeze* the Space in order to check or modify its contents and to take a *SnapShot* of its state. A "frozen" Space can be later reinstated via the *UnFreeze* service, or else it can be *Dispose*d. Furthermore, if a Space is disposed of by the server stored in it, then the U-mgr notifies the "guardian" of that Space, if any, so that the latter can debug or recover it. The guardian is simply a service associated with that Space, a binding for which is handed to the U-mgr at Space creation.

4.4.2. Matchmaking Services

A matchmaker is used as a repository of bindings. Service providers deposit bindings for their advertised services, associated with descriptive names and types. A server can associate a resource name with a binding, indicating that the relevant service is used for accessing that resource. (This feature would require the matchmaker to define some standard resource names.) Service users look up the services they need and obtain bindings for them. A service provider may leave a "reincarnation order", which will be used by the matchmaker to regenerate the server if its services are requested after it has terminated. Applications may have private matchmakers to bind their servers to each other. In this section we outline the services of the COS *Matchmaker*, which serves as a centralized name clearinghouse for servers in different applications. The interface of the Matchmaker's services is summarized in Appendix C.2.

A service provider can announce the service and furnish a binding for it via *Expose*. A service is declared with a unique (server name, service name) pair. Servers must choose unique names for themselves so that potential customers are not mislead. Disputes about "name rights" should be resolved transparently to the Matchmaker.[8] For simplicity, the Matchmaker supports a flat name space without aliasing. A server that wants to *Expose* a service under several different server or

[8]Note: In the particular event that a server is restarted and tries to *Expose* the same services as it did before termination, *Expose* is not rejected because the Matchmaker recognizes that the new server is a reincarnation of the former one, as explained shortly.

service names can do so by exposing it with different bindings. These bindings are considered by the Matchmaker independently of each other. This decision is based on experience with Charlotte, in which the *SwitchBoard* name server supports aliasing [54]. The designers concluded that the infrequent use of aliasing is not worth the complexity of supporting it.

A type is associated with a binding. The type can be one of a predefined list of types suggested by the Matchmaker or an application-defined type. A type can indicate a resource name. For simplicity, the Matchmaker supports types merely as (sub)names, used to assist service users in locating a desired service. Hence the Matchmaker does not verify types.

A service user obtains a binding for the service via *Bind*. For the sake of convenience, a service can be located by using a list of names or types to describe the service. This feature is useful when a service provider advertises the service under different names or types. For simplicity, the customer may omit the server's name from the *Bind* request. This feature is useful when an identical service is provided by a community of servers, all of which adhere to the same interface, or when a customer expects only one server to provide that service. Notice that the Matchmaker does not check the eligibility of the invoker to become a customer of the requested service. This task is left to the service provider.

For storage efficiency, the Matchmaker occasionally archives bindings that were not requested for long a time or removes them if their providers have terminated. Bindings are also removed upon request. A service provider can announce that it no longer wants to provide a given service (or *all* services) by invoking the *Close* service. The Matchmaker removes the relevant binding(s) from its store. This operation, however, cannot invalidate copies of the bindings held by other customers or matchmakers. To disable invocations of the service altogether, the provider should remove the binding's origin or modify its lock. A server may *Close* services exposed by another server only if the latter has terminated. This feature allows the creator of a server or its U-mgr to remove the server's bindings upon its abrupt termination.

A customerless service provider may want to disappear until its services are requested. For instance, a compiler used infrequently can terminate until some server needs its service. It can leave a reincarnation order at the Matchmaker via *Revive*, or another server can do so on its behalf. The invoker of *Revive* specifies how to reload the server and, if necessary, how to reload its loader, recursively. This feature allows a hierarchy of servers to disappear when they are not needed, and to be reactivated on demand. When a server terminates, the Matchmaker does not remove the bindings that the server has deposited, but rather freezes them. When one of these bindings is requested by a customer, the Matchmaker asks the server's loader to load it. The Matchmaker activates the bindings by placing the new id of the revived server in them. Should the server be revived for another reason, e.g. by being triggered by a user command, it can invoke *IamAlive* to activate its bindings and to remove a former *Revive* order. For protection reasons, the latter server has to authenticate itself as the former one via a *secret id* which has been specified in a former invocation of *Revive*.

4.4.3. Other Services

The COS includes a battery of additional servers which support activity and server management, disk storage, filing, network communication, and debugging services. We now present a preliminary design of some of these servers.

Server Management

Servers are created through a *server manager* (denoted S-mgr). The creator specifies where the image of the new server is. The image should contain a descriptor in a format defined by the S-mgr, which tells the structure of the server and its initial service. This information is passed to the U-mgr so that it may structure the Space into which the server is loaded by the S-mgr. The S-mgr creates an activity to invoke the initial service of the new server, which initializes the server. A set of bindings is passed to this service, including a binding to a matchmaker, to the server's U-mgr, and to the S-mgr.

It is left to the creator to decide and implement any resource-inheritance policy. Accordingly, the creator asks the hosts whose resources it owns to allocate the resources to the new server, or to share them with the new server. For example, suppose Server *S* owns file *f*, whose host is a file server *FS* (see Figure 4-8a); *S* asks the S-mgr to fork a new server (Figure 4-8b); S-mgr creates *S'* and returns its id to *S*; finally, *S* tells *FS* to allocate *f* to *S'* (Figure 4-8c). This scheme puts the burden of birth-inheritance announcements on each server. A simpler scheme is to let the S-mgr announce the birth of *S'* to all interested servers, e.g. the above *FS*. However, the latter scheme requires dictating conventions regarding the methods by which such servers would tell the S-mgr that they are interested in being notified of new offspring servers of *S*.

The S-mgr functions as a "watchdog" over the servers that it manages. As discussed earlier, a server can specify a list of servers to be watched. It can choose to be blocked until any of these servers terminates, or to be notified of their termination via a "reply" service.

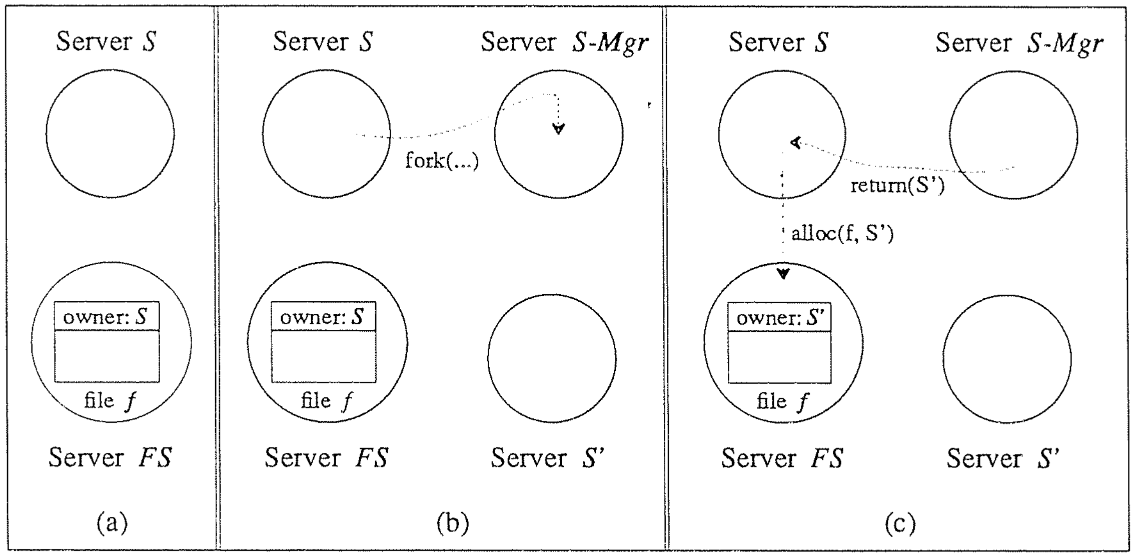

Figure 4-8: Birth-Inheritance of a Resource — an Example

Disk Services

A disk host is included in the COS to provide low-level disk management services. It divides the disk space into *blocks*, which are the units of allocation. For simplicity of design, the disk host allocates blocks by their physical addresses; it does not maintain mapping of customer-relative block addresses to disk addresses. For reasons of access efficiency, a customer may request that an allocated region be of contiguous blocks. It may also indicate the address at which the region should start or end, so that the region can be coalesced with a region the customer already owns. The server ids of the owners of each block are stored in the block itself, so ownership verification does not require an extra disk access.

To allow a customer some flexibility in scheduling disk accesses, the customer can specify that a *Read* or a *Write* request be nonblocking. In such a case, the disk host acknowledges the access completion by invoking a predesignated handler supplied by the customer. Moreover, a customer can indicate that a *Read* or a *Write* request be carried out immediately. The disk host enqueues such a request in a queue of urgent accesses which are performed before ordinary ones. To prevent customers from abusing this feature, the disk host charges more for it. In addition, non-urgent requests are promoted to the urgent queue after some time threshold to prevent their starvation.

Filing Services

Suppose that a file server in the COS provides Unix-like file services. When a file is created, the principal whose accounting permit the creator has supplied becomes the file's owner. Any server who uses that principal's account can access the file with the same rights. There are two alternative methods to represent file permits. First, the file server associates a key with each file. The file owner creates permits by using a one-way function in a manner similar to producing accounting permits. This method is simple and efficient but does not support invalidation of individual permits to a given file. Second, a permit's origin is created for each permit issued by the file owner, to be used to verify these permits — similar to the method described earlier regarding memory access permits. However, a permit's origin cannot be stored in the address space of its issuer, since a permit might be used after the issuer has terminated. Therefore, there is provision to store permits in "control" files, e.g. file "xx.prmt". This method solves the problem of the former method at the expense of extra disk accesses to write and to validate permits.

When a file is opened, the file server returns an access permit, which is analogous to a file descriptor returned by the Unix kernel. The customer can share the file with other servers by transferring the permit to them. Consequently, when a customer dies, the file server does not automatically close the files that were opened by that customer; the files can be closed by whoever holds a permit to them. Files that are not accessed for more than a predefined period of time are closed by the file server so that they are not left dangling. The file server writes in a closed file the file's current state, so if there are still valid permits to the file, the file can be reopened transparently to its users.

Network Communication

A single machine can be connected to other machines via one or more networks. Each network connection is controlled by a network-communication host. Assume that one such host is included in the COS. The resource is the communication bandwidth, and the unit of allocation is a time slot to send packets. Packets are received by the host whenever they arrive, no matter who owns the resource at that time. The bandwidth is allocated to senders employing a dynamic precedence scheme similar to the one used in CPU allocation.

The host provides *Send* and *Receive* services on byte streams. Reliability and flow control are relegated to communication servers implementing higher-level protocols. Each server defines a *port* into which received messages are queued. The server associates a *logical name* with a port, so that remote peers do not have to rely on special handshaking protocols in order to locate the port. The port owner supplies access permits for buffers where messages can be placed. *Send* and *Receive* are nonblocking. At a completion of *Send* or *Receive*, the host tells the customer by invoking a predesignated handler of the customer.

4.4.4. Language Extensions, Compiler and Runtime Services

This section proposes new programming language features aimed to simplify the writing of servers. It is shown how the features spare programmers the burdens of declaring bindings, allocating resources to activities, maintaining bookkeeping of such allocations, synchronizing activities, accessing shared structures and verifying service invocations. We describe the support functions of a hypothetical compiler and a runtime package which assist servers in obtaining bindings and invoking services efficiently. The discussion in this section follows an example server that provides calendar management services. Excerpts of the server's program are listed in Figure 4-9.

A server declares the services it uses and provides in the clauses *use* and *provide* (lines 4-9). These new clauses are reminiscent of the *export* and *import* clauses of modular programming languages such as Ada and Modula [142]. Each service is assigned to a variable which can be used in references to the service. For instance, at program initialization, the server calls library routines (lines 118-119) to *Expose* its service and *Bind* to the file server's "Sequential files" service. These library routines would invoke the Matchmaker's services and check for correctness of the result. The variable *FS* (line 5) is of a first-class type *binding*, and is used to invoke the appropriate service (line 71). Likewise *S1* represents a binding's origin. Bindings and permits can also be created "on the fly," using built-in functions such as *MakeBinding* and *MakePermit* (line 71).

A service operation is defined as an *entry* procedure (lines 63-84) which are similar to those in languages that support remote invocations [114]. It is listed in the *provide* list (line 9) so that the compiler can include it in the binding's origin. The compiler inserts code at the beginning of an entry procedure to obtain internal structures needed at each invocation, e.g. a stack frame. Code is inserted at the end of an entry procedure to release these structures. Likewise, the compiler generates code to save registers prior to a service invocation, to restore them thereafter, and to check for any exceptional returned values.

To mutually exclude accesses to data structures shared by multiple activities, one can embed these accesses in a monitor, e.g. in a Modula-like *interface module* [141]. Alternatively, one may

```
1     module main;
2     import
3          Advertise, Locate;
4     use
5          FS = "Universal_FS" . "Sequential files" :
6                      Open, Close, Read, Write, Seek;
7     provide
8          S1 = "CAL" . "Calendar Mgmt" :
9                      SetAppt, Cancel, Enquire;
15    type
16    #include "/usr/ios/includes/fs/sf_types.h"
17    #include "../includes/cal_types.h"
21    var
22         exclusive buff : array    ...   of char;
23         file-permit : record
                . . .
27           end file-permit;
28         resident info : array    ...   of  ... ;
41    module Cal_services;
42      export  SetAppt, Cancel, Enquire;
43      import  MakePermit;
63      entry Enquire ( ... ) returns Appt_Descriptor;
64        begin
70      with  buff  do
71         if FS.Read (file_permit, MakePermit(buff)) ≠ OK
72             then - - - -
75             else  - - - -
79         endif;
80      od;
81         <<   - - - -
82              - - - -   >>
84      end Enquire;
99    end Cal_services;
116   begin main
117        Initialize ( ... );
118        Locate (FS);
119        Advertise (S1);
120   end main.
```

Figure 4-9: A server providing calendar management services

use compiler-provided code to accomplish this protection. For instance, a variable can be defined as *exclusive* (line 22), Uses of the variable that require concurrency control can be embedded in a *with* statement (lines 70-80). The compiler would generate code at the beginning of such a statement to check whether the variable is used by another activity with a conflicting access right. If so the activity's scheduler's *Block* service is invoked. Likewise, code would be added at the end of the *with* statement to check if another activity is blocked awaiting the variable, and if so, to *UnBlock* that activity. This code would also handle the bookkeeping of structures occupied by an activity so that the runtime package can recover the structures when the activity terminates. Similarly, declaring a procedure or a variable as *resident* (line 28) tells the compiler to generate invocations to *Pin* them in core.

The language simplifies the mechanism for protecting short critical sections. The programmer can define a critical section by bracketing a sequence of statements (lines 81-82). The compiler then inserts instructions to postpone quantum expiration during the critical section. If the the compiler suspects that the execution of the section will be lengthy, it adds invocations of the activity's scheduler's *Prio* service to increase the activity's precedence before the section, and to reduce it thereafter.

To further simplify programming, the compiler automatically includes bindings for installation-default or language-default services. For example, when the compiler detects calls to external *Read* or *Send* operations which are not declared in any *use* clause, it generates the *use* declarations of the appropriate default services as well as the necessary *Bind* requests. These tasks can alternatively be relegated to an interface-definition language preprocessor like the Accent Matchmaker [67]. Based on operation names detected in a program, it can insert the necessary language constructs for binding creation, invocation, protection, concurrency, and exception handling.

A server might want to provide different services by using various subsets of a given set of operations. For example, it might want to declare a service for restricted customers from a restricted subset of the operations. Likewise, a service provider might want to enforce certain rules for the order of invocations, e.g. that a *Read* operation is never invoked before *Open*. Both requirements can be supported by extending the notion of *Capability Managers* of Kieburtz and Silberschatz [77,78]. A *Services Manager* is then included in the runtime support package. It *Exposes* the services declared by the programmer and verifies the adequate order of invocations. The manager defines new services which extend the former ones with additional operations such as operations for debugging. It inserts an initialization and a termination operation into services, and performs them.

It is the role of the runtime-support package to implement language-based rules of inheritance. Therefore, when a server is created, the runtime-support package can pass segment-sharing information to the U-mgr and bindings to the new server. In this way, different inheritance philosophies such as those of Smalltalk [58] and Emerald [28] can presumably be concurrently supported in the system.

One notorious problem with monitors is that they impose low-level scheduling decisions which can conflict with those of higher-level schedulers [73]. This is especially a problem in a fully open system, where a monitor can be used by multiple activities scheduled by different schedulers. A possible solution is to let a scheduler specify the precedence of a dispatched activity; the precedence can be discovered via the *Check* service (see Appendix A.1). A compiler inserts code into entry procedures of a monitor to find whether the invoking activity has precedence over the one that currently occupies the monitor. If so, the latter activity would be rolled back, or be switched-to in order to leave the monitor. Likewise, the declaration statement of a monitor can specify a precedence level as in Modula, which the compiler translates into invocations of *Prio* for each activity entering the monitor.

4.5. Example Revisited

We return to the example of the ED DBMS presented in §3.6.5, discussing it at the design level. Each of the major specific requirements of ED is discussed in turn, and it is shown how ED meets its requirements. This discussion argues that the particular service needs of database systems can indeed be satisfied in a FOCS.

Memory Management

The Universe of ED is created as a segment of a server running in the Universe of a Unix-like system. This server acquires a partition of physical memory in one or more contiguous region(s) through its U-mgr. It then creates a new Universe into which it is loaded to become the manager. The ED U-mgr then creates the permanent servers of ED (see Figure 3-11), assigns them to Spaces, and creates their initial bindings.

Suppose that for the purpose of efficient sharing between ED servers the structure of the Universe is similar to the Universe of the COS. Data structures shared by all or most of ED servers, such as the buffer pool, are mapped into one segment. This segment is added to the Space of every ED server that requests access to it. Since the shared segment has a single page table, any modification of a buffer or its mapping to physical memory is visible to all ED servers. This method of sharing extends other proposals based on segmented virtual memory [17, 138]. If the ED U-mgr owns a sufficient amount of physical memory, the entire segment is core resident. In addition, ED servers may indicate to their U-mgr which structures or code to pin in core.

Buffer Management

The ED U-mgr serves also as a buffer manager in order to efficiently coordinate memory and buffering services. It acts as a monitor for buffer allocation and access synchronization. Though each ED server can access the entire buffer pool directly, for integrity purposes a server must be allocated a buffer before accessing one. When an ED server makes an allocation request, it specifies a buffer size, the urgency of the demand, and indicates whether the buffer should be mapped to contiguous physical memory. (This feature is important only when the disk host uses physical addresses to access data and it does not support data chaining.) If the buffer manager cannot map the allocated buffer to contiguous physical memory and the urgency is high, then it preempts frames from other servers until a contiguous region is formed. If the urgency is low, then the buffer manager invokes the scheduler's *Block* service to suspend the current activity. The sizes of the buffer pool or of individual buffers need not be "tuned" in advance to any anticipated demand level. The buffer pool can grow dynamically in virtual memory, and likewise in physical memory if the ED U-mgr owns a sufficient number of frames. This property allows the U-mgr to support sophisticated buffer-allocation algorithms such as DBMin [43]

When a buffer is allocated, its address in the shared segment is returned. The allocated server can share the buffer with other servers by passing a pointer to them. This method eliminates the inefficiency problem of transferring data between the components of a DBMS in a conventional system [125]. No buffer permits are needed since the ED servers trust each other. If it is possible that different activities would attempt to read and modify the same buffer concurrently, then they need to acquire locks through the buffer manager.

It is important for every DBMS to preserve logical dependencies between records, e.g. between update records and their relevant log records in a write-ahead-log protocol [61]. These dependencies imply the order in which the buffers that store the records should be written to disk. Hence, ED servers can specify dependencies between buffers to the buffer manager. When a buffer fills, or a buffer needs to be replaced, or a transaction has to commit immediately, the buffer manager schedules disk requests in the order of the dependency graph rooted at the focal buffer. In addition, ED servers specify which buffer updates should survive crashes. Accordingly, the buffer manager occasionally flushes modified buffers to disk.

Disk Access

Similar to the COS disk host described earlier, the ED disk host lets its customers choose between blocking and nonblocking *Read* and *Write* requests. An ED server can specify the order of its *Write* requests. Requests with specified numbers are performed in the sorted order of their numbers (per customer). *Write* requests with no numbers are performed in any order based on the disk-head scheduling algorithm of the host. Optionally, a *Write* request can supply a list of blocks, which are written in that order. This feature is used mainly by the buffer manager. If a request is made by an activity with a high precedence, e.g. when an immediate commit is required, then the disk host issues the access immediately.

Query Processing

ED compiles a query on demand and generates a "query module" for it. Two important issues are: how efficiently can a query module be linked to other ED servers, and how can access methods be shared by servers executing queries? These questions are of common concern to database systems [37, 124]. A segmented-memory architecture simplifies this task: Since addresses are segment-relative, no address reconciliation is needed at load time as long as the segments of a query module are placed in fixed positions in all Spaces. Therefore in ED a query is compiled into two segments. A module is loaded into an existing frame of a server (see Figure 4-10a), which consists of data structures, bindings, and the mechanism necessary to execute a query. As illustrated in Figure 4-10b, each query module consists of segments 2 and 3 containing the query's specific access methods and data structures. At execution time, each query module is loaded into a new replica of the "query executer". However, except for the query-module segments, all other segments of the query executers (e.g. servers S and R in Figure 4-10b) are not duplicated — they share the same virtual and physical address spaces. This design supports the loading of queries in a simple way.

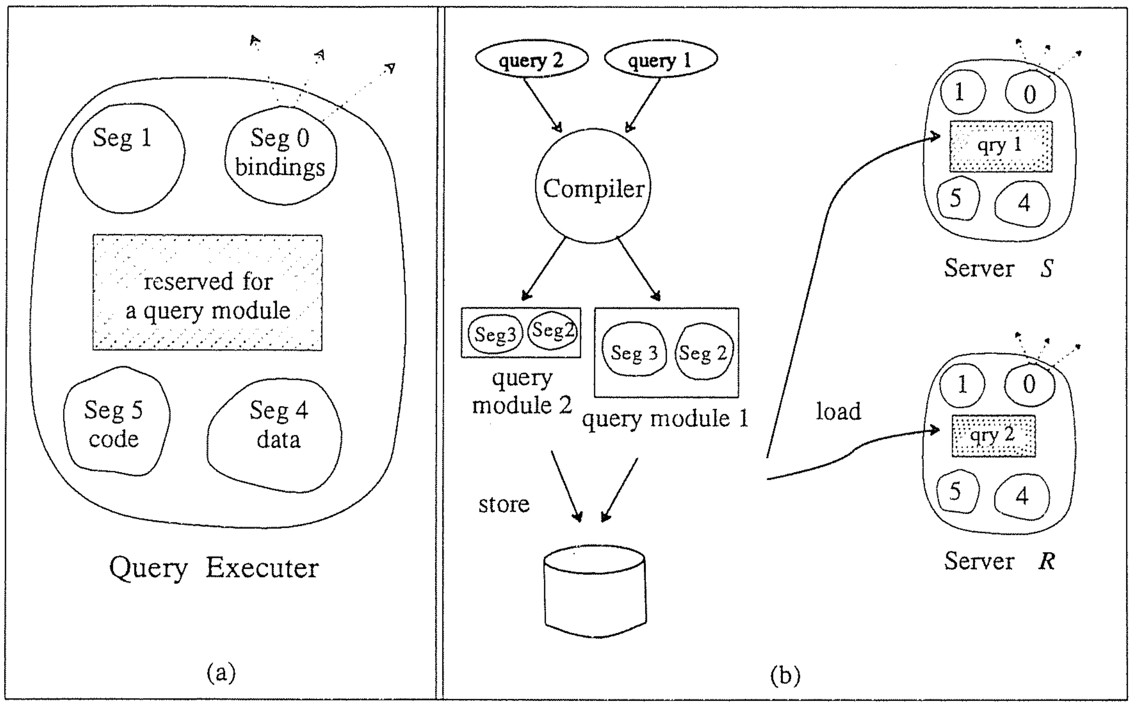

Figure 4-10: Compiled queries in ED

Scheduling Activities

Suppose that ED prefers non-preemptive query scheduling, allowing queries to block only when awaiting data. Since the customers of ED issue queries via activities controlled by different schedulers, ED cannot impose a non-preemption policy on them. Instead, the ED Query Interface Server or the Transaction Manager asks the scheduler of each customer's activity to *Block* it. Then another activity is selected from a pool of idle activities controlled by the scheduler of ED to perform the query. The selected activity is notified of its task through a work queue shared by ED servers. When the activity completes its task, it makes a request to *UnBlock* the customer's activity and returns to the pool.

Security

ED maintains its own object-ownership information. Therefore, when an object is created, it becomes owned by the principal that owns the customer's account. The creator may transfer access permits or ownership to other servers or principals. Upon access, the Security Manager verifies that the requester either represents that principal, or that it has a valid permit to access the object.

Upcalls

Suppose that a customer of ED prefers to refine a query while it is processed, for instance because of certain data dependencies discovered dynamically, or because the query requires specific operations on an abstract data type not supported by ED. The customer can specify a procedure to be consulted during query processing by providing a binding for the procedure. Invoking a customer's procedure is unsafe [127], and contemporary DBMS's have not found a simple solution to this problem. In our design ED can invoke such procedures on the customer's activity. Alternatively, an ED server can set a time limit on the duration of the procedure by simply telling the scheduler to abort the activity when the time limit is exceeded.

Chapter 5

Implementation Issues

5.1. Overview

This chapter examines the feasibility of an implementation of the design. The main purpose of the examination is to shed light on the implied execution efficiency in a FOCS and the complexity of the required architectural support. We chose to discuss certain components and features of the design that require a closer examination with regard to these aspects. The discussion focuses on OSB services, service invocation, representation of servers, p-hosts, and activities, as well as on issues of addressing and memory management.

The physical environment assumed in this chapter is depicted in Figure 5-1. It consists of a collection of CPU's, memories, and I/O resources connected via a common bus — similar to the emerging class of multiprocessor systems known as *multis* [22, 5, 9]. The major characteristics of this environment are the ability of each component to address any other component, and the ease with which components are added or removed. Such an environment allows an open architecture in which only minor restrictions apply to attaching customized equipment.

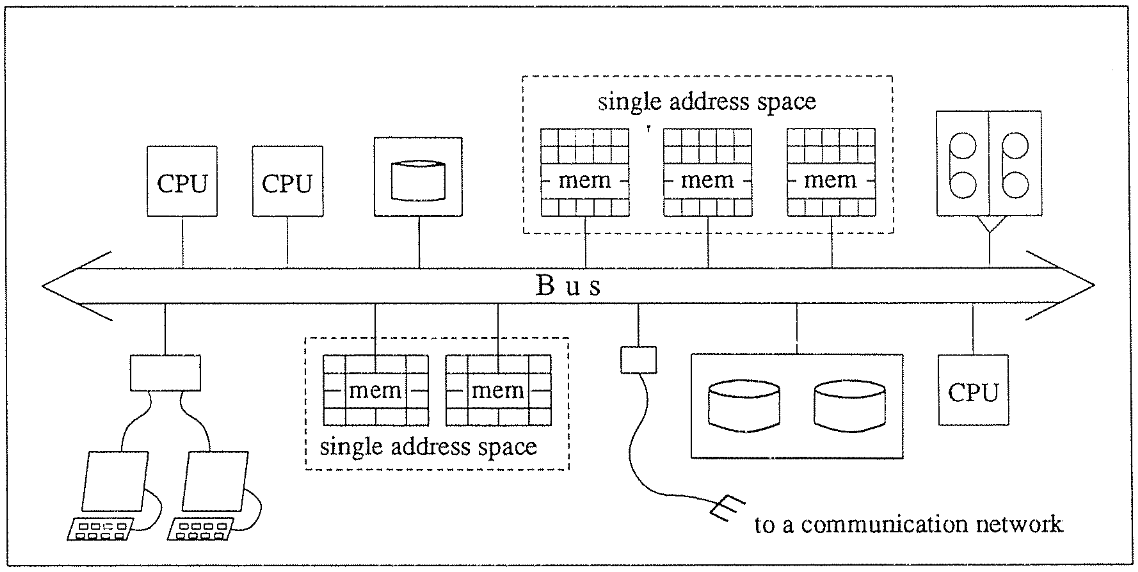

Figure 5-1: The physical environment

5.2. Representation of Servers

Servers are implemented in all three levels of hardware, firmware,[1] and software. The implementation of software servers follows directly from the design of memory management. Each server is mapped to a Space and its services are executed on the CPU's. Services of a p-host can be implemented in hardware or firmware in the device that contains the p-host's resource.

It is desirable to allow merging of cooperating software servers into a single Space in order to reduce the overhead of crossing Space boundaries. Each server in the merged collection would remain functionally a separate server for its customers. This merging can be done statically, analogous to linking Ada packages or Modula modules to form a single program. It can be done dynamically by adding a new server to a Space which already stores another server, as follows. Assuming a segmented virtual memory, each server may have "private" segments. In such a case, the segment numbers in compiled addresses are either updated at load time, or segments are referenced via pointers which are set dynamically. Alternatively, counterpart program components (e.g. code segments) can be merged to form a single segment, reconciling addresses in references to them. In such a case, a segment that contains references to other segments cannot be shared with other servers because the reference are changed. Otherwise, the segment can be shared if it uses base registers to references other segments in the Space, similar to the base registers used in the IBM System/360 [26].

The ability to merge servers raises questions as to whether merged servers can have individual ids or separate exception handlers. One way to support such a distinction between servers in a single Space is to extend server ids to include an index within the Space in addition to a Space id. Each Space would then contain a dynamic vector of "server descriptors." Each descriptor contains a flag indicating whether the respective server is valid and an address of an exception handler. The vector is at a well-defined location, say at the Space-relative address 0. It is prepared by the servers' loader and can be modified by the servers themselves. A CPU validates a given server id by inspecting the appropriate entry in the vector using the server id as an index, for instance at service invocation and upon exceptions. The advantage of this organization is its simplicity; however, it adds overhead to the above CPU services.

In order to reduce the binding time of a new server, a system-wide convention about the initial bindings required by a server can be defined by the implementor. Information about these bindings, e.g. the provider's and service's names, would be grouped in a predefined structure similar to a symbol table of a separately-compiled program module [13]. The loader would inspect the structure and fill it in. This method saves the search time for commonly used bindings.

The memory of a device or its processor might be too limited in its capacity or sophistication to support all the services of a p-host. In particular, it might lack the ability to maintain ownership information, to support concurrent service invocations, or to schedule activities on a CPU to communicate with customers. Rather than complicating the I/O architecture in a FOCS, these tasks can be relegated to a software component of a p-host. This component would be a "front-end" server for

[1]As a convenient simplification, we consider the terms *firmware* and *microcode* to be synonymous.

the users of the p-host. It is analogous to a device driver embedded in a conventional OS kernel [109], except that it does not require a privileged mode to access the device. It is the responsibility of the device-resident component of a p-host to verify the eligibility of its "driver" to access the physical resource.

5.3. Activity Management

5.3.1. Service Invocation

A hardwired or microcoded service of a p-host is invoked through the bus. A binding for such a service includes the bus address of the device in which the p-host resides, plus a device key. The key and an operation selector are passed through the control lines, and the invocation parameters through the data lines [62]. They are verified by the device. The invocation of such services can be via generic machine instructions, similar to the I/O instructions of the IBM System/360 and other conventional architectures. For protection reasons, the bus controller or a CPU adds an artificial delay to a failed invocation, for instance by busy waiting for a predefined period, in order to reduce the effect of erroneous invocations or intruders on bus contention.

As mentioned earlier, at the invocation of software services the CPU verifies the target address and the key of a binding, and transfers control to the selected operation. To locate an operation efficiently, it is assumed that the binding's origin contains a vector of addresses, one for each operation. The zeroth entry is reserved for a termination operation. It is left to the compiler to translate an operation name in an invocation request to an index into the vector, based on the specification of the service in the *provide* clause.

If different services expect their parameters to be passed in some of the same registers, then service invocation bears the extra overhead of saving the registers that will be *live* after the service returns, and restoring them thereafter. To reduce this overhead, the CPU architecture can support a *register window* per invocation, as in the Berkeley RISC-I [103], BBN C/70 [2], and Bellmac-8 [35] architectures. In such an architecture, register saving is not required at service invocation, since windows overlap (see Figure 5-2a). Register windows can be viewed as a stack in absolutely addressed memory, protected by the CPU. Since part of the window used by the invoking service is invisible to the invoked service, the return address as well as other vulnerable values can be hidden there. Hence, this attribute saves the overhead of maintaining local stack frames (by servers) and maintaining the return address stack (by the CPU). Furthermore, if register windows are used for procedure calls as well, the two stacks can be combined as shown in Figure 5-2b. Each window of a service invocation may include multiple windows for procedures and exception handlers invoked during the service. The implications of register windows on the CPU architecture are mixed: they require extra architectural support, but on the other hand they eliminate the need to support stacks in memory. CPU support for exception handling is also simplified, since the exception stack can be mapped onto "ordinary" register windows.

The actual sizes of the register file and windows are a tradeoff between the efficiency of Space switch and activity switch: the larger these sizes, the more overhead register saving incurs on activity switch. In order to reduce this overhead, each CPU may include a per-window bitmap of registers in

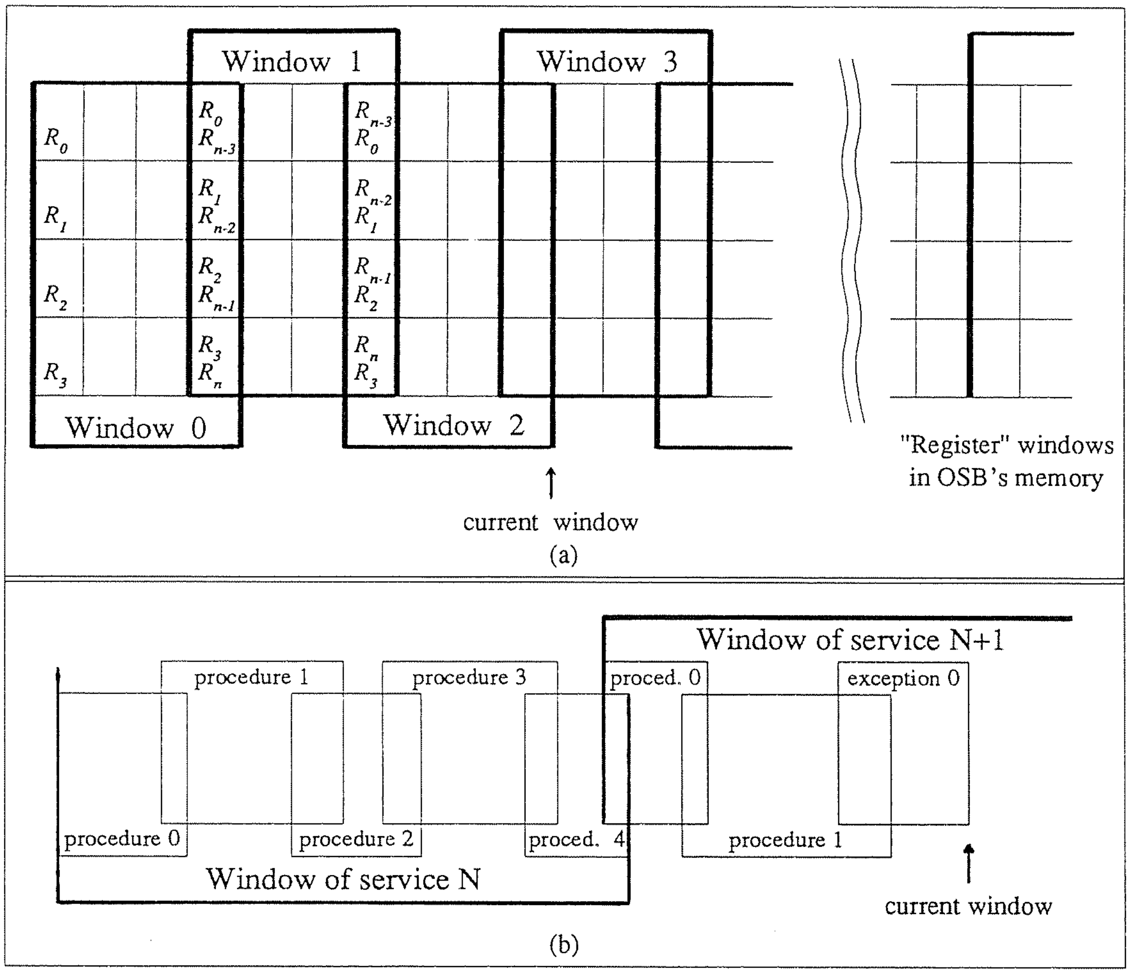

Figure 5-2: Register windows

(a) Fixed-size register windows, one per service invocation.

(b) Variable-size register windows, one per service invocation. Each window includes multiple windows for procedure calls and exception invocations.

use, and a dynamic window count. The bitmap is statically set at compile time. It would be dynamically modified at register allocation and service invocation. At activity switch, the CPU would consult the bitmap and window count in order to avoid unnecessary register copy. This feature extends a similar one employed in the VAX [3] to store and save registers at procedure call and return.

How large should a register file be in order to support common patterns of service invocation? Suppose service invocations will exhibit a pattern similar to kernel calls in (conventional) RISC-based systems. If so, 6 registers per window should suffice, as in the CLIPPER architecture [97]. Alternatively, if service invocations will exhibit procedure-call patterns, then we can adopt a window size of 8 to 16 registers as in most of the implementations of fixed-size window registers [64]. Therefore, a practical window size can perhaps be of 6 to 16 registers. We presume that the length of dynamic chains of invocations will usually be short, say three hops or fewer. This presumption is based on the observation that there are not many services that form a linear dependency (where service A invokes B which invokes C and so forth). In addition, recursive service invocation—a potential source of long dynamic chains—makes little sense in practice. Consequentially, it is conceivable that about 64 registers would suffice. Suppose that additional 32-64 registers are used for parameters and locals of procedures called during a given service. Hence, the size of the register file can be on the order of 300 to 400 registers, which is a medium-size file compared with other architectures. If one adopts a large register-file size of 1024 registers or more, as in some contemporary architectures [2, 50], the above assumptions about window size and length of dynamic chains can be largely relaxed. That is, the number of windows and their sizes will suffice for most patterns of service invocation.

On some occasions the length of a dynamic chain may exceed the number of windows in a fixed-size window architecture or the size of the registers file. To solve this problem, additional windows can be mapped into the physical memory controlled by the OSB, as shown in Figure 5-2a. A similar approach is taken by the RISC II architecture [64, 130]. This window spill-out is transparent to the invoking and invoked services (or calling and called procedures) and is protected by the OSB. However, this solution relegates memory allocation tasks to the OSB, which consequently may impose a limit on the length of dynamic chains.

5.3.2. Activity Context Descriptors

Two important issues of activity management are the efficiency of activity creation and the overhead of maintaining the context of activities. These issues are examined through the activity's context descriptor (ACD), an example of which is shown in Figure 5-3. An activity id includes a pointer to its ACD and a random number which is stored there. This number is modified upon activity creation and termination to guarantee unique and verifiable ids. The key in the ACD is used to generate a verifiable permit which allows its holder to modify certain fields in the ACD. Other fields in the ACD are straightforward; most relate to issues discussed in previous chapters. The return address stack and the exception stack are eliminated in a register-window CPU architecture. The priority value would be set by the activity's scheduler. As discussed earlier, it can be consulted by a server to decide whether to preempt a mutually-exclusive resource or a critical section from another activity of a lesser priority. The *aliased to* field is used when an activity splits into two activities during abnormal termination (see §4.2.4). Since servers use an activity id to locate the internal resources they have allotted to the activity, they would be confused by the new activity id — hence the aliasing to the old id. The statistics field is an optional extension to help servers make decisions. For instance, the *idle since* field can help a server in the activity's chain decide whether to preempt an internal resource from the activity.

How efficiently can activities be created? Only a few fields need to be set initially: the random number, key, owner id, current server, bindings, a return address, and parameters to the initial service. Some of these fields are copied from the parameters supplied in the *NewACD* service. Other fields can be set directly through the *SetACD* operation, which can be implemented as a machine instruction similar to the *mtpr* (move to processor register) instruction of the VAX.

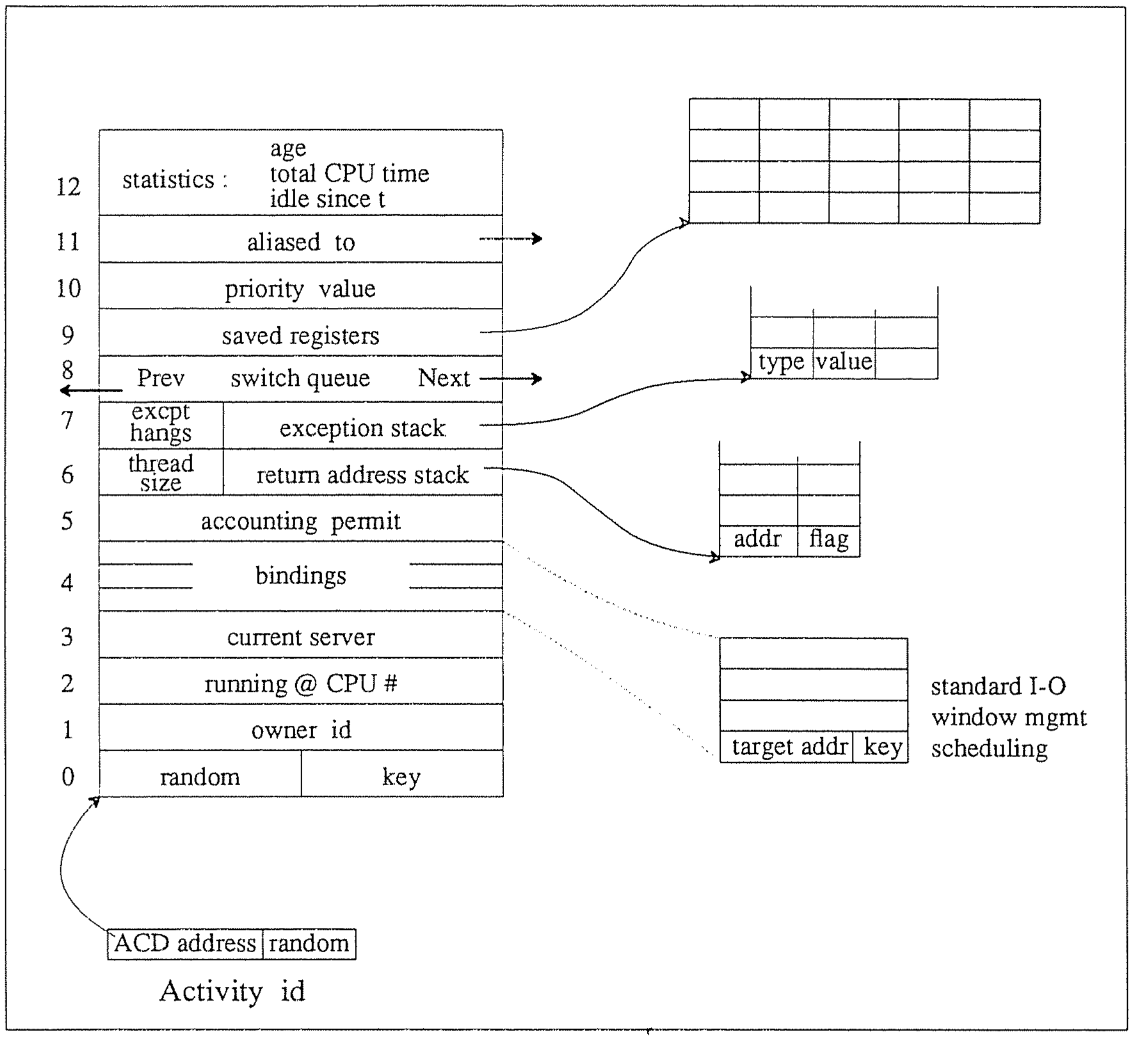

Figure 5-3: An Activity Context Descriptor

Observe that activity creation does not require manipulation of memory-mapping structures. Thus, activity creation is conceivably simple and incurs little overhead.

Another issue of concern is how many activities can be supported by the OSB. This question is important, since if activity creation is cheap, applications may want to employ many activities in order to achieve high computational concurrency. If many activities can be supported, a server would be able to create an activity for each logical task; writing concurrent programs would be simplified. In the following discussion we develop an estimate of the number of activities which can reasonably be supported.

Most of the fields in an ACD are short; they are Boolean flags, small numbers, or pointers into the heap area of the OSB where ACD's and their structures are stored. Thus, the size of an ACD (except for the bindings and the saved registers) would conceivably be on the order of 32 bytes (32B). The saved registers occupy a variable-size area of, say 256B on the average (assuming 64 registers of 4B each). The bindings occupy an additional fixed-size area of, say, 48B (assuming 4 bindings of 12B each, of which 8B is a target address and 4B is the key). The two stacks are each of a variable size, but probably short on the average, perhaps 1 to 4 entries of 16B each. Therefore, an ACD with its structures may occupy about 400 to 500 bytes.

Assume furthermore that the number of activities is bounded by the number of CPU's, and that on the average only up to a few tens of activities would exist per CPU. (Otherwise, the performance of all of them will sharply degrade.) Hence, a few KB of virtual memory per CPU is needed to store ACD's and their structures, say 4KB to 10KB. Since the amount of physical memory per CPU in current multiprocessor systems is about 1 to 4 megabytes (MB) [8, 9, 22], about 0.2% to 0.5% of physical memory is required to keep all ACD's core resident. We conclude that the space overhead of activity management is relatively small, and that thereby tens of activities per CPU can coexist.

Observe that only a small fraction of the space overhead of supporting activities derives purely from considerations of openness. Any implementation of light-weight processes must allocate space to save each process's registers and other context. For example, in a more simplistic and less open implementation of ***threads*** in Mach [135], each thread is allocated a fixed-size stack. In fact, the cumulative sizes of stacks of a computation across multiple "servers" in Mach might be larger then the size of an ACD, since several threads represent the computation which is represented by a single activity in our design.

5.4. The O S B: Structure and Services

The OSB is implemented as a single software component—analogous to a reduced OS kernel in a conventional system—and a collection of software, hardware, or microcoded p-hosts of system resources. As for any other p-host, an OSB p-host may have a front-end server. This server is linked to the "kernel" at compile time or at load time, or it runs separately, analogous to an OS utility in a conventional system. This merging of OSB servers is not mandatory. It is desirable for efficiency considerations, in order to reduce the overhead of service invocations among OSB components.

For reasons of protection and simplicity, the OSB can disable interrupts in its critical sections so it never depends on the schedulers of activities executing its services. Services that need not await the completion of certain events, such as *Allocate* or *Invoke*, would be performed atomically. However, OSB services are vulnerable to page faults in a customer's address space, e.g. when verifying a binding. In such an event, prior to invoking the application-level fault handler, the OSB restores the integrity of its data structures that have been modified by the activity, and preempts any critical section or resource occupied by the activity. This prevents an application from compromising the services of the OSB.

Data structures shared by all CPU's, such as ACD's and CPU ownership information, are stored in the memory of the OSB. Their locations can be statically fixed at compile time or at system boot time. Thus, the structures are accessible to every CPU, locating them does not incur execution overhead, and OSB services are simplified. This fact enhances the ability to migrate OSB services to the firmware, hardware, or the CPU architecture levels.

The CPU host maintains a per-CPU stack of owners and a per-scheduler stack of ownership information, as illustrated in Figure 5-4. These stacks are consulted and updated at slice allocation and revocation, as well as at activity dispatching and quantum expiration. The pointers shown are not referenced until the current quantum expires. Hence, the management of ownership information does not incur overhead during execution. The slight overhead for verifying and updating these values occurs at the infrequent events of activity dispatching, and at the presumably less frequent events of CPU allocation and revocation. Based on the above assumptions we conclude that most machine instructions will not suffer execution overhead due to openness in CPU management.

Some of the CPU host's services are relatively simple and thus can be implemented as machine instructions. No Space switch would occur when they are invoked, so they can be performed more efficiently. We presume that this will be the case for *Allocate, Release, Dispatch, ChangeQntm* and *Switch*, which are the more frequently used services of CPU management. These services need to verify only a few parameters and must check the eligibility of the invoker to perform the operation. Such verification can be done by comparing two scalars: the invoker's id with the owner's id. Similarly, it is conceivable that the more frequent services of activity

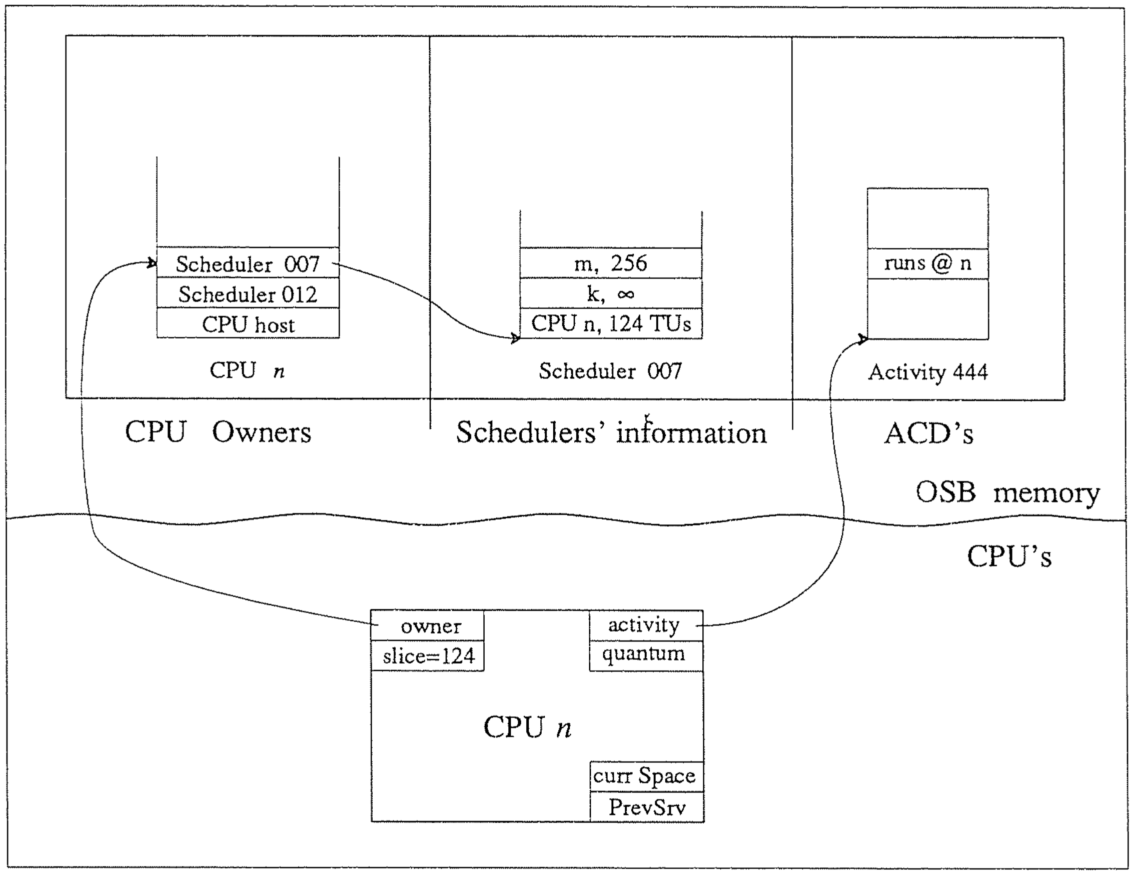

Figure 5-4: CPU ownership information

management—*Invoke, Return, Raise, UnRaise, ReturnFromExcpt* and *Check*—are so simple that they can be migrated to the CPU architecture or microcode. For example, *Return* is only slightly more complex then an ordinary procedure-return instruction. *ReturnFromExcpt* is analogous to the *rte* (Return from Exception) machine instruction supported by the MC68000 architecture [4] or the *rti/reti* (Return from Interrupt) supported by VAX [3] and CLIPPER [97].

Appendix D outlines the algorithms of *Invoke, Dispatch*, and *Allocate*. The following observations can be made. First, these algorithms employ rudimentary checking and simple manipulation of data structures. Second, the data structures themselves are either in local registers of a CPU or in fixed locations in the OSB. The most complex structure is a list of per scheduler CPU ownership, which is simple and conceivably will be short. Third, the dependency on a customer's data structures is minimal. These observations support our earlier assumptions that the particular services can be implemented in machine instructions.

5.5. Addressing and Memory Management Issues

This section discusses the formats of the addresses assumed in the design. It examines their impact on aspects of execution and storage efficiency. We analyze the implications of the Universe structure on the number of servers and segments that can coexist and discuss the efficiency of address translation. The discussion assumes a Universe structure as presented in §4.4.1, and a direct, index-based mapping architecture.

5.5.1. Addresses and Address Spaces

A *full* virtual address is the tuple <Universe id, Space number, segment number, page number, offset>, as shown in Figure 5-5a. This address is globally unique; it is used in bindings, permits, and return addresses of services. Such an address can be obtained statically if the referenced Space id is known at compile time. A *working* address denotes an address within a Space. A *short* address is within a segment, and is independent of the position of the segment in a Space. The latter two formats are generated by the CPU. A Space id is shown in Figure 5-5b.

Table 5-1 presents several alternative schemes of bit distribution among the components of an address, and the implied sizes of virtual spaces. These schemes assume that the number of different physical memories in a system will be small (8 to 64), and likewise the number of Universes per host (16 to 64). Nonetheless, these numbers are still large when compared to contemporary computing systems which support a single physical memory and a single Universe.

The lengths of addresses vary between 25 and 32 bits for a working address and between 48 and 64 bits for a full address. 64-bit long addresses are not uncommon in contemporary architectures [38]. A full address is generated infrequently at Space switch time. During normal execution only working and short addresses are generated in a current Space. According to the schemes of Table 5-1, these addresses are equivalent in size or shorter than those supported by most conventional architectures. Hence, the extra bits required due to openness, i.e. the Space id, incur translation-time overhead infrequently; they incur minor space overhead, since only bindings and permits use full addresses.

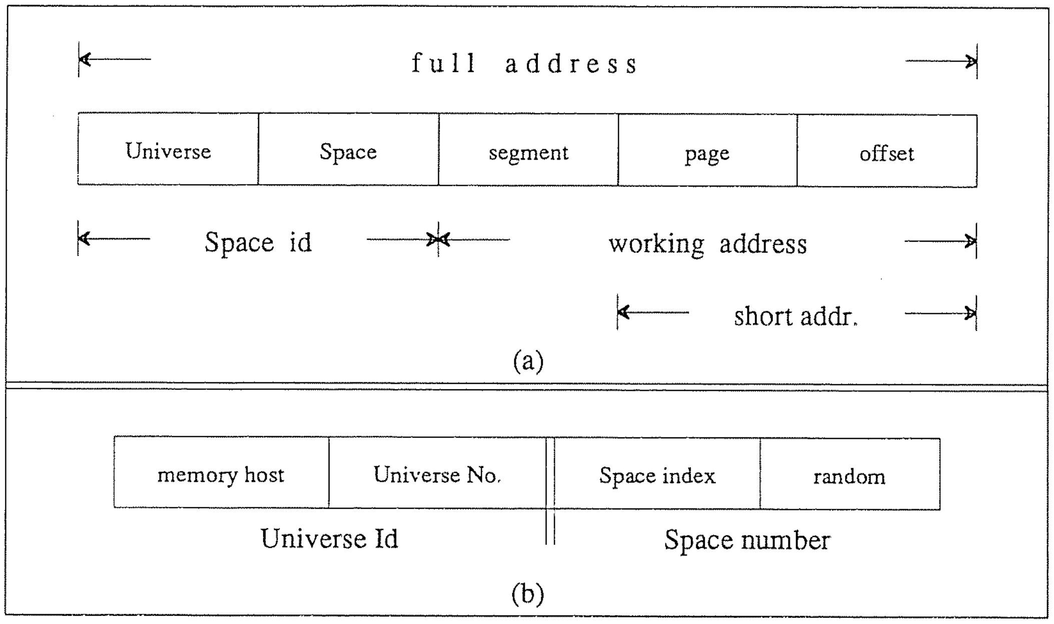

Figure 5-5: Virtual addresses
(a) Full, working, and short address; (b) Space id.

X per Y (# of bits)	a	b	c	d	e
bytes / page	2048 (11)	1024 (10)	4096 (12)	4096 (12)	8192 (13)
pages/segment	1024 (10)	1024 (10)	4096 (12)	256 (8)	8192 (13)
segments/Space	32 (5)	4096 (12)	256 (8)	32 (5)	64 (6)
Spaces/Universe	4096 (12)	4096 (12)	4096 (12)	1024 (10)	4096 (12)
Universes/host	32 (5)	64 (6)	64 (6)	16 (4)	64 (6)
memory hosts	16 (4)	64 (6)	64 (6)	8 (3)	64 (6)
random #	(8)	(8)	(8)	(6)	(8)
Size (# of bits)					
page	2 KB (11)	1 KB (10)	4 KB (12)	4 KB (12)	8 KB (13)
segment	2 MB (21)	1 MB (20)	16 MB (24)	1 MB (20)	64 MB (26)
Space	64 MB (26)	4 GB (32)	4 GB (32)	32 MB (25)	4 GB (32)
Total # of bits					
(working) [full]	(26) [55]	(32) [64]	(32) [64]	(25) [48]	(32) [64]

Table 5-1: Some alternative address sizes

The segment size in schemes (b), (c), and (e)—which offer conventional address sizes—varies between 1MB and 64MB. The Space size in this schemes is fixed at 4 gigabytes (GB), which is comparable to a process size offered by most 32-bit architectures. Observe that a Universe size is 16 *terabytes* (16,000GB!). Hence, an application that uses data structures larger than 4GB can create a Universe and distribute its structures among several Spaces. Each server would access some structures locally and others via service invocations. This option, therefore, increases the size of the available virtual space at the cost of access overhead and increased complexity in managing a Universe.

Below we examine the implications of these address lengths on other aspects of memory management. According to the schemes above, a page table would have 256 to 8K entries. Since an entry is short (see Figure 4-6), it may occupy 4 bytes. Hence, a full-size page table will require 1KB to 32KB of virtual space. Assume that most of the virtual space of the base segment is allocated to page tables. (This assumption will be justified shortly.) A base segment of 1MB to 2MB long, as in schemes (a), (b), and (d), can thus store between 256 and 1024 page tables, respectively. That is, 256 to 1,024 full-size segments can be concurrently allocated in a Universe. A base segment 16MB or 64MB long, as in the other schemes, can support 1K or 2K segments. Since presumably most segments are partially full, page tables are smaller, and thus the number of coexisting segments per Universe can be much larger then the above figures.

Assuming that on the average each Space will consist of only a few segments at a time, several hundred servers can coexist in each Universe. Segment sharing among Spaces can increase that number. Therefore, these schemes can support a large community of servers.

If the base segment's page table is organized as a regular page table, then it requires 1KB to 32KB of physical storage. This requirement is not prohibitively large. However, this table can be much smaller. A Universe manager will presumably own a few large, contiguous memory areas rather than many small, dispersed ones. Therefore, each entry in the base segment's page table only requires a starting address and a size; thereby, the table will require only a small amount of physical memory. This saving implies slower translation, since the table must be searched rather than indexed. Finally, in an inverted-mapping architecture the base segment does not contain page tables. Thus, it can be much smaller, and many more segments can be supported.

Assuming 4K Spaces per Universe, a full-fledged Spaces Table requires 4K entries. A segment descriptor is short (see Figure 4-6) and thus may occupy about 2 to 4 bytes. Suppose that most Spaces rarely exceed 8 segments. An entry in the above table may contain 8 descriptors (totaling 16B to 32B) plus a pointer to an overflow bucket of additional segment descriptors (3 bytes). With an additional random number (1 byte) and status flag (a few bits), an entry in the fixed-size portion of the table would be 21 to 37 bytes long. Hence, the entire table will occupy 84KB to 148KB of the base segment's virtual address space (about 0.5-8.4%). This calculation supports our earlier assumption that most of the virtual space of the base segment is devoted to page tables.

A page fault may occur during address translation at two occasions: (1) at locating a Space's entry in the base segment, and (2) at fetching an entry in a page table. (Additional page faults may occur when searching for a segment descriptor in an overflow bucket.) Assume that normally at most a few hundred servers will coexist in each Universe, say 250. Hence, only a small fraction of all Spaces will be allocated at a time. Namely, about 6KB to 10KB of physical memory is needed to keep that portion of the Spaces Table in core and thereby avoid any double page fault during address translation. Next we discuss caching techniques that can further reduce the frequency of page faults.

5.5.2. Translation and Caches

A CPU interfaces with the memory subsystem through an attached memory management unit (**MMU**). The MMU contains a cache of recent or anticipated accesses. Assume a virtually addressed cache [118], in which every entry is prefixed by a unique identifier, such as the pair <Space id, segment number>. This assumption is in accordance with recent trends in cache design [42, 47], although current architectures support shorter prefixes—typically 8 to 16 bits long. As a result, there would be no need to flush the cache upon Space switch. This saving, in turn, reduces the overhead of service invocation, service return, and dispatching. The space overhead of the cache is minor: the suggested prefix is about 3 to 4 bytes long (see Table 5-1), whereas a common cache entry in most architectures is a few hundred bytes long, typically 128 to 256 bytes. On the other hand, long prefixes may significantly increase the complexity of the cache hardware, and consequently increase access latency.

In order to maintain cache consistency, memory hosts would use the system bus to update caches, e.g. to invalidate or to replace modified entries in other caches. In addition, when a Universe manager modifies the mapping information of some segment, it should update or invalidate any stale copies of this information held at other CPU's. This operation can be accomplished by a regular machine instruction. It should be noted that the Universe manager needs not run in a special *privileged mode* in order to be permitted this service, since it can only affect translation information relevant to its Universe.

If supporting virtually addressed caches with relatively long prefixes is too complex, then caches could be indexed by physical addresses. An MMU must then also contain a translation look-aside buffer (**TLB**) to avoid searching the cache. A TLB stores <virtual address, physical address> pairs and is searched without translation. A translation-and-access request is passed to the appropriate m-host only on TLB or cache misses; the m-host returns the resulting physical address together with its contents. It might be impractical to construct a TLB large enough to hold such pairs for all the entries in a cache, since the search might become prohibitively slow. Therefore, an alternative solution would be to let each MMU perform address translation. This alternative implies that all memories should employ the same mapping scheme. Such a restriction, however, does not restrain openness in other aspects, such as frame size and access speed.

If caches are associated with individual CPU's, a server which is executed frequently by concurrently running activities is duplicated in multiple caches. Consequently, modifications of its data structures will result in high cache replacement or invalidation traffic and many memory update requests. One way to overcome this problem is to let a Universe manager, or perhaps any server, specify via a bus instruction that certain memory blocks should not be cached. Such an instruction is supported (or has been proposed) in some architectures [118]. Alternatively, a cache can be placed at a memory host, or as an independent module on the system bus. If so, a cache can be shared by several CPU's, so only one or a few copies of a shared server are cached. The access time to a shared cache would presumably be faster than to main memory [118]. Simple versions of shared caches are supported in Amdahl 470 computers [1], where a cache is shared by a CPU and one or more I/O processors, as well as in the UNIVAC 1100 [31] and in the Multimax® [9], where a cache is shared by two CPU's.

To simplify memory-ownership verification during address translation, each frame is marked with a *key* that identifies its *current* owner. This key equals the Universe number, so that at each memory access it suffices to simply compare the key with the Universe-number fraction of the current Space id. Notice that at memory access only one key is verified — that of the current owner. so verification is efficient. Other keys are verified during the conceivably less frequent operation of memory revocation. There is no need to verify the key when an addressed datum is found in a cache, provided all cached references of a given frame are flushed by the memory host when the frame changes owners. This use of keys is analogous to the protection method employed in the IBM System/360 [26]. The key can be stored in the frame or in a single inverted table at the memory host, similar to the table shown in Figure 4-6. Each entry in such a table would contain a stack of owner ids. For reasons of space efficiency, an implementation-dependent limit would apply to the size of these stacks and hence to the number of concurrent owners of each frame.

5.5.3. Register Padding

Another aspect of addressing is the size of general-purpose registers. A commonplace register size in contemporary architectures is 32 bits. In our design, however, parameters are passed in registers, and some parameters would be longer than 32 bits (e.g. bindings and permits). Requiring larger registers or special registers leads to extra architectural complexity. Instead, we assume that a sequence of registers can be padded to form long or contiguous values, as with locations in physical memory. For instance, to store a datum 8 bytes long in registers R_5 and R_6, each of which is 4 bytes long, one should be able to specify the target address as R_5 of length 2. A similar architectural extension is used in the C-Machine [50].

5.6. Miscellaneous

Several architectural features that support atomic actions can help simplify programming. First, memory locks can be associated with various memory locations, as in Multimax [9]. Thus, a server can implement a simple mechanism of atomic mutual-exclusion for activities that share its data structures. Second, a simple machine instruction can be provided to postpone timer interrupts during a critical section. This instruction would merely set a flag in the CPU. If the timer interrupts to indicate a quantum expiration while the flag is on, then the CPU grants a short additional quantum to the current activity. During this quantum that instruction is disabled, so a server cannot dishonestly or erroneously monopolize a CPU. If an activity switch does not occur during that period, then the switch is enforced thereafter. Third, other machine instructions common to architectures that support multiprocessing, such as Test-and-Set [8,9] and queueing operations [3,8], can accommodate atomic actions without fear of loosing the CPU during the operation. Notice that a page fault may occur during such an instruction; thereby, the operation cannot be guaranteed to complete within the current quantum, since the fault is handled by an application-level U-mgr. However, the action of changing a shared queue or variable is atomic, since it can be undone on a page fault

®Multimax is a registered trademark of Encore Computers.

and then reissued afterwards.

Chapter 6

Lessons on System Openness

This chapter summarizes the contribution of our study to the understanding of the issues involved in system openness. Its purpose is to present the lessons on the extent of openness and the interplay between the various aspects of openness which derive from the discussion in the three preceding chapters. In the following discussion we recapitulate observations made in previous chapters and add insights from our experimentations with defining the system.

6.1. Overview

Our major lessons on system openness are the following:

- A fully open system is viable.
- Protection requirements only slightly restrict the ability of applications to customize policies and mechanism.
- Our analysis suggests that protected openness will not result in large degradation of execution efficiency.
- A FOCS requires specialized architecture and microcoded services, but it can employ or extend features of contemporary architectures.
- The programming complexity of writing simple programs would be comparable to that of writing them in conventional systems. Writing utility programs in a FOCS would be simpler in some aspects than writing them in a conventional system, but easier in other aspects where direct control of execution is necessary. Most of the programming complexity, however, derives from concurrent execution, not openness.
- Hierarchies can be supported efficiently when they are not rigid, and levels can be bypassed dynamically.

Below we elaborate on each of these lessons.

6.2. The Extent of Openness: General Lessons

The major lesson of our study is that despite protection restrictions, different applications can simultaneously satisfy conflicting demands in a shared environment. An application may have private resources and provide its own services independently of services provided by other applications. Openness is restricted by the application's dependency on the OSB services. However, of these services, only the CPU allocation and activity creation services may potentially restrict the policies or mechanisms employed by the application. But if an application initially creates all the activities it needs and is allocated CPU's on a long-term basis, then during that time it can effectively and independently employ whatever policies it likes. Moreover, the only OSB services it must use during execution are the services for service invocation and activity dispatching. Assuming that these services are implemented in machine instructions, then the application runs *without interaction with any OS* (in the traditional view of an OS)!

A significant restriction on full openness is that applications cannot replace system resources. In addition, not every application will be able to supplement all the resources that it needs. Thus, policies and mechanisms used by applications are not fully open — they depend on allocation policies and access mechanisms employed by hosts of resources they use. However, we learned that this dependency may only slightly restrict openness. According to the model, the hosts should provide rudimentary mechanisms that accommodate resource owners which want to implement more elaborate mechanisms and allocation policies. Therefore, once an application can *bid* for resources and become their owner, it can implement policies and mechanisms as it likes.

Openness extends to the computer architecture as well. Of all the physical resources, only the CPU's and a clock must be in the OSB. Applications may choose whatever interfaces they want for private physical devices. The sole restriction on the choice of interfaces is that devices adhere to a common bus interface. Memories have to comply with a uniform addressing structure. In the case where MMU's perform address translation, memories must also comply with a uniform mapping scheme.

Openness requires that servers communicate with each other, although they might not know each other's protocols. For example, a file server might need to tell different memory managers to pin the buffers of its customers in memory. This communication requirement necessitates the definition of either a standard interface or a complex protocol by which servers can dynamically learn each other's service interface. The standard interface that we chose does not restrict the semantics of services, nor is it mandatory. A server that provides services included in the standard interface may extend or ignore the suggested interface, at the risk of compromising the services provided to its customers by other servers.

Another lesson is that IPC can be made fully open by reducing the mandatory IPC facility of the OS to merely support protected invocations of services across address-space boundaries. Our model and design demonstrate that multiple paradigms of communication can coexist, constructed above the rudimentary facility of service invocation and activity dispatching. An interserver communication server can support any communication facility: connection-based, connectionless, urgent messages, communication via shared memory, and other paradigms.

An additional restriction on openness derives from the limited number of components defined by the model and the design. Only a single execution entity and a single invocation paradigm are supported. The design supports only a few hierarchically structured memory entities and requires that a virtual space is mapped to a single physical memory. However, openness is only slightly restricted by these facts, since not all components are mandatory, and since the semantics attached to them are minimal. The levels of granularity of memory buffers, execution units, I/O transfer units, or communication units are *problem oriented*, defined independently by each application. An application can support a variety of semantics for sharing and inheritance. For instance, memory can be shared in various levels. Universe managers can share physical memory through allocation. Each of them can decide how physical memory is shared among its customers. Spaces can share segments through mapping, each with possibly different access rights. Servers can share address spaces by being merged into one Space, by sharing segments, or by having access permits to each other's Space.

6.3. Openness *vs.* Protection

The major lessons from our work regarding the interplay between openness and protection are that protection can be open too, and that a minimal set of protection restrictions allows opening the services and resources customarily provided by the OS to applications. The notions of encapsulation and light-weight capabilities allow applications to tailor their protection mechanisms and security policies as they see fit. Since a server is guaranteed that its services are entered in an orderly manner, it can impose further application-specific restrictions. Bindings and permits are light-weight capabilities with respect to their protection flavor and their overhead. First, they reside in the address space of any server so that they can be created and distributed without OS support. Second, they do not specify generic rights. Therefore, such capabilities allow openness of access mechanisms as well as of inheritance. Because these capabilities can be used and passed around efficiently, they support openness of low-level and frequent services that use them.

Protection considerations constrain the openness in CPU and execution management, whereas protection does not restrict the management of virtual and physical spaces. For protection considerations, the model encloses the CPU's and a clock in the protected domain of the OSB. Except for the memory in which the OSB is loaded, memories are open to applications. Likewise, the execution domain of the traditional *process* abstraction is not fully open, whereas the space domain is fully open. Activities are created, terminated, and dispatched in a protected way through the OSB, since they cross address-space boundaries; servers and resources can be created and terminated without the intervention of the OSB. Traditionally, process management has been retained in the OS in order to guarantee protected communication and authentication. One of the major lessons from our separation of the space and execution domains into servers and activities is that the role of server management can be relegated to applications without compromising protection objectives. Communication can be protected as discussed above. To support unique server ids, a hierarchical addressing scheme is employed which is reminiscent of naming methods used in distributed systems. The OSB provides rudimentary authentication services, above which applications can construct higher-level authentication schemes.

Support for orderly termination and recovery are two traditional roles of operating systems. We have learned that these roles, too, can be taken out of the protected domain of the OS, as long as it is guaranteed that servers can discover the termination of other servers, and that servers can protect themselves from a failing server. Furthermore, openness motivated relegating the tasks of activity scheduling and termination to ordinary servers, but protection considerations necessitated that the OSB help servers to recover from unorderly termination of activities.

The issues involved in critical sections and in passing parameters (i.e. registers *vs.* stack) demonstrate the tradeoffs between openness and protection, as well as the interplay of these aspects with efficiency. A critical section raises the conflict of needing to protect it from the activities executing it—that is, allowing the server in which the critical section resides to fully control it—and to protect a CPU from being monopolized by that server. This conflict motivated the model's dual view of the activity–server relationship; namely, to let a server switch between activities, and to allow servers to temporarily postpone quantum expiration. The solutions adopted at the model and the design levels are a lesson in resolving the above conflict.

Passing parameters in a stack raises similar protection–openness conflicts. A stack stored in some Space would be unprotected; protecting stacks by the OSB implies restrictions. Furthermore, putting the stacks in the OSB would impose access inefficiency or would necessitate a special capability management mechanism as in [48], which has adverse implications on the complexity of the hardware. As shown in the previous chapter, registers can be viewed as automatically-allocated stacks protected by the CPU.

The bottom line of the interplay between openness and protection is twofold. First, despite restrictions on openness set by protection requirements, an application can choose the services and resources that it needs. Second, a secure, open system can be constructed from untrustworthy components, provided one component is generally trustworthy, and each application can choose the components that it trusts as well as be protected from those it does not trust.

6.4. Protection *vs.* Efficiency

Protection inherently incurs execution overhead because it requires separation of entities and hence checking of rights at protection boundaries. The overhead can be minor if checking is done efficiently and infrequently. Below we discuss various facilities where this lesson is clearly illustrated, such as activity management, accounting services, and service invocation. The discussion emphasizes the tradeoffs between protection and efficiency offered by several design alternatives.

Protection of resources incurs the overhead of verifying ownership and permits. As we pointed out, this overhead will not be large since the checking can be simple and performed infrequent. It involves comparing a scalar, e.g. a Universe id, or a vector of values, e.g. the fields of a permit. Although ownership itself can be hierarchical, ownership verification needs not be hierarchical, and hence can be efficient. Likewise, having hierarchical resource ownership without requiring multilevel mapping of virtual resources to physical resources contributes to the efficiency of accessing resources. Moreover, the effective overhead can be further reduced by reducing the frequency of verification. As shown with CPU management, the overhead of protection can occur relatively infrequently at context switch and allocation. The overhead of an extra mapping level on accesses to physical memory can be circumvented by appropriate caching techniques or interfaces between memory hosts and Universe managers.

Protection of services has required that invocations use verifiable bindings. The overhead of verification should be minor, as illustrated in Chapter 5 and in Appendix D. Acquisition of bindings can be efficient, since bindings can be obtained at compile time, at load time, or only on demand, and they can be obtained in groups.

The decomposition of the system to separate servers incurs overhead on service invocation. The various techniques discussed in Chapters 4 and 5, such as caching, register windows, and eliminating automatic manipulation of stacks can reduce this overhead. We therefore believe that a service invocation can be performed nearly as efficiently as calling a procedure in a different locality.

The efficiency of service invocations decreases when the invoker or the invoked server must preserve the state of a service in a local stack frame, since then each invocation will require extracting a frame from a heap and later freeing it. This execution overhead can be reduced by employing simple heap allocation techniques, such as keeping a list of free stack frames sorted by frame size. In addition, frame sizes can be fixed or decided at compile time [50], trading space efficiency for execution efficiency. Passing parameters via buffers, which themselves are passed via access permits, incurs overhead on accessing these buffers. We presume that this overhead should be smaller than that incurred by the heavier IPC mechanisms of conventional systems, in which buffers are transferred back and forth in messages.

The protection mechanisms offered by the model and by the design illuminate a well-known lesson: supporting the simple cases efficiently can improve the general efficiency, although some cases might experience higher overheads. The default protection mechanism can be efficient since it is crafted to be as low as can fit the minimal protection needs of mutually trusted servers. The overhead of protection, however, grows with the level of mistrust. Mutually suspicious servers or servers shared by different applications must protect their resources or services by additional means. Hence, they might suffer greater overhead in implementing their own protection mechanisms. Relegating capability mechanisms to servers will probably be less efficient than if capabilities were supported directly by the architecture. But imposing capabilities on all servers runs counter to openness and efficiency. It is left for further investigation to find whether some extended, non-mandatory capability mechanism that is supported by the architecture can be beneficial in terms of the balance between protection, efficiency, and architectural complexity.

Another lesson about tradeoffs between protection and efficiency is illustrated by the amount of control over the distribution of capabilities. Conventional capability-based systems suffer large overheads due to the centralized control over capability management. In our approach, bindings, permits and application-specific capabilities can be passed efficiently, since a server who passes them to a customer (or *vice versa*) cannot restrict the other server from passing them to others. Alternatively, a server may provide an intermediary service for accessing a resource rather to give a permit for the resource, and thereby increase the access overhead.

The design of activities demonstrates additional lessons on the tradeoffs between efficiency and protection. Since an activity can span multiple servers, a single entity carries a transaction from the end-user's server down to the lowest-level resource. Therefore, a transaction or any computation that crosses server boundaries will be executed more efficiently than if it were composed of different threads as in Mach — because in our approach service invocations do not require a context switch, only an address-space switch. However, the need to protect servers from activities has motivated protecting ACD's in the OSB. This protection will presumably increase the cost of activity creation and termination. The same need has motivated the elimination of stacks, which itself has a mixed impact on the overhead of invocations: service invocations can be made more efficient if variable-size register windows are supported, or when a service does not need a stack; in all other cases, service invocations will bear more overhead. Furthermore, for protection reasons a server executed by multiple activities has to keep track of the internal resources occupied by them. The dependency of a server on the schedulers of the activities threatens the server's ability to recover its resources when activities terminate abnormally. Hence such protection must be guaranteed by the OSB. Likewise, in order to protect itself from uncooperative schedulers, a server can schedule its own activities to carry out service invocations or recovery operations, thus increasing the execution cost of its services.

An analogous dependency exists between a server and its customer during "reply" invocations or obliged invocations. For example, this dependency arises when a server invokes a customer's service to announce the completion of an asynchronous operation or to consult a customer about a service option. Similar dependency arises when a server invokes another service provider of its customer, for instance to *unblock* an activity or to copy data from a customer's buffer. Similar dependencies exist in closed, layered systems [44]. In these situations the invoker might not trust the invoked server with respect to the time taken to perform the service. Hence, the invoker may use a less efficient asynchronous invocation mechanism in order to protect itself.

The accounting subsystem has provided another angle to examine the interplay between protection and efficiency. A provider of a service or a resource needs a reliable accounting service to handle charges. But the customer's account has to be protected too. The discussion in §4.2.5 illustrates possible protection levels and their implied overheads.

6.5. Architectural Complexity

A major conclusion of our study regarding architectural support is that a FOCS cannot be realized with standard, off-the-shelf hardware components. Several frequent operations must be supported by the architecture or be implemented in microcode, since emulating them in software would be prohibitively slow. However, this requirement does not necessitate drastically different architectures.

One of the main requirements of architectural support is to perform virtual-to-physical address translation and memory ownership verification. If these functions are performed in software, they cannot attain reasonable efficiency. Openness has necessitated an extra mapping level to support different Universes, which should increase the complexity of the translation mechanism. However, the mapping structure suggested in the design is an extension of the VAX memory architecture, in which page tables reside in the kernel's address space. In our design the latter scheme is extended to support multiple memory management "kernels." The scheme is further extended by allowing a variable number of segments per server, and by allowing segment sharing. Overall, we believe that these extensions would not preclude the ability to perform efficient address translation by the architecture. Moreover, avoiding segment sharing via an extra level of indirection (as in Multics) helps simplify the architectural support. We believe therefore that the complexity should be lower than that of systems that support structured memory mapping such as the iAPX-432 [99]. A designer of a specific system can choose to reduce the architectural complexity by reducing flexibility, e.g. by fixing the number of segments per Space. It should be noted that this architectural complexity buys access efficiency and supports direct manipulation of mapping information.

Ownership verification conceivably will not increase the complexity of the memory architecture, since it requires merely comparing two scalars. Changing ownership can be done in microcode or even software. Verifying access permits is merely a matter of comparing a vector of scalars.

Other techniques discussed in Chapter 5, such as virtually-addressed caches, shared caches, and register windows, extend notions found in other architectures. Although no extension is radical, their combination might incur considerable complexity on the architecture of a FOCS. This issue is left for further investigation.

An additional source for architectural complexity is the implementation of OSB services as machine instructions. However, most of these services are simple, as argued in Chapter 5. The services increase the complexity of the CPU architecture only slightly, since they add a few machine instructions and a few registers. Some of these instructions are similar to or slightly extend machine instructions found in contemporary architectures.

6.6. Programming Complexity

In discussing the programming complexity of a FOCS, we distinguish between three levels of program sophistication: an *end-user* server which is a simple server providing no services to any server; a *utility* server providing high-level services or logical resources, including user-interface, compiling, and runtime support services; and a *physical-resource manager* serving either as a p-host or as a server that manages the allocation of physical resources, e.g. a scheduler or a U-mgr. An end-user server is typically executed by a single activity, whereas the others are executed by multiple activities. A physical-resource manager is the counterpart of a process embedded within the OS kernel in conventional systems, or an OS utility providing low-level services. Of course, servers classified in one level can be of diverse sophistication. We discuss the programming complexity incurred by protected openness in general, and by our approach in particular. These complexities are compared with the complexity of writing the counterpart programs in conventional systems.

The major complexities of writing an end-user server are to select resources, services, and to obtain their bindings. However, services can be selected by a compiler or by a preprocessor utility. The programmer will merely name the functions needed, e.g. *Read*, or explicitly include a given library, e.g. by writing *#include stdio.lib.* A novice programmer may be oblivious to the structure and semantics of the underlying service layers used by the program. Bindings can be obtained either by the compiler, by the loader, by the runtime support package, or through a library package. Therefore, the complexity of writing an end-user server in a fully open system will be comparable to doing so in capability-based, object-oriented, or message-based systems that support dynamic selection of objects or operations.

There are several additional complexities involved in writing a utility server. First, the server has to choose the services upon which *its* services depend, and to ensure that they fit semantically. For instance, an interserver communication service that supports urgent messages has to choose scheduling services that support urgent scheduling. The complexity of choosing such services is essentially similar to that of choosing services in a distributed system. This complexity can be mitigated by name servers that support association of types or semantic information with bindings.

Second, a utility server has to coordinate between multiple activities executing its services concurrently. This task includes synchronization, sharing, exclusion, and recovery from failed or aborted activities. Most of the complexity involved in these tasks stems from the concurrent execution of activities. This complexity is conceptually analogous to the complexity of managing multiple threads in other multiprocessor systems, and can be mitigated through monitors or support packages such as the Mach *C-Threads* package [135]. The bookkeeping that a server must undertake in order to be able to recover resources from a customer or an activity is comparable to the bookkeeping a shared utility in a conventional system must perform. In a fully open system, the management of concurrent activities raises additional complexities because the activities are scheduled by different servers, some of which might not be trusted by the focal utility server. These complexities are reduced by the mechanisms of the OSB to recover from abnormal activity termination, to postpone quantum expiration, and to switch between activities. In fact, the latter feature simplifies the handling of conflicts between activities as compared with the management of threads in Mach.

Third, the need to communicate with unrelated servers exposes another source of complexity in writing a utility server. This complexity should be greater in our approach, since activities span multiple servers. The standard interface should mitigate this problem with respect to the services that one *must* invoke. We believe that opting for a facility that allows a server to dynamically learn the interface of other services would result in much more complex programming. Each installation may add other conventions to service interfaces in order to simplify communication.

Fourth, charging for resource usage and for services adds programming complexity and execution inefficiency to servers. The major complexity is in deciding prices and in distributing charges fairly among customers. The principle of charging is not peculiar to a FOCS — other systems [131] employ similar notions, and incur comparable complexity or inefficiency. Some attempts in devising pricing strategies in an open system have been made [51], but the problem is still open. As shown in §4.2.5, different levels of protection of user accounts can distribute the complexity of dealing with charges differently between customers, service providers, and the OSB.

Lastly, a utility server has to own certain resources in order to implement its services. The complexity of obtaining them depends solely on the provider–customer relationship. For instance, writing a scheduler can be simplified if it is allocated as much virtual time as it needs. It is harder to write a scheduler if it has to compete with other schedulers for CPU time in a free-market style.

Writing a compiler, a library, or a runtime package bears extra complexity in an open system due to the additional architectural features and the dynamic binding between servers. As alluded to earlier, we opted to increase the complexity of writing such programs in order to reduce the complexity imposed on less sophisticated programs. This choice is based on the assumptions that the former programs are written by sophisticated programmers, and infrequently.

We conclude that writing a utility server in a fully open system should be more complex than writing counterpart programs in other systems. However, the mechanisms supported by the OSB, compilers, and other servers should mitigate the complexity. In some aspects this support will make writing a utility server even simpler.

Several aspects of writing a physical-resource manager are simplified in a FOCS. Only problem-oriented semantics, that is, of a particular application or a group of similar applications, need to be addressed. For instance, it is simpler to write the memory manager of a particular DBMS than to write one for all DBMS's, because the programmer can choose the services that support the required semantics. Moreover, since an application may have private p-hosts or implement services above the low-level service interface of a p-host, there will be no extra complexity involved in trying to bypass or undo unwanted facilities as in closed systems [94, 112, 125]. We believe that in general it is simpler to implement a policy than to tell a monolithic OS the preferred policies. A great deal of simplification is achieved by letting customers specify ***absolutes*** instead of ***hints***. Additionally, the support of activities simplifies the interaction with p-hosts, since a single activity is devoted to a transaction. On the other hand, it is more complex to write such a manager in our approach, since allocation of physical resources must be negotiated.

6.7. Other Complexities

Aborting a transaction can be a complex task in any system where a transaction consists of language-level threads (as in Lynx), or of threads visible to the OS (as in Mach). In a FOCS this task might be harder, since in a system that is decomposed there are potentially more servers involved in the transaction. Our choice of an activity as a continuous thread that spans multiple servers simplifies this task, especially since the OSB supports recovery.

Resource inheritance can be implemented in a simple way in a closed system such as Unix, where the OS owns the resources and maintains the context information of processes. It is much more difficult in a FOCS, because the OS does not own all the resources and servers. Complex protocols might be required to transfer ownership at server creation and at termination. This problem can be mitigated with the support of libraries that communicate the requests of inheritance to the appropriate hosts. Obviously, if an application is constructed so that a single server controls the ownership information of several resources and the server management functions, then inheritance is supported simply as a side effect of server management functions.

Similarly, deadlock avoidance or detection is more complex in a FOCS, because the management of resources is dispersed among many servers. To detect a deadlock and resolve it, one needs to define complex protocols and hope that servers are willing to adhere to them. We have designed a preliminary protocol for deadlock detection. But, because of possible mistrust among servers, not every deadlock can be detected or resolved. It should be noted that these issues are complex even in conventional systems, and hence have been abandoned by most general-purpose OS's [133]. A similarly complex task in a FOCS is debugging. In order to discover the source of a bug, one might need to inspect the state of several servers, not all of which might trust the debugger.

6.8. On Dependencies between Service Domains

We have shown the viability of a unified view of services in which each service can be independently selected. Specifically, using or replacing a set of services in a given service domain (e.g. memory management) does not restrict the flexibility of selecting services in related service domains (e.g. buffering and CPU scheduling). This separation has the advantage that one can gradually customize the services one needs.

However, it might be hard to achieve such separation when the support of particular semantics in one domain demands very specific services in another domain. Moreover, efficiency concerns might necessitate the coordination of services in different domains. For instance, the scheduler and the U-mgr of a given application may need to coordinate their policies to achieve better performance. As an example, before scheduling an urgent computation, the scheduler would prompt the U-mgrs of the involved servers (if known) to prepage their working sets. This task might be a complex one if different servers are involved dynamically.

6.9. On Hierarchies

It is conceivable that in an open system multiple hierarchies will be formed, since service provision and resource allocation are inherently hierarchical. For instance, a mail server might use DB services, which use some structured filing services, which in turn use flat filing services, which use disk access services. Or, a scheduler can itself be scheduled by another scheduler, and so forth. Some of these hierarchies must be dynamically constructed, since it might be the case that only at execution time it is known which services are needed. For efficiency reasons, it should be possible to construct other hierarchies statically.

An important lesson of the model and design is how to support both dynamic and static hierarchies in a single service domain as well as across service domains. The FOCS model does not impose hierarchies. There is no need to explicitly define new levels as in some other hierarchical systems [63, 115]. Moreover, since hierarchies are represented merely by bindings and ownerships, they can virtually be of infinite depth. (As a practical matter, however, the number of owners of a given resource unit might be limited.) Service domains can be mutually independent in their hierarchical structures.

A hierarchy of resource users usually implies that users at the higher levels may access only virtual resources. The model's view of ownership illustrates a lesson in combining a hierarchy of resource users with a flat scheme of access; the scheme allows resource users to bypass all intermediate owners when resources are accessed. As discussed earlier, the latter feature is invaluable for allowing servers at different hierarchical levels to efficiently perform frequent or urgent functions such as CPU scheduling, memory mapping, and handling page faults.

The inefficiency of a rigid hierarchy motivates components at different levels to make special "deals" which complicate the hierarchy. Extra complexity is incurred when a deal breaks. For example, it is difficult to undo a deal when one of the involved parties terminates abnormally. To avoid such a problem, we made the deals visible and supported. Resource-allocation deals among servers at different logical hierarchical levels are possible. They are made through the resource's host or through a resource manager that controls the allocation. Death-inheritance rules allow undoing a deal when a participating server dies abruptly.

Chapter 7

Conclusion

Ever since people first founded societies to organize the sharing of common wealth, they have had to resolve the conflict between freedom and order: On the one hand, it is desirable to let members use private or shared resources in the way they like. On the other hand, a centralized authority must supervise resource usage in order to optimize the benefits to the society as a whole. Computing systems are no exception in this respect — such a conflict has always been a dilemma in designing operating systems. Early OS designs resolved the conflict by choosing one of two extremes. They have either included most of the resource-control facilities in the OS, allowing users only limited decision-making capabilities; or they have included a limited set of facilities in the OS, imposing on users the burden of supplementing the necessary services. The trend of modern OS designs, however, is to open up the OS services and interfaces to users. In the last two decades, systems have increasingly allowed users to be more involved in the decision making of the OS, as well as to customize the OS services and interfaces.

The goal of this research was to explore the feasibility and implications of continuing this trend to its ultimate extent. We have studied how to construct a fully open computing system and investigated the interplay between openness, protection, efficiency, and complexity in a multiuser environment. In Chapters 3 through 5 we presented a detailed model of such a system, examined a variety of components of a specific design, and addressed implementation issues. These chapters show that a fully open computing system is viable. Our approach to constructing such a system employs a unified view of resources and of services. It is based on the notions of resource ownership and service provision. The role of the OS is radically reduced to providing rudimentary mechanisms to protect resources and applications. The OS allows an application to select the resources and services it needs, as well as to share them with other applications. Physical resources can be directly controlled by unprivileged, and in fact untrustworthy, applications. Our analysis suggests that such openness can be supported without breaching protection, undermining efficiency, or requiring extensive architectural support.

Our research method manifests that a system should be evaluated at different levels of abstraction, starting from understanding the crucial problems at the conceptual level. This method contrasts the more common, *ad-hoc* process of first designing—or even implementing—a specific system, and then redefining the problem.

7.1. Contributions

We view our work as basic research in computing systems design. It is a pioneering attempt to study full openness in a methodological way. The major contributions of this dissertation to computer science are in the field of systems design in general, and operating systems design in particular:

1. The viability of constructing a fully open computing system was shown, and the interplay between the various aspects of openness was analyzed.

   The dissertation identified the minimal bounds on system openness due to protection requirements. It examined many facets in which protected openness interacts with execution efficiency and resource utilization. It provided insights into programming complexity in such a system and into the complexity of the required architecture.

2. A novel model of a computing system was defined.

   The model presented an abstract, fully open computing system and exposed problems with openness. This model, and in particular the notions of resource ownership and of charging for resources and services, has been recognized as a promising foundation for future, market-oriented open systems [92]. Some of the concepts introduced by the model, such as the dual view of server–activity relationship, are amenable to less open systems.

In the course of defining a fully open computing system, the dissertation presented several mechanisms which offer novel approaches to computation and communication. In this respect, the dissertation contributes to a number of research areas in Computer Science:

- The dissertation provides a novel view of resource management.

  It presented a unified view of logical and physical resources. Resources can be owned by a hierarchy of servers but are accessed directly. Applications may own private or shared resources and provide the policies as well as the mechanisms for using them.

- The dissertation redefines the role of the OS.

  It defines the role of an OS as the provider of rudimentary services necessary to protect system resources and applications. In contrast with the traditional definition of an OS as the resource manager [55, 104, 133], the dissertation relegated the management of resources to applications. The traditional roles of an OS as the process manager and as the provider of an IPC facility were also relegated to application-level servers. In contrast with other "minimalist" systems, our approach emphasizes sharing in a multiuser environment.

- The dissertation offers a powerful multithreading facility.

  The activity abstraction extends recent attempts to support threads of control by the OS [10, 11, 12, 75]. It seems to fit better than such threads in representing computations that span multiple address spaces, since it easily conveys state information. Our facility allows simpler propagation of exceptions, abortion of transactions that span multiple servers, and recovery. It also offers the advantages of a language-based approach to multithreading, which allows each server to control the threads executing its code. The separation of the space and execution realms into servers and activities, together with activity-management support services, provide an attractive alternative to the traditional *process model* employed by conventional systems.

- The dissertation introduced a simple but flexible communication facility.

  The service invocation and return facility extends the semantics of a local procedure call to a cross-address-space call. This facility, however, offers simple semantics similar to those of low-level operations in traditional systems, e.g. a copy machine instruction. It is simpler and should be more efficient then contemporary message-based, remote-procedure-call facilities. The synchronous invocation paradigm also supports asynchronous invocations and asynchronous service provision.

7.2. Directions for Future Research

Our work provides a framework for continuing research in system openness by investigating policies appropriate for an open environment and extending the definition of a fully open computing system. Such extension is needed in all three levels of abstraction — extending the model to support distributed systems, and expanding our design. An implementation is needed to verify our assessments about the implications of protection on efficiency and complexity. We suggest three major avenues to continue our research:

Developing a Testbed

A prototype implementation should be constructed to further scrutinize the complexity aspects and study the performance of a FOCS. The development process will refine some of the general lessons on system openness presented in Chapter 6. An implemented FOCS will allow us to quantify and to further qualify the tradeoffs between protected openness, efficiency, and complexity. Such an environment can serve a pedagogical purpose, allowing students of OS classes to concurrently manipulate components of an OS.

This avenue includes several possibilities:

- A prototype implementation, possibly by simulating or emulating some of its components. We plan to pursue this possibility in the near future.
- Designing additional services of the OSB or a COS, language processors and specific applications.
- Elaborate design and evaluation of the computer architecture.

Devising Policies and Algorithms

A fully open system offers an environment for more sophisticated policies and algorithms for resource management. The dissertation sketched a few such policies and discussed two algorithms. There is a need to devise more detailed policies for various resources, both the levels of the OSB and individual servers. A quantitative evaluation of allocation algorithms is also needed.

This avenue includes several possibilities:

- Examining policy issues for charging, such as pricing strategies that servers can adopt, fairness in pricing, and distribution of charges among customers.
- Defining policies and evaluating algorithms for resource allocation.

- Devising algorithms for deadlock detection, recovery, and debugging.

Extensions to Distributed Systems and Networks

Currently the FOCS model is tailored to a shared-memory environment. It does not support ownership of machines, scheduling activities on different machines, and cross-machine service invocation. We have started to examine the possibility of extending the model to define a fully open, distributed computing systems, as well as to support sharing between fully open independent machines connected in a network.

This research avenue includes the following possibilities:

- Extending the model to support distributed transactions composed of "local" activities.
- Extending the notion of resource ownership to "ownership in-the-large," namely, ownership of resources across machine boundaries.
- Supporting server migration between heterogeneous machines, each of which runs a uniform OSB.

Appendix A

OSB services

A.1. Services for Activity Management

NewACD[1] (StartService, EndReturn, accounting permit, (args))
  returns { A-id, InvalidReturn, InvalidPrmt, NoMore };

DisposeACD (A-id)
  returns { OK, InvalidAid, NotOwner };

Invoke (binding, op [, rtn addr, NoRtn])
  returns [InvalidBndg, InvalidOp, InvalidRtn, InvalidSrvr];

Return [A-id, # of hops]
  returns [NotOwner];

Raise (A-id, ExcpType [, Value] [, UponRtn])
  returns { OK, InvalidAid, NoRight };

UnRaise (A-id [, ExcpType])
  returns { OK, InvalidAid, NoRight };

ReturnFromExcpt ( )
  returns ( );

SetACD (ACD permit, field, value)
  returns { OK, InvalidPrmt, NotOwner, Invalid... };

Check ([A-id] [, what])
  returns (quantum [, precedence] [, switched] [, excptns]);

Table A-1: Services for activity management

Activity Creation and Termination

NewACD : `StartService` is a binding for a service at which the new activity will start, or an address in the invoker's Space. `EndReturn` specifies the return address from `StartService`. `Args` is a list of arguments to `StartService`. `Accounting permit` is associated with the activity, to be

[1]We denote a service invocation simply as a function call
Operation (arguments) **returns** (results);
We use the following list notations:
( ) all items included; [ ] optional — one item or none; { } exclusive — one item only

used by servers in the activity's chain to charge the activity for using their services and resources. The invoker of *NewACD* becomes the activity's owner. If the request is satisfied, the returned `A-id` indicates the new activity's id. `NoMore` indicates that the request has failed because a system-wide or a per-scheduler limit on the number of activities has been reached.

DisposeACD terminates the activity `A-id` and frees its ACD. This request is rejected if the invoker is not the activity's owner.

Service Invocation and Return

Invoke : This service is used to invoke any other service. `Binding` points to a binding held by the invoker server, the current activity, or some other activity. `Op` names an operation of the service selected by the binding. `Rtn Addr` is an optional return address (default: the one following this invocation). `NoRtn` indicates that the invoker does not want to be returned to at the end of the invoked service. An error result indicates whether the binding, the operation, or the server id is incorrect. *Invoke* does not return any result if the invocation takes place — a result is returned by the invoked service or by the OSB in the event of enforced abortion of that service.

Return : A return from a service to the address indicated in the ACD. It always succeeds, even if the returned-to server has died. A *Return* with arguments can be invoked by the activity's owner. The `# of hops` can be *all* (i.e., emptying the activity's chain), *last n* or *first n* (i.e., removing from the activity's chain *n* hops or all but the first *n* hops, respectively).

Exception Raising

Raise and *UnRaise :* For symmetry, an exception can be removed by the server who has *Raise*-d it. `A-id` names the target activity. `ExcpType` can be any. `Value` is an optional argument to the exception handler. *Raise* with `UponRtn` may be used by any server in the activity's chain; it means that the exception would be noticed when the invoker is executed by activity `A-id`. Otherwise, the invoker should be the activity's owner or current server.

ReturnFromExcpt indicates the end of handling an exception or a trap. For efficiency, *ReturnFromExcpt* can be combined with service *Return* and procedure return.

Miscellaneous

SetACD allows any server who presents a valid permit to an ACD to set certain fields in it, such as its bindings, accounting permit, and priority indicator. The owner of the activity can also name another server as an owner — either immediately or upon the former's death.

Check lets any server discover the remaining quantum of activity `A-id` (default: current activity), its precedence level or exceptions pending for the activity, or whether the activity is dispatched or *Switch*-ed to.

A.2. Services for CPU Management

GetCPU (# of CPU's, (demand list), ...)
  returns (allocation list);

Allocate (to server, CPU, slice [, A-id])
  returns { OK, InvalidSrvr, InvalidCPU, InvalidSlice, InvalidA-id, NotOwner };

Revoke (from server, CPU, slice [, reason])
  returns { (revocation list), InvalidSrvr, InvalidCPU, NotOwner };

Release { reservation #, ([CPU,] [slice]) }
  returns { OK, InvalidCPU, InvalidSlice, NotOwner };

Dispatch (A-id [, CPU] [,quantum])
  returns { OK, InvalidQntm, InvalidAid, NotActOwner,
    InvalidCPU, NotCPUOwner, AlreadyRunning };

ChangeQntm (A-id [±] [,quantum])
  returns { OK, InvalidAid, InvalidQntm };

Switch (A-id [, subquantum])
  returns { OK, InvalidAid, InvalidQntm, NoCurrent, AlreadyRunning };

Switch ( )
  returns [NoFormer];

Register (A-id [, buffer])
  returns { OK, InvalidAid, InvalidBffr };

YourCustomer (accnt permit, TU's, id, ...)
  returns { OK, InvalidId, InvalidPrmt, InvalidTU };

Table A-2: CPU management services

Control of CPU Ownership

GetCPU : This service is invoked by the CPU host's customers of CPU allocation. `# of CPU's` specifies the minimal number of CPU's the invoker needs. It can be *any*, *all*, or a specific number. The `demand list` consists of the sublists (TU's [,urgency] [,cont] [,reservation #]). `TU's` specifies the amount of Time Units requested, which can be *any*; `urgency` indicates the urgency level of the request, if any. In order to establish a standard range of urgency values to which all servers comply, `urgency` is the distance from a real or an imaginary deadline, specified in Time Units. `cont` means that this demand is a continuous one, to be renewed when the allocated slice is consumed. This feature saves repetitive invocations of *GetCPU*. The `reservation #` is used as a tag, so a regretful server can *Release* a demand even before it has been granted. The CPU host's decision is made known through the returned `allocation list`, which consists of the sublists (which CPU, TU's). Obviously, this list might be empty. A later allocation decision of the CPU host is made known through a notification. *GetCPU* service can be extended with additional arguments, such as a `Reason` to justify this demand.

Allocate : This service is used by a server to allocate a portion of its ownership (`CPU` and `slice`) to `to server`. `CPU` and `slice` may indicate *all* that the invoker owns. If the allocation is valid and `CPU` names the CPU which the invoker is currently using, then the invoker is suspended until the allocation is consumed. `A-id` indicates which activity to schedule to consume the allocation. It can be *myself*. Otherwise, the activity should be owned by the allocator or the recipient. The default activity to be dispatched is the recipient's preregistered *scheduling activity* (see *Register*). A possible extension of this service is to let the invoker specify a list of allocations.

Revoke is used by a former owner of a given slice to reclaim it. `From server` specifies the server to whom the slice was allocated earlier. `From server` and `CPU` each may name a particular one, *any* or *all*. `slice` specifies Time Units or *all*. If the named server had further allocated that slice, it is reclaimed from those allocated servers(s) — as much as is necessary to satisfy the *Revoke* request. An *Interrupt* request is equal to *Revoke* (*any*, *any*, *all*). A possible extension of *Revoke* is to let the revoker specify a list of revocations. The returned `revocation list` specifies the unconsumed slices that the CPU host salvaged from the revoked server(s), if any. This list consists of the sublists (CPU, slice). A revoked scheduler is notified of the revocation by the CPU host. `Reason` is an optional argument to be passed to the revoked server via this notification.

Release is invoked to give away CPU ownership. If `reservation #` is specified, the relevant demand(s) are cancelled; this argument can specify *all*. Alternatively, the specified `CPU` and `slice` are released. `CPU` indicates a specific one or *all* (default: the one used to invoke this service). `Slice` specifies Time Units (default: the remainder slice of the invoker for the released CPU(s).

Context Switch between Activities

Dispatch : This service is used by a scheduler to dispatch activity `A-id` which the scheduler owns, for a given `quantum` of Time Units and on a given `CPU`. The default `CPU` is the one which the invoker uses. The default `quantum` is the remainder slice of the invoker for that CPU. If the named activity has been dispatched on another CPU, *Dispatch* is rejected.

Switch is used to switch execution to another activity `A-id`, of which the invoker is the current server. The `subquantum` (default: the remainder of the current activity's quantum) allows the invoker to set a time limit on this switching, after which the invoking activity resumes execution. If the switched-to activity is already running on another CPU, this request is rejected. The parameterless version of *Switch* is used to switch back.

ChangeQntm is invoked by activity `A-id`'s scheduler to increase or decrease the activity's quantum. `A-id` can be *myself*.

Handshaking with Schedulers

Register allows a server to register an activity `A-id` as its *scheduling activity*. `A-id` can be the current activity (*CurrAct*). The server can supply a `buffer` into which the CPU host places notifications of CPU allocation, revocation, and consumption.

YourCustomer : This service is invoked by a server who wants to become the CPU host's customer of CPU allocations. An accounting permit (`accnt permit`) is supplied, which the CPU host uses to charge for demands and for allocations. `TU's` is an initial allocation request, specified in *Time Units*. `Id` is a password or a customer's name, used to identify the invoker.

A.3. The Accountant's Services

Basic Mechanism
NewAccount (merged account id, mkey, credit [, debit limit] [, new key]) **returns** { (account id, key), Invalid... };
DisposeAccount (account id, key [, subaccount ids]) **returns** { OK, Invalid... };
MergeAccounts ((ToAccount id, key), (account id, key) [, subaccount ids]) **returns** { OK, Invalid... };
ChangeKey (account id, key [, new key]) **returns** { key, InvalidAcnt, InvalidKey };
Verify (cheque) **returns** { OK, InvalidChq };
Deposit (amount [, cheque] [, ToAccount]) **returns** { OK, MaxCharge, InvalidChq, InvalidAcnt };
Extended Mechanism (Additional Services)
Cancel (cheque) **returns** { OK, InvalidChq };
Verify (cheque) **returns** { OK, InvalidChq, Cleared };
PartialDeposit (amount [, cheque] [, ToAccount]) **returns** { OK, new cheque, MaxCharge, InvalidChq, InvalidAmnt, InvalidAcnt };
SplitCheque (Sum, cheque) **returns** { (new cheque 1, new cheque 2), Invalid... };

Table A-3: The Accountant's services

Basic Mechanism

A cheque is the pair (FromAccount, Signature), where
Signature := **Sign** (FromAccount, Account's private key), and *Sign* is a publicly known one-way function.

NewAccount is invoked to open a new account, which becomes the subaccount of `merged account`. `mkey` is the merged account's private key. It is supplied to prove that the invoker has the right to open subaccounts for that account. The new account is assigned a private `key`, which is the same as `new key` if the creator has supplied one. The owner can set the account's initial `credit` and `debit limit`.

DisposeAccount closes an account (`account id`), or its subaccounts if `subaccount ids` is supplied. The invoker must be the account's owner, that is, it must present a valid `key` for the account. For any account that is closed, its subaccounts are closed too, recursively. Any cheque referencing a closed account is invalidated.

MergeAccounts makes `account id` or its `subaccount ids` become the subaccounts of `ToAccount id`. The invoker must present valid `key`s for both accounts.

ChangeKey allows changing the `key` of an `account id`, which effectively invalidates any outstanding cheque of that account.

Verify returns whether the `Signature` of `cheque` is valid. *Deposit* is used to charge the `FromAccount` specified in the `cheque`. The default `cheque` is the current activity's one. This account is debited with `amount`. `ToAccount` (if specified) is credited with the same amount. The Accountant verifies *Signature* by using the same *Sign* function and `FromAccount`'s key. *Deposit* is refused if the cheque is invalid, `FromAccount` does not exist, or if the balance in that account will exceed the debit limit with this operation. `MaxCharge` indicates the maximal amount one can charge at that point.

Extended Mechanism

The operations of the extended mechanism assume that a cheque is either the quadruple

FromAccount	Serial #	Max amount	Signature

or the tuple

From Account	Serial #	Max amount	Max each time	Time of charge	Time interval	Signature

where the *Signatures* are analogous to the former one. `Time of charge` is the earliest time the cheque can be deposited. The `serial #` can be *none*. The `Max amount`, `Max each time`, `time of charge`, and `time interval` can be *any*.

Cancel allows any holder of `cheque` to cancel it, so it cannot be deposited later. *Verify* validates the `Signature` of the `cheque` and that it has not been cleared before (that is, deposited or cancelled).

Deposit is now restricted by the `Max amount` prescribed in the cheque, which is the default if `amount` is missing. The Accountant verifies that the cheque has not been cleared before. If the larger cheque format (seven-tuple) is used, *Deposit* can be invoked no sooner than the `time of charge` prescribed in the cheque, and `amount` is limited to `Max each time`.

PartialDeposit is similar to *Deposit*, except that only `amount` is debited or credited. In addition, if the larger cheque is used, the Accountant verifies that `amount` does not exceed `Max each time`, and that `time of charge` does not exceed the current time. If the operation is valid and `amount` is smaller than `Max amount`, then the Accountant returns a `new cheque`. This cheque has the same `FromAccount` as the former one, has a new `serial #`, and its `Max amount` is the former one reduced by `amount`. In the large cheque format, the new `time of charge` is set to be the former one increase by `time interval`. The cheque is *Sign*-ed by the account's private key, which is known to the Accountant.

SplitCheque allows a holder of `cheque` to split it into two cheques, one for the specified `amount` and the other for the remainder of `Max amount`. Both cheques are written under `FromAccount` and as before, *Sign*-ed with the account's secret key.

The Accountant's Algorithm

The Accountant's algorithm is used to record and to check the serial numbers of cheques that have cleared. The algorithm is designed to be efficient both in execution time and storage space, in order to reduce the Accountant's overhead in verifying cheques. For efficiency, it is assumed that servers usually issue cheques in sequential order, and that most of the cheques are cleared roughly in the order they have been issued. A lost cheque, that is, one whose only holder terminates, is removed from the Accountant's store.

Associated with each account are two sets of serial numbers of the cheques that have not yet cleared. One set is for the serial numbers assigned by the account owner, the other for those assigned by the Accountant. Since these sets are infinitely large, the Accountant keeps two windows of size w per account. For a faster search, each window is a list of sequential numbers. It is represented by a base number (first in the window) and a bitmap of length w. Clearing a cheque follows these steps:

(1) Initially all numbers in a window are unmarked.

(2) Suppose the cheque's number falls in the relevant window. If it is already marked invalid, then the cheque is rejected; else, the number is marked invalid.

(3) Suppose the number is *beyond* the relevant window, that is, the cheque is *newer*. The window moves to include the new number, which is marked invalid. When the window moves, some numbers of cheques which have not been yet cleared (that is, older cheques) may drop out of the window. They are added to a database of old cheques.

(4) Otherwise the number is *below* the relevant window, that is, it is an old cheque. The database is searched. If the number is found there, then it is deleted from the database; otherwise, the cheque is rejected.

(5) For search efficiency, the database is organized in a hierarchy of stores, from old to very old serial numbers. Numbers in the lower level are removed after that level is filled or after a threshold period, assuming they became lost cheques.

Notice that the algorithm is correct, albeit inefficient, when the above assumptions do not hold. If the account is owned by several servers, each of which issues cheques in different orders, or if a single owner issues cheques out-of-order, then windows change rapidly, increasing the expected search time in the account's database. If the owners issue multiple cheques of the same serial number, then all but the first of them are invalid, and the algorithm will reject them. If an account owner has written a cheque for gradual use over a long time, the cheque is renewed each time it is *PartialDeposit*-ed, and thus never becomes obsolete. If a holder of a cheque dies and the cheque is lost, the Accountant's algorithm will eventually spill it out of the database and not record it forever. To improve storage efficiency, the Accountant charges for space occupied by each account's database, encouraging servers to cancel cheques they will no longer use.

Appendix B

An Example Standard Interface

Activity Management	
Pause [how long] **returns** ( );	
Block ( ) **returns** ( );	
UnBlock [A-id] **returns** [InvalidA-id, NotAllowed];	
TerminateActivity ([A-id] [, Reason]) **returns** [InvalidA-id, NotAllowed];	
Prio ([A-id] [urgency] [Reason]) **returns** { OK, InvalidA-id };	
AbortService (A-id, Reason) **returns** { OK, InvalidA-id };	
Memory Management	
Pin (virt mem access permit [, translate]) **returns** { OK, InvalidPrmt, phys mem access permit };	
UnPin (virt mem access permit) **returns** { OK, InvalidPrmt };	
Store (access permit, data) **returns** { OK, InvalidPrmt };	
Fetch (access permit) **returns** { data, InvalidPrmt };	
General Operations	
TerminateBinding [server id] **returns** { OK, InvalidBndg };	
ExcptHandler ([server id], exception type [, args]) **returns** ( );	
Return Codes	
ServiceAborted_ServerDied	Return enforced because the invokee has terminated
Exception Types	
GonnaTerminate	The activity will terminate.
AbortCurrService < Reason >	As requested by a server in an activity's chain

Table B-1: Standard operations and parameters

Activity Management

These operations may be invoked by servers in the activity's chain. In operations where activity id (`A-id`) is omitted or is optional, the request refers to the current activity. *Pause* indicates that the invoker releases the CPU because it has nothing to do. `How long` is a *hint* to the activity's scheduler to avoid dispatching that activity for that time frame, specified in Time Units. *TerminateActivity, Prio*, and *AbortService* are used to tell an activity's scheduler to terminate the activity, to increase or reduce the activity's priority, or to indicate to the activity's current server to abort its service, respectively. `Reason` is one of a standard list or an application-specific one. We have not enumerated a comprehensive list. As an example, `Reason` for *Prio* can be: *EnterCriSec*, *LeaveCriSec*—indicating that a critical section is entered or exited, respectively, *RealTime*—indicating that this activity must meet a real-time deadline, and *BlockingOthers*—indicating that the activity holds an internal resource that delays other activities. In order to establish a standard range for urgency levels, the `urgency` for *Prio* is specified as the distance from a real or an imaginary deadline. A *none* urgency level (the default) means that the activity's priority can be reduced to a former level.

Memory Management

Pin and *UnPin* are provided by Universe Managers. The `virt mem access permit` indicates which area of virtual space to fix or unfix in physical memory. `Translate` indicates that the invoker wants an access permit to physical memory to be returned. This translation, for example, might be required by a p-host in order to access a buffer referenced by `virt memory access permit` more efficiently. *Store* and *Fetch* are operations provided by a memory host.

General Operations

TerminateBinding is an operation associated with any binding. It may be invoked on behalf of the binding holder (`server id`) by a server that terminates the holder. The default `server id` is the invoker. *ExcptHandler* is a generic exception handler, invoked for any exception (including traps). The `exception type` is one of the types defined below or of the types defined by the CPU host, e.g. *InvalidAddress* and *InvalidInstruction*. `Server id` indicates that the exception relates to another server, not the invokee. This is the case when the U-mgr or a debugger of `server id` is invoked to handle an exception that should be handled by that server. `Args` are specific to each exception type. For example, an argument can be the address where the exception occurred.

Return Codes

The only return code listed here might be returned by a CPU host, U-mgr, or any other server on behalf of an aborted service.

Exception Types

GonnaTerminate and *AbortCurrService* are exceptions raised by an activity's owner.

Appendix C

System Provided Services

C.1. Services of the COS Universe Manager

New (accounting permit, (structure info), (sharing info) [, Dispose handler])
returns { Space id, InvalidPrmt, InvalidSegs, InvalidShr, NoMore };

Pin (* See Appendix B *)

UnPin (* See Appendix B *)

Fix (start addr, size) **returns** { OK, InvalidAddr };

UnFix (start addr, size) **returns** { OK, InvalidAddr };

MemCopy (from, to [, copy-on-wrt] [, align {page, seg}])
returns { OK, InvalidPrmt, permit to a new area };

Expand (size, acc Rts, [, Space id] [, seg #] [, shared seg] [, copy-on-wrt])
returns { OK, InvalidSeg, InvalidId, Invalid... };

Dispose [Space id] **returns** [OK, InvalidId, NotOwner];

Freeze (Space id) **returns** (new id, InvalidId, NotOwner);

UnFreeze (new Space id)
returns (OK, InvalidId, NotOwner);

SnapShot (Space id) **returns** ((seg #, info), ..., (seg #, info));

Table C-1: Memory management services

New is invoked to get a Space. `structure info` is list of (seg #, size, access Rts) per segment, and indicates the initial structure of the allocated Space. `sharing information` indicates that a given segment should be a copy of an existing segment or share its mapping. The invoker should have a valid permit to the shared segment or own the Space containing the segment. For simplicity, shared segments must be in the Universe of the COS. The `accounting permit` is needed by the U-mgr to charge for resources such as virtual store, frames, and disk blocks used for swapping. `Dispose handler` is an address or a binding for a handler, to be invoked by the U-mgr when the server stored in the new Space terminates itself. `NoMore` indicates that the request has failed because a limit on the number of allocated Spaces per Universe has been reached. If the request succeeds, a new `Space id` is returned, of which the invoker becomes the owner.

Pin and *UnPin* are standard-interface operations (see Appendix B). *Fix* is used similarly to request that a given region remains core resident. However, *Fix* is invoked by the owner of the Space in which the region resides, and it is assumed that the fixing is required on a long-term basis.

MemCopy is used to copy across address spaces. `From` and/or `to` are access permits to another server's Space. Each can be an address in the invoker's Space or in another Space owned by the invoker. `Copy-on-wrt` indicates that the target buffer should be shared, and that a copy should be performed if either the target or the source buffer is modified. This feature is allowed only if both the source and the target buffers are in the Universe of the COS. `To` may indicate that a new virtual area should be allocated with the copy. In this case, an optional `align page` or `align seg` indicates whether the new area is aligned on a page or segment boundary.

`Expand` is a request to create a new segment of the indicated `size`, `access Rts`, and (optional) number. If the number indicates an existing segment, then this segment is expanded. `Space id` indicates in whose Space the segment is created (default: *myself*). The invoker should be the owner of that Space. `Shared` indicates that the new segment should share the mapping of the specified `seg`, possibly with **copy-on-wrt** as above.

Dispose is invoked to terminate oneself or to release a Space of a terminated server. The invoker should be the owner of `Space id`. The default `Space id` is *myself*, in which case the operation succeeds (with no reply). In this case, the `dispose handler` specified in *New* is invoked and notified of the event. The Space id is invalidated by the U-mgr when the Space is released.

Freeze is invoked by the owner of `Space id` (other than the server stored there) in order to inspect the state of the Space, correct errors, or copy its contents to an auxiliary storage space. While the Space is frozen, its id is changed to `new id` so that accesses to that Space are refused. The owner can later *Dispose* the Space or *Unfreeze* it, in which case its former id is restored. *SnapShot* allows the owner of `Space id` to obtain information about every segment of that Space. This information includes the segment's size and indications whether the segment has been expanded or has been modified.

C.2. The Matchmaker's Services

```
Expose ( server name,  service name,  binding  [, service type]  [, once-only]
         [, accounting permit]  ( add'l args ) )
  returns { OK,  Duplicate,  InvalidPrmt };

Bind ( [ server name ],  { (service names), (service types) } )
  returns { binding,  NoMatch,  TooMany };

Close ( {server name, server id} [, service name] )
  returns { OK,  NotAllowed,  NoMatch };

Revive ( server name,  secret id,  binding,  loading info  [, loader name] )
  returns { OK,  InvalidLdrName };

IamAlive ( server name,  secret id )
  returns { OK,  InvalidName,  InvalidId ;
```

Table C-2: The Matchmaker's services

Expose is used to announce a service and to furnish a `binding` for it. The invoker should be the service provider—that is, *PrevSrv* should match the server id in the `binding`'s target address—so that another server cannot expose a service which its provider does not want to be exposed. The service is identified by a `server name` and a `service name`, each of which is a string of characters up to some implementation-dependent length. If some other server has already deposited a binding with the same server name, this invocation is rejected with the `Duplicate` indication. The service provider may associate `service type` with the binding. `Once-only` is a hint to the Matchmaker to remove the binding from its store after the first customer has acquired the binding. This feature is useful when a server expects a single customer, e.g. when cooperative servers form a pipeline configuration. An `accounting permit` is supplied if the server has not supplied one before. The Matchmaker uses this permit to charge for services and storage space. A list of `additional arguments` can be supplied to *Expose*, to be passed to an acquirer of this binding. This list conceivably specifies service attributes that the acquirer should know before using the binding.

Bind is used to locate a service and obtain its binding. The customer may specify a list of `service names` or a list of `service types`, each of which may contain an implementation-dependent number of items. A `server name` can be omitted. A binding is returned if a service name or a service type matches one of the items in the respective list. If several bindings match the specified names or types, `TooMany` is indicated.

Close is invoked to declare that a service is no longer provided, and hence the Matchmaker should remove its binding. The `service name` may specify *all* (the default). If *Close* is invoked by other than the provider of that service, then the specified `service name` (or *all*) is closed *only* if the provider has terminated.

Revive is invoked to tell the Matchmaker how to restart a server if its services are requested after the server has terminated. The invoker supplies a `binding` for a loading service of some "loader". That service is invoked by the Matchmaker to regenerate the focal server. The descriptive `loading info` is passed to the loader to indicate whom to load. The invocation protocol of the loading service should be in accordance with a protocol defined by the Matchmaker. If it is possible that the loader would not exist when needed, then *Revive* should specify a `loader name` too. The Matchmaker will attempt to regenerate that loader prior to loading the focal server, recursively, provided the loader has left a *Revive* notification too. The `Secret id` is used to authenticate the server in *IamAlive*. The bindings of a terminated server who has left a *Revive* notification are "frozen" and not deleted from the Matchmaker's store.

IamAlive is invoked by a revived server, regardless whether its revival was triggered by the Matchmaker or another server. If `secret id` matches the one associated with a server that has previously invoked *Revive*, then all the frozen bindings of the latter server are reactivated.

Appendix D

Algorithms of Three OSB services

D.1. The *Invoke* Service

```
entry Invoke ( Binding, Op [, Rtn addr, NoReturn ] ) returns
    [ InvBndg,  InvOp,  InvSrvr,  InvRtn ];

  ( Addr, Key ) := fetch Binding's target address and key;
    (* if Binding is an activity's binding, it is found in the ACD;
       check also that the invoker is in the activity's thread *)
  if fetch is rejected then return ( InvBndg ); fi;
    (* if fetch causes a page fault —
        the appropriate handler is invoked by the OSB elsewhere *)
  current Space := Addr . Space id;        (* it's a Space switch *)
  Lock := fetch the lock at Addr;
  if fetch is rejected then return ( InvSrvr ); fi;
  if Key ≠ Lock then return ( InvBndg ); fi;
  OpAddr := fetch Op by indexing Addr;
  if fetch fails then return ( InvOp ); fi;
  (* Note: The following compound IF is eliminated
         in a registers-window CPU architecture *)
  if Rtn addr ≠ null then
    if Rtn addr ⊂ current Space then
      push Rtn addr on CurrAct's return-addresses stack;
    else return ( InvRtn );
    fi;
  elsif not NoReturn then
    push program counter on CurrAct's stack of return addresses;
  fi;
  PrevSrv := CurrAct's current server;
  CurrAct's current server := current Space;
  program counter := OpAddr;
    (* continues to execute at the invoked Op *)
end Invoke;
```

D.2. The *Dispatch* Service

```
entry Dispatch ( Aid [, CPU ] [, quantum ]) returns
    { OK, InvAid, InvCPU, InvQntm, NotActOwner, NotCPUOwner,
          AlreadyRunning };

  if invalid ( Aid | CPU | quantum ) then
    return ( InvAid | InvCPU | InvQntm ); fi;
  if CurrSrv ≠ Aid's owner then return ( NotActOwner ); fi;
  if CPU = null then CPU := thisCPU; fi;
  if CurrSrv ≠ CPU's owner then return ( NotCPUOwner ); fi;
  if Aid -> State = running then
    if Aid is in a Switch queue then split the queue;   (* see §4.2.3 *)
    else return ( AlreadyRunning );
  fi; fi;
    (* so far so good *)
  if CPU = thisCPU then ContextSwitch ( Aid, CPU, quantum );
    (* the doomed activity resumes here when redispatched *)
  else
    halt CPU;        (* to avoid a race condition *)
    signal CPU to ContextSwitch ( Aid, CPU, quantum );
    return ( OK );      (* and do NOT block *)
  fi;
end Dispatch;

procedure ContextSwitch ( Aid,  CPU,  quantum ) returns ( ) ;
  DoomedAid := current activity at CPU;    (* doomed for preemption *)
  DoomedAid -> State := idle;
  add DoomedAid's remaining quantum to DoomedAid -> owner -> slice;
  push ( registers ) unto DoomedAid's context descriptor;
  (* optional: update statistics: total CPU time; idle since now *)
  Aid -> State := running @ CPU;
  if (quantum = null) | (quantum > Aid -> owner -> slice) then
    quantum := Aid -> owner -> slice; fi;
  Aid -> owner -> slice := Aid -> owner -> slice - quantum;
  SetTimer ( quantum );
  (* optional: update statistics: not idle *)
  CurrAct := Aid;      (* CPU's current activity *)
  pop ( registers ) from Aid's context descriptor;
   (* including CurrSrv, PrevSrv, and the program counter *)
   (* continues now at the popped program counter *)
end ContextSwitch;
```

D.3. The *Allocate* Service

```
entry Allocate ( to server, CPU, slice [, Aid] ) returns
      { OK, InvSrvr, InvCPU, InvSlice, InvAid, NotOwner };

   if invalid ( to server | CPU | slice | Aid ) then
      return ( InvSrvr | InvCPU | InvSlice | InvAid ); fi;
   A := allocator's list of ownership;
      (* find invoker's (≡ CurrSrv's) entry in "schedulers information area",
        fetch and lock it *)
   if A = ( ) then return ( NotOwner); fi;
   B := to server's list of ownership;        (* similar to A *)
   if B = ( ) then create an entry for it in scheduler's information area; fi;
   if Aid = null then
      Aid := recipient's dispatching activity;
      if Aid = null then return ( InvAid ); fi;
   fi;
   if CPU = all then
      for each ($CPU_x$, $slice_x$) in A do
         if A2B (A, B, slice, $CPU_x$, $slice_x$) then cw := true; fi;
      rof;
   else
      find ($CPU_x$, $slice_x$) in A so that CPU = $CPU_x$ ;
      if null then return ( NotOwner ); fi;
      cw := A2B (A, B, slice, $CPU_x$, $slice_x$);
   fi;
   if cw then ContextSwitch ( Aid, thisCPU, slice); fi; (* see above *)
      (* will resume here when the former activity gets redispatched *)
   return ( OK );
end Allocate;

procedure A2B ( A, B, slice, $CPU_x$, $slice_x$ ) returns Boolean;
   if (slice = all) | (slice > $slice_x$) then slice := $slice_x$; fi;
   push ($CPU_x$, slice) unto B;
   replace ($CPU_x$, $slice_x$) in A by ($CPU_x$, $slice_x$ - slice);
   if $CPU_x$ = thisCPU then
      push recipient on thisCPU's owners stack;
      return ( true );         (* namely, do ContextSwitch *)
   else
      halt $CPU_x$;      (* to avoid a race condition *)
      push recipient on $CPU_x$'s owners stack;
      signal $CPU_x$ to ContextSwitch ( Aid, $CPU_x$, slice ); (* see above *)
      return ( false );
   fi;
end A2B;
```

GLOSSARY

The following is a brief summary of the terms introduced in Chapter 3.

Accountant	A server of the OSB which provides services to manage accounts, charge and credit them.
Activity	A thread of control that executes services. It is an independently schedulable entity that can span multiple servers.
Allocation	Transfer of ownership of a resource.
Application	Any logical collection of non-OSB servers.
Binding	A reference to a service that enables invocations of that service. It is a light-weight capability.
Chain	A sequence of services (servers) which have been invoked by a given activity and which have not yet returned.
Clock host	A host of the OSB, necessary to indicate time expiration.
Computing System	A collection of servers (of which one distinguished set of servers is the OSB). It includes p-hosts and other servers.
COS	An application which provides services customarily provided by operating systems.
CPU host	A server included in the OSB which encapsulates all the CPU's in the system.
Current Server/Service	The last server (service) in an activity's dynamic chain.
Current Space	The Space in which the currently running activity (per CPU) is running.
Customized OS	See COS.
Dispatch	A service of the CPU host which allows an activity to execute on a specified CPU.
Host	A server that encapsulates a given resource.
Initiator	A server of the OSB which creates the OSB at system-initialization time.
Matchmaker	A server that allows service providers to announce their services by depositing bindings, and service users to obtain the bindings.
Memory	A distinguished type of resource, necessary to store servers.
OSB	A collection of generally-trusted servers which provide the minimal set of services necessary to protect resources and servers.
Operating System Base	See OSB.

Ownership	Allows a server to access the owned resource, to issue permits, and to allocate the resource. See Allocation.
Permit	A reference to a resource that enables accessing the resource with specific access rights. It is a light-weight capability.
P-host	Denotes a host of a physical resource. A p-host may have private processors and memories.
Processor	A distinguished type of resource, necessary for execution. The shared resources are called CPU's. P-hosts may have private processors.
Resource	An entity encapsulated within one server—its host. A resource is divided to units of allocation, each of which can be owned by multiple servers.
Revocation	Reclaiming a resource from its current owner(s), which then loose their ownership. Revocation is allowed to any former owner of the resource.
Server	A dynamic representation of a program that implements services.
Service	An abstraction of a set of actions that uses resources to carry out a logical function. A service can be performed synchronously or asynchronously.
Space	A reference environment in which a server is stored. It is the unit of allocation of a Universe.
Standard Interface	A set of conventions that defines the invocation protocols for a given set of services. These are services which a server might be *obliged* to invoke without knowing their invocation protocols.
System Administrator	A user, the primordial owner of all accounts, whose servers are the primordial owners of the system resources.
System Resources	An installation-dependent list of resources shared by all servers. The host of these resources must be in the OSB.
Universe	A logical resource which stores multiple servers. It is composed of Spaces and is mapped into physical memory.
User	A person or a group of people using the computing system. A user is represented in the system by servers and activities.

REFERENCES

[1] *470 V/6 Machine reference manual*, Amdahl Corp., 1976.

[2] *The C/70 Macroprogrammer's Handbook*, Bolt, Beranek and Newman, Inc., Cambridge, MA, 1980.

[3] *VAX-11 Architecture Reference Manual*, Digital Equipment Corporation, Bedford, MA, May 1982.

[4] *MC68000 User's Manual*, Motorola Inc. and Prentice-Hall, Englewood Cliffs, NJ, 1982.

[5] *Balance 8000 System Technical Summary*, Sequent Computer Systems, Portland, OR, December 1984.

[6] *Virtual Machine/System Product: System Programmer's Guide (IUCV facility)*, IBM Corporation, 1984.

[7] *Mesa Programmer's Manual*, XEROX Corporation, Palo Alto, CA, November 1984.

[8] *Butterfly Parallel Processor Overview (Draft, Version 1)*, BBN Laboratories, June 13, 1985.

[9] *Multimax technical summary*, Encore Computers Corporation, Marlboro, MA, 1986.

[10] *Dynix Programmer's Manual*, Sequent Computer Systems, Portland, OR, 1986.

[11] *UMAX 4.2 Programmer's Reference Manual*, Encore Computers Corporation, Marlboro, MA, 1986.

[12] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tavanian, and M. Young, ''Mach: A new kernel foundation for Unix development,'' *Proc. of USENIX Summer Conference*, USENIX Association, July 1986, pp. 93-111.

[13] A. V. Aho and J. D. Ullman, *Principles of Compiler Design*, Addison-Wesley, Reading, MA, April 1979.

[14] G. T. Almes, A. P. Black, E. D. Lazowska, and J. D. Noe, ''The Eden system: A technical review,'' *IEEE Trans. on Software Eng.* SE-11:1 (January 1985), pp. 43-58.

[15] Y. Artsy, H-Y Chang, and R. A. Finkel, ''Interprocess communication in Charlotte,'' *IEEE Software* 4:1 (January 1987), pp. 22-28.

[16] Y. Artsy, H-Y Chang, and R. A. Finkel, ''Charlotte: Design and implementation of a distributed kernel,'' Computer Sciences Technical Report 554, University of Wisconsin–Madison, August 1984.

[17] C. R. Attanasio, ''801 architecture support for database – A case study,'' Research Report RC-12416 Revised, IBM T.J. Watson Research Center, Yorktown Heights, NY, January 1987.

[18] T. P. Baker and G. A. Riccardi, ''Implementing Ada exceptions,'' *IEEE Software* 3:5 (September 1986), pp. 42-51.

[19] F. Baskett, J. H. Howard, and J. T. Montague, ''Task communication in Demos,'' *Proc. of the Sixth Symposium on Operating Systems Principles*, November 1977, pp. 23-31.

[20] D. S. Batory, ''Principles of database management systems extensibility,'' *Database Engineering* 10:2, IEEE Computer Society (June 1987), pp. 40-46.

[21] C. G. Bell, J. C. Mudge, and J. E. McNamara, ''The PDP-8 family'' and "The evolution of the PDP-11,'' pp. 175-208 and 379-408 in *Computer Engineering: A DEC View of Hardware Systems Design*, Digital Press, Maynard, MA, 1979. Reprinted in D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples*, McGraw-Hill, NY, 1982, pp. 767-784.

[22] C. G. Bell, ''Multis: A new class of multiprocessor computers,'' *Science* 228 (April 1985), pp. 462-467.

[23] A. Bensoussan, C. T. Clingen, and R. C. Daley, ''The Multics virtual memory: Concepts and design,'' *Commun. of the ACM* 15:5 (May 1972), pp. 308-318.

[24] P. A. Bernstein and D. B. Lomet, ''CASE requirements for extensible database systems,'' *Database Engineering* 10:2, IEEE Computer Society (June 1987), pp. 2-9.

[25] A. D. Birrell and B. J. Nelson, ''Implementing remote procedure calls,'' *ACM Trans. on Computer Systems* 2:1 (February 1984), pp. 39-59.

[26] G. A. Blaauw and F. P. Brooks, Jr., ''The structure of System/360,'' *IBM Sys. Journal* 3:2 (1964), pp. 119-135. Reprinted in D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples*, McGraw-Hill, NY, 1982, pp. 695-710.

[27] A. P. Black, ''Supporting distributed applications: Experience with Eden,'' *Proc. of the Tenth Symposium on Operating Systems Principles*, December 1985, pp. 181-193.

[28] A. P. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter, ''Distribution and abstract types in Emerald,'' *IEEE Trans. on Software Eng.* SE-13:1 (January 1987), pp. 65-76.

[29] G. S. Blair, J. R. Malone, and J. A. Mariani, ''A critique of UNIX,'' *Software — Practice and Experience* 15:12 (December 1985), pp. 1125-1139.

[30] D. G. Bobrow, J. D. Burchfiel, D. L. Murphy, and R. S. Tomlinson, ''TENEX, a paged time sharing system for the PDP-10,'' *Commun. of the ACM* 15:3 (March 1972), pp. 135-143.

[31] B. R. Borgerson, M. D. Godfrey, P. E. Hagerty, and T. R. Rykken, ''The architecture of the Sperry Univac 1100 series systems,'' *Proc. of the Sixth Annual Symp. on Computer Architecture*, ACM, April 1979, pp. 137-146.

[32] S. R. Bourne, ''Unix time-sharing system: The Unix Shell,'' *Bell System Technical Journal* 57:6 (July 1978), pp. 1971-1990.

[33] P. Brinch Hansen, ''Processor management,'' pp. 133-153 in *Operating Systems Principles*, Prentice-Hall, NJ, 1973.

[34] P. Brinch Hansen, ''Distributed Processes: A concurrent programming concept,'' *Commun. of the ACM* 21:11 (November 1978), pp. 934-941.

[35] S. T. Campbell, ''MAC-8 microprocessor summary,'' in *MAC-8 Systems Designers Handbook*, Bell Laboratories, March 1976.

[36] M. J. Carey, D. J. DeWitt, D. Frank, G. Graefe, J. E. Richardson, E. J. Shekita, and M. Muralikrishna, ''The architecture of the EXODUS extensible DBMS: A preliminary report,'' *Proc. of Int'l Workshop on Object-Oriented Database Systems*, September 1986, pp. 52-65. Available also as Computer Sciences Technical Report 644, University of Wisconsin–Madison (May 1986).

[37] D. D. Chamberlin, M. M. Astrahan, M. W. Blasgen, J. N. Gray, W. F. King, B. G. Lindsay, R. Lorie, J. W. Mehl, T. G. Price, F. Putzolu, P. Griffiths Selinger, M. Schkolnick, D. R. Slutz, I. L. Traiger, B. W. Wade, and R. A. Yost, ''A history and evaluation of System R,'' *Commun. of the ACM* 24:10 (October 1981), pp. 632-646.

[38] A. Chang and M. F. Mergen, ''801 storage: Architecture and programming,'' unpublished paper, IBM T.J. Watson Research Center, Yorktown Heights, NY.

[39] D. R. Cheriton, M. A. Malcolm, L. S. Melen, and G. R. Sager, ''Thoth: A portable real-time operating system,'' *Commun. of the ACM* 22:2 (February 1979), pp. 105-115.

[40] D. R. Cheriton and W. Zwaenepoel, ''The distributed V kernel and its performance for diskless workstations,'' *Proc. of the Ninth Symposium on Operating Systems Principles*, November 1983, pp. 129-140.

[41] D. R. Cheriton, ''The V Kernel: A software base for distributed systems,'' *IEEE Software* 1:2 (April 1984), pp. 19-42.

[42] D. R. Cheriton, G. A. Slavenburg, and P. D. Boyle, ''Software-controlled caches in the VMP multiprocessor,'' *Proc. of the 13th Int'l Symp. on Computer Architecture*, IEEE Computer Society, June 1986, pp. 366-374.

[43] H-T Chou and D. DeWitt, ''An evaluation of buffer management strategies for relational database systems,'' *Proc. of the 1985 VLDB Conf.*, August 1985.

[44] D. D. Clark, ''The structure of systems using upcalls,'' *Proc. of the Tenth Symposium on Operating Systems Principles*, December 1985, pp. 171-180.

[45] G. W. Cox, W. M. Corwin, K. K. Lai, and F. J. Pollack, ''A unified model and implementation for IPC in a multiprocessor environment,'' *Proc. of the Eighth Symposium on Operating Systems Principles*, November 1981, pp. 44-53.

[46] O. J. Dahl and K. Nygaard, ''Simula—an Algol-based language,'' *Commun. of the ACM* 9:9 (September 1966), pp. 671-678.

[47] M. DeMoney, J. Moore, and J. Mashey, ''Operating systems support on a RISC,'' *COMPCON '86*, March 1986, pp. 138-143.

[48] J. B. Dennis and E. C. Van Horn, ''Programming semantics for multiprogrammed computations,'' *Commun. of the ACM* 9:3 (March 1966), pp. 143-155. Reprinted in *Commun. of the ACM* **26**:1 (January 1983), pp. 29-35.

[49] E. W. Dijkstra, ''The structure of the "THE" multiprogramming system,'' *Commun. of the ACM* 11:5 (May 1968), pp. 341-346.

[50] D. Ditzel and R. McLellan, ''Register allocation for free: The C Machine stack cache,'' *Proc. of the Symposium on Architecture Support for Programming Languages and Operating Systems*, March 1982, pp. 48-54.

[51] K. E. Drexler and M. S. Miller, ''Incentive engineering for computational resources,'' in *Ecology of Computation*, ed. Bernardo Huberman, Elsevier Science Publishers, to appear, 1987.

[52] R. S. Fabry, ''Capability-based addressing,'' *Commun. of the ACM* 17:7 (July 1974), pp. 403-412.

[53] R. A. Finkel, M. L. Scott, W. K. Kalsow, Y. Artsy, H-Y Chang, P. Dewan, A. J. Gordon, B. Rosenburg, M. H. Solomon, and C-Q Yang, "Experience with Charlotte: Simplicity versus function in a distributed operating system," Computer Sciences Technical Report 653, University of Wisconsin–Madison, July 1986. Extended abstract appeared in the *Workshop on design principles for experimental distributed systems*, IEEE Computer Society, (16-17 October, 1986), held at Purdue University, West Lafayette, Indiana.

[54] R. A. Finkel, M. H. Solomon, D. DeWitt, and L. Landweber, "The Charlotte distributed operating system: Part IV of the first report on the crystal project," Computer Sciences Technical Report 502, University of Wisconsin–Madison, October 1983.

[55] R. A. Finkel, *An Operating Systems Vade Mecum*, Prentice-Hall, New York, 1986.

[56] R. Fitzgerald and R. F. Rashid, "The integration of virtual memory management and interprocess communication in Accent," *ACM Trans. on Computer Systems* 4:2 (May 1986), pp. 147-177.

[57] M. J. Forthofer and K. B. Adams, "Real-time control services for space shuttle ground support," *Proc. of the 7th Texas Conference on Computing Systems*, IEEE, October 1978, pp. 3.1-3.8.

[58] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, MA, 1983.

[59] R. P. Goldberg, "Survey of virtual machine research," *IEEE Computer* 7:6 (June 1974), pp. 34-44.

[60] D. Goldhirsch and J. A. Orenstein, "Extensibility in the PROBE database system," *Database Engineering* 10:2, IEEE Computer Society (June 1987), pp. 24-31.

[61] J. Gray, "Notes on database operating systems," pp. 393-481 in *Operating Systems: An Advanced Course*, Springer-Verlag, 1978.

[62] D. B. Gustavson, "Computer buses -- a tutorial," *IEEE Micro*, August 1984, pp. 7-22.

[63] A. N. Habermann, L. Flon, and L. Cooprider, "Modularization and hierarchy in a family of operating systems," *Commun. of the ACM* 19:5 (May 1976), pp. 266-272.

[64] J. L. Hennessy, "VLSI processor architecture," *IEEE Trans. on Computers* C-33:12 (December 1984), pp. 1221-1246.

[65] C. A. R. Hoare, "Monitors: An operating system structuring concept," *Commun. of the ACM* 17:10 (October 1974), pp. 549-557.

[66] R. M. Jensen, "A formal approach for communication between logically isolated virtual machines," *IBM Systems Journal* 18:1 (January 1979), pp. 71-92.

[67] M. B. Jones, R. F. Rashid, and M. R. Thompson, "Matchmaker: An interface specification language for distributed processing," *Proc. of the 12th ACM Symposium on Principles Of Programming Languages*, ACM, January 1985, pp. 225-235.

[68] A. K. Jones, "The Object Model: A conceptual tool for structuring software," pp. 8-16 in *Operating Systems—An advanced course*, ed. G. Seegmuller, Springer-Verlag, New York, 1978.

[69] A. K. Jones, ''Protection: Mechanisms and the enforcement of security policy,'' pp. 229-251 in *Operating Systems—An Advanced Course*, ed. G. Seegmuller, Springer-Verlag, New York, 1978.

[70] A. K. Jones, R. J. Chansler Jr., I. Durham, K. Schwana, and S. R. Vegdahl, ''StarOS, a multiprocessor operating system for the support of task forces,'' *Proc. of the Seventh Symposium on Operating Systems Principles*, 1979, pp. 117-127.

[71] K. C. Kahn, W. M. Corwin, T. D. Dennis, H. D'Hooge, D. E. Hubka, L. A. Hutchins, J. T. Montagus, and F. J. Pollack, ''iMAX: A multiprocessor operating system for an object-based computer,'' *Proc. of the Eighth Symposium on Operating Systems Principles*, November 1981, pp. 127-136.

[72] H. Katzan, *Invitation to Ada and Ada Reference Manual*, Petrocelli Books, NY, 1980.

[73] J. L. Keedy, ''On structuring operating systems with monitors,'' *ACM Operating Systems Review* 13:1 (January 1979), pp. 5-9.

[74] J. Kepecs and M. H. Solomon, ''SODA: A simplified operating system for distributed applications,'' Ph.D. dissertation Computer Sciences Technical Report 527, University of Wisconsin--Madison, January 1984.

[75] J. Kepecs, ''Light-weight processes for Unix implementation and application,'' *Proc. of USENIX Summer Conference*, USENIX Association, June 1985.

[76] J. Kepecs and M. H. Solomon, ''SODA: A simplified operating system for distributed applications,'' *Operating System Review* 19:4 (October 1985), pp. 45-56.

[77] R. B. Kieburtz and A. Silberschatz, ''Capability managers,'' *IEEE Trans. on Software Eng.* SE-4:6 (November 1978), pp. 467-477.

[78] R. B. Kieburtz and A. Silberschatz, ''Access-right expressions,'' *ACM Trans. on Programming Languages and Systems* 5:1 (January 1983), pp. 78-96.

[79] J. F. Kurose, M. Schwartz, and Y. Yemini, ''A microeconomic approach to decentralized optimization of channel access Policies in Multiaccess Networks,'' *Proc. of the 5th Int'l Conference on Distributing Computing Systems*, IEEE, May 1985, pp. 70-77.

[80] B. W. Lampson and R. F. Sproull, ''An open operating system for a single user machine,'' *Proc. of the Seventh Symposium on Operating Systems Principles*, December 1979, pp. 98-105.

[81] B. W. Lampson and D. D. Redell, ''Experiences with processes and monitors in Mesa,'' *Commun. of the ACM* 23:2 (February 1980), pp. 105-117.

[82] B. W. Lampson, ''Hints for computer system design,'' *Proc. of the Ninth Symposium on Operating Systems Principles*, October 1983, pp. 33-48.

[83] K. A. Lantz, K. D. Gradischnig, J. A. Feldman, and R. F. Rashid, ''Rochester's intelligent gateway,'' *IEEE Computer* 15:10 (October 1982), pp. 54-68.

[84] R. J. LeBlanc and C. T. Wilkes, ''Systems programming with objects and actions,'' *Proc. of the 5th Int'l Conference on Distributing Computing Systems*, May 1985, pp. 132-139.

[85] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf, ''Policy/ Mechanism separation in Hydra,'' *Proc. of the Fifth Symposium on Operating Systems Principles*, November 1975, pp. 132-140.

[86] H. M. Levy, ''The Cambridge CAP computer,'' pp. 79-101 in *Capability-Based Computer Systems*, Digital Press, Maynard, MA, 1984.

[87] H. M. Levy, ''The Hydra system,'' pp. 102-125 in *Capability-Based Computer Systems*, Digital Press, Maynard, MA, 1984.

[88] B. H. Liskov, ''The design of the VENUS operating system,'' *Commun. of the ACM* 15:3 (March 1972), pp. 144-149.

[89] B. H. Liskov, ''Guardians and Actions: Linguistic support for robust distributed programs,'' *Trans. on Programming Languages and Systems*, July 1983, pp. 381-404.

[90] J. McPherson and H. Pirahesh, ''An overview of extensibility in Starburst,'' *Database Engineering* 10:2, IEEE Computer Society (June 1987), pp. 32-39.

[91] B. P. Miller, D. L. Presotto, and M. L. Powell, ''Demos/MP: The development of a distributed operating system,'' *Software—Practice and Experience* 17:4 (April 1987), pp. 277-290. Also available as Computer Sciences Technical Report 650, University of Wisconsin–Madison, July 1986.

[92] M. S. Miller and K. E. Drexler, ''Markets and computation: Agoric open systems,'' in *Ecology of Computation*, ed. Bernardo Huberman, Elsevier Science Publishers, to appear, 1987.

[93] M. S. Miller, D. G. Bobrow, E. D. Tribble, and J. Levy, ''Logical secrets,'' *Proc. of the Fourth Int'l Conf. on Logic Programming (ICLP-4)*, MIT Press, May 1987, pp. 704-728.

[94] J. E. B. Moss, ''Getting the operating system out of the way,'' *Database Engineering* 9:3, IEEE Computer Society (September 1986), pp. 35-42.

[95] S. J. Mullender and A. S. Tanenbaum, ''The design of a capability-based distributed operating system,'' Report CS-R8418, Centre of Mathematics and Computer Science, Amsterdam, October 1984.

[96] R. M. Needham and R. D. H. Walker, ''The Cambridge CAP computer and its protection system,'' *Proc. of the Sixth Symposium on Operating Systems Principles*, November 1977, pp. 1-10.

[97] L. Neff, ''CLIPPER microprocessor architecture overview,'' *COMPCON '86*, March 1986, pp. 191-195.

[98] E. I. Organick, *The Multics System*, MIT Press, Cambridge, MA, 1972/ 1980.

[99] E. I. Organick, *A Programmer's View of the Intel 432 System*, McGraw-Hill, New York, 1983.

[100] J. K. Ousterhout, D. A. Scelza, and P. S. Sindhu, ''Medusa: An experiment in distributed operating system structure,'' *Commun. of the ACM* 23:2 (February 1980), pp. 92-105.

[101] D. L. Parnas, ''A technique for software module specification with examples,'' *Commun. of the ACM* 15:5 (May 1972), pp. 330-336.

[102] D. L. Parnas, ''On the criteria to be used in decomposing systems into modules,'' *Commun. of the ACM* 15:12 (December 1972), pp. 1053-1058.

[103] D. A. Patterson and C. H. Sequin, ''A VLSI RISC,'' *IEEE Computer* 15:9 (September 1982), pp. 8-21.

[104] J. L. Peterson and A. Silberschatz, *Operating Systems Concepts*, Addison-Wesley, Reading, MA, 1983.

[105] J. S. Quarterman, A. Silberschatz, and J. L. Peterson, ''4.2BSD and 4.3BSD as examples of the UNIX systems,'' *Computing Surveys* 17:4, ACM (December 1985), pp. 379-418.

[106] K. Ramamritham and J. A. Stankovic, ''Dynamic task scheduling in hard real-time distributed systems,'' *IEEE Software* 1:4 (July 1984), pp. 65-75.

[107] R. F. Rashid and G. G. Robertson, ''Accent: A communication oriented network operating system kernel,'' *Proc. of the Eighth Symposium on Operating Systems Principles*, November 1981, pp. 64-75.

[108] D. D. Redell, Y. K. Dalal, T. R. Horsley, H. C. Lauer, W. C. Lynch, P. R. McJones, H. G. Murray, and S. C. Purcell, ''Pilot: An operating system for a personal computer,'' *Commun. of the ACM* 23:2 (February 1980), pp. 81-92.

[109] D. M. Ritchie and K. Thompson, ''The Unix time-sharing system,'' *Bell System Technical Journal* 57:6 (July 1978), pp. 1905-1929.

[110] R. E. Schantz, R. H. Thomas, and G. Bono, ''The architecture of the Cronus distributed operating system,'' *Proc. of the 6th Int'l Conference on Distributing Computing Systems*, IEEE Computer Society, May 1986, pp. 250-259.

[111] K. Schwan, W. Bo, and P. Gopinath, ''A High-performance, object-oriented operating system for real-time, robotics applications,'' *Proc. of the 1986 Real-Time Systems Symposium*, IEEE, December 1986, pp. 147-156.

[112] M. L. Scott, ''Design and implementation of a distributed systems language,'' Ph.D. dissertation, Computer Sciences Technical Report 563, University of Wisconsin–Madison, May 1985.

[113] M. L. Scott, ''The interface between distributed operating system and high-level programming language,'' *Proc. of the 1986 Int'l Conf. on Parallel Processing*, IEEE Computer Society, August 1986, pp. 242-249.

[114] M. L. Scott, ''Language support for loosely coupled distributed programs,'' *IEEE Trans. on Software Eng.* SE-13:1 (January 1987), pp. 88-103.

[115] L. H. Seawright and R. A. MacKinnon, ''VM/370 - a study of multiplicity and usefulness,'' *IBM Systems Journal* 18:1 (January 1979), pp. 4-17.

[116] D. P. Siewiorek, C. G. Bell, and A. Newell, ''The System/360 and System/370 family,'' pp. 829-892 in *Computer Structures: Principles and Examples*, McGraw-Hill, NY, 1982.

[117] A. Silberschatz, ''ROSI: A user-friendly operating system interface based on the Relational Data Model,'' Computer Sciences Colloquium, University of Wisconsin–Madison Computer Sciences, November 24, 1986.

[118] A. J. Smith, ''Cache memories,'' *ACM Computing Surveys* 14:3 (September 1982), pp. 473-530.

[119] M. H. Solomon and R. A. Finkel, ''The Roscoe distributed operating system,'' *Proc. of the Seventh Symposium on Operating Systems Principles*, December 1979, pp. 108-114.

[120] E. H. Spafford and M. S. McKendry, ''The design of the CLOUDS distributed kernel,'' *Workshop on Design Principles for Experimental Distributed Systems*, IEEE Computer Society, October 1987. held at Purdue University, West Lafayette, IN 47907.

[121] A. Z. Spector, J. Butcher, D. S. Daniels, D. J. Duchamp, J. L. Eppinger, C. E. Fineman, A. Heddaya, and P. M. Schwarz, ''Support for distributed transactions in the TABS prototype,'' *IEEE Trans. on Software Eng.* SE-11:6 (June 1985), pp. 520-530.

[122] A. Z. Spector, J. J. Bloch, D. S. Daniels, R. P. Draves, D. Duchamp, J. L. Eppinger, S. G. Menees, and D. S. Thompson, ''The Camelot project,'' *Database Engineering* 9:3, IEEE Computer Society (September 1986), pp. 23-34.

[123] M. J. Spier and E. I. Organick, ''The Multics interprocess communication facility,'' *Proc. of the Second Symposium on Operating Systems Principles*, ACM, October 1969, pp. 83-91.

[124] M. Stonebraker, ''Retrospection on a database system,'' *ACM Trans. on Database Systems* 5:2 (June 1980), pp. 225-240.

[125] M. Stonebraker, ''Operating system support for database management,'' *Commun. of the ACM* 24:7 (July 1981), pp. 412-418.

[126] M. Stonebraker and A. Kumar, ''Operating system support for data management,'' *Database Engineering* 9:3, IEEE Computer Society (September 1986), pp. 43-50.

[127] M. Stonebraker, ''Object management in POSTGRES using procedures,'' *Proc. of the Int'l Workshop on Object-Oriented Database Systems*, September 1986, pp. 66-72.

[128] M. Stonebraker and L. Rowe, ''The design of POSTGRES,'' *Proc. of the 1986 SIGMOD Conf.*, Washington, DC, May 1986.

[129] D. C. Swinehart, P. T. Zellweger, R. Beach, and R. B. Hagmann, ''A structural view of the Cedar programming environment,'' *ACM Trans. on Programming Languages and Systems* 8:4 (October 1986), pp. 419-490.

[130] Y. Tamir and C. H. Sequin, ''Strategies for managing the register file in RISC,'' *IEEE Trans. on Computers* C-32:11 (November 1983), pp. 977-988.

[131] A. S. Tanenbaum and S. J. Mullender, ''An overview of the Amoeba distributed operating system,'' *Operating System Review* 13:3 (July 1981), pp. 51-64.

[132] A. S. Tanenbaum, ''Network protocols,'' *Computing Surveys* 13:4, ACM (December 1981), pp. 453-489.

[133] A. S. Tanenbaum, *Operating Systems: Design and Implementation*, Prentice-Hall, Englewood Cliffs, NJ, 1987.

[134] A. Tevanian, R. F. Rashid, M. W. Young, D. B. Golub, M. R. Thompson, W. Bolosky, and R. Sanzi, ''A Unix interface for shared memory and memory mapped files under Mach,'' *Proc. of the USENIX Summer 1987 Conf.*, USENIX Association, June 1987, pp. 53-67.

[135] A. Tevanian, R. F. Rashid, D. B. Golub, D. L. Black, E. Cooper, and M. W. Young, ''Mach threads and the Unix kernel: The battle for control,'' *Proc. of the USENIX Summer 1987 Conf.*, USENIX Association, June 1987, pp. 185-197.

[136] S. S. Thakkar and A. E. Knowles, ''A high-performance memory management scheme,'' *IEEE Computer* 19:5 (May 1986), pp. 8-22.

[137] K. Thompson, ''Unix implementation,'' *Bell System Technical Journal* 57:6 (July 1978), pp. 1931-1946.

[138] I. L. Traiger, ''Virtual memory management for database systems,'' *Operating Systems Review* 16:4 (October 1982). Also Research Report RJ-3489, IBM Research Laboratory, San Jose, CA (May 1982).

[139] R. P. Warnock III, ''User-mode development of hardware and kernel software,'' *Proc. of the USENIX Summer 1984 Conf.*, USENIX Association, June 1984, pp. 224-226.

[140] M. V. Wilkes and R. M. Needham, *The Cambridge CAP Computer and its Operating System*, North-Holland, New York, 1979.

[141] N. Wirth, ''Modula: A language for modular multiprogramming,'' *Software Practice and Experience* 7:1 (1977), pp. 3-35.

[142] N. Wirth, *Programming in Modula-2*, Springer-Verlag, NY, 1984.

[143] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack, ''HYDRA: The kernel of a multiprocessor operating system,'' *Commun. of the ACM* 17:6 (June 1974), pp. 337-345.

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

[145] W. Zhao and K. Ramamritham, ''Distributed scheduling using bidding and focused addressing,'' *Proc. of the 1985 Real-Time Symposium*, IEEE, December 1985, pp. 103-111.

[146] H. Zimmermann, ''OSI reference model—the ISO model of architecture for open systems interconnection,'' *IEEE Trans. on Commun.* COM-28:4 (April 1980), pp. 425-432.