AN APPROACH TO THE DESIGN OF FULLY OPEN COMPUTING SYSTEMS

by

Yeshayahu Artsy
Miron Livny

Computer Sciences Technical Report #689
March 1987

CONTENTS

An Approach to the design of Fully Open Computing Systems

Yeshayahu Artsy
Miron Livny

University of Wisconsin – Madison
Computer Sciences Department

March 1987

Abstract

Every designer of a general-purpose operating system (OS) faces the problem that the service needs of potential applications can neither be perceived nor completely satisfied. New needs emerge and services required by one application might conflict semantically or incur heavy overhead on other applications. Therefore, an ideal system would be one that lets *any* application choose, replace, add, and extend OS services and resources. Such a *fully open* system lifts the burden of selecting the optimal set of services. However, full openness raises difficult questions regarding the structure of the system and the mechanisms required to support openness. Our research focuses on how to construct a fully open computing system in a multiuser environment. We investigate the interplay between openness and the constraints imposed by protection requirements and efficiency objectives. We examine the impact of different prospective designs on the architectural complexity and programming complexity.

In this paper we present a novel model of a multiprocessor fully open computing system. We sketch a specific design based on the model and touch upon its implementation. These three levels of abstraction are used to shed light on the problems regarding full openness and to evaluate prospective solutions. The model is based on the concepts of *resource ownership* and *service provision*. The OS is viewed by the model as a minimal set of mandatory services, necessary to protect users and system resources. It accommodates construction of customized, shared "OS"s in a protected and efficient way. Ordinary, untrustworthy applications can provide most of the traditional OS services. They can own shared physical resources and access them via services at the lowest level. We focus on CPU and memory management and illustrate how openness can be attained within protection and efficiency constraints. At the design and implementation levels we point at techniques and features aimed to reduce overhead and complexity. Finally, we evaluate our approach and discuss its practicality.

1. Introduction

Suppose you have just completed the design of your next-generation, general-purpose operating system (OS) after thoroughly examining the needs of its prospective applications and carefully crafting the services to support them. The question now is how soon designers of databases, programming languages, or other special applications will complain about missing features or inadequate facilities in your OS. This phenomenon was observed in many cases [1,2,3]. It is a symptom of two problems designers of general-purpose OS's face: First, they cannot perceive the service needs of potential applications, since needs change as new applications emerge. Second, they cannot satisfy them all, since needs may conflict, or satisfying one may impose unbearable overheads on some applications. As it turns out, even OS's considered as providing 'generally adequate' services might fail to appropriately support certain applications, in particular because their general services impose excessive overheads or coercive semantics on these applications.

Consider the following examples. General memory and buffer management services could become very inefficient for a DBMS [1], or redundant for a DBMS that uses different buffering schemes [4]. A flexible and elaborate interprocess communication (IPC) mechanism could be semantically coercive for a language specifically designed to support this mechanism [5], or inefficient and insufficient for a tightly-coupled, multiprocess transaction system [6]. Moreover, even extending the OS with services favorable to such an application [7] does not necessarily remedy the inefficiency problem due to the generality of the services [8]. A real-time application has scheduling requirements different from other applications, and it may necessitate efficient accommodation of several communication models at the same time [9]. While most applications would prefer a virtual, higher-level view of the hardware, some applications require direct access to physical resources for efficiency or correctness reasons. Testing a new device driver on top of relatively high-level OS services, for instance, could become a complex and costly task, without the guarantee of correctness [10]. Finally, an inflexible filing facility can restrain the applicability of a user-friendly interface to the OS itself [11].

The alternative to one general-purpose OS is a number of application-tailored OS's. However, it is usually impractical to write a full-scale OS for each application, nor to have several OS's coexist in a computer shared by different applications. The main concern of this paper is what is required to make a general-purpose OS behave as many application-tailored OS's. (We henceforth discuss only a general-purpose OS, referring to it simply as OS.) An ideal OS is one that dynamically adapts its services to the needs of applications, without incurring extra overheads or impeding their protection needs. We believe that only an *open system* that lets applications choose, add, replace, and extend OS services and resources can satisfy the quest of adaptability; it will be ideal if openness is accommodated in a protected and efficient way. An open system thus offers a solution to the design problems mentioned above: it lifts the burden of evaluating the needs of the potential applications, and the selection of an optimal set of services.

The open system approach contrasts the traditional *closed system* approaches. One such approach is to design the OS as a closed, high-level interface to physical resources that provides ample or functionally rich services. Although in general this approach offers convenience to applications (as illustrated in Figure 1a, applications are 'small'), it has several drawbacks. It imposes heavy burden on the OS designer/implementor to extend and modify services as new needs emerge. Some modifications might be impossible because of conflict with existing services. Moreover, in an all-if-anything service provision style each application pays a performance penalty for features it does not want. As we learned from designing a flexible IPC mechanism [12, 13, 14], adding features to improve generality might increase the complexity of both the OS and applications, impose high execution cost, and still be insufficient. Another approach is to design the OS as a closed, low-level interface that provides a reduced set of services, as shown in Figure 1b. Although in this approach applications presumably can satisfy their needs, including efficient access to physical resources, writing applications is more complex. Each application suffers programming overhead in having to supplement the basic services. For many users the complexity of managing physical resources and the inconvenience of the interface are unacceptable burdens. An open OS can combine the benefits of the two approaches, as alluded to in Figure 1c. Applications can access the OS at different levels or define new levels. The system offers the convenience of the former approach, and the

efficiency and customization of the latter approach. However, in order to be fully adaptable (that is, reconstructible and extensible), a system should be *fully open* to all applications. By fully open we mean that even in a multiuser environment, services and resources are open to *all* applications, including services in the IPC, memory and processor management domains.

The fully open system approach presents several intriguing and intricate questions.

(a) **How is openness achieved?** Specifically, how is the system structured, what are the constructs that represent services and resources, and what are the primitives that accommodate dynamic customization? Some of the aspects of this question are: to what extent services can be open independently of each other, shared by different applications, and what architectural support might be required?

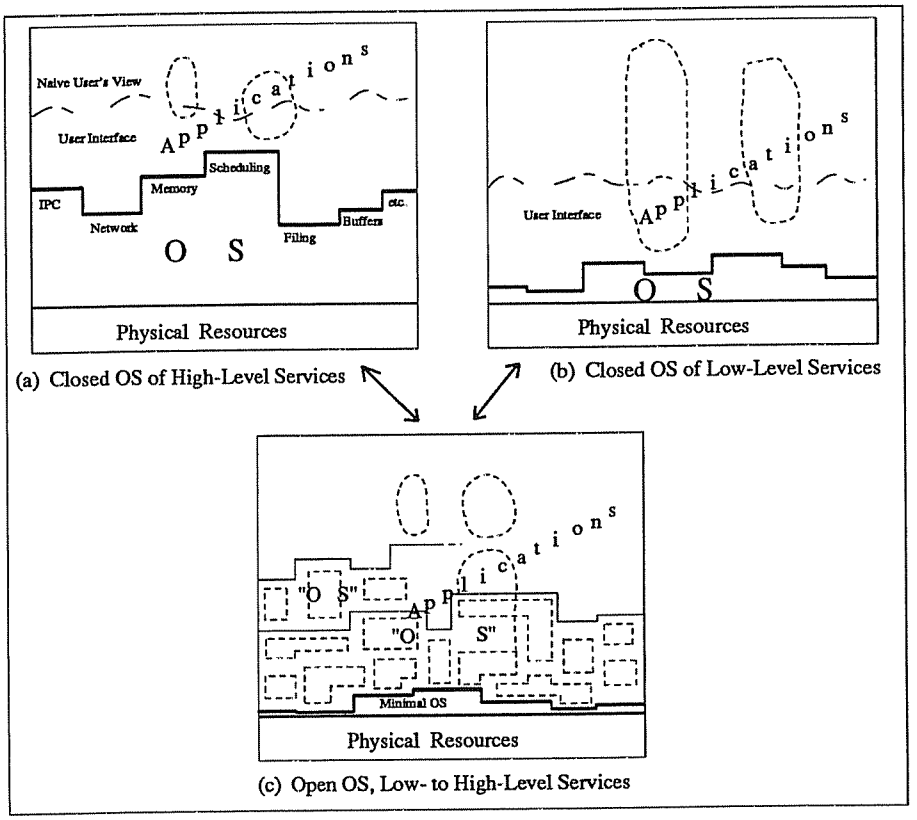

Figure 1: The Operating System in Various Approaches

(b) **How does openness interact with protection considerations?** In a multiuser environment, in order to protect resources from being inadequately used and to protect users from each other, some services *must* be retained in the OS and some restrictions *must* be imposed on users. It is unknown, however, what is the minimal set of such services and restrictions. This set should facilitate additional protection measures implemented by individual applications.

(c) **Does protected openness impede efficiency or incur complexity?** If so, what are the tradeoffs between them? Can efficiency be improved through architectural support?

In summary, the problem is how and to what extent a system can be open to its customers, and what is the interplay between openness and various constraining, mutually-dependent aspects of protection, efficiency, architectural support, and complexity.

In order to explore these questions and to evaluate prospective solutions, we decided to experiment with a fully open computing system (**FOCS**) defined in three levels of abstraction. First, we have developed the fully open computing system model (called the **FOCS model**), which defines the system components, their interrelationship, and the constraints imposed by protection requirements. Second, guided by the model, a design of a specific multiprocessor system has been laid out. Third, we have examined techniques to implement various components and features of the design. Each of these refinement stages has contributed to our understanding of the problems regarding openness and has provided a different framework to evaluate solutions. We found the model instrumental in exposing many obstacles in achieving full openness, and in shedding light on dependencies between the components of a FOCS. The design and the implementation techniques provide insights into the practicality of openness, and into the tradeoffs between openness, efficiency, and complexity.

A new model was necessary since no other models or OS designs support openness to its ultimate extent. Several models, such as the object [15], message–based client–server, and virtual machine models [16], and systems motivated by them [17, 18, 19, 20, 21, 22, 23] open high-level services or virtual resources to applications. However, none of them enables general users to manage shared physical resources or

provide low-level services. Most services for IPC, memory and processor management in these models and systems are provided by irreplaceable OS components. 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 notions we wanted to introduce, such as letting any application directly access physical resources and provide allocation policies, conflict with the model philosophy. Some systems isolate their entities from each other and impose protection barriers between them, which in turn imply expensive communication costs. As a result, opening low-level or frequent services is restricted or impractical. We therefore preferred to develop a new model, guided by the goal to pursue openness to its ultimate extent. In a way, our model extends all these models with features that support full, protected openness, and simplifies them by removing features obstructing openness.

This paper is an interim report of our research in system openness and focuses on the model. The model is detailed in the next section. Several components of the specific design are briefly discussed in Section 3, and implementation issues are touched upon in Section 4. These sections are not intended to illustrate a complete design or implementation, but rather to emphasize techniques and features aimed at reducing overhead and complexity without undermining openness or protection. In Section 5 various aspects of our approach are discussed, and a comparison to related work is drawn in Section 6. We conclude with a summary of the key ideas and indication of future directions.

2. The Model

The FOCS model assumes a multiuser, multiprocessor shared memory environment. The system consists of one or more memory devices, each defining a separate physical address space. All memory devices comply to the same service interface. There is one or more tightly-coupled and architecturally identical CPU's. Only a few assumptions are made by the model for necessary architectural features. These assumptions derive from protection considerations and the need to support openness. We present these assumptions where the need for them is described. We first present a general overview of the model and then elaborate on its application in various domains. We focus on the processor and memory

management domains, in which openness sharply confronts protection requirements and efficiency objectives.

2.1. Overview

Central to the model is the observation that full openness can be achieved by letting each computation select the services and resources it needs. It is assumed that all computing requirements can be captured in two abstractions, a service and a resource. A **service** is a logical function and a **resource** — the means needed to perform the service. 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 each other. Each server can dynamically choose, extend, replace, or add resources and services. Services are executed by **activities**, which are threads of control that span multiple servers. The activity represents a computation; it starts at one service, and can transfer control to other services to be performed on its behalf. Ownership of resources and access to resources is accomplished via services. Motivated to support full openness, the model leaves the semantics and function of resources and services to the mutual understanding of the provider and users of each resource/service. Only a few default rules apply to the mechanisms of service provision, as required by protection requirements.

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 structures) that implements services. A **service** is an abstraction of a set of actions, whose functionality and invocation conventions, such as the number and types of input arguments and of returned results, are decided by the service provider. Services are invoked via **bindings**, each of which is a reference to a particular service. A service can be performed asynchronously with its invocation or return results multiple times. 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. It is encapsulated in one server, called its **host**. A resource is composed of units. Each unit can be owned by several servers concurrently. The default owner of the entire resource is its host; other servers become owners via allocation. The host defines the

semantics of accessing the resource and the rights the owners have to the resource. Through the host's services an owner can access the resource and allocate units it owns to other servers. Depending on the resource, these functions might be allowed only to *current* owners, which are the last allocatees for each resource unit. A *former* owner, which is a previous allocator of a unit, may revoke it; revocation from an owner implies revocation of all allocations made by the revokee. An owner can also issue permits for other servers to access its share of the resource. A **permit** is a reference to a portion of a resource that specifies access rights, such as modify or copy.

Figure 2-1 illustrate the relationship between servers, resources, and services. Multiple hierarchies of services and resources can be formed. At the bottom of the resource hierarchy we show the physical resources, whose hosts are called **devices**. A resource may be mapped by its host to several physical or logical resources which the host owns or is permitted to access. The dependency of a resource or a service on other resources or services can be transparent to the users of the resource/service. Figure 2-2 further illustrates the notion of resource ownership and access. Notice that server S_y, for instance, accesses resource R through the host of R, and not through S_x who has allocated the resource to S_y.

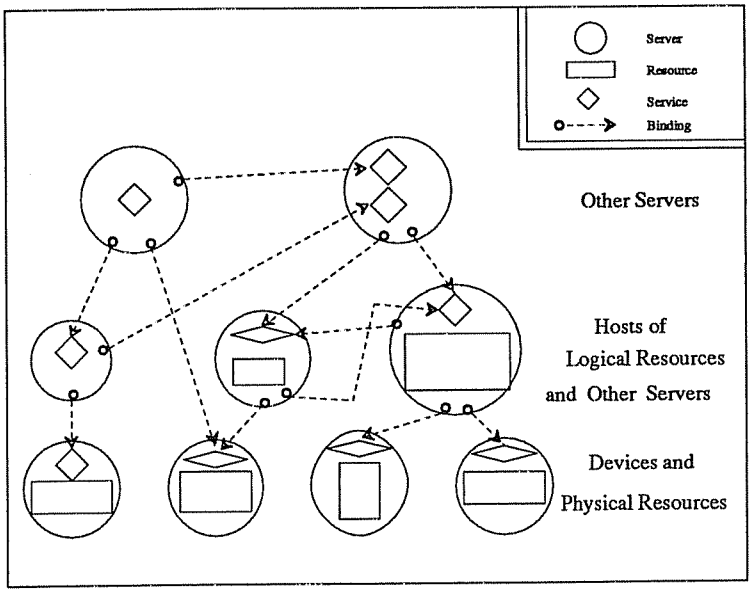

Figure 2-1: Servers, Resources, and Services

The model view of naming and typing is very simplistic, as stems from our desire to minimize the imposition of default semantics and overheads. Servers, services and resources are identified by unique names. Service and resource types have application-specific semantics, and so there is no support in the model for formal definition and checking of types. Names are used by hosts and service providers to announce their resources and services. These announcements can be made statically, e.g. in user manuals, or dynamically through intermediary servers. We denote such servers as *name servers*. A name server would allow servers to deposit bindings/permits, to locate required resources/services and to obtain the necessary bindings. In general, servers acquire bindings/permits statically, e.g., at load time, or dynamically, e.g., as a result returned by another service.[1]

Services are executed using processors. Each device owns a private processor, dedicated to perform access to its resource. The CPU device is peculiar since its resource, the CPU, is a processor too. A CPU is needed by *every* server to realize services; devices, for instance, need a CPU to initiate access services or to inform customers of access completion. Therefore, every server may be a CPU owner or be permitted access to a CPU by an owner. We emphasize in the model the ability of former CPU owners to reclaim the CPU, since there are times when this reclamation is urgently needed by devices. The protocol of CPU reclamation, called a *CPU interrupt*, and the dynamic ordering of CPU owners are detailed in § 2.4. CPU's are allocated by time. An **activity** is the entity that consumes CPU time to execute services. Through service invocation and return the activity transfers access permission for the CPU to the servers whose services it executes.

Protection in a multiuser environment prescribes restrictions on openness. On the one hand, since applications vary in their protection requirements, they should be let decide and implement their protection mechanisms. Accordingly, it is left to each server to decide discrimination among customers, rejection of

---

[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 termed identically, such as two disk access services, provide identical function.

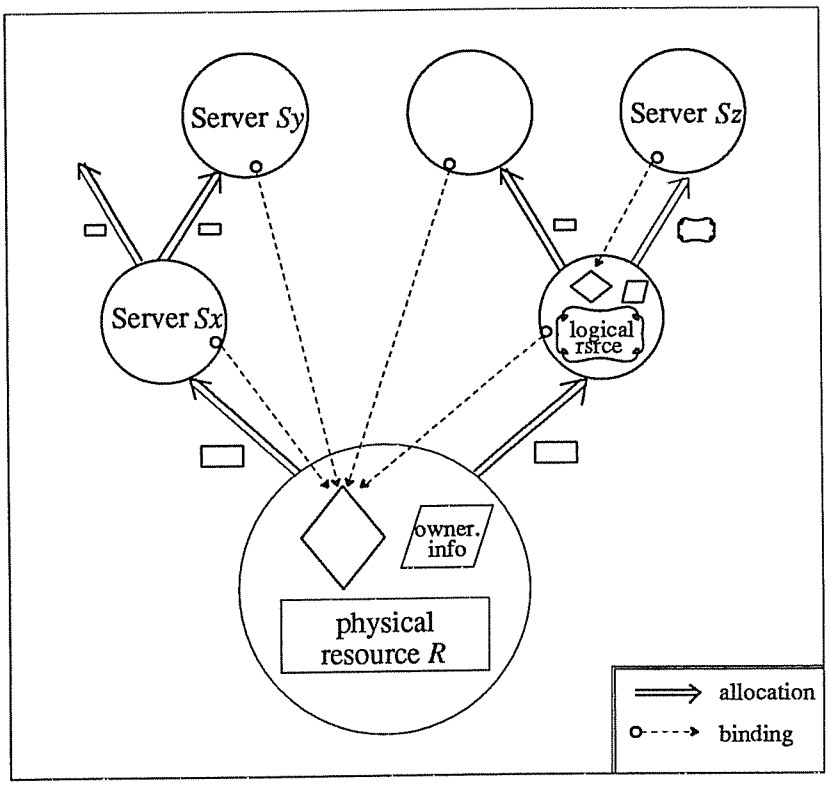

Figure 2-2: Resource Ownership

Allocation of a resource is illustrated as transferring units of it. A logical resource is depicted differently than a physical resource, although the user of the logical resource (Server S_x) should not be aware of the difference.

invocations, checking service inputs, and if it is a host — to verify ownership and access permission to its resource. It is assumed that a service is entrusted by its customers to provide the expected function. Hence, a customer is not protected against a faulty service, other than disallowing the service provider to use resources it has not been granted access to. Also, bindings and permits are supported as light-weight capabilities: A binding/permit is created and can be invalidated only by the service provider/resource owner to whose service/resource the binding/permit refers; each holder of the binding/permit can duplicate and transfer it to other servers as an ordinary data structure.

On the other hand, there are three basic protection requirements that must be guaranteed. First, a faulty server should not fail unrelated servers. This requirement is supported by the restriction that servers cannot access resources they are not granted access to and the ability to revoke a resource, e.g., preempt a CPU from a server after its allocation expires. Second, servers should be able to implement their

protection mechanisms. This requirement can be achieved through a set of mandatory services provided by generally-trusted servers. The model view is that this set of services is minimal, in that it excludes any service or restriction that servers can provide for themselves. This set consists of verification of bindings at service invocation and of return address at service return, verification of another server's id, and maintaining accounts. The latter two services are necessary so servers can discriminate among customers and reduce contention for scarce resources. The third requirement is that certain resources, called **system resources**, be protected by generally-trusted hosts. The definition of what constitute the system resources is installation-dependent; it presumably would consist of only mandatory or globally-shared resources. The model assumes that the mandatory services provided by these hosts is minimal, in that they include only services necessary to control allocation and access to the system resources; all other services to use the resources are relegated to the resource owners. The last two basic protection requirements are guaranteed by a set of generally-trusted servers called the operating system base (**OSB**).[2]

The OSB constitutes the 'real' OS in the model. Its power derives from the fact that at system initialization it annexes the system resources and all resources necessary to impose the mandatory services. What servers we consider as appropriate to be included in the OSB? It should include the hosts of the system resources, and of resources the OSB depends on, such as the memory into which the OSB is loaded. We assume that this installation-defined list includes all the CPU devices[3] and a clock device, which is essential for proper reclamation of CPU's. In addition, the OSB includes: (a) the *accountant* — a unique server that provides services to open/close accounts and maintain their balances, and (b) the *initiator* — a unique server that at system-initialization time creates the rest of the OSB and other servers required in the system. The CPU device provides, among other services, binding and return address verification. For the sake of allocation simplicity, it is assumed that all CPU's are owned by the **CPU driver**, a server which

---

[2]Additional restrictions are required to standardize the representations of bindings, permits, and other forms of naming and addressing. The specification of these standards is relegated to the designer of a specific system.

[3]The reason for this requirement is that *any* untrustworthy CPU device may maliciously modify the OSB and thus endanger the entire system.

facilitates the allocation of CPU's.[4] The CPU driver and any driver of a system resource are included in the OSB.

The non-OSB servers are called **application-level** servers, and a logical collection of them is called an **application**. Above the OSB, applications can construct virtual OS's. A **virtual OS** is merely a collection of servers that includes the OSB. One such virtual OS is assumed to be created at system-initialization time, to provide common services required by different applications. For instance, this virtual OS may include the following servers: (a) a default *name server*, (b) a *server creator*, which creates other servers, and (c) an installation-defined list of various hosts, drivers of shared resources, language processors, and editors.

2.2. Execution Management

We turn now to describe the details of service provision and resource management in specific domains. In this section we elaborate on processor management[5] and in the next section on memory management; other physical resources are discussed in § 2.4. We show that using the principles introduced above, every application can achieve its peculiar CPU scheduling or memory management policies. We describe enhancements of the model with new features, some which derive from efficiency and simplicity considerations.

The CPU driver is allocated all the CPU's for eternity. It routinely suballocates them to its customers, and provides services to further allocate/revoke CPU's. The unit of allocation is a *time unit*. Any server may own a CPU for some *time slice*. Such a server is called a **scheduler**. To become a scheduler, a server must establish an interface with the CPU driver, which is necessary so that the former is informed when it is allocated a CPU, and that its allocation requests can be verified. The scheduler therefore hands a binding for its scheduling service, the latter to be invoked whenever control of a CPU is transferred to the

---

[4]In general, a **driver** denotes a server that controls resource allocation for one or several devices. We introduce this term only for reasons of efficiency and convenience of allocation.

[5]We focus on the management of CPU's only, since the private processors of other devices are stati-

scheduler.

Let us briefly describe the application of the model principles of ownership and allocation to CPU management. A scheduler can allocate a portion of its time slice of a given CPU to another scheduler, which then becomes the current scheduler for that CPU. Control of the CPU returns to the allocator for the remainder of its slice when the current scheduler's slice expires. As with any resource, a scheduler can release the CPU before its slice expires; a former scheduler can reclaim the CPU, for instance when it is reactivated via a service invocation. Reclamation or release imply recursive revocation of all further allocations made by a later owner. The ordering of CPU ownership is maintained by the CPU driver. The latter can alter the order when schedulers require urgent execution. This feature allows devices and other servers to acquire a CPU urgently, as well as to prevent interruption during urgent work (see § 2.4). Once allocated a CPU, the scheduler uses the device's services directly to schedule usage of the CPU.

The CPU is a resource necessary for execution of services. The machine instructions, in fact, are services provided by the CPU device to use the CPU. An **activity** is an independently schedulable entity. A scheduler *dispatches* an activity for execution on a particular CPU for a *time quantum.* Dispatching is a service provided by the CPU device, and it implies granting an access permit to the server whose service the activity is currently executing. Why couldn't a scheduler grant the access permit directly to the server that needs the CPU? The reason is that the scheduler has some knowledge about the precedence of the computations it schedules. This precedence is associated with activities created to execute these services, and must somehow be conveyed to services the former services invoke. Therefore, activities are viewed as threads of control across server boundaries (see Figure 2-3). Upon a service invocation the activity continues to execute the invoked service at its server, suspending the invoking service until the invoked service returns. Thus, service invocation and return, which are services provided by the CPU device, to transfer CPU access permit among servers. Multiple activities can run simultaneously at one server. They all share the server's address space, resources, permits, and bindings. It is the server's responsibility to synchronize

---

cally allocated to them.

or mutually exclude them. Each activity has a private execution context (detailed shortly). Notice, however, that a service can be provided asynchronously with its invocation, for instance when the service returns after invocation but its function is performed by another activity.

These features allow implementing any application-specific CPU scheduling policy. All an application needs is that one of its servers becomes a scheduler, acquires CPU's from the CPU driver or other schedulers, create activities and dispatch them. Or, an application might choose an existing scheduler to be bound to. However, there are several problems regarding execution management, which require additional features, as we show in turn.

First, a server might be unable to continue a service, for instance because a specific event should be awaited, such as the completion of an IO service or the release of a buffer used by another activity. Therefore, the server needs to notify the activity's scheduler to block the activity, and likewise later to unblock it. Notice that the server, being executed by activities started by different applications, might not necessarily know the invocation conventions of their schedulers' services. Hence, a *standard interface* must be defined for such services, and in general for all services required for a particular activity by different servers. These services are distinguished from services required by a server regardless of the activities executing it, such as OpenFile for temporary results. The standard interface defines what these services are, where their bindings should be found, and what their invocation conventions are. The model does not define a particular standard interface; this task is relegated to the designer of a specific system. The bindings of these services are considered part of an activity's context; they are denoted as the activity's **private** bindings. The server may also *pause*, which implies the expiration of the current quantum.

A second problem arises with expiration of time quantums. A server does not control the time allotted to activities executing its services, other than through suggesting scheduling preferences to their schedulers. Time expiration, however, is undesirable to a server particularly in two situations: (a) within a *critical section*, which is necessary to mutually exclude accesses to shared structures; this expiration may cause a delay to other activities arriving at the critical section and form *convoys*, and (b) while executing an

urgent function such as handling a trap or an interrupt, both which are discussed in § 2.3 and § 2.4. Being generally untrustworthy, ordinary servers cannot disable time expiration. Two features are added to the model in order to remedy these problems: First, a server may request from the CPU device to postpone time expiration during the critical section/urgent function. The CPU device sets a limit to the extra time granted to a server in this way. Second, should an activity get 'stuck' in a critical section/urgent function, or at occupying a structure required by another activity, the server can *push* that activity to release the critical section/structure or complete the urgent function. A server executed by activity A may transfer execution to activity B currently idle at the server, for the rest of A's quantum. This transfer tantamounts to using the same access permit in two contexts. The operation is performed via a service provided by the CPU device, whose intermediation is necessary in order to preserve A's execution state and restore B's (detailed shortly). Notice that the focal server cannot transfer execution to another activity currently executing or idling at another server, since such a transfer might confuse the state of the latter server, for instance if B is awaiting a particular event. For the same reason, a scheduler is allowed to dispatch only the activities it controls. This control is represented by having an access permit to an activity's context descriptor.

A third problem occurs at activity switch. The execution state of the suspended activity should be preserved until this activity resumes execution. This state consists of information held at the CPU device, e.g., in the CPU registers, and includes the environment of the currently executed service.[6] There are three requirements imposed on this operation: (a) the state needs to be copied into a structure which is of a predefined, CPU-dependent format, so that the copy can be automatically performed by the CPU device, (b) the structure must be core-resident when the copy is required, at a location known to the CPU device, and (c) the preserved values should be protected from corruption by other servers. To satisfy these requirements, a standard structure called an Activity's Context Descriptor (**ACD**) is maintained for each activity at the CPU driver, as illustrated in Figure 2-3. The latter guarantees requirements (b) and (c). For the same

[6]The environments of services in the activity's thread are stored either in these registers or in local variables in the servers at which the activity runs. It is left to a server to preserve the environment of its service prior to invoking another service, using for instance local *stack frames*, and restore the environment

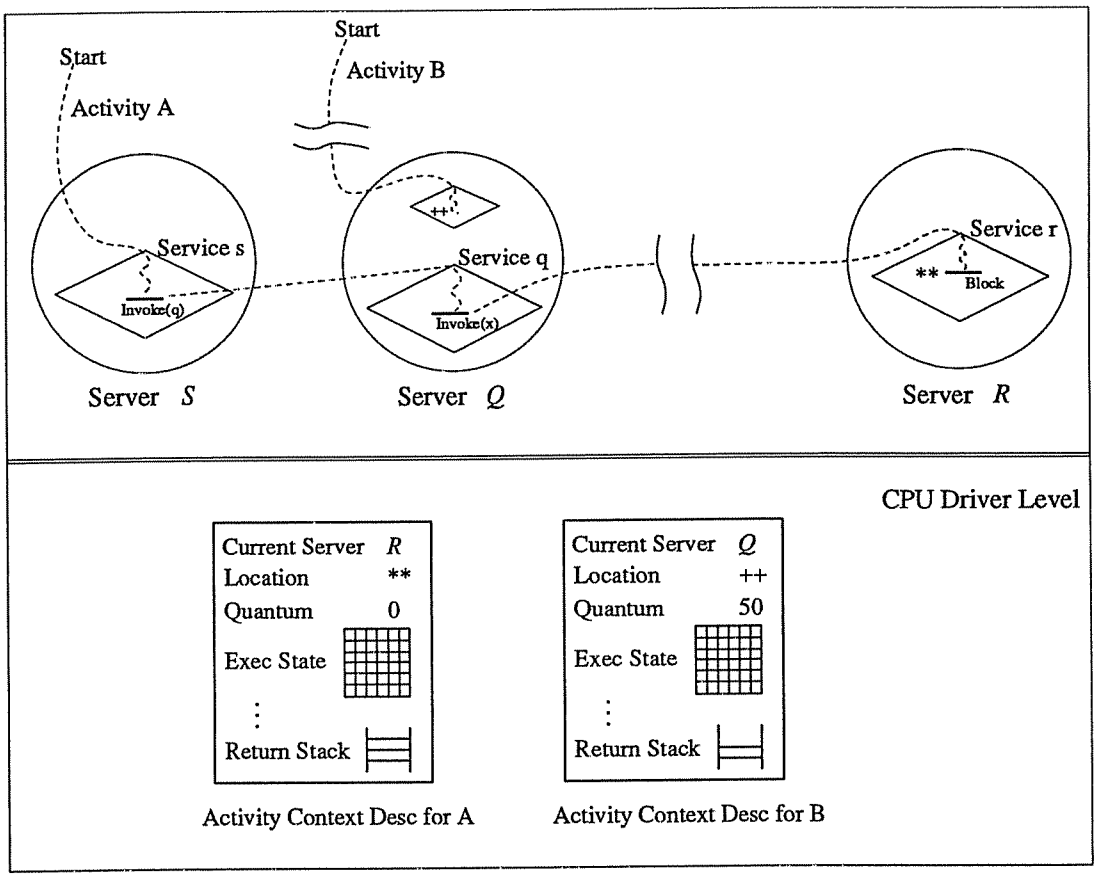

Figure 2-3: Activities and Context Descriptors

Activity *A* has started in Server *S* and is currently blocked in Server *R*. Activity *B* is currently running in Server *Q*.

line of reasoning, the ACD contains other volatile values such as the activity's account id, quantum, and a stack of return addresses for services invoked but not returned yet. Notice that ACD's are not stored at the CPU device to avoid static association of an activity with a given CPU. The CPU device is permitted access to all ACD's. The implication of having ACD's maintained by the CPU driver is that activity creation is through the latter's services. The creator of an activity receives an access permit to the activity's ACD with certain rights, including the rights to dispatch and set the activity's initial service invocation. The control over an activity can be passed among schedulers by passing this permit. Moreover, the holder of such a permit can also direct the CPU driver's policy of retaining ACD's in memory, in cases when not all ACD's can remain core-resident all the time. This feature allows more efficient execution of urgent

---

thereafter.

activities.

Finally, how does a scheduler know when it is allocated a CPU? When the driver notifies the CPU device of the new owner of the CPU, the CPU device switches to its designated activity to invoke the scheduler's scheduling service. This activity is reactivated upon each slice or quantum expiration. Since a scheduler may own several CPU's, this service might be invoked concurrently by different activities. This situation is not different from any other server being executed by several activities. Therefore, it is left to the scheduler to make the service indivisible and to mutually exclude the activities where necessary. It is also up to the scheduler to restore the consistency of its structures, should one of the activities get preempted prior to completing this service, for instance when a CPU is revoked from this scheduler.

2.3. Memory Management

In this section we reiterate the model general principles as applied to management of physical and virtual memory. We present additional features that allow an application to directly and efficiently map its virtual spaces to physical memory, while memory accesses are performed efficiently and in a protected way. The particular features are motivated also by considerations to accommodate sharing of virtual and physical spaces, and to keep the interfaces simple.

The model views virtual memory as a collection of Universes. A **Universe** is a self-contained, autonomous resource composed of Spaces. A **Space** is the unit of allocation — a reference environment into which a server is loaded. It is composed of **segments**, each of which is a single virtual address space composed of pages, as illustrated in Figure 2-4b. A **page** is a contiguous range of virtual addresses. The host of a Universe is called its **manager**. A Universe is mapped into the physical memory owned by its manager. Although the Universe is embedded within its manager, as shown in Figure 2-4a, the manager is virtually embedded in the Universe, as it runs in one of the Universe Spaces, as shown in Figure 2-4b. This feature allows a Universe manager to directly and efficiently manipulate the structure and mapping of the Universe (detailed shortly). A server has an access permit to the entire Space it is loaded into. Thus, it may create and transfer access permits to regions of its Space, which can be used, for instance, as buffers

by other servers. Similarly, a Universe manager may create access permits to the physical space it owns.

The physical memory consists of one or more physical address spaces, each of which is defined by a memory device and is composed of frames. A **frame** is a contiguous, device-dependent range of physical addresses. The frame is the unit of allocation. For the sake of clarity, we will distinguish between memory devices and drivers. A **memory driver** is a server that provides services to create/destroy Universes, to allocate/revoke physical memory, and to copy from one Universe to another one. The memory device is assumed to perform only accesses. We assume that the entire physical address space of a memory device is owned by one memory driver. We detail below the interaction between Universe managers, memory devices and drivers.

As shown in Figure 2-4, a Universe is mapped into one physical memory space. The restriction of one space is necessary since different memory devices may employ different mapping schemes, as discussed shortly. At each memory access, a virtual address must be translated to a physical address. It is impractical and insecure that address translation be performed by Universe managers, and thus it must be performed by the device. Therefore, the model requires some low-level uniformity of mapping to allow efficient and protected access. The device needs to know the Universe's structure and mapping to physical memory. This information is contained in a Universe segment, called the **base segment.** The structure of the base segment is predefined and device-dependent. Its mapping to physical memory is known to the device at every instant. This mapping is provided by the manager at Universe-creation time, and can be dynamically modified thereafter. Figure 2-4 shows that the device holds ownership information, which can be set by the driver, and translation information, which is used for memory accesses.

Mapping to physical memory is as follows. The frame is also the unit of mapping. The mapping information visible to the device must follow the device-dependent frame size(s), though each Universe may support different page sizes. The structure of the mapping information depends on the mapping scheme employed by the device. At the model level we consider two basic schemes, since other schemes can be a combination thereof. In a direct-mapping, index-based scheme, a per-segment page table specifies

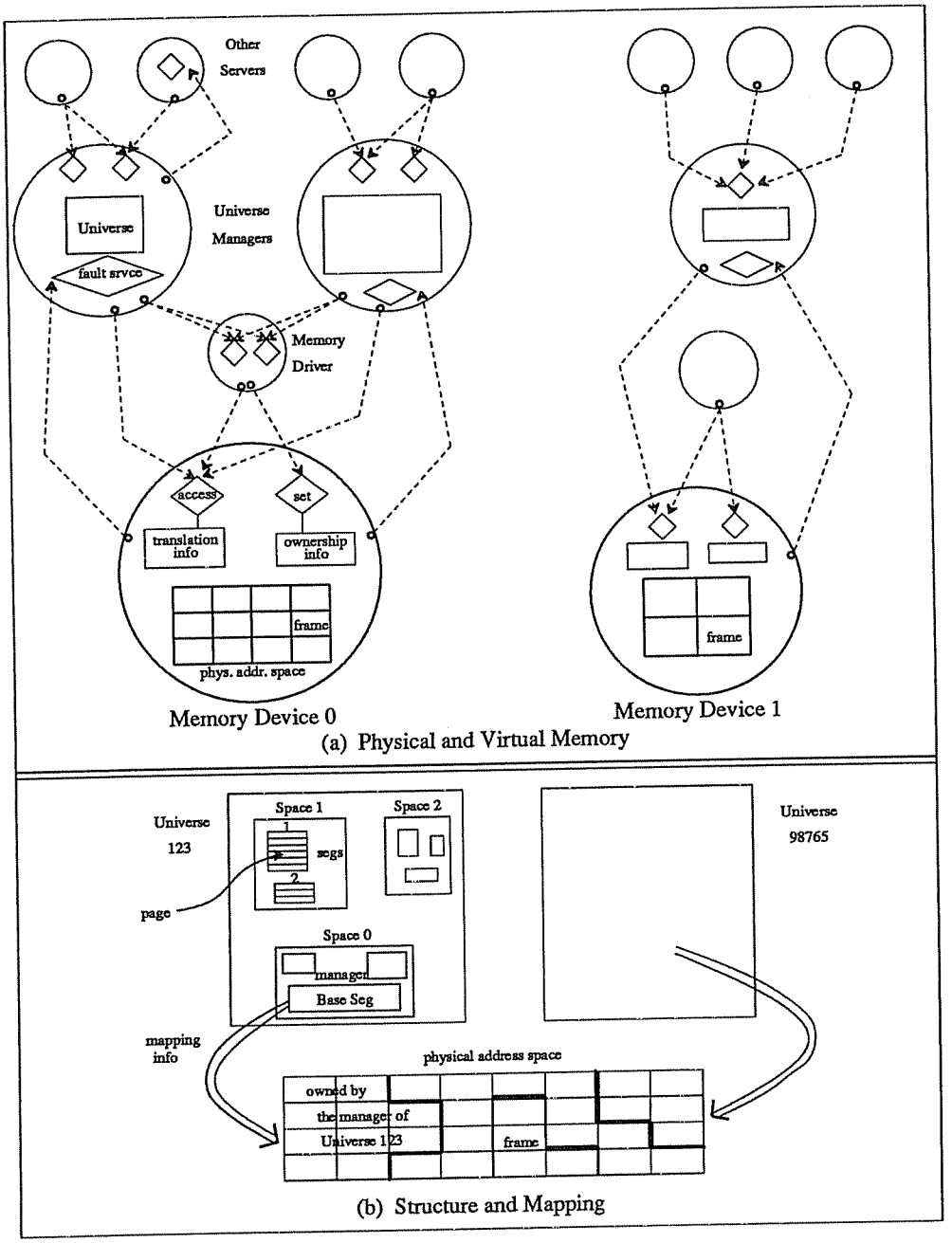

Figure 2-4: The Memory Subsystem

for each page the frame it is mapped to. Page tables reside in the base segment. They are pointed to from segment descriptors, which are placed in the base segment too. The base segment's page table *must* reside in physical memory owned by its manager. (Assuming a contiguous table, a manager therefore must own at least one *contiguous* physical memory area of the table size.) Figure 3-2 exemplifies these issues for a

specific design. In an inverted-mapping scheme, where a single frame table is maintained by the device, each frame's descriptor specifies the page mapped to it and a group id. Each segment's descriptor specifies the segment's group association.[7] The device allows a Universe manager to manipulate the descriptors of the frames it owns. The reason for having group ids or references to page tables in segment descriptors is to allow sharing of segments between Spaces. However, to avoid the problem of unique ids for shared segments as in Multics [24], it is the mapping information, not the segment itself, that is shared. Figure 3-2 illustrates the structure of the base segment and the information held at the memory device, assuming a direct-mapping scheme. Notice that the mapping information of the Universe manager and of the base segment are also held in the base segment. As a result, the Universe manager can manipulate page tables, change structural information of its Universe, and perform paging, all by accessing its local structures.

Knowing the structure of the base segment, the device can examine the structure of the Universe, locate the mapping information, and perform the necessary translation. It is assumed that the CPU generates virtual addresses within a Space, in the form { <segment number>, <page number, offset> }.[8] Therefore, a per-CPU *current* Space, that is, a <Universe id, Space number> pair, must be known at every instant. (For instance, this information can be maintained at the memory device, as shown in Figure 3-2.) Spaces are switched only at activity switch (e.g. at dispatching), service invocation, and service return. Copy between two Spaces does not cause a Space switch: the copy is performed by the Universe manager in a single Space, if the two former Spaces are in the same Universe, or else by the device. The translation process is the following. Given the current Space, its segment descriptors are located, based on the base segment's structure and mapping information. Likewise, the segments' mapping information are located, including their page tables in a direct-mapping scheme. Given a virtual address, the referred segment's mapping information is examined; searching for, or indexing by the page number, a frame number is

---

[7]The base segment's group association, though, must be explicitly told to the device by the Universe manager, so that the device can locate the necessary information held in the base segment.

[8]To be more accurate, a virtual address is a permit to access a location in a Space, furnished by the running server to the CPU device. The CPU device passes the permit to the memory device as a parameter to a memory access service.

derived, which added the offset yields the desired physical address. See an example of the translation process in § 3.3.

For protection purposes, the device verifies the following: First, that every frame accessed during the address translation process and that the yielded frame are owned by the manager of the addressed Universe. Second, that the virtual address is valid, e.g., that the segment number and the segment descriptor are valid. Last, that the intended access specified in the machine instruction or the access permit does not conflict with the access rights associated with each segment. Upon a page fault or a violation, the device, depending on the cause, either rejects the access service with an appropriate error indication, or invokes a designated service of the Universe manager, labeled as "fault service" in Figures 2-4. A binding to this service is handed by the manager at Universe creation. This service may invoke a fault-handling service of the server that caused the fault or another designated server such as a debugger. For this purpose, a server can register a fault-handling service with its Universe manager.

2.4. More on Devices

Communication with devices follows the regular service invocation paradigm, but is more intricate in that it requires transfer of control between two processors. (For clarity, we refer to devices other than the CPU devices as *IO devices*.) Invocation of an IO device's service is performed on the CPU where the invocation is issued. We assume that the IO device has an internal mechanism to schedule an activity on its private processor to perform the access. This mechanism can be based on the IO device's local memory shared among its activities.

At access completion, control needs to be transferred back from the IO device's processor to a CPU, in order to perform service epilogue tasks such as telling about the event to a waiting customer. This requirement implies that the IO device needs to revoke ownership from the CPU's current owner. The CPU device provides a service to request this reclamation, that is, to *raise an interrupt*. This service decides whether to reject or honor the request, based on the relative precedence of the IO device and the current owner of the CPU. The precedence information is prescribed by the CPU driver whenever the CPU

device is told its new owner. If the request is honored, then the IO device becomes the CPU's current owner. For protection and efficiency reasons, before the ownership switch takes effect the CPU driver is notified of the interrupt by the CPU device.[9] The CPU driver then determines the length of the slice that the interrupting IO device is granted with. The former owners regain ownership of this CPU at the end of this slice, or, pending the CPU driver's discretion, of another CPU at another time. Finally, the CPU device invokes a designated service of the IO device, similarly to invoking a scheduler's scheduling service upon CPU allocation.

Therefore, similarly to a scheduler, an IO device needs to establish an interface with the CPU driver, in which its precedence for CPU ownership is negotiated, and a binding to its interrupt-handling service is handed. The CPU driver decides precedence levels of IO devices/schedulers based on external parameters provided statically at system-initialization time or dynamically by a privileged user called the *System Administrator*. Bear in mind that no matter what the precedence is, the slice allocated to an IO device/scheduler is restricted by protection considerations. We assume that the CPU driver would routinely reallocate CPU's to satisfy pending allocation requests.

To perform an IO operation into/from a customer's address space, an IO device must present a valid access permit to the memory device that performs the access. For efficiency reasons, we assume that the interface with a memory device allows translation of an access permit from virtual to physical addresses. Therefore, an IO device can avoid the presumably longer address translation process when performing a mass copy, by copying directly into physical memory. In a copy between two memory devices the requester of the copy is viewed as an IO device.

The model principles presented in § 2.1 are sufficient for the management of other physical resources. We exemplify management of a disk space and network communication bandwidth in discussing specific design issues in § 3.4.

---

[9]Specifically, the CPU is revoked from all owners. The CPU driver is reactivated via a return from a service it has formerly invoked to set the CPU ownership.

2.5. Miscellaneous

The model assumes the existence of an accounting system responsible for the maintenance of accounts. Similar to Dennis and Van Horn's model [25], there are *principals* that pay the bills. The System Administrator mentioned above is also responsible to add principals and set their balances. The accountant server issues account ids to the principals. It provides services to debit and/or credit accounts. The OSB servers charge for using their resources in order to reduce contention, and so can do any server. Each activity and server are equipped with an account id.

In some situations an asynchronous service invocation is required in order not to suspend the invoking service. One notorious example is at interrupt handling: the interrupt-handling service might need to notify a customer awaiting this event, by invoking a reply service of the customer; however, the latter service cannot be entrusted to return quickly, and so accesses to the device might be delayed for too long. Instead of supporting another invocation paradigm, the model suggests that an asynchronous invocation be accomplished via synchronously invoked services. For instance, to invoke service S asynchronously, one invokes service S' whose purpose is to relegate the invocation of S to another activity.

Communication between activities is necessary to report exceptional events. For example, an activity's scheduler might want to announce intended termination of the activity, so that all uncompleted services in the activity's thread can properly clean up their state. As another example, an event discovered by one activity may imply an immediate change in the execution of another activity running or idling at the same server. Using shared memory in such a case might be impractical. For this purpose, the CPU driver accommodates exception raising on activities. This service is allowed to the target activity's scheduler and to the server at which the activity is currently executing or postponed. When the activity resumes execution, the exception is noticed. Analogous to memory and CPU fault handling described above, the current server's fault-handling service is invoked. Since this mechanism allows communication between a scheduler and servers it is not bound to, we assume that the standard interface mentioned in § 2.2 is extended to include a set of exception types. Notice, however, that for protection reasons a server (say S)

is precluded from raising exceptions on an activity currently running or idling at another server (say R). Should S discover an event worth alerting R, and possibly altering the behavior of a service previously invoked by S, then S should notify R of the event through an ordinary service invocation.

To accommodate proper state cleanup, intended termination of a server should be announced to its peers by the server itself or by another designated server, e.g., its creator. This notification should be done through regular service invocations. As another standard, a binding for any service can additionally refer to a termination-notification service, to be invoked prior to terminating the binding. However, servers cannot count on being properly notified of the termination of other servers they are bound to, or of activities executing their services. Therefore, we designate the CPU device to support servers to interrogate whether a given server or activity has terminated.

The attentive reader has probably noticed a problem we have ignored in § 2.3: How can a Universe manager create a Universe if the manager runs in one of the Universe Spaces? To resolve this problem, a manager starts running as a server mapped to another Universe, and then after creating a new Universe migrates into that Universe to become its manager. This intricate process is supported by the memory driver.[10] For instance, Universe creation can follow these steps: Server S_1 running in Universe U_1 intends to create a new Universe U_2. S_1 acquires the necessary frames and constructs the base segment of U_2 as one of its segments. Next, S_1 invokes a service of the memory driver for the target memory (into which U_2 is mapped). This service creates U_2 as a new Universe and 'downloads' into it its base segment and S_1. Note that S_1 does not disappear, but rather its copy, S_2, becomes U_2's manager. S_1 activates S_2 by invoking the latter's initialization service, or by creating an activity to do so. Through an appropriate interface with the memory driver, S_1 can serve as a recovery server for S_2. Namely, should S_2 fail to handle a fault in U_2, e.g., if its fault-handling service is paged out, then S_1 can be called to rescue U_2.

---

[10]An initial Universe, however, is predefined at system-initialization time. The OSB is loaded into this Universe.

3. Design Issues

In this section we sketch several components of a specific design based on the model. In the next section we highlight several implementation issues. These sections are intended to demonstrate how overhead and complexity can be reduced, and to exemplify how various features of the model can be realized.

3.1. Servers and Services

The default servers in the system are the OSB and a set of commonly-used servers. This set includes compilers, server creators, a Universe manager, command interpreters, schedulers, drivers, providers of higher-level interserver communication mechanisms, and file servers. The servers needed at system-initialization are listed in a *configuration schema*, which specifies also the initial bindings between them. The schema is read by the *initiator*, who then loads and binds the initial servers. Other servers can be loaded dynamically by command interpreters, Universe managers, or server creators, when their services are needed. The initiator takes inventory of existing physical resources and creates their device servers. Devices may be microcoded or software servers, and the shared and trustworthy ones can be embedded in the OSB. The schema is prescribed by the *System Administrator*. He or she may use a privileged interface with the OSB that allows them to add devices, to set accounts and balances, and to change precedence levels. The system administrator is recognized by a special account id.

Language constructs let a server declare the services it *provides* and *uses*, similar to *export* and *import* constructs in module-based programming languages such as Ada® [26] and Modula [27]. A service is in fact a list of operations associated with a descriptive name. Compilers group this information in two separate segments of a standard structure, which are similar to symbol tables for exported and imported identifiers. The standard structure facilitates loaders and runtime-support servers to externally bind service providers with their customers. Services can be registered and located via global (commonly-used) or private *name servers*. The provider of a service hands to the name server a binding to the service, a service

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

name, the server name, and an optional type. Types are application-defined or standard types supported by the name server. The service provider also specifies whether the binding should be duplicated for each new customer, or handed only to the first customer to request it. Servers can query a name server to locate a required service, and get a binding on a successful match. The name server defines matching rules for service and server names, that is, between names associated by a service provider and those specified in a query. Each server is born with at least one binding to a service provided by the default, global name server, which allows obtaining bindings to other services or other name servers.

Each binding contains a vector of operations, a reference and a key (see Figure 3-1a). The matching lock and the operations' addresses appear in the referenced address. The key is verified upon invocation. The service provider may invalidate a binding by changing the lock. Any binding holder may duplicate and transfer it, but not forge a seemingly valid one. A memory access permit has a similar structure (see Figure 3-1b). Its key is verified against a lock at the target address. The permit indicates access rights and memory size. These features support efficient creation, verification, and invalidation of bindings and permits. Additionally, each application can decide and easily implement inheritance rules for services and resources.

A service provider may inscribe in a binding an *initialization* operation and a *termination* operation. Both are *hints*, proposed for invocation at every new binding creation and termination, respectively. The initialization operation should be invoked by the name server, the new customer, or whoever transfers the binding to a new customer. The termination operation should be invoked prior to terminating the binding, by the customer or by whoever terminates the customer. These features, which we assume are observed by well-behaved servers, facilitate service providers to properly initialize and clean up their state. As mentioned in § 2.5, a server may discover through the CPU device whether a given server or activity has terminated. For this purpose, a server must have a unique id, which is invalidated at the termination of the server. Thus, we assume that the Space number is the pair <index, id>, where the index is used to locate the Space in the Universe (see the table in the bottom of the base segment in Figure 3-2), and the id is

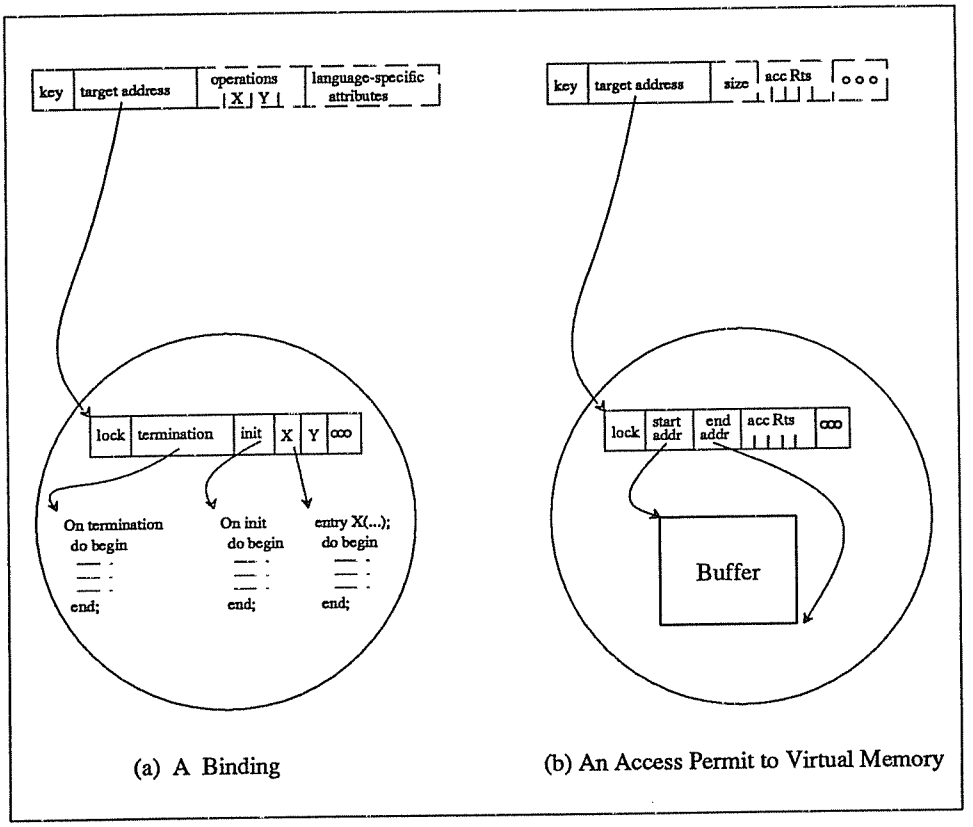

Figure 3-1: A Binding and an Access Permit

(a) A binding. Language-specific attributes are optional. At invocation, the customer specifies a reference to the binding and the operation requested. (b) An access permit to virtual memory. The permit may include the size of the target block and the access rights, but only those specifies at the target server are consulted at access time.
Target address is the 5-tuple <Universe id, Space number, segment number, page number, offset>.

unique per server.

We mentioned in § 2.2 that an activity has private bindings. Where are they located? Each ACD contains a reference to a list of bindings, which are the activity's private bindings. For simplicity, this list is ordered in a standard order, so services can be located easily. For instance, the binding for *block activity* would be the *n*-th in each list. At service invocation request, which as mentioned before is a service provided by the CPU device, the requester distinguishes between bindings which it refers to directly and private bindings.

3.2. Execution Management

The interface between the CPU driver, CPU device and schedulers is enhanced in order to reduce the inherent overhead of dispatching. This overhead stems from the need to invoke a scheduler's scheduling service whenever a quantum allotted to an activity expires. A service provided by the CPU driver and coordinated with the CPU devices allows a scheduler to be periodically *detached* from the short-term scheduling mechanism whenever the scheduler wishes so. The scheduler gives a list of scheduling orders, located at its address space. When the scheduler is in *detached-mode*, then activities are dispatched from this list by the CPU device as long as the scheduler owns a CPU. The CPU device updates the list with the results of completed dispatching or allocations. On a faulty or empty list, the CPU device invokes the scheduling service of that scheduler. The scheduler may dynamically interrogate and rearrange the list, change quantums, or disable/enable being detached. For instance, these changes can be made by a designated activity, dispatched periodically to perform housekeeping chores. Or, they can be made when the scheduler is reactivated via a service invocation.[11]

We have designed a preliminary version of a CPU-allocation algorithm for the CPU driver. It is a multilevel feedback algorithm, based on the notions of *urgency* and *demands*. Each customer of the CPU driver specifies its demand in terms of number of CPU's required (total and how many thereof are needed concurrently), for how long, and the urgency level of the demand. The urgency level is in a range defined by the CPU driver. The CPU driver maintains scheduling queues, one per urgency level. A scheduler may post simultaneously different demands in different queues. CPU allocation by the driver is as follows. Each queue is scanned in turn and schedulers are allocated CPU's in a round-robin fashion. There is an upper limit on an allocation, which is a function of the contention for CPU's (that is, the system load), the urgency level, and the recent history of allocations to that scheduler. The last element is meant to give a

[11]Similarly, a memory device can provide a service to Universe managers to be periodically detached from page fault handling. Upon a page fault, replacement would be performed by the memory device, provided the Universe manager supplies a valid list of free frames, and page swap to or from a swapping device is not necessary.

bonus to schedulers with low or unsatisfied demands, and is somewhat similar to credits used by Kurose *et al* [28] to reduce contention for multiaccess networks. Likewise, there is an upper limit on the total allocation for each queue, in order to provide some fairness to demands of lower urgency. The unsatisfied portion of a demand becomes the scheduler's new demand for the next round, or is added to the scheduler's demand if it has a continues one. The position of a demand in a queue or between queues can change, depending on the scheduler's recent demands and whether they were fully satisfied. The order of scanning the queues is altered with each new demand posted in a queue of an urgency higher than that of the queue currently being scanned. In addition, the OSB may dictate preferred allocation to a given scheduler, for instance, by granting the scheduler's entire demand unequivocally. It should be noted that the CPU driver charges, although differently, for both posting a demand and being allocated a CPU, with charges increasing with the urgency level.

3.3. Memory Management

In this section we merely focus on how the model view of memory mapping and address translation can be realized. We limit the discussion to a direct, index-based mapping scheme. As Figure 3-2 illustrates, the base segment contains the following components: (a) general structural information that tells the memory device the sizes of the tables; (b) a Spaces table, with a fix-size entry per Space. Using a fix-size entry, the memory device can fetch the necessary information of an addressed Space by indexing in the table. Notice, however, that an entry can be extended using "overflow buckets" when new segments are added to a Space; (c) information about the Space, such as the id of the server mapped to it, and a descriptor per each segment; (d) as mentioned in § 2.3, a segment descriptor specifies mapping information and access rights. It may also contain control information used by the Universe manager, such as indication of being shared; (e) page tables.

The memory device maintains a Universes table with an entry per Universe mapped to this device. An entry would include the physical address of the page table of the Universe's base segment, and the address of a fault handler for that Universe. The memory device would also contain caches of data and of

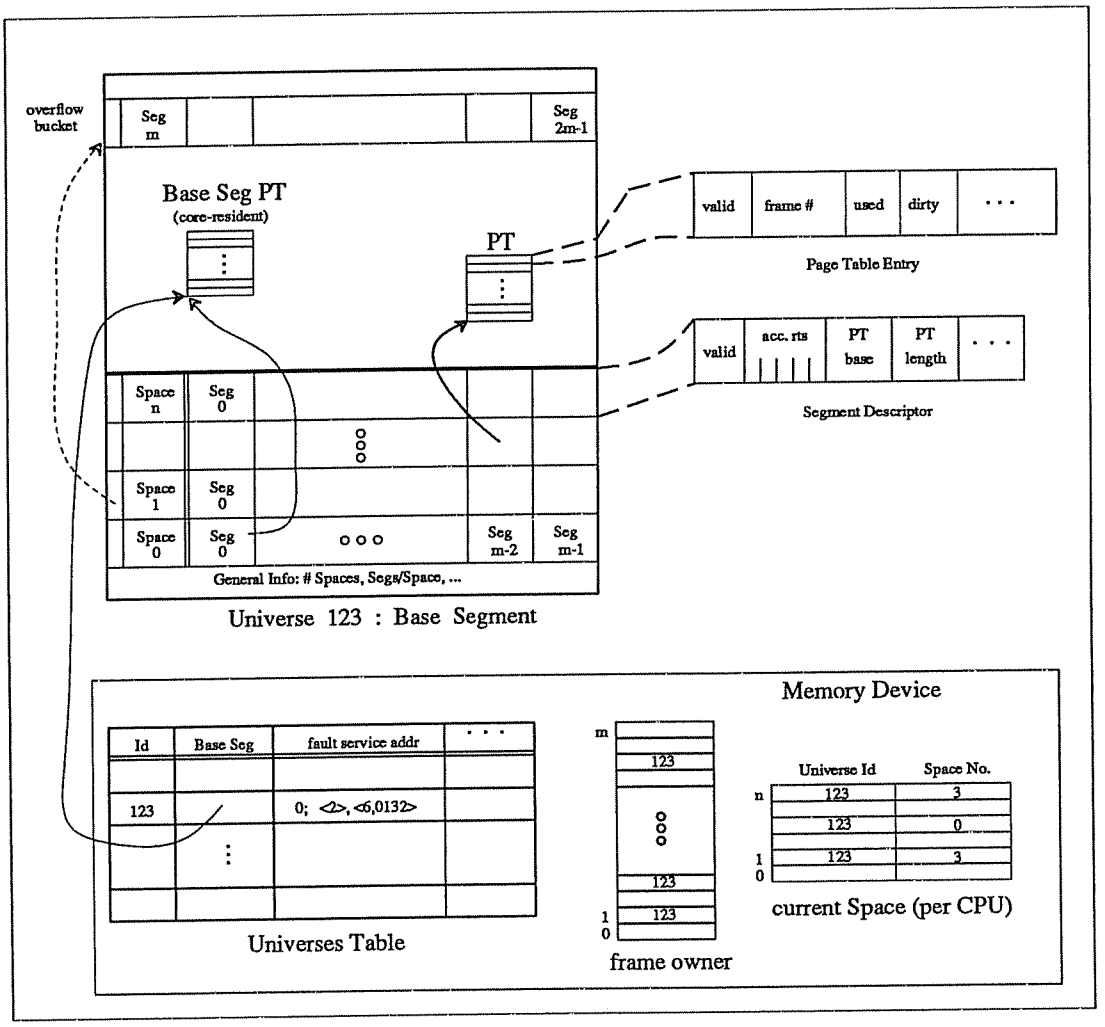

Figure 3-2: Structure and Mapping Information Employing a Directed-Mapping Scheme— An Example
The manager resides in Space 0; the base segment is segment 0 of Space 0. In an inverted-mapping scheme, this example would change in the following: (a) There would be no page tables, (b) A segment descriptor would contain a group id instead of PT base and PT length, and (c) An inverted table, similar to the *frame owner* table would contain at each entry two pairs of <group id, page #>: one for the manager, one for other Spaces.

recent translations.

Following the example of Figure 3-2, translation of the virtual address {<4>,<2,32>} in the Space <123,3> follows these steps: (a) The device locates the page table of Universe 123's base segment, (b) provided Space 3 is valid, its entry in the Spaces table is located in virtual memory (as you notice, each entry is of a fix size); using the latter page table, the entry is fetched from physical memory, (c) the descriptor of segment 4 is located in virtual and physical memory (if necessary, following pointers in an overflow

bucket), (d) provided the segment is valid, the segment's page table is fetched, (e) finally, the frame number of page 2 is obtained. Using caches of translation information, the device can skip memory look-ups at repetitive accesses to a Universe, Space, or segment.

Translation information held by the memory device must be consistent with the mapping information maintained by Universe managers. For this purpose, it suffices that a memory device maintains the relationship between its translation tables and the respective mapping information. Additionally, a memory device can let a Universe manager explicitly invalidate specific items in the device's translation tables.

3.4. Devices

In this section we exemplify how the model principles can be employed for managing two other physical resources. Applications may include private disk devices. A disk device divides its disk space into *blocks*, which are the units of allocation. A disk driver might be necessary to coordinate allocation of several disks, mainly for convenience of the disk users and the efficiency of disk utilization. For synchronization or protection reasons, the driver might require that accesses are made through its services only, in which case the driver is the sole owner of the disk space; otherwise, the disk device is given the ownership information, and customers access their disk shares directly through the device's services. By owning part of a disk, a server can provide filing services or define virtual disks and provide services to control them.

A single machine can be connected to other machines via one or more networks. Each network connection is controlled by one network-controller device. The resource is the communication bandwidth and the unit of allocation is a time slot to send packets. The device provides *send* and *receive* services of byte streams. Reliability and flow control are relegated to communication servers implementing higher-level protocols. A communication server has to create a *port* into which received messages can be queued. The server associates with a port a *logical name* (so that remote peers can locate the port), an access permit to memory where messages are placed, and a binding for a service to be invoked upon message receipt or send. Allocation of the bandwidth is analogous to allocation of CPU's, and similarly the device may employ a dynamic precedence scheme.

3.5. Miscellaneous

In order to support the decision making of schedulers, Universe managers, and other servers, various devices and drivers expose usage information of their resources. For instance, the CPU driver collects load information, and provides read-access permits for schedulers to inspect this information. Similar facilities are available at the memory and network drivers. The information includes a price vector for the resource. We assume that some servers provide parametric policies rather than a single policy. Thus, one scheduler can serve several applications with different scheduling policies.

We assumed above that servers are entrusted by their customers, and thus the system does not interfere in how much they charge for resources and services. However, some restrictions are needed to protect a customer from a faulty server. Thus, the accountant allows a server to set a limit on both the amount its account can be charged and the total of all charges. Notice that setting such a restriction may result in other servers refusing to provide services or resources to the focal server, if they cannot charge for their services/resources.

As an additional protection measure, the activity's and the server's account ids cannot be read or modified by servers other than the accountant. How charging can be possible then? The activity's account id is found in its ACD, and charging thus can specify the currently executing activity — which tantamounts to the activity charging itself. A server does not pass its account id to another server when invoking the latter's service. Rather, the former server may deposit its account id in reserved slot(s) in the ACD (of the currently executing activity). Charging requests, therefore, can refer to this slot(s).

4. Implementation Issues

At the implementation level we assume that the system is structured as a collection of CPU's, memories, and IO resources connected with a common bus — similar to the emerging class of multiprocessors known as *multis* [29]. A device can have hardwired or microcoded representation (services and structures) loaded into its private memory and executed by its private processor, both attached to its resource. It can also have software or microcoded representation executable by a CPU.

To reduce potential overhead at crossing server domains, several cooperating servers can be linked at load time to form a single server, similar to the way Ada packages or Modula modules are linked to form a single program. In such a case, each 'module' can still be viewed as a separate server by its customers. To accommodate linking, allocation of segments numbers by compilers should be standardized, so that linkers can merge the servers. For example, the instructions of a server's program would always form segment *n*. Moreover, linking servers together should not change invocation protocols. Therefore, the service invocation and service return machine instructions behave as procedure call and return instructions when the invoking and invoked services reside in the same Space. In order to avoid indirect level of addressing of segments, as in Multics, the segment number field of a virtual address can be a register number that contains the actual segment number.

To make service invocation more efficient, we assume the CPU architecture provides a large register file and a register window per invocation [30]. Therefore, using a stack for invocations is not necessary. The actual register file and window sizes would be a tradeoff between the efficiency of Space switch and activity switch. We anticipate that devices' services and several drivers' frequent services, such as dispatching, interrupt granting, and CPU allocation, be implemented in hardware or microcode. They can be invoked through generic machine instructions, similar to IO instructions in conventional architectures such as the IBM System/360 [31].

Every CPU device interfaces the memory subsystem through an attached memory management unit (MMU). The MMU contains a cache of recent or anticipated accesses. We assume a virtually-addressed cache, that is, every entry is prefixed with a unique identifier, such as the triple <Universe id, Space number, segment number> or the pair <server id, segment number>. Therefore, there is no need to flush the cache upon Space or activity switch, which in turn reduces the overhead of service invocation, service return, and dispatching.

For the sake of cache consistency, caches are updated by memory devices via a common bus to invalidate or replace entries modified in other caches. In particular, mapping information held at CPU's

should be modified when this information is modified by the relevant Universe managers. Therefore, via regular machine instructions a Universe manager may direct the CPU's to invalidate mapping information relevant to its Universe. The MMU holds a table which indicates the memory device to which each Universe is mapped, so that memory references can be directed to the appropriate memory devices. Thus, Universe ids must be globally unique.

Maintaining virtually-addressed caches with large prefixes might be architecturally complex. If alternatively the cache is indexed by physical addresses, an MMU contains also a translation look-aside buffer (**TLB**) to hasten searches in the cache. A TLB stores <virtual address, physical address> pairs and is searched without translation. Only on TLB or cache miss a translation-and-access request is passed to the appropriate memory device, which returns also the yielded physical address. Having a large TLB to hold all translations for the cache might impose efficiency problem. Therefore, the alternative would be that each MMU performs the required translation. In this case, all MMUs and memory devices must comply to a single mapping scheme. This is somewhat a restriction on the extent memory devices can differ from each other. Note that memory devices can still differ in other aspects, such as frame sizes and access speeds.

We have ignored so far the question of how servers are named and where context information is held. Few standards should be made at the implementation-level. First, each server has a context descriptor of a standard format at a well-known location in its Space, for instance at the beginning of segment 0. This descriptor, among other values, contains the server's name and account id, and is set by its creator. Reading designated fields of this descriptor, such as the server name, is a services provided by the CPU device. Second, the server id equals the id of the Space it is loaded into, and the latter id is modified by the Universe manager upon each Space deallocation. In this way, it is easy to validate whether a given server is still alive. Additionally, all permits and bindings issued by a server are automatically invalidated when the server terminates.

To simplify memory ownership verification at each access, a frame is marked with a *key* that identifies its *current* owner. This key equals the Universe id, so at each memory access it is verified against the id of the addressed Universe. This usage of keys is similar to a method employed by IBM System/360 [31]. The key is stored in the frame or in a single inverted table at the memory device, as shown in Figure 3-2. Ownership of other resources can be represented by similar primitive capabilities or by lock-key pairs, where the lock is maintained by the device and the key is presented by the invoker of an access service. We assume that such keys as well as keys stored in bindings and permits (see Figure 3-1) are long enough to prevent forging a seemingly valid one.

5. Discussion

The FOCS model supports maximal openness, limited only by what we consider as minimal protection constraints that are necessary in a multiuser environment. An application can substitute most of the existing services and resources, and construct its own virtual operating system from a set of servers and devices. Decomposition of the OS and of applications to separate servers allows easy replacement of services and resources, or policies and mechanisms. The replacement is dynamic — without need to recompile or relink the OS.

In some sense the system in our model is closed: similar to conventional operating systems resource allocation depends on decisions made by the OSB. The model tries to minimize this centralized control to only as little as is dictated by the contention between users and by their protection needs. Openness is achieved when applications are allocated the resources. Once an application can bid for the resources it needs and get them, it can provide the policies and mechanisms to use them. In particular, the model was carefully crafted to support real-time computations. A scheduler can dispatch an application to run uninterrupted, provided it buys urgent time or increases its precedence for as many CPU's as it needs. To further improve execution time of a computation, its scheduler or servers executed by the computation can negotiate memory replacement preferences with their Universe managers. To simplify coordination between servers, the servers that form, or serve a single application — so called *problem-oriented* servers, can be

combined into a single server.

Our model allows coexistence of different paradigms of computation and communication. Servers can provide higher-level, connectionless or connection-based interserver communication mechanisms, with different reliability and synchronization flavors. Issues of sharing resources and inheritance of access to resources or services can be decided by the application-specific Universe manager or server creator. The model supports a variety of possible approaches to these issues. The granularity of execution tasks, resource units, shared memory buffers, communication and IO transfer units can all be decided by each application. Likewise, transactions can be supported at different levels by various applications, each selecting the appropriate mechanisms to implement its required semantics for concurrency control, recovery, and reliability.

The model supports sharing of memory in various levels. Universe managers can share physical memory through allocation, and each of them decides sharing of physical memory among its customers. Spaces can share segments through mapping, each possibly with different access rights. Servers can share address spaces by being merged into one server (that is, loaded into one Space), by sharing segments, or by having access permits to each other's Space. Other resources held by a server can be shared with other servers through allocation, permits, or services.

In the following we evaluate the aspects of openness and their interplay, as exposed in all three levels of our approach: model, design, and implementation.

- *On Protection*:

The basic, default level of protection is low in order to fit protection requirements of mutually-trusted servers, and thus avoid overhead and limitations imposed by unnecessary protection measures. The lack of parameter or capability checking at service invocation, except for the validity of the referred address, makes communication among cooperating servers efficient. However, mutually-suspicious servers or generally-used servers that *must* protect their resources would suffer greater overhead in implementing their own protection mechanisms. Some of the additional protection measures, though, should be very

simple to implement — requiring merely checking a scalar or a vector of values. This approach fits well the requirements of database and transaction management systems. Such systems prefer that the OS provides rudimentary protection for communication with them, and let them control their internal interactions in more efficient and sophisticated ways [3].

Relegating capability verification to servers can be less efficient than if the verification would be done by the hardware in a capability-based architecture. But using capabilities in our approach is not mandatory, and thus communication among cooperating servers is more efficient. It is still an open question whether some form of more restrictive default capability mechanism, possibly supported by the hardware, can be more beneficial in terms of both protection and efficiency.

Our approach does not provide protection requirements that might be considered necessary, but supports different means to accomplish them. We consider here two examples. First, a server *S* cannot control the distribution of permits, bindings, and private capabilities it has passed to its customer *C*. That is, *C* can transfer them to another server *S'*, and so forth. We do not consider this issue a protection gap. Instead of passing a permission to a resource or service, a server can impose an intermediary service to access the target resource; the server would then check at each access the validity of the access requester. This example illustrates a trade off between protection and access efficiency. A different balance between the two can be selected by each application designer.

Second, a customer of a service is protected very little from its service providers. Its resources, including its address space, and to some extent its account id are protected. However, it cannot prevent misuse of its account id, unfair charging for services or resources, or misprovision of services. We do not consider this issue a problem, since a server can always choose services from servers it entrusts, or it can substitute them. This example illustrates the trade offs between openness, protection, and efficiency. The default protection measures are tailored to applications consisting of mutually-trusted servers. They are intended to protect against errors a server cannot otherwise protect itself. Adding more measures to protect servers from all potential hazards inflicted by untrusted servers would incur higher execution overheads

and would backlash in impeding openness itself (cf Hydra [32], especially the designers' retrospectives in Ch. 7).

- *On Execution Efficiency*:

Once the OS is decomposed to separate entities, the communication among them should be efficient, especially for frequent and urgent tasks such as page replacement, CPU scheduling, and interrupt handling. Therefore, we were careful to simplify server-domain crossing. between servers. The semantics chosen for service invocation, service return, and resource access, and the assumptions made about virtually-addressed caches and register windowing, were all motivated by this concern. We presume therefore that interserver communication would be (almost) as efficient as calling a procedure in a different locality of the same address space.

Service invocation and return can be very efficient since parameters and results are passed in registers. Since there is no default stack, then service invocation and return do not require time-consuming stack manipulation operations. The efficiency of service invocation/return can decrease when the environment of the invoking service must be preserved in a local stack frame. The invoker needs then to extract a frame from a heap and later free it. This overhead and its complexity can be reduced by employing simple heap management techniques, such as maintaining a list of free stack frames sorted by frame size. Moreover, frame sizes can be decided at compile time [33] or be fix, trading space efficiency for execution efficiency. Passing access permits for buffers used as parameters or results of a service invocation incur additional overhead when the buffers are accessed. We presume that such overhead is lighter than that incurred in conventional systems that provide more heavy-weight IPC mechanism, requiring for instance transferring the buffer in a message back and forth.

To further reduce the communication cost, cooperating servers can be linked to form a single server or be loaded into a single Space, without reducing their flexibility to act as separate servers. The process of binding a server with its service providers should not bear high performance cost, as bindings can be obtained at load time or via parameters; bindings can be obtained in groups, for instance as a library of

bindings.

Inefficiency increases with the decrease in the "trust-level" among servers, as the overhead of protection increases. However, by not imposing complex or costly default protection mechanisms, our approach allows servers to tune their mechanisms to avoid extra overheads. The most inefficiency-implying feature is the need to undertake bookkeeping of resources allocated to other servers or used by activities, and checking periodically if these servers/activities are still alive. This bookkeeping, though, is not radically more complex or expensive than what conventional OS kernels and shared utilities do to protect their resources.

On the positive side, the decomposition of a monolithic OS to separate servers offers some efficiency gains, as each server controls fewer customers. For instance, scheduling decision making can be faster, as it involves scanning shorter lists. In addition, since scheduling is problem-oriented, decomposed scheduling presumably results in more efficient resource allocation. The suggested detached-mode interface for CPU scheduling and memory replacement reduces the interface cost for communicating scheduling and memory allocation decisions to the devices. It allows schedulers and Universe managers to define medium- to long-term policies, while still being responsive to short-term changing requirements.

The support of activities in our approach offers additional gains of execution efficiency. Various applications such as DBMS's [3, 8] and commonly-used servers [5] require support of light-weight threads, each devoted to a single interaction with a customer. These threads should be cheap to create and destroy. The application should be allowed efficient control over scheduling of these threads, which involves dynamic setting of priority and dependence parameters. Carrying a single interaction or transaction means that a thread should efficiently traverse across layers of services, down to the IO and communication devices. Our approach accommodates these notions by the semantics chosen for activities and the facilities supporting them.

The issue of interrupt handling demonstrates the interplay between protection, efficiency, and complexity. Interrupts are admitted as soon as is dictated by the device's relative precedence. The device can

perform whatever urgent tasks are required. Nonetheless, there is a back-pressure on the device against monopolizing the CPU.

- *On Architectural Support:*

The major requirement from the architecture is to support virtual-to-physical address translation and memory ownership verification. These functions performed in software cannot attain practical efficiency. Our solutions in all three levels of the model, the specific design and the implementation, demonstrate a balance in the interplay between architectural complexity, access efficiency, protection, and openness. Untrustworthy Universe managers can treat mapping tables as ordinary data structures in a protected way, while translation can be performed directly by a memory device. The extra mapping levels increase the complexity of the translation mechanism. The mapping structure is based on notions borrowed from the VAX® memory architecture [34], in which all page tables reside in the kernel address space. In the FOCS model this scheme is extended by (a) there is no single kernel; a 'kernel', i.e. a Universe manager, is selected dynamically, (b) variable number of segments per server is allowed as in Multics [24], and (c) the structure of a Universe, Space and segment is more complex. Unlike Multics, we avoid an extra level of indirection by supporting sharing of mapping information rather than sharing segment numbers. Overall, we believe that these extensions would not preclude supporting address translation in the architecture. The incurred complexity is not high, compared to other systems that support structured memory mapping such as VAX or iAPX-432 [35]. A designer of a specific system can choose to reduce the architectural complexity by reducing openness, for instance by fixing the number of segments per Space.

Ownership verification does not increase the complexity of the memory architecture, since it requires merely comparing a key. Setting ownership can be done in microcode or even software. Verifying access permits is merely comparing a vector of values. Other functions relegated to the architecture or assumed to be implemented in microcode are very simple. Service invocation and return are simple extensions of regular procedure call and return instructions. Checking permits and bindings are similarly simple, since they

---

®VAX is a registered trademark of Digital Equipment Corporation.

involve merely comparing one or several fields or branching to the address of the service operation.

Our assumption of employing virtually addressed caches in the MMU's is in line with recent trends in cache design [36]. This assumption is important in reducing Space-switch overhead upon service invocation, service return, and activity dispatching. In making the cache somewhat more complex, we simplify the function of the MMU, which performs virtual-to-physical address lookup instead of translation. This means that memory devices do not need to distribute mapping tables to all CPU's whenever the tables get modified by Universe managers. If the size of translation buffers is limited by space restrictions, and thus address translation (rather then lookup) needs to be performed at the CPU chip, then the implementation would require all memory devices to adhere to a single mapping scheme. This issue demonstrates a trade off between openness (allowing memory devices to employ different mapping schemes), architectural complexity and access efficiency.

- *On Programming Complexities*:

A major consideration in shaping the model and particularly the specific design was to restrain the programming complexity incurred by protected openness. Below we compare programming complexity in our approach to that in a conventional system. We focus on three levels of program or programmer sophistication. First, there is no extra complexity imposed on *novice* programmers, as they can be oblivious of the internal structure of the real or virtual OS they use. Programmers do not have to write supporting servers to provide the services and manage the resources required by their programs. Services required by a program can be selected by language processors such as compilers, linkers, and runtime packages. Services can be also selected through libraries, which the programmer selects by merely knowing their high-level functionality (e.g., that the library is needed to obtain Unix[TM]-like filing services) without having to select the lower-level services herself.

Second, writing a more sophisticated program, such as a server that provides high-level services, is basically of comparable complexity as writing a utility program in a conventional system. Both programs

[TM]Unix is a Trademark of Bell Laboratories.

have to cope with synchronization and mutual exclusion of multiple concurrent threads of control executing in the server/utility. In our approach these threads are scheduled by other schedulers, so synchronization and mutual exclusion, as illustrated in handling critical sections, is somewhat more complex. To reduce this complexity, the model and the specific design offer means for proper initialization and termination of services, bindings, and activities. Charging for resource usage adds programming complexity and execution inefficiency to servers. Charging in principle is not peculiar to a fully open system — other systems employ the notion of servers charging for their resources and services [37], and incur similar complexity or inefficiency. Choosing simpler charging schemes at the design level can reduce the complexity, and potentially reduce the efficiency of resource utilization.

Third, writing a server to provide low-level services — the counterpart of an *OS module* in a conventional system — is in several aspects simpler in our approach than in a conventional system. Only problem-oriented semantics, that is, of a particular application or of a class of applications, need to be addressed. For instance, it is simpler to write the memory manager of a DBMS in our approach because one can choose the services that provide the required semantics. Since one can implement services above the low-level services of devices, or have private devices, there is no extra complexity involved in trying to bypass or undo unwanted facilities as in some other systems [3,1,5]. We believe that in general it is simpler to implement a policy than to tell a monolithic OS the preferred policies, especially if the OS accepts *hints* rather than *absolutes*. Additionally, it is simpler to program the interaction with devices having the support for activities, each of which can be devoted to one access transaction. In another aspect, it is more complex to write such a server in our approach, since physical resources must be negotiated for.

- *On Dependency between Services and Service Domains*:

The model provides a unified view of services, with no dependency between service domains in the level of openness. Specifically, using or replacing one set of services for a given domain such as memory management, does not restrict the flexibility of selecting services in other domains such as buffering and CPU scheduling.

In practice, applications may choose to restrain this independence. For reasons of execution efficiency and provision of required semantics, it might be necessary to coordinate services in different domains. For instance, the scheduler and Universe manager of a given application may need to coordinate their policies to achieve better performance. As an example, before scheduling an urgent computation the scheduler could prompt the Universe managers of the involved servers (if known) to prepage their working sets. This might be a complex task, especially since different servers could be dynamically involved. Likewise, a server providing interserver communication services with special features, such as copy-on-write or urgent messages, would rely on schedulers and Universe managers to support the semantics of its services. Therefore, it may turn out that at some cases it is more beneficial to combine all dependent services into one server. These decisions are left to each application designer to make.

- *Miscellaneous*:

Dividing a resource among several applications could result in uneven utilization of the resource. For instance, one Universe may underuse its frames, while at the same time another Universe experiences a high page-fault rate. This problem is alleviated to some extent in a conventional system, where physical memory is dynamically and frequently multiplexed among processes by a monolithic memory manager in the OS. However, the memory manager of a conventional OS observes *when* code and data is used but not *how* it is used. Therefore, a general page replacement algorithm which is efficient from the OS's point of view can result in inefficient memory usage by a given application [1]. In our approach, the Universe manager, knowing usage patterns of its customers, can improve the utilization of its share of physical memory.

Our approach incorporates several features to improve global utilization of resources as well. We believe that the accounting system can contribute to improve utilization, particularly if it supports the notion of crediting a server for not using a resource, in addition to charging the server for using the resource. Unfortunately, the price–credit balance to achieve optimal resource allocation is not clear. For example, cost-conscious servers may use too few resources, and further reduce resource utilization. Other

means of back-pressure on servers to release unused resources might be necessary. Resource utilization is enhanced by the model through letting allocation be independent of static hierarchies, and through support of resource revocation.

Hierarchies usually have conflicting merits, and thus our approach supports hierarchies but does not impose them. Service provision and resource allocation is inherently hierarchical. For instance, a mailing server might use DB services, which use some structured filing services, which use flat filing services, which use disk access services. Or, a scheduler can itself be scheduled by another scheduler, which is scheduled by another scheduler, and so forth. In our model, these hierarchies are formed and changed dynamically and efficiently — without needing explicit introduction of new levels as in other hierarchical systems [23, 38]. Moreover, the model accommodates hierarchies of virtually infinite depth in any service domain or across service domains, including in memory and processor management domains. Service domains are mutually independent in their hierarchical structures. On the other hand, hierarchies can be bypassed Applications can directly access resources at the lowest possible levels. This feature is especially important to efficiently accommodate repetitive, frequent, or urgent operations such as address translation, handling page faults, and quantum expiration.

Besides the efficiency drawback of rigid hierarchies, they may introduce complexity in various situations. Efficiency considerations may motivate special *deals* between components at different levels, which complicate maintaining the hierarchy. Extra complexity is incurred when deals break, for instance due to the abrupt termination of one of the parties involved in a deal. To avoid such a problem, we make the deals visible and fully supported. Resource-allocation deals among servers at different logical hierarchical levels are possible. They are made through the device or driver that control the resource.

6. Related Work

The FOCS model borrows several notions from other conceptual frameworks as the object model [15], server–client model, virtual machines [16], and monitors [39]. Our model extends them to support further customization of services, direct control of physical resources, and dynamic reconfiguration of the

system. Our model simplifies them by removing features that obstruct openness. The object model and the module construct [40] promote decomposition of a system and information hiding. A service in our model is analogous to an entry point in an object/module. A service, however, does not have a system-defined type or access rights associated with it. A service invocation is conceptually similar to a cross-process procedure call in the language Distributed Processes [41]. Servers can be viewed as monitors [39] regarding their tasks in mutually excluding and scheduling activities.

Many designs permit system openness to some extent. Neither of them can be considered as an adequate framework to study full openness in a multiuser environment. The open system [42] and Pilot [43] facilitate static or dynamic modification of the OS, but being single-user systems they ignore many important aspects of sharing and protection.

The idea that application-level servers may control processor scheduling was first demonstrated by Hydra [32, 18]. However, *Policy Modules* in Hydra, which were responsible for setting scheduling policies, were OS components. Such a module could not be installed by any application, since it had to be trustworthy and 'ordained' by the OS. Hydra bound CPU scheduling and memory management domains together, in that memory replacement decisions were made by the kernel based on the scheduling state of a process. Using a highly protective environment that incurred high performance overhead, Hydra eventually precluded openness of low-level and frequent services. As the designers of Hydra admitted in retrospect, the elaborate protection mechanisms were barely useful for any application. Other object-oriented systems [17, 35] support easy replacement of high level services and logical resources, but untrustworthy application-level objects cannot be allocated physical resources or control them directly.

Several designs address openness through hierarchical structuring of the system. VM [23] allows any user to introduce a full-fledged virtual OS above another virtual or real OS. Communication and resource sharing between OS's at various paths or at non-neighboring levels of the hierarchy are very limited. Except for few implementation short-cuts, services and resource management traverse rigid hierarchical lines. Applications view only virtual resources. Habermann *et al*'s work [38] was an attempt to

develop families of OS's, similar to computer architecture families. They define a rigid hierarchy and statically associate functions with each level. As in our model, each 'server' defines a different address space, in which several processes can run concurrently. Sharing among 'servers' is limited. Special instructions allow cross-space invocations. Interserver communication is very expensive in terms of execution time, since at each invocation a complete address space must be created. CAP [44] defines a process tree in which every node is a *coordinator* for its offspring processes, supporting them with the services they need. All services originate at the *Master Coordinator*, which is the OS kernel. Similar to the former hierarchical systems, sharing and communication among processes at non-neighboring nodes in the tree is restricted. Due to the expensive capability-management mechanisms and the overemphasized hierarchical structure, the hierarchy was practically limited to two levels only, and the openness of OS services was constrained.

Various message-based systems that follow a general server–client model [19,20,21,22,37] permit application-level utilities to provide high-level services to applications and to dictate allocation policies to the OS kernel. In most systems, however, the utilities must be considered part of the OS, since they are irreplaceable by general users and must be generally trustworthy. Mach [7] and Accent [20] open memory management to some extent by letting applications specify policies and implement mechanisms. These two systems provide an elaborate but *closed* IPC mechanism. Mach and some other multiprocessor operating systems [45] support the notion of multiple *threads* within a single address space which are visible to the OS kernel. Unlike the FOCS model, a single scheduler must be told or must decide the grain, execution frequency, and mutual dependency of these threads.

In summary, our work should be compared to the related work not only on the extent of openness attainable, nor on the relative merits and deficiencies observed in various aspects. Rather, we would like to distinguish our work for providing a conceptual framework to expose problems with openness and evaluate the interplay between its aspects.

7. Conclusion

We have presented a conceptual framework to build a fully open system, and discussed the practicality of our approach in light of protection, efficiency and complexity considerations. The thrust of our work is in redefining the role of the operating system and placing the line between the OS and the applications. Our work offers a novel view of resource management. The moral of this paper should be that an operating system and the physical resources can be fully open to direct control of unprivileged, and in fact untrustworthy components, without breaching protection, undermining efficiency, or demanding extensive architectural support. A small increase in complexity can buy a lot of flexibility. A specific system can trade off full openness for increased efficiency or decreased complexity.

We view our work as opening the door to grasp the major questions of system openness and understand the interplay between its aspects. There are still open questions regarding full openness, in particular about the practical impact of protected openness on efficiency and complexity. Our research plans call for a more detailed design and further evaluation of implementation techniques. We will be looking at extending our model to distributed systems.

8. Acknowledgements

This work was inspired by Michael Stonebraker's paper [1]. We owe our gratitude to Marvin Solomon for his invaluable suggestions and support, and to many colleagues for providing constructive criticism, in particular to Raphael Finkel, Hung-Yang Chang, Prasun Dewan, Kishore Ramachandran, and Cui-Qing Yang for many fruitful discussions. Roger Haskin of IBM Almaden deserves special thanks for his encouragement at an early stage of this work.

9. References

1. Michael Stonebraker, "Operating system support for Database management," *Commun. of the ACM* 24(7) pp. 412-418 (July 1981).
2. Michael L. Scott, "The interface between distributed operating system and high-level programming language," *Proc. of the 1986 Int'l Conf. on Parallel Processing*, pp. 242-249 IEEE Computer Society, (August 1986).
3. J. Eliot B. Moss, "Getting the operating system out of the way," *Database Engineering* 9(3) pp. 35-42 IEEE Computer Society, (September 1986).
4. Donald D. Chamberlin, Morton M. Astrahan, Michael W. Blasgen, James N. Gray, W. Frank King, Bruce G. Lindsay, Raymond Lorie, James W. Mehl, Thomas G. Price, Franco Putzolu, Patricia Griffiths Selinger, Mario Schkolnick, Donald R. Slutz, Irving L. Traiger, Bradford W. Wade, and Robert A. Yost, "A history and evaluation of System R," *Commun. of the ACM* 24(10) pp. 632-646 (Oct. 1981).

5. Michael L. Scott, "Design and implementation of a distributed systems language," Ph.D. dissertation, Technical Report 563, University of Wisconsin–Madison Computer Sciences (May 1985).

6. 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) pp. 520-530 (June 1985).

7. Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tavanian, and Michael Young, "Mach: A new kernel foundation for Unix development," *Proc. of Usenix Summer Conference*, (July 1986).

8. 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) pp. 23-34 IEEE Computer Society, (September 1986).

9. Karsten Schwan, Win Bo, and Prabha Gopinath, "A High-performance, Object-oriented operating system for Real-Time, Robotics applications," *Proc. of the 1986 Real-Time Systems Symposium*, pp. 147-156 IEEE, (December 1986).

10. Robert P. Warnock III, "User-Mode development of hardware and kernel software," *USNIX Summer Conference*, pp. 224-226 USNIX, (June 1984).

11. Abraham 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). (And personal communication)

12. Yeshayahu Artsy, Hung-Yang Chang, and Raphael A. Finkel, "Charlotte: Design and Implementation of a Distributed Kernel," Technical Report 554, University of Wisconsin–Madison Computer Sciences (August 1984).

13. Y. Artsy, H-Y Chang, and R. Finkel, "Interprocess communication in Charlotte," *IEEE Software* 4(1) pp. 22-28 (January 1987).

14. 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.

15. Anita K. Jones, "The Object Model: A conceptual tool for structuring software," in *Operating Systems—An advanced course*, ed. G. Seegmuller,Springer-Verlag, New-York (1978).

16. Robert P. Goldberg, "Survey of Virtual Machine Research," *IEEE Computer* 7(6) pp. 34-44 (June 1974).

17. Guy T. Almes, Andrew P. Black, Edward D. Lazowska, and Jerre D. Noe, "The Eden system: A technical review," *IEEE Trans. on Software Eng.* SE-11(1) pp. 43-58 (Jan. 1985).

18. 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*, pp. 132-140 (November 1975).

19. F. Baskett, J. H. Howard, and J. T. Montague, "Task Communication in Demos," *Proc. of the Sixth Symposium on Operating Systems Principles*, (November 1975).

20. R. F. Rashid and G. G. Robertson, "Accent: A communication oriented network Operating System kernel," *Proc. of the Eighth Symposium on Operating Systems Principles*, pp. 64-75 (November 1981).

21. Raphael Finkel, Marvin Solomon, David DeWitt, and Laurence 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).

22. David Cheriton, "The V Kernel: A software base for distributed systems," *IEEE Software* 1(2) pp. 19-42 (April 1984).

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

24. A. Bensoussan, C. T. Clingen, and R. C. Daley, "The Multics Virtual Memory: Concepts and Design," *Commun. of the ACM* 15(5) pp. 308-318 (May 1972).

25. Jack B. Dennis and Earl C. Van Horn, "Programming semantics for multiprogrammed computations," *Commun. of the ACM* 9(3) pp. 143-155 (March 1966). Reprinted in *Commun. of the ACM* 26(1) pp. 29-35 (January 1983).

26. Harry Katzan, *Invitation to Ada and Ada Reference Manual*, Petrocelli Books, NY (1980).

27. Niklaus Wirth, *Programming in Modula-2*, Springer-Verlag, NY (1984).

28. James F. Kurose, Mischa Schwartz, and Yechiam 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*, pp. 70-77 IEEE, (May 1985).

29. C. Gordon Bell, "Multis: A new class of multiprocessor computers," *Science* 228 pp. 462-467 (April 1985).

30. David A. Patterson and Carlo H. Sequin, "A VLSI RISC," *IEEE Computer*, pp. 8-21 (September 1982).

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

32. W. A. Wulf, R. Levin, and S. P. Harbison, *HYDRA/ c.mmp: An Experimental Computer System*, McGraw-Hill, New-York (1981).

33. 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*, (1982).

34. Digital Equipment Corporation, *VAX-11 Architecture Reference Manual*, Bedford, MA (May 1982).

35. Elliot I. Organick, *A Programmer's view of the Intel 432 system*, McGraw-Hill Book Co., New York (1983).

36. David Cheriton, Gert A. Slavenburg, and Patrick D. Boyle, "Software-Controlled caches in the VMP multiprocessor," *Proc. of the 13th Int'l Symp. on Computer Architecture*, pp. 366-374 IEEE Computer Society, (June 1986).

37. A. S. Tanenbaum and S. J. Mullender, "An overview of the Amoeba Distributed Operating System," *Operating System Review* 13(3) pp. 51-64 (July 1981).

38. A. N. Habermann, Lawrence Flon, and Lee Cooprider, "Modularization and Hierarchy in a family of Operating Systems," *Commun. of the ACM* 19(5) pp. 266-272 (May 1976).

39. C. A. R. Hoare, "Monitors: An Operating System Structuring Concept," *Commun. of the ACM* 17(10) pp. 549-557 (October 1974).

40. David L. Parnas, "A technique for software module specification with examples," *Commun. of the ACM* 15(5) pp. 330-336 (May 1972).

41. Per Brinch Hansen, "Distributed Processes: A concurrent programming concept," *Commun. of the ACM* 21(11) pp. 934-941 (November 1978).

42. B. W. Lamport and R. F. Sproull, "An Open operating system for a single user machine," *Proc. of the Seventh Symposium on Operating Systems Principles*, pp. 98-105 (December 1979).

43. David D. Redell, Yogen K. Dalal, Thomas R. Horsley, Hugh C. Lauer, William C. Lynch, Paul R. McJones, Hal G. Murray, and Stephen C. Purcell, "Pilot: An operating system for a personal

computer," *Commun. of the ACM* 23(2) pp. 81-92 (February 1980).

44. M. V. Wilkes and R. M. Needham, *The Cambridge CAP computer and its Operating System*, North-Holland, New York (1979).

45. Sequent Computer Systems, *Balance 8000 System Technical Summary*, Portland, OR (December 1984).