RESOURCE ACCESS AND PROTECTION IN THE SOMBRERO PROTECTION MODEL, SOFTWARE PROTECTION DATA STRUCTURES and HARDWARE RANGE PROTECTION LOOKASIDE BUFFER

ASU 64-bit OS Group White Paper 2

Alan C. Skousen and Donald S. Miller[1]

Computer Science and Engineering Department
Arizona State University
Tempe, AZ 85287-5406

Technical Report TR-95-013[2]
January 1996 (revised May 7, 1999)

ABSTRACT

In a single address space operating system (SASOS) all objects in memory are visible to all other objects by *default*. As a consequence some means is needed to protect the privacy of object contents and prevent the occurrence of accidental or deliberate corruption of the contents of objects not participating in the current thread of execution. Every access to memory must be tested to determine if the current thread of execution has legal access to the instructions or data at the memory location. The Range Protection Lookaside Buffer (RPLB) is a hardware device which implements the required protection and enforces the communication constraints between protection domains in the Sombrero operating system. We have named Sombrero protection domains 'ubiquitous' protection domains, or UPDs, because of the manner in which they enforce protection.

The benefits realized by this model include the virtually costless domain switching enabling traditional InterProcess Communication (IPC) to be replaced by much faster InterDomain Communication (IDC). It also includes an object based rather than page based protection mechanism. Object based protection is important to simplify access and consistency management as well as generalizing protection granularity.

We have two objectives in writing these White Papers. First, we want to get our ideas out on the net and hear what you think of them. Second, they serve as an information basis for proposals we are making to industry to obtain hardware and software platforms on which to build a prototype and for proposals to government and industry for funding for the prototype effort.

---

[1] The authors can be reached as follows:

e-mail: alan.skousen@asu.edu and donald.miller@asu.edu

phone: (602)-965-5935

fax: (602)-965-2751

[2] Web page for technical reports: HTTP://www.eas.asu.edu/~sasos

1. INTRODUCTION AND OVERVIEW

Resource access and protection in Sombrero is described in six parts. In this section we present an introduction to some of the single address space concepts that motivate our design. The next section contains an overview of the basic resource access and protection concepts in our model including a formal description of it. This section also introduces the associated software data structures and required protection hardware. Section 3 presents Sombrero operating system abstractions for resource access and protection including the principle data structures and their operation. Section four discusses Sombrero hardware for resource access and protection including the design and operation of the hardware Range Protection Lookaside Buffer (RPLB). Fifth, we delve more completely into Sombrero's protection model and look at it from a number of different aspects. Finally, we present a brief summary of Sombrero's protection structures and the way they take advantage of the properties of a very large single address space.

The Sombrero protection model and software and hardware protection structures were motivated by our observations of the properties of a very large single address space in addition to some reasonably widely held views on how a modern operating system should be built. We list some of the principle ones here in order to motivate the rest of this White Paper.

1.1 Properties of a Very Large Single Address Space and their Implications

1. In a process-oriented operating system processes are protected from one another because they cannot address data or execute code in another process' virtual address space, keeping one from corrupting the other. The single address space strategy does not provide this natural barrier, rather all code and data are visible to all of the threads of execution as under a single multithreaded process, even across the network. Protection in a SASOS must be done with protection domains laid atop a single virtual space forming hard walls between principals'[3] activities as effective as the separate address spaces in a process oriented system. Every memory access must be checked to enforce the protection domains. This implies a need for a set of data structures that organize per-object virtual addresses and access rights by protection domain and as well as hardware which use the data structures to enforce the protection.

2. The efficiency of using a single address space and the introduction of protection hardware make it more feasible for protection to be independent of the storage mechanism. This has two very important ramifications. First, protection can be provided *directly by the hardware* to the logical objects that modern high level object-oriented software deals with. Second, the granularity of protection need no longer be constrained by virtual address mechanisms, e.g., it no longer need be related to page size. Object granularity can range from bytes to terabytes or more and can be adjusted dynamically to respond to concurrency demands.

[3] We use the term principal to name the ultimate entity whose access rights we wish to describe. So a principal could be an ordinary user, the superuser or some other privileged kernel-level or user-level server. Operationally, in a single address space, all computation is performed by sets of one or more threads acting on behalf of each principal. In a process-oriented system these tasks are performed by processes and every process has an associated 'real' user id, the principal in these systems. This definition is somewhat different from [Wilkes and Sears].

3. In general it is desirable to allow different applications to have different views of the same data as demonstrated by various database schema. In particular, it may be desirable for applications to access different parts of the data with different protection rights. This in turn suggests that it would be useful for the region of virtual addresses that comprise memory objects to be variable sized or even noncontiguous with possibly different access privileges.

4. In a single address space there can be a single kernel protected place where the data structures representing authorized resource accesses for principals and authorized principal accesses for resources can be kept. We maintain that this information, normally kept in distinct capability-like and ACL-like data bases can be merged. Moreover, it's important to note that this merged information can make use of the fact that in a single address space both resources and protection domain entry points can be named permanently by virtual addresses. In addition to the high level software that defines management policy and authorizes the contents of protection matrices and their related data structures (we are not concerned in this document with management policy), the protection data contained in an access matrix and enforced by the system are normally kept in two layers. The first is a set of directly useable high level data structures including capability lists and/or access control lists. The second layer consists of the protection information kept in page table entries and/or segment descriptors. The lack of process boundaries and the global nature of virtual addresses in a single address space (i.e. the removal of translation requirements between entities) allow both sets of information to be merged in a single set of data structures. The resulting data structures can be searched directly by hardware to load protection information.

5. Reducing the cost of software development can be accomplished by simplifying the number of abstractions which the programmer has to manage. In this light it is desirable to get protection "out of the way" and invisible to application processes. We can also reduce the cost of inter-domain communication by reducing the participation of the kernel. This means, that after an initial check, the kernel should, wherever possible, no longer be involved in cross domain transfers of control or data accesses. We use the term *ubiquitous* and call our protection domains *ubiquitous protection domains* to confer the notion that protection is always being provided for both intra-domain and inter-domain operations but users and their programs are not aware of it happening, i.e., it is *transparently* provided by hardware and the kernel managed protection data structures. Domain switching and protection are *background* operations, not seen by user level programs. Moreover, these operations occur in parallel as against serially, i.e., no domain switching CPU instructions need be required. Access descriptors available to the protection hardware provide the necessary information and overrideable methods of the protection domains and memory objects provide flexibility for protection semantics.

6. The virtual addresses in a single address space uniquely name all persistent data on secondary storage including the execution locations of the code and data of every program. A single address space operating system can take advantage of this fact by eliminating any separate I/O data address spaces, the inter-address space mappings, and the file system that manages them. It can also provide utilities that simplify the binding of modules to permanent addresses and for naming, linking to and loading these modules.

7. The process boundaries that encapsulate execution state, principals and protection attributes are not present in a single address space. Because of this, we take the view, as do many SASOS designers, that processes per-se do not exist in a single address space. Stated differently, neither threads nor principals nor the access rights associated with these principals are bound by a dedicated and temporary virtual address space as occurs with processes in a process-oriented system. Hence these resources may be freely reassociated as may seem appropriate from time to time. The resulting orthogonalities, combined with other single address space properties, provide considerable opportunity for performance and functional enhancements. For example: 1) Object persistence becomes natural. 2) Objects can be assigned access rights and these pairs located independently of protection domains instead of being remapped in every domain. 3) Threads executing on behalf of principals can do so at addresses independent of the addresses associated with the protection domains owned by its principle, i.e. threads can travel becoming messengers and replace message passing and context switching with an unbroken thread of execution.

8. Virtual addresses have the same meaning anywhere on a local area network. Globally shared memory is independently supported on every node by the local demand paging hardware. An important consequence of this is that not only can the kernel itself be distributed, but by distributing the kernel and its data structures we can create the abstraction of a single large node across the network. This has important implications for simplifying the programming effort. We take the view that considerable performance and functional advantages can be found in distributing the kernel. The tradeoff of needing more complex mechanisms to handle replicated kernel structures is well worthwhile. We also maintain that virtual memory management hardware support for a single address space can be used to support a "higher level cache" mechanism that provides considerable fault tolerance across a network.

9. The properties of a single address space and the fact that it is supported by independent virtual memory management on every node can be used to simplify considerably the handling of networkwide consistency semantics. In particular, this implies that very simple data structures and pointers can be used to support coherency under the replication that is necessitated by network latency. Simple graph structures can be used to locate the most recent copy of a modified page and a token can be used to validate access. Version numbers and logical page pointers are not necessary. Put differently - in a single address space, per-node demand paged virtual memory management can be used to simplify and make transparent the support for copy set management when there are multiple readers, making the most recent copies of modified pages available to readers and writers and initiating page invalidate copy set multicasts when pages are modified.

10. Protection data structures for a single address space operating system can be designed so that they are naturally extensible across a network. This means that the methods and meta-data associated with control blocks can be designed to efficiently support coherency and synchronization under the additional requirements of network latency and replication. These should be designed into the protection system from the outset.

1.2 Summary

The key features of a single address space OS that distinguish it from process-oriented multiple address spaces OSs are a unique, network wide, persistent virtual address namespace and the non binding of resources, execution state, principals and protection attributes to each other by a dedicated and temporary virtual address space as occurs with processes in a process-oriented system. In such a system every memory resource can be considered as an object named by its virtual address and operating system data structures can be designed to associate access rights with regions in memory objects. These memory object regions can be arbitrarily placed in multiple protection domains each associated with some principal. This allows not only global access by principals but global access control by the resources. In addition, since threads are not bound to code or protection domains, it is useful to bind varying amounts of state to a thread as it moves between domains to extend register storage. Finally, individual virtual addresses can be associated with protection domain entry points and these used to define the protection domain switches which are ubiquitously supported by the hardware.

The single namespace uses virtual addresses to name resources, protection domains and thread state. In such a system there exist principals represented by threads and objects which carry out their purpose. Protection domains are the access the former have to the latter. Inserting and deleting *access descriptor* data structures can be used to determine this access.

All of his can be done by adding relatively simple virtual address-based protection hardware to that ordinarily required for virtual memory support. CAD designs for this protection hardware have already been generated and simulation tested as student projects in the ASU CSE department ULSI design laboratory.

This White Paper discusses the *local node* issues of resource access and protection in Sombrero. Resource access and protection issues that come up to make Sombrero work properly across a network, in particular those mentioned in items 8-10 above, are discussed in depth in White Paper 10. They will only be referred to in this White Paper when they affect local node design or tradeoffs.

The above observations have led us to the software and hardware designs described in more detail in the following sections. Considerably more detail on our observations and views on protection, persistence and naming in a very large single address space can be found in White Paper 16 [4] and an introduction to the Sombrero operating system can be found in White Paper 0 [] and an overview of the entire Sombrero operating system in White Paper 1 [1]. White Paper 4 describes how we plan to emulate an extended 64-bit RISC processor with an RPLB [3] and White Paper 12 presents a requirements specification and VHDL design for the RPLB [2]. More on Sombrero features related to integrating persistent storage into the single very large address space can be found in White Paper 7 on the Sombrero Nameserver, linker, compiler and loader [7].

White Paper 10 delves into the operation of Sombrero's protection model across multiple nodes [8] and White Paper 6 contrasts Sombrero's protection model and design with that of most of the known SASOS's [9].

2. OVERVIEW OF RESOURCE ACCESS AND PROTECTION IN SOMBRERO

2.1 Motivations

Our objective in designing Sombrero is to propose the most simple possible resource access and protection computing paradigm which preserves the services required in modern computers. Our design makes new approaches possible and makes possible the removal of workarounds which were designed to overcome past hardware limitations. Such workarounds include separate virtual address spaces for every task, file systems and interprocess communication mechanisms. These workarounds resulted in a complex and costly setting for designing new applications. They also consume much of the computing power in constructing, destroying and translating between temporary entities such as processes which to now have been acceptable because of the constant advance in processing power. Our objective is to move more of the complexity of information management design away from the user level programmer, down to the operating system which is the entity least frequently dealt with by designers. Programmers should not be required to manage communications, sharing, persistence or protection issues, rather these functions should be available by default. The advent of the 64 bit processor along with advancing storage and communication hardware provides new opportunities by making a single address space operating system practical. The simple user level abstraction of a CPU, memory and I/O (without the file system and other separate namespaces) is a more productive environment in which to program. The need for namespace translation, interprocess communication and a file system creates costly unnecessary complexities. Our model eliminates these and imposes protection ubiquitously in order to maintain a simple computing paradigm. We expect by these means to reduce the cost of application design and direct more of the available processing power toward improving productivity.

2.2 Basic Abstractions

The Sombrero d-Kernel (distributed kernel) has four major abstractions: Memory Objects, Threads, Principals and Ubiquitous Protection Domains (UPDs). The memory object represents all memory resources and the UPD represents concurrently accessible collections of these memory resources as well as access to other protection domains. Threads represent the unit of execution. Principals represent the user and ownership by the user. In addition there are two parts to the UPD: General protection domains define what is accessible by all its associated threads and carrier protection domains define what is accessible by an individual thread of execution regardless of which general protection domain it is associated with. A UPD then is the union of a general protection domain and a thread's carrier protection domain.

2.2.1 Memory Object

Sombrero's basic storage abstraction is the *Memory Object*. Informally a memory object is characterized by its range. *Memory ranges* are distinct, variable sized, contiguous allocations of the virtual address space. A memory range may encompass multiple pages or only part of a page. An individual memory object is bounded by its range and is a fixed sized string of bytes with no other structure. A memory object is named by the range's lowest address. Memory objects are accessed via memory regions (note a region is not a range) which are associated with an access permission. *Memory regions* are possibly noncontiguous sets of virtual addresses within a range that have a particular protection attribute. One or more protection attributes, R, W and X, are associated with a memory region and the same protection attribute can not apply to more than

one region in a range, i.e. protection attributes are non-overlapping. So each memory object has up to three regions and the virtual addresses in regions can overlap but the protection attributes can not. In Sombrero, ultimately all memory resources are represented and named by memory objects. The distinction between region and range allows memory objects to be accessed differently for each of the three different access types.

Following is the definition of a memory object and its control blocks. All control blocks are stored in the d-Kernel and are named by their starting virtual address.

2.2.1.1 Memory Object - MO

Fundamental storage abstraction, variable sized, not related to virtual page granularity.

- Formally a Memory Object is an ordered pair (*low*, *hi*) which defines its range:
  - *low* ≡ The lowest virtual address in the range of the memory object. The name of the MO.
  - *hi* ≡ The highest virtual address in the range of the memory object..

The Memory Object is the entity managed by the Lampson matrix described in sec 2.5. This is represented as: MO(*low*, *hi*) or as MO or *range*.

2.2.1.2 Memory Object Control Block - MOCB

Access to a memory object is managed by a Memory Object Control Block - MOCB.

- Formally a Memory Object Control Block is an Ordered triple (*range*, *rcb-set*, *acl*):
  - *range* ≡ A memory object..
  - *rcb-set* ≡ A set of 1 to 3 Region Control Blocks.
  - *acl* ≡ List of GPDs and CPDs which can access this MOCB - an ACL (Access Control List).

The MOCB is the connecting entity which provides the bridge between the Lampson and RPLB access matrix models. It is represented as: MOCB(*range, rcb-set, acl*) or just MOCB.

2.2.1.3 Region Control Block - RCB

A MOCB has up to 3 RCBs. Each RCB is used to manage a subset of the *range*. In practice the region and range are often the same.

- Formally a Region Control Block - RCB, is an ordered pair (*R, P*):
  - *R* ≡ A region or set of Virtual Addresses bounded by *range*.
  - *P* ≡ Access Permission Set: {R|W|X}.

P is Read (R), Write (W), or Execute (X) access permission or any combination of the three, however, each access permission may appear at most once in any MOCB(*range, rcb-set, acl*). The RCB is the RPLB operational entity (sec 2.5). This will be represented as: RCB(*R,P*) or just RCB.

In summary a memory object is represented by a *Memory Object Control Block (MOCB)* and a set of up to three *Region Control Blocks (RCBs).* These collectively define a storage resource and the access privileges to regions within it. It should be noted that the Memory Object is the only abstract storage structure in Sombrero. This is justified because memory objects are not specialized and have no imposed internal structure such as headers or access lists, rather these

structures and the corresponding information are provided by the MOCBs, RCBs, ADs[4] and other control blocks.

2.2.2 Protection Domain Objects

Sombrero's basic protection abstraction is the *Ubiquitous Protection Domain (UPD).* It is an abstract entity describing the set of objects and domains that are accessible by a thread operating within the UPD. It is so named because unlike other protection schemes not only is the protection mechanism transparent, the protection domain switching mechanism is also transparent to the programming environment. This permits the computing environment to be viewed as though the protection boundaries did not exist. An example of a similarly transparent entity is the virtual paging system, it too is ubiquitous as it maps the pages of virtual memory onto the more limited number of physical pages and yet the programmer and user need not be aware of its participation in order to use it.[5] Sombrero has two kinds of protection domains, *General Protection Domains (GPDs)* and *Carrier Protection Domains (CPDs)* named by the virtual addresses of the control blocks that represent them. Slightly simplified, GPDs contain the code, data and GPDs accessible to threads in the currently invoked GPD and CPDs contain the data and the protection domains privately accessible to the currently executing thread. The UPD is the union of the GPD and CPD. Following are the definitions of the GPD and CPD.

2.2.2.1 General Protection Domain - GPD

A set of Memory Objects that can be accessed and their associated access permissions and a set of Protection Domains that can be Switched into and their associated switch permissions.

- Formally a GPD is an Ordered Triple (*M,S,A*):
  - *M* ≡ Set of Accessible MOCBs - a PAL (Protection domain Access List[6]).
  - *S* ≡ Switch Permission set *to* other GPDs - an additional PAL for switches.
  - A ≡ List of GPDs and CPDs which are permitted to access this GPD - an ACL (Access Control List).

The two access lists *M* and *S* defined above are collectively known as the Protection Access List or PAL. For later use this will be represented as: GPD(*M,S,A*) or just GPD.

A GPD is represented by a General Protection Domain Control Block (GCB). A list of Access Descriptors (ADs) collectively implement *M, S* and *A* and determine the set of resources and domains accessible to this protection domain and the set of domains which may switch into this domain. Each GPD is owned by a principal.

---

[4] AD - the Access Descriptor is the link which connects protection domains and their resources.

[5] Ubiquitous is quite different from virtual. Virtual whatever makes something look like a model of whatever. For example virtual memory makes pages randomly ordered in physical memory look like a linear memory and virtual terminal software makes an arbitrary terminal appear to its caller like a particular type of terminal. Ubiquity-producing software makes something that does something all of the time invisible (i.e., look like nothing!)

[6] A PAL is the list of resources accessible to a Protection Domain. Sombrero's PALs contain the same sets of <Resource, Rights> pairs accessible to a subject domain as do the capability lists or C-Lists of many systems. Because Sombrero does not use these entities in the same way, we do not refer to them as capabilities. The control blocks, access lists and access descriptors that contain this information are not "presented" by one domain to another domain to obtain access to it or its resources and the notion of a protected pointer either safely stored in a table in kernel space or passed around in encrypted form in user space that contains this information is inapplicable. In Sombrero, the PALs are stored in a single interlinked set of control blocks and access descriptors, the contents of which are determined by higher level policy software and which are searched on a protection fault or for rights revocation by the kernel.

2.2.2.2 Carrier Protection Domain - CPD

The purpose of the CPD is to extend the available state of a thread beyond the register storage by binding part of the address space. Assigning Memory Objects, such as its stack, directly to a thread[7] enables the thread to act as a carrier and transport data between the GPDs. Execute access cannot be associated with the state carried with a thread CPD. This is to prevent Trojan horses and to prevent the violation of the extended register abstraction which is able to store only state not instructions.

- Formally a CPD is an ordered Quadruple (*M,S,G,C*):
  - *M* ≡ Set of Accessible MOCBs - a PAL (Protection domain Access List).
  - *S* ≡ Switch Permissions *to* change GPDs - an additional PAL for thread switches.
  - *G* ≡ General Protection Domain - the context the Thread is currently executing in.
  - *C* ≡ Current CPU State of the Thread.

For later use this will be represented as: CPD(*M,S,G,C*) or just CPD.

A CPD is bound to a thread and is represented by a Thread Control Block (TCB). A list of Access Descriptors collectively implement *M* and *S* and determine the set of resources and domains accessible to this thread.

2.2.2.3 Switch

In Sombrero the right to enter another protection domain to execute code or to access data is called a Ubiquitous Switch or just a Switch. A Switch entry stored in an Access Descriptor data structure provides the needed virtual address.

- Formally a Ubiquitous Switch is an ordered triple (*V, G, P*):
  - *V* is a virtual address.
  - *G* is a GPD name.
  - *P* may be any combination of read, write and execute.

The operation is: An access to V causes the current protection domain to be switched to G if the attempted access matches *P*. *P* is typically execute permission. For later use this will be represented as: Switch(*V, G, P*) or just *switch*.

2.2.3 Thread Objects

Thread objects are the unit of execution in Sombrero. They are the CPD objects listed above. Both are represented by the Thread Control Block (TCB). Thread Objects and CPD objects will be treated as two names for the same thing.

Protection Domains \ Resources	**MO1**	**MO2**	**MO3**	**MO4**	**Ptr1**	**Plt2**	**PD1**	**PD2**	**PD3**
PD1	read	R/W						Enter	
PD2			read	RWX	write				
PD3					write	write			

Figure 1 Lampson Access Matrix: PD - protection Domain, MO - Memory Object Ptr - Printer, Plt - Plotter.

2.2.4 Principal Objects

A Principal object represents the user or a service and binds the activities carried out on behalf of the user.

- Formally a Principal is an ordered pair (*Sn*, *O*) where:
  - *Sn* ≡ A Session. A session represents the bindings of a user to I/O devices, a user ID and other system resources.
  - *O* ≡ An Owner list. The owner list is the set of Sombrero resources owned by this principal.

For later use this will be represented as: PO(*Sn*, *O*) or just PO.

2.3 Memory Object and Switch Rights Allocation

Memory object allocation is carried out in the Sombrero kernel by allocating a range of memory, constructing the MOCBs including specifying their regions and access rights within the range based on specifications sent from the application, and placing the MOCBs in the protection access lists (PAL) of the appropriate UPDs. Switch rights are provided by placing the name of the target GPD in the protection access list of the UPD being given the Switch right. These are discussed in Section 3.

2.4 Execution and Domain Switching

During normal (intra-domain) execution the code and data of the memory objects in the current General Protection Domain and the data in the memory objects in the Carrier Protection Domain are accessible to the executing thread. Cross domain transfers of control and data accesses are determined by the Switch permissions possessed by the current General and Carrier Protection Domains. In the next three sections we present an overview of how this works in Sombrero. First an access matrix which embodies the Sombrero protection model is described and contrasted with the traditional access matrix introduced by Lampson [LAM71]. Next we introduce the Sombrero software protection data structures that store the Sombrero access matrix data and lastly we briefly overview the Sombrero hardware which provides runtime protection.

2.5 Protection Matrix

The traditional access or protection matrix which we will call a Lampson Access Matrix can be viewed as a vector function with arguments protection domain (PD) and resource which returns the access rights allowed. In the access matrix corresponding to this function PDs are listed as rows. The computer's resources, such as memory objects or printers or, in fact, other protection domains when viewed as resources, provide the columns. The entry at the intersection of a row and a column contains the access rights the PD has to the resource. This is the value of one element of the vector function. For memory objects any combination of *R/W/X* access can be present in an element or, when columns represent protection domains, the value *Enter* can be present, which specifies an allowable switch from the row protection domain to the column protection domain. This function which we will call the Lampson Access Function is:

Access_rights = **Lampson(**PD, resource**)**

---

[7] Trust relationships are managed independently from domain switches and can be implemented using proxy threads and ½ context switches. A ½ context switch saves a thread's context and then changes the name of the running thread to the name of the proxy thread effectively performing a CPD context switch.

where *Access_rights* are some combination of *R/W/X* or *Enter*.

Protection Domains	All 2^64 Virtual Addresses/ Access Rights for each address														
	$VA_{0000000000000000}$			$VA_{0000000000000001}$			. . . . .			$VA_{FFFFFFFFFFFFFFFE}$			$VA_{FFFFFFFFFFFFFFFF}$		
	R	W	X	R	W	X	R	W	X	R	W	X	R	W	X
GPD-1	**A**											**A**			**A**
GPD-2	**S-3**									**A**	**A**		**A**	**A**	
GPD-3	**A**	**A**		**A**	**A**							**S-1**			
GPD-4															

Figure 2 RPLB Access Matrix for a 64 bit virtual address system - A for *ACCEPT*; *REJECT* is indicated by a blank entry and S for *SWITCH* which returns a GPD name.

Entries in a row of the Lampson Access Matrix provide a Capability list (C-list) for a particular protection domain while column entries provide an Access Control List (ACL) for a particular resource or protection domain. Figure 1 shows a Lampson Access Matrix.

The Lampson Access Matrix works well for policy management of a computer's resources but, as we will see, does not supply the information needed by hardware mechanisms during instruction execution. To show this, consider the above Lampson function with UPD in place of PD and Memory Object in place of resource. When a computer needs to verify the validity of a memory access, assuming it has a UPD register, it can provide a value for the first ordinal, the current UPD, but not for the second ordinal, the resource. What the CPU can present is: 1) a value for the UPD, 2) the virtual address for the memory access and, 3) the type of access, R, W or X. We can therefore represent whether or not a memory access should be allowed by a 3-dimensional function, with arguments protection domain, virtual address and access type, that returns a value indicating whether an access is valid or a switch may occur. We call the function that defines this relationship the *RPLB Function*. It is defined as follows:

Access_allowed = **RPLB(**UPD, VA, Access**)**

where *Access_allowed* is *ACCEPT*, *REJECT* or *SWITCH* GPD name, and the name RPLB comes from the hardware used to implement the function (see sec. 4).

Corresponding to the RPLB Function there is an access matrix which we call the *RPLB Matrix*. The RPLB Matrix is three dimensional and has the UPDs as rows as before but the columns are the set of all valid virtual addresses and the third axis is the set of all valid access rights: R, W, X. This Access ordinal produces three distinct VA - UPD planes. At the intersection of the ordinals of the RPLB matrix is the value which determines the validity of an attempted access by the CPU. Figure 2 is a 2-dimensional rendering of a 3-D RPLB matrix for a 64-bit virtual address space.

The RPLB protection matrix like the Lampson matrix is two matrices in one: 1) an access matrix and 2) a switch matrix. The access matrix returns a Boolean ACCEPT or REJECT on a hit while the switch matrix returns a GPD name on a hit unlike the Lampson matrix which returns R,W,X on a hit or enter which allows entry into another domain on a hit. The access matrix

takes precedence over the switch matrix. In an actual implementation there would be three units: 1) the IRPLB to support the instruction stream, 2) the DRPLB to support the data stream and 3) the SRPLB to implement the switch matrix.

The functional difference between an RPLB matrix and a Lampson matrix is that for the benefit of the CPU, which can only see the world as addresses (not objects), permissions and domains, the RPLB assumes that all objects are one byte in length represented by a unique address. An attempt to access the object is either accepted, rejected or the domain context is changed ubiquitously. In particular, on a switch the PD context change is not performed by a dedicated instruction which initiates a change of context, rather any kind of allowed attempted access returns the name of the new GPD which is then stored in the Current Context register (GPDBR see sec. 4). This is what makes the switch ubiquitous. Put another way, instead of heading the RPLB matrix column with a PD, the PD is the value stored in the matrix. Put yet another way, instead of the protection function returning *enter* using the arguments PD-1 and PD-2 as in the Lampson matrix, the RPLB function returns GPD-2 using the arguments GPD-1, VA and *P*. The effect is that any instruction which accesses memory can be used to perform a GPD context switch without requiring the complexity of domain boundary management to be a part of the programming effort.

The RPLB third dimension corresponding to the access rights is separated into three distinct planes and permits each access right to be managed independently, i.e. each access right can manage a separate view of the memory object. Sombrero uses this separation to simplify the implementation of the management of network wide data consistency and access.

An application of the ubiquitous protection environment could be as follows: A program creates an object complete with methods using the C++ *new* function. An option of the Sombrero C++ *new* function allows the object to be created as a separate protection domain. Access to the object is as though it were local without IPC, RPC, LRPC, PIPES, SWITCHES or any other communication barrier. It's methods are accessed normally using '->' or '.' and the underlying object code is the usual machine control transfer instruction: call, return, jump, etc. Furthermore, any other program or user may also be granted equally ordinary control transfer instruction access to the object as though it were created on its local heap. Even more interesting, this ubiquitously shared object is available network wide by default in the same ordinary control transfer instruction manner to all who are granted access. After the initial access all further access cost is the same as any non-UPD crossing control transfer instruction. We estimate that the simplicity of this environment will have enormous implications to reducing the cost of program development and performance by making programs and program performance independent of the nature of the objects (e.g. local program heap allocated object or GPD shared object) they access.

It's worth noting that the RPLB matrix and RPLB function are comparable to the 'protection matrix' and 'protection function' of the garden variety TLB, except that the TLB does not provide a switch function, the TLB protection matrix columns are page numbers derived directly from the virtual address and the TLB PD matrix ordinal is a single row implied by the current process (this is because, in a process-oriented operating system, a PD is permanently associated with the virtual address space in which execution is currently taking place). It is becoming more common to also include an 'address space identifier' with each TLB entry to improve the efficiency of

context switching. An address space identifier is another name for a protection domain name and protection matrices for such TLBs would have more than one PD row. The RPLB differs from the TLB in that it is designed to manage the complete range of object sizes rather than limiting itself to managing page groups as the DEC Alpha and MIPS R10000 do. To this end it is disassociated from the task of translation. This is practical in a SASOS since there will only ever be one address map rather than one for each process. A consequence of this is that the need to store physical tags in the level 1 CPU cache (the synonym problem), to keep from flushing on every context switch, is overcome. A SASOS oriented CPU using an RPLB can be completely virtual address oriented allowing translation to be moved out of the CPU to the system bus where it can be shared in a multiprocessor environment. The PLB [Ko*92] design suggests a similar strategy.

Traditional virtual memory systems provide hardware protection to pages in the virtual memory to prevent unauthorized access, via a Translation Lookaside Buffer (TLB). In the Sombrero single address space operating system the protection abstraction is separated from the address translation and paging abstractions and consequently is no longer a part of the TLB function. The virtual pages are no longer flagged as Readable (R), Writeable (W), Executable (X) or Inaccessible rather they are simply Accessible or Inaccessible, the TLB simply performs a virtual to physical translation for pages and tracks dirty bits. Pages are no longer the unit of protection, instead access rights are enforced on the objects in which the pages participate by the RPLB.

2.6 Software Resource Access and Protection Data Structures

Sombrero's software protection data structures (Protection and Resource Access List - PRAL) consist of PALs and ACLs and provide a repository for resource access and protection policy information determined by higher level software in a form that is directly usable by the lower level RPLB protection hardware. The RPLB hardware uses the PRAL to export to and project object-oriented access and protection onto user space. The PRALs, which are described in section 3, consist of sets of control blocks representing protection domains and memory objects interlinked by access descriptors. They define the access privileges of threads executing in protection domains to memory objects and to other domains. In addition they define and store Sombrero virtual address allocation, relate resources to virtual addresses, relate principals to protection domains and store the necessary information to support a very simple effective networkwide replication and coherency mechanism.

2.7 Hardware Range Protection Lookaside Buffer (RPLB)

The Sombrero RPLB is a CPU-resident hardware device that implements the RPLB function. For RWX access, somewhat simplified, it is an associative memory with each entry containing a field representing a region of virtual addresses, a second field containing access privileges to that region by the protection domain or thread that is specified in a third field. Its inputs are the virtual address for the current memory access and the type of access request made by the CPU and the contents of two registers in which the currently executing general and carrier protection domains (collectively known as a UPD) are stored. It operates in a manner similar to modern TLBs. CPU requests for accesses to memory and current general and carrier protection domain base register contents are associatively compared against RPLB entry fields during the early stages of pipelined instruction execution to determine whether the requested access should be allowed. For cross-domain access, again somewhat simplified, the RPLB entry contains a single

address through which the cross domain access may be made, the type of access and the domain from which the transfer can be made. These are compared during instruction processing against the CPU emitted memory access request and the contents of the protection domain registers. A flag in the RPLB entry indicates that this is a switch request and another RPLB field indicates the destination GPD for a valid request. The operation of the RPLB is described in greater detail in section 4. The Sombrero RPLB directly determines, at the hardware level, the permissibility of requests by threads, for access to memory regions.

2.8 Extensibility

The Sombrero kernel and its protection model and RPLB matrix, protection and resource access list (PRAL) and hardware RPLB are designed to operate in a single address space that is distributed and replicated across multiple nodes. For example memory object control blocks and region control blocks, represented by surrogates on nodes other than the originating nodes, define access privileges to the same ranges of memory on every node that they are present on. Similarly, general and carrier protection domain control blocks are represented by surrogates on other nodes that head lists that define the objects accessible by the principals they represent no matter what node they are located on. These data structures are so designed that they are replicated only as needed by lazy evaluation stemming from page and protection faults on the given node. The token tracking mechanism and token track data structure support the required coherency. This is discussed in detail in [SkMi96c].

3. SOMBRERO PROTECTION AND RESOURCE ACCESS LIST (PRAL)

The Sombrero PRAL are designed to organize and store the data contained in the RPLB matrix entries in such a way that:

- They implement the translation from the Lampson style protection matrix to the RPLB matrix.
- They represent higher level policy to conduct both rights authorization and rights revocation.
- They provide the constraints for rudimentary hardware protection for object oriented design strategies as well as methods to manage the four fundamental Sombrero object abstractions.
- Their contents can be rapidly searched by the Sombrero kernel on a RPLB protection miss.
- They can be easily modified for addition and deletion of access privileges, memory objects and protection domains.
- They provide in a simple and efficient manner the additional functions required for extensibility to a multi-node network of computers.
- The Sombrero PRAL are kernel resident i.e. part of the privileged level of access.

3.1 Control Blocks

Figure 3 contains Sombrero's principle software protection data structures and indicates the contents of the main fields of each. The Memory Object Control Block (MOCB), Region Control Block (RCB) and Access Descriptor (AD) provide protection-related information about the resources contained in memory objects. The General Protection Domain Control Block (GCB), Carrier Protection Domain Control Block or Thread Control Block (TCB), Access Descriptor (AD) and Principal Control Block (PCB) provide information on what can be accessed from a given protection domain and the user or principal the protection domain is associated with. The Root MOCB serves as a list head for the MOCBs, GCBs, TCBs and PCBs.

The Token Tracking structure contains information about copy set locations and read and write access permissions relevant to distributed systems in order to manage the replication and consistency issues caused by network latencies. Much of the functionality of the Root MOCB is covered in White Paper 10 [] where partitioning and fissioning of the virtual address space is discussed in detail as are issues related to token tracking and the Token Tracking structure. An exception to the latter is that the impact of network considerations on the design of Sombrero's data structures is discussed at the end of this section.

3.1.1 Memory Object Control Block

The MOCB (see Figure 3a) defines and provides control information for Sombrero's basic storage abstraction, the Memory Object, introduced in section 2.1.1. Starting and ending virtual addresses are stored in the Range field. A memory object is named by its lowest virtual address. The ACL field is the head for a list of protection domain control blocks representing the protection domains which have access to this memory object. This list called the *Resource Access Control List (RACL)* is described in section 3.2. The Token Tracking field contains a pointer to the Token Tracking structure for this memory object. The token tracking structure, described below in Section 3.1.8, contains information needed to manage the copy set of this memory object in distributed systems. The PCB and Methods field contains the address of the PCB representing the principal that owns the memory object and addresses of exported functions used to manipulate the object. These functions, invoked by applications with a system call, are intended to implement object-oriented concepts and permit a custom calling signature for object method invocation. For example replication and consistency semantics can be placed under user control. The next three fields contain pointers to the up to three Region Control Blocks, described in the next section, that define memory object regions and associated access privileges. For non leaf MOCBs the Fission List is a list head for MOCBs controlling the subranges which collectively comprise the memory range controlled by this MOCB. The MO Link field links this MOCB with its sibling MOCBs on this node (see section 3.1.7). Use of the Fission List and MO Link Fields is discussed under memory allocation in White Paper 10. An MOCB is named by the VA of the lowest address in its range. The name field stores the address of the originating MOCB. In distributed systems, on nodes other than the originating node, called *foreign nodes,* the surrogate MOCB on that node will have a different address and when it is necessary on a foreign node to search for the MOCB by its 'global' name (the address of the MOCB on the originating node) the Name field has the required address.

3.1.2 Region Control Block

Memory regions are described in 2.1.1. The RCB Region field (see Figure 3b) contains the set of virtual addresses and protection attributes that define the memory region in a form that is useable by the hardware RPLB called the *range mask* (see Section 5.1). The Token field contains a pointer to the corresponding Token Track structure and a 'token' flag that is used to enable the access privilege(s) on the given node.

3.1.3 General Protection Domain Control Block

The GCB (see Figure 3c) defines and provides control information for part of Sombrero's UPD protection abstraction, the General Protection Domain, introduced in section 2.1.2. A GCB is named by the virtual address of its control block and the GCB Name field serves the same function as for the MOCB. The virtual address of the GCB is loaded into the RPLB GPD base

register on entry into a GPD to indicate access privileges for this domain are now in effect (see Sections 3.4 and 5.2.). The PCB and Methods field contains the address of the PCB representing the principal who is associated with the GPD, typically a user, the super-user or a Sombrero kernel demon or user-level server and also the addresses of exported functions used to manipulate the protection domain. The PAL field is the head for a list, called the *Protection Access List (PAL)*, that specifies the resources and domains the protection domain has access to and the associated permissions. The RACL field is the head for a list of thread and general protection domain control blocks representing the protection domains which can switch into this protection domain. RACLs are described in section 3.2. The Token Tracking field contains a pointer to the Token Tracking structure for this memory object. The token tracking structure (see Section 3.1.8) contains information needed to manage the copy set of this GCB in distributed systems. The Heap Space field contains a pointer to the base of this GCBs heap space. The GCB heap space is used to allocate static storage for data which will be accessible to all threads which operate in the GCB. The Link field links this GCB with the other GCBs on this node (see section 3.1.7).

3.1.4 Carrier or Thread Protection Domain Control Block

The TCB (see Figure 3d) defines and provides control information for the other part of Sombrero's UPD protection abstraction, the Carrier Protection Domain, introduced in section 2.1.2. The TPD Name, PCB and Methods, PAL, Token tracking and Link fields perform similar functions to the corresponding GCB fields. The virtual address of the TCB is loaded into the RPLB Thread base register on entry into a CPD and indicates access privileges for this thread's domain are now in effect (see Sections 3.4, 5.2 and 5.3). The GPD field contains the address of the GCB for the GPD the thread is currently executing in and the CPU field contains CPU state needed for context switches for this thread. When it simplifies explanations, we refer to GPDs and CPDs and the GCBs and TCBs collectively as the Ubiquitous Protection Domains (UPDs) and the Ubiquitous Protection Domain Control Blocks (UPDCBs).

3.1.5 Principal Control Block

The PCB (see Figure 3e) contains the name of the principal, a list of the attached resources in use in the various sessions in which the principal is currently a participant and a list of the Sombrero software protection domain data structures owned by the principal. As noted, a principal could be an ordinary user, the superuser or some other privileged pseudo-user like a kernel-level or user-level server.

3.1.6 Access Descriptor

The AD (see Figure 1f, Figure 4 and Figure 5) links UPDCBs and MOCBs, specifying UPD memory access and switch privileges. For the MOCB, the UPDs that can access it. The AD is the list element for the RACLs and PALs, collectively referred to as the Sombrero *Protection and Resource Access Lists (PRALs)*. It has fields for the link pointers and owners of these lists and for the relevant access privileges and domain switch addresses. It can also store domain specific access method overrides such as a domain specific RCB. The operation of and the connections to an AD are discussed further in section 3.2.

3.1.7 Root Memory Object Control Block

The Root MOCB (see Figure 3g) indicates the virtual address range allocated to a Sombrero logical node and serves as a list head for the GCBs, TCBs, MOCBs and PCBs in the node. It has a pointer to its parent, a link to any other Root MOCBs on its physical node and pointers to the heaps used for managing the CBs in its logical node. The function of the Root MOCB is discussed further in White Paper 10 where global memory allocation is discussed.

3.1.8 Token Tracking Structure

The Token Tracking structure (see Figure 3h) is the basic structure for managing the copy set of replicated control blocks in a distributed system. It provides the information necessary for readers to find the most recent copies of pages, for writers to broadcast invalidations for obsolete pages and for the maintenance of internode coherency and synchronization under replication. The Token Tracking structure is pointed to by the Token Tracking field of GCBs, TCBs and MOCBs. It has been made a separate structure since not all control blocks need it and because it is of variable size. Token tracking is discussed in White Paper 10. The key point to note here is that, the mechanisms to support distributed operation have been built into Sombrero's basic resource access and protection data structures.

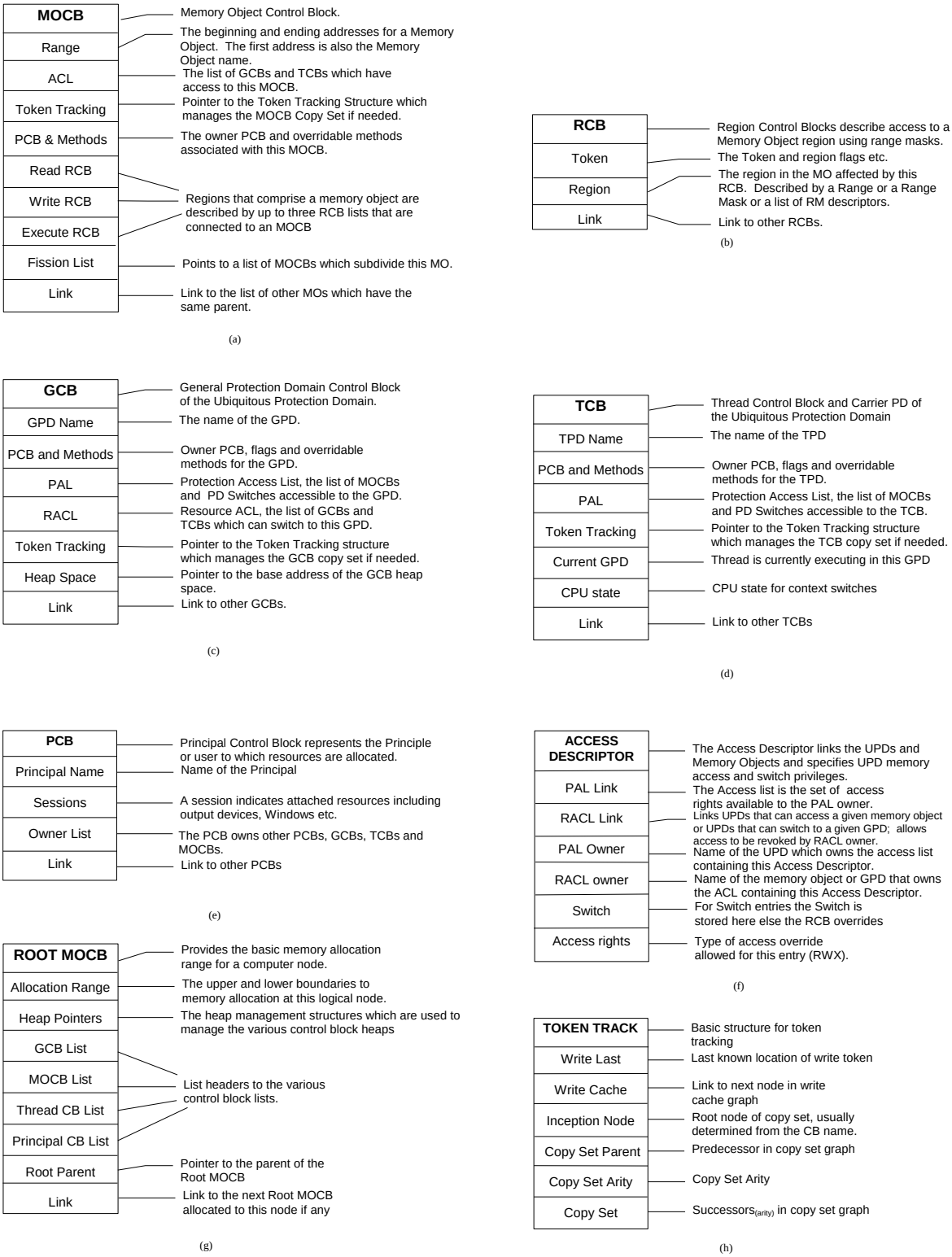
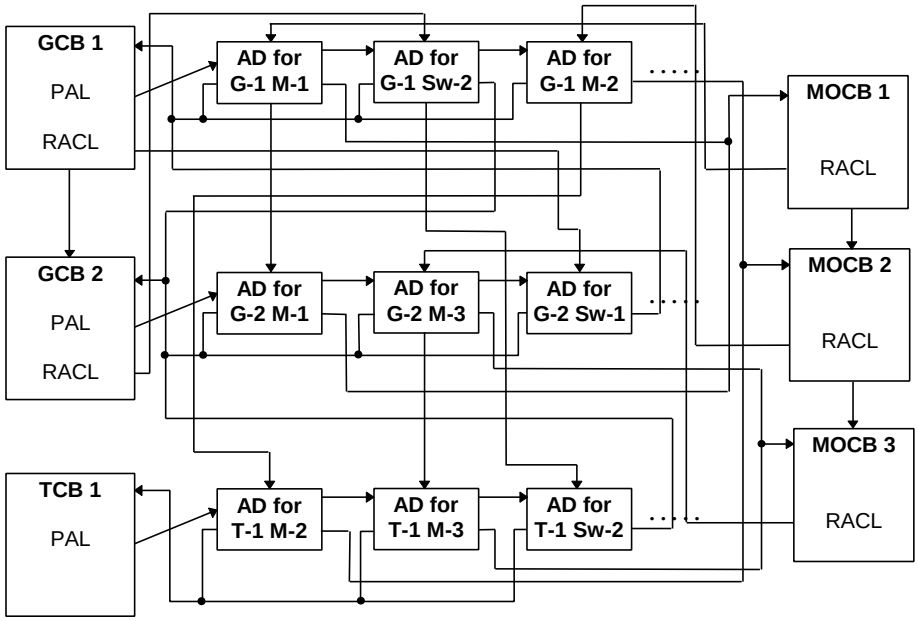


Figure 3. Sombrero Software Protection Data Structures. (a) Memory Object Control Block. (b) Region Control Block. (c) General Protection Domain Control Block. (d) Carrier or Thread Protection Domain Control Block. (e) Principal Control Block. (f) Access Descriptor. (g) Root Memory Object Control Block. (h) Token Tracking Structure.

3.2 Protection and Resource Access Lists (PRALs)

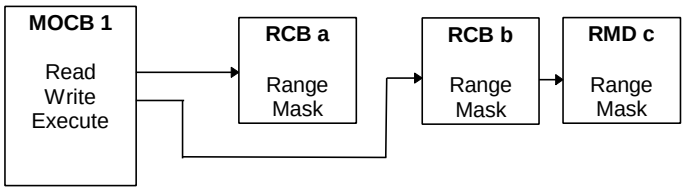
The MOCBs, GCBs and TCBs are linked together by AD structures into Protection and Resource Access Lists (PRALs) (see Figure 4a). Access to each Memory Object can be further refined by one or more RCBs in a list headed by its MOCB (see Figure 3a&b). A PRAL, which at first glance might appear somewhat formidable, is a web of Resource Access Control Lists (RACLs) and Protection Domain Access Lists (PALs) combined together in one place to speed up software protection checks and resource access rights revocation. The PRAL represents an integration of the Lampson and the RPLB matrix.



LEGEND:
AD for G-k M-k ≡ Access Descriptor containing access rights of GPD_i to MO_j and entry in MOj's ACL for GPD_i.
AD for G-i Sw-j ≡ Access Descriptor containing access rights of GPD_i to GPD_j and GPD_j entry address and GPD_j ACL entry for GPD_i.

The Protection and Resource Access List (PRAL). The connections to the Root MOCB are not shown. The ADs for switches and MOCBs are shown here mixed together. In practice they would be in separate lists. Switches also require a switch address in the AD that is not shown here.

(a)



MOCB 1 is a data memory object with read and write regions. There is no execute region. The read region is described by one range mask in RCB a. The write region is described by two Range Masks in RCB b and RMD c.

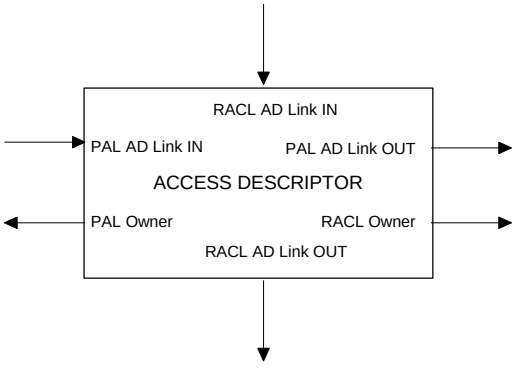
(b)

Figure 4. Sombrero Protection and Resource Access List.
(a) UPD Control Blocks and Memory Object Control Blocks interlinked by Access Descriptors.
(b) Detailed view of MOCB 1 showing Region Control Blocks associated with it.

A RACL specifies which UPDs that have been granted access to its attached resource (the resource is the memory object or protection domain that heads the RACL). The RACLs are the columns of the Lampson protection matrix. The RACLs are usually used by the resource that heads the list to revoke access. Revocation is generally used to force links to be re-resolved so that objects can be logically moved or deleted.

A PAL, headed by a UPDCB, specifies the resources and domains a protection domain has access to and the associated permissions. The PALs have the information contained in a row of the Lampson matrix and the principals associated with the protection domains that head these lists are the subjects of access control. Most systems that use PALs refer to the list as a capability list or C-List and the entries as capabilities. Sombrero does not use the PAL list entries in the same way as capabilities are used so we eschew that name. For more on this see section 6 and White Paper 16 [].

Figure 5 is a blow-up of a Sombrero Access Descriptor showing only the fields relevant to the PRAL in Figure 4a.

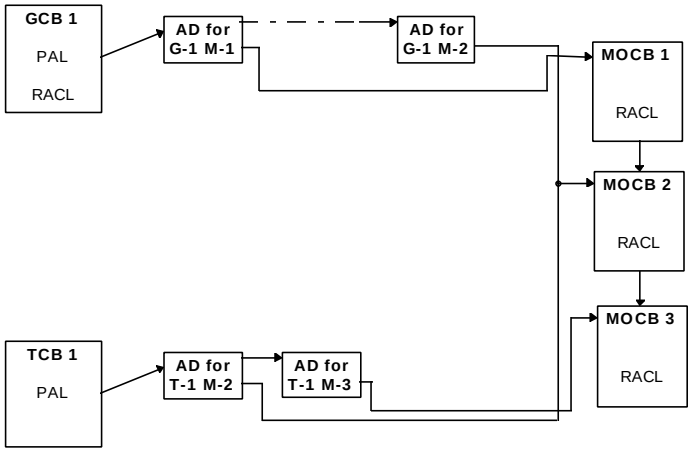


Access Descriptor defining entry and exit points and list owners.

Figure 5. Sombrero Access Descriptor.

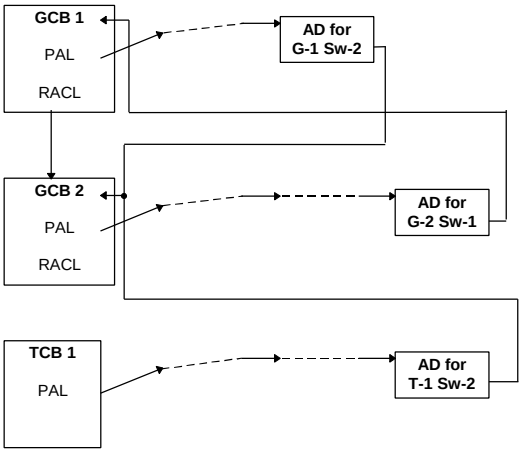
Figure 6 depicts two sets of PALs contained in the PRAL of Figure 4a. Figure 6a shows two PALs which contain the access permissions for GPD 1 to Memory Objects 1 and 2 and the access

rights of CPD 1 to Memory Objects 1 and 3. Figure 6b shows the PALs which contain access permissions for GPD 1 to switch to GPD 2, GPD 2 to switch to GPD 1 and threads in CPD 1 to enter GPD 2. Note that Figure 4 and Figure 6 assume that a single linked list contains both the MOCBs and the UPDCBs accessible to a UPD. It is not clear at this time whether there should be a separate PAL for accessible memory objects and for accessible domains for each UCB.



A view of the PRAL drawing extracting an example of a GCB PAL and a TCB PAL for the MOCBs. The Switch ADs are excluded for clarity.

(a)

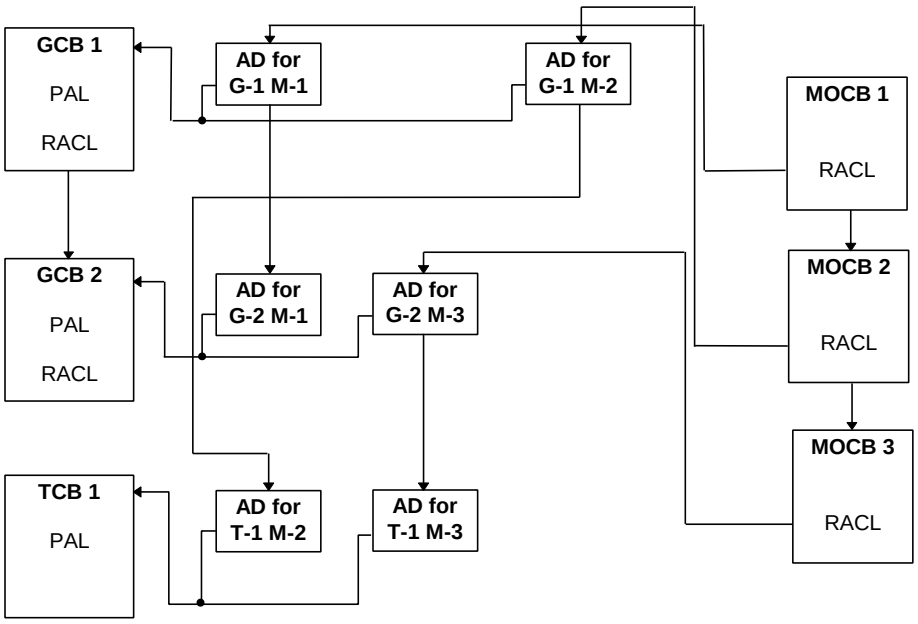


A view of the PRAL drawing extracting the GCB PALs and TCB PAL used in Switching.

(b)

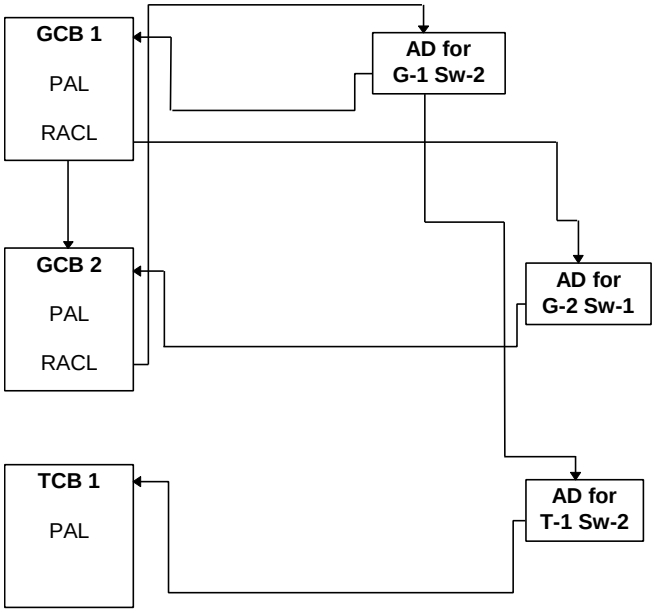
Figure 6. Protection Domain Access Lists (PALs). (a) Memory Object Access Lists headed by GCB 1 and TCB 1. (b) Protection Domain Access Lists for GPD 1, GPD 2 and CPD 1.

Figure 5 shows two sets of RACLs contained in the PRAL of Figure 2a. Figure 5a shows three RACLs which contain the ACL entries for Memory Objects 1, 2 and 3. Figure 5b shows the RACLs which contain switch entries allowing entry into GPD 1 from GPD 2 and entry into GPD 2 from GPD 1 and CPD 1. The latter would typically be for the return to GPD 2 of a 'traveling thread' in CPD 1 from a called domain if the caller, in GPD 2, trusted the callee (whose domain is not shown.)



A view of the PRAL drawing extracting The MOCB RACLs.

(a)



A view of the PRAL drawing extracting the GCB RACLs used in switching.

(b)

Figure 7. Resource Access Control Lists. (a) Memory Object ACLs headed by MOCBs 1, 2 and 3. (b) Protection Domain ACLs indicating which domains can enter GPD 1 and GPD 2.

The Root MOCB defines the range of virtual addresses present on a logical node. In addition it acts as list head for the UCBs, MOCBs and PCBs that protect, represent and own respectively, the resources in that virtual address range. Typically there is one logical node per physical node, but VA space fissioning, merging and subsequent additional VA requirements can cause the presence of multiple logical nodes on a physical node. See White Paper 10 [] for more on this. Figure 6 shows a Root MOCB and its associated control block lists. Note that in Figures 2a, 4 and 5 the GCBs and MOCBs are connected in linked lists. These are the GCB and MOCB linked lists headed by the Root MOCB shown in Figure 6.

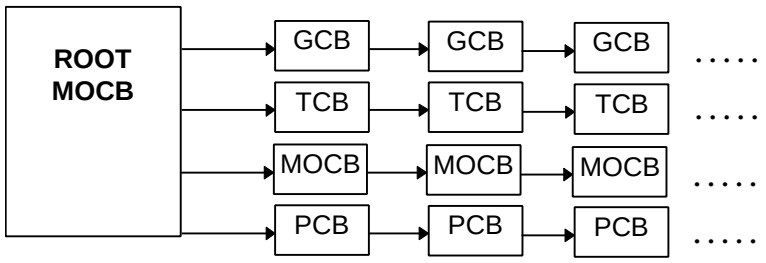


Figure 6. A Root MOCB and its associated linked lists of control blocks.

3.3 Additional Sombrero Resource Access and Protection Mechanisms

3.3.1 GARBAGE COLLECTION

Garbage collection in Sombrero is facilitated by per-MOCB ACL's (generally known as RACLs). Each MOCB is the list head for a RACL which is a linked list of the UPDs that have access to it. Therefore virtual address garbage collection can occur automatically without cleanup or policy management intervention when a Memory Object becomes unreferenced. When no UPD has access permissions to a memory object, the RACL headed by its MOCB becomes empty and the MOCB and its range is available for reallocation. Before it gives up access to an MOCB, each UPD should appropriately manage its pointers into the virtual address range. Dangling pointers can lead to a Sombrero protection exception or, if the range is reallocated to the same UPD, improper access. Given the advantages of reduced VA space fragmentation that accrue from reallocation, avoiding dangling pointers with careful programming is probably the preferable practice - but a no-reallocation policy is certainly possible.

3.3.2 UPD SWITCHING

The *S* term, or Switch Set of a UPD describes a set of switches which, when associated with execute permission, are entry points into other UPDs. The switch causes the domain to change beginning with the current instruction without altering the program flow, i.e., any normal memory access such as an instruction or data fetch, can trigger the switch just by accessing a switched address. Switches do not have the same non overlapping access privilege restrictions in the RPLB Matrix as MOCB regions as the switch region is trivial containing only one address. The only consistency requirement for a Switch is, that for the current UPD, the Switch can not fall within a region defined by an RCB for an MOCB that is accessible to the current UPD. The

PAL for the UPD *M* term, the set of accessible Memory Objects, is searched before the PAL for the *S* term, the set of accessible UPDs, is searched. If the *M* term PAL produces a hit the, *S* term PAL will never be searched causing the switch to be ignored[8]. The switch entry itself does not permit a read/write/execute to occur at the specified VA and does not constitute an *ACCEPT* entry in the RPLB Matrix, rather it causes the Ubiquitous Protection Domain to be switched to the new UPD before the *ACCEPT* or *REJECT* is determined. The new UPD must have the corresponding R, W, or X access to the location of the switch for the memory access to be allowed.

3.3.3 LOADING THE RPLB

Extracting the RPLB entry stored in Sombrero's software protection data structures that implement the RPLB matrix is initiated on a hardware RPLB miss. If an access to memory is attempted and no entry can be found in the RPLB which *ACCEPT*s the access or initiates a *SWITCH*, the CPU searches the current UPD's PAL for an MOCB having a virtual address range including the attempted access. If such an MOCB is found, a search is made in the MOCB's RCB list corresponding to the current access type for a legal access. This is done by determining whether there is a non-empty RCB list for the current access type. If an Access match is found a 'hit' is sought in the region by examining the RCBs in the MOCB list that includes this type of access. This is done by examining the Range Masks in the RCBs for a region until a 'hit' is made after which that Range Mask is loaded into the hardware RPLB (usually there will only be one RCB with one Range Mask per MOCB access type). If no MOCB is found, a search of the Switch PAL is made for an exactly matching Virtual Address. If a switch is found it is loaded into the hardware RPLB and the name of the current UPD is replaced by the new UPD name provided by the Switch and the access is reattempted. Permission for the type of access, e.g., execution (X) for a cross domain transfer of control or data (R or W) for a cross-domain load or store is determined by the hardware RPLB and software protection data structures for the new domain in the same way as for the initial intra-domain search. A complete miss results in a protection fault which must be resolved by higher layer policy-level software.

4. SOMBRERO HARDWARE RPLBS AND THEIR OPERATION

The Sombrero Kernel software protection data structures provide data in a form that is directly useable by a CPU augmented with some additional memory protection hardware. In our view the required hardware is a reasonable extension to contemporary CPU protection hardware and its implementation, which we refer to collectively as the hardware RPLB, is well within the capabilities of modern VLSI implementation.[9]

The RPLB is intended to operate in a manner analogous to a TLB. It stores the most recent hits in an associative cache and on a miss it searches a list for a valid entry.

---

[8] The discussion premises two PALs associated with each UPD, one for intra-domain memory object accesses and one for cross-domain switches. It is not clear at this time whether one list as shown in Figure 2a or two lists are better. Clearly a single list could be searched twice on a switch.

[9] The required hardware has been synthesized and simulated in ASU's ULSI laboratory using Mentor Graphics AUTOLOGIC II and SYS1076 VLSI Design Software and shown to be feasible [Ga*95, Sk*96a]. The ASU ULSI laboratory is directed by B. M. Huey.

To accomplish its access control function the RPLB requires two protection domain registers to point at the currently valid GPD Control Block and Thread Control Block. These control blocks will in turn provide access to the list of objects and switches which determine valid access for the current state of the machine. The contents of the base registers also serve as name registers naming the currently active GPD and CPD for purposes of determining a hit in the RPLB. They are, in fact, the virtual addresses of the corresponding GCB and TCB respectively.

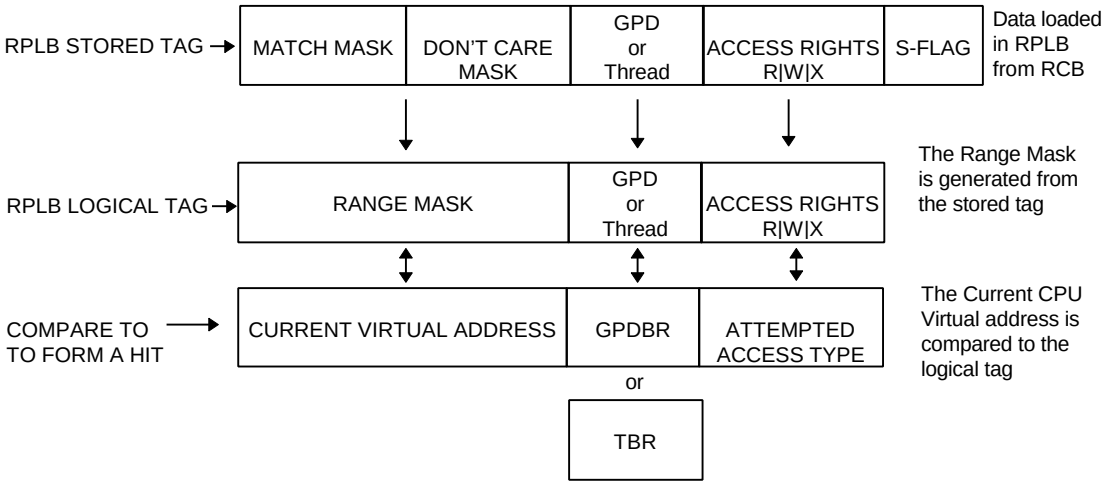


Figure 8 - RPLB mode 1 operation a.k.a. IRPLB or DRPLB.

4.1 RPLB Mode 1

An entry in the RPLB for mode 1 is shown in Figure 8. The S-Flag is used to invoke mode 2 in the RPLB and is discussed later. Assuming for now that this flag is off and we are in mode 1, notice that there is an RPLB Stored Tag and an RPLB Logical Tag. The stored tag is the match mask and don't care mask data loaded from the current GPD or CPD MOCB list RCBs into the RPLB (sec 5.3). The logical tag is the presentation or view of the stored tag bits in a manner suitable for validating the virtual address of any attempted memory access by the CPU and is called the Range Mask. The transition from the stored to the logical tag bits is called the Range Function and is performed by the RPLB hardware when the stored tag is loaded.

The Range Function has as its arguments the Match Mask, the Don't care Mask and as its product the Range Mask which is a set of tri-state bits. The Range Function is that all bit positions in the Range Mask are enabled and have the value of the corresponding bits in the Match Mask except when that bit position is disabled by a corresponding 1 value in the Don't Care Mask. See Figure 9. A hit in the RPLB occurs when the corresponding bits of the virtual address of the attempted memory access from the CPU match exactly the enabled bits in the Range Mask, and the attempted access is also allowed by the ACCESS RIGHTS bits, and either the GPD Base Register(GPDBR) or the Thread Base Register(TBR) match the 'GPD or Thread' field of the tag entry.

RPLB Hit example (we will use a 16 bit address space). See Figure 9.

Assuming the access permission and GPDBR are also a match, this particular example will *ACCEPT* a hit in the RPLB on two blocks with addresses in the range 0C80h to 0CFFh and 0D80h to 0DFFh. A few of the implications of this access strategy are:

- The size of the memory object is determined by the number of Don't Care bits. Object sizes vary according to need allowing the RPLB entry to cover any size object of 2^n bytes where n ranges from 0 to the number of address bits minus 1. By associating a set of Range Masks with a Memory Object any Memory Object of any size and position may be described, i.e. Object granularity to any size is completely under the control of the application and yet fully supported by the operating system and hardware.
- The region accepted by the Range Mask can be disjoint. The advantages this disjointedness may offer in database vertical or horizontal object fragmentation, view extraction and locking strategies are worth serious investigation. Notice that if we vary the Don't Care bits in the range mask from 0 to 2^{n-1} (n = don't care bit count) the Range Mask steps through the disjoint regions in memory as though they were a solid block. If an auto increment or offset mapping

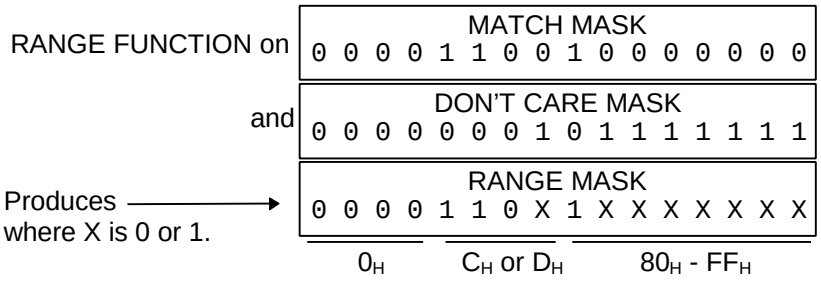


Figure 9 - Range function example

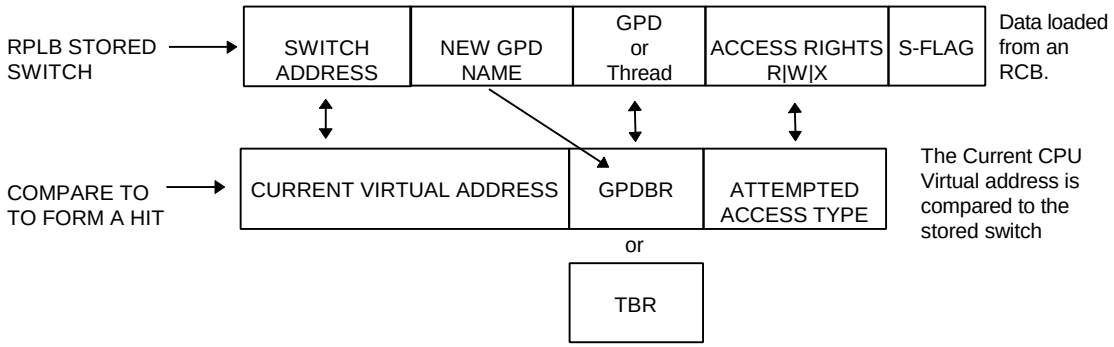


Figure 10. - RPLB in mode 2, a.k.a. SRPLB

instruction is provided to take advantage of this property it can be used to mask in selected data from fixed size memory objects. Access rights can be combined in one Range Mask or distributed over multiple Range Masks giving for example read rights to a block of memory and write access to only a subset of the memory block.

- The cost of implementing the Range Function in hardware is minimal, i.e. no compare or range check is required, only a bitwise propagation by combinational logic which is relatively easy to implement. Accessing unaligned data creates complications[10].

[10] The unaligned access problem is similar to that which occurs in paged memory when reading a data item which begins in one page and ends in another. For memory objects the RPLB must insure that an access which starts at a legal address is not reaching

- Only entries matching the current GPD *or* Thread names in the RPLB (see Figure 8) are accessible giving the ability to cross Protection Domain boundaries without flushing any caches.
- Large even disjoint objects can be expressed by a single entry with the potential of dramatically reducing the RPLB miss rate.

4.2 RPLB Mode 2

Switches from one domain to another are accomplished using the S-flag. When the S-flag is set, the RPLB stored entry is considered a special case and the Range function is not applied. Instead the RPLB is in mode 2 and the stored entry is a switch which was loaded from either the current GCB access list or the TCB access list of switch permissions. See Figure 10. For mode 2 the Match Mask field is a switch address which must match the attempted memory access virtual address exactly with no disabled bits. When a hit occurs on a stored entry with the S-flag set the GPD Base Register(GPDR) is loaded with a new GPD name using the value stored in the Don't Care Mask (A GPD name is the address of the GCB in the kernel memory).

A difficulty of GPD switching occurs in the sharing of data between GPDs. Frequently a switch will occur because a subroutine in another GPD is called to provide some service. Arguments passed to the subroutine or values returned cannot always be stored in registers. They may be pushed on the thread stack or placed in some other memory object. In order to access these shared memory spaces every GPD in which they will be accessed needs to have corresponding RPLB matrix entries. The number of RPLB entries needed to support access to one thread stack which may access a number of GPDs and for a single GPD accessed by many threads can be very large and is exponentially related to the number of threads and GPDs.

The RPLB provides the option of allowing threads which change domain context via a switch to form RPLB hits by matching not only the GPDBR but the Thread Base Register(TBR) as well. This union operation permits memory objects to be associated with a thread name (a thread name is the address of the thread's TCB). The effect is to give each thread its own protection domain which travels with the thread. The accessible memory for any given thread is then the union of the GPD memory objects and the thread memory objects i.e. the General and Carrier PDs. This means thread-particular memory objects such as a stack may be carried to any level of GPD nesting without the exponential profusion of RPLB entries for every GPD that would otherwise be needed, instead the problem becomes linear. The thread objects are accessible only by the controlling thread (unless they are deliberately shared) which permits threads to have private data. In essence threads have an independent protection domain.

In Sombrero memory objects in a CPD are R/W accessible but not executable. This prevents a thread which originates in the outer user level from carrying a Trojan horse into a distrusting server. By this means only data may travel with a thread and complete control of execution is retained by the GPDs. This is justified by the idea that we are binding additional state to a thread to enhance its register storage capacity and since thread storage is not executable neither is the expanded capacity of the thread state. Clearly the commonly used concept of a stack associated

---

beyond the end of the Range Mask. CPUs which permit unaligned access to memory generally break the access up into as many separate accesses as it takes to load the entire data item using smaller data types. This is also the solution to the Range Mask problem if each of the accesses are separately validated by the RPLB.

with a thread is regarded as an expansion of thread state and the placement of executable modules on the stack is generally discouraged. Sombrero uses its protection mechanism to support this concept, thus the IRPLB (instruction stream RPLB) will not need a TBR in an actual implementation.

5. SOMBRERO PROTECTION MODEL

In this section we delve more completely into Sombrero's protection model and look at it from different aspects. First in section 5.1, starting with the most flexible possible definitions of region, we refine the model based on practicality constraints to obtain the model described in section 2.2.1. Next in section 5.2 we relate the RPLB and Lampson matrices. Finally in section 5.3 we demonstrate how the PRALs and hardware RPLB implement the RPLB matrix.

5.1 Protection Model Derivation

Given:

RCB(*R*, *P*) or just RCB (sec 2.2.1.3)
MO(*low*, *hi*) or *range* (sec 2.2.1.1)

and the RPLB function (sec 2.5):

***Access_allowed* = RPLB(UPD, VA, Access)**

In this section we work back from the most general form for the RPLB matrix to the constraints imposed on it by the Lampson matrix.

We start off with a 'pure' 3-dimensional RPLB matrix. Ordinal one is the rows, one for each UPD. Ordinal two, the columns, 18.5 quintillion of them to represent each byte in the 64 bit address space. Ordinal three, the permission axis with three columns, one each for R, W and X. The intersection of the ordinals stores a value which is *ACCEPT*, *REJECT* or *SWITCH* for each element in the matrix (see Figure 2).

As a first step in the translation, instead of *ACCEPT* in each RPLB element, let us store the name of the RCB that is the data structure that is to be used to control access to some memory region. The presence of the RCB name in an RPLB element acts as the value *ACCEPT*. For this general protection matrix, the *R* (region) term in the RCB definition becomes: *R* is the set of all virtual addresses in the RPLB Matrix in which the name of the controlling RCB appears in a VA column.

Given this definition of *R*, the following observations about the region *R* can now be made. First, it can be quite arbitrary, for a given UPD the region *R* can be any number of isolated bytes each byte having any combination of R/W/X in *P*. Second, the address range of this region (minimum and maximum VAs in the region) can overlap the address ranges of other regions. Third, every UPD can map a whole different set of regions.

While there may be a use for such a flexible set of constraints on *R*, the above observations produce several problems. The First problem is, if regions overlap it is very costly to find all of the RCBs which contain an address. Second, if an address in the virtual address space participates in more than one region in a single UPD, control of the access to that address is managed by more than one RCB. Multiple RCB control of the same address will increase the

difficulty of maintaining a consistent regulation of access to the value at that address. Third, allowing regions to be remapped for each UPD compounds the consistency and management problems as mentioned in the second problem for a single UPD. Fourth, multiple combinations of protection attributes are managed by the same RCB which increases the difficulty of representing a region compactly. All four of these points compound the cost to manage the access to and consistency of any given VA storage location.

In our view, efficient access and consistency management is more important than an unrestricted distribution of regions and protections in every memory object. Therefore we refine the definition of the *R* term. As before, Given an RCB, *R* is the set of all virtual addresses in the RPLB Matrix in which the name of the controlling RCB appears in a VA's column with these additional constraints: First, an RCB is always assigned to a single Memory Object and every Memory Object has a fixed range which does not overlap that of another Memory Object. Second, a Memory Object region *R* does not extend beyond the *range* of the Memory Object to which it is assigned. Third, an RCB may apply only one value of *P* to its region *R*. Fourth, for RCB regions sharing the same *range*, *P* must be exclusive, i.e. the logical AND of R/W/X from *P* in any 2 RCBs in a *range* must be *false*. Fifth, all UPDs must respect the same set of *range*s, i.e. if a *range* appears in one UPD a *range* appearing in another UPD may not overlap it.

Some observations about the new RCB can be made. First, a *range* must be established for each MO before its regions, defined by RCBs are generated. Second, for the same *range,* up to three distinct RCBs can exist having non overlapping *P*. Third, within the *range* and *P* restrictions the region *R* does not need to be contiguous or to compass the entire *range*. We premise that **the cost of controlling access to and enforcing the consistency of a region *R* is acceptable under these constraints since all that must be verified is that the address range for an RCB does not extend beyond its containing Memory Object's *range* and that *P* does not overlap between the RCBs within the same *range*.**

It can be seen that the Lampson matrix is capable of controlling the Memory Objects directly and manipulating RWX access. Thus the Lampson matrix, which enforces policy for resources, establishes the bounds for a region *R* and its access permission *P*. However, *R* and *P* within these bounds can be manipulated, independent of the Lampson matrix, to produce differing views or schema of an object due to the flexible nature of the RPLB matrix. This independence is used to manage consistency semantics on the network [SkMi96c] and along with some heuristics is used to manage access concurrency and object masking.

We cannot store the contents of an RPLB Matrix directly, it's just too large. Software protection data structures must organize and store the currently required subset of this data in a manner similar to C-lists or ACLs. Since we must also store the equivalent of a Lampson matrix in order to represent the higher level protection *policy* decisions, i.e., allowed access to resources, it makes sense to merge the information in these two matrix lists and place it in one set of data structures. In Sombrero this set of data structures is the PRAL (Protection and Resource Access List) described earlier.

The information stored in these data structures and lists implements the protection policy of the Lampson matrix in terms of the parameters emitted by the CPU (i.e. as an RPLB matrix).

5.2 Relationship of RPLB Matrix to the Lampson Matrix

Given the above derivation of the consistency restrictions on RCBs in memory objects which bound the allowable entries for the RPLB matrix, we can now complete the relationship between the Lampson and RPLB access matrices by discussing the derivation of switches. The MOCB is the bridge which links the RPLB and Lampson matrices together. Any two RCBs which have the same range represent different protection attributes of the same Memory Object. MOCBs control access to VA region/protection pairs within the region that, at the lowest level, specify allowable access to every computer system resource.

The RPLB Matrix has three axes and the Lampson Matrix two. The Access ordinal is added in the RPLB Matrix allowing up to three RCBs to be named in the RPLB Matrix for each <PD, resource> pair in the Lampson Matrix or a Ubiquitous Protection Domain *switch* for each Lampson Matrix <PD_i, PD_j> pair. The entries in the Lampson matrix translate into the entries in the RPLB matrix as follows:

***MOCB RCB Set* = Lampson(PD, Resource)**

where the *MOCB RCB Set* consists of one to three of the **RCB(*R, P*)** data structures that manage a memory object and

***enter* = Lampson(PD_i, PD_j)**

where *enter* corresponds to *enter* in a Lampson matrix and is stored as Switch(*V*,G,*P*) in the RPLB Matrix with *P* assumed to be execute (X) access[11], i.e. *enter* represents a protection domain entry point, and *V* is the VA of the entry point. Each GPD may have several entry points each of which require a column in the Lampson matrix. PD_i is the current UPD and PD_j is the new GPD.

Switches have fewer constraints then RCBs. They do not belong to a Memory Object, the range and region are trivial having only one *VA* and *P* may be any combination of R/W/X.

5.3 Implementation of RPLB Matrix by Sombrero Hardware and Software Protection Data Structures

Having demonstrated that the RPLB matrix and Sombrero data structures represent and store the protection policy information contained in a Lampson matrix about protection domains and objects, we next need to show that we can successfully determine *ACCEPT*, *REJECT* or *SWITCH* using PRAL data structures and then we need to show the mapping between these and the RPLB hardware. That is, we need to demonstrate that the RPLB hardware does in fact implement the protection specified in the RPLB matrix and the corresponding Sombrero protection data structures.

Given a UPDBR, a CPU emitted VA and Access, the CPU makes a request to the RPLB function. If the Range mask is already cached in the RPLB for the given domain and access type, then we get, in parallel with instruction execution:

***ACCEPT* <= RPLB(UPD, VA, Access)**

[11] By appending R/W to *enter* non-control transfer switches can be defined.

where '<= ' º 'is mapped to'.

Otherwise if a Switch is already cached in the RPLB for the given VA, domain and access type, then we get, in parallel with instruction execution:

***SWITCH* <= RPLB(UPD, VA, Access)**

On a miss the emitted CPU request leads to an access to the Sombrero software protection data structures as follows. First by applying the content of the UPDBR to choose a PAL:

PAL <= RPLB(UPD)

Next, VA is applied to the range to select a Switch or an MOCB, the RPLB examines the MOCB case first:

MOCB <= PAL(VA)

The MOCB, in turn, selects an RCB using Access:

RCB <= MOCB(Access)

The Range Mask is extracted from the RCB:

Range Mask <= RCB(VA)

Lastly the Range Mask is loaded into the RPLB so the validity of the original access can be determined:

***ACCEPT* <= RPLB(UPD, VA, Access)**

If an MOCB was not selected on the first PAL search then a Switch is selected using VA and Access:

Switch <= PAL(VA, ACCESS)

The selected Switch is then loaded into the RPLB and the validity of the original access is checked for a cross domain transfer:

***SWITCH* <= RPLB(UPD, VA, Access)**

At this point a failure to find a Switch is equivalent to *REJECT*. After a *SWITCH* is completed the process of getting an *ACCEPT* is restarted in the new domain before the instruction stream is allowed to continue.

The key point is that the low level protection mechanisms provided by the hardware RPLB implement the protection specified by the information contained in the software protection data structures which as we have shown above are fully capable of enforcing the policy level protection specified by the Lampson Matrix.

6. SUMMARY OF SOMBRERO PROTECTION STRUCTURES

Sombrero's protection structures take advantage of the properties of a very large single address space in a number of ways. In a single address space all resources can be uniquely named by virtual addresses and all protection domains can be defined by listing the access rights of

principals to these resources. It is therefore possible to build protection data structures with the following two attributes:

1.They can be related without ambiguity or translation to the resources and protection domains that principals wish to access.

2.They can be supported by relatively simple hardware extensions to the hardware which already exist in modern 64-bit RISC CPU's.

The principle software protection data structures are MOCBs and UCBs. For each protection domain a UPD Control Block acts as a list head for all of the Memory Object Control Blocks that represent the virtual address regions and access privileges of all of the objects accessible to the protection domain. Furthermore, the virtual address of the UCBs and the range mask and access privileges in the MOCBs and RCBs provide all of the information needed to be stored in hardware to directly determine the validity of every memory access.

We have presented the information contained in these protection data structures as a 3-D matrix, the RPLB matrix. The rows of this matrix are protection domains and the two column ordinals are the set of virtual addresses and the protection attributes, read/write/execute. The entries are values representing MOCB's and Ubiquitous Switches. This matrix presents a directly useable representation of the resource access information that must be stored in hardware protection data structures to allow principals to correctly access objects and cross domains and to disallow unwanted accesses to memory objects.

We have also presented the design and operation of the hardware device that stores the required protection information to be used on every memory access. We call this device a Range Protection Lookaside Buffer or RPLB. It consists of several subdevices, an Instruction RPLB, a Data RPLB and a Switch RPLB

On a hardware protection miss, information from the data structures is provided to the hardware protection device, the RPLB, in a directly usable form, the match mask, don't care mask, protection domain name, access rights and two or three flag bits. This information is stored in the MOCBs RCBs in such a way that it can be loaded into the RPLB without any intermediate transformations. On a miss in the software protection data structures, higher level policy-making software is invoked to determine whether to authorize privileges not currently present in the PRAL. The important point here is that the information that needs to be supplied by the higher layers to Sombrero is directly and simply related to the memory objects these programs would normally be manipulating. In other words, Sombrero projects objects and protection domains into user space.

We have compared our resource access and protection model, the RPLB matrix, with the one developed by Lampson a quarter century ago and have shown why it provides a more directly useable representation of information for protection in a very large single address system. The principle reason for this is that in such a system all resources and protection domains can be named by their virtual addresses. This permits labeling of columns and rows with directly useable quantities, virtual addresses. Operationally, the most important consequents of this are

parallel rather than serial cross domain communication and transfer of control, i.e., access permissions are determined within an instruction's execution, and direct access to long term storage. For example, client/server communication, RPC's which involve kernel intervention, context switches and message passing are replaced by a simple procedure call and return, and file access by loads and stores. It has been known for some time that these kinds of things are possible in a very large single address system. What we have presented that is new is a correct model and corresponding set of kernel protected data structures and a corresponding hardware device to efficiently implement these features in a way that is directly useable by object-oriented higher level programs. These are the RPLB matrix, the Sombrero software protection data structures known as the Protection and Resource Access List (PRAL) and the hardware RPLB.

7. REFERENCES

[Ch*94] Chase, J. S., Levy, H. M, Feeley, M. J., and Lazowska, E. D., "Sharing and Protection in a Single-Address-Space Operating System", *ACM Transactions on Computer Systems*, Vol. 12, No. 4, November 1994, pp. 271-307.

[Ga*95] Ganesan, R., Sukavanam, V., and Torla, M., "Range Protection Lookaside Buffer, VHDL Synthesis Report", Term Report CSE 518 VLSI Synthesis using HDLs, December 1995.

[He*94] Heiser, G., Elphinstone, K., Russell, S., and Vochteloo, J., "Mungi: A Distributed Single Address-Space Operating System", *The 17th Annual Computer Science Conference*, Australian Computer Science Communications, January 1994.

[Ko*92] Koldinger, E. J., Chase, J. S., and Eggers, S. J., "Architectural Support for Single Address Space Operating Systems", *Proceedings of 5th International Conference on Architectural Support for Programming Languages and Operating Systems, ACM SIGOPS Operating Systems Review*, October 1992.

[Lam71] Lampson, B. W., "Protection", in *Proc. Fifth Princeton Symposium on Information Sciences and Systems*, Princeton University, March 1971, pp. 437-443, reprinted in *Operating Systems Review*, 8, 1, January 1974, pp. 18-24.

[Mu*93] Murray, K., Wilkinson, T., and Osmon, P., "Design and Implementation of an Object-Orientated 64-bit Single Address Space Microkernel", *USENIX Symposium on Microkernels and Other Kernel Architectures*, 1993, pp. 31-43.

[RoAb85] Rosenberg, J. and Abramson, D., "MONADS-PC - A Capability-Based Workstation to Support Software Engineering", Proceedings of the Eighteenth Annual Hawaii International conference on System Sciences, 1985, pp. 222-231.

[Sc*90] Scott, M. L., LeBlanc, T. J, Marsh, B. D., Becker, T. G., Dubnicki, C., Markatos, E. P., and Smithline, N. G., "Implementation Issues for the Psyche Multiprocessor Operating System", *Computing Systems*, Vol. 3, No. 1, Winter 1990, pp. 101-137.

[Sk*96] Skousen, A. C. and Miller, D. S., Feigen, R. G. "The Sombrero Operating System An Operating System For A Distributed Single Very Large Address Space General Introduction", *ASU 64-bit OS Group WP 0*, Computer Science and Engineering Department, Arizona State University, TR-96-005, April 1996.

[Sk*96a] Skousen, A. C., Huey, B. M.., and Miller, D. S., "RPLB Requirements Specification and VHDL Implementation", *ASU 64-bit OS Group WP 12* , Computer Science and Engineering Department, Arizona State University, July 1996 (to appear).

[SkMi96b] Skousen, A. C. and Miller, D. S., "Resource Access and Protection in Sombrero: Protection Model, Software Protection Data Structures and Hardware Range Protection Lookaside Buffer", *ASU 64-bit OS Group WP 2* , Computer Science and Engineering Department, Arizona State University, TR-95-013, May 1996.

[SkMi96c] Skousen, A. C. and Miller, D. S., "Implementing a Single Very Large Address Space across Multiple Nodes: Memory Partitioning, Protection Domain Migration, Kernel Replication, Consistency and Fault Tolerance", *ASU 64-bit OS Group WP 10*, Computer Science and Engineering Department, Arizona State University, TR-95-021, May 1996.

[SkMi96d] Skousen, A. C. and Miller, D. S., "Sombrero Nameserver/Linker/Compiler/Loader Design and Modifications", *ASU 64-bit OS Group WP 7*, Computer Science and Engineering Department, Arizona State University, TR-96-018, July 1995.

APPENDIX 1 ASSORTED INFORMATION THAT NEVER GOT INTO PAPER BODY

Domain switching is also efficient adding as little as no overhead to the switch. RPLB hardware enables cross-domain transfer of control and data accesses to occur in parallel with other CPU operations, i.e., without additional CPU instructions.

In Sombrero terms, the UPD *M* term corresponds to the memory object C-list and the *S* term to the Ubiquitous Switch C-list for a UPD. The access control lists for the memory objects and UPD switches exist and are important to other kernel services but do not participate directly in the RPLB's operations.

In Sombrero the Read, Write and Execute access rights are described by the Memory Object Control Block (MOCB) which controls access to a range of memory which may encompass multiple pages or only part of a page. These MOCBs are the sources for the entries to the RPLB. The RPLB takes over the TLB protection function and expands that functionality to generate the UPDs.

The Memory Object definition can now be presented: Memory Objects in Sombrero are distinct, contiguous ranges of the virtual address space accessed by one to three MOCBs that contain access rights to memory regions within the ranges. More formally a Memory Object is an ordered pair (*Range*, MOCB Set) where *Range* is an ordered pair (VA_{min}, VA_{max}) and MOCB

Set is a set of one to three MOCBs having non-overlapping *P*. The MOCB regions are bounded by the Memory Object *Range*.