SOMBRERO: A VERY LARGE SINGLE ADDRESS SPACE DISTRIBUTED OPERATING SYSTEM

by

Alan Clifton Skousen

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

ARIZONA STATE UNIVERSITY

December 1994

SOMBRERO: A VERY LARGE SINGLE ADDRESS SPACE DISTRIBUTED OPERATING SYSTEM

by

Alan Clifton Skousen

has been approved

December 1994

APPROVED:

_____,Chairperson

_____

_____

Supervisory Committee

ACCEPTED:

_____
Department Chairperson

_____
Dean, Graduate College

ABSTRACT

With the advent of processors providing access to very large address spaces, 64 bits and up, a new window of opportunity is opened to reapproach operating system design. By conducting all of the activities of a computer network in the same single very large address space, matters pertaining to protection, naming, persistence, network consistency, shared memory, memory allocation and management, the use of threads, and the extensibility of these matters to the network can all be optimized in new ways. Sombrero is a design study which takes a broad look at some strategies to take advantage of this new paradigm. Particular emphasis is made in this thesis concerning protection domains and protection domain management using tokens and the usefulness of these for object distribution.

For my family

Noreen, Noah, Emily, Seth and Adam

ACKNOWLEDGEMENTS

Many thanks for the hours of effort, particularly for my committee chair, Dr. Donald Miller, and also my committee members, Dr. Bruce Millard and Dr. Susan Urban.

TABLE OF CONTENTS

Page

LIST OF FIGURES viii

INTRODUCTION 1

BACKGROUND 2

1. UBIQUITOUS PROTECTION DOMAIN 4

1.1 AMOS - Alpha Micro Operating System 5
1.2 General UPD Concept 6
1.3 UPD-I or Direct PDM 10
1.3.1 UPD-I Switching; S-lists 12
1.3.2 UPD-I Example 13
1.3.3 Pros and Cons 15
1.3.4 UPD-I Hardware 17
1.3.5 UPD-I Summary 18
1.4 UPD-II or TLB Oriented UPD 19
1.5 UPD-III or Ideal UPD 20
1.5.1 Independent PD Concept 21
1.5.2 Range Protection Lookaside Buffer (RPLB) 22
1.5.2.1 RPLB Switches and Sharing 27
1.5.2.2 RPLB Comparison 29
1.6 UPD Tokens 30
1.7 Comparison to Other Work 31
1.8 Contributions and Summary 34

2. MEMORY OBJECT TOKENS 36

2.1 Complex MOTs 38
2.2 Garbage Collection 39
2.3 Conclusions 40

3. THREADS 43

3.1 Inter UPD Thread Management 44
3.2 Thread Networks 48
3.3 Thread Conclusions 48

4. EXTENDING THE CONCEPTS TO A VLSASDOS 50

4.1 Token Tracking 50
4.2 Kernel Object Extensibility 54
4.3 Very Large Memory Allocation 55
4.4 Virtual Memory Consistency and Semantics 57
4.5 Network types 58

Page

5. ADDITIONAL TOPICS IN SUPPORT OF SOMBRERO 59

5.1 Token Execution 59
5.2 Name Server Design 59
5.3 Initial User Logon 60
5.4 Virtual Memory 60
5.5 Multiprocessors in Sombrero 62
5.6 Device Drivers and I/O Ports 62
5.7 An Analysis of UPD Switching and Lexical Support 63
5.8 Comparison to Conventional MASOS 69

6. VALUE OF RESEARCH 71

REFERENCES 73

APPENDIX

A GENERATING RANGE MASKS USING A FROM-TO RANGE 74

B KERNEL CONTROL STRUCTURES 76

C SOMBRERO BLOCK STRUCTURE 80

D PA-RISC and RPLB 82

INDEX 85

LIST OF FIGURES

Page

Figure 1-1 Archetype of a PD matrix 8

Figure 1-2 UPD-1 s-list entry 12

Figure 1-3 Representing the hardware used in the UPD-1 example 13

Figure 1-4 Bus address selecting PDM bit pair 14

Figure 1-5 Protection Domain Matrix Store (PDMS) 14

Figure 1-6 The RPLB operating in mode 1 23

Figure 1-7 Range function example 24

Figure 1-8 RPLB in modes 2 and 3 27

Figure 3-1 UPDs and connecting edges representing boundary crossings 45

Figure 4-1 Write cache graph 51

Figure 4-2 Write cache graph pruning 51

Figure 4-3 Write last graph pruning 52

Figure 5-1 Comparing Lampson's switch to the UPD switch 64

Figure 5-2 Simple Sombrero return from call 66

Figure A-1 Range Mask program 75

Figure B-1 Kernel control structures 77

Figure C-1 Sombrero block structure. 81

INTRODUCTION

This thesis is a design study directed at the concept of a Very Large Single Address Space Distributed Operating System (VLSASDOS) and how it might be implemented in a 64 bit or larger address space. The Operating System design presented in this thesis is called Sombrero.

A Single Address Space Operating System (SASOS) can be viewed as an operating system in which all of its parts and processes operate in the same virtual address space. A VLSASDOS such as Sombrero extends the concept to a very large address space and broadens the abstraction to encompass an entire network of computers within the single very large virtual address space.

The SASOS concept is not a new idea. All early computer designs operated using a single address space. These early computer designs were restrictive in their memory space and its protection and storage persistence and were inadequate to a burgeoning demand for software complexity. Methods were then invented for extending the memory space capabilities without going beyond the reach of CPU designers by resorting to innovations like mass storage and file systems, bank switched memory, memory swapping and paged virtual memory.

With the introduction and wide acceptance of UNIX, which is based on the process oriented and Many Address Space Operating System paradigm (MASOS), the reintroduction of a SASOS concept has been viewed by many as a new idea. Another perspective is that the currently accepted process oriented paradigm can be regarded as an entrenched workaround for the limitations of CPU and persistent storage hardware.

It may be stating the obvious but ***if we accept for the moment the notion that process-like OS designs such as UNIX, MACH and others are basically due to***

inadequate hardware, then by assuming an adequate hardware design we should be able to develop an Operating System which exhibits some startling improvements without introducing a whole new stable of problems. This thesis professes to be such an Operating System design study and makes conceptual comparisons in support of the above argument. ***More importantly it should be noted that modern hardware implementations are within reach of providing the needed hardware design.***

As a dominant theme to Sombrero, this thesis introduces the Ubiquitous Protection Domain(UPD), a concept which provides a uniform abstraction for operating in a single address space distributed over many computers. It also describes supporting concepts and innovations which might be associated with an actual implementation of the UPD in a VLSASDOS. The UPD is introduced with respect to three different hardware assumptions in order to investigate some of the possible avenues an implementation might take.

Since the actual hardware is not available to implement the UPD, this thesis has taken a broad look at a number of concepts rather than implementing and testing all or part of the design proposal. The intent is that the arguments put forth herein are sufficiently compelling to warrant further investigation, implementation and testing.

BACKGROUND

A number of research groups are exploring the opportunities offered by a very large address space. The research group developing Monads is perhaps the earliest steady effort to explore this issue and is currently engaged in developing hardware and software around a 128 bit address space. Most other groups such as OPAL, ANGEL

and MUNGI are working with existing 32 bit hardware and are moving to 64 bit hardware as it becomes available. Some of these groups were inspired by the simplifications that a SASOS offers after becoming frustrated by expanding complexities in naming and sharing data in a MASOS. Recently at Arizona State University a research group was formed by Dr. Donald Miller which is also exploring this issue.

1. UBIQUITOUS PROTECTION DOMAIN

It is interesting to note that as computer hardware has evolved and become more powerful, the operating system software has become more complex, even convoluted. It seems clear that many of the complexities introduced were needed to overcome limitations brought about by limited hardware resources including inadequate address space, protection and storage persistence. Some of these complexities include Interprocess Communications, Multiple Virtual Memory Address Spaces and File Systems.

When computers first began to appear, the major concern was to keep enough vacuum tubes on hand to keep them in working order. It is noteworthy that these ancient computers (the MARK IV or the ENIAC, etc.) share something in common with the new modern computers that SASOS researchers are investigating which most of today's more powerful computers do not have - they operated in a single address space. All of the computers' resources (such as they were) were readily available to any part of the program.

Under the influence of the logical/physical separation of memory paradigm exemplified by UNIX, and hardware that could support it, the illusion of multiple computers in one became a compelling idea. This was accomplished initially by bank switching the user space to distribute the computing resource among physically separate address spaces. This separation provided protection and elbow room for logically separate tasks and overcame the address bit limitation. With the advent of virtual paged memory it also led to the coupling of runtime address translation and domain protection. This coupling is found in most current CPU designs and is often considered so natural that the two concepts are rarely thought of separately.

In essence, this UNIX model has been retained up to now and is so pervasive that the concept of using a single logical address space is almost a new invention. There are of course operating system designs such as the Alpha Micro Operating System(AMOS) that retain the SASOS model and still support the multi-user concept but this is done at the expense of user domain protection (Alpha Micro, 1982). In fact there is only one multi threaded user domain.

1.1 AMOS - Alpha Micro Operating System

In many applications as in small businesses where security is not a major issue the AMOS computer works quite well. Security depends on user ignorance, protection on reliable programmers and reliable up time is desirable but not essential. The redeeming value of such a computer is performance. It is fast for its hardware inasmuch as AMOS uses only a fraction of the time required by process oriented OSs which depend on inter process communications for cross-domain communication.

Since AMOS operates in a SASOS environment, efficiency is accomplished by first writing all of the programs to be reentrant and relocatable including most of the kernel. If a kernel service is needed the thread of execution just traps to the kernel with a small stack (just the IP and PS) and executes there. If a server is needed its address is obtained by a kernel service and then the server is called like any other program subroutine. If sharing is needed all that is required for access is to pass a pointer. The use of messages is almost unheard of since messages are used by modern computers mostly to cross the protection domain barrier between isolated processes and address spaces. The point is that the thread of execution goes where the service needed to be performed exists rather than passing a message and doing a context switch to get into

another domain. Critical Sections, Signals and semaphores etc. are used to sequence and synchronize traffic.

The concept of a process, as known in UNIX, was invented in order to isolate, protect and provide a set of computing resources including the address space, stack, memory heap and thread of execution. Unfortunately protection by isolation also introduced overhead in the form of Interprocess Communication(IPC) and increased task switching. If an OS could be designed to provide protection without isolation and the address space could be made large enough then we could perform like the efficient SASOS described by AMOS and still meet modern requirements.

With the advent of the 64 bit CPU, it has become practical to return not only to the efficient SASOS described above but to extend the concept to an entire network.

1.2 General UPD Concept

The Ubiquitous Protection Domain is a proposal to return to the AMOS like model of executing programs in a single address space where the thread of execution is able to access and use the resources it needs without the overhead of inter process communications and context switching and yet be protected. The name Ubiquitous comes from the nature of the protection strategy, it is ever present but does not participate serially with normal program flow. In the UPD, protection acts in the background and intercedes only when violations occur. Otherwise the system functions as seamlessly as if no protection existed. No special instructions are required to cross user/server level protection domains and kernel services can be accessed with simple traps. In short the UPD abstraction operates in parallel with instruction execution rather

than acting serially with instruction execution yet protects as well as process isolation does.

The main advantages of the UPD are performance improvements and ease of programming while providing absolute protection within its defined role. The performance improvement comes from the elimination of the need for Interprocess Communications (IPC), in fact there are no processes in the generally accepted sense. The need for task switching is also greatly reduced. Ease of programming comes from the ability to use a normal subroutine call to make a server request. Absolute protection comes from the inability of any thread to access any memory not specifically allowed with specific rights.

The UPD is made possible only because of the single address space feature of the virtual environment. It allows the context (registers, pointers etc.) to be valid from one protection domain to the next. When a domain switch occurs, a pointer still points to the same object even if the object becomes inaccessible.

Formally, a UPD is an ordered pair (***M,S***) where ***M*** is a set of Memory Object Tokens and ***S*** is a set of switches or permissions for switching to a different protection domain. In practice, ***M*** would be a capability list or matrix and ***S*** would be a capability list representing domain switch rights. UPDs are represented by UPD tokens. Memory Object Tokens(MOTs) represent ranges of memory and ***M*** is the capability list granting premission to the UPD to access these objects. These will be described later.

It is a good idea at this point to introduce the concept of the token. A token, as used in the context of Sombrero, is a descriptor which is the root structure for the sets of information that comprise the state of some entity, eg. a UPD, a thread or a memory object.

The general concept of a protection domain, proposed by Lampson (Lampson, 1969) can be described as a matrix where domains make up the rows and objects make up the columns. Access Rights in the matrix can exist at the intersection of a domain and an object. The ability to make transitions from one PD to another are accomplished by placing an Entry Right at the intersection of a protection domain and a protection domain object. For example, in the matrix shown below a thread in protection domain 1 is allowed to switch to protection domain 2.

	MO1	**MO2**	**MO3**	**MO4**	**Ptr1**	**Plt2**	**PD1**	**PD2**	**PD3**
PD1	read	R/W						enter	
PD2			read	RWX	write				
PD3					write	write			

PD - protection Domain, MO - Memory Object, Ptr - Printer, Plt - Plotter

Figure 1-1 Archetype of a PD matrix

Since the Protection Domain Matrix(PDM) is generally sparse, other more space efficient methods can be used to store it. Access Control Lists(ACL) and Capability Lists(C-lists) are two approaches which overcome the sparse nature of the PDM and record the protection domains and access rights in a reasonable amount of space. The difficulty with lists of course is that they have to be searched.

Dr. D. Miller made a presentation of the 64 bit wide concepts to the Computer Science students at ASU and was asked by Dr. B. Huey if it was not inefficient to store 64 bit indexes. Essentially Dr. D. Miller replied with “Yes, but memory is cheap now." On that note we suggest that the advantages of using a PDM directly are practicable. The resource which is not cheap in protection domain checking is time. ACLs and C-Lists are fine if we have time to search them. The trade off is that it is impracticable to search a list to check the access rights every time an instruction accesses memory. A

popular and effective compromise is to use associative caches. Caches store recent list entries where, based on the principles of temporal and spatial locality, the cost of searching a list is amortized on repeated use of the same reference.

Since the UPD is basically an implementation of a protection domain matrix(PDM) and the implementation of a PDM is controversial, a number of approaches might be used for a UPD. The next few sections will present the UPD in the light of three different hardware assumptions in order to make a more thorough study of the possible strengths and weaknesses of the final approach.

UPD-I is implimented as a direct matrix, employing a protection matrix in its bulky but easily accessible form. This method comes the closest to accomplishing full parallelism between instruction execution and interdomain protection and has advantages in a multiprocessor environment. This approach requires a modest amount of special hardware, namely a coprocessor and a separate memory.

UPD-II is a list by way of a conventional Translation Lookaside Buffer(TLB) and is the poorest performer but would permit the concept to be tested on conventionally protected hardware.

UPD-III is a list by way of a Range Protection Lookaside Buffer(RPLB). An RPLB is a hardware cache dedicated to protection domains and decouples the translation and paging mechanism from the protection domains (Koldinger et al. 1992, p. 4). This is the most flexible approach but requires the greatest commitment to hardware development. It will be the assumed configuration in the balance of the thesis beginning with Chapter 2.

It should also be established that the three UPD mechanisms are all supported directly by the Kernel using Memory Object and Protection Domain Tokens which will be rigorously described later.

1.3 UPD-I or Direct PDM

In a protection domain matrix(PDM) used to manage memory, the objects are pages of memory and the domains are the users and servers. Every attempt to access memory must be checked against the access rights stored in the PDM for that memory page in the current protection domain. We propose that it is viable to store the entire matrix in RAM rather than converting it to a list and searching it.

Let us name such a memory the Protection Domain Matrix Store(PDMS) and separate it from the main memory so that it can be checked in parallel with main memory accesses. Using such a PDMS during a normal memory access will result in a protection fault if an attempted access does not match the access rights stored in the PDMS. The advantage to this approach is that it can be made to work by adding a small amount of hardware, for which the technology already exists to build, to existing CPU hardware and that no time is required for searching lists.

We will explain why this is a viable alternative.

The basic access rights to a page of memory are no-access, execute, read and read/write. These four rights are represented using only two bits. This means that four PDM objects can be represented in each PDMS byte. If we assume for discussion purposes in the remainder of the paper that our pages are 4kB each then 4kB of memory can be represented in a PDM by just 2 bits. The memory cost to provide a PDMS then becomes related to the ratio of the (size of the PDM object entry)/(size of a

memory page) which is (.25 bytes)/(4096 bytes), one sixhundred thousandth or .0061% of the memory space.

Even with this low ratio, a PDMS directed at the virtual memory would be too large to be practical. If however the PDMS is maintained to reflect the access rights to the set of physical pages to which virtual pages have been mapped the matrix is reduced in size to a manageable fraction of main memory. To support this approach requires a close association between the page allocation and replacement algorithms and the PDMS management.

Using the above scheme the number of pages which require protection and therefore the size of a PDMS is determined by the number of pages of physical memory rather than of virtual memory. If our main physical memory is 1MB then it is divided into 256 4kB pages. So 256 pages / 4 pages / byte = 64 bytes for each PDM. If our system normally requires under 128 simultaneous PDMs, the entire PDMS can be implemented using a coprocessor and 128 * 64 bytes = 8kB of memory. With a ratio of 8kB/1000kB or .8% this does not seem to be an excessive burden or overhead in memory cost to implement the PDMS. If the size of physical memory is scaled up, the cost ratio remains the same.

The UPD coprocessor mentioned above would be a fairly simple device designed to monitor bus activity and validate memory accesses. It would work by checking the PDMS against every bus memory request causing a bus fault on an illegal access. Some of the algorithms it would use are described later. In the case where the number of defined PDMs exceed the size of the PDMS the PDMS will have to be regarded as a cache for the most recently used PDMs. An LRU algorithm would probably work for managing the PDMS as a cache.

1.3.1 UPD-I Switching; S-lists

The next issue to address is Protection Domain switching. Since the entries in the matrix are only 2 bits wide it should be apparent that entry points to other domains cannot be stored directly in the matrix unless we add sufficient bits to each entry to store an address or two. Since domain switching is expected to occur infrequently compared to memory access, it seems more practical to store the entry points as a switching capability list (S-list). This list can be stored in main memory and searched in the event of a no-access fault. It might also be practical to store the switch list in parallel memory with the PDMS if the UPD coprocessor is more sophisticated. This would certainly be the better performance approach since it is expensive to interrupt an instruction and restart it.

An entry in the capability list for domain switches is an ordered triple (A,D,X) where A is the address being accessed, D is the PDM to switch to and X is the 2 bit rights set on which a transition is allowed.

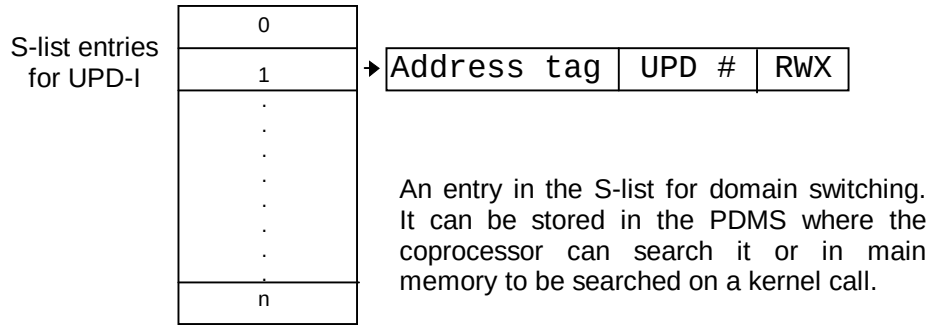


Figure 1-2 UPD-1 s-list entry

1.3.2 UPD-I Example

Recall that in the example system there is 1MB of main memory (address range from 0 - FFFFFh) or 256 4kB pages. Each PDM must then be 64 bytes. We will also assume that there is space in the PDMS for 128 PDMs requiring the PDMS to be 8kb. The current PDM number is stored in a coprocessor register called the Protection Domain Matrix Register(PDMR). This coprocessor register provides a base address in the PDMS for the rights matrix of the current protection domain. It is maintained via an I/O port.

If virtual address page 800h translates to physical address frame 64h and the current value of the PDMR is 10h a PDM switch might occur as follows:

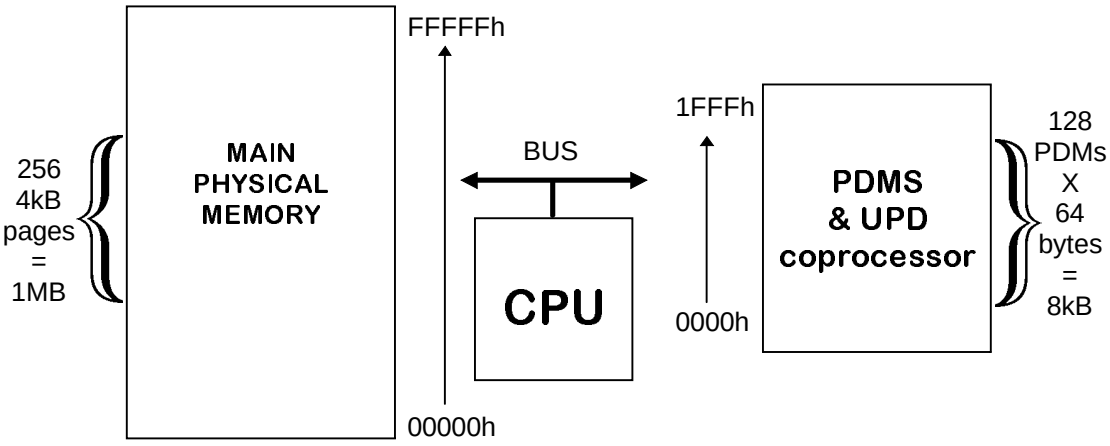


Figure 1-3 Representing the hardware used in the UPD-1 example

- A thread attempts to execute at address 800000h in the current protection domain.
- The read misses in the CPU memory cache and the access is translated to address 64000h which is frame 64h in physical memory. We are assuming a virtual cache for this example.

- The UPD coprocessor forms an address from the address on the bus by appending 7 bits from the PDM register(PDMR) and the high order 6 of the 8 bits from the frame number. The PDMS address is 419h. The byte at address 419h in the PDMS is read and the required bit pair is selected by the remaining 2 bits of the frame number.

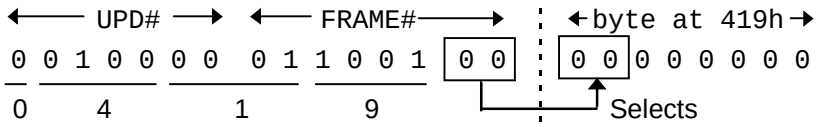


Figure 1-4 Bus address selecting PDM bit pair

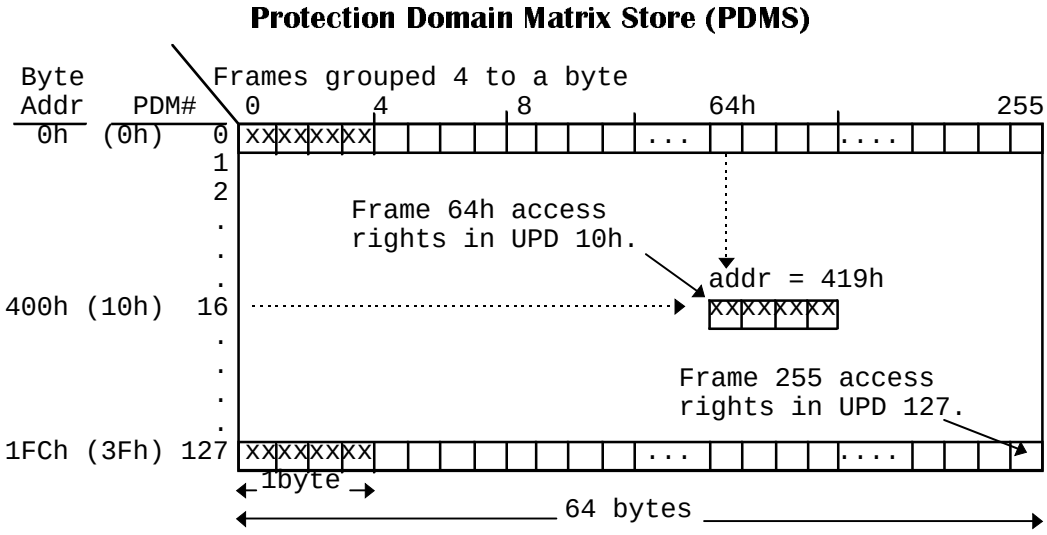


Figure 1-5 Protection Domain Matrix Store (PDMS)

- The selected 2 bits are compared to the attempted access while the instruction fetch continues in main memory.
- Lets assume for the example that the rights turn out to be no-access.

- For the case where the S-list is in main memory, an illegal access fault occurs and the S-list is searched in kernel mode. For the case where the S-list is in parallel memory, the bus cycle is extended while the coprocessor checks the S-list for the current PD and remains in user mode.
- If an entry is found, the current PDMR is loaded with the new PDM number from the S-list and the CPU cache is flushed. Flushing is necessitated by the fact that the on board cache may contain data which the new domain has no right to access. A hit to the CPU cache is not checked against the external UPD coprocessor.
- The access rights are rechecked against the new PDM and if the new rights allow an instruction fetch the bus cycle is completed and program execution is resumed.
- If a no-access fault is not backed by an S-list entry then the problem is resolved by the kernel.

1.3.3 Pros and Cons

A number of advantages exist in providing protection using the UPD-I concept as opposed to building domain switching and protection into compilers or to remaining in a MASOS environment. The advantages of a SASOS have already been pointed out. Protection Domain switching by building the mechanism into the compiler creates additional structure and overhead and formalizes domain boundaries better left simple and efficient. Placing capabilities in messages implies Inter Process Communication which UPD is attempting to eliminate as one of its goals. Placing the capability list in the object or segment header creates a formality at a higher level of abstraction when it should be left unseen at a level where only the Kernel and object servers deal with it.

Some of the disadvantages include the need to purge the CPU cache on every domain switch since the PDMS coprocessor is external to the CPU and cannot validate the current contents of the CPU cache. Another is the need to maintain the PDMS at every change of physical frame assignment which increases the complexity of the page replacement algorithm. We suspect this will not be a serious problem but this can only be determined by experiment which is beyond the scope of this paper. Some reasons why we do not think PDMS maintenance will be serious burden to the page replacement algorithm are: Almost all of the information needed to maintain the PDMS is also required to perform the page replacement; The list of memory objects has to be searched anyway to determine the access rights to the new page. The PDMS can be updated during this search. Remember an update to a PDMS for a page entry requires only changing 2 bits, this can be done with 2 or 3 instructions * the number of active PDMs. A small number compared, for example, to the cost of transferring a page of memory (4kB of data) let alone the other operations involved in the paging algorithm.

Other disadvantages includes the necessity of restricting to a uniform page size the granularity of the protection mechanism for all PDMs. This problem is alleviated by UPD-III but can also be largely done in UPD-I by increasing the complexity of the coprocessor. Another disadvantage is the need to design additional hardware, in particular a coprocessor for reading and interpreting the data in the PDMS. Note that the Kernel maintains the PDMS treating it as normal memory while in Kernel mode.

1.3.4 UPD-I Hardware

The UPD-1 concept can be implemented on today's CPU hardware. In particular we have looked at the Intel 486, Motorola MC68040 and the new DEC Alpha processor.

The Intel and Motorola processors are 32 bit processors and are interesting to this discussion in that they are indicative of the signals made available to the bus by modern processors and general CPU architectures. They both support signals which would allow the PDMS coprocessor to detect the type of access and determine its legality. They also support bus fault and interrupt signals which would allow the coprocessor to report access violations and invoke the kernel when needed. They are also able to extend the number of clock cycles in an access when needed. They are also able to support burst cache filling and writethrough in conjunction with the PDMS coprocessor. Cache writeback cannot be supported unless delayed access violations are acceptable. A pipeline mechanism must also be established which will allow the processors to write at normal speeds while the PDMS coprocessor has opportunity to detect violations before memory can actually be altered.

A signal which the processors do not support is a means to flush the internal caches using an external input. Ergo CPU cache flushing on domain switching requires an interrupt to access the privileged cache flush instruction. This problem occurs for the case where the coprocessor searches the S-list in PDMS memory and the processor stays in user mode.

The DEC Alpha processor besides its 64 bit architecture provides an on board SRAM controller intended to manage external cache RAM. It also supports a

sophisticated signal suite which can be used by an external device to manage the internal cache. Both of these devices may be useful in improving the coprocessor.

The PDMS coprocessor itself need not be very complex. Bitfield concatenation used to form the PDMS internal addresses are just a matter of bringing the appropriate bus and PDMR signals into the right bit positions, no ALU or other devices required. The indexed data is read into a register and the appropriate bit pair is selected by the lower bits of the page number for comparison to the attempted access. The register also has a caching effect in that as long as the PDMS address does not change the PDMS need not be reread, rather the appropriate bit select need only be made. This caching can be done by modern DRAMs which can retain the last set of outputs, or by a special register.

1.3.5 UPD-I Summary

In summary the UPD-I architecture provides a ubiquitous protection mechanism which has no overhead (time consumption) in normal operation, i.e. the most frequent case. The overhead is dealt instead to the paging mechanism but is not expected to be burdensome. This no overhead condition is brought about by the ability to check access rights in parallel with memory access. Faults would normally only occur on a domain switch and then only rarely depending on the PDMS memory available compared to the number of active domains and switches.

This architecture provides a protection mechanism which is always present but not visible to programs and thus does not require special constructs or instructions. In essence the efficient AMOS type SASOS described earlier in the paper can be implemented and still have protection domains.

UPD-I has an additional attraction inasmuch as it provides a global protection mechanism for a multiprocessor environment. If each processor has its own on-board protection mechanism the complexity of maintenance is increased. UPD-I is external and updating UPD-I updates all of the processors at once.

Other issues dealing with the operating system and PDM support not directly related to the abstract UPD mechanism will be discussed in later sections.

1.4 UPD-II or TLB Oriented UPD

Another mechanism which operates conventionally but can remain ubiquitous to the users and servers is UPD-II. By conventionally it is meant that the current hardware concept of retaining the close association between virtual paging and page protection will be retained for this model. It will be seen that retaining this association reduces performance to the level of process oriented machines but this model has the virtue of not requiring any new hardware and still permits the UPD abstraction to be studied.

The major visible difference from a process oriented OS such as UNIX will be the use of a single virtual address space in which to carry out all program execution and data storage using the UPD abstraction.

Each UPD in this model has its own memory map similar to a conventional UNIX process memory map with the addition of a switching capability list (S-list) for domain switching. The memory map is limited to the objects the UPD can legally access. Domain switching occurs when an access violation prompts the kernel to search the appropriate S-list for a legal domain switch. When this happens, if a legal switch is found, the address of the new UPD memory map is stored in the Translation Lookaside Buffer(TLB) base register which points to the current memory map, the translation and

CPU memory caches are flushed if needed (depending on whether the cache is virtual or physical and how the TLB operates) and the original thread of execution is resumed. Unfortunately this scenario probably takes as much time as context switching for IPC but it does retain the basic ubiquitous abstraction for the users and the advantages of a flat name space are still available.

If an academic implementation of the UPD concept is to be attempted, UPD-II is probably the approach that will be used. However UPD-III, which is explained next, would be my choice for a serious project since UPD-II lacks the TLB and PLB detachment important for some optimizations.

One advantage which UPD-II does have over UPD-I is that the thread objects (described later) can be implemented by providing a secondary memory map to be searched in the event that the UPD memory map does not contain an expected entry. Thread objects and their memory maps are discussed later. A poor performance is expected for thread objects except on the DEC Alpha which allows the TLB search mechanism to be reprogrammed. NOTE: The TLB search is implemented as a PALcode instruction on the DEC Alpha. All PALcode instructions are calls to privileged code and may be reprogrammed on site to implement special OS requirements.

1.5 UPD-III or Ideal UPD

Yet another approach to the UPD concept proposes what can be considered the *Ideal* implementation and provides the hardware assumption for Sombrero. The drawback of course is that DEC would have to redesign its Alpha processor. This may take a while.

In short the UPD-III or Sombrero design is implemented in the processor using an independent associative cache dedicated to protection domain management, a Range Protection Lookaside Buffer(RPLB). Using this approach the protection of memory no longer has any relationship to address translation or page size, nor is object granularity required to be in any way uniform, an effect previously tied to page size.

1.5.1 Independent PD Concept

The independent PD concept was presented in a paper written at the University of Washington (Koldinger et al. 1992, p.4). The paper inspired the following concept:

On a Many Address Space Operating System (MASOS) like UNIX, MACH or Windows NT etc., the PD is always mapped to the current process virtual memory map. When a process switch occurs so does the memory map and the Read/Write/Execute (RWX) rights along with it. Up to this time it has been common to think of protection management and Virtual Memory management as a single inseparable operation.

In a UPD environment protection and translation no longer have a 1:1 correspondence, any number of PDs might be accessed by a thread during a single time slice whereas there is no need to be spending the effort to also manipulate the page translation table to cross PD boundaries. There is also no longer a forced correspondence between the number of PD entries and page translation entries. One page may represent a non integer multiple of memory objects or one memory object may have a similar multiple of pages. Another side effect of this separation is that the translation and memory caches no longer need to be purged on a context switch. In fact translation no longer needs to be on board the CPU if the memory cache uses

virtual addresses. Considering all of the above the protection mechanism is therefore best considered conceptually independent of the translation mechanism.

1.5.2 Range Protection Lookaside Buffer (RPLB)

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. Beyond these basic similarities they are quite different. They will not be further compared but rather this document will describe the intended operation of the RPLB. Again as in the preceding Section the RPLB concept presented here is based initially on the ideas of the PLB presented by Koldinger et al. (1992).

Before we proceed further it might be wise to take a brief look at what a thread object token is. This discussion will be expanded in Chapter 3 on threads. A thread object token denotes a thread and its context and like a UPD token has an associated memory object and entry point or switch list. Thus we will not only need to search the UPD capabilities on an RPLB miss but also the currently executing thread's capabilities.

The RPLB requires two protection domain registers to point at the currently valid UPD and Thread object tokens, having an effect similar to PA-RISC which uses 4 PD registers (Wilkes & Sears, 1992). These tokens 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 UPD and thread for purposes of determining a hit in the RPLB.

An entry in the RPLB for mode 1 is shown in Figure 1-6.

The S-Flag and ATO-Flags are used to invoke modes 2 and 3 in the RPLB and will be discussed later. Assuming for now that these flags are off and we are in mode

1, notice that there is a RPLB Stored Tag and a RPLB Logical Tag. The stored tag is the capability loaded from the UPD or thread list into the RPLB for memory ranges. The logical tag is the presentation or view of the stored tag bits in a manner suitable for validating any attempted memory access 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.

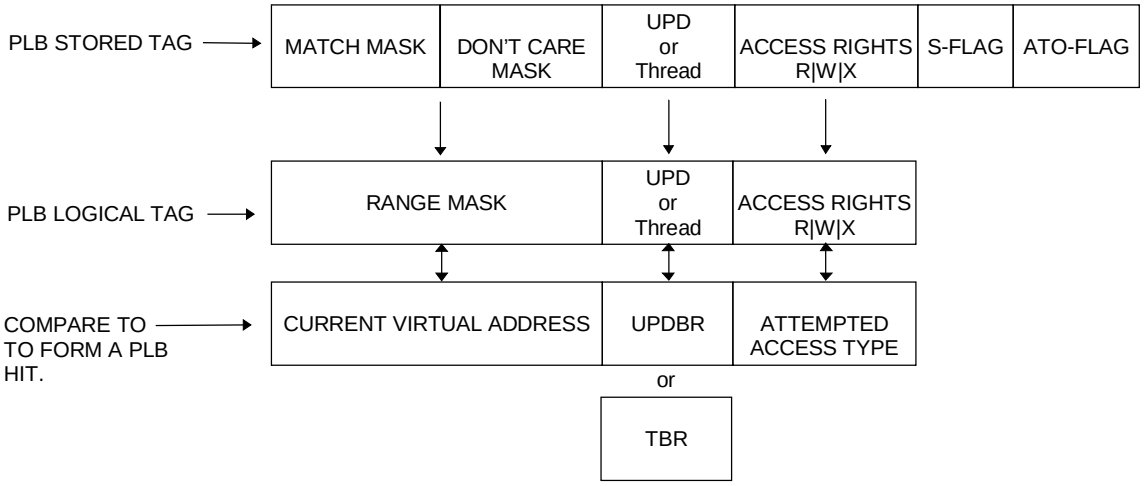


Figure 1-6 The RPLB operating in mode 1

The Range Function is that all bit positions in the Range Mask are enabled except those bit positions which have a corresponding 1 value in the Don't Care Mask. The enabled bits in the Range Mask copy the value of the corresponding bits in the Match Mask.

A hit in the RPLB occurs when the corresponding bits of the virtual address of the attempted memory access match the enabled bits in the Range Mask, and the attempted access is permitted by the access rights bits, and either the UPD Base Register(UPDBR) or the Thread Base Register(TBR) match the tag entry.

RPLB Hit example (we will use a 16 bit address space for the example, see Figure 1-7):



Figure 1-7 Range function example

Assuming the access permission and UPDBR are also a match, this particular example will hit in the RPLB on 2 blocks with addresses in the range 0C80h to 0CFFh and 0D80h to 0DFFh. A few of the features 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. By combining masks any object of any size and position may be described.
- The region accepted by the Range Mask(RM) can be disjoint. The advantages this disjointness may offer in vertical or horizontal object fragmentation and locking strategies are worth serious investigation. Notice that as the Don't Care bits are incremented in the range mask the value indexes the disjoint regions as though they were a solid block. If some auto increment or offset mapping instruction is provided to take advantage of this it might be quite interesting for masking 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 is minimal, i.e. no compare or range check is required, only a bitwise propagation which is relatively easy to implement in the hardware. Accessing unaligned data creates complications which are discussed below.
- Only entries matching the current UPD or Thread names in the RPLB (see Figure 1-6) are accessible giving the ability to cross Protection Domain boundaries without flushing any caches.

Some of the advantages and disadvantages to using the range function are:

- Object granularity to any size is completely under control of the application and yet fully supported by the operating system and hardware.
- Large even disjoint objects can be expressed by a single entry with the potential of dramatically reducing the RPLB miss rate.
- On an RPLB miss searching a table to load a Stored Entry into the RPLB is not as simple as the table searches done by the TLB. Strategies for designing an optimal search graph based on the range mask will need to be studied. This might include an algorithm similar to a Hamming code where the frequency of the use of match mask bits determine the placement of an entry in a search tree. Object sizes can also be stored as ranges to reduce the number of table entries when an object is unaligned, with the mask generated from the range to load into the RPLB. This provides a form of lazy evaluation where a mask will not exist unless it is needed.

◆

The unaligned access problem mentioned above 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 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.

The use of unaligned data can be a source of inefficiency in the kernel, an increase in complexity which should be avoided. Consider a perspective which shows a possible *increase* in efficiency.

Assume the case of an object which requires 35 pages of memory. In order for this object to be expressed using only a single range mask in a single MOT will require that 64 pages be allocated on a 64 page aligned address. Such an allocation gives access to too much memory and may not be acceptable to the application designer. In order to make an exact allocation the MOT representing the memory object may still be 64 page aligned but divided into at least three sub-MOTs (see Section 2.1, page 38 on complex MOTs); 1 to represent the first 32 pages (must be 32 page aligned), another for the next 2 pages and a third for the 35th page. It seems more efficient from the perspective of MOT management to use an unaligned memory object using a single MOT expressed as a from-to range. The kernel will then generate the required Range Masks (see appendix A) from the MOT as needed for the RPLB on a lazy evaluation basis just as other OSs generate page table entries from ranges. Certainly the address space is large enough to accommodate the alignment of even very large objects without fragmentation becoming an issue but there may be other reasons to conserve space. Lack of backing store, convenience of placing objects, MOT management etc.

Some objection may be raised to the management by the OS of fine grained memory objects below the level of page size. Should protection and granularity be determined by the page size allowed by the hardware? Object granularity is an abstraction which like memory protection, has been lumped in with the page translation tables due to hardware and design limitations. Actually Protection and Object granularity are closely related since granularity cannot exceed the size of the protection range and has been tied to the hardware as a matter of expediency not abstract need. If page granularity works optimally for an application then it should be used but OS design may not be the best design level at which to make such a decision. The Sombrero memory management abstraction has no direct relationship to the page tables and so there is no fundamental reason to limit range masks from describing objects smaller than a page. **In Sombrero the cost to manage a memory object is unrelated to its size rather to its alignment.** This design concept increases portability to machines using different page sizes and leaves design optimization up to the application designer.

1.5.2.1 RPLB Switches and Sharing

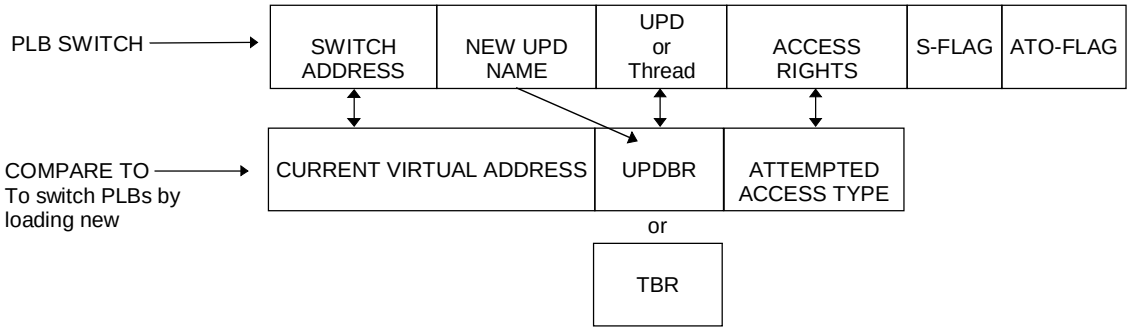


Figure 1-8 RPLB in modes 2 and 3

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 capability which was loaded from the list of switch permissions. For modes 2 and 3 the Match Mask must match the attempted memory access virtual address exactly. When a hit occurs on a stored entry with the S-flag set the UPD Base Register(UPDR) is loaded with a new UPD address/name using the value stored in the Don't Care Mask (A UPD name is the address of the UPD token in the kernel memory).

A difficulty of UPD Switching occurs in the sharing of data between UPDs. Frequently a switch will occur because a subroutine is called in another UPD 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 UPD in which they will be accessed needs to have corresponding RPLB entries. The profusion of entries needed to support access to just one thread stack for a thread which may access any number of nested UPDs is large.

The Allow Thread Objects(ATO) flag in a stored entry switch turns on RPLB mode 3 (only if S-flag is also set) and provides the option of allowing threads which enter a domain via that switch to form RPLB hits by matching not only the UPDBR but the Thread Base Register(TBR) as well. This OR operation permits memory objects to be associated with a thread address/name (a thread name is the address of the thread token). The effect is to give each thread its own capability list which travels with the thread and is accessible when the ATO flag is true. The accessible memory for any given thread is then the union of the UPD memory objects and the thread memory

objects. By this means thread-particular memory objects such as a stack may be carried to any level of UPD nesting without the profusion of RPLB entries for every UPD that would otherwise be needed. The thread objects are accessible only by the controlling thread (unless they are shared) which permits threads to have private data. In essence threads may have an independent protection domain.

It is suggested that a policy be followed that thread memory objects are R/W but not executable. This prevents a thread which originates in the outer user level from carrying a Trojan horse into a trusted server. By this means only data may travel with a thread and complete control is retained by the UPDs.

1.5.2.2 RPLB Comparison

To conclude the discussion of the RPLB a comparison between Sombrero's UPD-III RPLB and Koldinger's PLB is in order (1992, p.4). Koldinger suggested a PLB which like the TLB is built around the page entries. This requires each page in each memory object in as many PDs as that memory object is shared to have an entry. The Range Function provides a more efficient and flexible solution by providing a variable granularity and requiring as few as one entry per MOT for even very large memory objects.

In other respects the two PLB concepts provide similar advantages such as never having to flush the TLB or memory caches. In both cases the TLB can be moved into the secondary cache to perform translations outside the CPU since translation is needed only to access main memory (assuming a virtual CPU cache). The offboard translation may in fact have a performance advantage in a multiprocessor environment since only one translation buffer will need to be managed.

The question of efficiency of using an RPLB comes up. By creating another device, the RPLB, the number of CPU faults must increase in order to support it. We do not believe this will be a great disadvantage. While page faults commonly occur in the translation mechanism most interruptions are the relatively inexpensive table searches. Actual faults only occur to load a missing page. In the RPLB likewise most interruptions correspond to the table search variety. In fact the correspondence ends there since capabilities do not sit out waiting to be loaded by a fault like missing pages do. I.e. the RPLB almost never faults. As for the table searches the Range Mask reduces the need for RPLB table searches and when they occur they will most likely correspond to a TLB load. If the two are tied together the increased load the RPLB causes can almost be discounted.

1.6 UPD Tokens

Tokens in general are used by the kernel to manage access to abstract objects. They provide a root point for the state data for abstract objects. They represent a fundamental object class within the kernel. They can be locked, copied, executed and in general determine who has which resources when. UPDs, memory objects and threads all are represented by a token and may travel about the network. The usefulness of these abstractions is dependent on the application.

A UPD Token is a Kernel resident memory object which is the root descriptor for the state and resources associated with a protection domain. It provides a uniform means of handling UPDs in and across nodes. It is persistent and has a unique name (address) which allows it to be identified network wide no matter

where it originates. It is a member of the token class of kernel objects and like other kernel objects it represents itself.

A UPD token is executable and can be used to start a service or restore a user state, i.e. it points to an entry point in one of its executable memory objects which can provide a service like restoring a user's last configuration or starting a service. The semantics of executing a UPD include creating a thread for the UPD and starting execution in the indicated entry point. This permits services to be named on a per instance basis (the name is the address of the UPD token) rather than generically. Each version of the service can be configured to a particular requirement such as the user with its particular set of modules.

Only one token can be the original but copies can exist on multiple sites so that a single service may appear on many nodes.

Token tracking will be discussed in Section 4.1 page 50.

1.7 Comparison to Other Work

Other researchers investigating SASOS benefits have chosen suitable strategies to meet their design purposes. We will compare the UPD concept to the comparable parts of some of these efforts.

Opal is a funded project at the University of Washington and has been implemented on the DEC Alpha processor (Chase et al. 1994). It works on top of MACH and is able to communicate with conventional coresident UNIX processes. Opal itself is a MACH server which manages a wide address space as a SASOS but is actually implemented as a separate virtual space in a MASOS environment.

Opal uses password capabilities to name memory segments (Chase et al. 1994, p. 8). An ACL is used to validate attempts to access the named segments and attach the segments to the current protection domain. UPD requests attachment of memory objects directly by name (symbolic or otherwise) via a kernel service and proceeds on an affirmative response. Both Opal and UPD may request attachment by faulting.

Domain switches in Opal are accomplished by way of portals. A portal is an entry point for program execution in a new protection domain and requires serial execution. The UPD analog is the UPD switch which performs in parallel (in reality as well as abstractly assuming it has an entry already loaded in the RPLB) rather than serially. Password capabilities in Opal are implemented on portals.

Opal functionally resembles UPD-II in terms of the underlying hardware and probable performance. When UPD-III is implemented we are confident it will compete nicely with Opal.

In an additional note, the Opal authors criticize the use of the OS or hardware for managing memory protection below a course grained level. Opal segments are also bound to a contiguous multiple of the page size. UPD leaves both options up to the user.

ToM is an effort at Toshiba Japan to optimally use a very large address space (Okamoto et al. 1992). ToM divides the address space into sections which can be either Text, Data, Stack, File or Gate. Each of these sections has specific kinds of relations to the others. ToM uses a gate Section, similar to Multics and the 80x86 (x>=2), validated by an ACL to implement a user level RPC and avoid kernel calls. ToM intends to use a fine grained MMU. ToM also uses inline switching via the Gate

sections. On this basis a UPD-II level of performance is predicted. ToM associates protection domains with threads - UPD does this to some extent.

It is difficult to compare UPD with particular aspects of ToM as the paper did not supply sufficient details on which to base a comparison.

Angel has abandoned UNIX support in order to have the flexibility to develop SASA (same as SASOS) concepts (Murray et al. 1992). It is however retrofitting a POSIX interface. Angel loosely retains the process concept but detaches protection from a direct association with processes. Instead protection may cross process boundaries or there may be more than one protection domain in a single process. PDs in Angel are associated primarily with thread/memory object associations. Communication between these PDs is by shared memory objects and LRPC. The concept of kernel supported messaging has been abandoned.

Again based on the serial aspects of the switching mechanism we rate Angel with UPD-II, however the overlapping of processes and protection domains may provide a performance advantage.

Mungi uses a capability based protection mechanism to attach objects to protection domains (Heiser et al. 1994). An execute mode called Protection Domain Extension allows temporary alteration of the protection domain and is, it appears, the closest analog to UPD switching.

All of the systems noted above operate in a manner philosophically different from UPD-III. This is due in large measure to the use of currently available MMU hardware which has page granularity and associates protection with specific page translations thus involving the paging mechanism in every protection issue. The association with paging also prevents the implementation of subroutine like entry points

outside the current domain without the overhead of serial kernel intervention. This is because page size granularity is too large to restrict an entry point into another PD to a single address (it could point to anywhere on the page) and there is no mechanism to change the TLB base register to point to a new memory map based on a mere access. It also creates the need for cache flushing in the TLB and memory cache to prevent illegal access on domain and context switches. UPD-III overcomes all of these issues.

Most of the SASOS strategies mentioned above seek to improve performance by optimizing operations within a PD to reduce the number of PD switches. This has inherent limits since a PD can not contain all of the system's resources and must eventually reach across the PD boundary. UPD-III has little or no penalty (assuming a 95%+ hit rate in the RPLB) and encourages a very modular system design.

1.8 Contributions and Summary

To recap and distill these are the contributions presented in this Chapter:

- The Ubiquitous Protection Domain(UPD) is presented which provides protection and parallel domain switching. This has the potential of being extremely efficient and simpler to use. Other concepts provide a serial domain switching mechanism such as password capabilities and portals or switch instructions.
- The Protection Lookaside Buffer(PLB) is improved to become the Range PLB(RPLB). It will cache protection domain switching from an S-list in addition to Koldinger's page capabilities.
- The RPLB is also improved to compress the number of entries to as few as one for any size and even disjoint memory objects. This is called the Range Function.

- The RPLB is also improved to optionally support thread protection domains in addition to the expected UPDs. This extends the accessible memory for any executing thread to be the union of what the thread permits and what the thread's current UPD permits. The value of this is in reducing the profusion of switch entries in the S-list for threads entering and exiting UPDs. It also allows memory unique to the thread within a UPD.

In most modern operating systems the use of virtual paging is an example of a ubiquitous facility. It extends the address range beyond the limits of physical memory but otherwise imposes no structure or limits (except page granularity) on the programming or user environments. One goal of this research has been that the UPD should be equally unobtrusive, operating in the background and as fundamentally supported by the kernel. I.e. a low level operation not interfering with normal operation and not imposing implementation or structural constraints.

To close this Chapter a little off the subject it should be noted that the SASOS concept, though ambitious, does not eliminate the need for the MASOS concept. Indeed, no matter how large the network which may exist within the umbrella of a single address space OS there will always be a neighboring network to communicate with which exists in a separate virtual space. SASOS only changes the base unit of the problem but in so doing provides the opportunity for many optimizations within each base unit.

2. MEMORY OBJECT TOKENS

This Chapter describes the Memory Object Token(MOT) and some of its potential uses. The MOT itself is a kernel object and varies in size according to its current use. It is referenced internally via a handle giving one level of indirection which permits the actual data structure to change as needed in order to conserve memory. One structure is used for leaf MOTs, another for parent MOTs and another to trace transfers across the network. At the user level, unlike Thread and UPD tokens, the MOT is named after the lowest address in the memory object it defines rather than after the token address. MOT execution semantics are intended to implement object oriented concepts and permit a custom calling signature to invoke object methods. Kernel objects like the MOT are executed using a system call.

Since the same virtual address space is common to the entire network, a means must be provided to regulate access and control ownership to regions of memory. MOTs provide the fundamental units of memory access and control across the network. The main strategies behind using the MOT are:

- To remove any control structure from the memory objects themselves.
- To place all control structures in a common protection domain (the kernel) so that they may be freely managed without barriers. The RPLB provides the bridge between the user space and its kernel level structures.
- To provide a single view of memory. All MOTs can be viewed as the root of an independent memory object or as a child of a complex MOT of which it is a part. This view persists from the initial MOT which describes the entire address space to MOTs which describe as little as a single byte.

- To provide a path to find memory objects which have been transferred about the system. This will facilitate various semantics to be employed in read/write access and object consistency.

Formally a MOT is an ordered pair (***R***,***P***) where ***R*** is a region of the address space and ***P*** is an access permission set (Read, Write, Execute or No access). Regions may be expressed as input to the Range Function described in Section 1.5.2 page 22 or as an address range. Permissions describe the type of access the owner of a MOT may have to the memory object bounded by the region R.

The fundamental storage abstraction in Sombrero is the Memory Object. Currently most systems use the file abstraction for static storage and volatile memory is organized into pages / segments / memory objects.

For contrast compare the file abstraction to Sombrero memory objects. Files are given names and stored over physical sectors or partitions on hard disks. File data is buffered and managed by reading from or writing into pages of memory which can be grouped into the segments or memory objects mentioned above. Of late UNIX has also provided memory mapped files as does MACH, Windows NT and other new OSs. In a SASOS environment such as Sombrero it becomes natural to use a memory address as the permanent name and flatten the file abstraction into a simple memory object abstraction which is always memory mapped. This works because any body of data with few exceptions will fit readily into the very large address space. Conventional names may be assigned via a name server which provides a symbolic name to/from address translation service for memory objects.

There remains only one abstract storage structure in Sombrero; the memory object. This is justified because memory objects are not specialized and have no

imposed internal structure such as headers or access lists, rather these structures are provided by the MOTs. Any allocation of memory for any purpose from I/O ports to data storage or program execution is a memory object and is handled via the MOT. As noted above a MOT describes the memory object and grants access to an arbitrary sized area of the memory space which is also not necessarily contiguous (refer to the Range Function).

In most systems, the logical unit of memory which operates between disk and RAM is the page. In Sombrero, memory is described as a region ***R.*** ***R*** has two components, a Match Mask and a Don't Care Mask, or when the region is unaligned a more compact representation is to use a from-to address range. The two components of ***R*** are used to generate a third component, the Range Mask. Whenever a from-to address range is used, it is translated into appropriate match and Don't Care Masks as needed. These logical regions are mapped onto the virtual pages and are only coincidentally ever a multiple of the page size. For further information refer to the RPLB in Section 1.5.2 page 22 and appendix A which describe the range function and its use.

2.1 Complex MOTs

Complex MOTs are MOTs whose region R is described in terms of two or more other MOTs. These MOTs form a rooted tree whose leaves are simple MOTs and jointly describe the access configuration for an entire memory object. Since the entire memory space can be regarded as a memory object, lower order memory objects can be regarded as sub-memory objects which likewise together describe the configuration of the entire memory space. Sub-regions of complex MOTs not explicitly described by

a leaf MOT default in meaning to a MOT with no-access permission. Sub-memory objects may only have a subset of the rights held by their immediate ancestors.

Via the use of MOTs memory objects can be dynamically partitioned and rejoined as programs request or give-up access to portions of a memory object. These portions can be moved about the network as objects in their own right. When they are released by all users, i.e. no capabilities remain for a given MOT in any UPD or Thread memory object list (normally the user counter in the MOT is decremented to zero), the MOT is discarded or rejoined with its neighbors to form a larger contiguous region of memory. If all of the children of a node are discarded, the parent reverts to a leaf. If a lone remaining child because of merging becomes identical to its parent, the child can be coalesced with the parent also returning the parent to the status of a leaf. This amounts to automatic garbage collection.

2.2 Garbage Collection

The principal reason garbage collection is considered a difficult problem is because of the difficulty of preventing dangling references. In the token based Sombrero OS, all users, servers etc., in order to access memory must also retain access rights to that memory thus making the pointed to objects persistent. All pointers therefore automatically remain valid until they are deliberately made invalid when the owner of the pointer discards its access rights to that memory. If a dangling pointer then remains it will be invalid anyway and cause an illegal access violation when an attempt is made to use it. Furthermore, in the case of a system failure where MOTs or other vital data may be lost, recovery can be accomplished by analyzing the capability and access control lists of all UPDs and Threads. Where references exist without

tokens, tokens can be recreated, conversely where tokens exist without references they can be held for a time and discarded if no reference is reestablished. A reference is of course a handle to a MOT held by the referencing UPD.

Searching the capability list of all objects to perform a recovery is normally a daunting problem but since Sombrero provides a compact and uniform mechanism (the MOTS) for storing this information it should be reasonable. Most systems store capabilities in applications and other non uniform areas in user space which makes the problem unwieldy. Some SASOS developers indicate that garbage collection is probably an intractable problem. The intractability is due not so much to the volume of capabilities but to the nonuniform storage of capabilities. Sombrero provides a uniform method for accounting for its memory objects. The need to search the C-lists will also be rare (hopefully the system will be reliable) since garbage collection is normally automatic. It should be searched only to perform a system failure recovery.

Another point is with regard to Memory Object Persistence. In Sombrero persistence is accomplished by placing a MOT handle for a memory object in the name server which never gives up its handles until a user directs it to. The memory space will then be persistent since the reference count is non zero. Persistence is maintained for any object which has an outstanding reference. For example a UPD will retain objects until the object is removed from all UPD capability lists or the UPDs are deleted.

2.3 Conclusions

The authors of Opal put forward the conclusion that fine grained memory management should not occur at the kernel level (Chase et al. 1994, p.6). Indeed Sombrero permits fine grained control to the level of a single byte. The Opal authors

base their conclusion on the difficulty others have had attempting below the page level control in the past using traditional architectures. In fact Opal is paged based in its protection which virtually forces at a minimum page granularity. Sombrero is not page based, a vital difference.

The Sombrero architecture using the RPLB described in UPD-III requires no additional cost to manage fine or course grained objects simultaneously. In addition since everything is operating in the same virtual memory and the need for pointer conversion is absent in a SASOS the kernel's task is simplified. Below a course grained level, it is not argued that everything should be managed by the kernel but with Sombrero the limitations are a choice of the application designer rather than something imposed by the OS designer. See Section 1.5.2 page 22 on the RPLB for the important discussion on this subject.

The use of the MOT concept affords a simple structure on which to build a highly flexible yet systematic means of controlling access to memory. For example by separating read and write access into separate tokens, a mask can be generated which allows read access to an entire set of records stored in a memory object while allowing write access to only a portion of each record. This can be done with as few as two tokens even if the memory object is very large. Additionally if a record or a group of records is to be locked for access by another thread on another node, the read and write MOTs can be partitioned leaving an access 'hole'. This hole representing exclusive access or a lock to a portion of the original object can then be temporarily treated as an independent memory object and transferred and held by another thread or UPD on another node. When released the hole is filled and the original token restored.

The last example demonstrates an advantageous programming strategy since locking can be dealt with defacto at a system level which simplifies programming efforts. The ‘lock’ i.e. the hole also disappears when no longer in use thus allowing fine grained locking without precommitting locking resources. That means that every memory object or part of an object is lockable without special provision. This arrangement when combined with an optimistic update service that guarantees an uninterrupted validate-then-write sequence may cover many of the network internode update issues. See the network Chapter 4.

It is not suggested that this method of locking be the only method supported by the OS for lock management. We are only pointing out that the complex MOT structure has the side effect of providing a locking method which may prove advantageous especially on an internode basis. It is further suggested that allocating a locking object as Windows NT does and then performing locking operations is convenient for a single node but does nothing for a network. Sombrero can use its MOTs as a locking mechanism seamlessly across intra or inter node boundaries for applications which require broad locking abilities.

The viability of these masking, partitioning and locking strategies can only be determined by testing. Certainly fine grained locking/allocation at the system level would be inefficient for many applications whereas courser grained or infrequently accessed objects might be managed readily. In any case the uniformity and absolute scalability of memory management from very large grained objects to very small grained objects is appealing and warrants further study.

3. THREADS

The Thread is the fundamental unit of execution in the OS. Formally a Thread is an ordered quad (***U***, ***S, M***, ***C***) where ***U*** is the current UPD in which the thread is executing, ***M*** is a set of MOTs i.e. a capability list, ***S*** is a set of switches, also a capability list and ***C*** is the CPU state for the thread (Registers, SP, IP, CCR and other state and control data). A change to ***U*** occurs automatically without serial inline requirements if program execution is attempted at a permissible transition address. The value of ***U*** is simply changed to the value of the new UPD by fetching the new ***U*** from the RPLB. Changes to ***M*** are by attempting an access which the kernel accepts by updating ***M*** or by requesting or deleting entries via a system call. Changes to ***S*** occur in a manner similar to ***M***. ***C*** changes dynamically.

The major distinguishing feature of Sombrero threads from UNIX type threads is that they travel from one PD to another. This concept is taken from the Alpha Micro operating system (AMOS), a multi user SASOS which has no memory protection except for kernel mode. AMOS initializes the system with a pre-set number of threads or Jobs as they are known in AMOS parlance. These threads are never terminated. A user is assigned a thread and all services for that user are performed by that single thread. In modern parlance this might be described as a traveling thread. Sombrero threads will be isolated via the RPLB unlike AMOS but the efficiency of the traveling concept remains. Unlike AMOS Sombrero threads are created and terminated as needed.

By comparison the UNIX OS threads are isolated to a single process. MACH continuations amount to an attempt at providing the concept of a traveling thread in a MASOS.

Thread tokens are the structure for storing the current thread UPD, MOTs, and state. Thread tokens are kernel objects and like UPD tokens and MOTs are

executable. The semantics of executing a thread amount to performing a context switch to transfer the remainder of the current time slice to the executed thread, a hand-off. The side effect is that an idle thread is placed in the scheduler run queue. Thread hand-offs also support a custom signature which can pass arguments into the registers i.e. the service which performs the hand-off accepts an argument list containing register values to be placed in the other threads register stack.

3.1 Inter UPD Thread Management

While the UPD abstraction allows simple subroutine calling and otherwise normal instruction execution to initiate the crossing of UPD boundaries, there are a number of issues that must be addressed in order to make this abstraction workable. The issues are:

1. Arguments and data marshalled for a thread in one UPD need to be accessible in another.
2. Permission to enter a UPD does not automatically grant permission to return from a UPD.
3. Security between UPDs. One UPD may not trust another.

All three issues are access issues created by the boundary between the UPDs. The straightforward approach to these problems is to engage in a great deal of permission setting and memory sharing. It can be shown that the first two issues can be solved by giving the threads their own set of memory objects and switches.

For example assume that five users in UPDs A,B,C,D and E each with a single thread need access to a service in UPD S. Assume further that S needs access to UPD X. Permissions need to be granted to A-E to enter S. Permission is also granted for S to enter X. Permissions must also be granted for threads in S to return to A-E and threads in X to return to S. So far 12 switch permissions.

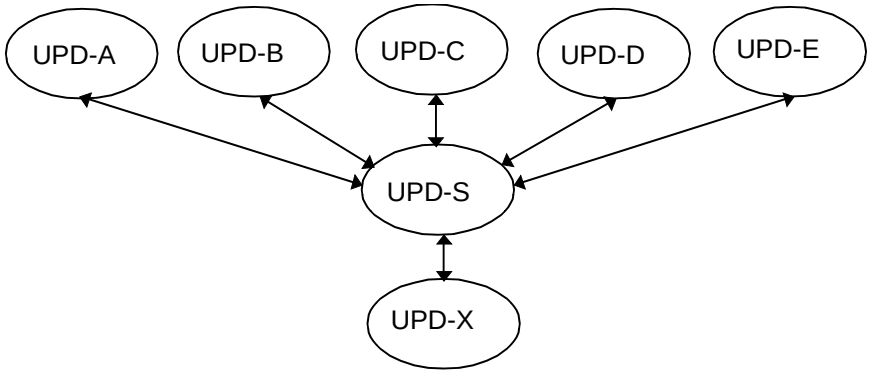


Figure 3-1 UPDs and connecting edges representing boundary crossings

Next, access must be allowed for shared workspace memory and stacks. These objects must have R/W access in not only UPDs A-E where they originate but in S and possibly X as well. Permissions for 5 stacks and 5 shared workspaces then require 30 additional UPD object entries. It can be seen that the number of permissions is nonlinear when the number of threads and involved UPDs increase as follows:

*Threads * UPDs each thread may enter * shared memory objects per thread + UPD boundaries to cross * 2 = number of permissions required to support thread access.*

*Ex: 5 * 3 * 2 + 6 * 2 = 42*

As complexity increases it is easy to see that the number of permissions to support thread access can become unwieldy, $O(n^2)$. To alleviate the shared memory problem the objects which are primarily associated with the thread can be attached to the thread instead of the UPDs. The number of permissions then becomes:

*Threads * shared memory objects per thread + UPD boundaries to cross * 2 = number of permissions required to support thread access.*

*Ex: 5 * 2 + 6 * 2 = 22*

Which is a considerable improvement in that it reduces the problem to a linear O(n) problem and solves issue number 1. By assigning the memory objects to the threads another benefit is that the objects not only travel with the threads but the threads can have private access to their own stack and data areas even when other threads are executing in the same UPD.

In the example 5 threads from 5 different UPDs A-E are entering and returning from S. If return permission is placed in S's UPD permissions then what is to prevent the threads from returning to the wrong UPD, i.e. what is to prevent the E thread from returning to A? This problem becomes more apparent if A-E are all calling from the same shared executable code and thus returning to the same address. In this case the RPLB tags all look the same for the 5 switches and this cannot be allowed since the RPLB will only ever recognize one of them returning all of the threads to the same UPD. The solution is to place the return switch in the ***S*** set of the executing thread instead of in the UPD S switch set. With the switches held privately the thread return cannot be ambiguous. Return permission is granted either by confirming the attempt or by granting permission ahead of time. Issue number 2 is largely settled although should the situation arise where two UPDs both using some of the same code and the

same thread have the same return address there is fertile territory for a difficult bug. An attempted access which is done without permission expecting permission to be granted in the fault may instead grab an existing switch and return to the wrong UPD. Care needs to be exercised where ambiguous returns can exist and the best way to prevent them is to get return permission before the call or not to share code in nested thread execution. It is anticipated that this problem will be rare. Handling the problem lexically will require participation from the compiler and imposes structure which is better kept to a minimum.

Issue number 3 has to do with the trust one UPD has in allowing another UPD to manipulate its common data areas (Chase et al. 1994, p.4; Carter et al. 1992, p.76). Since threads now have the ability to carry their own memory objects, the possibility also exists for a thread to carry a Trojan horse into a server as an executable object. The Trojan horse problem can be avoided by restricting thread object permissions to R/W but no execute. If threads cannot execute any code except that allowed in a UPD by the policies guarding the UPD then the UPD will retain control. The shared memory problem can be managed by the UPD using the ATO flag in the RPLB entry. The ATO flag can be used to disallow or allow thread objects to be accessible after switching to a new UPD. If thread objects such as the stack are not allowed then the new UPD will have to assign a stack to the thread when it enters the UPD. Exchange of data will have to be by other means such as passing parameters in a thread hand-off or by whatever means the designer comes up with. For a hand-off the request would wait until the target thread became idle and then pass the new parameters and schedule it in the run queue. This sounds like a return toward the UNIX style protection.

3.2 Thread Networks

Recall that in a SASOS all memory objects are unique network wide. If a thread token is created on node A it has a unique address on A and therefore on every other node as well. There is no fundamental obstacle to transferring this token to another node to continue execution. The RPC in Sombrero is replaced by the Traveling Thread Call (TTC). R/W/X semantics can insure that the thread did not execute in more than one node at a time although even that idea has intriguing possibilities for Multiple Data Single Instruction parallel execution using a broadcast. Perhaps different threads can be endowed with different semantics allowing both cases mentioned above to be used.

As an alternative to the traveling thread call thread hand-offs might instead be passed across the network scheduling a waiting thread on another node with an appropriate set of parameters.

3.3 Thread Conclusions

Sombrero threads allow a seamless program execution across protection and network boundaries. This has the essential effect of making the entire network look like one giant multiprocessor. While this seamless (in parallel execution and protection as opposed to serial or inline) abstraction is attractive care must be taken in the distribution of memory objects between threads and UPD to prevent a nonlinear increase in the number of permissions. Care must also be taken in the use of shared executables to prevent ambiguity in domain switching.

Sombrero threads in conjunction with the UPD are designed to avoid serial or inline requirements for inter UPD access. This is an ideal which cannot always be met in practice. For example whenever an access occurs for a UPD switch or memory

access not loaded in the RPLB a fault must occur which invokes the kernel. On other occasions permissions must be consciously set by the program by making a system call before attempting to access a resource. This amounts to moving the serial burden to another place in the program rather than eliminating it. The benefit of the Sombrero abstraction is seen however from Amdahl's law which quantifies the advantages gained from optimizing the most frequent case. I.e. the serial or inline overhead is moved out of the frequently executed loop and once an entry is loaded in the RPLB the expense is amortized over future parallel accesses via hits in the RPLB.

4. EXTENDING THE CONCEPTS TO A VLSASDOS.

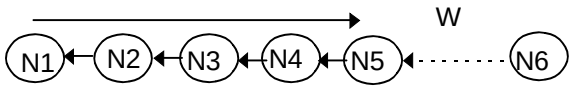
The UPD, MOT and Thread abstractions are readily extensible and can be applied to an entire network forming a distributed operating system. A number of concepts are to be explored which use the advantages of the SASOS concept. Some concepts such as the use of the token were presented above but more specific and unifying issues need to be developed for the thesis.

4.1 Token Tracking

Tokens of any sort, UPD, MOT and Thread, can be replicated depending on the assigned semantics and need to be locatable in the system. A naming service can be informed of token location and then queried by users but a better performer for managing memory object consistency is to have a low level feature automatically track token distribution. This would permit an automatic distributed name service which acts selectively based on policy choices. By selectively it is meant with or without consistency and what degree of consistency and caching if any.

For the MOT, a Read_Parent pointer, Read_Graph pointer, Read_Arity count, Write_Last pointer and Write_cache pointer are data which aid in determining the location and state of every distributed token and its virtual pages. These data about token distribution also provide an automatic means of enforcing data consistency and Memory Object coherence without depending on a broadcast.

The basic strategy is to maintain a directed graph, known as the write cache graph, pointing back to the nodes which have previously owned the write token. This list usually terminates at the inception node and provides a cache of the complete set of up-to-date virtual pages belonging to a MOT. The pointers are stored in the write_cache variable in the MOT.



The write cache graph: N5 currently holds the write token W which originated at N1. N6 is attempting to read the Memory object.

Figure 4-1 Write cache graph

The Figure 4-1 depicts the previously described write cache graph. The cache effect is realized in that N5 possesses the most recently changed virtual pages of W. When N6 seeks to load a page of W it finds the most current page reading from the start of the graph. This strategy creates the opportunity to access changed pages most recent version first guaranteeing consistency. By the principal of temporal locality recently altered pages are in a nearby node on the graph while the concept of lazy evaluation leaves less popular pages nearer the end of the graph. What is the benefit of this strategy? Pages in the memory object are moved only if they are needed and when a page is read the reader is guaranteed to access the most recent version. This is done without using version numbers.

The write cache graph is pruned automatically when it intersects itself. Using Figure 4-2 as an example the pruning is accomplished by gathering all of the pages



The Write_Cache Graph intersecting itself in A prunes itself in B when W is transferred from N5 to N3.

Figure 4-2 Write cache graph pruning

updated in N4 or N5 and copying them to N3. To close the graph and return all pages to the inception node the write token need only be transferred back to N1 which causes the graph to prune itself to a single node and copy back all of the updated pages. The

nodes which are pruned out of the write cache graph remain in the read graph unless they remove themselves.

Whenever the write token is moved the write_last pointer is also updated with the new node name to preserve a path to the current write token holder. This path is used primarily by readers who want to locate the start of the write cache graph in order to obtain the most recent version of the data.



The Write last Graph before (A) and after (B) the write last graph is pruned.

Figure 4-3 Write last graph pruning

For the distributed read token (execute is treated like read) the read graph is a tree rooted at the inception node. Additions to the tree occur when any node requests initial access to a MOT. The read_graph pointer is an array with arity read_arity and indicates the degree of the read graph tree. The purpose of the read graph is to notify all parties holding a given MOT of important changes pertaining to that MOT such as:

- One or more of its pages has become obsolete by a write. If the reader needs a current copy it will obtain it from the top of the write cache graph. Page invalidates are initiated when a dirty bit is set on a page and are sent to the read graph. The effect is to leave only one primary copy of the page and depending on the semantics force the read graph to flush the page copies and reload them from the write cache graph as needed.

- The MOT is to be surrendered to a request for exclusive access. All nodes will retain only a noaccess stub for the MOT. Page invalidates are still maintained.

Messages sent to the read list are complete when the leaf nodes acknowledge back to the parent nodes (via the read_parent pointer) recursively back to the root node. This removes any doubt of the completion of the message and allows the root node to broadcast a recovery message if any children are reported severed from the tree. Recovered children are nodes which see the broadcast message without having seen it through their parent node first. They are replaced in the tree with new parents. When the old parent recovers it is informed of the divorce upon checking with its parents. If the inception node fails the nearest write cache node becomes the root and a recovery message is sent out. Any blocks needed from the inception node and not found in any other node are unrecoverable and cause an unrecoverable error. Rebuilding the read graph is done by sending a recovery message without first sending a message by the read graph.

Memory object dirty bits are stored in the virtual page table with the first and last bits redundantly stored in the MOT. The redundancy provides for Memory Objects not aligned with a page boundary or smaller than a page. It would also be useful to provide a Notify Dirty Bit in the RPLB entries and page tables so that small modified objects and pages could notify their read list when an object is modified. Perhaps the RPLB ATO-flag could be used for this purpose since it used only when the S-flag is set and switches do not need dirty bits.

The read graph has an arity in order to shorten the path to any given node holding a read token. If many nodes hold the read token the path to the furthest token

can be reduced to $O(\log_{arity} n)$. By distributing the read token requests evenly to each branch of the tree a balanced read graph tree can even be achieved and a very large graph can be supported if necessary.

The semantics which this design will support are discussed in Section 4.4 page 57 although the above facility will handily support UNIX semantics.

Tracking of the UPD and thread tokens uses simpler graphs depending on the assigned semantics. UPD and thread tokens have two semantics; replication or unique. Unique semantics mean that only one active copy of the token can exist in the network. Replication means any number of copies, one per node called clones, can operate with the access rights for the UPD or thread as a whole being the sum of the rights of the UPDs or threads on the separate nodes. MOT semantics will still prevail but UPDs or threads operating on multiple nodes can perform in parallel on separate data and exchange data when useful to do so. Memory object pages automatically travel to the correct node when requested as long as their internode semantics are not violated. Replicated tokens have a read graph. The read graph is used to perform actions on all of the clones for some operations such as determining if an access permission exists for some MOT.

Unique UPD and thread tokens when traveling take their capability lists with them and use the write cache graph to stay connected to the inception node.

4.2 Kernel Object Extensibility

Kernel objects in Sombrero are readily extensible to the entire distributed system. This is accomplished mainly via the tokens. Tokens like the tokens of a token ring network confer access to the holder and combined with the tracking features

extend the ability of any node holding a token or its clone to determine its distributed state. For example, an attempt is made to access memory object X in distributed UPD A on node N. If node N has no capability for X in A the read graph of A is summoned to check other participating nodes. The read graph will either provide a copy of a capability from another node or report no access which results in a noaccess fault.

Another powerful feature is that individual nodes need only concern themselves with their own address map since memory distribution is defined by the set of tokens not the address maps. This means that pages which are not actively stored on a node need have no allocation or representation in the address map. As pages are created or copied from other nodes the address map is extended and as pages are removed the address map can be compressed. Pages which are not primary copies (known by a mark in the address map) can be discarded when room is needed on the backing store.

4.3 Very Large Memory Allocation

Memory allocation is the process of distributing and recovering the very large memory space in the network. Garbage collection was addressed in Section 2.2 page 39 but when a node is deactivated what happens to its piece of the virtual space? How does a new node receive a piece of the virtual space and which piece? How does an address get assigned to a new block request?

Each node has a unique segment of the address space from which objects that originate at that node are allocated. Mungi preallocates a set of fixed sized partitions which it assigns to nodes as they join the network (Heiser et al. 1993, p.4). This fixes the size of the regions and the number of nodes which can be attached to the network

under the same virtual space. These regions may be large but who knows what the future will bring.

Sombrero uses a different strategy to accomplish the memory allocation but without the fixed regions or node count and without maintaining an allocation table. These features may or may not be advantages but they fall out from the use of MOTs to manage memory.

When the first node is brought on-line the entire 64 bit address region is assigned to that node under a single root MOT. As new memory objects are created they are assigned via sub-MOTs out of the lowest aligned address with a sufficient range to cover the requested block of memory leaving the upper ranges of memory unpartitioned.

When additional nodes are brought on-line they initially operate in their own virtual space and send out a broadcast requesting a parent from which they can obtain a root MOT for themselves. One of the parents offering the largest split is accepted. The parent fissions its root MOT into two equal aligned parts, the lower half containing all of its local memory allocations and the upper unpartitioned half is sent to the new node. The new node joins the virtual space and may later fission its root MOT as well. When a node is removed from the network it may send its root MOT back to be fused with its parent or another node may pick it up as a second disjoint root MOT. This second root MOT may later be given to a new node complete with its set of existing sub-MOTs. In any case you get the idea.

In the network the base address of a node's root MOT also becomes the name of the node. Owner's of multiple root MOTs are treated as multiple logical nodes. This allows a request for an address to always be directed to the inception node for that

object. This is done by having the network router route to the location of the root MOT with the highest address which is less than or equal to the requested address.

All Memory Objects are represented by a token. At least a stub of this token will remain at the inception site as long as the memory object exists. Memory allocations recognized network wide are as simple as allocating on the local node since each node manages it own unique piece of the virtual space.

A final issue, memory fragmentation. In such a large address space memory fragmentation is not a major issue except that virtual page tables also become more fragmented. Some effort to fill holes will probably be rewarded by better virtual paging performance. Sombrero's allocation algorithm and garbage collection schemes described above reduce memory fragmentation by allocating from the open regions closest to the root MOT and by fusing neighboring unused regions. The page tables are then more clustered. A hashing table might be used to grade the fragments by size for faster allocation.

4.4 Virtual Memory Consistency and Semantics

The semantics of memory access can be as stringent as UNIX semantics (one write copy or multiple read copies but not both at the same time) or as relaxed as no consistency requirement at all. These semantics can all be enforced by manipulating the tokens. Semantics might include:

1. Restricting the token to one copy which is passed around and is present on only one node at a time.
2. One write and many read with consistency (UNIX).
3. Various levels of #2 UNIX consistency with immediate or lazy update.

4. Read only.

5. Open or unrestricted Read/Write.

All of these semantics can be supported using the graph structure outlined in Section 4.1 page 50. The profusion of messages needed to support memory object consistency is reduced to no broadcasts as long as the relevant nodes remain accessible. As far as space needed to support all of these graphs, recall that these graphs are distributed and for any given node require only the space for one MOT for each set of graphs.

4.5 Network types

Since Sombrero is designed to reduce the dependency on broadcasts large connection oriented networks would work optimally here. Broadcast networks like ethernet would gain little advantage from the read graph and the opportunity for parallelism is reduced. Connection oriented ATM networks which are becoming attractive should make a likely marriage with Sombrero.

5. ADDITIONAL TOPICS IN SUPPORT OF SOMBRERO

This is the catch-all Chapter for minor topics not discussed elsewhere. Each of the sections will introduce themselves.

5.1 Token Execution

Each of the three token types have been discussed in detail in previous Chapters including the semantics of invoking or executing a token. In general the intent is to treat tokens as classic OOP objects complete with methods.

Other standard methods besides token execution are to determine policy handlers for guiding R/W/X semantics, or how to handle a request to gain access to or relinquish control of a token or even to fission a token in the case of MOTs. These methods can be from a standard set of semantics provided by the kernel and selected by flags. A flag is also provided to indicate a pointer to a custom policy handler. Generally token semantics will have default settings which can be changed or a custom policy can be chosen. Specific semantics are discussed in a later Section.

5.2 Name Server Design

The Name Server provides the ability to locate tokens. If a node needs access to a particular memory object and it has only a symbolic name to go by the nameserver will translate it and provide an address. If the address of the object is known the network will route the request to the inception node for that memory object. If the inception node is missing, the nameserver will trap the error and broadcast looking for the object. The nameserver will provide a hierarchical ordering to the symbolic names similar to the UNIX name path. The only new aspect is that the directory for the entire virtual space/network is under a single root and nodes are not the partitioning boundaries. When a user logs in the user directory will consist of the objects attached

to the users UPD which have been given a symbolic name and registered with the nameserver. This means an object can potentially be from anywhere in the network and belong to more than one directory even under different names. The nameserver will register only one user as the owner and this owner will determine the access rights for the other owners.

The nameserver is an example of a UPD distributed over multiple nodes. In fact it is distributed over all of the nodes. Its capability list is the sum of the capabilities of all of its clones. Usually it will have a local working set with the occasional need to go outside. For example when the directory of names is searched for the entire network the local clone of the name server UPD can replicate a thread over the read graph to search all of the other clones in parallel.

5.3 Initial User Logon

When a new user is created and given a password a user UPD is also created. Some standard startup facilities are attached to the UPD and it is automatically persistent but static until the user logs on. The Logon server executes the user UPD when the user logs on, this has the effect of creating a thread and starting it at the designated entry point for the user's UPD. With the current display device attached to the thread a display server is called such as an x-window server, a simple prompt or Sombrero's own interface. In any case the user is ready to go.

5.4 Virtual Memory

Virtual memory on a Sombrero node encompasses the entire memory space but a local address map contains only the entries for the pages present on its node. It would be more efficient on a TLB miss to have a separate address map for each UPD

and thread but the page replacement algorithm would then be slower and more complex.

The virtual memory no longer participates in the protection scheme but it still needs to maintain a dirty bit for memory writes. It would also be useful in coordination with token tracking to have a notify dirty bit. When this bit is set a write to a page initiates the invalidate page notice for pages belonging to distributed memory objects.

The virtual store on disk has the same address map as the map stored in main memory and acts as the backing store for the address map. It must be sanitized on system startup to remove any stale frame references. The first page on the disk holds a mini virtual map listing the disk blocks which hold the main virtual map.

This Section also seems the appropriate place to discuss the mapping of memory objects to the virtual paged memory. Conventionally object resolution is limited to the granularity that the paging system offers. This is because the paging system also manages protection. To go below the page level of protection requires a whole new protection structure redundant to the paging mechanism. Certainly a costly effort better left to applications. Sombrero's protection mechanism has no relationship to the paging mechanism other than the need to map its memory objects to the virtual pages, a fairly simple lower level mechanism which in the climate of faster computers and cheaper memory is not expensive. The boundary which determines the fineness of protection granularity depends on the application. Numerous objects accessed frequently and locally with similar access push the efficiency boundary to larger MOT granularity with finer access managed by the application. Objects which are distributed or few in number or which need to be protected differently tend to make the case for the use of fine grained management at the OS level as being efficient.

5.5 Multiprocessors in Sombrero

Multiprocessing in Sombrero introduces the same kind of difficulties as in other systems. All of the caches including the memory cache, the TLB and RPLB can become stale in one processor when another processor performs an update. Similar solutions as in other systems such as snooping and cache line invalidating should work as well for UPD-III hardware including the RPLB. It would be advantageous to invalidate the RPLB by UPD or thread number.

5.6 Device Drivers and I/O Ports

Device drivers in Sombrero are separate UPDs which have the access rights and programs (i.e. methods) to manage devices such as printers, video display memory and storage devices. Access to display memory or I/O ports can be shared with other UPDs when warranted.

Memory mapped I/O ports and display memories are accessible via MOTs just as any other memory is. Since I/O devices are at physical addresses rather than virtual addresses the address mapper must take care to assign the page and frame to have the same address, in essence no translation. The I/O ports may likely occupy the same address on all nodes thus the MOTs for similar I/O ports are replicated to nodes as required and with independent read/write semantics (i.e. no consistency enforcement since they change out of synch). The drivers and ports are paired for each node on system start-up.

5.7 An Analysis of UPD Switching and Lexical Support

The case for UPD switching would not be complete without a discussion of its implementation in Sombrero. There has already been discussion about the ubiquitous nature of the protection mechanism and its ability via the RPLB to act in parallel with program execution rather than having a serial component. Now there needs to be a discussion of its three forms as well as the lexical methods used to obtain optimal performance.

The actual switch takes three forms: 1) The domain switch presented in Section 1.2 page 6 from the protection matrix described in Lampson's paper, 2) its partial inverse and 3) an explicit switch. It is convenient to name the first Lampson's switch, its partial inverse the UPD switch and the 3rd the serial switch.

Lampson's switch has three requirements, 1) you need to know the currently active PD, 2) you need to know the intended PD and 3) an entry permission (which leads one way or another to the address of the next instruction in the new PD) at the intersection of the two PDs permits the domain change. The UPD switch by contrast requires a knowledge of the current PD and an entry point address in the intended PD. A permission (the new UPD number) at the intersection of the active PD and address permits the domain switch. The difference between the two strategies is that for the UPD switch the address or entry point is a participant in the RPLB tag and the new UPD number is provided on a hit. The serial switch specifies all three components and is implemented serially as a system call.

LAMP	PD1	PD2	PD3
PD1			04560h
PD2	0ABC0h		
PD3			

UPD	0ABC0h	01230h	04560h
PD1			PD3
PD2	PD1		
PD3			

Figure 5-1 Comparing Lampson's switch to the UPD switch

The paradigm is that with Lampson's switch the PDs are the objects which are called and provide services. With the UPD switch the memory objects are called and provide services and are grouped into protection domains to prevent illegal or accidental access. Another way to look at it is that Lampson's switches are an explicit PD switch with an implied entry point and UPD switches are implied PD switches with an explicit entry point. The third form, the serial switch is not ubiquitous. Which concept is better? They all have their uses but focusing on the memory objects via the UPD switch seems the more intuitive approach and is well supported by the SASOS environment.

Since the UPD name is also an actual address, for Lampson's switch the UPD name appears both as the switch address and as the new UPD name in the RPLB (see Figure 1-8). This is the only occasion for using the Lampson switch in Sombrero and provides an example of UPD token execution.

If Lampson's switch and the UPD switch are treated as functions (in the mathematical sense) then for every entry in the domain of the function there can only be one entry in the range. If the function for the UPD switch is:

New_UPD = Current_UPD(Entry_point)

then given UPDs A and B, A can have many entry points in B.

B = A(XXX), B = A(YYY), B = A(ZZZ).....

where XXX, YYY and ZZZ etc. are addresses for different entry points.

If the function for Lampson's switch is :

Entry_point = Current_UPD(New_UPD)

then given UPDs A,B,C,D.... and entry point XXX, XXX can serve as entry point for many UPDs:

XXX = A(B), XXX = A(C), XXX = A(D).....

Given the nature of SASOS and Sombrero's use of the UPD name for both address and name, Lampson's switch will in practice only ever have one UPD for each entry point.

With the above discussion in mind Sombrero must insure that for any given protection domain there must be one and only one outcome for any given address, otherwise the RPLB will make an unpredictable choice. It must also be kept in mind that in Sombrero the actual current protection domain (assuming the ATO flag was set for the thread) is the union of the current UPD and the domain of the currently executing thread. Several interesting anomalies can appear as a result:

If two domains have an intersection (shared memory and access rights) a switch with the same access rights cannot take place inside the intersection. In support of the RPLB the kernel only accepts one hit at any given address for each UPD. EX: For A to switch to B with execute permission the switch must point to an instruction at a legal address XXXX in B. This means a switch capability must exist in A for XXXX. If A shares text XXXX with B then A must have two capabilities for XXXX, a normal range mask and a switch, both with execute privilege. The RPLB cannot hit on both capabilities and it will choose randomly between them so Sombrero will not allow both capabilities to be created in the first place. Mathematically A cannot have two values in

the range for the domain entry XXXX. To perform such a switch requires the use of the serial switch.

A similar domain/multiple range problem occurs when multiple switches exist for the same address but different target UPDs. The kernel prohibits most conflicting entries.

The kernel cannot prohibit the case of a thread entering a UPD where the union of the two domains results in some ambiguous conditions. This problem can be helped by giving preference to one domain over the other (the order of search on a RPLB miss) or by clearing the ATO flag. The problem also becomes mute if the intersecting domain entries are identical. If the intersection is not identical and the ATO flag is set and one or both of the entries is already cached in the RPLB the results are unpredictable.

Now the lexical discussion.

The UPD strategy is based on a change in protection which works in one direction only i.e. no returns. This is equivalent to a programming language without subroutine call support. The features of a language built around goto instead of call,

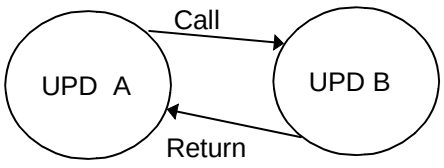


Figure 5-2 Simple Sombrero return from call

while, for etc. are cumbersome and have been largely discredited. Yet all computers operating at the lowest level are merely goto machines in the final analysis. The use of

constructs like call, while for etc. are lexical aids not fundamental features. UPD is an extension of the goto machine. It has no inherent return feature and like a computer's instruction set requires lexical support to make it elegant.

Sombrero provides a rudimentary return capability for its UPD switch feature. The basic rule is this: if a thread can exit UPD A to enter UPD B then it also has sufficient privilege to return from B to A as long as no domain conflicts are created.

The return rule comes into play when a thread attempts to travel to an entry point in another UPD when a switch permission has not yet been granted. The resulting fault is analyzed by the kernel using the target UPD's policy flags and if a switch can be granted it is. Furthermore if the analysis reveals that the attempted instruction was a subroutine call a return switch is automatically created to the intended return address.

Implementation of this rule has a pitfall: Once the switches are established from A to B returning to XXXX any other switches to the same entry point in B but to a different return address will have trouble. If the second call to B returns to YYYY instead of XXXX the switch already exists for the subroutine call but will fault on the attempted return. Unless A's UPD policy wants to start granting arbitrary entry points to B or cataloging all of the possible return addresses this will be a one way subroutine call. A solution is to have a compiler construct as a part of a function's declaration or prototype which causes the compiler to perform all such function calls through the same exit point so that only one return switch need exist, a gate declarator. This can be implemented on the machine by substituting the call instruction with a push or load register instruction which stores the entry point address followed by a jump to the exit

point which uses the stored address to perform the actual call. We will call this an exit gate.

There are two benefits to an exit gate: 1) The pitfall is eliminated, 2) The tendency for the RPLB to thrash is reduced by the reduction in the size of the RPLB working set.

A complimentary solution to the exit point problem is to consolidate the entry points in a target program, an entry gate. The entry gate would be the flip side of the gate declarator where the compiler would resolve all external references for entry points to the same address. It would then treat the first parameter in the argument list or a special register in the traveling thread as an address to the actual entry point or an index to a table of pointers to the actual entry point. The benefit would be the same as before, i.e. the reduction of the working set in the RPLB.

Optimal performance comes when the working set stays fairly static in the RPLB as with any caching mechanism. When the RPLB working set is stable the protection validation and switching is completely parallel and ubiquitous.

An additional lexical problem is that of resolving links to programs via symbolic names based on library utilities. It would make sense for compiled program objects to store their externally declared symbols permanently and make this list available to the name server or a library version of the name server. With such a utility new programs on startup would permanently resolve all of the missing links or resolve them as needed (lazy evaluation). This access can also provide an avenue to notify programs of updated program modules so that they will relink when opportunity permits.

It is expected that UPDs will be used to some degree to group objects of like priority (such as library modules) rather than simply to encase individual objects in their

working segments, even then it is expected that application programs be composed of separately compiled objects which may not be fully resolved until the first execution time similar to Windows NT and its DLL libraries. Since multiple objects might exist in a single UPD it makes sense that separately compiled objects will have separate gates even when sharing a UPD. UPD permits multiple entry points so multiple gates should pose no problem.

Another strategy is that every UPD is created with its own standard gate and a thread register is set aside to point to it. In this way there will only need to be one gate per UPD. UPD-I would work best in this environment where entry gates exist only at the start of a block and UPD switches form a hit only at block address 0. Multiprocessing is relatively easy in this environment since there is only one protection manager and consistency is not then an issue. There are all kinds of ideas to kick around. Only experience will settle the issue.

5.8 Comparison to Conventional MASOS

Sombrero as a SASOS has some advantage over today's conventional operating systems.

- Address resolution is much simpler.
- Pointers remain valid in all contexts.
- Shared memory is a simple matter of providing a pointer and granting access.
- Object persistence is simple to implement.
- Node routing has a fixed and simple method based on the root MOT.

- The protection and translation hardware such as the RPLB and TLB provide a bridge between the kernel structures and the user space. The kernel can ubiquitously manage the user space by enforcing protection largely in parallel with user program execution.
- The kernel structures are readily extensible to the network.
- Object consistency is managed across the network without version numbers and access becomes automatically decentralized.
- The name server provides object transparency.
- Network token transfers and the read graph provide location independence.
- In general the Sombrero kernel provides a new level of abstraction independent of the underlying hardware which gives the network the appearance of a single large multiprocessor computer.

No doubt there are other advantages. The drawbacks to Sombrero include:

- Compatibility with current OSs.
- New hardware is required to implement Sombrero.
- The kernel structure is exposed on the network and may render Sombrero less secure unless some scheme like password capabilities is used to validate internode transactions. Even then the security is based on a probability not a guarantee (Anderson et al. 1986).

6. VALUE OF RESEARCH

Sombrero offers a setting to present some new ideas which though untried are worthy of further study. These ideas include:

- The exploration of various methods to provide ubiquitous memory protection: UPD-I, II and III. These protection schemes provide a memory object oriented approach to grouping program and data resources. They then provide a mechanism for interaction which requires no structure outside the kernel and retains the abstraction of a single very large freely interacting computer. See Chapter 1 page 4.
- The extension of Koldinger's PLB to use the Range Mask with its highly efficient and versatile representation of memory as well as PD switch management abilities (Koldinger et al. 1992). The RPLB is viewed as a device which bridges the barrier between the kernel and the user space by providing access to the kernel structures in user mode. See Section 1.5.2 page 22.
- The use of tokens for memory management. The entire address space is treated like a recursive heap space with tokens to represent the partitioning. Manipulation of the tokens then controls access to memory. See Chapter 2 page 36.
- The token tracking graphs which though simple provide a medium for memory consistency, independence and transparency. See Section 4.1 page 50.
- The use of tokens to represent other kernel resources such as protection domains and threads. Manipulation of these tokens makes the kernel extensible on the network. See Section 4.2 page 54.
- The use of tokens to create a highly object oriented presentation of resources. See Section 5.1 page 59.

- The use of tokens to create an abstraction level above the level of virtual paging so that the page size is no longer a barrier to object protection and granularity. See Section 2.3 page 40.

Additional summary and contribution detail is available in sections 1.3.5 and 1.8.

◆

This thesis is intended to provide a perspective for exploring the advantages of a VLSASDOS environment. The issue of name management is perhaps the greatest issue and is greatly reduced in a SASOS since any abstract object with a memory address already has a permanent and unique name. This leads to many kinds of simplifications and new abstractions. The most significant being the unifying effect of having a single large address space in which every abstract object (threads, UPDs, MOTs) can exist independently of the hardware. The implications for location independence, transparency, replication, communication, RPC, object persistence etc. are enormous compared to the barriers to these which must exist in a many name space environment. The value of this research is in casting further light on this subject.

The goal of this design study, which admittedly is incomplete, is to provide a starting point and inspiration to researchers for an actual implementation of a VLSASDOS.

REFERENCES

Alpha Microsystems. AMOS Monitor Calls. Santa Ana, CA., April 1991.

Anderson, M., R. D. Pose, & C. S. Wallace. “A Password- Capability System”, The Computer Journal 29, NO 1 (1986):1-8.

Carter, John B., A. L. Cox, D. B. Johnson, & W. Zwaenepoel. “Distributed Operating Systems Based on a Protected Global Virtual Address Space.” Proceedings of the Third Workshop on Workstation Operating Systems, April 1992:75+

Chase, Jeffrey S., H. M. Levy, M. J. Freeley, & E. D. Lazowska. Sharing and Protection in a Single Address Space Operating System. Department of Computer Science and Engineering, FR-35, University of Washington, Seattle, WA 98195: Technical Report 93-04-02, Rev. January 1994.

Heiser, Gernot, K. Elphinstone, S. Russell, & J. Vochteloo. Mungi: A Distributed Single Address-Space Operating System. School of Computer Science and Engineering, The University of New South Wales: SCS&E Report 9314, November 1993.
Available by anonymous ftp @vast.unsw.edu.au:pub/Mungi/9314.ps.

Heiser, Gernot. The Mungi Manifesto. 1994. Available by anonymous ftp @vast.unsw.edu.au:pub/Mungi/Mungi-Manifesto.

Koldinger, Eric J., J. S. Chase, & S. J. Eggers. Architectural Support for Single Address Space Operating Systems. Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195: Technical Report 92-03-10, Rev. July 1992.

Lampson, B. W., “Dynamic Protection Structures.” Proceedings of the AFIPS Fall Joint Computer Confrence. 1969:27-38.

Murray, Kevin, T. Wilkinson, P. Osmon, A. Saulsbury, T. Stiemerling, & Paul Kelly. “Design and Implementation of an Object-Oriented 64-bit Single Address Space Microkernel.” Proceedings of USENIX Symposium on Microkernels and Other Kernel Architectures, September 1993:31+.

Okamoto, Toshio, H. Segawa, S. H. Shin, K. Maeda, H. Nozue, & M. Saito. “A Micro Kernel Architecture for Next Generation Processors.” Proceedings of USENIX Workshop on Microkernels and Other Kernel Architecture, April 1992:83+.

Wilkes, John, B. Sears. “A comparison of Protection Lookaside Buffers and the PA-RISC protection architecture.” HP Laboratories Technical Report, HPL-92-55, March 1992.

APPENDIX A

GENERATING RANGE MASKS USING A FROM-TO RANGE

The RPLB is designed to operate using the Range Function. Some memory objects are described more compactly using a simple from-to range when the alignment does not favor a single range mask.

The range mask appropriate to a hit in a from-to range is derived using the following algorithm expressed in C:

```
void rangeToRangeMask(addr *matchMask, addr *dontCareMask,
                  addr rangeFrom, addr rangeTo, addr address)
{     addr part1;
      addr part2;
      boolean done = FALSE;

      part1 = rangeFrom;
      part2 = (part1-1) | part1;
      while(part2 <= rangeTo && !done)       // work up to larger blocks
            if (address >= part1 && address <= part2)
            {     *matchMask = part1;
                  *dontCareMask = part1 ^ part2;
                  done = TRUE;
            }
            else
            {     part1 = part2+1;
                  part2 = (part1-1) | part1;
            }

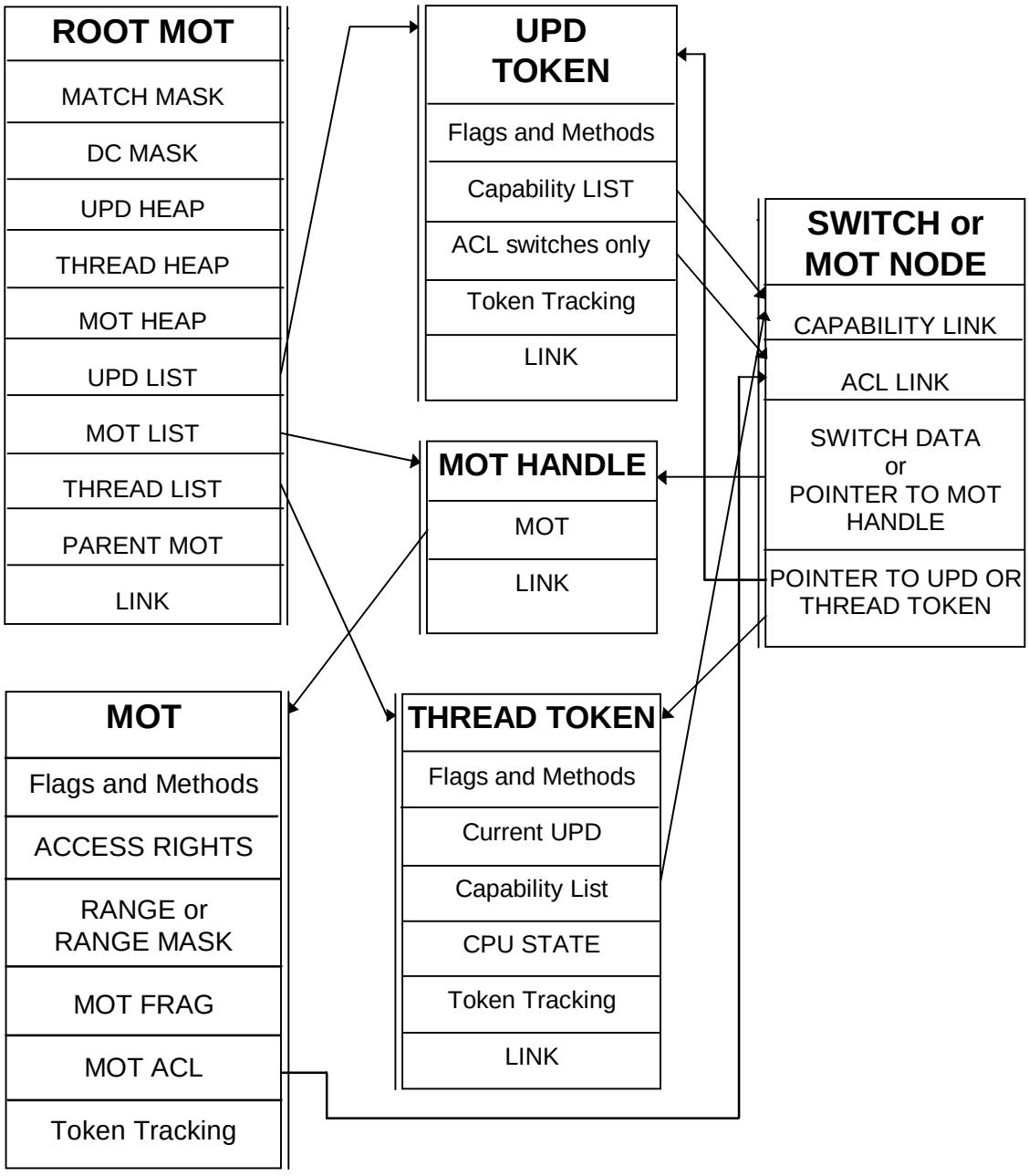
      part1 = rangeTo;
      part2 = (part1+1) & part1;
      while(part2 >= rangeFrom && !done)     // work down to larger blocks
            if (address >= part2 && address <= part1)
            {     *matchMask = part2;
                  *dontCareMask = part1 ^ part2;
                  done = TRUE;
            }
            else
            {     part1 = part2-1;
                  part2 = (part1+1) & part1;
            }
}
```

Figure A-1 Range Mask program

The number of iterations depends on the alignment, the size of the range and the address and is at most equal to the number of one bits in the largest of the Don’t Care Masks. Pointer casting and the use of unsigned numbers were left out to make the program more readable.

APPENDIX B

KERNEL CONTROL STRUCTURES



Data structures in the Sombrero kernel used to manage the user space. They are extensible supporting distributed object management.

Figure B-1 Kernel control structures

The kernel data structures in Figure B-1 describe the relationship between the three token classes beginning at the Root MOT which contains the memory allocation for the local node. The tokens are extensible via the token tracking described in Section 4.1 on page 50 and Section 4.2 on page 54.

Flags and methods select access policy, network consistency semantics and point to the method text for each token. See Section 5.1 page 59 for additional discussion.

Notice the existence of both capability lists and Access control lists. Since all of the objects are in the kernel under one protection domain there is little cost to linking access permissions both ways. It is not clear whether searching MOT ACLs or capability lists will produce the better performance. To locate a MOT by the MOT list the ACL would then provide the access permissions for that MOT. To locate a MOT by the current UPD or thread the capability list would provide the access permission to a given MOT.

In Mungi access across the network is better served using capabilities since users need to be authenticated anyway. In Sombrero the ACLs are extended across the network and could still be used but using capabilities requires less traffic. ACLs may be more efficient locally.

Another idea from Psyche is to cluster permissions so that they may be attached to existing capabilities instead of inventing new keys. Perhaps a solution for Sombrero is to have a separate authentication system, kernel to kernel, as a carrier for the underlying traffic. Since local kernels are individually secure the only necessary password validation may be to validate the right to access other kernels in a group. By

this means one capability password may serve to validate a group of extended kernels to form a secure super kernel.

APPENDIX C

SOMBRERO BLOCK STRUCTURE

The block structure of the Sombrero OS showing the token layer of abstraction.

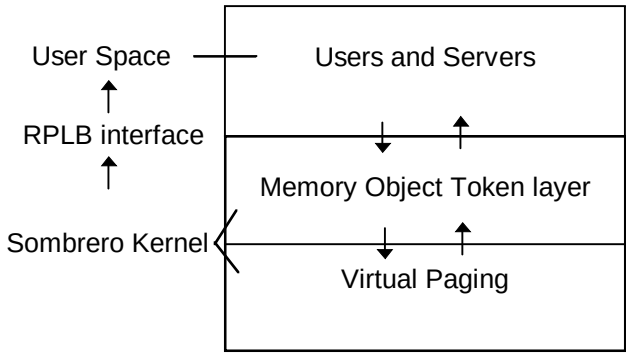


Figure C-1 Sombrero block structure.

In user mode the User Space is regulated invisibly by the structures in the kernel via the RPLB. The token layer describes the virtual space, the resulting tapestry is mapped onto physical space by the virtual paging system via the external TLB. It might also be efficient to allow user space to have read but not write access to the structures that regulate its protection.

APPENDIX D

PA-RISC and RPLB

This material is a discussion of the relative merits of the PA-RISC architecture as presented in the Wilkes and Sears paper and the RPLB presented in this thesis (Wilkes & Sears, 1992). It assumes a familiarity with the aforementioned paper.

The Koldinger PLB comparison to PA-RISC is the object of the Wilkes paper. Comments will be made on that as well as the effect of the enhancements in RPLB.

Wilkes establishes the need for a SASOS environment and the need of virtually addressed caches. He then refutes the use of the virtually addressed caches assumed by Koldinger's PLB because of the current limitations in silicon and does some additional bit straining for the same reason. It is likely that silicon will become available, historically it has. The RPLB is wider than the PLB but not a lot more.

The RPLB consumes more silicon in width but it is far more efficient in depth because of the Range Mask. It also takes advantage of the union of multiple protection domains to form a working PD from a UPD and a thread PD. RPLB has at least two PD registers, PA-RISC has four and proposes the need for more.

PA-RISC works as a page based ACL with a single entry ACL for each page. Koldinger's PLB works as a page based capability list with multiple entries for shared pages. RPLB finds a balance between both by doing both and is object based rather than page based. In addition RPLB buffers domain switches.

Another discussion in the paper revolves about the idea of moving the TLB off board the CPU. Koldinger asserts that it is no longer needed on board and an off board TLB could be larger. He also asserts that this will save silicon since the PLB is a smaller version of the TLB sans the stored physical address. Wilkes says its not worth the savings and an off board TLB is not accessible enough since physical addresses will be needed on board the CPU.

The RPLB provides additional incentive to move the TLB off board since it is object based and the TLB is page based. In addition by Wilkes own opening argument a virtually addressed cache is the way to go so there is no need to have access to the physical address. Just send data with its virtual address to the TLB and let the TLB worry about the physical address and bus control. To support the pipeline and off board propagation delay the virtual address pins can be driven early in the pipeline as if the TLB were on board. A staging mechanism can match pipelined translations with actual bus events on primary cache misses. The TLB can stall the pipe on TLB misses when the staging mechanism actually needs an address or fault the CPU on missing pages. Access to the TLB in this scenario is an output requiring no response from the TLB except in the event of an actual bus utilization in which case everything stalls to wait on the bus anyway.

It seems in the paper that Wilkes arguments devolve around the current limitations of the PA-RISC design and not general principle. He has the CPU fetching an address from the TLB and then sending it out on the bus, of course that would be slow.

It also seems reasonable that there would be some advantage to an off board TLB in a multiprocessor environment since updates to the TLB would need to only happen once for all the processors. It seems that the real reason for having the TLB on board the CPU is to validate access to the memory in the primary cache. In Sombrero's hardware environment translation and protection are separate so let physical translation be part of the function of the physical bus interface and protection will remain on board the CPU.

INDEX

A

abstraction
1, 2, 6, 15, 19, 20, 27, 37, 44, 48, 49, 70, 71, 72, 81
ACL....8, 32, 78, 83
Address range....13, 35, 37, 38
algorithm....11, 16, 25, 57, 61, 75
alignment....26, 27, 75
Allocation....vi, 55
Alpha Micro....vi, 5, 43, 73
ambiguous....46, 66
AMOS....vi, 5, 6, 18, 43, 73
Angel....33
ATM....58
ATO....22, 28, 47, 53, 65, 66

B

broadcast....48, 50, 53, 56, 59

C

cache
viii, 9, 11, 13, 15, 16, 17, 20, 21, 22, 29, 34, 50, 51, 52, 53, 54, 62, 84
Capability List
7, 12, 15, 19, 28, 40, 43, 54, 60, 78, 83
clone....55, 60
communication....5, 72
Complex MOT....vi, 26, 38
consistancy....78
consistency
iii, 37, 50, 51, 57, 58, 62, 69, 70, 71, 78

D

dirty bit....52, 53, 61
DLL....69
DRAM....18
Drivers....vii, 62

E

Entry Point
12, 22, 31, 32, 33, 60, 63, 64, 65, 67, 68, 69
ethernet....58
Exit Point....67, 68
Extensibility....vi, 54
Extensible....50, 54, 70, 71, 78

F

fission....56, 59
Frame....13, 14, 16, 61, 62

G

Garbage Collection....39, 40, 55, 57
gate....32, 67, 68, 69
GoTo....66
granularity
16, 21, 25, 27, 29, 33, 35, 41, 61, 72
graph
viii, 25, 50, 51, 52, 53, 54, 55, 58, 60, 70

H

hardware
viii, 1, 2, 4, 5, 9, 10, 13, 16, 17, 19, 20, 23, 25, 27, 32, 33, 62, 70, 72, 84

I

I/O....vii, 13, 38, 62
independence....70, 71, 72

K

Koldinger....9, 21, 22, 29, 34, 71, 73, 83

L

Lampson....viii, 63, 64, 65, 73
Lazy Evaluation....25, 26, 51, 57, 68
Lexical....vii, 63
LPRC....33
LRU....11

M

MACH....1, 21, 31, 37, 43
MASOS...vii, 1, 3, 15, 21, 31, 35, 43, 69
Matrix....viii, 7, 8, 9, 10, 11, 12, 13, 63
Memory
1, 4, 6, 7, 8, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 50, 51, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 64, 72

Memory Object
7, 10, 16, 21, 22, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 36, 37, 38, 39, 40, 41, 42, 44, 45, 46, 47, 48, 50, 51, 53, 55, 56, 57, 58, 59, 61, 64, 71, 75
MOT
vi, 7, 26, 29, 36, 37, 38, 39, 40, 41, 42, 43, 50, 52, 53, 54, 56, 57, 58, 59, 61, 62, 69, 72, 78
Multics....32
Multiprocessor....vii, 62
Mungi....33, 55, 73, 78

N

Network.... vi, 58, 70

O

Opal....31, 32, 40

P

Page....vi, vii, viii, 52, 53
PDM
vi, viii, 9, 10, 11, 12, 13, 14, 15, 16, 19
PDMS.viii, 10, 11, 12, 13, 14, 16, 17, 18
persistence....iii, 1, 4, 40, 69, 72
PLB
vi, 9, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 32, 34, 35, 36, 38, 41, 43, 46, 47, 49, 53, 62, 63, 64, 65, 66, 68, 70, 71, 73, 75, 81, 83
Ports....vii, 62
POSIX....33
Process....15

R

Range Mask
viii, 23, 24, 26, 30, 38, 71, 74, 75, 83
register
13, 14, 18, 19, 34, 44, 60, 67, 68, 69
replication....54, 72
Root MOT....56, 57, 69
RPC....32, 48, 72
RPLB
vi, vii, viii, 9, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 32, 34, 35, 36, 38, 41, 43, 46, 47, 49, 53, 62, 63, 64, 65, 66, 68, 70, 71, 75, 81, 82, 83, 84

S

SASA....33
Semantics....57
serial....32, 33, 34, 43, 48, 63, 64, 66
Sharing....3, 5, 28, 44, 69
Sombrero
iii, vii, viii, 1, 2, 7, 20, 21, 27, 29, 37, 38, 39, 40, 41, 42, 43, 48, 54, 56, 57, 58, 60, 61, 62, 63, 64, 65, 66, 67, 69, 70, 71, 78, 80, 81, 84
Switching
4, 6, 7, 12, 15, 17, 19, 32, 33, 34, 47, 48, 63, 68

T

Thread
iii, 5, 6, 7, 8, 13, 20, 21, 22, 23, 28, 29, 30, 31, 33, 35, 39, 41, 43, 44, 45, 46, 47, 48, 54, 60, 61, 62, 65, 66, 67, 68, 69, 71, 72, 78, 83
Threads
22, 28, 30, 33, 35, 39, 43, 44, 45, 46, 47, 48, 54, 72
TLB
vi, 9, 19, 20, 22, 25, 29, 30, 34, 60, 62, 70, 81, 83, 84
Token
iii, vi, vii, 10, 22, 30, 31, 36, 38, 40, 41, 43, 50, 54, 55, 57, 59, 71, 72, 78
Token Execution....vii, 59
Token Tracking.... vi, 50
ToM....32, 33
transparency....70, 71, 72

U

ubiquitous 18, 19, 20, 35, 63, 64, 68, 71
unique....30, 35, 48, 54, 55, 57, 72
UNIX
1, 4, 5, 6, 19, 21, 31, 33, 37, 43, 47, 54, 57, 59

UPD
vi, vii, viii, 2, 6, 7, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 25, 28, 29, 30, 31, 32, 33, 34, 35, 36, 39, 40, 41, 43, 44, 45, 46, 47, 48, 50, 54, 55, 59, 60, 62, 63, 64, 65, 66, 67, 69, 71, 78, 83
User Mode 15, 17, 71, 81

V

Virtual Memory ..... vi, vii, 4, 21, 57, 60

W

Windows NT 21, 37, 42, 69
working Set 60, 68

© 1994. Alan C. Skousen.