Wide-address operating system elements

Hertogs, P.H.F.

Award date:
1993

EINDHOVEN UNIVERSITY OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING
SECTION OF DIGITAL INFORMATION SYSTEMS

Wide-Address Operating System Elements

P.H.F. Hertogs

Master Thesis Report by P.H.F. Hertogs.
Work performed from April 1993 to December 1993.
Coach: Ir. A.G.M. Geurts.
Supervisor: Prof. Ir. M.P.J. Stevens.

Summary

Last years, microprocessors have appeared with increased addressing capabilities. Architecture designers have always tried to find ways in which the addressing capabilities of processors could be increased. Examples are virtual memory, segmentation, and address space identifiers. In conventional operating systems the address space was enlarged by placing processes into separate address spaces. This also increases security and modularity but limits sharing. Sharing of data requires the shared data to be named uniformly by sharing parties. Most simple and efficient naming in supporting sharing is realised by using virtual addresses as identifiers. However, virtual addresses are only unique within a local context when using the multiple address space model (i.e. one address space per process). Therefore virtual addresses cannot be used as unique identifiers for sharing, hence a second unique identifier space has to be created or the address space model has to be discarded.

The goal of this study is to develop a common view of how to use the advantages wide-address architectures offer with regard to addressing, naming, and protection issues in an operating system because these are the issues most likely to be affected first by architectural changes. This so-called wide-address architectures provide for much more than only running larger applications.

With wide-address architectures, new possibilities in address space models become realisable. The most simple address space model in supporting sharing would be one in which all processes and other objects are named uniformly and without extra costs. The implementation of such an address space model is looked at and advantages together with disadvantages are proposed.

It seems that the single virtual address space is the most simple organisation in supporting sharing. All objects reside within one address space and can be named by their unique virtual addresses. This virtual address space can be ported to a distributed environment to make all objects uniformly addressable by their virtual address within a distributed system.

Two extensions of this virtual address space are presented. The first is the single-level store which treats all storage (primary and secondary) uniformly, objects are always identified by their unique virtual address, both in primary and on secondary storage. The second extension is persistence, objects can be made persistent without knowledge of an user. Both extensions increase transparency of a distributed system. Transparency can be seen as one of the distinct characteristics of a distributed operating system.

However, because all objects reside within the same address space, security is endangered. To offer the same amount of security as within the multiple address space model, the protection domain abstraction can be used. Objects are placed into logical groupings that define their accessibility, these groupings can be compared to the address spaces used in the multiple address space model.

To illustrate the single virtual address space concept, a case study is done on two single virtual address space operating systems: the Opal distributed operating system and the MONADS networking system. These systems are the living prove that a single virtual address space is realisable on both conventional wide-address architectures as well as on dedicated hardware.

Finally, some extensions to the case studies are made concerning addressing, naming, and protection.

Summarizing, the single virtual address space is more suited in supporting sharing than conventional address space models because of the ability to use unique virtual addresses as object identifiers. However, to be efficiently implemented, wide-addressing capabilities are required. These wide-addressing capabilities can be offered by wide-address architectures.

Contents

Chapters

Appendices

1 Introduction

Due to rapidly increasing demands of software together with newer hardware design technologies, microprocessors have grown from 8-bit processors in the 1970s to 16- and 32-bit processors in the 1980s and 1990s, respectively. Now we are at the beginning of a new architectural era; namely that of 64-bit and wider architectures. Usually, architectures are called x-bit architectures when the register size is x bits. The virtual and physical addresses do usually not influence naming of a processor. In this report, we will focus on wide-addressing and therefore only wide registers do not suffice, the virtual- and physical addresses have to be wide (compared to conventional architectures) too. We define a wide-address architecture as an architecture that provides for wide virtual addresses. The term wide is being defined as larger than 32 bits which seems to be the most occurring address size in today's architectures. Of course, the physical addressing capabilities have to be sufficient too; mapping a 64-bit virtual address onto an 8-bit architecture is highly inefficient in terms of both address translation and protection.

At the moment there are already some architectures that can be called wide-address architectures according to the definition given above. Appendix A describes three of these so-called wide-address architectures; the MIPS R4000, the Hewlett-Packard PA-RISC, and the DEC Alpha.

The address bits growth (both physical and virtual) of microprocessor series that contain the above mentioned wide-address architectures, together with the Intel processor series, has been depicted in figure 1.1 to illustrate the growth in addressing capabilities over the last 10 years.

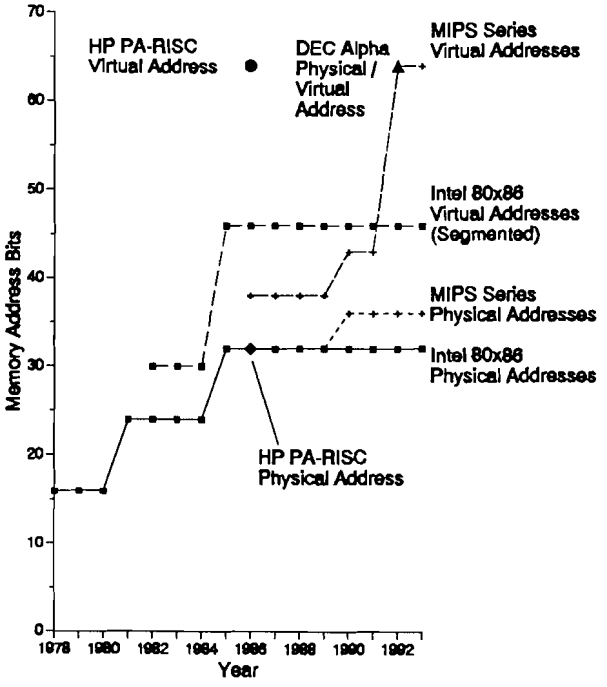


Figure 1.1 Address width growths of popular microprocessor series.

As one can see in figure 1.1, the growth in virtual memory varies from approximately 1 (Intel series) to 4 (MIPS series) bits a year following the growth in physical memory. The limit of 32-bit virtual addresses has already been broken but the 32 bits physical address limit still holds. However, figure 1.1 shows that it will only take a few years before this limit will be broken too (the MIPS series already went beyond the 32 bits physical address limit).
The change from 32-bit to 64-bit (and wider) microprocessors has several consequences; the integer register-, databus-, and the addressbus widths all increase. However, the increase of data registers and -busses does not cause any new viewpoints in operating system design, it is comparable to the shift from 16- to 32-bit processors; the only change being that integer arithmic performance scales upwards. The change that causes operating system designers to reconsider their common approaches is the fact that the address size has increased so much.
However, why would one want such large address spaces? Some examples of applications that tend to consume virtual address space rather quickly are:

- Databases, to speed up operations files are often mapped into memory. This requires enormous amounts of memory.
- Video, e.g. 24-bit color, 1280x1024 pixel screens require 3.75 Megabytes of memory. At a rate of 24 frames a second, 4 Gigabytes (2^{32} bytes) of memory is consumed by only 45 seconds of video.
- CAD, these systems often consist of large networks of servers and desktops. The servers run simulations and manage large databases. This requires for wide-addressing capabilities. For compatibility it is convenient when desktops have equivalent properties.
- Geographic/astronomic information systems, in these systems huge amounts of data are usual.
- Traditional number crunching, researchers in this area have never been satisfied with the amount of memory available.

Besides satisfying this hunger for virtual memory another consequence of the increase in address size is that the addressing model used in many of today's systems, that is, a separate address space for every executing process, can be discarded. The design rationales behind this multiple address space model were: extending the total amount of virtual address space available and enhancing modularity, extensibility, and security. With increased addresses, the virtual address space becomes large enough to encompass all processes in this one and only address space. A major advantage of executing all processes in the same address space is that it greatly enhances sharing. This is a result of objects (for now it is sufficient to define an object as an entity that requires a unique name, e.g. processes, files, communication ports, etc.) now being uniquely addressable by their virtual addresses (e.g. the virtual address of the first byte of an object). Hence, virtual addresses can be passed between domains and domains can address any piece of shared data without risk of name conflicts with other data.
In conventional multiple address space models, pointers crossing address space boundaries had to consist of virtual addresses extended with some sort of address space identifier or had to be mapped onto unique identifiers before they could be used as unique object names. The decrease in modularity, extensibility, and security that results from discarding the multiple address space model has to be accommodated by finding appropriate solutions to this problem, but it seems that this problem is merely an organisational one.
When using virtual addresses as names it should be noted that the virtual address space can provide unique names during an enormous time span. This is being illustrated by the following example. A 64-bit virtual address space in a centralized system (i.e. one processor) can provide unique virtual addresses at a consuming rate of 150 Megabytes/second for 3900 years. Concluding from this example, it should also be possible to map the single virtual address space in a distributed system. Then, in a system consisting of 100 machines, unique virtual addresses out of the same amount of address space could be provided for 39 years at the same consuming rate. Even a distributed system consisting of 1000 or 10,000 machines sharing the same address space becomes possible. The size of the virtual address thus defines, among others, the size of the network to be used because it determines the amount of memory available per node, e.g. a 64-bit virtual address space provides for 1600 Terabytes

of memory per node in a distributed system consisting of 10,000 nodes. One of the advantages this distribution offers is that virtual addresses can serve as uniform unique object names in the entire distributed system. Hence, the advantages of sharing hold equally in a distributed system that is using a single virtual, now global, address space.
This global address space concept has been visually illustrated in figure 1.2.

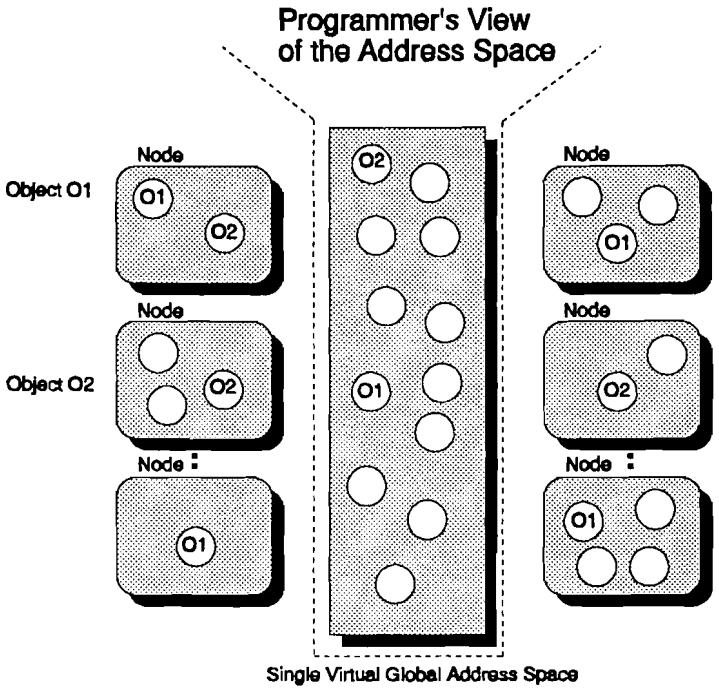


Figure 1.2 One universal address space shared by objects on different locations.

The main goal of this study is to develop a common view on how to construct an operating system using the advantages wide-addressing can offer in distributed systems. Specific topics in this study are addressing, naming, and protection issues of an operating system. Protection can be described as the tool to achieve security. These issues are most likely to change first when architectural changes occur. The lower region changes of an operating system that result from the implementation of a single wide-address space are obvious. First, the partitioning of the address space changes. In conventional systems (i.e. systems that used multiple address spaces), there was one address space per process independently of the nature of the system (i.e. centralized or distributed). When cross address space referencing had to be done, an unique identifier (across address spaces) had to be used. When cross address space referencing among nodes was needed, the unique address space identifier had to be extended even further, now with a unique node identifier.
In a single address space system, the address space can be partitioned among nodes and virtual addresses can serve as unique (inter node) identifiers. Thus the address space seen from the viewpoint of an application designer has changed from a local view to a global view (i.e. a view of the entire address space as has been illustrated in figure 1.2). However, this does not automatically imply that all entities in the system are aware nor need to be aware of the location of objects.
Second, names can change. In multiple address space models, virtual addresses can be used as object identifiers. However as was explained before, they cannot be used to identify across address space boundaries. Therefore an unique address space identifier is needed. This results in multiple name formats being used (one inside of address spaces and one across address spaces). Another limitation of non-uniqueness of virtual addresses is the fact that convenient storing mechanisms like single-level

stores are harder to implement. A single-level store maps all objects into the virtual address space, also objects that reside on secondary storage devices. This increases uniformity of data but requires again object identifiers for primary and secondary storage to be unique. We use the term primary storage to denote short-term storage, e.g. main memory or cache, and the term secondary storage to denote long-term storage, e.g. disks or optical storage media. As was stated before, unique virtual addresses used as names can solve these problems. However, there are also some disadvantages when using virtual addresses. For example, relocating objects is no longer trivial because the name of an object, i.e. its virtual address, changes when an object is being relocated.
Third, as a result of placing all objects in a single address space, security is endangered. What is needed is a secure protection mechanism that protects (groups of) objects from being accessed by other entities in the system. A concept for describing such a protection mechanism is the protection domain abstraction. Protection domains can be seen as regions in the address space that encompass certain objects accessible by other objects with certain access rights. Of course, proper hardware support is needed to implement this protection domain abstraction. In this way objects that reside in memory can be protected. However, objects might be moved between nodes or between primary and secondary storage which makes them vulnerable to attacks. To protect them against these forms of attack, encryption could be used to achieve communication security. Encryption can be described as a special computation using so-called keys on messages that converts them into a representation which has no meaning for any entities other than the designated receiver. Encryption can also be used to implement authentication. Authentication is the process of communicating entities achieving mutual trust which is a necessary condition for secure communication. Therefore, the entities have to prove their identity to one another before communication can begin.
The organisation of this report according to the goal stated above now is as follows. Chapter 2, distributed systems, describes some general components of a distributed operating system. First, memory management techniques and consequences of large virtual memories on them are discussed. This discussion is applicable on both distributed and centralized systems. Second, naming is discussed. Classification schemes of names are given followed by a discussion of unique system names in which will be pointed out that virtual addresses share many characteristics with these unique system names. After system names have been looked at, the naming facility in its general form is being described. Functions a naming facility should provide for are: communication between name servers and clients and between name servers themselves, database management of names, and name management. Third, security is discussed. Issues in security are resource protection, communication security, and authentication.
Chapter 3, wide-addressing, contains a discussion of the advantages as well as disadvantages of the single virtual address space. However, this discussion is preceded by a description of distributed shared memory, one of the basic mechanisms on which the single virtual global address space is based. Distributed shared memory (DSM) is a mechanism based on the message passing inter-process communication system that gives users the illusion of a physically shared memory but overcomes many of the problems related to closely coupled shared memory systems. Finally two extensions to the single virtual global address space are presented. The first extension is a single-level store while the second will be persistence. In a persistent system all objects either transient or persistent are treated in a uniform, user transparent, manner.
Chapter 4, two case studies, describes two operating systems that are based on the single virtual global address space concept. The operating systems described are the Opal system and the MONADS system. The Opal system is a distributed operating system that resulted from an investigation into the effect of wide-address architectures on the structure of operating systems and applications. The goal of this investigation is to determine how software can best exploit the large virtual address spaces of these architectures. Research is being done at Chicago State University (USA). The MONADS system was originally established with the main aim of investigating improved techniques for designing and developing large software systems. While originally being designed as a centralized system, the MONADS system has been extended to run as a distributed system. Research on this project originated

from Monash University (Clayton, Australia) and is now being done at several other places.
Issues described for both systems are: (network) architecture, virtual storage, protection (security), and object support. Finally, chapter 4 contains a discussion of both systems. The reason for choosing these two operating systems is that they both cover wide-addressing yet differ significantly in implementation. Besides this, both of them have been described in literature rather extensively.
Chapter 5, wide-addressing concepts, contains some desired distributed operating system requirements and the consequences of these requirements on addressing, naming, and protection when using wide-address architectures to build a distributed operating system. Also, some extensions to the case studies discussed in chapter 4 will be presented.
Finally, chapter 6 contains conclusions to the main goal of this study, a common view in constructing operating systems using the advantages wide-addressing can offer in distributed systems, in particular in relation to addressing, naming, and protection.

2 Distributed Systems

To start a discussion about distributed systems, first we will take a look at centralized systems. Centralized systems are defined as systems in which one processor is attached to some memory and possibly a disk. Multiple processes use the same processor power by sharing the main processor.
In this report we will focus completely on distributed systems because wide-address machines' advantages are optimally exploited in distributed systems. The address space can be ported to a distributed system as was discussed in chapter 1. Distributed systems evolve from centralized systems by coupling these centralized systems to a network. This coupling can be weak (wide area networks, large geographic span) or strong (multiprocessor systems, connected via a common bus). In between are the so-called local area networks for which characteristics are: connected systems are at short distances from each other, transmission media and interfaces are inexpensive, transmission rates are very high, and sharing of resources is natural and achievable because computers connected by LAN usually belong to the same organisation.
When multiple computers are connected, each individual computer can be thought of as a node. A node consists of one or more processing units, local memory connected to these unit(s) and possibly a disk connected.
There are several reasons why people would want distributed systems, some of them are:

- People want to communicate with each other and share information.
- Resources which are expensive can be shared. By sharing the resources they become cheaper.
- Availability is increased over centralized systems because data may be replicated and all resources that can fail can have built-in redundancy. For the same reason reliability is increased.
- Distributed systems are capable of incremental growth.

There are also some disadvantages to a distributed approach:

- Security may dictate that some data cannot be transferred in whole or in part to another machine. Therefore a user may have to use a remote processor to access certain data.
- Physical distribution of resources may not match the distribution of the demands for services. Therefore some resources may be idle while others are overloaded.

The first distributed systems were networking operating systems. Therefore networking operating systems can be seen as the predecessors of distributed operating systems. There are some characteristics that separate networking operating systems from distributed operating systems. These are:

- Each node in a networking operating system has its own operating system instead of running a part of the global, system-wide operating system.
- Each user has its own computer. The operating system does not dynamically allocate processes to CPUs.
- File management is not transparent to the user. Each user has knowledge of the place where his/her files are stored.
- The system has little or no fault tolerance. If computer A crashes, all users of A are unable to continue their jobs.

In figure 2.1 the environment of a networking operating system is depicted.

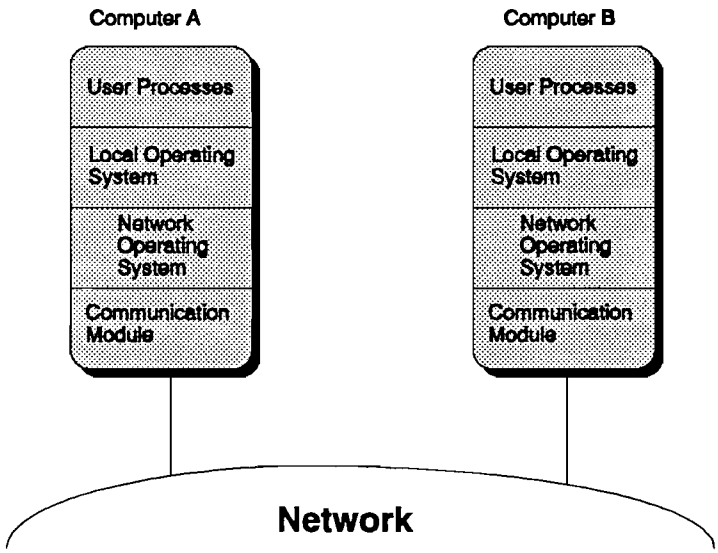


Figure 2.1 Environment of a networking operating system.

An exact definition of a distributed operating system is hard to give but there are some properties when satisfied indicate that, probably, the proposed system can be classified as a distributed operating system. When one of these properties is not satisfied it certainly is not a distributed operating system. These properties are:

- Multiple processing elements that can run independently.
- Interconnection hardware for communication and synchronization between processes running in parallel on different nodes.
- Processing elements fail independently, this results in some fault tolerance.
- A shared state has to be kept by all nodes to be able to recover from failures.

A distributed operating system can be thought of as a system that looks to the user as an ordinary centralized system, but runs on multiple independent CPUs.
The general structure of a distributed operating system is given in figure 2.2.

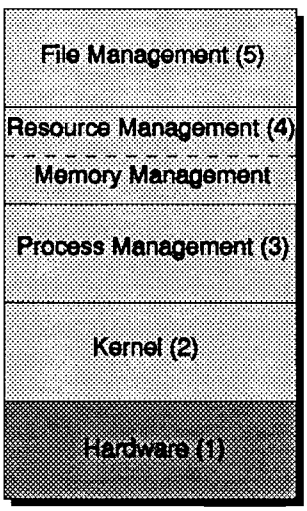


Figure 2.2 The building blocks of a distributed operating system.

We will now look at the individual building blocks in greater detail. Inter-process communication (IPC) is one of the most fundamental facilities of a distributed operating systems and is strongly dependent on efficient IPC primitives and a suitable transport protocol offering services to these primitives. A protocol is defined as a formal set of rules and conventions that define the format and relative timing of message exchange between two or more communicating processes. The hardware of a distributed system (1, see figure 2.2) defines the performance limits of IPC primitives and the transport protocol to be used.

If no shared memory is available, processes communicate by sending messages. Two widely used forms of remote IPC to develop IPC primitives are:

1. Message passing, this is a logical extension of inter-process communication in centralized systems. Client and server have to be synchronized. A server can be defined as a subsystem that provides a particular kind of service (a software entity running on one or more machines) to unknown clients. The server is the service software running on a single machine.
2. Remote procedure call (RPC), this is a type-checked mechanism which automatically turns a language-level call on one machine into a language-level call on another machine. To the user, the type of call (local or remote) is transparent.

However, if shared memory is available it can be used to support IPC. Distributed shared memory, which can be used as an IPC mechanism, will be discussed in paragraph 3.1.

The kernel (2) acts as an interface between management components of an operating system and the hardware of a computer system. A kernel is the part of an operating system which allows each processor to operate. It is also responsible for communication among connected computers, therefore the kernel is replicated in all computers of a distributed operating system. A major decision issue in the development of distributed operating systems is the amount of functionality encompassed by the kernel. Large kernels offer more services and do so more efficiently but reduce the overall flexibility and configurability of the resulting operating system. In the case of small kernels, the majority of services are put into management components, in particular into separate servers. Each server can be programmed separately and can be replaced easily. The small kernel requires less programming, thus simplicity is a major factor. The problem is that by using servers, the servers themselves have to use some form of inter-process communication to communicate between each other. This problem is of major concern for small-kernel operating systems because in those systems much functionality has moved towards the servers. Most recently developed operating systems have been built around kernels containing only minimum functionality (e.g. IPC primitives) assisted by servers implementing higher-level functionality (e.g. file management). These operating systems are called micro-kernel or kernelized operating systems; examples of kernelized operating systems are Mach [Jone86] and Chorus [Abro89]. In this report we will assume that the operating system is kernelized because of the increased flexibility kernelized operating systems offer.

In centralized systems, the kernel can be seen as a secure part of the operating system. In distributed systems, however, this is no longer true. Kernels are potential subjects to malicious modification. A malicious user could load a modified kernel and all kinds of secrets could be stolen. So kernels in distributed systems cannot be trusted to store important system information like protection data unless extra security measures are taken. This will be explained in paragraph 2.3.1.2 and these extra security measures have been implemented in the MONADS operating system which is discussed in paragraph 4.2.

Together with the IPC facility and the memory management system (4), the process management system (3) offers the execution environment for processes. A main task of process management is to provide policies and mechanisms for sharing processing resources spread around the network among all processes.

Resource management (4) controls the use of resources and supports services of a distributed operating system. It is complicated task because of the physical distribution of resources, communication delay, redundancy, and the lack of complete global state information. Memory management can be seen as a part of resource management. The goal of this component is the management of available memory

resources, that is, keeping the memory loaded with that data from external memory which is most likely to be accessed in the next period of time.
File management (5) has as goal the translation of files from the logical to the physical level and the manipulation of files at the physical level. Also, a single logical file system has to be created from a collection of files physically distributed over the network, while hiding this distribution to the user.
Most operating systems today can be placed in one of two major categories: process-based systems and object-based systems. Logically, both categories are equivalent, but they differ at the level of mechanisms. A process can be seen as the activity resulting from executing a program with its data by a processor. The process itself can be divided further into one or more so-called threads, which are pieces of executable code having access to data and the stack of a process. A process-based operating system can be seen as a collection of interacting, concurrently running processes on one or more processors. A process is one type of object; other possible object types may include procedures, communication ports, hardware devices, etc. An object-based system treats parts of the system as objects. An attractive characteristic of these systems is that of abstraction; each object itself embodies some form of abstraction, it hides the internal implementation of that concept (data and code are encapsulated in the object) and provides a set of operations (methods) applicable to the object. This higher level of abstraction is necessary to develop a standard view on the interacting components of an operating system. For example, without this abstraction it would be nearly impossible to develop a common protection mechanism for all entities (e.g. files, processes) in a system. Therefore, in this report we will use the object abstraction to describe operating systems.
An object-based system is an object-oriented system only if a class system exists (an object class specifies a number of visible operations, a set of variables and a set of methods that implement the operations, an object is an instance of its class) and inheritance (inheritance of specific properties from one class to a new class, i.e. subclass) is possible. The use of object models to define and support sharing of data and services between applications results in so-called object sharing systems.
In this chapter, two parts of the distributed operating system will be looked at in particular. These parts are naming of objects in the system and security. Especially these two parts have to be reconsidered when implemented on newer architectures with wide addresses.
As we will see, these two parts are heavily intertwined in a distributed operating system. However, first we will take a look at some memory management issues and the way in which they are being affected by wide-addressing.

2.1 Memory Management

In this paragraph some techniques in memory management will be discussed. Some requirements memory management should fulfil are:

- Sharing.
- Logical organisation.
- Physical organisation.

To ease handling of large amounts of memory, memory can be divided into equal fixed-sized chunks. The chunks of a process (pages) can then be assigned to available chunks of memory (frames). Each process is then assigned a page table which maintains the frame location for each page of the process. A logical address now consists of two parts: a page number (n bits) and an offset (m bits). The paging process is depicted in figure 2.3.

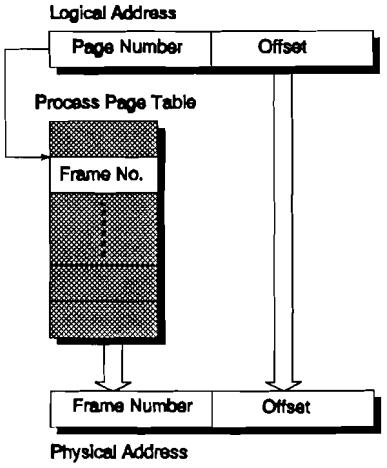


Figure 2.3 Paging.

Address translation now exists of the next three steps:

1. Extract the page number as the leftmost n bits of the logical address.
2. Use the page number as an index into the process page table to find the frame number k.
3. The starting physical address of the frame is k x 2^m, and the starting address of the referenced byte is that number plus the offset.

The size of a page depends on several factors:

- Internal fragmentation increases as the page size is increased. A smaller page size however, means more pages per process and thus larger page tables.
- Larger pages are transported more efficiently from/to secondary storage.
- Larger pages will cause more isolated code parts in the pages and therefore small pages with few isolated code parts will most likely reduce the number of page faults.

Another technique for organizing programs and data is segmentation. Segments vary in length although there is a maximum segment length defined by the number of offset bits. Segmentation is, in contrast with paging, usually visible to the programmer. The process of segmentation is depicted in figure 2.4 where the segment number exist of n bits.

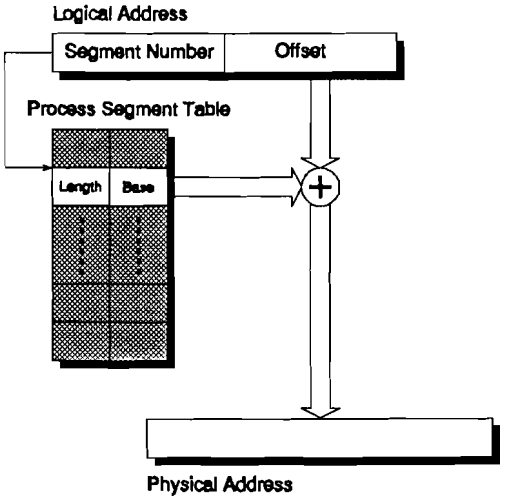


Figure 2.4 Segmentation.

Address translation now exists of the next four steps:

1. Extract the segment number as the leftmost n bits of the logical address.
2. Use the segment number as an index into the process segment table to find the starting physical address of the segment.
3. Compare the offset to the length of the segment. If the offset is larger than the length of the segment, the address is invalid.
4. The desired physical address is the sum of the starting physical address of the segment and the offset.

With paging and/or segmentation (both may be combined) it is not necessary that all pages and/or segments of a process are in main memory during execution. Advantages of this fact are:

- More processes may be maintained in main memory.
- It is possible for a process to be larger than the total size of main memory.

To allow for effective multiprogramming and relieve the user from unnecessarily tight constraints of main memory, virtual memory (for a discussion of primitive virtual memory operations see [Appe91]) can be introduced by using paging and/or segmentation. The main memory serves as a sort of cache for the complete virtual memory. When an address is referenced which is not in main memory, the appropriate page/segment needs to be brought in from disk (this process is handled by the so-called page fault handler). Replacement of pages in main memory is according to the used replacement strategy. The page table structure to translate virtual memory addresses into physical memory addresses has been depicted in figure 2.5.

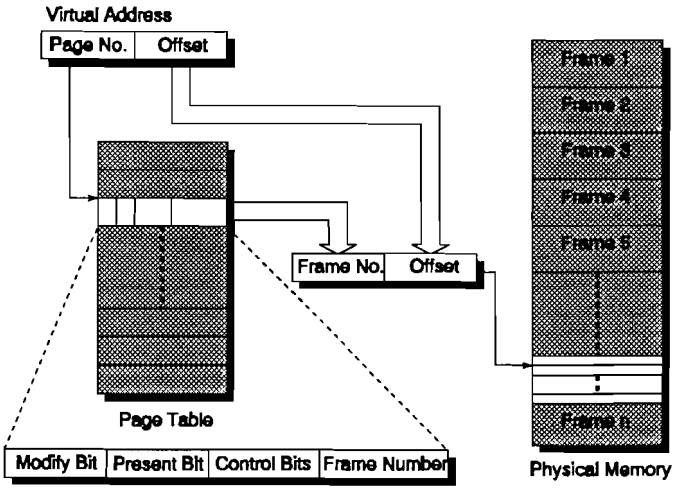


Figure 2.5 Paged virtual memory.

To increase performance, a translation lookaside buffer (TLB) can be used. A TLB is a special cache for page table entries so that not every virtual memory reference requires a complete virtual address translation. Each entry of the TLB must contain the page number and the complete page table entry because not all of the page table entries are present (the order of entries is dependent on the replacement strategy of the TLB). Thus, associative mapping (in associative memory cells are referenced by a key, in this case the page number, rather than an address) is used to see if the required page number is present in the TLB.

In many older architectures, page table handling was done in microcode depending on the particular page table structure. In newer architectures (e.g. MIPS R4000, appendix A) page table handling has moved towards the operating system, hence page table structures can be freely chosen by the operating system designer. This is necessary when unconventional addressing techniques are used (wide-addressing) as is described in chapter 3.

A disadvantage of the page table approach is the fact that page tables can grow very large in systems were virtual memory is large. The large size of conventional page tables results from the fact that one entry is required for each page in virtual memory. A large virtual memory is inherent to wide-address architectures so the normal page table approach is hardly applicable. A disadvantage of large page tables is the fact that the page table has to be placed in virtual memory itself. This adds complexity to both the page fault handler and the address translation buffer.
A widely used solution to solve large page tables is the inverted page table approach which is found in many operating systems with a large virtual memory. In this scheme which is depicted in figure 2.6 there is one entry in the page table per real page.

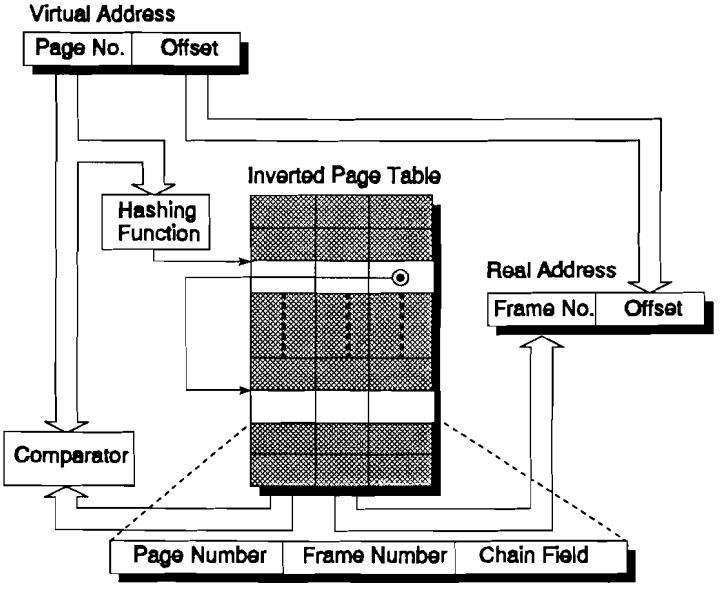


Figure 2.6 Inverted page table.

The length of the inverted page table increases linearly with the size of main memory (depending on the amount of overflow allowed in the table, for a discussion of hashing see [Aho87]) while the width of the table increases only logarithmically with the size of virtual memory.
When a virtual address needs to be translated, the page number acts as a hashing key. The hash value is created from the hashing key by the hash generator on every address translation and must thus be a simple function such as an exclusive or operation on some specified bits of the page number (key). The hash value now points to an entry in the page table. The page number in this entry is compared to the original page number and if a match occurs, the corresponding frame number is used to form the real address. If there is a mismatch, the chain field is used to follow a chain of synonyms (multiple virtual pages mapped onto the same real frame) until the correct real frame or an end of chain status bit is found. If no real frame number is found, a page fault can be generated to update the inverted page table.

2.2 Naming

Names can serve several purposes in distributed operating systems. One is to facilitate sharing, if various users want to address/access an object, they have to have a unique name for that object. An object is defined as any entity in the system that deserves a name to make it distinct from other entities (e.g. files, processes). Associated with each object is a server which manages the object.
The difference in naming between centralized and distributed systems arises from the fact that in distributed systems there is no single place that contains global information about all instances of all objects (a place that contains all names). This requires several nodes to cooperate and form a global name space for the system.
In the last decade, the design of name systems has grown from a single centrally controlled server providing only host name to physical address mapping, to a complex system consisting of multiple and distributed servers, providing not only name mapping, but also general directory lookup services. In this paragraph we will look at several issues in naming concerning distributed systems.

2.2.1 Classifying Names

A name (identifier) is defined as a string of symbols (bits or characters) which identifies an object. From the computer point of view, digital identifiers (binary numbers or also called system names) are most suited but human users cannot cope efficiently with these numbers. Therefore, another type of identifiers is introduced, called user names. These higher level names are also used to provide for multiple copies, and to support object relocation.
Names are always associated with contexts. A context is the environment (i.e. some sort of state information) in which a name is valid. A context can be explicit or implicit. A context is explicit when it is visible in the name itself and implicit when it is not visible in the name itself.
Names can be classified according to several schemes [Yeo93], some of them are:

- Ambiguous/unambiguous, unambiguous names are names that refer to at most one object. This means that the same name cannot be used to refer to different objects. Of course, the definition of ambiguity depends on a context. A name can be unambiguous in one context while ambiguous in another.
- Unique/non-unique, a name is unique if it represents the only name for its referent. Several non-unique names can be used to identify the same object. One name is then treated as the preferred name, while the others are called aliases or nicknames. The definition of uniqueness also depends on the context of a name.
- Global (absolute)/non-global (relative), if a name is global it is interpreted in a consistent manner by all clients and all services, regardless of their location in the distributed system. A name is relative when it is interpreted according to some context.
- Pure/other (impure), a pure name is only a bit pattern serving as an identifier. It contains no extra information in its structure. An advantage of pure names is that they are short.

Names can be distinguished on the basis of their structure as primite/flat, partitioned, or descriptive names. The set of names complying with a given naming convention is called the name space or name domain. A primite/flat name is equivalent to a pure name which is only a bit pattern unambiguously identifying a particular object.
A partitioned name is commonly structured as a series of primitive names. Explicit contexts can be found in parts of partitioned names.
A descriptive name is list of attributes which are true for exactly one object. A partitioned name is in fact a descriptive name with a predefined structure.
Time can also be used to differentiate names. Static names are names which permanently denote the same object even after relocation. These names are independent of the communication facility.

Dynamic names are assigned to objects only for a limited period of time, short compared to the life-time of an object.
Finally names can be differentiated as group or individual names. Group names denote a set of objects while individual names denote one single entity.

2.2.2 Unique System Names

Unique computer-oriented identifiers have been widely used in file-systems. These identifiers are guaranteed to be unique in both space and time. As we will see in next chapters of this report, they can also be used as general object identifiers. This especially holds for wide-address machines in which virtual addresses can almost serve as unique system names; objects can be named by their virtual address if virtual addresses are made unique.
Advantages of unstructured unique system names are:

- The name is independent of the location of the corresponding object.
- The names are absolute, so they do not have to be changed when objects are moved.
- All objects are named uniformly.
- Objects which refer to other objects (composite objects) can be constructed.
- They are short, so they can be easily stored, passed, and hashed.

According to the definitions given in paragraph 2.2.1, unique virtual addresses are global names because they are interpreted the same by all entities in the distributed system regardless of the location of those entities. Virtual addresses are also unambiguous names because a unique virtual address denotes only one object at a time. Whether virtual addresses are pure names or not depends on the internal partitioning of virtual addresses (e.g. some bits of the virtual address could be used to identify the node on which it resides which makes the virtual address impure). A disadvantage of using a virtual address as a name for an object is that the name becomes dependent on the location of the object, thus making relocation of objects harder to implement. Let alone that, the last three advantages of unstructured unique system names hold equally for unique virtual addresses.
Unfortunately, there are also some problems associated with unique system names:

- Unique identifiers have to be created. There are three well known strategies to attack this problem:
  1. Hierarchical concatenation, a unique identifier is created for each identification domain, usually a host computer, then the identifier used within this domain is concatenated to the identification identifier. Disadvantages are explicit visibility of the computer boundaries and the fact that with heterogeneous domains, identifiers greatly differ in both length and form. In case of virtual addresses, the address could consist of a node number concatenated with a local part. In this way the virtual address become impure because it now contains location information in its structure.
  2. A standard, uniform global identifier form and space is developed for all resources. The local identifiers are then bound to these global identifiers either temporary or permanent. The created level of indirection offers some advantages; the local identifier can be chosen to optimally suit the language or operating system environment, several local identifiers can be mapped onto the same global identifier or vice versa, and the global identifier can be reassigned if objects are relocated. When the virtual address space is taken to be the uniform global identifier space, virtual addresses have to be unique and thus no local identifier is necessary. Also, with unique virtual addresses there is no indirection.
  3. Concatenation, the node identification of the generating node is concatenated with a reading from its real time clock.
- Locating the object given the identifier is a complicated task. Binding is defined as the identification of the location of the managing server and of the path between object and server in order to locate an object. A specific server can be passed a message if one knows the network

address of that server. This network address can be found either by sending a request to a name server or by sending a broadcast message across the network. This problem holds equally when pure virtual addresses are used as identifiers.

- Handling of different versions of an object. The problem is how to create a consistent view of an object, that has several versions, towards the user. This problem could be solved by keeping a mapping of user names onto version names of the object. The user name then acts as an alias for the version name of an object. This mapping can be maintained either by the user or by the object server.
- Replication of objects. Objects can be mutable or immutable. Immutable objects do not create any problems with respect to naming. All replicas use the same name so there is only one logical object. Mutable objects have to be kept consistent which is a very complicated task. Users should not be aware of replication because replicas can be in different states. The naming facility should map the unique system names onto the correct replica.
- Lost objects. The problem of lost objects is that they still claim physical space but are not addressable any more, i.e. the object exists but the reference for that object has been removed. The problem of lost objects could be attacked by never re-using a virtual address to name another object.

2.2.3 Naming Facility

Components in a naming facility can be classified according to structure and function. The structure implies building a naming facility out of active entities called name servers which are instances of the name service. They can be accessed directly or through name agents so that clients can name objects and share information about these objects. The distribution of the naming facility can be classified into two models:

- Centralized model, a logical centralized server provides naming data for the whole name space.
- Distributed model, several name servers collectively manage the name space.

The name server locations should be transparent to users. This transparency can be achieved by name agents that act between clients and name servers. Name agents can be shared between clients when acting as a separate process or part of the kernel or every client can have its own agent as a set of subroutines which are linked into the clients program. In chapter 4, an operating system is looked at which uses name agents to provide the necessary transparency to the user resolving names (see paragraph 4.1).

When components of a naming facility are classified according to their functionality, the following functional categories can be distinguished:

- Communication, between name servers and name agents/clients, and between name servers themselves.
- Database management, a name service database is necessary to store and manage objects' attributes. This database has to be distributed among the name servers to save space.
- Name management, two major tasks of name management influenced by the name structure are:
  1. Name distribution, this is the assignment of authority for managing parts of the name space to name servers. Naming authorities or authoritative name servers for an object are name servers that store and manage data for that particular object. There can be more than one authoritative name server per object to increase reliability but this creates a consistency problem.
  2. Name resolution, locating the authoritative name servers for an object given its name. There are two groups of name resolution mechanisms:
     1. Structure oriented name resolution, which locates the set of authoritative name servers based on the structure of names, e.g. when the unique virtual address is partitioned (i.e.

impure), structure oriented name resolution is used.

2. Structure free name resolution. This is the ideal solution because it permits maximum flexibility in the administrative assignment of authority and simplifies name management.

Names can be cooperatively resolved by servers and agents in three different ways: recursively, iteratively, and transitively. These three ways of resolution are depicted in figure 2.7.

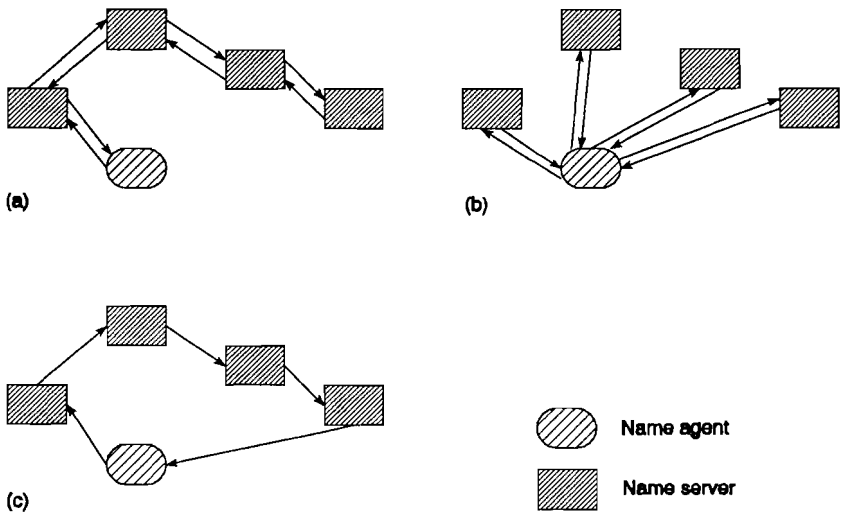


Figure 2.7 Name resolution styles. (a) Recursive, (b) iterative, and (c) transitive.

In the recursive implementation, the name agent accepts a request and transfers control to the first name server who is responsible for performing the name service operation. The resolution activity migrates from server to server until the authoritative server is found. In this implementation, the name agent has little work to do but the name servers could be heavily loaded by processing several requests at the same time. Because a name server could be forced to pass the process of resolution to another server it should be able to start another job when waiting for a response. Because of this call response model the recursive approach is efficiently supported by RPC based communication systems.
In a system based on the iterative solution, the name agent calls every name server while retaining control over the process. A name server tries to resolve the request and returns the answer or resolution state including the server that should be tried next. Name agents have a lot of work to do and the servers do not communicate between each other. As with the recursive implementation RPC communication systems can efficiently support the iterative implementation.
In a system with transitive resolution, a server which is not able to continue the resolution process passes activity to the next name server. The name server which successfully completes the operation returns the result to the name agent. This implementation has the fewest messages to be sent. However, this implementation should be used in systems with reliable communication because the sender does not receive any acknowledgement messages once operation is passed on.
Concluding, the choice of a particular name resolution mechanism should be based on:

- The relative processing powers of name servers and name agents.
- The semantics of the communication protocols used.
- The structure of used names. A partitioned name can influence the decision of the name agent on contacting a particular name server.

2.3 Security

Protection is being defined as the control on access to resources, that is, controlling who can when perform which operations on what objects. Protection is a technical term which encompasses certain aspects of security. Security is a goal; protection is a set of mechanisms to achieve parts of that goal. Protection mechanisms must deal with:

- Secrecy, users must be able to keep data secret.
- Privacy, users must be guaranteed that information they give away is used for the purpose it was intended for.
- Authenticy, if data is to come from a source X, the user must be able to verify that it was indeed X who sent the data. Also, if the user sends data, he/she has to prove his/her identity.
- Integrity, neither the system nor an unauthorized user should be able to corrupt any data.

Intruders, persons or programs trying to obtain illegitimate access to data or resources, can use a variety of methods to obtain unauthorized access. These methods can be classified as:

- Leaking, an intruder may have an accomplice in the form of an legitimate user who leaks information to the intruder. The problem of making sure that information cannot leak into the outside world is called the confinement problem.
- Browsing, an intruder attempts to read all files in the system, read all packets passing by on the network, read other processes' memory and so on without modifying the data.
- Inferencing, an intruder tries to steal information by reading stored data, listening to data in transfer and so on, both methods described above are used in combination.
- Masquerading, an intruder or program may masquerade as an authorized user or program in order to gain access to system objects. A masquerading program is called a Trojan horse.

In distributed systems there are three major aspects to security:

1. Resource protection, objects in the system have to be protected against unauthorized access. In operating systems based on wide-address architectures, resource protection has to be carefully implemented because of the specific addressing model used which results in all objects being accessible to all subjects. This is in contrast with common operating systems in which each process executes in its own address space, hence processes are separated from each other.
2. Communication security, protection of information sent on communication media. All protection information sent on communication lines in a distributed network has to be protected.
3. Authentication, assuring the user that the message came from the reputed source and is unmodified.

These aspects will be discussed in next paragraphs.

2.3.1 Resource Protection

Resource protection can be implemented based on several access control models:

- Access matrix model, the access rights of each user are defined as entries in a matrix. Access control is through classification of users and information. Only the rights of users to access objects are checked. A disadvantage of the access matrix model is vulnerability to Trojan Horse attacks (i.e. a malicious entity enters the system pretending to be someone else) and in general, the confinement problem is undecidable; it is impossible to determine whether unauthorized information flow exists.
- Information flow model, all information transfer is subject to the flow relation among security classes. The focus in this model is on flow of information from one object to another rather than on individual accesses to objects. The advantages of this model are added flexibility in analysing flow over storage channels and the ability to analyze flow at a fine-grained level. Thus the confinement problem is decidable and the system can stand attacks by Trojan horses.

- Security kernel model, a small part of the system is responsible for security in the system. This part should monitor all accesses, should have to be correct, and should have to be isolated from the rest of the system. Problems with this model are performance degradation because of an extra level of interpretation beneath the operating system and specialized hardware requirements.

We will now focus on the access matrix model because of its widely spread usage in distributed systems. The access matrix model consists of three basic components:

1. Passive objects or resources that have to be protected. These are associated to some type (e.g. file, process) which determines the set of allowable operations to be performed on them.
2. Active subjects, individuals wishing to access objects and perform operations on the objects. Subjects may be users or processes. Subjects are also objects because they have to be protected too. A subject can be defined as a pair (process, domain) where a domain is the protection environment in which a process is executing. During execution, a process can change protection domains.
3. A set of access rules that define the manipulation of objects by subjects. Associated with a (subject, object) pair is a certain access right. In this context, a domain can be seen as the set of objects that are accessible by a subject at a specific moment.

The general form of the access matrix (AM) is presented in figure 2.8.

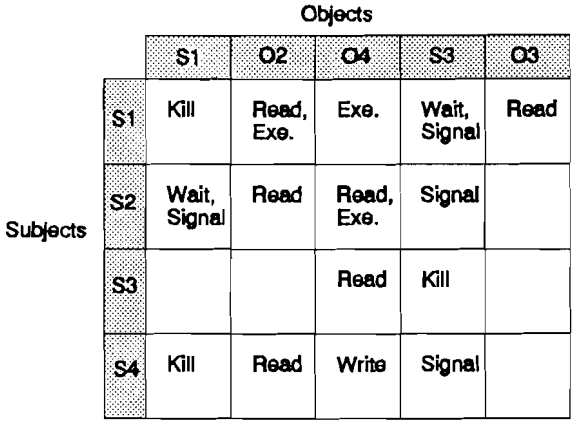


Figure 2.8: Access matrix.

Each type of object has associated with it an object monitor. Based on the access matrix, each access to an object is validated in the following way:

- A subject S wants to perform an operation Op on object O.
- The triple (S, Op, O) is formed by the system and passed to the object monitor of O.
- The object monitor looks for the attribute Op in AM [S, O]. If present, access is permitted and the operation Op is allowed to proceed; otherwise, a protection violation occurs.

The correctness of this model is based on correctness of the object monitors assuming that the identification of subjects is unforgeable.

The association between subject and domain may be static or dynamic. The static relationship is simpler to implement than the dynamic relationship. The dynamic relationship should provide for domain switching as well as changing the contents of a domain.

In a typical access matrix, most matrix elements will be empty sets or have the same value. Since there are typically hundreds of domains and tens of thousands of objects, storing the complete access matrix is not very efficient. Therefore the information in the access matrix is being split up. This can be done in two ways, by rows or by columns. The first way results in per domain, a list of objects and the operations allowed on each object. This is called a capability list. The second way results in, per

object, a list of domains and the set of operations allowed on the object for each domain. This is a so-called access control list. We will now look at both possibilities in greater detail.

2.3.1.1 Access Control Lists

The access control list evolves from the access matrix by decomposing it by columns. The concept of implementation is identical for centralized and distributed operating systems:

- Each column in the access matrix, empty places are discarded, is implemented as an access list for an object.
- This list consists of pairs <domain, right> which define all domains with a non-empty set of access rights for the specific object. Objects can thus be shared with possibly different rights.

In practical systems it is often sufficient to classify domains and assign sets of rights to each class of domains. A few classes often suffice, for example the Unix system has three domain classes; owner, group, and others. The access lists are usually maintained by object monitors (servers managing the objects). The access to objects can now be visualized as is done in figure 2.9.

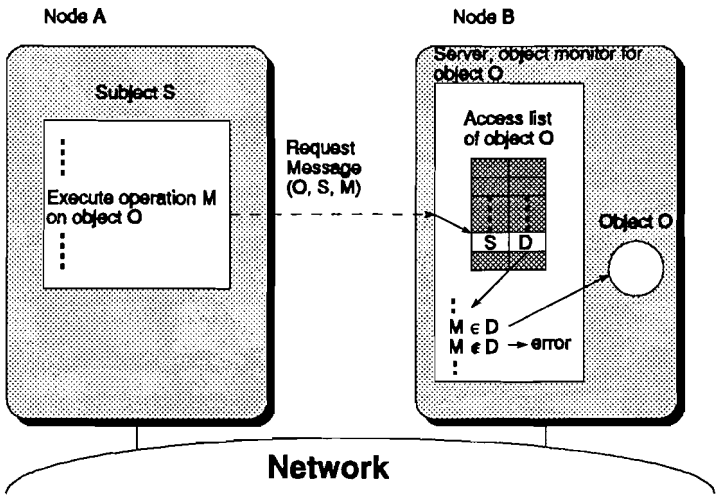


Figure 2.9 Access control list operation.

Advantages of the access control list approach are:

- For a given object it is possible to both simply as well as efficiently determine which subjects have access rights for it.
- Revocation of privileges is inexpensive, this can be done by deleting the appropriate entry in the list.
- The operating system does not require any special form of modification, the implementation of access control lists is straightforward.

However, disadvantages are:

- Given a subject it is difficult to determine its access rights on different objects.
- Because of checking the access list on every access, considerable overhead is created when access list are long.

2.3.1.2 Capabilities

Capabilities are the result of decomposing the access matrix by rows and looking at individual cells in such a row. A capability is defined as an unforgeable ticket which specifies one or more permission modes and allows its holder to access the specified object. A capability combines a reference to an object with the access permission; naming and protection are combined in one structure. Capability lists can be kept by the kernel but often the capabilities are kept by processes themselves. Capabilities have the following properties:

- Possession of a capability gives a client the authority to access the relevant object.
- Objects can be shared, possibly with different access rights.
- Capabilities must be unforgeable and be protected from disclosure or else the system is not useful. Forgeability of capabilities would result in subjects changing their own access rights which would make the system unsafe.

The advantages of using capabilities are:

- For a given subject it is easy to determine the set of objects that can be accessed together with the access rights.
- Passing of access rights does not require involvement of a server as with the access control list implementation.
- Capabilities are the logical protection mechanism in object-based data models.

Disadvantages of capabilities are:

- Given an object it is difficult to determine which subjects can access it and what access rights they have.
- Revocation of capabilities is a complicated task. Revocation is defined as the process of revoking access rights to objects that are shared by a number of different users. In general, it is not easy to implement revoking in capability based systems. However, there are a number of schemes for revoking:
  - Reacquisition, capabilities are deleted from each domain. A process may try to reacquire the capability if it finds that the capability was deleted.
  - Back-pointers, a list of pointers is maintained by the object manager (i.e. object server) for each object, pointing to all capabilities associated with that object. In case of revocation, the back-pointers can be followed so that the capabilities can be deleted or changed.
  - Indirection, capabilities point indirectly to the object via a global object table. Revocation is then implemented by searching this table and deleting or changing the appropriate entries.
  - Keys, associated with each capability is a key which cannot be modified or inspected by the owner of the capability. A master-key is associated with each object and determines the value of the key associated with the capability during the creation of a capability. When an object is accessed, the key associated with the capability is being related to the master-key associated with the object and if this process proves to be successful, access is allowed to proceed. Revocation is implemented by changing the value of the master-key thus invalidating all accesses.
  - Renaming of objects, however this is a very brute form of revocation.

There are three major categories of design and implementation of capabilities:

1. Tagged capabilities, each memory cell is extended by some tag bits that indicate if a capability is contained in that cell or not. This implementation requires special hardware support and is therefore not very attractive.
2. Partitioned capabilities, capabilities are stored in special segments separately from data. These segments or objects can only be accessed by the system. A major disadvantage of this approach is the fact that it relies on correctness of the operating system kernel which cannot be guaranteed without expensive security measures (see paragraph 3.4.2) in distributed operating systems as was explained earlier. Because the kernel checks each capability, these capabilities can have a copy bit which can be used to limit the propagation of such capabilities.

3. Sparse capabilities, in this implementation capabilities are bit strings of a defined length which can not be distinguished from normal data. Therefore, this implementation especially, relies on the unforgeability of the capabilities.

The only implementation of capabilities which could be used in distributed systems without expensive security measures is the third one. Because sparse capabilities are stored in normal memory they have to be protected. A simple form of protection against forgery is the use of passwords or encryption. Encryption is a process of transforming data, usually referred to as plain text, into cipher text in a manner such that the cipher text can only be turned back into plain text by the use of a particular key and an appropriate algorithm. To protect capabilities from being seen and used by unauthorized users additional protection mechanisms are necessary. These mechanisms depend on the type of capability:

- Uncontrolled capabilities, if a subject possesses such a capability, it means that it has right of access. These capabilities can be protected from forgery with passwords or encryption.
- Controlled capabilities (note that the original capability protection mechanism has a fundamental weakness in not verifying the identity of the subject that possesses the capabilities [Levy84], this type of capability overcomes this weakness), capabilities of this type can only be accepted if from a legitimate holder. They can be protected in three different ways:
  1. With access lists that contain subject information.
  2. With encryption using an unique identifier from the holder (e.g. its address).
  3. With capability lists named by the origin address, in this way the server can check who has the specified capability.

How well access information is protected depends on the types of threats each protection scheme is intended to encounter, on the strength of cryptographic information handling procedures, and the encryption key management protocols supporting these schemes. This will be looked at in greater detail in the next paragraph.

2.3.2 Communication Security

Communication security is necessary to protect important information (e.g. capabilities) that has to be sent across a network. Because wide-address operating systems can be distributed systems, information sent between nodes can be vulnerable to attacks. Therefore, protection of this information is needed. The model used to explain some of the techniques used for communication security is the association model as depicted in figure 2.10.

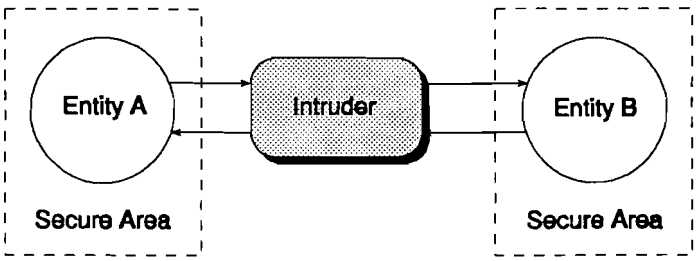


Figure 2.10 Association model.

The connection between two secure areas (e.g. host-host, user-host, process-process or client-server) may be subject to physical attacks. The intruder is represented by a computer located in the data path between both secure areas. All messages thus pass along the intruder.

Threats to network security can be divided into two categories:

1. Passive attacks, the intruder only observes messages. This results in release of message contents

and/or traffic analysis.

2. Active attacks, the intruder performs a variety of processing on messages. This results in message stream modification, denial of message delivery, delay of message delivery, and/or masquerading (the intruder pretends to be someone else).

Countermeasures against described threats require secure identification of communication entities. Two main goals for the design of mechanisms to provide communication security are: prevention of passive attacks and detection of and recovery from active attacks.

Encryption can be used against passive attacks, and in conjunction with protocols also against active attacks. Encryption can be described as a special computation, using a so-called key, on messages that converts them into a representation which has no meaning for any entities other than the designated receiver. Two main classes of encryption are:

1. Conventional or symmetrical encryption, the same key (or two keys of which one can be easily derived from the other) is used to encrypt and decrypt information. Therefore, a secure communication channel is needed to distribute the keys. The messages sent are secure as long as the used key remains secret. An example of this encryption class is DES (Data Encryption Standard).
2. Public-key or asymmetrical encryption, the encryption and decryption keys are different and it is impossible to derive one from the other. Each user has a public key which may be published and a secret key that has to be kept unrevealed. Both keys define a pair of transformations, each of which is the inverse of the other. A major advantage of this encryption method is the fact that no secure channel is needed to transfer keys. However, a disadvantage is the complicated key authentication problem. The sender is unable to verify if the receiver's public key is the right one once an intruder masquerades as the proposed receiver and claims that its key is correct. This can be solved by introducing a trusted third party which acts as a certification instance. This third party is used to authenticate public keys with help of digital signatures or certificates. An example of a cryptosystem belonging to this class is the RSA (Rivest-Shamir-Adleman). A characteristic of this system is that it cannot guarantee to maintain the length of the information encrypted by it.

Limitations of encryption are:

- Most operations performed on data (applications) require that the data be supplied unencrypted. Therefore unencrypted data (cleartext) should be protected.
- Revocation, encryption is based on an encryption key that has to be distributed among senders and receivers. The encryption keys are similar to simple forms of capabilities and act as tickets. The revocation problem as with capabilities also exists with keys. The only solution is to decrypt the data and re-encrypt with a new key, however, this method is not selective.
- Protection against modification, encryption does not prevent encrypted data from being modified. As a solution, check bits and/or error correcting codes could be added to encrypted data to detect and recover from modification.
- Key storage, -distribution, and -management are complicated tasks.

2.3.3 Authentication

Authentication was defined as the process of assuring the receiver that a message came from the reputed source and is unmodified. Of course, this also has to hold the other way around. This is called identity authentication. Two other types of authentication are: message content authentication and message origin authentication. These types are in fact extensions of the general identity authentication and will not be discussed in this report. Extensions to methods for user identification in centralized systems cannot be used because such methods rely on personal secret parameters (e.g. passwords) to be sent that could be attacked by intruders.

In distributed systems all authentication schemes reduce to schemes enabling each communicating entity to obtain or posses some information which makes it possible to identify the other. This information is secret and can be protected by encryption. Authentication of each message received, and indirectly user identification, is required because of fear of active attacks on transmitted messages. Different approaches may be used to protect against different threats, and may provide different levels of assurance. This level of assurance needed depends on the costs of masquerades that stay undetected. Two authentication methods, depending on the amount of security needed, are:

1. Simple authentication. This method requires an authentication server and a user name plus password of every user. The receiver uses the authentication server to check the identity of the sender. However, the user name and password are transferred as cleartext which makes this method vulnerable to passive and active attacks.
2. Strong authentication. This method uses additional encryption techniques (based on public-key, private-key, or hybrid mechanisms) in the authentication process. In some cases this is not even enough (e.g. military systems), then digital signatures are required. Digital signatures serve as an extra identification from the sender of a message (the message becomes "signed"). The digital signatures also serve communication security because when they are decrypted successfully, the receiver knows that the message was not interfered with in transit (i.e. they serve message integrity). A well-known strong authentication protocol can be found in [Need78].

When designing an authentication protocol, in addition to choosing one of the above methods, the following questions should be taken into consideration [Lieb93]:

- Does the protocol work; is it secure against presumed attacks (e.g. replay in future protocol runs, insider attacks by masquerading, and brute force attacks like password guessing)?
- What does the protocol achieve, what believe does it establish for both communicating parties?
- What are the assumptions the protocol requires (e.g. secure generation of keys, trusting a third party)?
- Are there any unnecessary actions (e.g. could the number of exchanged messages in the protocol be reduced)?
- Is encryption being used and if so, which encryption algorithm has been implemented?

In the next chapter we will look at the consequences for the mechanisms described in this chapter when implemented on wide-address architectures.

3 Wide-Addressing

Microprocessors have evolved from 8-bit address- and datawidths in the mid 1970s to 32 bits in the early 1990s. Reasons for this growth are: the need for 16- and 32-bit data types (more precision), larger and more powerful instruction sets, and more addressable memory. Several applications requiring large amounts of memory were presented in chapter 1.
Nowadays, so-called 64-bit architectures arise. But what are 64-bit architectures? To be able to answer this question we will take a look at addressing in computer systems. This is depicted in figure 3.1 [Mash91].

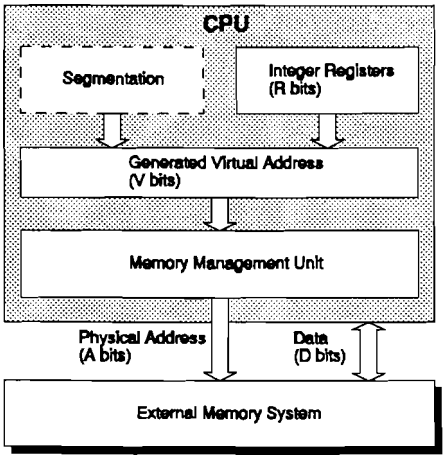


Figure 3.1 CPU addressing.

Segmentation was described in chapter 2 as a technique for handling large amounts of memory more easily. The integer registers on the processor have a size of R bits, the generated virtual address is V bits wide, the physical address by which the external memory system is addressed is A bits wide, and the data bus is D bits wide. Usually $R \leq V$ but by using segmentation, the physical size R can be expanded to reach V. Segmentation can be used to address more memory by using the same amount of virtual address bits. Address calculations are limited by the integer register width R while memory access is limited by the size of virtual address bits V and address buswidth A. Data buswidth D defines I/O (memory) bandwidth. For user-level programs, R and V are programmer visible while A and D are usually less visible, implementation specific values. It should be noted that segmentation is also visible to the user.
An architecture is often called an x-bit architecture if the integer registers are x bits wide. In this report, the name of an architecture is coupled to the virtual address size. Hence, an architecture is called 64-bit if and only if the virtual address is 64 bits wide. The integer registers are then usually 64 bits wide also to ease address calculations. Of course the physical addressing capabilities have to scale too because mapping a 64-bit virtual address onto an 8-bit physical address is highly inefficient in both address translation as well as protection.

The consequences of bigger integer registers are straightforward [Krue89]:

- Longer strings of bits or bytes can be handled more easily; clearing and copying parts of memory, logical operations, and other integer operations speed up.
- Integer arithmic scales up in performance.

This win in performance is comparable to the win in performance when 8-bit processors were replaced by 16-bit processors and 16-bit ones by 32-bit ones. So the only win is performance. However, the most important consequence of 64-bit architectures is the fact that the generated virtual and physical addresses scale upwards too. Therefore, one of the most basic assumptions of today's operating systems, the assumption that virtual addresses are sparse resources that have to be re-used, no longer holds. Hence, the multiple address space model that was to be the solution for extending a process' address space, is no longer necessary. Illustrating the importance of this statement is the fact that this address space model is supported by almost any current operating system implementation.

Like architectures, operating systems have changed also [Ande91]. They have changed to meet new requirements: fast local communication, distributed and parallel processing, virtual memory, and others. For improved extensibility, maintainability, and fault tolerance, modern operating systems are moving from the traditional monolithic, centralized kernel structure to a more decentralized structure. More and more functions are put into separate servers instead of in the kernel (two examples of modern operating systems are Mach [Jone86] and Chorus [Abro89]). These servers communicate with users, the kernel, and with each other through message passing. The reasons for this evolution are two-fold:

- By structuring the operating system as independent address spaces communicating through messages, modularity, fault tolerance, and extensibility are enhanced.
- Using messages rather than shared memory as the principle communication mechanism simplifies the move to a physically distributed structure.

However, operating systems and architectures have evolved somewhat independently. Reasons for this independent development of operating systems and architectures are:

- Simulation and measurement studies have been used to guide hardware design trade-offs but operating systems have been overlooked. Performance of the architecture has been mainly influenced by applications rather than by operating systems.
- "Traditional Unix" has driven the design of architectures, but most new operating systems differ significantly from traditional Unix.
- Operating system research has been strongly focused on performance. Therefore these operating systems make optimal use of specific hardware facilities of earlier architectures. These operating systems do therefore not exploit the specific advantages of newer architectures optimally.
- While some high-level operating system functions have been heavily optimized, software implementors have overlooked the importance of low-level functions.

Processes in an operating system share information, therefore they have to be able to address the piece of information to be shared. In the next paragraph a summary of distributed shared memory, a logical extension of shared memory in centralized systems, will be given.

3.1 Distributed Shared Memory

Distributed shared memory (DSM) is a shared memory abstraction that can be placed on top of a message passing system to give the users the illusion of a physically shared memory. DSM can be presented visually as is done in figure 3.2.

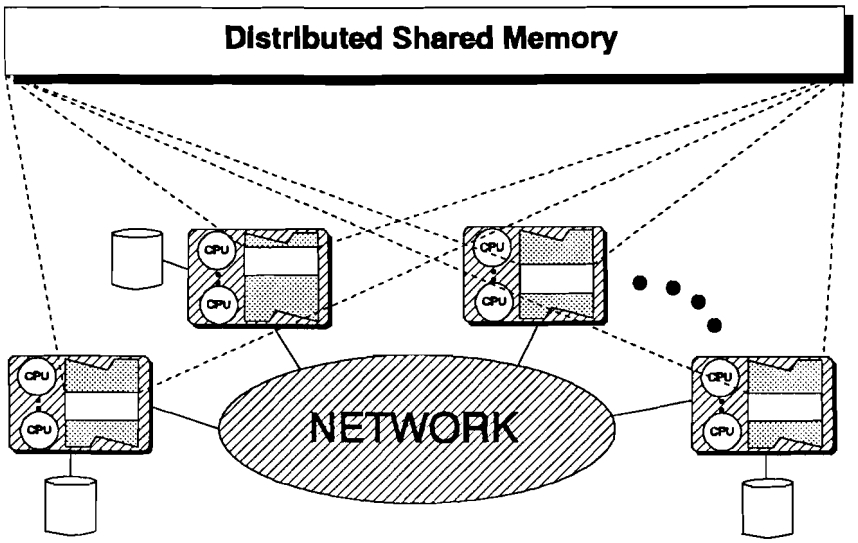


Figure 3.2 Distributed shared memory.

A part or all of the local virtual memory of a node is mapped as shared memory. Therefore, to the programmer on one node it seems as if a part of his/hers accessible virtual memory is being shared with other nodes, i.e. changes made to the shared memory area on another node will be made visible on all other nodes by the DSM system.
For DSM to be effective, it is not necessary that all nodes have the same architecture, this is called heterogeneity. Heterogeneity is especially useful when highly specialized hardware is necessary. An example of a DSM system (called Munin) can be found in [Cart91]. We will now look at some aspects of DSM in greater detail ([Flei88], [Nitz91], [Scot92]).

3.1.1 Advantages and Costs of Distributed Shared Memory

Some advantages of distributed shared memory systems are:
- There is no global bottleneck as in tightly coupled systems where the common bus to the shared memory is the major bottleneck. This implies that much more machines can be connected.
- It is the natural extension of a single processor system, therefore it is more straightforward to program than a "normal" distributed system. This implies that there is a high degree of distribution transparency (i.e. the user is not able to detect that the shared memory part is actually shared between multiple processors).
- Fine grain multiprocessing is possible. Processors can be mapped onto shared memory objects. The DSM system can then be used to synchronize access to processors.
- Fast IPC is possible when dealing with large data structures. A shared memory area can also be used in RPC to serve as a communication channel [Bers91].
- A shared memory region could reduce the number of I/O operations when files are mapped into the shared memory area.

However, there are also some problems associated to distributed shared memory:
- The machines connected to the network have to be close together for the access times to be acceptable. Of course a high-speed network solves this problem too. An example of a high-speed network is a DAN (i.e. Desk Area Network) which has been described in [Hayt92].
- Synchronization of shared memory access can complicate the semantics of access (message

passing has implicit synchronization).
- Shared memory can introduce a single point of failure in a computer system.
- There is a consistency problem inherent to shared memory. If one node updates the shared memory, all other nodes have to get this update or else data will become incorrect. This is a result of replication of data between nodes which is necessary for performance, reliability, and recoverability reasons.
- Heterogeneity is often unavoidable in a network because specific hardware could be available for particular operations. This complicates the shared memory implementation.
- Thrashing can occur when a page is needed on several nodes during the same period of time.
- The programmer's view of memory changes when using distributed shared memory.

3.1.2 Extensibility

A theoretical benefit of DSM systems is that their extensibility is better than that of tightly coupled shared memory multiprocessors. However, the limits of extensibility are greatly reduced by two factors:
1. Central bottlenecks (e.g. databus, network speed).
2. Global common knowledge operations and storage (i.e. broadcast messages or full directories whose sizes are proportional to the number of nodes).

DSM relieves the programmer from explicitly managing the underlying communication paths and allows the system to do more of the bookkeeping needed for communication management in large scale systems. This makes extending a DSM system an easier task for the application programmer. Such a system should be extensible in an application-level transparent manner.
A fundamental resource utilization question arises when extending a DSM system; as a system scales upwards, how much memory can be shared by users until memory contention becomes a problem. This is dependent on the amount of physical memory locally available and network speed.

3.1.3 Structure and Granularity

The structure of a DSM system is defined as the layout of the shared data in memory (objects, language types, associative memory, or no structure at all). Granularity is defined as the size of the unit of sharing (byte, word, page, or complex data structure). Often, the granularity of sharing is a page of memory (or multiple pages).
In systems implemented using the underlying virtual memory support system, it is convenient to choose a multiple of the hardware page size as the unit of sharing. In this case the memory management unit hardware can be used to trigger page faults in the DSM system.
The size of the shared memory page is dependent on several factors as was described in chapter 2 for the virtual memory page size.

3.1.4 Coherence and Consistency

Coherence is defined as the semantics of memory operations while consistency is a specific kind of coherence. The most intuitive form of consistency is strict consistency. In this case, a read operation returns the most recently written value. However, most recently is an ambiguous concept in distributed systems and for performance reasons this form of consistency is often weakened.
The reason that consistency problems occur in DSM systems is that data is being replicated among nodes for performance reasons. If data were not being replicated, all operations would have to be

serialized, creating a major bottleneck in the system.
There are a number of schemes to attack the consistency problem when data is replicated among nodes (virtual memory with combined paging and segmentation is assumed):

- One site stores the primary copy of the segment. Any time an item is changed in the segment, the entire segment must be transferred to all nodes that have copies of the segment. Such a large unit of granularity for updating can have merits depending on the frequency of updates, the frequency of complete segment copy, and the requirements for immediately seeing the updates. Another possible unit of replication is a page. The page is a suitable unit of replication because programs typically have a high degree of locality in data access.
- A variation, using the page as unit of replication, is that the primary node be different for each page of the segment. Each page of the segment could have its primary storage node, which is different from the primary storage nodes of other pages of that segment.
- A circulating token scheme where there is no primary site. The token, which contains all pages that have been updated, is passed around the logical ring.
- Write-update protocol, pages are assigned rights (read-only, write-only, read-write). When a writable page is changed, all copies of that page are updated. The write-invalidate protocol works the other way round, there can be multiple read-only copies of a page but only one copy of a writable page. The protocol invalidates all copies of a page except one before a write can proceed.

3.1.5 Heterogeneity

Heterogeneity is usually unavoidable because specific hardware and its software is often designed for a particular application domain. For example, super computers and multiprocessors are good at compute-intensive applications but often poor at user-interfaces. Personal computers and workstations on the other hand, usually have very good user-interfaces.
Heterogeneous DSM is a mechanism by which the advantages of both systems can be obtained in an integrated system, allowing applications to exploit the best of both [Zhou90].
Aspects of heterogeneity are:

- Data conversion, when data is transferred from a node of one type to another type of node, it must be converted before it is accessed on the destination node. This complicates transparency because the conversion process cannot be simply delegated to the DSM system. The reason for this is the type specific character of the conversion process and the fact that the DSM system does not a priori know how memory is being used. Hence, the application must either perform all data conversions by itself, or specify the layout of a page to the DSM system.
  Normally, the performance impact of page conversion is small compared to page migration overhead.
- The DSM page size is either (i) the largest page size of all heterogeneous systems, a drawback of this decision is that more data than necessary is moved when nodes with smaller page sizes are involved. Thrashing is more likely to appear with large pages. However, more virtual memory pages will fit into one DSM page. When a DSM page fault occurs, it can be handled as a group of virtual memory page faults. The DSM page size can also be (ii) the smallest page size, less data contention but more page faults on hosts with small virtual memory pages will then occur. This results in more fault handling overhead and (small) page transfers.

3.1.6 Performance

Performance of a distributed shared memory system will greatly depend on the underlying network architecture as was indicated in the preceding paragraph. Fibre optics, which is a viable alternative to

copper wire and coaxial cable in many types of computer networks, allows for very high bandwidths. Security is enhanced also by use of this medium because at the moment active and passive attacks still require expensive equipment. By using higher bandwidths, more nodes can be added without considerable increase in access time to the shared memory region.
We now consider the hypothetical case of an object that is shared between nodes. For the DSM system this means that the object resides at one node while a copy of the object is present at the other node. To keep the object consistent among these two nodes, messages containing changes to the object have to be exchanged by the DSM system in a user-transparent manner. Therefore, the performance of applications that use DSM is expected to be worse than if they use message passing directly, since message passing is a direct extension to the underlying communication mechanism of the system while DSM is implemented as a separate layer between the application and the message passing system. However, for some existing applications, DSM can result in superior performance. This is possible for two reasons:

1. For many DSM algorithms, data is moved between nodes in large blocks. Therefore, if the application exhibits a reasonable degree of locality in its data access, communication overhead is amortized over multiple memory accesses, reducing overall communication requirements.
2. Many distributed parallel applications execute in phases, where each compute phase is preceded by a data collection phase. The time needed for this data collection phase is often dictated by the throughput of existing communication bottlenecks. In contrast, many DSM algorithms move data on demand as they are being accessed, eliminating the data collection phase, spreading the communication load over a longer period of time allowing for a greater degree of concurrency, and reducing the amount of data transported.

DSM can be implemented depending on the address spaces model implemented in an operating system. In the next paragraph several address space models will be presented including the most simple one in supporting sharing; the single virtual address space.

3.2 The Single Virtual Address Space

The main goal of an addressing scheme is to store information and to make it available. Besides this main goal, the addressing scheme has to facilitate sharing, protection, and persistence of information. Two mechanisms to facilitate sharing influenced by the addressing scheme are [Sark90]:

1. By using a system wide address space, all processes in the distributed system execute in this single virtual address space. A virtual address is always unique in such a system in contrast with conventional operating systems in which every process has its own address space that serves as a context for virtual addresses. A system wide address space is also called a single virtual global address space. A single virtual global address space because there is only one virtual address space that is shared by the entire distributed system. Pilot is an early example of an operating system based on a single virtual address space and a description can be found in [Rede80].
2. Through an extra level of mapping. In systems in which processes have their own address spaces, an additional level of mapping is necessary to facilitate sharing. Virtual addresses can be extended with address space identifiers or can be mapped onto system-wide unique identifiers.

Thus, the two ends of addressing schemes are one address space per process at one side and a single virtual global address space at the other side. Between these two addressing schemes, a sort of combination between the two of them is possible. Processes could have separate address spaces of which a predefined part could be used to facilitate sharing. Disadvantages of this approach are:

- The mix of shared and private regions introduces dangerous ambiguity.
- The virtual memory hardware and software must continue to support multiple sets of address translations in contrast with a single system wide address space.

The three discussed address space models are visualised in figure 3.3.

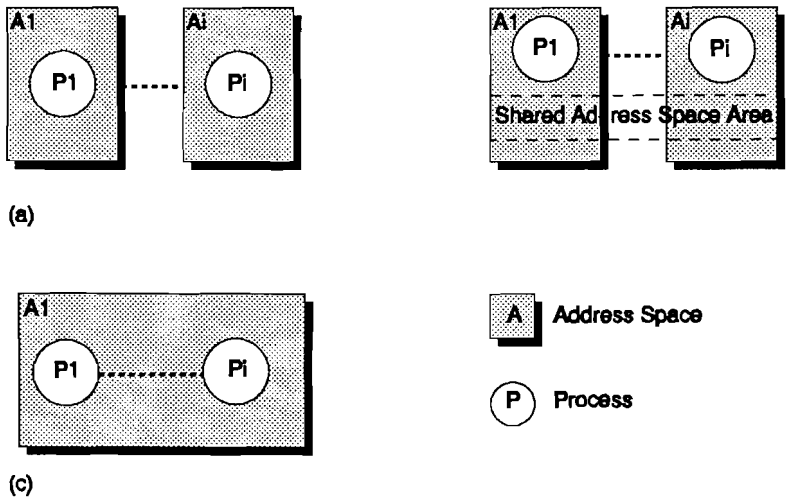


Figure 3.3 Address space models. (a) Multiple address spaces (one per process), (b) Same as (a) but a shared area between address spaces is possible, and (c) single virtual address space.

The benefits of wide-address architectures could be optimally exploited in systems with a global single virtual address space (i.e. system wide virtual address space). This addressing scheme requires wide virtual addresses that can serve as unique names for all objects in the address space. The uniqueness of virtual addresses is the principal advantage of this addressing scheme. This context-independence supports sharing in two ways. First, virtual addresses can be passed between domains and linked data structures stored in the global address space are meaningful to any domain that can access them. Second, a domain can address any piece of shared data without risk of name conflicts with other data. This property is essential when supporting efficient passing of data and procedure call by reference. Hence, we can state that the single virtual global address space is the most natural and efficient address space model in supporting sharing.
Schemes in which a separate address space per process is used, do not benefit significantly from newer architectures with wide-addressing capabilities. The only consequence for these schemes is that the address spaces per process can be larger; however, in today's systems the address space per process is approximately 2^{32} bytes (see figure 1.1) which is considered large enough for most applications.
A single virtual global address space could also be implemented on conventional architectures that were designed to support operating systems with multiple address spaces. However, in doing so, unnecessary performance costs would occur. Usually those architectures maintain linear page tables per process (hence per address space). The address space can then be treated as a protection domain, resulting in one separate page table per domain. Possibly supported ASIDs (i.e. address space identifiers, for example see the MIPS R4000 architecture in appendix A) can be used as protection domain identifiers resulting in TLB entries that can survive protection domain switches.
This organization causes two problems for single virtual address space systems. First, domains in such a system have a sparse view of the global address space, each domain would typically reference small and widely scattered pieces of the address space. Linear page tables cannot represent such sparse sets of mappings compactly. However, this problem could be overcome by using the inverted page table approach (see paragraph 2.1). Second, translation mappings for shared pages must be duplicated in the page tables for each domain because access rights may differ, this wastes space and forces the kernel to keep duplicated mappings consistent. Besides this, conventional architectures are usually 32-bit architectures, which do not offer enough virtual address space to name each object in a distributed system uniquely by its virtual address. Systems that use segmentation can construct a wider virtual address by concatenating a segment identifier and an offset but disadvantages of segmentation in

supporting wide-addressing are:

- Cross-segment pointers are not supported.
- Multiple pointer forms must be treated differently by applications because segmented architectures usually offer short- and long-form addresses.
- Software must coordinate segment register usage to create an illusion of a single virtual address space.

The advantages of systems using a single virtual global address space are:

- As was stated before, virtual addresses can be used as unique system names. This is the most simple and efficient name organisation. Virtual addresses can be parsed in minimum time compared to other name organisations, e.g. hierarchically structured names. In systems with multiple address spaces this is not possible because virtual addresses are not unique across address spaces.

  However, a disadvantage of using virtual addresses as unique system names is that objects cannot be relocated easily. In that case, the system names of objects would change. Possible solutions to this problem are: (i) use an extra level of indirection to name objects, and (ii) leaving forwarding records when objects have moved.
- Shared memory is easier to use than a multiple address space environment (see paragraph 3.1). Shared memory can reduce the need for explicit communication between programs executing in separate protection domains. Explicit communication (e.g. message passing, RPC) is becoming more expensive because it is based on data copying and protection domain switching, operations that have increased in cost relative to integer arithmic performance on recent processors [Oust90]. At first glance, one would say that shared memory limits safety or isolation, the design rationales for developing the multiple address space model. However, distrusting domains can limit their interactions by restricting the scope of the shared region and the way in which it is used. Memory can be shared sequentially by transferring access permissions so that distrusting programs can verify data before using it. In this way usage of shared memory is no less safe than more expensive mechanisms for cooperation.
- Hardware based memory protection is cheaper and more flexible in this addressing scheme. Protection domains can be created more efficiently when the system does not have to set up a new address space for each domain. This can result in a decoupling of protection and address translation. A major advantage of this decoupling is the possibility of separate granularities for protection and translation.
- Virtually indexed, virtually tagged caches [Whee92] can be used without problems encountered in multiple address space systems. This type of cache does not require address translation before the cache access takes place, thus speeding up the cache lookup process. Two problems that arise when using a virtually indexed, virtually tagged cache in multiple address space systems are:
  1. Synonym problem, this occurs when a physical page is mapped onto two or more different virtual pages. References to an item through different virtual addresses causes that item to appear simultaneously in multiple cache lines, creating a coherency problem on writes. In single address space systems, write shared data always appears at the same virtual address in each process that uses it, hence the synonym problem does not occur.
  2. Homonym problem, this occurs when each address space has a different translation of the same virtual address. In multiple address space systems this problem can be solved by extending the virtual address with address space identifiers, by flushing the cache on process switches, or by physically, instead virtually, tagging the cache. However, all solutions have drawbacks. In a single address space system, homonyms cannot occur because by definition only one translation for each virtual address exists.

  A virtually indexed, virtually tagged cache, which is the most attractive caching organisation for performance reasons can thus be used without problems and without the costs of extra ASID bits or flushing the cache on domain switches in a single virtual address space system.

There are also some disadvantages when using a single virtual global address space:

- Because addresses are now shared resources, new problems arise that were not present in conventional systems (multiple address spaces), e.g. different nodes using the same address ranges. The solution to this problem is not difficult (e.g. assigning exclusive address ranges to nodes) but several optimisations might be possible.
- Using virtual addresses as names (virtual addresses are unambiguous, global, and possibly pure names, see paragraph 2.2.1) has its limitations. Because virtual addresses are possibly pure names, they only identify but do not help the system when finding the location of an object. This can be solved by using a second kind of, hierarchically structured, name (the virtual address could also be subdivided into a node number and some local information to use the virtual address as a partitioned name itself, the name then becomes impure). Thus, the interaction between operating system and processes can be the following. Before being able to access an object, the process has to tell the operating system that it is going to use the object by passing its hierarchical name. The operating system then physically locates some administrative information about the object (e.g. virtual address, access rights, disk storage position, etc.) whereafter the process is able to access the object according to the assigned access rights. Another argument for using a second, hierarchically structured name that is mapped onto the virtual addresses are users not being able to cope efficiently with virtual addresses as object names. Virtual addresses are nothing more than identifiers, they do not contain any extra information in their structure.
- In a single segmented address space, each object is treated as a separate segment. A central object table could be used to keep track of the location of each object, and usually the number of outstanding references to it. This reference counting is expensive to implement as a secure mechanism and may require hardware support. However, the major drawback of this approach is the size of the object table. An object could be as small as the entry in the object table required to register it. This can result in very large object tables. Therefore, objects cannot be collected in a central object table because of the enormous size of the address space (and thus the enormous amount of objects possible).
- In a single paged address space, the representation of an object can start anywhere in some page and cross page boundaries. Reference counting cannot be used because there is no separate object table to hold the counts. Therefore, moving or deleting an object is very hard to implement correctly because pointers to it must be found and updated. The page table for such a single global virtual address space is thus very large and may need to be organized as a hash table. A possible solution to this problem is the inverted page table that was discussed in chapter 2.
- Because all objects reside within the same address space, a major disadvantage is the lack of isolation the multiple address space model offered. Thus, the single virtual address space requires a securely implemented protection model. Protection in a single virtual address space can be described by the protection domain abstraction. Protection domains provide the necessary isolation available in multiple address space models.
  Protection can be managed independently from the virtual-to-physical mapping in contrast to conventional systems where translation and protection information are usually stored in one place (e.g. page table). This separation of issues allows for different granularities in translation and protection.

Disadvantages of the single global virtual address space can be summarized as organisation issues. Organisation in such a large virtual address space can be seen as a complicated task.
In the next paragraph two almost trivial extensions to the single virtual global address space will be presented.

3.3 Extensions to the Single Virtual Address Space

The single virtual address space is the most simple and efficient address space model in supporting sharing. This results from virtual addresses being used as unique object names. Because using this simple unique object identifiers, two extensions to the single virtual address space become almost trivial.
The first extension is a single-level store which will be discussed in paragraph 3.3.1 and the second will be persistence, presented in paragraph 3.3.2.

3.3.1 The Single-Level Store

Virtual memory usage has grown steadily over the last years. Some systems have made an extension to the conventional virtual memory concept by introducing techniques which allow files to be addressed as segments in virtual memory, thus making a conventional filestore unnecessary. This concept can be implemented in two different ways. First, secondary storage can be seen as a part of the address space, data is stored in a format understood by the underlying virtual memory hardware. Second, a separate secondary storage format can be used to store memory data (i.e. memory data is converted to another format before written out or read in). This approach incurs costs that result from the difference between secondary and primary storage formats because data now need to be converted, buffered, and kept consistent.
The first approach is the most logical one. All objects (also disk resident ones) are mapped into virtual memory and are made homogeneously addressable. This is called a single-level store [Cope90].
However, some disadvantages of a single-level store in conventional operating systems are:

- The virtual address is too small to name all transient and persistent objects in the system uniquely.
- Page table size is excessive.

The first drawback can be overcome by integrating secondary storage into the single virtual global address space residing on a wide-address architecture. The second problem is common to all large virtual memory systems and can be solved by using an inverted page table.
Virtual addresses can then serve as unique object identifiers for objects in primary as well as on secondary storage. Advantages that arise as a result of implementing a single-level store are:

- Virtual memory addresses can be used as object references in both memory and disk, hence object traversing becomes faster.
- Format conversion between data structures in primary- and secondary storage is no longer necessary.
- Objects can be accessed without being copied because they can reside in page buffers. Objects appear in pages as meaningful data structures, therefore an extra object buffer is not necessary any more.

Usually the single-level store is organised as a paged system for the following reasons:

- Disk I/O becomes less excessive when pages and blocks are taken into account. A complex object can be transferred as a collection of pages. Otherwise, every access to a complex object could require disk I/O.
- Main memory does not become fragmented.
- Locality of information can be exploited.

Concluding, a single-level store can be implemented easily in a single virtual global address space by using the unique virtual addresses as identifiers for all objects. Examples of single-level store operating systems are Multics [Bens72] and the IBM 801 system [Chan88]. However, both rely on segmented memory management.

3.3.2 Persistence

The single-level store concept can be extended even further to allow arbitrary data structures to be created and manipulated in a uniform manner, regardless of how long they persist. Informally stated, persistence provides for a user-transparent single-level store. This concept of persistence has been incorporated in several programming languages, resulting in a flexible and powerful programming environment. Orthogonal persistence is a special form of persistence which has three properties:

1. Persistence independence, the persistence of data is independent of how data is manipulated. The user does not have to, indeed cannot, program to control the movement of data between long term and short term storage. This is performed automatically by the system.
2. Data type orthogonality, all types of object should be allowed the full range of persistence. Note that data type orthogonality does not automatically imply that all types of objects are identified uniformly.
3. Persistence identification, the mechanism for identifying persistent objects is not related to the type system. All object are identified uniformly.

Information is said to be persistent if it has an arbitrary lifetime, which is unknown at the moment of creation. To be able to store such information, the system must support process-independent names or addresses that can be retained between program executions. When a persistent store is integrated with virtual memory management, a persistent address space is created. In a single virtual global address space system, the virtual addresses can also serve as identifiers for persistent data assuming the address is wide enough to identify each object in the system uniquely.
In systems with narrow virtual addresses, persistent pointers are created at the moment an object is stored. This causes an extra level of indirection. These extra identifiers are called surrogate pointers. Surrogate pointers require special compiler support to insulate the application from the need to manage two pointer forms. Also, mapping tables are needed to translate surrogates into virtual addresses. This translation is expensive. Swizzling is a technique to reduce the costs of this translation by overwriting the surrogate in place with the virtual address after the first translation. However, disadvantages of swizzling are:

- It adds to the overhead of pointer translation. Swizzled pointers must be translated on input and output.
- It interferes with virtual memory caching. Pages are marked dirty even when an application does not reference them.
- It inhibits sharing. Each domain must maintain a separate copy of the data, since pointers are swizzled to different virtual addresses in each domain.
- Compiler support is needed to locate pointers in stored data.

Surrogates and swizzling are necessary compromises for narrow-address machines. In wide-address machines, virtual addresses can serve as uniform object references that are never re-used again, not even when an object is deleted because the name space is large enough to provide unique names. They are simpler and more efficient and can be used directly as object references when proper operating support for a persistent single virtual global address space on wide-address architectures is available.
Advantages of systems supporting persistence are:

- Reduced complexity for application builders.
- The number of transfers between main and secondary memories can be considerably reduced and thus execution time is reduced. In most conventional systems the decision to start executing a new program requires the entire program to be transferred from its permanent location on secondary storage to a new temporary program image in primary storage. When using virtual memory, most pages or segments of a large program will then be moved out of primary storage and before being used will be brought in when a page fault occurs. This results in movement of program parts from one area of secondary storage to another.
- Persistent systems avoid duplication of mechanisms (e.g. separate protection of executing programs

and stored files).

- Persistent systems allow the same software to be used to manipulate temporary and permanent data. This results in reduced code size.

Persistent systems have the described advantages but at cost of:

- Constructing a stable object store. The system must guarantee that, following a failure, the persistent store always returns to a consistent state as at some previous checkpoint.
- Loss of efficiency with some applications due to the abstraction. The user does not have complete control of all physical properties of the system. This is comparable to virtual memory, for example the user does not have control over the paging mechanisms used.

Summarizing, the single-level store and persistence are extensions to the single virtual global address space that can be implemented without costs occurring in conventional systems because of the virtual address properties in a single virtual address space.

In the next chapter, two case studies will be presented that include many of the issues described in this chapter concerning the single virtual global address space.

4 Two Case Studies

As stated earlier in this report, single address space operating systems are operating systems in which all processes run within a single global virtual address space. This virtual address space can map all primary and secondary storage across a local area network.
We will now focus on two different approaches to this subject to illustrate some important aspects of single address space operating systems. The two operating systems we will be looking at are: the Opal operating system (paragraph 4.1) and the Monads system (paragraph 4.2). The reason for choosing particularly these two operating systems is that both cover the single virtual address space concept yet differ significantly in implementation. Also, both of them have been described in literature rather extensively. Covered issues are: (network) architecture, virtual storage, protection (security), and object support.
A description of both systems is given whereafter a comparison is made.

4.1 The Opal System

The Opal project is an investigation into the effect of wide-address architectures on the structure of operating systems and applications. The goal is to determine how software can best exploit the large virtual address spaces of these architectures. Research is being done at Chicago State University (USA).
No special hardware support for Opal is assumed or required except for wide-addressing. However, single address space operating systems can benefit from hardware that is optimized for the way they use virtual memory. This will be explained in paragraph 4.1.3.

4.1.1 Architecture

Opal is designed for a distributed environment composed of one or more nodes connected by a network. Each node contains physical memory, one or more processors attached to that memory, and possibly some long-term storage (e.g. disk). For simplicity, homogeneous processors are taken; all nodes consist of MIPS based workstations. No assumptions are made about the topology of the network. It is assumed that two nodes that need to communicate can do so, and that the network communication mechanism is reliable. Issues related to achieving this service, such as handling of duplicate packets, will not be considered. The layout of the system is depicted in figure 4.1.

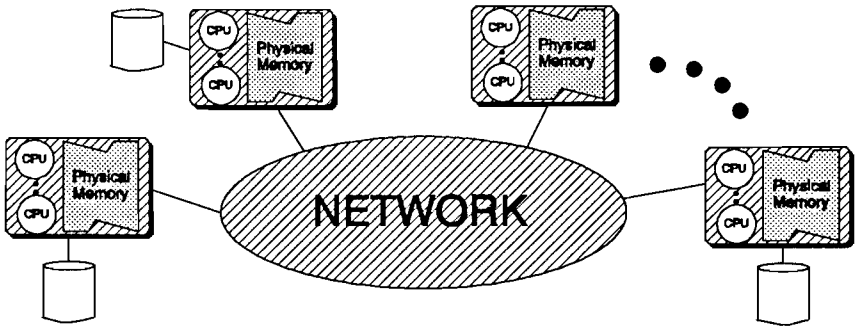


Figure 4.1 Layout of the Opal network.

4.1.2 Virtual Storage

Opal's virtual storage is partitioned into virtual segments of varying size in which all data is stored. Segments are composed of a variable number of contiguous virtual pages. Each segment occupies a fixed range of virtual addresses, assigned when the segment is created, and disjoint from address ranges occupied by other segments. Segment boundaries are unknown to the hardware, and addressing is independent of segments; all memory references use a fully qualified flat virtual address. The virtual segment structure is shown in figure 4.2.

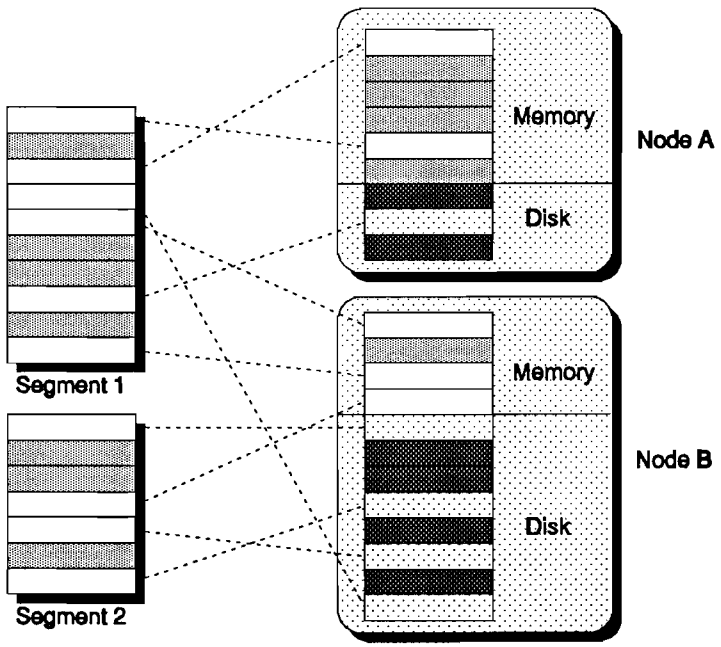


Figure 4.2 Virtual segment structure.

Segments are potentially distributed, persistent (i.e. the segment will continue to exist even when it is not used) and recoverable (i.e. the segment is backed on nonvolatile storage and can survive system

restarts; it can be compared to a mapped file in a traditional filesystem), but the storage management policies are not dictated by the kernel. Instead, consistency and recoverability are managed by external paging servers according to segment type. The paging servers transparently manage some aspects of physical storage backing the virtual segments:

- When data is first written into a virtual page, the system allocates a physical page to back it.
- The system may move a physical page between node memories and disk, or it may make multiple copies of the page.
- When a program references a virtual page, a copy of the page is faulted into the memory of the program's node.

4.1.3 Protection

Programs execute within protection domains that define rights which limit their access to global virtual storage. A protection domain defines private data, code, and stacks that an application can access, along with any data shared with other domains. In many ways it is the analog of the address space of a Unix process; the primary difference is that it defines a private set of access privileges to globally addressable segments, rather than a private virtual naming environment. Opal domains can grow, shrink, or overlap arbitrarily, by sharing or exchanging segment permissions with other domains. In figure 4.3 virtual segments in relation to protection domains are shown.

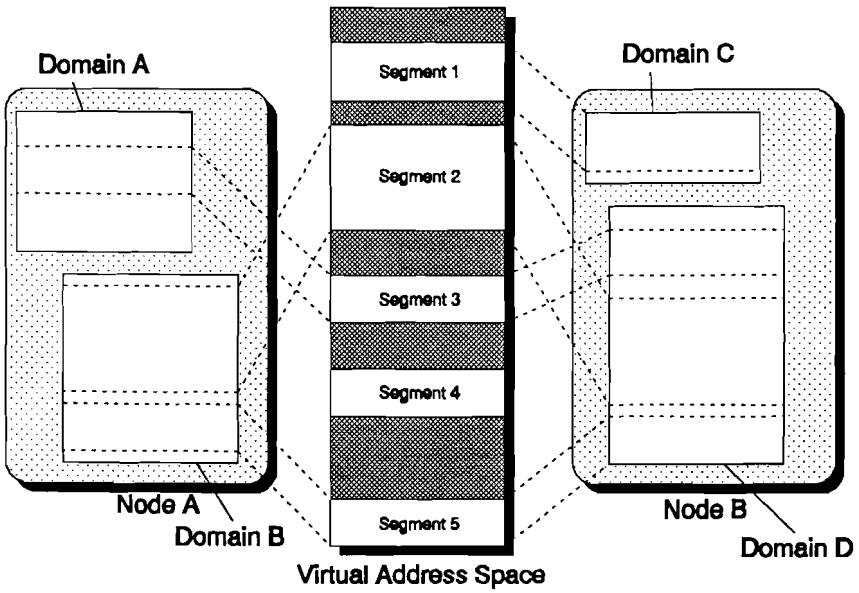


Figure 4.3 Protection domains and virtual segments.

Protection domains cannot always address any given piece of data directly. Resources are pieces of data (objects) that are managed by server domains through a protected RPC interface because it is not safe for them being accessed directly. For client domains to name resources and show permission to operate on it, the resources are named by sparse capabilities (see paragraph 2.3.1.2). Sparse capabilities can be passed through shared or persistent memory, just like ordinary pointers. Figure 4.4 shows an Opal capability, also denoted as a protected pointer.

Figure 4.4 An Opal protected pointer.

An Opal pointer is a 256 bit value, consisting of the following fields:

- Portal number, identifies an RPC endpoint registered for some server domain. When a program wishes to export a set of services it registers with the global name service that allocates a unique portal to the set of services. The portal number is allocated from the global address space to be unique. Any thread that knows the name of a portal can make a domain switch through the portal; conceptually, the thread begins executing in the server domain at a virtual address chosen by the server and specified when the portal is registered.
- Object identifier, the system does not dictate a format for this field but in the common case this will be the virtual address of the object.
- Check field, the structure of this field is not dictated by the system. The value is chosen by the server and allows the server to validate the protected pointer by matching it with a corresponding check field stored with the object's representation. The check field can be calculated from the other fields in the capability.

Any domain that knows the value of a sparse capability can gain access to the named resource.
Virtual segments are examples of protected resources named by capabilities that support the operations attach and detach. A segment capability confers permission to the holder to attach the segment and access its contents directly with load and store instructions. In general, each domain has permission to attach only a small subset of segments of the global virtual store, limited because of its access rights, with some combination of read, write, and execute privilege. When an unattached segment is referenced, a segment fault is generated which causes the system to transparently load and attach this segment (this procedure depends on the type of segment) if access is permitted. In this case access control lists managed by segment servers are used to determine permission.
A great part of memory management is done by two separate servers on each node:

- The address space service coarsely partitions the virtual address space, assigning large ranges to segment servers which subdivide them into segments.
- The portal service assigns unique portal names to services, and can locate a server given a portal name.

The way in which these servers cooperate with their peers on other nodes is comparable to distributed name management (see paragraph 2.2.3).
Within each node, Opal maintains a single mapping of virtual to physical pages. However, across nodes this constraint would be too restrictive, because caching of multiple copies of a page is crucial for performance and reliability. In this way the system allows for different types of coherency algorithms.
For performance reasons, services should be structured as a group of cooperating servers, by placing a server acting as an agent for the service at each node. Local clients access the distributed service through the agent. The agents may keep replicated copies of the service's data for performance, reliability, and availability. The two separate memory services described above (i.e. address space service and portal service) are also structured in this way.
Opal portal names are unique within a node but may be registered to different agents of the service across nodes. Of course this approach assumes that the object identifier embedded in the capability is resolved to the same resource by each agent. Resolution and caching can be simplified by delegating organisation issues to the DSM system if the agents cooperate using distributed shared memory.
At the hardware level there are other protection problems to consider. Most current architectures have been designed to support multiple address spaces and thus generate multiple page tables. These

architectures could support a single address space but in doing so they may incur unnecessary performance costs. Architectures with software-loaded TLBs, such as the MIPS R4000 or the DEC Alpha (a description of both architectures can be found in appendix A), are better suited to support single address space systems. Without a hardware-enforced page table structure, the operating system can choose the most convenient representation for its virtual memory data structures, such as a single table of translations that is shared by all protection domains and a separate protection table for each domain. In that case the concepts of translation and protection are being separated. Possibly available address space identifiers could be used as protection domain identifiers.
Two alternative high-level models for supporting protection in single address space systems are [Kold92]:

- The domain-page model, which specifies access rights explicitly for each (domain, page) pair. This model can be implemented by using a so-called protection lookaside buffer.
- The page-group model, which defines groupings of pages called page-groups. A protection domain is defined by the set of page-groups that it can access. The page-group model has been used in the Hewlett-Packard PA-RISC (appendix A).

In paragraph 4.1.3.1 the domain-page model implementation will be discussed while in paragraph 4.1.3.2 the page-group model as implemented in the HP PA-RISC (appendix A) will be looked at. Paragraph 4.1.3.3 will contain a discussion on both mechanisms.

4.1.3.1 The Protection Lookaside Buffer

The domain-page model can be implemented by a new memory system protection cache, called the protection lookaside buffer. This PLB caches protection mappings on a per-domain, per-page basis, that is, each PLB entry contains protection information (the access rights) granted to one protection domain for one specific virtual page. The high-level organisation of a PLB is depicted in figure 4.5.

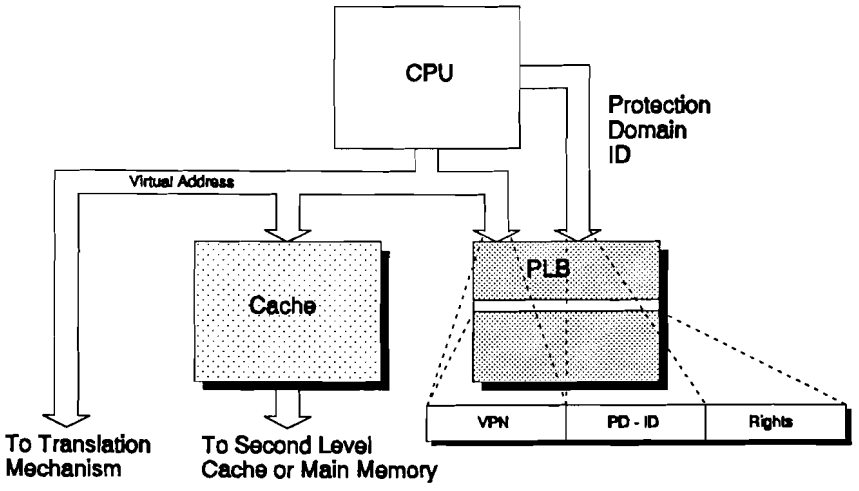


Figure 4.5 System with a protection lookaside buffer and a virtually indexed, virtually tagged, cache.

On each memory reference the PLB is accessed by the virtual page number (VPN), which is extracted from the address, and by the current protection domain identifier (PD-ID) of the executing process which can be kept in a separate processor control register. If a match occurs on both the VPN and the PD-ID the protection information is extracted directly from the entry in the PLB. If no match occurs the PLB must be loaded with the appropriate protection mapping. The cache and PLB lookup can occur completely in parallel because the cache lookup is independent of the information provided by

the PLB. If the cache lookup succeeds, the PLB provided information indicates whether the memory reference has permission to proceed. If the cache lookup fails and the PLB indicates an illegal reference then an exception will be generated, otherwise, a cache miss occurs and a translation must be provided by the TLB. The cache is then updated according to the cache policy.
When pages are shared across domains, the PLB contains multiple entries for those pages, one per domain. This indicates the major advantage of separating translation and protection mappings; when both protection and translation mappings were stored in the PLB (comparable to a traditional TLB), each entry could contain different rights but identical translation information. This would mean duplication of information and the inability of using a virtually indexed, virtually tagged cache (for a discussion of virtually tagged, virtually indexed caches [Whee92] is referred to).
With a virtually indexed, virtually tagged cache only a small percent of cache accesses miss or require a writeback. Therefore the TLB can be moved off the processor chip, which permits an even bigger TLB than typically found in microprocessors. The TLB need only contain the VPN/PPN mapping and the dirty and reference bits. Furthermore, the TLB requires only one entry for each virtual-to-physical page mapping; therefore, a purge is required only on the change of a virtual-to-physical translation. Protection domain switches do not require a purge of either the PLB or the TLB.

4.1.3.2 The Page-Group Protection Mechanism

The page-group model has been implemented in the Hewlett-Packard PA-RISC architecture as described in appendix A. The protection hardware is depicted in figure 4.6.

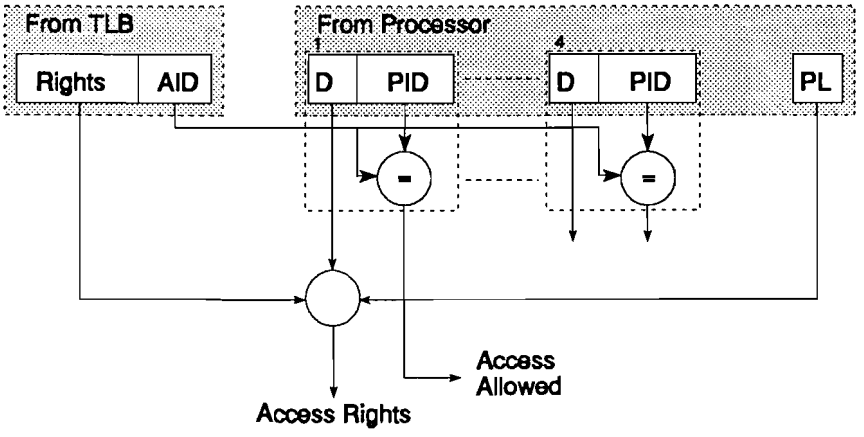


Figure 4.6 Hewlett-Packard PA-RISC protection hardware.

However, the current PA-RISC architecture, with only four page-group registers, limits the number of page-groups that a domain can efficiently access at a time; in addition, it provides no information (such as last recently used (LRU) information) to help the operating system manage the loading of page-group registers. Therefore the PA-RISC's page-group registers should, for example, be replaced by a cache of permitted page-groups, with support for replacement strategies.
Because the page-group model requires only one set of access rights per page (and therefore one entry in the protection cache), protection and translation information can be combined in a TLB without duplicating translation data. Since the TLB must be accessed on each memory reference to obtain the protection information, the implementation requires an on-chip TLB.

4.1.3.3 Comparison of Protection Mechanisms

We will now compare the PLB and the Hewlett-Packard PA-RISC protection hardware. Each system has its advantages and disadvantages. Some of the most interesting aspects are:

- Attaching virtual segments is comparable in complexity for both systems. However, detaching is more complicated in the PLB system. In contrast with the page-group system where detaching simply means removing the page-group identifier from the page-groups encompassed by the current domain, the PLB system requires purging the PLB of any entries that map the detaching segment from the current protection domain. In the worst case, this could require inspecting all the entries in the PLB and eliminating those that match.
- Changing of individual (per page) access rights is straightforward in a PLB system. However, in the page-group system this depends on the number of domains involved. If the rights are being changed for all domains that can access a shared page-group, the change is easily made in a single TLB entry. However, changing the access rights for a subset of all domains that can access a shared page might require moving pages between different page-groups.
  Therefore, a PLB system will take fewer faults in situations where there is active sharing and frequent protection changes at cost of redundant entries in the PLB while the page-group system performs relatively better when sharing is static or protection changes are infrequent.
- The cost of domain switching, an activity which is becoming more frequent as many operating system services are now provided by application-level servers accessed through RPC calls is less on PLB systems. In PLB systems, a domain switch requires changing only one register; the PD-ID register in the processor. In a page-group system, domain switching requires purging the page-group cache and loading in the page-groups for the new domain.
- Protection checking in the PLB system requires only one cache lookup to provide the requested protection information while in the page-group system this activity requires two cache lookups (one for the TLB and one for the page-group cache). However, the tags being compared in the PLB are wider than either of the compared values in the page-group implementation (VPN and PD-ID).
- In the PLB system, the granularity of protection and translation can easily be different because protection and translation are separated.

4.1.4 Object Support

In Opal the term object is defined as a contiguous sequence of bytes that is an instance of a language type. The size and structure of an object are statically determined by its type, and known only to the type implementation. An object reference is the virtual address of the first byte of the object.
Opal objects can be used to represent data, code, and protected service objects. The term service is used to denote a service class or type (e.g. file service), while a service object is an instance of the type (e.g. file). Opal makes no distinction between data and protected service objects as many other systems do. An Opal object is a "protected" object only from the perspective of a client that is insufficiently privileged to attach the object and invoke its methods through ordinary procedure calls.
Objects are clustered into segments that can be attached. Instead of placing objects in separate segments they are clustered to prevent waste of memory for small objects; without clustering small objects like integers would have to be placed in very small segments. However, the smallest segment equals the page size in the Opal system. After attaching, domains can directly address the objects in the segment according to their privileges.
Clients and servers can be classified trusted or untrusted. Trusted clients calling trusted servers can directly access everything in the other's protection domain. An untrusted client (server) only has access to the parameters (results) and must perform a trap to change protection domains when the client-

server protection boundary is crossed. When a program wishes to export a set of services, it registers these services with the global name server. A server can specify the protection domains (users) that are permitted trusted access to its protection domain.
The key object property for services is encapsulation, which has two aspects:

- Enforced integrity. A client cannot maliciously or accidently corrupt the service because it cannot operate on it except by invoking its methods. The method code is chosen by the implementor of the service rather than by the client.
- Object-grain access control. A client cannot invoke the methods of a service without having a capability for the service. The capability for the service can only be obtained from another client which holds a capability for that service or from the service directly.

In system environments where unsafe programming languages may be used, encapsulation is enforced by two means:

- The object is separated from the client by a hardware enforced memory protection boundary. This is done by classification of clients and servers into trusted and untrusted ones.
- Capabilities are used to reference an object.

Clients invoke services by passing capabilities for other objects that they want the service to operate on, and only on those objects. Services do not run as some kind of "superuser". Protected service objects are extensible by defining new services, simply by creating objects and passing capabilities to clients.
It should be noted that object support has been implemented as a run-time package in the Opal system and is only a solution for organizing and controlling sharing [Chas92a].

4.2 The MONADS System

The MONADS project was originally established with the main aim of investigating improved techniques for designing and developing large software systems. The solutions have been sought in terms of an integrated architecture involving both hardware and software. Research originated from Monash University (Clayton, Australia) and is now being done at several other places. The project has three major design philosophies which affect the hardware design in several different ways [Rose85]. These three philosophies are:

- Information hiding, all conventional program units and files are implemented as information hiding modules and all data is accessed via a procedural interface. Limiting the scope of access to data is one of the ways to successful software engineering. This philosophy is comparable to encapsulation in object-based systems.
- A procedure-oriented or in-process model, in which operating system functions are requested by procedure calls within the user's process. This model requires hardware support for domain switches on procedure call and return.
- Uniform virtual memory, all of the primary and secondary storage across the network is part of the virtual memory. A consequence is that addresses must be quite large to uniquely identify all objects.

The original MONADS project was based on a purpose-built microcoded microprocessor designed for a centralized environment and was denoted MONADS-PC (Persistent Computing). Although its address space (2^{60} bytes) was much larger than that found on conventional architectures, it was not considered large enough for realistic implementations, particularly with regard to networks. In addition it was constructed out of quite old technology, limiting performance. For these reasons a new implementation of the MONADS system was born, named the MONADS-MM (Massive Memory) system which among others features a 2^{128} byte address space. This system is based on an adapted SUN SPARC processor to meet the hardware requirements. In the remainder of this discussion we will speak off the MONADS system without distinguishing between MONADS-PC and MONADS-MM when not

explicitly stated.

4.2.1 Architecture

The MONADS system consist of homogeneous MONADS computers connected by a local area network. Each MONADS computer is equipped with a specially designed address translation unit (ATU), one processor, and physical memory connected to it. A node may, but need not, also have its own disks. This results in an equivalent scheme as is shown in figure 4.1 but with only one processor per node.

4.2.2 Virtual Storage

It is important to notice that the virtual-to-physical mapping is separated from the virtual-to-disk address mapping of pages. By separating these two issues different mechanisms and structures may be applied to the relatively static disk address information and to the volatile main memory information.
We will first take a look at the virtual page-to-physical page mapping [Rose92]. At the hardware level, a special mechanism is provided for translating virtual to physical page addresses. This mechanism is proposed in figure 4.7 and comparable to the inverted page table discussed in paragraph 2.1.

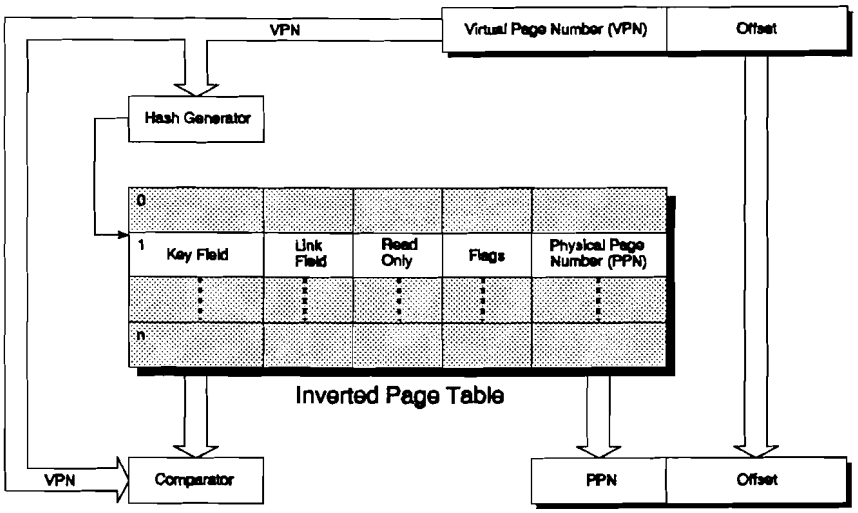


Figure 4.7 Address translation hardware in the MONADS system.

The proposed scheme uses a hash table with embedded overflow to resolve synonyms, held in high speed memory. The hash table is indexed with a hash value generated from the VPN. If the obtained key field matches the VPN, the PPN is used to form a physical address. If a miss occurs, the link field is used to traverse the chain of synonyms. The chain is terminated by an end of chain status bit. When no VPN has been found, a page fault is generated. Insertion and deletion operations are only performed when a page is loaded or unloaded from physical memory. These operations can be implemented in microcode or in the kernel of the operating system.
The hardware address translation unit handles pages which are present in main memory. When a page is not in main memory, a page fault is generated and the page must be retrieved from disk. This is the

second translation that can take place and is based on the virtual-to-disk address mapping of pages. All data in the MONADS network exists in a single virtual address space which is partitioned in such a way that each node is allocated an individual range of addresses. These addresses are guaranteed to be unique across the network by including the owning node number in the high order bits of each address. The range of addresses owned by a node is further divided into volumes. Each volume corresponds to a physical disk, or partition of a disk, located at that node. Finally, to simplify the management of the virtual memory, in terms of both allocation of space/addresses and organisation of page tables, the virtual space is divided into address spaces, identified by address space numbers (note that the term address space used in this context does not correspond to the term address spaces used in the discussion of multiple address space operating systems in paragraph 3.2). Each major object (e.g. program, file, or code module) created in the system is assigned its own address space, which is never re-used for any other object throughout the life-time of the system. Thus the address space number (in fact the node number, the volume number, and the address space number) can serve as a unique object number. Whereas the size of an address space can vary, it is divided into fixed size pages. The full structure of a virtual address is depicted in figure 4.8.

Figure 4.8 Virtual address format in the MONADS system.

Each address space has a separate page table of disk addresses for its pages. The format for these page tables is not fixed because the virtual-to-physical mapping of addresses is separated from the virtual-to-disk location mapping, but might vary according to the virtual memory software.
In the MONADS-PC system the maximum length of an address space is 2^{28} bytes and the page size is 2^{12} bytes. This requires a page table length of 2^{16} entries, which must be placed in virtual memory. Unlike conventional page table entries, the entries in these tables do not need additional status bits. It therefore becomes possible to restrict the size of an entry to a 16-bit relative disk block number, thus allowing a maximum logical disk size of 2^{16} blocks each of page size 2^{12}, i.e. 256 Megabytes. For the node-, volume-, and address space numbers remain a total of 32 bits which is, as was mentioned earlier, not sufficient. In the MONADS-MM system however, there are 32 bits for page number and offset and another 96 bits to identify the node-, volume-, and address space numbers.
In order to maintain the disk addresses of the pages of such a page table, which is called the primary page table of an address space, a secondary page table is required. Even for a full-size address space this secondary page table is relatively small. For performance reasons the secondary page table must be rapidly addressable, therefore it is placed at a pre-defined location in the address space itself (which allows the page-fault handler to form its address directly from the virtual address which caused the page fault). In the MONADS system the secondary page table is placed at a well-defined address in page zero of the address space alongside some other so-called red-tape information which is used by higher level architecture. The primary page table (which can be of different sizes) is placed at the end of the address space and therefore has to grow backwards. Many address spaces may contain only a small amount of data (e.g. small files and programs), therefore to reduce the number of disk accesses to resolve page faults in such cases, the first 256 entries of the primary page table are placed alongside the secondary page table in the first page of the address space. The total address space structure is shown in figure 4.9.

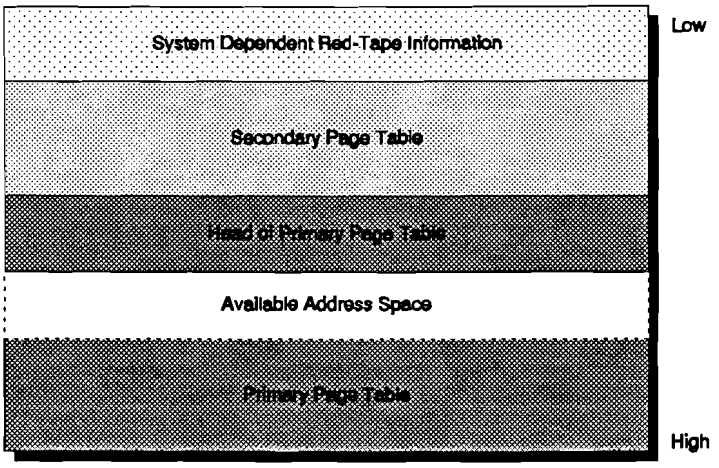


Figure 4.9 Address space structure.

The scheme now operates as follows. When a page fault occurs, the primary page table entry for the page is read. If this causes a further page fault when the required page of the primary page table is not in main memory, the secondary page table entry is read. At this stage it is also possible that another page fault occurs, in which case a separate mechanism must be used to find the disk address of the page containing the secondary page table, i.e. page zero of the address space. To serve this purpose, address space zero on each volume acts as a volume directory containing an entry indicating the disk address of page zero of each valid address space on the volume. Because address space numbers are sparsely distributed and large, a hash table provides an appropriate lookup mechanism for the volume directory. The volume directory also contains a free space bitmap which indicates unallocated disk blocks. The organisation of the data section of the volume directory structure is depicted in figure 4.10. The overall structure of the volume directory is the same as that of other address spaces.

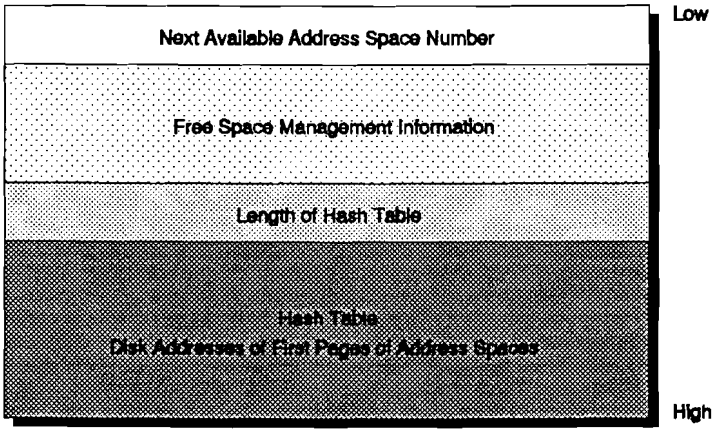


Figure 4.10 Volume directory structure.

Page zero of address space zero (also called the root page) has to be placed at a well known location to be able to locate the secondary page table of address space zero, therefore the MONADS kernel automatically reads and locks into memory page zero of address space zero for each mounted volume. A local mount table contains these volume numbers of mounted volumes and the corresponding root page addresses.
Another data structure worth mentioning is the main memory table (MMT). The MMT is maintained

on a global basis by the kernel and is locked into main memory. It contains one entry for each page frame of main memory. This entry contains the VPN corresponding to the page frame and the disk address of the virtual page or indicates that the page frame in main memory is currently unallocated. The MMT is used, among others, for the allocation of page frames. The disk address is used when a modified page is discarded to avoid accessing the page table again. The format of a MMT entry is depicted in figure 4.11.

Figure 4.11 MMT entry format.

When a page fault is generated, the node number field of the faulting address is examined to determine if the volume containing the required record is stored on the local node [Abra85]. If it is, the page fault is resolved by copying the faulting page into physical memory, and the waiting process is reactivated. If the page is stored on another node, the kernel transmits a message to the remote node requesting a copy of the page. The system supports mechanisms that allow for movement of objects between volumes, and of volumes between nodes. These extensions to the structures described here are explained in paragraph 4.2.3.
Each node in the network maintains an exported pages table (XPT) and an imported pages table (IPT) [Hens90]. These tables have the same structure and contain one entry per exported/imported page with following fields: node number, volume number, address space number, within address space page number, and access rights. These tables are used to implement a specific memory coherency algorithm. The algorithm used is a single writer, multiple reader protocol. Before an owner node brings a page in from disk to resolve a page fault, it must ensure that no read-write copy of the page exists at another node. This check is done by referencing the XPT. If a read-write copy does exist at another node, the other node must mark the page as read-only in its ATU and send a copy of the page back to the owner node. The page fault at the owner node is then resolved using the copy of the page returned by the other node rather than by disk access.
Pages are always exported with read-only access rights, and an importing node must request promotion to read-write access rights, if necessary, as a separate operation. Multiple read-only copies of a page are allowed to exist in physical memories of nodes in the network. If a read-write copy exists in the physical memory of a node, it must be the only instance of the page in physical memory network wide. When an owner node receives a request for promotion of a page to read-write access, it must use the XPT to ensure that any other copies of the page are removed from the ATUs of all but the requesting node before granting the promotion request. The protocol described here is the write-invalidate protocol mentioned in paragraph 3.1.4.

4.2.3 Protection

The most natural and attractive protection mechanism for object-based systems is based on capabilities. The MONADS system provides dedicated architectural support for a capability scheme used to efficiently address and protect objects in its large virtual memory.
Capabilities in the MONADS system have the following properties:

- They contain a virtual address to identify the object to which access is provided.
- These identifiers are unique, they are never re-used to refer to another object (unambiguous).
- They contain a list of operations the holder may invoke on the object.

- Several capabilities per object may exist to facilitate sharing, possibly with different access rights.
- The contents are unforgeable in an unauthorized way because they are protected by architecture.

A unique characteristic of the MONADS design is the implementation of two forms of capabilities corresponding to two quite separate levels of implementation. The first of these, the segment capability, implemented with hardware support, is used to address individual segments. Segments are implemented to represent small objects like integers or resources. The second, the module capability, is fully supported in microcode and is used to represent major software resources such as programs, files and operating system entities. An address space is usually large enough to contain all segments of a module.

The reason for the distinction between capabilities for small and large objects is that the frequency and style of access varies between these two objects and this separation allows for an efficient implementation of small objects without comprising flexibility for large ones. The architecture does not enforce any minimum size rules and any object may be modelled as a module or a segment.

Both module and segment capabilities are protected by architecture so that it is not possible for a process to manufacture or modify a capability in an unauthorized way. A capability is provided by the system when a new object is created. The only other way to gain access to an object is to be given a capability by another process.

Segment capabilities are not visible to programmers or end users. They are collected in special lists which define the segments from which a module is composed. To a programmer it seems as if memory is divided into arbitrary sized segments accessible through different operations. The segment addressing environment of a process is depicted in figure 4.12.

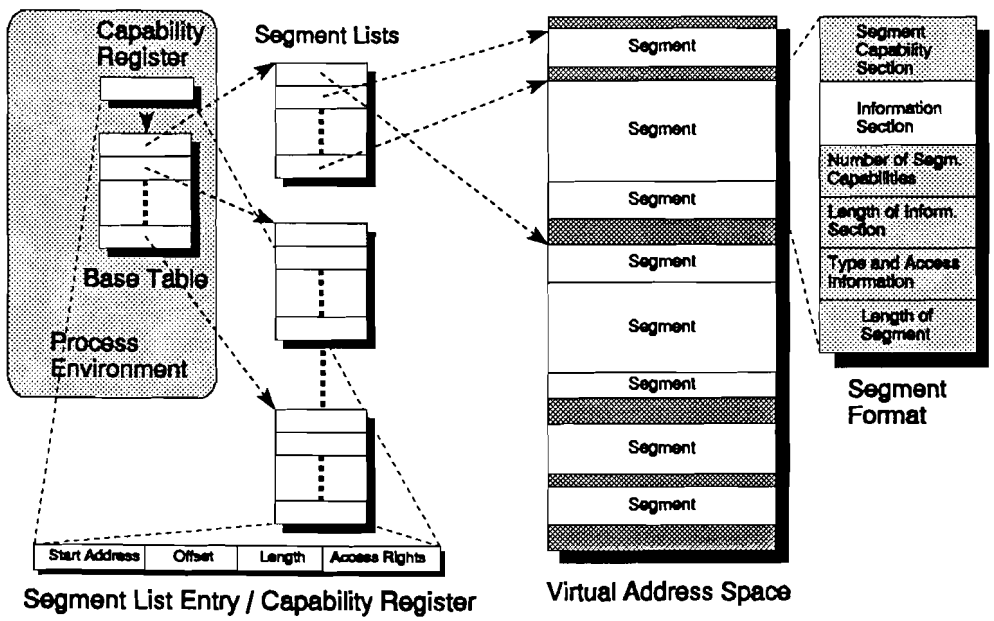


Figure 4.12 Segment addressing environment of a process.

The root of all addressing is defined by a special segment, of which there is one per process, called the base table. This table contains pointers to so-called segment lists which identify the segments this process may access (e.g. data, stack, code, etc.). To provide efficient addressing a number of capability registers is provided which can be loaded with segment list entries. A byte in a segment is accessed by supplying a capability register number together with an offset. A segment list entry contains the start address, the length of the segment, the type of the segment, and the access rights to the segment. The length of the segment is provided for array bounds checking so that continuous segments can not

be addressed with one segment capability. The base table itself is also pointed to by a capability register.
The base table and segment lists are placed in normal segments and thus have the same self-describing format as other segments. The control section defines the size of the segment and details about the content of the information section. Segments may contain module capabilities, one of a small number of system defined types, or arbitrary data. The architecture restricts access to the information section appropriately for the type it contains. The capability section may contain references to other segments.
A module capability uniquely identifies the object by the appropriate module name (node-, volume-, and address space number) and contains access rights indicating the interface procedures of the object which may be called by the presenter of the capability. The format of a module capability is presented in figure 4.13.

Figure 4.13 Module capability format.

Module capabilities are protected by being held in separate segments which may appear in any user object. Thus user objects may contain their own module capabilities and can thus define their own calling environment. What prevents them from being modified are the access rights in the segment capability by which they can be addressed. These define the content as module capabilities and specify which operations can be carried out on them.
Directory objects that are used to map object names represented as a string of characters, onto a module capability, can be supported as normal objects in the MONADS system. Module capabilities can be stored in user accessible segments and can be manipulated with a defined set of safe operations, including transfer between segments, duplication, etc.
As was mentioned before complete volumes and single modules may be moved between nodes. To keep the same name for moved volumes and modules two extensions are necessary. One for moved volumes and one for moved modules [Broe87].
For moved volumes there is a two step solution. The first step is to upgrade the information in the local mount table to contain a mapping from <unique volume number> to <actual disk, address of first block>, where unique volume number consists of the two parts <unique owning node number, relative volume number>. This allows a processor to recognise foreign volumes mounted on its local disks. When attempting to resolve a local page fault the system can then compare the unique volume number in an address with the unique volume numbers in the local mount table. This allows page faults for all locally mounted volumes to be serviced.
To resolve page faults for disks mounted at some other node, a foreign mount table is needed. The foreign mount table contains a mapping from <unique volume number> to <node where mounted>, thus allowing the page fault handler to direct a request for the page to the appropriate node. Volumes which have been mounted on their home nodes do not appear in the foreign mount table for efficiency reasons. Thus the assumption is that a volume not listed in the table is either mounted at the node whose unique node number appears in the address or is not mounted at any node.
For moved modules to keep the same name when moved there is a three step solution. These three steps are:

1. The hash key used to locate objects (i.e. module) in a volume directory from a relative address space number (relative to that node) is extended to a full object number (node-, volume-, and address space number). Objects which move retain their original full object number and at the old directory entry of the volume from which the object was moved is placed a forwarding address (i.e. the new node and volume number).

2. Module capabilities may hold additional (advisory) information. This is necessary in cases where the old volume ceased to exist by being removed or failure. In that case the forwarding address is not available. Therefore the advisory field consists of <node number, volume number> and can be freely changed by the owner of the capability.
3. Each node contains a moved object table mapping moved objects to new locations. This table is filled with the advisory fields from the extended capabilities allowing the page fault handler to access this information. An entry is set up the first time a page of an object is accessed. The page fault handler examines the moved object table before accessing its local mount table and the foreign mount table. This ensures that a locally resolvable page fault is caught before the rest of the network is invoked.

4.2.4 Communication Security and Authentication

In conventional systems user authentication is usually carried out by checking the user's password against a file of passwords. Disadvantages of this approach are obvious. First, the intruder already knows what kind of security procedures are active. Second, because all security information is stored centrally and accessible by one or only a few persons (usually supervisors), the intruder has a clearly defined target to obtain secret information of all users in the system.
Weaknesses of the password system are avoided in the MONADS system by giving the user the responsibility for its own security. When a user wants to log out, a security module is activated which deactivates the user process. When restarting the user process, the security module is entered again and user programmed security procedures are invoked (e.g. different questions, several passwords, etc.). The intruder does not know what kind of security measures he/she encounters and getting access to the user's working environment has thus been complicated.
Subject authentication, verifying the presenter of a capability, can be carried out by calling unprivileged machine instructions returning the identity of the caller. Subject identification may be necessary to implement controlled capabilities (see paragraph 2.3.1.2).
Both user and subject authentication rely on correctness of hardware and kernel. As was already mentioned before, this correctness can not be assumed in a distributed system. Therefore, techniques are needed to increase trustworthiness of kernel and hardware.
For users working in a distributed environment it is usually impossible to make sure that the system has correct functionality and does not leak any information. The only thing users can do is rely on the operating system and hardware producers. However, operating system and hardware producers cannot take any responsibility that the system, once delivered, works correctly and is not modified in any way. Needed is an installation procedure which ascertains the integrity and authenticity of the running operating system and underlying hardware. Such a mechanism has been described in appendix B together with a mechanism to support secure page transfer from (to) disk to (from) memory. The mechanism described in appendix B is a general mechanism that has been implemented in the MONADS system [Frei90].

4.3 Discussion

In this paragraph some important aspects of both operating systems will be discussed. In paragraph 4.3.1 we will be looking at architecture and addressing while in paragraph 4.3.2 protection will be discussed. The discussion is by no means complete in the sense that it will cover all issues related to addressing and security. However, most important issues will be highlighted.

4.3.1 Architecture and Addressing

Both systems are distributed, but the Opal system is a distributed operating system while MONADS is a networking system. The differences between these two types of systems were discussed in chapter 2. Summarizing, a distributed operating system has more fault tolerance and transparency than a networking operating system which consists of nodes running their own copy of the operating system. A reason for developing a networking system is the fact that it is relatively easy constructed from a centralized one by adding an extra module which handles communication. The MONADS system was originally designed as a centralized operating system and was extended to work in a distributed environment.

The size of the network in Opal is limited to a few tens of nodes because of the 60-bit addresses. In MONADS-MM, with 128-bit addresses, the size is limited by the 32-bits node number. This node number permits up to 2^{32} nodes but in such a large network, network latency is becoming a restrictive factor. However, with the oncoming fibre optics networks, huge single virtual address space networks can be created using the MONADS hardware.

The most important difference between Opal and MONADS is the specially designed hardware in the MONADS system compared to the standard architectures used by the Opal operating system. Opal is currently being prototyped running above an unmodified Mach 3.0 kernel [Jone86] on MIPS based workstations [Kane92]. However, it is expected to run efficiently on MIPS R4000 or DEC Alpha based stations. It would also be possible to run the Opal system on existing architectures with support for multiple address spaces but in doing so this might incur unnecessary performance costs. The Alpha processor and the MIPS R4000 are well suited for Opal because of their software loaded TLBs as was explained in paragraph 3.2. The MONADS system is running on specially designed hardware because capability based operating systems tend to suffer from lack of performance on standard architectures. The advantages of this specially designed hardware are:

- Support for very large address spaces is possible. The MONADS-MM system permits up to 128-bit addresses.
- High speed of operation because hardware can be optimally designed to support the operating system.
- Hardware support for capabilities was implemented.

There are also some disadvantages of this approach:

- It is expensive when designed for commercial usage.
- It is not likely to be used in general because of incompatibility with other systems. However, incompatibility may be overcome by emulation.

Another notable feature of both systems is their object-based structure. For MONADS this was the logical consequence because capability based architectures are object-based by definition [Levy84]. For Opal however, an object-based approach is purely a policy. An object-based data model is not necessary to support sharing above a global virtual store. However, the designers of Opal believe that object-based data models continue to be valuable for organizing and controlling sharing. Domains sharing data must have a common understanding of its internal structure and usage, best formalized as a common set of types. The purpose of Opal's object support package is to allow the code that implements object methods to be located and bound dynamically given an object reference, and to ensure that all domains operating on the data use the same implementations of those types. In addition, objects are a convenient granularity for distributed coherency, stable storage updates, and garbage collection.

Both systems are object-based, however the way in which they name and address their objects differs. In Opal objects are referenced by their virtual address (the virtual address of the first byte of an object) either directly by specifying a flat virtual address or indirectly by using a protected pointer. A symbolic name space for data exists above the shared virtual address space. This is realised in the form of a name server that associates symbolic names with segment capabilities or arbitrary pointers. This

does not hold for services, which have to be handled differently from data. Protected services are called through protected RPC interfaces. Opal makes no distinction between data and services, however, to the client data object pointers are pointers that have to be treated differently from capabilities. To Opal an object is a protected object only from the perspective of a client that is insufficiently privileged to attach the object and invoke it with an ordinary procedure call. The advantages of uniformity between data and services are:

- Any number of service objects may be clustered within the same protection domain and served through the same portal, reducing the cost of protection and communication especially in systems with large numbers of service objects.
- Protected pointers are context independent and can be passed between domains in shared memory.

In the MONADS system naming is done differently from naming in the Opal system. Directory objects, which are used to map an object's name, represented as a string of characters, onto a module capability can be supported as normal objects in the MONADS system. Module capabilities can be stored in user accessible segments and manipulated with a defined set of (safe) operations, including transfer between segments, duplication, etc. The standard MONADS directory object is actually a type manager which can be used to generate many individual directory objects. As a directory is an object itself, a capability for it may be inserted into another directory. Therefore directory objects may be individually programmed and used to construct arbitrary tree and graph structures.
Because in both systems flat virtual addresses are used, transparent object relocation is rather difficult to implement. Object relocation can be desirable for performance, security, or reliability reasons. There are two solutions to solve this problem:

1. Using one or more levels of pointer indirection for object addressing.
2. Leaving forwarding records for objects that have moved.

However, both solutions introduce new problems and add to the cost of pointer dereference. This is one of the reasons why object relocation has not yet been solved for the Opal system. The MONADS system uses solution 2 to attack the problem. The mount tables containing forwarding information were described in paragraph 4.2.3. This results in slight overhead when the object is locally available (the moved object table is referenced) and more overhead in case the object has to be brought in from another node.

4.3.2 Protection

At the hardware level the systems differ greatly because both use different architectures. A major difference is the lack of protection information in the ATU of the MONADS system in contrast with the PLB or page-group mechanism. This is a result of the capability based architecture of MONADS. The problems common to a capability based architecture are [Fabry74]:

- Restrictive object-based data model.
- Non-competitive performance.
- Lack of support for distribution.

In the MONADS system these problems have been well overcome. The restrictiveness to the object-based data model is not very significant because in single address space systems object-based models are useful to support sharing (they provide the necessary abstraction) as was explained in chapter 2. Performance measures of both systems are not available yet but the MONADS system should be competitive with the Opal system because of the dedicated hardware used in the MONADS system. Lack of distribution is overcome by implementing an extra layer above the centralized operating system and making some minor changes to this operating system. A disadvantage of the MONADS capability approach is the two-level capability scheme. This was implemented for performance reasons but does so at cost of user transparency. The decision of the capability format to be chosen has moved towards responsibility of the user.

While both operating systems use capabilities, there is a great difference between the used types of capabilities. The Opal system uses sparse capabilities (comparable to the capabilities used in Amoeba [Mull86]) while in the MONADS system, kernel controlled capabilities are used. Advantages of kernel controlled capabilities over sparse capabilities are:

- Sparse capabilities are not impossible to forge. However, they are difficult to forge because the value of the check field is probabilistically impossible for a client to guess, and there is no way to determine if a guess is correct except by asking the server. However, the possibility of a sparse capability being forged is present. Kernel controlled capabilities are absolutely unforgeable because they are only accessible by hardware mechanisms.
- Kernel controlled capabilities can be reference counted or garbage collected by the operating system.
- Kernel controlled capabilities are protected by the operating system by storing them in separate parts of memory. This technique is called segregation.

A disadvantage of kernel controlled capabilities is that they rely on correctness of the operating system kernel. Therefore the kernel has to be protected from malicious modification. The technique used to realise this was described as secure booting (appendix B) which requires some booting overhead. To protect information that is sent across the network, encryption is used in the MONADS system. Communication security is a problem that has not been addressed yet in the Opal system. However, the mechanism used in the MONADS case is also applicable to the Opal case. The Opal system is less vulnerable to attacks because of the properties of sparse capabilities:

- The server may choose not to grant access even if a client presents a valid protected pointer (capability). This means extra protection, e.g. disabling all accesses to an object immediately.
- Sparse capabilities can be revoked by changing the value of the check field.
- Sparse capabilities may be stored as normal data because they are probabilistically impossible for a client to guess.

The only major threat to the OPAL system are attacks on communication media. This threat could be encountered as was mentioned before by using the secure paging mechanism described for the MONADS system (appendix B).

The last important fact about protection is granularity. In Opal data objects are clustered into segments, allowing domains to attach segments and directly address the objects they contain. There are several trade-offs inherent to this approach:

- Object encapsulation for shared data is enforced by the type system rather than by hardware. The performance win is that language protection is much cheaper than hardware based memory protection.
- Access control is based on segments, not objects. Object grain access control can be bought by placing each object in a private segment, but this is wasteful of memory for small objects because the minimum segment size is a page. However this is not a significant restriction because data objects are usually used in clumps.
- Applications must manage the assignment of objects to segments. This assignment must be done carefully; if two objects reside within the same segment it is impossible to grant attach access to one but not to the other.

In contrast to data objects, service objects are protected at object grain level. They are invoked through an RPC interface that is invoked with protected pointers as its arguments. So data objects are cheap and service objects are protected.

In the MONADS system every object can be protected separately, so there is protection at the object grain level. This is a consequence of the capability based architecture. There are no size restrictions for objects so granularity of protection can be as fine as one byte. A major disadvantage, however, is the fact that in such a system shared data is just as expensive as shared services. Both are protected by capabilities and have the same protection overhead. This is solved partly by implementing two different types of capabilities in the MONADS system. The reason for this distinction is that in this way small objects can be implemented efficiently without compromising flexibility for large objects.

5 Wide-Addressing Concepts

In this chapter we will try to develop a common view on how to implement naming, addressing, and protection in operating systems designed to run on wide-address architectures thereby taking into account the new possibilities these architectures offer. To keep this view as global as possible, first we have to define some requirements the desired operating system should conform to. Then assumptions are made to define our working area to naming, addressing, and protection. When requirements and assumptions have been defined, specific topics in naming, addressing, and protection will be looked at.

5.1 Operating System Assumptions and Requirements

We will first give the requirements "our" operating system should conform to. These are:

- Object-based, the operating system should be constructed from the object abstraction. Arguments for choosing this abstraction are:
  - In an operating system there are a large number of interacting components. When systems evolve during time, the implementation of these components change. Therefore a common way to describe them is needed. A set of interfaces, defining in a standardized way the sets of operations that each component supports is the simplest way to present object-based systems.
  - There is a need for agreement on how shared subsystems will operate. The operating system itself consists of a collection of servers, and applications consist of a set of modules that should be aware of the correct rules for interacting with those services. Programs, regardless of the language used to implement them, consist of glue interconnecting a set of component modules into a coordinated whole, a collection of objects.
  - The abstraction is necessary to design a global view on naming and protection in distributed operating systems.

  Note that the object-oriented abstraction is not needed for this discussion, when needed the object-based approach could be extended to object-orientation.
- Single virtual address space, the operating system will consist out of one address space in which all objects reside. The reason for choosing this address space model was founded in chapter 3. The single address space model is the only address space model that can highly benefit from the advantages wide-address architectures offer (large addressing capabilities). In other words, the advantages of a single virtual address space are being unlocked by using wide-address architectures. This single address space will be a virtual one to be able to benefit from the advantages virtual memory offers.
- Distributed, the operating system we will be looking at will be a distributed operating system. As was stated in chapter 1, a single virtual address space can be mapped onto a network, resulting in a single virtual global address space. In that case, all processors from different nodes in the network share the same address space. This offers new viewpoints in distributed operating system design. The operating system itself will be distributed instead of a networking operating system,

because of the increased level of transparency in distributed operating systems. A distributed operating system is a single network-wide operating system viewed logically as a single operating system that exists for all the distributed computers. Differently stated, we want a transparent operating system, i.e. a user should not know were (and when) his/her files are stored.

- Object grained granularity. This implies that there will be object-grained naming (trivial) and protection. Objects will be named by their unique virtual addresses. We believe that this is the most simple and efficient naming scheme. No indirection is needed and therefore names do not have to be translated before an object can be addressed.
- The operating system should be secure. This statement can be interpreted in several different ways. Here we mean to say that the operating system should provide for some sort of protection against malicious modification of data. Thus, the system should provide for communication security and authentication.

These are the requirements our operating system should conform to. It should be noted that these are only minimal requirements, they could be extended by several other requirements, e.g. object-orientation but such extra requirements are not essential.

Before we can proceed with a discussion of addressing, we first have to make some assumptions about the system. These assumptions are:

- The system is built out of an arbitrary number of nodes, consisting of one or more processors connected to local memory and with possibly some secondary storage attached, e.g. a disk, connected by a network.
- No hardware limitations, our decisions made will not be influenced by any hardware limitations. An example of this assumption is that the underlying network will be considered "safe", i.e. transmitted packets will always arrive in correct form at the correct location and within a predictable time span. Processors, primary- and secondary storage will be assumed fast enough. The reason for choosing this assumption is that "hardware needed today will be available tomorrow", i.e. we do not want to make decisions based on available hardware.
- The operating system should run on wide-address architectures. Assumptions about the architecture are minimal; it should only be possible to implement page table support in software. The reason for choosing virtual paging is rather arbitrary. In fact, memory management techniques used are not important because they only define translation of virtual addresses. However, translation of virtual addresses can be done separately from protection translation which is one of the issues discussed in this chapter.

Now that we have defined "our" operating system, which has been visualised in figure 5.1, we can take a closer look at addressing, followed by naming and protection.

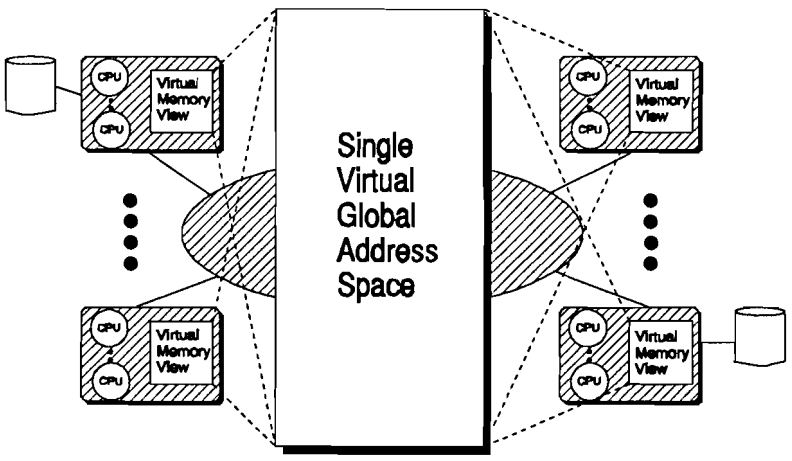
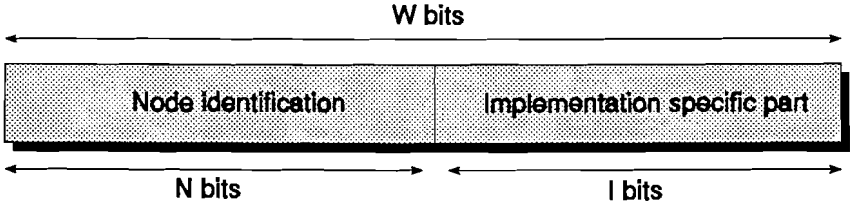


Figure 5.1 Single virtual global address space.

5.2 Address Space Partitioning

We stated that a single virtual address space shared by multiple nodes in the distributed system should be implemented. This results in two general approaches for dividing the address space among nodes. The first approach of address space partitioning is based on a fixed amount of virtual address space maintained by each node. This means that every node provides physical backing (e.g. main memory or disk) of a pre-defined part of the virtual address space. Because the range of virtual addresses per node can be statically determined, a part of the virtual addresses can be used to denote a specific node number. This node number then indicates the owner node of a virtual address. The remaining address bits are used to implement the local address part, e.g. page number and offset. The resulting virtual address has been depicted in figure 5.2.



Virtual Address Space : 2^W
Virtual Address Range per Node : 2^I
Maximum Number of Nodes : 2^N

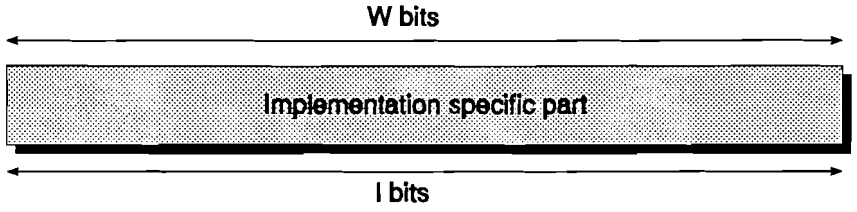
Figure 5.2 Virtual address when the address space has been equally divided among nodes.

Advantages of this approach are:
- Nodes do have an explicit identifier embedded in the virtual address.
- The remote page fault process (also name resolution) can profit from the explicit node identifier embedded in virtual addresses.
- Virtual addresses are made unique in the entire network by maintaining unique virtual addresses within a node.
- Extending the network does not incur problems as long as there is a free node number.

However, disadvantages are:
- The number of nodes is defined by the number of address bits reserved to specify the node number.
- Node boundaries are explicitly visible in virtual addresses. This results in a decrease of transparency. When virtual addresses are used as object names, users know the explicit location of an object.
- The virtual address range per node is fixed. This implies that the virtual address range cannot be adapted to the amount of physical storage locally available to a node. A node that has little memory available will be less efficient in supporting large ranges of the virtual address space.
- The complete virtual address will have to be presented when inter-node pointers are being used. For pointers within a node, only the implementation specific part is necessary. These two possible pointer forms do cause ambiguity.

This approach can be compared two the dual pointers supported in the Hewlett-Packard PA-RISC; a short pointer- and a long pointer form. In this case an inter-node pointer is a long pointer form and an intra-node pointer serves as a short pointer form. For intra-node pointers, only the implementation specific part will be sufficient. Note that this creates similar problems as with multiple address spaces because intra-node pointers are not unique and can thus not be used as unique names. Therefore the long format or inter-node pointer is the only pointer form that can serve as a unique name across

nodes.
The second approach considers the address to be a flat virtual address, i.e. no node information is available within the virtual address. The virtual address format has been depicted in figure 5.3.



I = W
Virtual Address Space : 2^W
Virtual Address Range per Node : variable
Number of Nodes : variable

Figure 5.3. Flat virtual address.

Achieving uniqueness of virtual addresses is realised by address servers at each node that maintain assigned ranges of virtual addresses. The address ranges should have to be assigned by a global address range service. In approach number one, servers had to provide for unique local address parts while in this approach a flat virtual address is used and therefore the complete virtual address has to be unique within a node. Note that in fact both approaches differ minimally in keeping virtual addresses unique. Advantages of this approach are:

- This approach provides for a higher degree in programmer transparency. A programmer, that has no access to address server information, is unable to detect where an object has been located.
- The virtual address ranges can be adapted to locally available storage. Nodes that do have large amounts of primary- and secondary storage could be assigned a larger range of virtual addresses. When this assignment is made depends on the implementation. It seems most convenient to assign these address ranges at the moment a new node is appended to the network because changing them at an arbitrary moment requires objects to be moved among nodes (when large numbers of objects are available this is a very time-consuming process).

Disadvantages of this approach are:

- At each node, an address server maintains the assigned virtual address range. Extensibility of the network decreases because the virtual address space could be fragmented at the moment of assigning an address range to a newly attached node.
- Remote page fault handling and name resolution has been complicated by the node number not being available in the virtual address. Therefore, name resolution mechanisms as were described in chapter 2 have to be used by the address servers, e.g. an address server can do bounds checking on the virtual address and when not successful pass the resolution process to another address server.

The two approaches described have been applied in a similar manner in Opal and Monads. In Opal the second approach is used and the entire address space is divided into large ranges of virtual addresses that are assigned to nodes. Servers at each node divide these address ranges into virtual segments (paragraph 4.1.2).
The MONADS system uses the first approach. A virtual address is divided into a node number and a local part consisting of volume-, address space-, page-, and node number.
Based on the increase in transparency the second approach is being preferred. The programmer can use virtual addresses of data without knowledge of the location of accessed data. This meets one of the goals of a truly distributed operating system; it should appear as a logically single system for all

connected nodes.
Also, only one pointer format is being used in this approach. This can result in increased code size compared to the partitioned virtual address format can be used but also increases uniformity.

5.3 Protection

To discuss protection, we will use the protection domain abstraction. The protection domain abstraction provides a handle to implement access to virtual memory. A protection domain specifies the objects a subject is allowed to access along with the associated rights. Informally stated, a protection domain defines (shared) data, code, and stacks an application can access. A protection domain may thus contain several objects. Some properties of protection domains are:

- They are of arbitrary length.
- They may overlap, i.e. an object may be contained in several protection domains with several different access rights.

As was pointed out in chapter 4, it is possible to separate protection and translation issues in a single virtual address space. Translation issues will not be considered here because conventional mechanisms can be used when stripping these mechanisms from protection information. Hence, a conventional page table is stripped from protection fields in the entries. This equally holds for inverted page tables and translation lookaside buffers.
To discuss protection apart from translation we need another abstraction mechanism. This mechanism will be that of regions. Regions will be windows in the virtual address space that define the access rights of the address space part covered by the window. Then, to protect objects, regions could be mapped onto objects.
Based on this region abstraction, protection domains now consist of several regions. Protection domains can grow by attaching a region and shrink by detaching one. Attaching and detaching allow for some administrative information being set up by region servers. Region servers are servers managing regions. Attaching can be seen as making a region active while detaching can be seen as deactivating a region. Inactive regions, i.e. regions that have not been attached by any protection domain at all do not provide any function so they can be discarded. Therefore a region starts to exist when it is being attached and ceases to exist when being detached.
Regions can also be shared by different protection domains. This allows for sharing data with equal access rights for sharing domains, i.e. the region covering the data is being attached to both protection domains. When sharing data with different access rights by several protection domains, regions need to overlap. One region covering the data is attached to a certain protection domain while another region covering the same data is attached to another protection domain. Both regions then do have different access rights associated to them.
Concluding, some characteristics of regions are:

- Regions are arbitrarily sized windows in the virtual address space.
- Regions start to exist when being attached to some protection domain and cease to exist when being detached.
- Regions are most likely sparsely distributed over the single virtual address space when mapped onto objects.
- Regions do have one and only one specific set of access rights associated to them.
- Regions may be shared by protection domains.
- Regions can overlap when equal data covered by the region has been attached to different protection domains with different access rights.

The relation of regions to protection domains has been visually presented in figure 5.4. As one can see in this figure, region D is being shared by protection domains 2 and 3. This implies that both protection domains require equal access rights for data covered by region C. However, when protection

domains do need different access rights for certain covered data, different regions have to be created. This is illustrated by regions B and C in figure 5.4. The same data has been attached to protection domain 1 as being region B while attached to protection domain 3 as region C. Regions B and C do contain equal access rights else the same region could have been used.
Because regions do not have any correlation to underlying memory management, protection and translation can be separated as was stated before. Therefore, different granularities of translation (e.g. virtual page) and protection (e.g. byte) can be used. In the next paragraph an equivalent scheme of the protection lookaside buffer as implemented in the Opal system will be described.

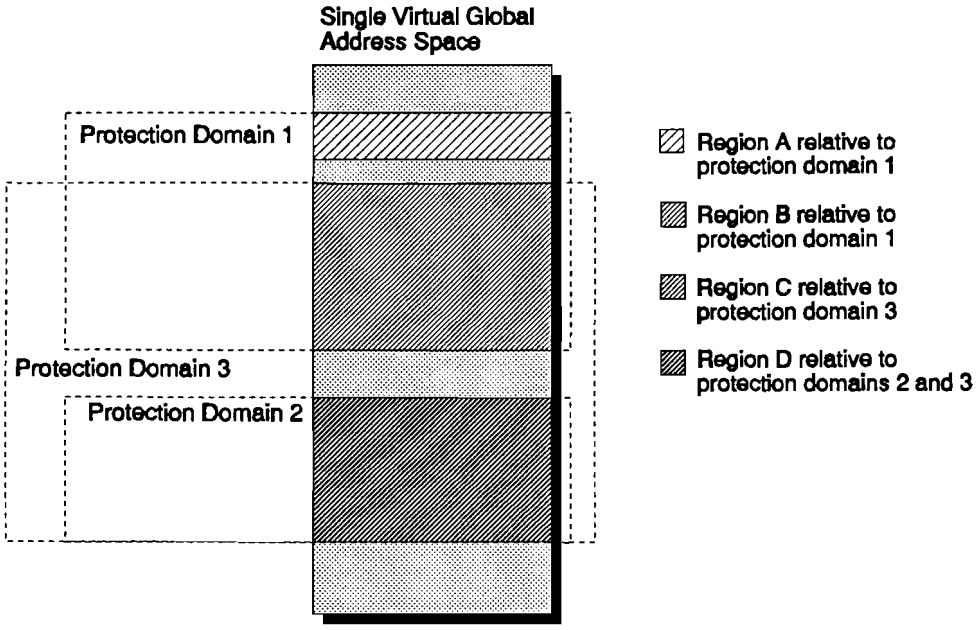


Figure 5.4. Protection domains consisting of several regions.

5.3.1 Low-Level Protection

In this paragraph we will look at protection at a level very close to the hardware. As was described in paragraph 4.1.3.1 for the Opal system, protection and translation issues can be decoupled. This decoupling provides for parallel processing of both translation information- as well as protection information lookup. Another advantage of this decoupling of translation and protection is that different granularities can be used for both issues.
We will now describe the consequences of the region abstraction for the PLB concept which was introduced for the Opal system. To perform protection checking an information structure is needed which provides for:

- The protection domain(s) to which a (shared) region has been attached. When a region has not been attached to any protection domain it is inactive and hence does not exist.
- The access rights associated to a region.

This information structure will be called a region table. This region table can be seen as the information missing from a conventional page table which has been stripped from protection information. In this way protection and translation information have been clearly separated. The stripped page table can be extended with a stripped translation lookaside buffer to increase performance while the region table can replace the protection lookaside buffer of the Opal system. This

adaptation provides for a complete separation of translation and protection. A virtual page serves as translation granularity because virtual paging is used while one byte, i.e. the smallest region, is the granularity of protection. It should be noted that granularity of translation can easily be different by implementing other memory management techniques (e.g. segmentation). In the Opal PLB system the granularity of protection is a virtual page. Therefore objects smaller than one virtual page cannot be protected separately without waste of space. Thus the major advantage of the region abstraction is the finer granularity of protection which becomes possible.
Processes needing access to the region table are:

- Instruction execution, the appropriate protection information belonging to some virtual address reference has to be determined. Hence, the region table is needed to provide the necessary protection information when executing instructions.
- Attach and detach operations to be performed by region servers. Therefore the region table could best be maintained within a region server.

To provide for the appropriate protection information when indexed with a virtual address, the region table should contain entries as presented by figure 5.5.

Figure 5.5 Region table entry.

The region identifier is needed to determine if the presented virtual address is part of the specified region. Note that a virtual address can be used as region identifier. The owner protection domain identifier indicates the protection domain to which the region has been attached and the access rights determine the allowed operations to be performed on data covered by the region.
The attach operation requires specific entries in the region table to be changed when an existing region is attached to another protection domain, or to be added when a new region is created by attaching it to a protection domain. Detaching requires specific entries to be deleted because a region ceases to exist.
The specific implementation of the virtual address-to-region resolution process is being influenced by another specific assumption about regions. When regions are considered to be contiguous sequences of virtual addresses, a region table entry could also contain the region length to provide for region bounds checking. In that case, when presented a virtual address, the region server could perform bounds checking of regions until the region that encompasses the virtual address is found. However, we do not want to restrict ourselves to regions being contiguous ranges of virtual addresses. As an example of this restriction think of an object that is distributed among nodes covered by a region. This could result in a non-contiguous region. Of course, this could be solved by mapping different parts of an object onto different regions with equal access rights. This is merely an implementation issue and will not be considered here.
The total protection system has been depicted in figure 5.6.

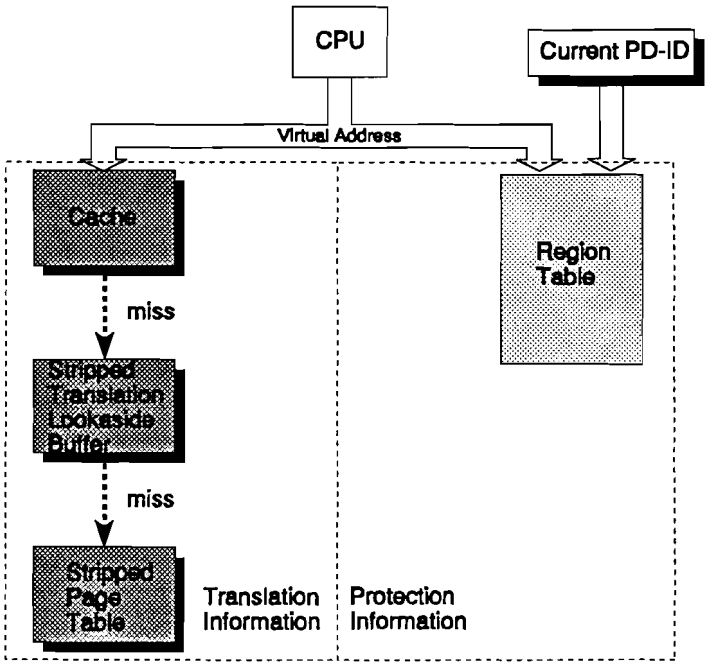


Figure 5.6 Protection system based on regions.

The operation of this adapted PLB system is as follows:

1. On each memory access, the cache is indexed. This cache can be a virtually indexed, virtually tagged cache as is used in the Opal system. Therefore cache lookup can occur before address translation needs to be performed.
2. In parallel to step 1, the region table is indexed. This region table is indexed with the virtual address and the current protection domain identifier held in a separate processor control register. A region table entry is looked for, which contains an owner protection domain identifier equal to the current protection domain and contains the region of which the virtual address is part of.
   When the current protection domain identifier does not correspond with the owner protection domain identifier an access to an unattached domain (attached to another than the current protection domain) is attempted and therefore a protection exception can be generated.
   If a virtual address is presented which does not occur in any region, the attempted access is also illegal.
   Both illegal attempts prevent the TLB or page table from being accessed when a cache miss occurred.
3. When the protection lookup succeeds, the access rights determine if the operation is allowed to proceed. In this case a cache miss results in the TLB (and possibly the page table if a TLB miss occurs) being accessed. Note that the TLB and page table can be replaced by an inverted page table. However, this is an implementation issue.

Shared regions result in duplicate region table entries (identical region identifiers) with different protection domain identifiers. Translation information is not being duplicated.
Concluding, results of the separation of protection and translation are:

- Translation information- and protection information lookup can proceed in parallel.
- Shared regions do not require duplicate translation information to be stored as with conventional page tables.
- Granularity of protection can be as fine as one byte, thus different from the granularity of translation.

5.3.2 High-Level Protection

Until now we have described protection only at a level very close to the hardware. The protection information at this level is provided by high-level protection mechanisms, i.e. the protection mechanism that is visible to the users. In this paragraph we will try to create a high-level protection mechanism. There are a couple of design principles for high-level protection mechanisms that we have to consider:

- Economy of mechanisms, use the simplest possible design to achieve the desired effect.
- Fail safe defaults, access decisions should be based on permission rather than exclusion, i.e. arguments must be made as to why objects should be accessible rather than why not.
- Complete mediation, every access to every object must checked using an access control database for authority.
- Open design, mechanisms should not depend on ignorance of the attacker but rather on possession of protected information (this principle does not have to hold for very sophisticated protection mechanisms as are used within military systems).
- Least privilege, subjects should be given only those access rights that are necessary to perform a task. This principle serves to limit the damage caused by accidents, errors, or malicious activities within a computer system.
- Acceptability, the mechanism should be easy to use.

The higher level protection mechanism we will be discussing is the access matrix model because of its widely spread usage in distributed systems. As was described in chapter 2, there are two possible ways to split up the access matrix, i.e. to create efficient implementations of it. The first way to split up the access matrix is by columns resulting in access control lists and the second one is by splitting it up in rows which results in capability lists. The arguments summarized below indicate why the capability approach has been chosen as the protection mechanism used:

- Capabilities are directly associated with names. They combine naming and protection. Because the virtual addresses can be used as identifiers, one problem when using capabilities, that is providing unique object identifiers, has been overcome.
- Capabilities provide for the principle of least privilege. Each subject can be given a personal capability. The capability mechanism provides open design, and fail-safe defaults, i.e. access is based on permission. They do not automatically provide complete mediation but that will be provided for by using controlled capabilities. Controlled capabilities can only be accepted if from a legitimate holder.
- Capabilities are a logical extension of object-based systems. Associated to each object are tickets (capabilities). Each subject has or has no tickets to access objects.

Before discussing various capability formats a list of requirements our capabilities should conform to is given:

- Subject verifying. When presenting a capability, the object server does not verify the identity of the holder in conventional capability systems. Controlled capabilities require the object server to base its decision on the holder's identity. We use the term object server to denote the server that manages access to a collection of specific objects, e.g. a file server.
- Revocation should be possible. It should be possible to invalidate all or a subset of the capabilities in the system.
- Propagation of a capability. A major advantage of the capability mechanism is that objects can be shared by simply passing the capability. This property should be maintained as much as possible. Note that it contradicts with the principle of controlled capabilities because a controlled capability cannot be passed without awareness of the object server.

We will now construct, step by step, a capability mechanism that is conform our requirements, starting with the basic capability format. While refining the constructed capability mechanism we will look at some other capability formats for their strengths and weaknesses.

The basic capability format has been depicted in figure 5.7.

Figure 5.7 Basic capability format.

This is the basic capability format, containing a name for the object to be accessed, a name to address the object server, and the access rights. Shortcomings of this kind of capabilities are:

- Unlimited propagation is possible, i.e. capabilities can be passed among subjects when servers do not explicitly test the subject presenting a capability. These kind of capabilities are also called uncontrolled capabilities.
- Forgeary, the holder of a capability can change its rights.
- No provision for revocation has been built in.

Solutions for these problems depend on the type of capabilities used (see also chapter 2):

- Tagged capabilities. This kind of capabilities is not considered here as it requires special hardware support, e.g. extra bits for every byte in memory to distinguish between capabilities and "normal" data.
- Partitioned capabilities, require special hardware support to be implemented efficiently (as in MONADS) resulting in a segregated address space. Another disadvantage is that possibly undesired profileration of objects occurs.

Both types of capabilities described above are unforgeable because they are protected by hardware or kernel support. Because the kernel is involved in most capability operations, propagation and revocation are not hard to implement. A major disadvantage is that hardware support is needed to implement those capabilities. Hence, conventional wide-address architectures cannot be used.

- Sparse capabilities, these capabilities are most suited to distributed systems for the following reasons:
  - Sparse capabilities do not have to be distinguished from normal data, they can be stored along with other data.
  - A trusted mechanism can be constructed based on sparseness of some object identifier space. In single virtual address spaces, the virtual address space can serve as the sparse object identifier space when large enough. Hence, wide virtual addresses can serve as sparse object identifiers.
  - Sparse capabilities can also be used to name objects, they are bit strings of a defined length.

Sparse capabilities are thus well suited for use in distributed systems but they need additional mechanisms to protect them from being forged, and to implement propagation and revocation. We will now look at some sparse capability formats that bare resemblance to capabilities constructed out of a virtual address that serves as a sparse identifier.

The first capability format we will be looking at is the Lawrence Livermore National Laboratory (LLNL) capability. This format has been depicted in figure 5.8.

Figure 5.8 LLNL's capability format.

The server address field is used to address the resource server. The property field contains some standard bits and fields to indicate resource type, access mode, resource lifetime, and security level.

These bits are added to reduce the number of messages sent to the server, hence they are not indispensable. The unique local identifier is comparable to a virtual address used in a certain context. This unique local identifier has been made sparse, i.e. without knowledge of this identifier, a process cannot access the named resource. Hence it could be replaced by a global unique identifier, e.g. a unique (in the context of the entire system) sparse virtual address. The redundancy field guards the unique local identifier against forgeary, e.g. by use of encryption. The capability will only be accepted if the redundancy field or password is correct.
The second type of capability we will be discussing is the Amoeba capability [Mull86] that served as a foundation for the Opal capabilities. The format of the Amoeba capability has been depicted in figure 5.9.

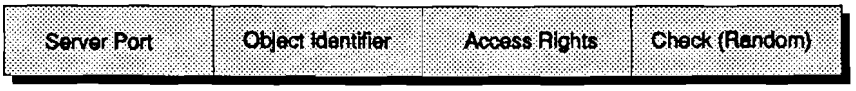


Figure 5.9 Amoeba capability.

The server port contains the address of the object server and is a sparse identifier randomly chosen from a large identifier space. This is in contrast with Opal in which the server (or portal) identifier consists of the unique virtual address of the server. The object identifier is a local unique (in the server context) identifier. In Opal this field again contains a unique virtual address, that is the virtual address of the object. The access rights field contains one bit for every possible operation and is protected by the check field. When a capability is to be created, the server picks a random number and stores it in its object table. Next, the access rights field is concatenated with the known random number in the capability that makes the capability sparse whereafter this newly formed number is encrypted by the random number stored in the object table. The encrypted value is put into the combined access rights and random field.
When an access is required, the server decrypts the combined access rights- and check field by using the appropriate key. This key is the random number retrieved from the object table that is indexed by the object identifier. When the decryption leads to the known random number field, the capability is assumed to be valid.
This system also allows for passing of capabilities. A subject that wants to pass its capability sends a request to the server together with the rights that may need to be changed. Then the server changes the rights field if necessary, and encrypts it with the key associated to the specified object. Drawbacks of this approach are: server interaction is required every time a capability with altered rights is passed, there is no subject authentication, and only global revocation is possible, i.e. all capabilities for one object can only be revoced at once.
To avoid the drawback of server interaction, a second approach has been made in the Amoeba system. Associated with each bit in the rights field is a commutative one-way function. A one-way function F(x) has the property that given x it is easy to determine F(x) but given F(x) it is nearly impossible to determine x. When an object is created the same operations are performed as in the former approach. However, both the rights field and the random number field stay initially unencrypted. When a client wants to delete permission p presented by bit k of the access rights field, the random number R in the capability has to be replaced by $F_k(R)$ and the rights field has to be adapted to present the current permissions, i.e. bit k of the rights field is made zero. When the capability comes back to the server, the server fetches the random number from its object table and applies the one-way functions according to the changed permissions in the rights field. The order in which these one-way functions are applied does not matter as they are commutative functions. If the deduced random number is equal to the random number stored in the capability, the capability is accepted.
In both approaches revocation is possible by changing the random number stored in the object table.

However, this is only global revocation. Selective revocation, i.e. revocation of capabilities belonging to a specific subject, is not possible. Propagation of capabilities can be controlled in the first approach but at cost of server interaction.
Both described capability mechanisms are uncontrolled, i.e. the holder of a capability remains unchecked. To protect controlled capabilities the next three methods can be used:

1. Access lists, the server maintains a list of object-subject relations.
2. Encryption using the legitimate holders address or another unique attribute as part of the capability.
3. Capability lists named by the origin address. The server uses the origin address or identifier of the subject to address a capability list. If the presented capability occurs in this list it is accepted.

Controlled capabilities offer more security but do that at cost of more server interaction. It should also be noted that masquerading is still possible, subject B can still pretend to be subject A. To prevent this form of attack, an authentication procedure could be used.
We will now construct "our" capability format based on the requirements stated before. We already have a capability consisting of the unique global virtual address of the object server, an unique global virtual address of the object, and a rights field. Note that this format is made possible by the large single virtual address space, virtual addresses can serve as unique object identifiers. Therefore no extra identifier space, as used in Amoeba, is necessary.
As was stated as one of the requirements for a capability mechanism, subject checking should be provided for when a capability is presented. The subject information can be maintained in two places; the first place being the capability itself (option 2 of protecting controlled capabilities) and the second place being the server (options 1 and 3). The server as location for this information is the preferred one as the server has to be assumed secure anyway or else the encryption mechanism is not solid (the server maintains the keys used for encryption). Another advantage of this location is that the capability length remains the same. Hence, two options to implement controlled capabilities remain; access lists or capability lists. In fact both options do not differ significantly. What is needed is an object-subject relation list. Given a capability presented by subject S, the server has to determine if this capability has once been provided to S.
The capability can be protected almost similarily to Amoeba's first approach. The only difference being that instead of using random numbers in the capabilities and as keys, nonces should be used to avoid duplicate random numbers. A nonce is an unique identifier that is used only once, e.g. a timestamp.
The server now maintains per object a list of subjects having a capability for this object, a nonce associated to this type of object, and a nonce serving as a key. When a capability is presented the server indexes the object list and verifies the calling subject. If the subject does appear in the object list the encryption key is fetched else the request is rejected. The combined access rights and check field is decrypted whereafter a comparison of the nonces proves, or does not prove integrity of the capability.
Revocation is implemented by changing the appropriate nonce. However, this invalidates all capabilities for the corresponding object. Selective revocation is possible by using separate keys to encrypt capabilities for identical objects but given to different subjects. Selective revocation will then be possible by changing the appropriate key.
A disadvantage is that free propagation of capabilities is not possible any more. This could be overcome partially by implementing one of the access rights bits as a copy bit. When the object server sets this bit, the capability may be freely passed. When receiving a capability the server first checks this bit and when it proves to be one the server discards subject checking. If this bit is zero, the server always performs subject checking. In fact such a bit serves to indicate if the capability is controlled or uncontrolled. Note that the choice for using controlled capabilities is influenced by the decision not to consider performance. Controlled capabilities offer more security but do that at cost of less performance (more server interaction).
To illustrate the created capability mechanism two basic operations are chosen: creation of an object

together with its capability and access on that object. The first operation has been illustrated in figure 5.10.

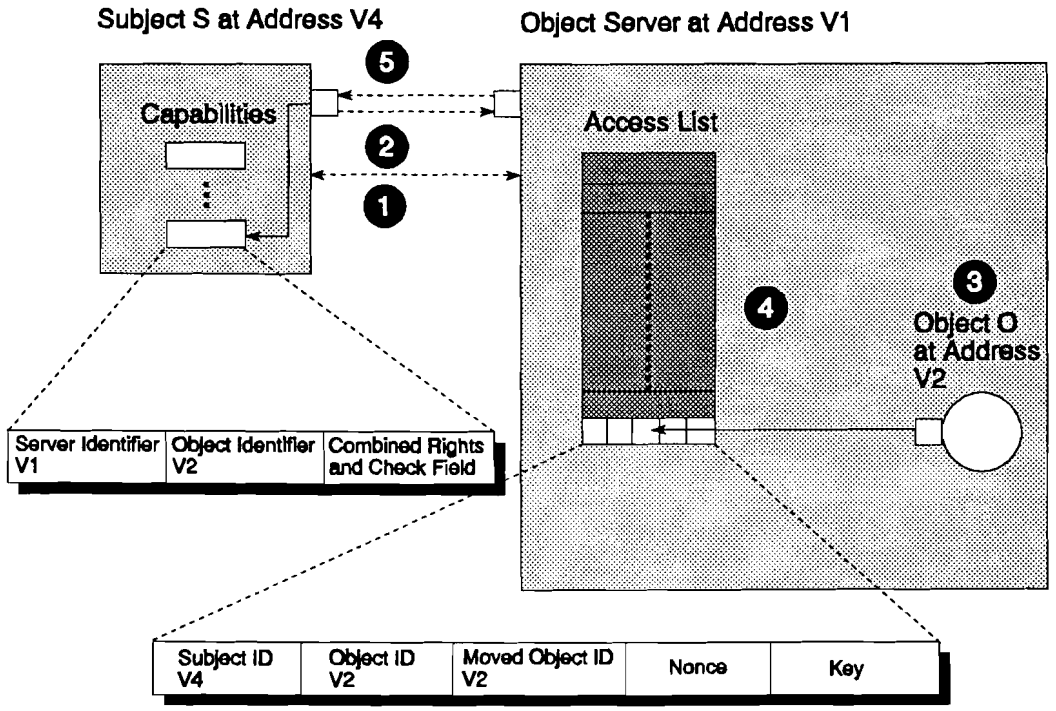


Figure 5.10 Controlled capability mechanism performing an object creation.

The process of object creation can be divided into 5 steps as is done in figure 5.10. These 5 steps are:

1. Both parties; subject S and the object server start to communicate. Depending on the amount of security needed a mutual authentication process can be started. As a result both parties trust the identity of the other. Another consequence of this authentication procedure is that both parties contain a secret identifier that is only known by the other party. This identifier or certificate can thus be used to authenticate future messages.
2. After authentication, subject S sends a message to the object server residing at virtual address V1. The location or name of the object server could have been made publicly available by a global name server. The message contains a request for creating an object, object specific characteristics, and the request to supply a capability for that object. Also, the message contains the identity of subject S, i.e. its virtual address V4, and the certificate created during the authentication phase. This certificate is added to authenticate the message. The message may cross node boundaries when the subject and the object server are located at different nodes. Therefore the message may need to be encrypted to prevent data from being seen by an intruder.
3. The object server creates the specified object together with a capability. Therefore, the object server picks a nonce and uses this identifier to extend the access rights field of the capability to be created. The nonce used can be identical for all objects of this object server because it serves to check the integrity of the capability and is being encrypted. This newly created combined access rights and check field is then being encrypted by a key associated to this unique combination of object and subject.
4. Both nonce and key are stored in the access list addressable by the subject identifier field V4. The other fields in the access list entry contain the object identifier and the moved object identifier.

This moved object identifier field provides for a simple indirection mechanism to maintain the object's name V2 when the object has to be moved.

5. The created capability is sent back to the subject. This message might also be encrypted to prevent both the object identifier and server identifier from being seen by intruders. The certificate can again be added for authentication.

After creation of the object and supplying a capability for it, another basic operation might be to access the object. We assume that subject S has the capability for the object and wants to access object O. This process has been depicted in figure 5.11.

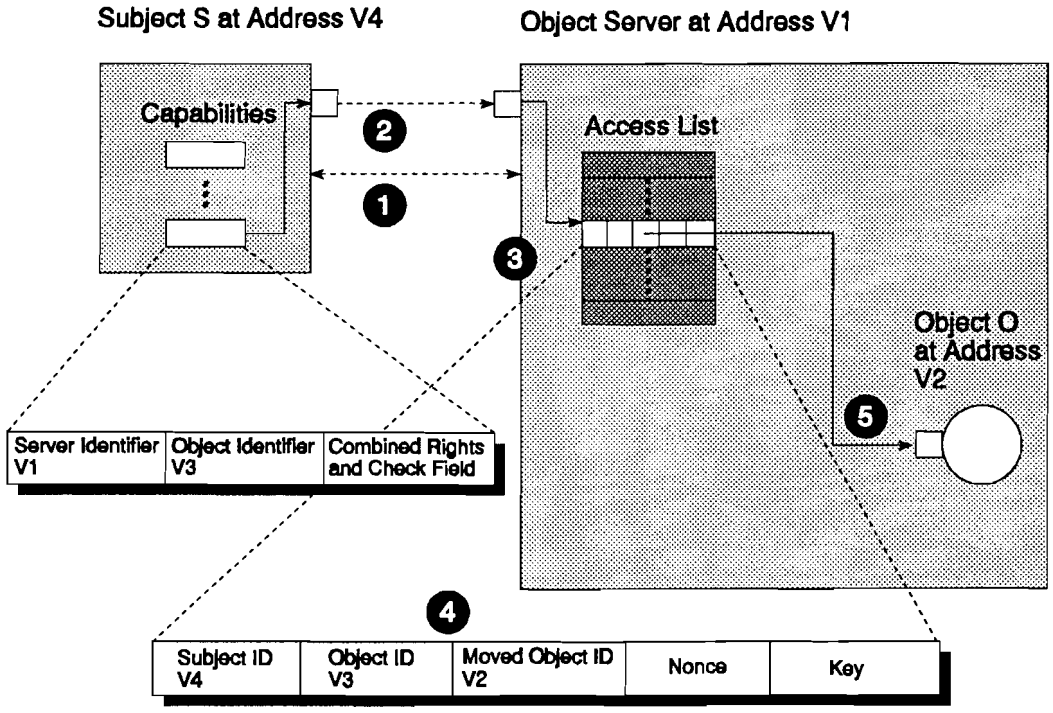


Figure 5.11 Controlled capability mechanism based on access lists for controlled object access.

The mechanism depicted in figure 5.11 for accessing an object has also been divided into 5 steps:

1. Both parties, subject S and the object server start to communicate. As with all communication between subject and object server, depending on the amount of security needed a mutual authentication process can be started. As a result both parties trust the identity of the other and contain a secret certificate whichcan be used to authenticate future messages.
2. After authentication, subject S sends a message to the object server residing at virtual address V1. The location or name of the object server can be extracted from the capability. The message contains the capability for object O, a request for access A on object O, the subject's identification S, and the secret certificate. This message may cross node boundaries when the subject and the object server are located at different nodes. Therefore the message may again need to be encrypted to prevent data from being seen by an intruder.
3. The object server receives the message and uses the subject identification field V4 to index the access list. Next, an entry is looked for which contains the object identifier V3 because duplicate entries (i.e. identical subject identifiers but different object identifiers, this happens when a subject does have capabilities for several objects) may exist. If no entry is present, it is assumed that subject S is an illegitimate holder of the capability, hence the request is discarded.
4. When an entry is found, the key is extracted and used to decrypt the combined rights and check

field. When the check field proves to be equal to the nonce in the access list entry, the capability is assumed to be valid. Else, the capability is not valid and the request is discarded.

5. The rights field is now used to decide if requested access A is allowed. If this access is allowed the object is referenced by the moved object identification V2. It should be noted that by using this construction only the server is aware of the fact that the object has moved from virtual address V3 to virtual address V2. Hence, location transparency has been maintained. However, this mechanism only provides for objects being moved by the server.

Passing of capabilities is a combination of both described operations. The subject sends a message containing the capability, its subject identification, the subject identification of the designated receiver of the capability, and the access rights it wants to pass. Then, the object server performs steps 3 and 4 of the access operation and determines the access rights to be passed. Next, the assumed receiver could be authenticated and a new capability can be created for the designated receiver following steps 3, 4, and 5 of the creation operation.

The described mechanisms offer protection depending on the next assumptions:

- Encryption is secure. Both authentication and transfer of data across the network depend on protection the used encryption techniques offer.
- There is some amount of trust in the system. Access matrix methods can not offer protection against Trojan Horse attacks as was mentioned in chapter 2.

6 Conclusions

In the introduction of this report it was stated that the main goal of this study was to develop a common view of how to construct an operating system that benefits significantly from the new possibilities wide-address architectures offer. Next, this view was narrowed to addressing, naming, and protection because these are the issues most likely to be affected first by architectural changes. Thus alternatively stated, this report's goal becomes: what are the advantages wide-address architectures offer with regard to addressing, naming, and protection and how do these new implementations affect the operating system?

As a first part of this study, hardware support for wide-addressing was looked at. Several wide-address architectures, that is architectures with flat (i.e. unsegmented) virtual address capabilities larger than 32 bits, have been studied and the most interesting ones have been described in appendix A. An alternative wide-address architecture, although not explicitly stated as being one, is the MONADS hardware which differs from the ones described in appendix A in that it is not commercially available but merely serves research. Another major difference is that it is a capability based architecture, i.e. capability support has been implemented in hardware. After studying several wide-address architectures, next observations can be made:

- The trend in wide-address architectures (or more general, newer architectures) seems to be that they become more and more RISC based architectures. All described wide-address architectures are called RISC (-like) processors. However, this does not automatically imply that they contain all properties of RISC based architectures but all of them do contain simple, directly executed instruction sets.
- The design rationale behind the extended addressing capabilities is often influenced by a long desired life-cycle for the wide-address architecture.
- Most wide-address architectures have been designed guided by popular application requirements, "Traditional Unix" has driven the design. This is illustrated by the fact that several wide-address architectures contain some kind of address space identifiers caused by necessary support for the multiple address space model used in Unix. A consequence of this application dependent design philosophy is that support for new approaches like the single virtual address space cannot be optimal.
- Virtual memory is a common feature in all of today's architectures. The way in which virtual memory has been implemented is practically the same; most processors support some kind of paged virtual memory.
- A final trend that can be noticed is that page table handling has moved from hardware to software. Page table- as well as TLB handling has moved towards responsibility of the operating system. This allows for much more flexibility when developing alternative address space models. In fact, this property is one of the requirements for efficiently implementing the single virtual address space.

Most of the given properties hold generally for new architectures. However, the wide-address architectures can serve as a foundation for new operating system implementations. The large address space provided by wide-address architectures can be used for much more than only running larger programs. A 64-bit architecture increases current addressing capabilities in a way certainly more than will be needed by any single application in the near future. This huge addressable space allows for a

more efficient implementation of the single virtual address space concept. This concept is a more simple operating system organisation for supporting sharing. All addressable data resides within the same address space and is therefore uniformly addressable by their virtual addresses, the most simple and efficient name structure. For many conventional architectures the address space is simply not large enough to support this concept.
Because of the enormous address space available on wide-address architectures it is also possible to extend this concept of sharing towards a distributed environment. The address space is then being partitioned among nodes. This results in all objects of a distributed system being uniformly named by their virtual addresses. As a result of this uniformity the system becomes much clearer to the application designer. The application designer sees an enormous address range in which all objects reside and can be addressed by using only their virtual addresses independent of their physical location.
The use of virtual addresses as unique object identifiers also allows for efficient implementation of a single-level store. A single-level store is a storage model in which objects are addressed independently of their position in storage, i.e. in primary- or on secondary storage. Virtual addresses can then serve as the unique identifiers needed in this model. An extension of this single-level store is persistence which results in objects being transparently transient or persistent.
Summarizing, by using the single virtual address space model, sharing is enhanced significantly and objects can be location transparent as well as persistent. Location transparency implies that the user or application designer is not aware of the node nor the type of storage (primary or secondary) on/in which an object resides. Thus the single virtual address space offers possibilities to create a high level of transparency, a major property of distributed operating systems, without the costs usually needed (i.e. the costs of an extra unique identifier format) to achieve this. It can be stated that the majority of advantages result from the possibility of using virtual addresses as unique object identifiers.
However, one issue has to be considered carefully in this concept. Because of storing objects in a single address space, one of the design principles of the multiple address space model disappears. This design principle is security which is increased when using a separate address space per process because of limiting the scope of view. Therefore objects in the single address space have to be logically grouped into protection domains to limit their access. These protection domains then serve as some kind of address space as in traditional operating systems.
Implementing protection domains in a single virtual address space requires low-level protection mechanisms to be re-designed because protection and translation can be decoupled. Protection information does not necessarily have to appear in page tables any more but may be placed in separate protection tables free of translation information. This allows for different granularities in protection and translation. We have discussed an efficient implementation called the region table that contains protection information independent of translation information. The foundation for this region table stems from the Opal system, one of the two case studies done on single virtual address space operating systems, in which a so-called protection lookaside buffer has been implemented.
The other operating system studied besides the Opal system is the MONADS system. Both operating systems show that it is possible to implement a single virtual address space operating system on conventional wide-address architectures as well as on dedicated hardware. Important properties concerning single virtual address space operating systems illustrated by those case studies are:

- The amount of transparency is influenced by the kind of operating system. A distributed operating system (the Opal system) provides for more transparency than a networking operating system (the MONADS system). This is illustrated by the virtual address format which is flat in the Opal system in contrast with the MONADS system that uses certain address bits for node identification.
- The size of a single virtual address space network is limited by the addressing capabilities of individual nodes. Therefore, conventional wide-address architecture operating systems are limited to a few tens of nodes, i.e. a local area network. Dedicated hardware such as the MONADS architecture can provide for much larger addressing capabilities and can therefore support much larger networks. The MONADS-MM system supports 128-bit addresses including a 32-bit node

number which permits for a theoretical network size of 2^{32} nodes.

- Conventional memory management should be adapted to large virtual memories. This is illustrated by the inverted page table used in the MONADS system.
- Besides the virtual addresses a second kind of name has to be used because users cannot cope efficiently with virtual addresses serving as object names. This second kind of name is provided by an extra hierarchical name space mapped onto virtual addresses.
- Capabilities can be an useful implementation of the access matrix as the protection mechanism in single virtual address space systems. The MONADS system is built around partitioned capabilities while in the Opal system sparse capabilities are used.
- For additional security, data sent along communication lines in the network and between primary- and secondary storage can be protected by encryption. This holds equally for general distributed systems. Also, kernels can be made secure by a special booting procedure as was described in appendix B.

As an extension to the case studies, several aspects of the case studies have been reconsidered in chapter 5. It was pointed out that the flat virtual address as used in the Opal system provides for more transparency but does so at cost of more administration compared to a partitioned virtual address. To implement a high level of transparency, more administration has to be done by the memory servers. The introduced region abstraction provides for a separation of protection and translation issues in the single virtual address space. An advantage of this abstraction compared to the protection lookaside buffer concept in the Opal system is finer granularity of protection which becomes possible. In the Opal system the granularity of protection was a virtual page while the region table allows for a granularity of protection as fine as one byte.

Another advantage of separating both issues is that translation- and protection information lookup can be performed in parallel.

At the end of chapter 5, a capability based protection mechanism was presented that contained several characteristics of protection mechanisms used in both case studies. The capability mechanism described has next properties:

- Sparse capabilities are being used, so capabilities can be stored as normal data.
- Subject checking is performed on every access.
- Capabilities are protected against forgeary.
- Encryption is used to protect data being sent among nodes.
- Revocation is possible.
- Free propagation is possible at cost of subject checking on every access.

We believe that such a capability mechanism provides for a reasonable amount of security in a single virtual address space system compared to conventional operating systems. When more advanced protection mechanisms are needed (e.g. in military systems) they can be implemented more easily because of the specific properties of the single virtual address space.

Concluding, a single virtual address space can be an efficient address space model for wide-address architectures. However, implementing protection requires a different approach than used in conventional systems because of the lack of security provided by separate address spaces in those systems.

Whether the single virtual address space concept will appear in commercially available operating systems will depend on support for conventional operating systems. To be commercially realizable the new operating system should support Unix as a separate module to maintain application compatibility.

Hopefully, this report has given some insight in the new possibilities of wide-address architectures besides only running larger programs.

Literature

[Abra85] Abramson, D.A. and J.L. Keedy.
Implementing a large virtual memory in a distributed computing system.
In: System Sciences, Proceedings of the 18th annual Hawaii international conference (HICSS), Honolulu, USA, 2-4 Jan., 1985.
North Hollywood: Western Periodicals, 1985.
P. 515-522.

[Abro89] Abrossimov, V. et al.
Generic virtual memory management for operating system kernels.
In: Operating System Principles, Proceedings of the 12th ACM symposium, Litchfield Park, USA, 3-6 Dec., 1989.
P. 123-136.

[Aho87] Aho, A. van et al.
Data structures and algorithms.
Massachusetts: Addison-Wesley, 1987.

[Ande91] Anderson, T.E. et al.
The Interaction of architecture and operating system design.
In: Architectural Support for Programming Languages and Operating Systems, Proceedings of the 4th international conference, Santa Clara, USA, 8-11 Apr., 1991.
Washington: IEEE Computer Society, 1991.
P. 108-121.

[Appe91] Appel, A.W. and K. Li.
Virtual memory primitives for user programs.
In: Architectural Support for Programming Languages and Operating Systems, Proceedings of the 4th international conference, Santa Clara, USA, 8-11 Apr., 1991.
Washington: IEEE Computer Society, 1991.
P. 96-108.

[Bart92] Bartoli, A. et al.
Wide address spaces - Exploring the design space.
ACM Operating Systems Review, Vol. 27 (1993), No. 1, p. 11-17.

[Bens72] Bensoussan, A. et al.
The Multics virtual memory: concepts and design.
Communications of the ACM, Vol. 15 (1972), No. 5, p. 308-318.

[Bers91] Bershad, B. et al.
User-level interprocess communication for shared memory multiprocessors.
ACM Transactions on Computer Systems, Vol. 9 (1991), No. 2, p. 175-198.

[Birm91] Birman, K.P.
Fault-tolerance in sixth generation operating systems.
In: Operating systems of the 90s and beyond, Proceedings of the international workshop, Dagstuhl, Germany, 8-12 July, 1991.
Ed. by G. Goos and J. Hartmanis.
Berlin/Heidelberg: Springer, 1991.
Lecture notes in computer science, Vol. 563.

[Broe87] Broessler, P. et al.
Addressing objects in a very large distributed virtual memory.
In: Distributed Processing, Proceedings of the IFIP WG 10.3 working conference, Amsterdam, The Netherlands, 5-7 Oct., 1987.
Ed. by M.H. Barton, E.L. Dagless and G.L. Reijns.
Amsterdam: North-Holland, 1988.
P. 105-116.

[Cart91] Carter, J.B. et al.
Implementation and performance of Munin.
ACM Operating Systems Review, Vol. 25 (1991), No. 5, P. 152-164.

[Cart92] Carter, J.B. et al.
Distributed operating systems based on a protected global virtual address space.
In: Workstation Operating Systems, Proceedings of the third workshop, Key Biscane, USA, 23-24 Apr., 1992.
Ed. by A. Copeland and E. Straub.
Los Alamitos: IEEE computer society press, 1992.
P. 75-79.

[Chan88] Chang, A. and M.F. Mergen.
801 Storage, architecture and programming.
ACM transactions on Computer Systems, Vol. 6 (1988), No. 1, p. 28-50.

[Chas92a] Chase, J.S. et al.
Lightweight shared objects in a 64-bit operating system.
In: Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), Proceedings of the conference, Vancouver, Canada, 18-22 Oct., 1992.
Ed. by A. Paepcke.
New York: ACM, 1992.
P. 397-413.

[Chas92b] Chase, J.S. et al.
Opal: A single address space system for 64 bit architectures.
In: Workstation Operating Systems, Proceedings of the third workshop, Key Biscane, USA, 23-24 Apr., 1992.
Ed. by A. Copeland and E. Straub.
Los Alamitos: IEEE computer society press, 1992.
P. 80-85.

[Chas93] Chase, J.S. et al.
Distribution in a single address space operating system.
ACM Operating Systems Review, Vol. 27 (1993), No. 2, p. 61-65.

[Cope90] Copeland, G. et al.
Uniform object management.
In: Advances in Database Technology - EDBT '90, Proceedings of the international conference on extending database technology, Venice, Italy, 26-30 March, 1990.
Ed. by G. Goos and J. Hartmanis.
Berlin/Heidelberg: Springer, 1990.
Lecture notes in computer science, Vol. 416.
P. 253-268.

[Fabr74] Fabry, R.S.
Capability based addressing.
Communications of the ACM, Vol. 17 (1974), No. 7, p. 403-412.

[Flei88] Fleisch, B.D.
Distributed shared memory in a loosely coupled distributed system.
ACM Computer Communications review, Vol. 17 (1988), No. 5, p. 317-327.

[Frei90] Freisleben, B. et al.
Capabilities and encryption: the ultimate defense against security attacks ?
In: Computer architectures to support security and persistence of information, Proceedings of the international workshop, Bremen, West-Germany, 8-11 Apr., 1990.
Ed. by J. Rosenberg and J.L. Keedy.
London: Springer, 1990.
Workshops in computing.
P. 106-119.

[Gosc91] Goscinski, A.
Distributed operating systems, the logical design.
Sydney: Addison-Wesley, 1991.

[Glas91] Glass, B.
The MIPS R4000.
Byte, Vol. 16 (1991), No. 13, p. 271-282.

[Grov90] Groves, R.D. and B. Oehler.
RISC system/6000 processor architecture.
Microprocessors and Microsystems, Vol. 14 (1990), No. 6, p. 357-366.

[Hayt92] Hayter, M. and D. Mc. Auley.
The desk area network.
ACM Operating Systems Review, Vol. 25 (1991), No. 4, p. 14-21.

[Hens90] Henskens F.A. et al.
Stability in a network of MONADS-PC computers.
In: Computer architectures to support security and persistence of information, Proceedings of the international workshop, Bremen, West-Germany, 8-11 Apr., 1990.
Ed. by J. Rosenberg and J.L. Keedy.

London: Springer, 1990.
Workshops in computing.
P. 246-256.

[Jone86] Jones, M.B. and R.F. Rashid.
Mach and matchmaker: kernel and language support for object-oriented distributed systems.
In: Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), Proceedings of the conference, Portland, USA, Sept. 29 - Oct. 2, 1986.
P. 67-77.

[Kane92] Kane, G. and J. Heinrich.
MIPS RISC Architecture.
Englewoods Cliffs: Prentice Hall, 1992.

[Kold92] Koldinger, E.J. et al.
Architectural support for single address space operating systems.
In: Architectural Support for Programming Languages and Operating Systems, Proceedings of the 5th international conference, Boston, USA, 12-15 Oct., 1992.
Washington: IEEE Computer Society, 1992.
P. 175-186.

[Krue89] Krueger, S.
Are 32 bits enough?
Byte, Vol. 14 (1989), No. 12, p. 299-305.

[Lazo91] Lazowska, E.D.
Operating system support for high-performance architectures.
In: Operating systems of the 90s and beyond, Proceedings of the international workshop, Dagstuhl Castle, Germany, 8-12 July, 1991.
Ed. A. Karshmer and J. Nehmer.
Berlin/Heidelberg: Springer, 1991.
Lecture notes in computer science, Vol. 563.
P. 40-43.

[Lee89] Lee, R.B.
Precision architecture.
IEEE Computer, Vol. 22 (1989), No. 1, p. 87-91.

[Levy84] Levy, H.M.
Capability-based computer systems.
Bedford: Digital Press, 1984.

[Lieb93] Liebl, A.
Authentication in distributed systems: a bibliography.
ACM Operating Systems Review, Vol. 27 (1993), No. 4, p. 31-41.

[Mash91] Mashey, J.R.
64-Bit computing.
Byte, Vol. 16 (1991), No. 9, p. 135-142.

[Mull86] Mullender, S.J. and A.S. Tanenbaum.
The design of a capability-based distributed operating system.
The Computer Journal, Vol. 29 (1986), No. 4, p. 289-299.

[Mull89] Mullender, S.J.
Distributed systems.
Workingham: Addison-Wesley, 1989.
ACM Press Frontier Series.

[Need78] Needham, R.M. et al.
Using encryption for authentication in large networks of computers.
Communications of the ACM, Vol. 21 (1978), No. 12, p. 993-999.

[Nitz91] Nitzberg, B. and V. Lo.
Distributed shared memory: A survey of issues and algorithms.
IEEE Computer, Vol. 24 (1991), No. 8, p. 52-60.

[Koch90] Koch, D. and J. Rosenberg.
A secure RISC-based architecture supporting data persistence.
In: Computer architectures to support security and persistence of information, Proceedings of the international workshop, Bremen, West-Germany, 8-11 Apr., 1990.
Ed. by J. Rosenberg and J.L. Keedy.
London: Springer, 1990.
Workshops in computing.
P. 188-201.

[Oust90] Ousterhout, J.K.
Why aren't operating systems getting faster as fast as hardware.
In: USENIX Summer Conference, Anaheim, USA, June 11-15, 1990.
P. 247-257.

[Rafi91] Rafiquzzaman, M.
Microprocessors and microcomputer based system design.
Boca Raton: CRC Press, 1991.

[Rede80] Redell, D.D. et al.
Pilot: an operating system for a personal computer.
Communications of the ACM, Vol. 23 (1980), No. 2, p. 81-92.

[Rose85] Rosenberg, J. and D. Abramson.
Monads PC - A capability based workstation to support software engineering.
In: System Sciences, Proceedings of the 18th annual Hawaii international conference (HICSS), Honolulu, USA, 2-4 Jan., 1985.
North Hollywood: Western Periodicals, 1985.
P. 222-231.

[Rose90] Rosenberg, J. and F.A. Henskens.
Stability in a persistent store based on a large virtual memory.
In: Computer architectures to support security and persistence of information, Proceedings of the international workshop, Bremen, West-Germany, 8-11 Apr., 1990.
Ed. by J. Rosenberg and J.L. Keedy.

London: Springer, 1990.
Workshops in computing.
P. 229-245.

[Rose92] Rosenberg, J. et al.
Addressing mechanisms for large virtual memories.
The Computer Journal, Vol. 35 (1992), No. 4, p. 369-375.

[Sark90] Sarkar, M. and E.F. Gehringer.
Dimensions of addressing schemes.
In: Computer architectures to support security and persistence of information, Proceedings of the international workshop, Bremen, West-Germany, 8-11 Apr., 1990.
Ed. by J. Rosenberg and J.L. Keedy.
London: Springer, 1990.
Workshops in computing.
P. 31-47.

[Scot92] Scott, M.L. and W. Garrett.
Shared memory ought to be commonplace.
In: Workstation Operating Systems, Proceedings of the third workshop, Key Biscane, USA, 23-24 Apr., 1992.
Ed. by A. Copeland and E. Straub.
Los Alamitos: IEEE computer society press, 1992.
P. 86-91.

[Site92a] Sites, R.L.
Alpha architecture reference manual.
Burlington: Digital Press, 1992.

[Site92b] Sites, R.L.
RISC enters a new generation.
Byte, Vol. 17 (1992), No. 8, p. 141-148.

[Stal92] Stallings, W.
Operating Systems.
New York: Macmillan Publishing Company, 1992.

[Tane92] Tanenbaum, A.S.
Modern Operating Systems.
Englewood Cliffs: Prentice-Hall, 1992.

[Whee92] Wheeler, R. and B.N. Bershad.
Consistency management for virtually indexed caches.
In: Architectural Support for Programming Languages and Operating Systems, Proceedings of the 5th conference, Boston, USA, 12-15 Oct., 1992.
Washington: IEEE Computer Society Press, 1992.
P. 124-136.

[Yeo93] Yeo, A.K. et al.
A taxonomy of issues in name systems design and implementation.
ACM Operating Systems Review, Vol. 27 (1993), No. 3, p. 4-18.

[Zhou90] Zhou, S. et al.
Extending distributed shared memory to heterogeneous environments.
In: Distributed Computing Systems, Proceedings of the 10th IEEE international conference, Paris, France, May 28 - June 1, 1990.
Los Alamitos: IEEE Computer Society Press, 1990.
P. 30-37.

Acronyms

AID	Access Identifier
AM	Access Matrix
ASID	Address Space Identifier
CAD	Computer Aided Design
CPU	Central Processing Unit
DAN	Desk Area Network
DES	Data Encryption Standard
DSM	Distributed Shared Memory
IPC	Inter-Process Communication
IPT	Imported Pages Table
LAN	Local Area Network
MMT	Main Memory Table
PD-ID	Protection Domain Identifier
PL	Privilege Level
PLB	Protection Lookaside Buffer
PPN	Physical Page Number
RISC	Reduced Instruction Set Computer
RPC	Remote Procedure Call
RSA	Rivest-Shamir-Adleman
TLB	Translation Lookaside Buffer
VPN	Virtual Page Number
XPT	Exported Pages Table

A Wide-Address Architectures

In this appendix three architectures with wide-addressing will be looked at. Wide-addressing is defined as an addressing mechanism in which the virtual address format exceeds the 32 bits. Most of today's microprocessors offer 32-bit virtual addresses as can be seen in figure 1.1.
The architectures described are: the MIPS R4000 [Kane92], [Glas91], the Hewlett-Packard PA-RISC [Lee89], [Kold92], and the DEC Alpha [Site92a], [Site92b]. There are several other architectures which support wide-addresses but they do so by using segmentation (e.g. Intel 80386/80486 [Rafi91], IBM RISC system/6000 [Grov90]). The disadvantages of segmentation in supporting wide-address operating systems are described in paragraph 3.2. The Hewlett-Packard PA-RISC is in fact a segmented architecture but it differs from other segmented architectures in that it can use long addresses directly.

A.1 The MIPS R4000

The MIPS R4000 is the latest processor[1] in the RISC processor series of the MIPS family. It is an extension of earlier MIPS designs, the 32-bit R2000, R3000 and R6000, all of which have solid track records. The most important difference, however, is that the R4000 increases the widths of the internal and external data paths, as well as the widths of the addresses, registers, and ALUs, to 64 bits.

A.1.2 Address Calculation

The virtual address format of the MIPS R4000 is depicted in figure A.1.

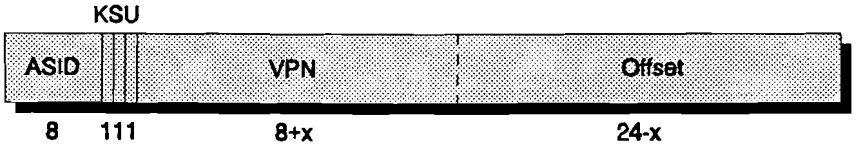


Figure A.1 Virtual address format of the MIPS R4000.

The fields in figure A.1 have the following meaning:

- ASID: address space identifier, this field is implemented to identify different user processes, each executing within a private virtual address space. On a context switch, translation lookaside buffer (TLB) entries continue to exist because they are tagged with this ASID. The size of the ASID field is 8 bits, thus allowing 256 user processes to exist

[1] **This is not completely true because lately the MIPS R4400 has been introduced which has a 64-bit virtual address in contrast with the 40-bit virtual address of the MIPS R4000.**

concurrently.

- KSU: these three bits indicate the operation mode of the processor, which determines the address partitioning (see paragraph A.1.3).
- VPN: virtual page number, identifies the virtual page. This number is translated into a physical page number with use of the TLB. The size of the VPN is variable between 8 (256 pages) and 20 (1048576 pages) bits.
- OFFSET: offset, indicates the offset within a page. This field is passed unchanged to physical memory. The size of the OFFSET is variable between 12 (page size of 4 Kilobytes) and 24 (page size of 16 Megabytes) bits.

Thus, the virtual address format (excluding the KSU bits) is 40 bits in size. The physical address format, however, is limited to 36 bits.

A.1.3 Protection

Hardware support for protection is very severe. The design rationale behind this is the fact that multiple operating modes and protection schemes add complexity to the hardware while support for operating systems is never optimal.
The KSU bits described in paragraph A.1.2 provide some form of protection. They indicate one of the three following modes: user mode, supervisor mode or kernel mode. Partitioning of memory in these three modes is depicted in figure A.2.

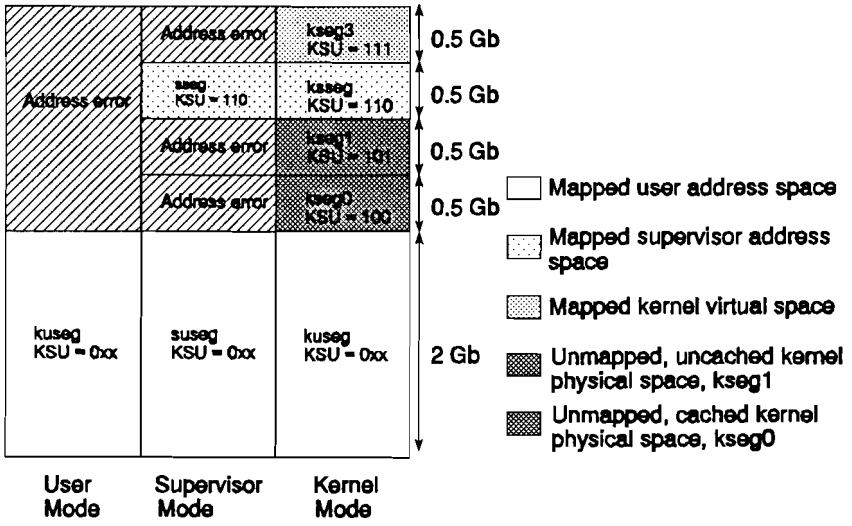


Figure A.2 Memory partitioning in three different operating modes.

In user mode all references to kuseg are mapped through the TLB, and cache use is determined by bit settings within the TLB entry for each page. An attempt to reference an address outside of kuseg results in an address error exception.
Supervisor mode is intended for those layered operating system implementations where a "true kernel" runs in R4000 kernel mode, and the rest of the operating system runs in supervisor mode.
Upon an exception, the processor enters kernel mode and stays there until a special instruction (return from exception) is executed. In kernel mode the entire memory area is divided into five regions, cached and uncached, mapped and unmapped, and kuseg.
The reason for having uncached regions is to avoid cache replacements on for example writes to I/O

buffers that are unlikely to be referenced again. The MIPS R4000 implements two on-chip caches. One for instructions and one for data. Both caches are virtually indexed and physically tagged.
The reason for an unmapped (i.e. not mapped through the TLB, physically based) segment is to avoid translation overhead (in this case to save TLB entries) for operating system components that are typically memory resident anyway.
While access to the user address space is always allowed, access to the other memory regions is only allowed when the processor is in the appropriate mode. This defines some form of protection.
The virtual address translation is depicted in figure A.3.

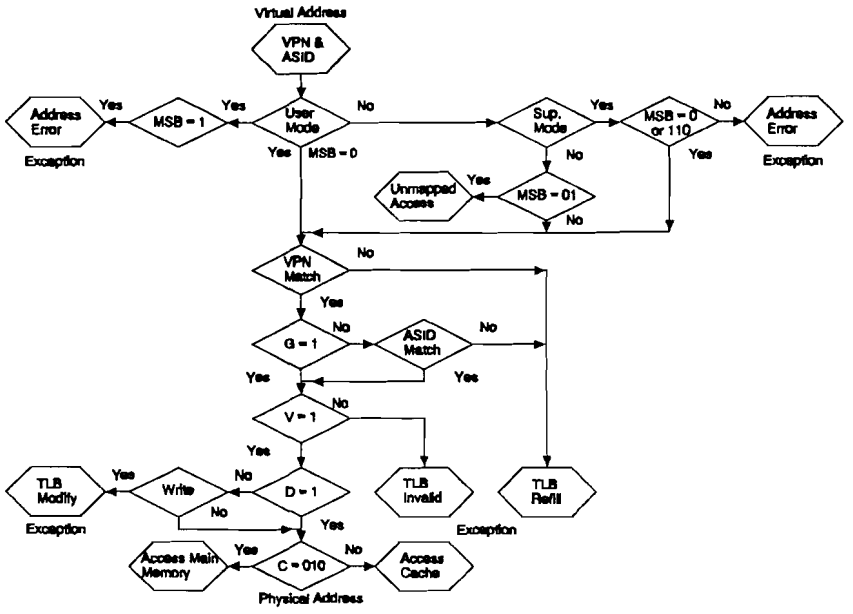


Figure A.3 Virtual address translation of the MIPS R4000.

The G (global) bit defines if ASID checking is on or off. The V (valid) bit indicates if the TLB entry is valid or not. If the entry is not valid, a TLB invalid exception occurs. In this architecture the D (dirty) bit marks a page dirty or not dirty. If a page is marked dirty it is writable. This bit is actually a "write-protect" bit that software can use to prevent alteration of data. The three bits C specify the cache algorithm or no caching at all.
Important to notice is the fact that TLB management has moved towards the operating system. On TLB misses the translation lookaside buffer is loaded by operating system software. This greatly enhances flexibility in page table structure. A disadvantage of this architecture is caused by the unmapped regions. Because they are accessed directly through a physical base register, there is no indirection and therefore no ability to specify page-level protection or access control, except to the entire region. The unmapped region of the address space is protected only by kernel mode execution. This organisation is therefore best suited to a monolithic kernel structure, like current Unix implementations. With small-kernel operating systems, much system code runs in user mode (user-level servers) and therefore cannot benefit from this unmapped regions.

A.2 The Hewlett-Packard PA-RISC

Hewlett-Packard designed Precision Architecture to serve as a common foundation for its computer systems, to enhance software portability, to provide price-performance advantages, and to streamline the company's hardware and software development, manufacturing, and support activities. One of the main goals of the project was a long, useful life for the 1990's and beyond. Among other strategic goals, this resulted in a simple RISC-like execution model.

A.2.1 Address Calculation

Precision architecture provides a 64-bit address range. However, its data path is only 32-bit wide. Therefore the virtual address is composed of one of eight 32-bit space registers and a 32-bit offset. In contrast with other segmented architectures, the HP PA-RISC allows applications to use long-form addresses directly. For efficiency reasons the concept also allows short pointer addressing in which 64-bit addresses are handled by short 32-bit pointers. It allows, at a given time, data access to four distinct virtual spaces, each space being one Gigabyte in size. Long pointer addressing provides access to four billion spaces, each space being four Gigabytes in size.
The virtual address is further partitioned into the space identifier, the virtual page number (VPN), and the page offset. Each page has a fixed size of 2 Kilobytes. The reason for this relatively small page size is to allow for a finer granularity of protection. The virtual address format is depicted in figure A.4.

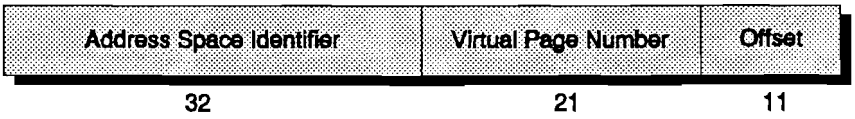


Figure A.4 Virtual address format of the HP PA-RISC.

The space identifier and the offset are translated into a 21-bit physical page number (PPN), which is then used to access physical memory. Two software tables are used: a hash table to index into a page directory table which is organized as an inverted page table. Both tables reside in physical memory for performance reasons. To speed up the virtual to physical translation process, a translation look-aside buffer (TLB) is defined as the processor's interface to the virtual memory system. If an address translation is not in the TLB, a TLB miss occurs, handled either by a software interrupt routine (the architecture defines memory management instructions for inserting, changing, querying, and deleting entries in the TLB) or by a hardwired sequence of operations. Handling of TLB misses by software interrupt routines allows for variable page table structures.
Both TLB and page directory table contain a dirty bit per entry to minimize page traffic. This dirty bit is cleared when a page is brought in from disk and remains clear as long as no writes to the page occur. Note that this implementation is different from the implementation of the dirty bit in the MIPS R4000 design.
A hardware-software optimization allows virtual cache indexing by not allowing software to do address aliasing or mapping of different virtual pages to the same physical page. This allows the cache to be accessed in parallel with the virtual address translation being done by the TLB, without restrictions to the size of the cache.

A.2.2 Protection

The architecture provides hardware support for access protection to be built into the storage unit and performed in the same cycle as virtual address translation and cache access. Protection checking is defined at the page level to control access to the page in three dimensions: the type of access allowed (read, write, or execute), the privilege level at which the access is allowed, and the group of processes allowed access to the page. For access rights checking, the architecture defines four hierarchical protection rings. The current privilege level of a process is checked against the privilege level for the read, write, or execute access being made to that page by the process. This allows a process to have different access rights over time without the overhead of changing TLB entries when access rights change or at process switch. Less privileged programs can invoke the services of an operating system supervisor or kernel using a procedure call to a gateway instruction, which branches to the body of the more privileged routine. The gateway instruction can promote the privilege level while saving the caller's privilege level in the return address register so that it cannot be forged by the caller. On return the original privilege of the caller is restored (this ring protection mechanism is comparable to the protection mechanism in the Intel 80386 [Rafi91).
Besides this, the currently executing process can claim membership in up to four page-groups simultaneously, each group having its own access identifier and write-disable bit, saved in four page-group registers. This is depicted in figure A.5.

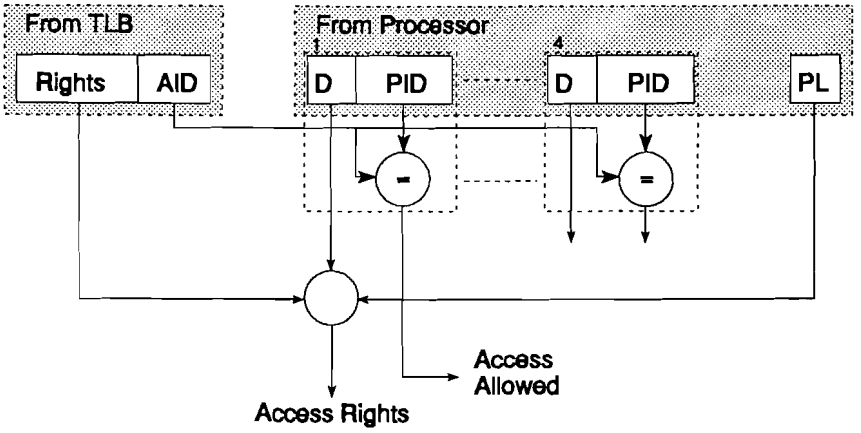


Figure A.5 HP PA-RISC protection architecture.

The TLB entry for a page includes a set of access rights, and a field called access identifier, or AID, that contains a page-group number to which the page belongs, in addition to translation information. On a memory reference, the TLB is indexed with the virtual page number, returning the physical address translation and the AID for the page. The processor must then determine if access to the page-group specified in the AID is permitted to the currently executing process. If the page-group number in the AID matches one of the PIDs or the AID field is zero (page-group global to all processes), then the exact access rights allowed are determined by a combination of:

- Access rights specified for the page in the rights field of the TLB entry, specifies different access rights for the different processor privilege levels.
- The current processor privilege level (PL).
- The write disable bit (D), which allows disabling writes from a process to an entire page-group, regardless of the value in the rights field.

If the page-group number in the AID does not match any of the PIDs or the access allowed is insufficient to complete the memory reference, then an access violation is signalled and the operating

system kernel is invoked. The kernel may respond by modifying the TLB or page-group registers and restarting the instruction, or it may deliver an exception to the protection domain that issued the instruction.
A disadvantage of this structure is the fact that only four page-groups are allowed. This disadvantage could be overcome by replacing the page-group registers with a cache of permitted page-groups, with support replacement strategies. This problem is attacked in this way in the Opal system, described in paragraph 4.1.3
Altogether, the protection features built into the architecture allow implementation of very secure, flexible environments.

A.3 The DEC Alpha

Alpha is a 64-bit (all registers and addresses are 64-bit wide) load/store RISC architecture that is designed with particular emphasis on three elements that most affect performance: clock speed, multiple instruction issue, and multiple processors. Only those design elements that appeared valuable for a projected 25-year design horizon were adopted. Thus, according to DEC: "Alpha becomes the first 21st century computer architecture".
The Alpha architecture is designed to avoid bias towards any particular operating system or programming language by implementing the Privileged Architecture Library (PALcode) which is a set of subroutines that are specific to a particular Alpha operating system implementation. The PALcode exists of specific extensions to provide access to low-level hardware used to implement the Alpha architecture. One of the goals of Alpha was that microcode would not be necessary for a practical implementation. However, an architected interface, which is consistent across an entire family of machines, is desirable. This is one of the reasons for developing the PALcode concept. The PALcode concept is depicted in figure A.6.

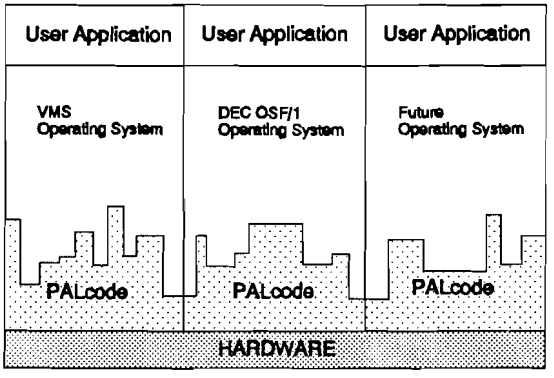


Figure A.6 Alpha PALcode.

The subroutines in PALcode which are written in standard machine code with some implementation-specific extensions to provide access to low-level hardware, provide operating system primitives for:
- Instructions that require complex sequencing as an atomic operation.
- Instructions that require VAX-style interlocked memory access.
- Privileged instructions.
- Memory management control (including translation buffer (TB) management).
- Context swapping.
- Interrupt and exception dispatching.
- Power-up initialization and booting.

- Console functions.
- Emulation of instructions with no hardware support.

One version of PALcode lets Alpha run the full OpenVMS operating system by mirroring many of the OpenVMS VAX features. Another version of PALcode lets Alpha implementations run the OSF/1 operating system by mirroring many of the RISC ULTRIX features. This illustrates the fact that PALcode makes Alpha an especially attractive architecture for multiple operating systems.

There are two functions which must be implemented in PALcode, not directly in hardware, to support the replacement of Digital-supplied PALcode with an operating system specific version. These are:

- Translation buffer fill, in this way complete flexibility in implementing page table structures is allowed.
- Process structure, so any process context switch routine is allowed.

Addressing and protection are strongly dependent on the specific PALcode version. However, some aspects hold generally.

Alpha physical memory is divided into four regions, based on the two most significant, implemented, physical address bits. Each region's behaviour can be described in terms of its coherency, granularity, width, and memory-like behaviour.

A system may choose to include a virtual instruction cache or a virtual data cache. A system may also choose to include either a combined data and instruction translation buffer or separate data and instruction translation buffers. PALcode mechanisms must be available to operate on these translation buffers.

The Alpha architecture allows a processor to optionally implement address space numbers to reduce the need for invalidation of cached address translations for process specific addresses when a context switch occurs.

We can state that the Alpha architecture provides optimal flexibility in supporting different operating systems through its PALcode. However, one could argue that this flexibility is achieved by PALcode that is in fact a part of the operating system and not of the architecture.

At the other hand, an argument for considering PALcode as part of the architecture is the fact that it contains very specific, hardware dependent instructions (this can be compared to the BIOS concept in personal computers, every hardware design requires its own specific BIOS).

B Secure Booting and Paging

In this appendix a general secure booting mechanism based on a public-key encryption system will be presented. Assumed is a network consisting of nodes that have a processor attached to some local memory and possibly a disk. The unit of transfer between nodes and between a node and disk is a page.
Secure booting is necessary in distributed systems because kernels and servers are subject to malicious modification (e.g. a user may load its own modified kernel whereafter several secrets could be stolen). Secure paging involves protecting the information sent between nodes and between a node and storage. The mechanism described in paragraph B.2 was designed to protect pages but can be used unaltered to protect arbitrary information. The mechanisms described have been implemented in the MONADS system (paragraph 4.2).

B.1 Secure Booting

The secure booting mechanism assumes two trusting third party entities, the hardware manufacturer (HWM) and the operating system manufacturer (OSM), both equipped with a pair of keys. The secure boot mechanism consists of three parts:
1. Preparation phase.
2. Boot process.
3. Authentication process.

The preparation phase consists of the three steps:
1. After being manufactured, the CPU generates a secret (SK_{CPU}) and public key (PK_{CPU}).
2. The public key is made available to the hardware manufacturer. The HWM then creates a CPU signature by applying its secret key (SK_{HWM}) to the unique identity of the CPU (Id_{CPU}) and the CPU's public key. The created signature is stored in the CPU key storage, a hardware extension which also holds the secret key of the CPU. In this way the CPU can obtain a certified pair of keys without showing its secret key to anyone. The CPU certificate is visualised in figure B.1.

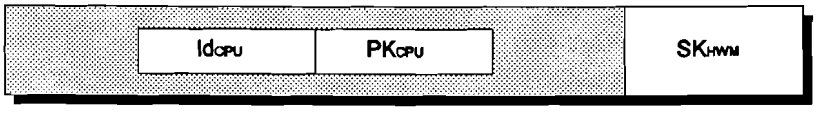


Figure B.1 CPU certificate.

3. The operating system manufacturer creates a secure boot module consisting of the required code and the unique operating system identity (Id_{OS}), encrypted by the secret key of the OSM. It is assumed that the public keys of the HWM and OSM can be made securely available to the CPU (this can be done by storing them in an EPROM). The boot module is visualised in figure B.2.

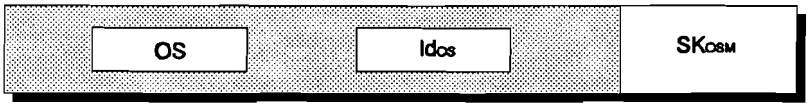


Figure B.2 Boot module.

The preparation phase is now complete. Booting can begin and the boot process consists of three steps:

1. The boot module is loaded into main memory and the signature of the OSM is verified with the public key of the OSM. In this way the integrity of the boot module is verified.
2. If step 1 proves to be successful, the CPU assigns a pair of keys to the operating system and stores them in operating system key storage which is a dedicated part of main memory.
3. The CPU now creates a boot certificate as depicted in figure B.3.

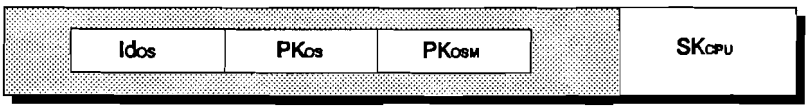


Figure B.3 Boot certificate.

The boot certificate consists of the unique identity of the loaded operating system, the public key of the OSM, and the in step 2 assigned public key of the operating system. This certificate is encrypted by the CPU's secret key.

The operating system now controls the CPU and is the only entity which has access to the CPU's key storage. This is achieved by making it unaccessible until the next hardware reset.

The boot process is now complete. The third part of the mechanism, the authentication process serves to check trustworthiness of a running system to a user working with it or to another system willing to cooperate with it. The authentication phase consists of four steps:

1. A random number is sent to the operating system. The operating system signs it with its secret key and returns it together with the CPU and boot certificates.
2. The public key and unique identity of the CPU are extracted from the CPU certificate by using the HWM's public key.
3. The CPU's public key is now used to decrypt the boot certificate which enables the testing entity to obtain the authentic public keys of the operating system and the OSM used during the boot process. A comparison of the OSM's public key and the already known authentic public keys of trustworthy OSMs shows the authenticy of the OSM.
4. Finally, the signature of the operating system on the random number is verified, which indicates that the operating system is still active and that no replayed message has been sent.

The authentication phase may then be employed by the tested system on the testing system to achieve mutual authentication.

B.2 Secure Paging

The described booting mechanism increases the trustability of the kernel but leaves information residing outside of main memory vulnerable to active and passive attacks. Communication security is increased by encrypting page transfers across the network and between main memory and disk. One solution serves both page transfer across the network as well as transfers between memory and disk if the rest of the network can be seen as one large disk (as in the MONADS system). A mechanism,

called secure paging is presented which relies on both DES and RSA encryption (for a description of both cryptosystems paragraph 2.3.2 is reffered to). This mechanism is based on the fact that every kernel has its own public and secret key as well as public keys of other kernels as a result of the described booting mechanism. The secure paging mechanism is visualised in figure B.4.

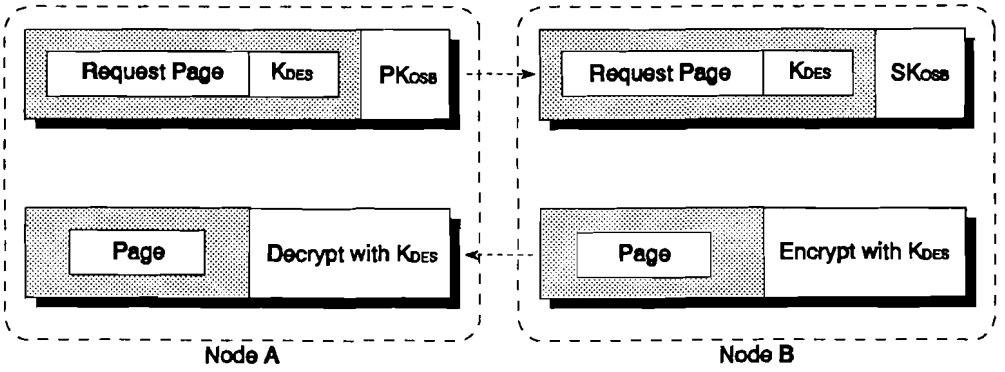


Figure B.4 Secure paging mechanism.

The secure paging mechanism can divided into three parts:

1. The virtual memory manager on node A detects a page fault. It then generates a DES key (K_{DES}), appends it to the message requesting the missing page and encrypts this message by using the public key of node B (PK_{OSb}) were the missing page is located. The resulting message is then sent to the remote page fault handler on node B.
2. The remote page fault handler on node B receives the message from node A and decrypts it with its secret key (SK_{OSb}). It then encrypts the page with the DES key and sends it back to node A.
3. The page fault handler on node A decrypts the page with the DES key and places it into its proper location.

The same mechanism can be used to store and retrieve pages on/from local disks. There are two reasons for using a combined DES/RSA approach:

1. Encryption rates of RSA implementations available today are insufficient. They would result in considerable remote page fault performance deteriorations.
2. RSA cannot guarantee to maintain the length of encrypted information. This incurs problems when writing encrypted pages to disk.