

SOMBRERO: IMPLEMENTATION OF A SINGLE ADDRESS SPACE PARADIGM
FOR DISTRIBUTED COMPUTING EXHIBITING REDUCED COMPLEXITY

by

Alan C. Skousen

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

ARIZONA STATE UNIVERSITY

December 2002

SOMBRERO: IMPLEMENTATION OF A SINGLE ADDRESS SPACE PARADIGM
FOR DISTRIBUTED COMPUTING EXHIBITING REDUCED COMPLEXITY

by

Alan C. Skousen

has been approved

August 2002

APPROVED:

_____, Chair

Supervisory Committee

ACCEPTED:

Department Chair

Dean, Graduate College

ABSTRACT

Operating within the bounds of a single address space has, in general, always been the simplest model for computing. The demands of modern computing have introduced various complexities to overcome the limitations of available hardware and to extend communications between computers. These complexities include the use of virtual address spaces to alleviate address space limitations of the hardware, and to provide protection between computational entities on the same machine. Multiple virtual address spaces and interprocess communication have become the standard paradigm on which complex operating systems are built. This research explores an alternative to the standard paradigm: reverting to the single address space model including distributing it across the network. The outcome of this research, which is supported by several theses and culminates with this dissertation, indicates that a single address space paradigm can reduce the complexity of system and application software and may greatly reduce the cost to design, maintain, and execute distributed computer programs.

This dissertation in particular documents to date the effort to implement the Sombrero Single Address Space Distributed Operating System. A kernelless modular system design and a new protection mechanism that offers zero-cost domain switches have been successfully implemented.

Sombrero's single address space design is ideal for multiprocessor environments, real-time and embedded environments, distributed shared memory, and load balancing. It lays the foundation for future "hands on" research for distribution and fault tolerance in a single address space.

Dedicated to the ones I love. Particularly to my wife Susan and children Noah, Emily, Seth, Adam, and Amanda who have put up with me for the last several years.

Thanks Nancy for watching the little one for 12+ weeks, and Rebecca too.

ACKNOWLEDGMENTS

Thanks Don Miller for your advice and support. It's been a happy association for the past few years. I hope it continues. Thanks to the committee for their suggestions and encouragement. Thanks to the other members of the research group who have come and gone for their contributions.

TABLE OF CONTENTS

	Page
LIST OF FIGURES.....	x
CHAPTER	
1 INTRODUCTION AND RELATED WORK	1
2 BACKGROUND OF SOMBRERO RESEARCH.....	3
2.1 MASOS to SASOS Paradigm Shift.....	3
2.2 Complexity Reduction Metrics	4
2.3 Complexity Reduction Theory.....	4
2.4 RPLB Design	6
2.5 Protection Policy	7
2.6 Distributed Scheduling	8
2.7 System Interface	8
2.8 Distributed Algorithms and Network Consistency	9
2.9 Summary.....	9
3 DESIGN SCOPE AND STRATEGY	10
3.1 Early Efforts.....	10
3.2 Current Efforts	11
3.3 Eventual Scope.....	12
4 SOMBRERO SYSTEM ARCHITECTURE	13
4.1 Object Oriented System Modules	13
4.2 System Protection.....	15
4.3 Token Tracking	16

CHAPTER	Page
4.4 Pager	17
4.5 Persistence.....	18
4.6 Design Influence	20
5 MIDDLE LEVEL SOMBRERO ARCHITECTURE	21
5.1 Basic Definitions	21
5.2 Sombrero Programs and Instantiations.....	25
5.2.1 Project Class Header File	29
5.2.2 Entry Point Implementation.....	32
5.2.3 Exit Stub Implementation.....	35
5.2.4 Return Point	37
5.2.5 Constructor and Static Constructor Entry Point	37
5.2.6 Standard OID.....	38
5.3 Program Class Object Instantiation.....	38
5.3.1 Grouping Class Instantiations.....	39
5.4 Symbol Resolution	40
5.5 Interdomain Communication (IDC) and Protection.....	40
5.5.1 Tail Switch.....	41
5.5.2 RPLB Implementation.....	42
5.6 PAL Code Support	43
5.6.1 Faults and Interrupts.....	44
5.6.2 Wait Queue	45
5.6.3 Context Switch.....	46

CHAPTER	Page
5.6.4 New Registers	46
5.7 Sombrero System Modules	47
5.8 Sombrero Libraries	49
6 TOOLS FOR SOMBRERO IMPLEMENTATION	50
6.1 Hardware.....	50
6.2 Language.....	50
6.3 Compilers.....	51
6.4 Development and Host Computers.....	52
6.5 Host Custom Tools and Services	53
6.6 Sombrero Compiler Custom Tools and Services on Linux	54
6.7 Other Tools	55
6.8 Applying the Tools.....	55
6.8.1 Sombrero Module Compile	55
6.8.2 Boot Loader and PALCode Build.....	57
7 SOMBRERO BOOT SEQUENCE	58
7.1 Boot Construction Phase.....	59
8 CONTRIBUTIONS AND REMAINING WORK.....	61
8.1 Contributions.....	61
8.2 Remaining Work	65
8.2.1 User Interface, Access Policy, and Directories	65
8.2.2 Versioning System	66
8.2.3 Fault Tolerance	67

CHAPTER	Page
9 FUTURE USES FOR SOMBRERO	69
10 CONCLUSIONS	71
REFERENCES.....	72
APPENDIX A.....	74
A.1 THE SOMBRERO DISTRIBUTED SINGLE ADDRESS SPACE OPERATING SYSTEM PROJECT	75
A.2 OPERATING SYSTEM STRUCTURE AND PROCESSOR ARCHITECTURE FOR A LARGE DISTRIBUTED SINGLE ADDRESS SPACE.....	76
A.3 USING A DISTRIBUTED SINGLE ADDRESS SPACE OPERATING SYSTEM TO SUPPORT MODERN CLUSTER COMPUTING	77
A.4 USING A SINGLE ADDRESS SPACE OPERATING SYSTEM FOR DISTRIBUTED COMPUTING AND HIGH PERFORMANCE.....	78
A.5 THE SOMBRERO SINGLE ADDRESS SPACE OPERATING SYSTEM PROTOTYPE A TESTBED FOR EVALUATING DISTRIBUTED PERSISTENT SYSTEM CONCEPTS AND IMPLEMENTATION	79
A.6 REDUCTION OF SOFTWARE DEVELOPMENT COSTS UNDER THE SOMBRERO DISTRIBUTED SINGLE ADDRESS SPACE OPERATING SYSTEM.....	80

LIST OF FIGURES

Figure	Page
1 Sombrero Architecture showing some services and relationships to PALCode and Hardware. Arrows indicate direction of initiated access.	14
2 Global Table to which a GP (OID) points.	28
3 Class header and entry point example from SOSLinker.	31
4 Runtime depiction of a proxy method invocation.	35
5 Layout of Development and Host Computers.	52

1 INTRODUCTION AND RELATED WORK

Single address space operating systems (SASOS) have been researched in a number of venues since the multiple address space operating system (MASOS) paradigm became the common standard (IBM MVS, Windows, Unix, etc.). With the notable exception of the IBM iSeries (formerly AS/400) a SASOS using protected pointers [Soltis 1995], most complex operating systems in use today have a separate virtual address space for each program that starts. These usually run over a common kernel that provides protected services accessed by system traps. Most SASOS researchers have built their experimental operating systems on top of one of these MASOSs to take advantage of the services they provide so as not to reinvent the wheel; Opal over Mach [Chase 1995], Mungi more recently over L4 [Vochteloo 1998] and the early versions of Sombrero over NT [Skousen and Miller 1998c]. The SASOS protection strategy in these experimental operating systems usually takes the form of issuing capabilities that allow access to resources. For AS/400, the only commercially available SASOS, capabilities took the form of protected pointers implemented over special memory and disk hardware. For Opal and Mungi capabilities are done in software.

The approach eventually chosen with which to experiment in Sombrero was to build it directly on the processor in order to better take advantage of SASOS features. We have observed that the set of processor optimizations that support a SASOS is somewhat different from that of a general MASOS. In particular, the protection mechanism employed in Sombrero, the Region¹ Protection Lookaside Buffer (RPLB), enforces

¹ ¹ In earlier literature the RPLB was known as the Range Protection Lookaside buffer. The word *region* is more appropriate and will be used henceforth.

protection policy directly in the processor, but unlike the TLB, performs no translation between virtual and physical address spaces. In addition, the design chosen for Sombrero is to implement a set of independently protected system modules that provide services through protected entry points.

System services are obtained by making subroutine calls to protected system entry points. This largely obsoletes the need for a special protected mode architecture using traps for services. To implement this over a MASOS operating system would combine the weakness of both systems without being able to fully utilize the strengths.

The remainder of this document discusses the research and simulations that have already been done. It also discusses the basic design of Sombrero and the reasons behind it. Most importantly, it introduces in detail the features of the actual prototype implementation and their purpose. Finally, remaining work is described.

At this writing, many of the basic features of Sombrero have been implemented including the protection mechanism with costless domain switching, but not the distributed aspects of the operating system.

2 BACKGROUND OF SOMBRERO RESEARCH

Sombrero research has been progressing for some time supported by a number of theses and papers. This dissertation represents an actual implementation of some of Sombrero's proposed features. Much of the background that led to the implementation is documented in the following sections.

2.1 MASOS to SASOS Paradigm Shift

A paradigm has been defined as a “constellation of beliefs, values, techniques and so on shared by the members of a given community that implicitly define the legitimate problems and methods of a research field for succeeding generations of practitioners.” In this case a paradigm shift occurs when going from the multiple virtual address spaces of process-oriented systems to the single address space of SASOSs².

We call it a paradigm shift because it has been our experience that most people who are used to dealing with a MASOS have difficulty developing an intuitive sense that a SASOS has special properties that can be advantageous³. Are the advantages of a SASOS sufficient to encourage movement toward a general paradigm shift? We have as yet not answered that ourselves but there are some tantalizing outcomes to the research so far that encourage us to continue. In particular, it appears, at least in a test environment, that distributed programs written to run on Sombrero are far less complex and would therefore be less costly to design, implement, and maintain. The reduced complexity should also impact overall performance making hardware utilization more efficient.

² For an extended treatment of the paradigm shift see [Skousen and Miller 1998d]

³ See Chapter 4.5 for an argument that identifies a source of this difficulty.

2.2 Complexity Reduction Metrics

Feigen, [2001; Feigen et al. 2002] using a functioning database application designed to run on Windows NT, compared the complexity of several versions of the application designed for different degrees of distributed behavior. These were also compared with a Sombrero version of the application designed to provide the same level of services.

The resulting set of comparisons using established software engineering metrics indicate a programming effort reduction in Sombrero upwards of 80% for highly distributed programs. Software engineering metrics usually consider a 5% reduction significant. Bug prediction was reduced by upwards of 65% [Feigen 2001 p76]. Even on the same computer in a non-distributed environment the results were significant.

While at the time there was no Sombrero operating system upon which to run the comparative programs, the measurements were not of running programs but of the software source code and its complexity. We believe the Sombrero version of the programs to have been representative.

2.3 Complexity Reduction Theory

The level of complexity reduction in Sombrero comes from the common substrate provided by the distributed address space itself. With all system and application programs residing and executing in a common persistent name space (the VA space), all data and code for every purpose have a unique address. Every address is also uniquely associated with an access policy enforced directly by the CPU via its RPLB even in a distributed environment. This common ground enables the entire network of participating computers

to appear to be running a single multithreaded program on a large address space multiprocessor, while parts of the program representing different interests are protected from one another. All services including access to parts of the application running on other computers are available via a simple subroutine call. Data is accessible across the network using standard string functions or pointer indirections. This differs considerably from the RPC and other middleware such as DSM required to marshal and translate data from one name space to another for MASOS communication. Atkinson et al. [1983] estimated that 30% of code in a database program was dedicated to translation between various namespaces.

One cause for reduced complexity in Sombrero is attributable to the relationship of the SASOS paradigm used by Sombrero to the conventional MASOS paradigm. In some ways the SASOS paradigm, in particular the Sombrero implementation, is more generalized in its behavior. The MASOS paradigm expects to construct an address space each time it starts a program, and in turn destroy it when the program exits. To preserve its functionality, the program must store enough information on a storage device to reload the program and reconstruct it when it starts again. Additionally, because of the isolation caused by using separate address spaces, the process must always have a thread associated with it. In contrast, the Sombrero VA space is designed to persist across a system boot with its program instantiations intact reminiscent of standby mode used by many PCs. This reduces the requirement to serialize data onto a storage device that has a different namespace, with different organization, addressing and access methods and allows the user to expect the program instantiation to persist for the life of the data on the

common distributed namespace used by all levels of store. It should also be noted that the Sombrero program instantiations are cataloged in a directory, the way files and data are typically cataloged in a file system, using symbolic representations. Another reason for reduced complexity is that Sombrero program instantiations don't require an associated thread; rather, they are passive. Threads enter and leave the program doing their own work, as services are required, like subroutine calls⁴ eliminating the protocols (usually referred to as Inter Process Communication (IPC)) necessary for one thread to pass information to another. These two features, persistence and passive services, indicate a generalization of the Sombrero implementation over the common MASOS paradigm, and when employed, reduce the complexity of Sombrero SASOS programs over MASOS programs.

2.4 RPLB Design

Khatri [1997] and Torland [1997] completed work testing the feasibility of the RPLB. As mentioned, the RPLB is the protection hardware Sombrero assumes present in a processor in place of the common TLB to enforce protection policy. It is part of the overall Sombrero design. In a Sombrero system, the address translation supported by the TLB is moved from the CPU to the memory bus since Sombrero with its single name space has no VA synonyms or homonyms [Koldinger 1992]. That is, there is only one mapping of a memory location in lower levels of cache (synonym) and for each VA there is only one address translation (homonyms). In a MASOS caches can become incoherent

⁴ In Sombrero these services are called passive services. Services with permanent threads are known as active services. Active services require a communication protocol to hand off data to its permanent thread increasing the complexity of the program.

and the same VA can translate to a different physical address depending on the current process. These synonym and homonym issues forced processors designed to support a MASOS to support internal physical caches with on-board address translation in order to maintain desired levels of performance.

The RPLB design for Sombrero goes beyond the protection features of the TLB and the earlier proposed Protection Lookaside Buffer (PLB) [Koldinger 1992]. It incorporates the ability to manage protection for objects rather than a few select page sizes [Skousen 1994]. It also provides a switching ability that allows a thread of execution to change its protection context from one instruction to the next without an intervening trap allowing very efficient context switches. Khatri [1997] produced simulations of the RPLB indicating a reduced miss rate due to the fact that very large objects can be represented by a single entry in the RPLB. Torland [1997] completed work on a VLSI design of the RPLB indicating it was feasible to build it into a CPU. The emulation of an RPLB is a central part of the current Sombrero implementation.

2.5 Protection Policy

Protection policy utilized by the RPLB in the CPU on every memory access is made available to the RPLB after a miss. A miss occurs in the RPLB when an attempted access is not allowed by the current set of permissions stored in the RPLB. A miss handler then searches the Protection and Resource Access List (PRAL) for an access descriptor granting permission for the access. Higher levels of access policy are consulted on a miss in the PRAL until a fault can occur for an illegal access. The PRAL is a set of data structures that combine capability lists and access control lists for all domains and

resources. This is effectively a compressed protection matrix [Skousen 1994]. As mentioned earlier, most implementations of a SASOS use capabilities, which are basically entries from a protection domain's set of access rights that are handed out like keys, and enable anyone who possesses them to use them. Revoking capabilities is difficult and there can often be dangling capabilities. Sombrero approaches this problem by maintaining the association between the capabilities and the access control list (in the PRAL), considerably simplifying rights revocation. Khatri [1997] implemented a simulation of the PRAL and RPLB misses that support the reasonableness of the data structures and algorithms used to implement it.

2.6 Distributed Scheduling

Patil [1999] researched and simulated an algorithm to distribute processor loads transparently across a network allowing thread management to have a default load balancing behavior that utilizes available CPU cycles across the network. Thread contexts remain the same from one node to another in Sombrero so thread migration is a relatively simple and natural behavior.

2.7 System Interface

Carnes [1997] produced a system interface that Sombrero might use that provided a starting point for the design of the system modules. The interface demonstrated that Sombrero can provide all the functionality of the Solaris 2.4 interface.

2.8 Distributed Algorithms and Network Consistency

Olson [2002] simulated the proposed distributed algorithms [Skousen 1994], which are called Token Tracking, and extended the concept and management of consistency in a distributed single address space such as that provided by Sombrero. The token tracking algorithms allow pages to be located while only tracking the objects that incorporate the pages. This greatly reduces the number of discrete items that must be managed by token tracking. It is also a non-centralized service allowing the reduction of choke points in distributed data management.

2.9 Summary

Sombrero theses, publications and talks are available at <http://www.eas.asu.edu/~sasos> and synopsized in Appendix A. This dissertation concentrates on issues that have come up in the implementation of the Sombrero prototype.

3 DESIGN SCOPE AND STRATEGY

Knowing the features to incorporate in an operating system does not define its implementation. Knowing the outcome wanted for the memory model, protection mechanism, and system structure doesn't automatically reveal the implications for the compiler, user interface, and a myriad of other issues. It is necessary to build an actual implementation to prove the concept and its feasibility. The design for the implementation combines top down and bottom up techniques allowing the available hardware to influence the design decisions since resources are limited. For example, the RPLB is emulated in Alpha processor PAL Code using its TLB since the means to build an actual RPLB are unavailable to us.

3.1 Early Efforts

The initial effort was to build a Sombrero implementation in an NT process by providing a pager at the user level to expand memory access to the 43-bit address space provided by the Alpha 21164-based NT target system. This was done to take advantage of NT device drivers and the Windows user interface while still having access to a 64-bit processor address space. There was also success in building and running Sombrero programs in the expanded address space. It became apparent however, that as the project grew to include more of the needed features, such as the RPLB, a scheduler, memory management, etc., that a great deal of the effort was dedicated to coexisting with NT, and that issue itself was greatly influencing the design decisions. The experimental issues were likely to be obscured by this course.

3.2 Current Efforts

From January 2001 to date, the implementation has been done without NT as a substrate on the Alpha 21164PC processor using a 164sx motherboard. AlphaBios 5.7 is the boot loader provided with the system. AlphaBios initializes the hardware and then loads the Sombrero boot loader. Thereafter, hardware initialization is modified as needed from that of the AlphaBios initialization to suit Sombrero, by the Sombrero boot program and subsequently loaded Sombrero modules.

The entire implementation process has an analog to the growth process of a biological system that develops from one cell, and eventually differentiates into specialized structures like organs until the creature is complete. In the Sombrero case the development began with what eventually became the Sombrero boot loader (the cell analog). Next, sophistication was added that allowed the Boot Loader to load modules from a host computer through a network interface card (NIC) and support a remote debugging interface to itself to aid in module development. As the modules matured and became more functional, the debug interface and NIC driver were eventually moved to their own module allowing the boot loader to be discarded after initialization. Each feature was added as the running portion became capable of supporting them. This was a very difficult task. Each small step forward exposed multiple unresolved issues that required debugging at the same time with limited tools. This debugging included the tools themselves, the network connection, the compiler, and the modules. Such low level interconnected debugging took the bulk of the development time. Currently the list of available system service modules includes the scheduler, memory allocator, protection

domain allocator, thread allocator, debug module, NIC driver, access (PRAL) module, run time linker, program loader, an interrupt service, pager, and a user service module. In addition, the PAL features of the Alpha processor emulate the RPLB and lock hardware access to service modules by mapping privileged PAL vectors only to those modules.

3.3 Eventual Scope

More differentiation will occur to system modules to sustain additional services demanding the development of new techniques. New modules will be added and expanded as the Sombrero development process continues. Future work, based on this prototype, will be discussed later in its own chapter. The eventual goal is to have a stable research platform that involves multiple computers with a persistent store and a good user interface.

4 SOMBRERO SYSTEM ARCHITECTURE

Sombrero system architecture is designed around the concept of a set of independently protected service modules that provide their services through entry points. Unlike legacy systems, there are no privilege levels, e.g., kernel mode and user mode. However, Sombrero does make use of PALCode executing almost entirely in the physical address space to provide some services. This affords a gray area with regard to the kernelless design in which not only is the RPLB emulated but also context switches, interrupt vectoring, wait queue management, CPU resource locking of privileged PAL vectors to system modules, and other register like services. (See Figure 1)

4.1 Object Oriented System Modules

Abstractly, the system modules appear as classic object oriented entities with methods and an encapsulated protected content. Access to the methods is determined by access policy and can be used to create access hierarchies implementing object oriented inheritance features. Modern kernel based operating systems are also typically hierarchical internally (NT), or even monolithic (UNIX), but have a flat defined API from the user point of view that is implemented directly or indirectly as a system trap. This limits user programs to services exposed by the API, which is a desirable but inflexible feature.

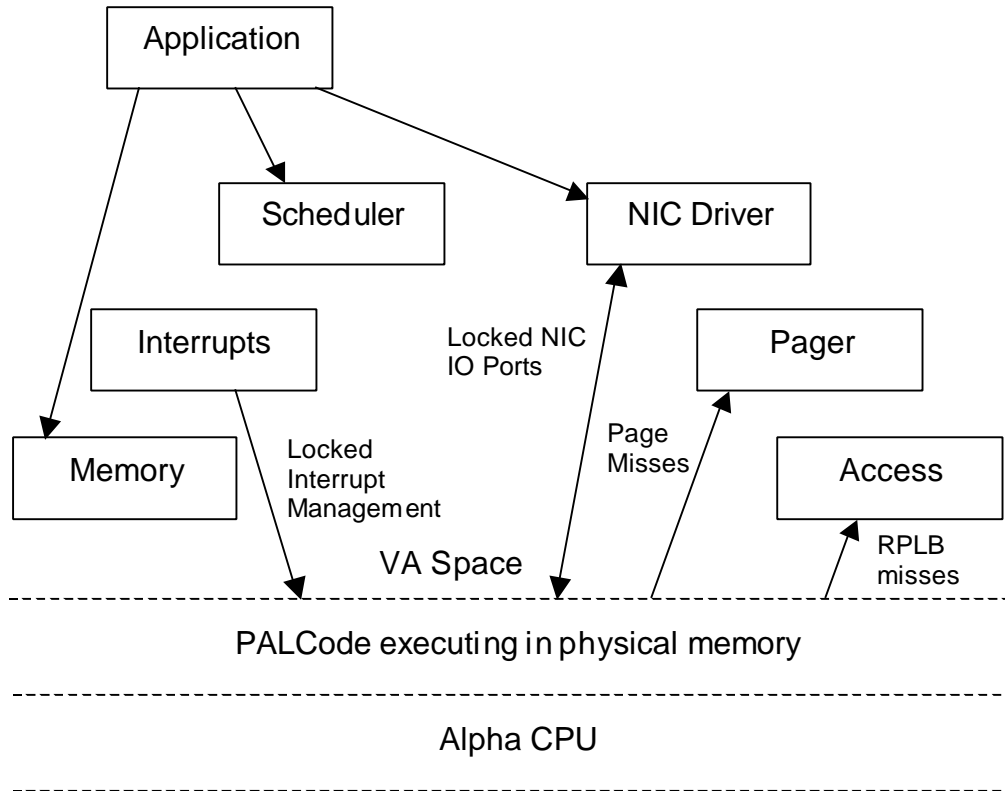


Figure 1 Sombrero Architecture showing some services and relationships to PALCode and Hardware. Arrows indicate direction of initiated access.

The Sombrero design is more flexible, for example, for real time requirements since base class entry points can become available through access policy changes. A real time system or a driver might need, for example, to bypass system accounting features to allocate a block of memory in order to meet time constraints if access time is deemed more important than accounting. In a kernel based system, the real time requirement would either have to be built into the kernel (loadable module), exposing the integrity of the kernel when new code is introduced, or a new system trap would have to be supported by the vendor, requiring a system release, perhaps undesirably exposing lower level access to user access in non-real time systems. In Sombrero, the administrator who manages system policy can expose the memory allocator by creating a permission to

access the low-level memory allocation entry point. No system code modifications are required. Perhaps more importantly however, is the ability of designers to build enforceable protected hierarchies using the system modules as base classes. By this means, system presentations for special users (such as a Windows or UNIX API) can readily be built with completely protected hierarchies. Chapter Five describes how the system classes and objects are implemented and how entry points work.

4.2 System Protection

Protection in Sombrero is designed to be functionally transparent and ubiquitous. Transparent because protected module entry points are entered implicitly rather than requiring an explicit protection domain switch, and ubiquitous because *every* memory access is checked against the system PRAL via the RPLB to insure that the correct access policy is enforced. Chapter One mentioned that most SASOS designs rely on capabilities. Software implementations of capabilities require explicit application of the capability in order to perform a protection domain switch. Sombrero records access policy, including protection domain switches, in the PRAL and at a lower level, in the RPLB, where the CPU has access to it. This allows the CPU to initiate a domain switch when it detects the need and permission is available. It is also costless since it is done in parallel with program execution except on an RPLB miss. This strategy also contributes to reduction in program complexity for Sombrero⁵. The theory and logical design of the RPLB can be found in [Skousen and Miller 1996a] and [Skousen 1994].

⁵ It should be noted that capabilities require the addition of at least one more namespace, the management of which increases program complexity.

Data structures that support the protection mechanism includes the General Protection Domain (GPD) control block, the Carrier Protection Domain (CPD a.k.a. thread protection domain) control block, Memory Object (MO) control block, and Accesses Descriptors (ADs). The control blocks, that provide information to the system about protection domains and memory objects, are all interconnected by the Access Descriptors. This interconnection constitutes the PRAL.

A protection domain is the set of all code and data that is reachable by a thread of execution in its current context as well as the set of protection domain switches available to the thread. Protection domains in Sombrero are actually the union of a GPD and a CPD. The GPD is the same as the protection domain definition, but can be exited by the thread for another GPD. The CPD is data and allowed switches, but not code that remain associated with the thread when it switches to a new GPD. By this means, access to memory such as the thread stack can be shuttled between GPDs without needing to have an AD for every occurrence. For this and other advantages of the CPD, see [Skousen and Miller 1996a; Skousen 1994; Khatri 1997].

4.3 Token Tracking

The proposed set of algorithms collectively labeled Token Tracking manage a set of graphs that are used to locate updated pages belonging to memory objects and maintain data consistency over the network. Token tracking also effectively distributes the access policy stored in the PRAL between requesting nodes using control block surrogates. Token tracking has not yet been implemented and remains an important goal.

Theory and figures for token tracking can be found in [Skousen 1994; Skousen and Miller 1996b; Olson 2002].

4.4 Pager

The Sombrero pager, unlike pagers usually found in a MASOS, is designed to present the same view to all programs running on a Sombrero network. The pager depends on a consistency service to obtain the correct pages from other pagers on the net. Pages are presented on demand on a page fault after an attempted access has cleared the protection mechanism. Like other pagers, it has a component to support virtual paging in RAM as well as backing to a persistent store. The pager developed for use on the early NT version of Sombrero provided a persistent view of the VA space that survived system boots. This has not yet been integrated into the current version of Sombrero.

Since Sombrero is implemented in a 64-bit address space, the pager cannot use traditional page tables indexed by the VA without consuming a significant portion of RAM. Instead, the existing implementation uses an inverted page table and a hashing technique to index into the page table. By this means, the RAM consumed by the pager always remains proportional to the RAM available on the current node.

As mentioned before, in Sombrero there is only one view supported by the pager for all programs and so only one set of page translations exist per node. In contrast, a MASOS must support a distinct set of translations for each process. As a consequence of the single set of translations found in Sombrero, it is feasible that, unlike a MASOS, a hardware accelerator (e.g., a Content Addressable Memory, (CAM)) on the system memory bus, relieving the CPU of all but table updates, can support the single set of

translations. This should bring the translation performance of even an inverted page table up to par with that of MASOS processors serving smaller address spaces.

Another effect of the Sombrero design is the perception of the role of RAM in the memory hierarchy. MASOS designs tend to be RAM-Centric or focused on moving data and programs into and out of RAM as mentioned earlier with respect to the need to preserve data between program executions. Sombrero allows RAM to be treated more like a level in the cache hierarchy enabling well understood cache schemes to be employed in its support.

4.5 Persistence

VA space persistence in Sombrero is a core issue that affects the degree to which system and program complexity may be reduced from that of the MASOS paradigm. The Sombrero design proposes to support the persistence of the VA space for the life of the data it stores, relieving the program designer of the requirement to convert data to a differently organized non-volatile secondary storage, such as a file system, for long-term availability. However, the VA space is intuitively identified as the most vulnerable part of the Sombrero design. A VA space is vulnerable to corruption and so is any other computer resource on any computer. Although we have not been able to experiment with distributed persistence in Sombrero yet, there is an argument that can be put forward in support of the Sombrero design to address the apparent vulnerability. First, a few observations need to be made.

VA space persistence is an essential contributor to reduced complexity in Sombrero and so it is valuable even if it is vulnerable. Past experience suggests that, in

general, computer systems become corrupt and need to be rebooted. Forms of corruption include unhandled program exceptions, power failures, and equipment failure. Corruption seeded early in a computational effort by human error or bugs propagates for several backup cycles until it is noticed but recent copies of data will now only reseed the problem. Many forms of corruption exist in current designs, so what happens to the data in Sombrero when a failure occurs? Is it worse? Of course some loss occurs. All of these problems exist to some degree in every computer system. However, users have learned to coexist with them, and techniques to cope with them get better over time.

The traditional MASOS line of defense against a corrupted operating system is the division between the persistent store, typically a hard drive, and the more corruptible volatile store in RAM where the operating system runs. If there is a failure, just press the reset button and everything is rebuilt from the disk. Using this line of defense gives a fallback position to what is typically considered a more reliable device: the hard drive or even additional levels of backup. This fallback position is the comfort zone in which we have learned to coexist with our computers. The elective absence of that fallback position in Sombrero (it is still available if you want it, but program complexity increases) makes people feel uncomfortable, but the reality is that none of the above mentioned system corruptions are erased by the fallback strategy, only mitigated. Our disk drives still fail. Our tapes and CDs become unreadable. Rebooting won't cure bad data stored on the disk.

Here is the argument: If we found reliability techniques with which we can work in the MASOS paradigm, we can also find workable techniques to coexist with our computers in the Sombrero paradigm. We can find these techniques if maintaining the

reduced complexity and the accompanying cost reduction for development offered by Sombrero makes it worthwhile to do so.

Reliability techniques in the works for Sombrero include the protection model featuring the RPLB, fault tolerance based on replication over the network, consistent snapshots via checkpointing and others.

4.6 Design Influence

The several preceding design issues discussed in this chapter have been the guiding influence for the top down part of Sombrero design. The bottom up influence has been the available platform and how it can be utilized. No formal specification document was developed for the middle level architecture, instead, this being an experiment, each problem and its solution has been handled in order and applied. This was done to gain experience with the expectation that a formal specification of the operating system can be developed for an implementation of Sombrero in the next iteration of the design. The next chapter documents the design decisions that arose out of problem resolution as the process of differentiation, mentioned earlier, and the issues it exposed, manifested itself.

5 MIDDLE LEVEL SOMBRERO ARCHITECTURE

The partial implementation goal for Sombrero to achieve an object oriented modular system design that allows peer entities to interact across protection barriers has been realized. A great deal of refinement remains and this chapter discusses both the middle level architecture put in place to achieve the current goal as well as some of the pending refinements. The addition of key words and concepts to the compiler, the chosen entry point protocol, PAL Code support, libraries, interrupt management, scheduling, wait queues and etcetera will all be discussed.

5.1 Basic Definitions

Definitions of some terms used in the following discussion are presented here first. Some of these terms have a particular usage in Sombrero that may differ from standard usage. Some terms, such as RPLB, defined earlier are not repeated.

- Thread – A context of program execution.
- General Protection Domain (GPD) – The set of code, data, and switches reachable by a thread of execution operating within the protection domain that are not directly associated with the thread.
- Carrier Protection Domain (CPD) – The set of data and switches reachable by a thread of execution that are directly associated with the thread.
- Protection Domain (PD) – The union of the GPD and the CPD.
- Memory Object (MO) – A discrete range of memory sharable between protection domains.

- Program Class Object (PCO) – A memory object containing object code, previously generated by the compiler, which is used to instantiate a Sombrero execution environment suitable to receive threads. PCOs are typically shared by instantiations of the same type.
- Program – Synonym for PCO.
- Executable and Linking Format (ELF) – A well known (not Sombrero specific) format for object files generated by a compiler.
- Instantiated Memory Object (IMO) – A memory object in which data, entry points, and a global pointer table are stored. The Read/Write portion of an instantiation of a program. The IMOs are typically initialized by decoding the ELF formatted section information in a program.
- Data Instantiation – Synonym for IMO.
- Instantiated Program Object (IPO) – A bound program/data instantiation pair that is associated to create a Sombrero execution environment capable of receiving threads.
- Instantiation – Synonym for IPO.
- Service – A synonym for IPO.
- User – Same as a service. Used to distinguish between callers and callees. Callers are users and callees are services.
- User Program – A non-system program.
- Library – A special program with no assigned heap or tuple space when instantiated. A reservoir of useful subroutines. Multiple libraries are generally coinstantiated with a non-library service in a GPD. Each has its own OID.

- Standard User or Service – Any instantiation that is not an instantiated library.
- Module – Usually refers to a service. Sometimes, in context, refers to the program by itself.
- Heap – instantiations not designated as library instantiations have a heap. The heap allows instantiations to allocate small or large amounts of memory as needed that are accessible to the instantiation within its protection domain. Heap allocations are not discrete memory objects sharable outside the GPD.
- Tuple space – A memory space, not currently in use, intended to provide storage for data with variable indexing depth. The first intended use is for environment variables or as a registry.
- Entry Point - A defined point of execution entry at which services are obtained from an instantiation. There must be at least one entry point, and there can be many in an instantiation. In conjunction with the function that it points to, an entry point corresponds to an object oriented method and acts like a subroutine expecting a return address as one of the parameters. The privilege of executing instructions, beginning at the first instruction of an entry point, is owned by the protection domain that encompasses the entry point's instantiation with its data instantiation and program. Entry Points are stored in the data instantiation and transfer control to the program.
- Method – The function accessed via an entry point. The term ‘method’ comes from object oriented programming terminology.
- Return point – A user has control returned to it by the service through the users return point.

- Exit stub – A caller uses the exit stub as a proxy for an external function to resolve the address of an external function’s entry point using its copy of the function’s signature. At runtime, the user invokes the exit stub to properly exit one instantiation to call another instantiation’s entry point.
- Proxy Class – A standard C++ class generated by the compiler at the same time as its program and stored in the class library as a C++ header file. It defines a program’s entry points to other programs.
- Proxy – An instantiated proxy class. A proxy is used in Sombrero C++ programs to provide uniform access to instantiation entry points. Proxy methods call exit stubs that in turn call external entry points.
- Tail switch – A one-shot implied return capability stored on a thread tail stack.
- Tail Stack – A thread stack for pushing tails in the order of occurrence. This prevents a tail return from being used out of order.
- Global Pointer (GP) – An address that points to a table of pointers and data produced during program instantiation and becomes the data instantiation identifier. The GP is typically stored in a register called the global pointer register.
- Global Table – The table pointed to by the GP.
- Object Identifier (OID) – the same as the GP. The OID is used as a unique identifier for data instantiations.
- Standard OID (SOID) – OID of the most recent calling standard user. Libraries don’t have a SOID. Used by a service to identify the caller.

5.2 Sombrero Programs and Instantiations

The first problem for the designer to solve in the middle level architecture is the nature of a Sombrero program. The early programs were compiled in VC++ 5.0 using the `__int64` and `__ptr64` type modifiers to run in the extended address space above Windows NT. When the project was moved to a bare platform it was compiled by GCC on a Linux platform with modifications. The chapter on tools has details of the Sombrero compile sequence.

Instantiated Sombrero programs are all Instantiated Program Objects (IPOs also know simply as instantiations). In any relationship between two instantiations one is a caller and the other a callee, or user and service. A user is a service calling another service. System services are normal services distinguished by having been instantiated at boot time or later for the purpose of managing system resources. There is no system program running in kernel mode so all programs, user and system alike, run as peers protected from each other by their respective GPDs (Figure 1).

A Sombrero instantiation exhibits the features of a classic object oriented object with encapsulated data and methods and provides the basis for Sombrero's Object oriented system architecture. Sombrero instantiations are instantiated from Program Class Objects (PCOs). A PCO (also referred to simply as a program) is generated by the Sombrero compile sequence (Chapter 6) and downloaded from the host machine during the Sombrero boot phase (Chapter 7) or by the Sombrero loader, which is a running Sombrero system program. During run time the Boot Loader or system loader generates an Instantiated Program Object (IPO) from a program by creating an Instantiated

Memory Object (IMO) and binding it with the program to form the IPO (instantiation). IMOs (data instantiations) and their bound programs must reside in the same GPD. Multiple instantiations of the same program are possible by instantiating multiple data instantiations for a shared program resulting in multiple instantiations of the same type. Data instantiations store the entry points, volatile data, and the object table and are what distinguishes one instantiation of the same program from another. The program stores the text, read only data, and initialization information for creating the data instantiation. Figure 1 shows some instantiations (boxes). Figure 4 shows data instantiations and programs interacting.

Sombrero programs communicate with each other through entry points that are stored in the data instantiation. The entry points are stored in the data instantiation so that every entry point for every instantiation has a unique address to prevent ambiguity during GPD switching. The entry points in the data instantiation point in turn to the corresponding function text in the program. Collectively, as viewed from a calling routine, entry points and their associated functions correspond to the methods of standard object oriented programming. To simplify the discussion we will usually refer to this combination as an entry point and sometimes as a method.

Entry points have a second purpose besides providing a unique address for protection domain switching. Every data instantiation has an identifier called an Object Identifier (OID) to distinguish it from any other data instantiation by name. The OID is the value of the pointer that points to the object table in the data instantiation. For each data instantiation/program binding there is only one object table. The object table pointer

is also known as the Global Pointer (GP). The GP resides in the GP register during program execution to allow the text to reference data and pointers stored at offsets into the Global table. The Global table (Figure 2) stores pointers to static data and some functions used during program execution as well as some management pointers and data. The object table is created by the loader at load time and is the mechanism that allows shared text to run in independent programs.

Given the nature of the global table and the GP (OID), it can be seen that if an instantiation is entered and begins to execute with the wrong GP it will likely fail. To prevent this from occurring, the second important purpose for using separate entry points for every instantiation can be seen. During entry into the entry point a thread's GP register is loaded with the new data instantiation's GP. On return from an entry point the Return Point in the caller's data instantiation restores the caller's GP.

Deller and Heiser [1999] provide a good treatment of dynamic and static linking in MASOS as well as SASOS environments. Sombrero does not as yet take advantage of statically linked libraries since they generally involve making a copy of the library code in each program. The more interesting treatment in a SASOS like Sombrero is the sharing of a single copy of library code for all programs that use the library. Sombrero's link strategy can be classified as lazily evaluated dynamic linking. The link strategy is lazy because it only resolves a link when it is used, and dynamic because it makes use of a shared copy of the library code. The pro and con of such lazily evaluated dynamic linking are respectively: The instantiation doesn't require all of its resources to be available before it starts, and the instantiation may fail after it has started because a resource is not

available. The pro outweighs the con in terms of flexibility, but the con can result in data corruption if the program is not ready to handle it.

Another link feature, although perhaps only a temporary one, is the use of a version number to obsolesce a running program's dynamic links. This version number is the first entry in the global table. If the entry in the global table changes, such as when an externally linked instantiation is destroyed, all entry point links in the local instantiation are made obsolete by incrementing the version number and the run time linker reevaluates them on a lazy basis. This feature has not yet been fully implemented but the pieces are basically in place⁶.

The preceding paragraphs of this section described the behavior and facets of a Sombrero program. The remaining discussion will revolve around the alterations to the GCC output required to compile a Sombrero program.

```
typedef struct oidstruct {
    UINT64 version;           // update version number for oid
    UINT64 returnpoint;      // return entry point for oid
    UINT64 space;            // tuple space base address
    UINT64 heap;             // heap pointer for oid system heap
    struct oidstruct *oidnext; // pointer to next oid in chain
    pENTRYPOINT eabase;      // base of entry points in oid
    UINT64 *pEPList;         // point to list of EP symbols
} OIDSTRUCT, *pOIDSTRUCT;
```

Figure 2 Global Table to which a GP (OID) points.

GCC in the Alpha architecture imposes a global pointer (GP) on the executable code that is resolved by the linker. For Sombrero, programs are not compiled to a particular address as they are in Linux, rather they need to be relocatable to support multiple instantiations in the same address space. This relocatable object is called the

⁶ Ultimately, when performance becomes more important than debugging the version number strategy will no longer be used.

Program Class Object (PCO), or just program, since it is a named template for an instantiation. The relocatability is achieved by altering the output of GCC to remove the use of the GP for control transfer within the program, substituting branch and branch subroutine instead, and reserving GP for data references. As described previously, GP points to a table of pointers to static data and some functions called the Object Table. The first several pointer entries are reserved by Sombrero in the OIDSTRUCT to support system management (Figure 2). The pointers that follow the structure are determined by the compiler and are required by the program as function pointers and indirections to load static data from the correct data instantiation.

Other alterations to the code support the C switch term, an auxiliary return register for entry points, and support for integer division.

The semantic changes in C++ source code that support Sombrero include directives for generating a program as well as a class header file that represents the program for use by other programs. A key word, `//ENTRYPOINT`, is also inserted before functions that are intended to act as externally available entry points.

5.2.1 Project Class Header File

The Sombrero compiler outputs a Project Class header file providing a class type to represent the program at the same time as the program. This class type is stored in a library as a header file for use by other projects and is used to provide a declaration of type *program* to other programs. The initial project class header information is stored in the source code on the Host computer as comments (Figure 3a) and begins with the `//BEGINCLASSHEADER` directive. The initial lines of a class declaration are provided

here including the identifier (the class name) by which the class will be known during run time linking. The compile process generates the remaining lines of the class header including the class methods. The class methods are derived from the list of declared entry points (Figure 3c) for the project that are embedded further on in the source code. The class header file (Figure 3b) is output by the compiler and placed in a public library during the compile process. There, projects can get declarations of each other's class type (external class type) so they can access each other through the class methods. During run time, external class types are used to instantiate internal proxies for external instantiations.

Libraries come in two flavors. A regular program, constructed from a list of static functions marked as entry points, is one type and is represented by only one program class type. The other flavor is a Class Library, much like Microsoft's Foundation Class Library, containing actual methods for one or more normal C++ class types in addition to the program class type.

Multiple class type declarations are allowed to appear in the same project if the project is declared as a `LIBRARY` by a directive on the same line after the `//BEGINCLASSHEADER` directive and the other declared classes are for class library entries.

Library data instantiations coexist as secondary data instantiations in a GPD with the instantiations that use them. They are embedded in the same MO as the primary data instantiation after the primary data instantiation's global table. Other Sombrero directives that appear on the same line are `LOCALCLASS`, `DEFAULT`, `NOOID`, and `BOOTINIT`.

a) - Declaration in SOSLinker.cpp source file

```
//BEGINCLASSHEADER BOOTINIT
///

```

b) - Final Class header SOSLinker.h generated by the compiler for other calling classes to use. Compare the first few lines to a.

```
//NAME:SOSLinker
//
// Class definition file for SOSLinker - SOSLinker.h
//
class SOSLinker
{
public:
    void SOSLinker_CONSTRUCTOR(void *bl, void *dl);
    void SetAlphaUser();
    UINT64 FindBootEP(UINT64 targetoid, char *epname);
    void *EPSymbolResolution(char *epname, char *path, UINT64 gpdarg = 0,
UINT64 instarg = 0, UINT64 oidarg = 0);
    void SavePrint(UINT64 pv);
    UINT64 GetPrint();
    void AssignBootListToMOS();
    UINT64 FindClass(char *cname);
private:
    long __sosobjectversion;
    long __sosobjectid;
    long __sosclassversion;
    char *__sosobjectpath;
    long *__sossuperclassoid;
public:
    inline SOSLinker(){__sosobjectversion=__sosobjectid=__sosclassversion=0;
__sossuperclassoid=0; __sosobjectpath=0; };
    inline void *__GetOid(){return (void*)__sosobjectid;}
    inline void *__SetSuperOid(long *superoid){__sossuperclassoid=superoid;}
};
```

c) Example of an entry point declaration for SOSLinker

```
//ENTRYPOINT PUBLIC
UINT64 FindClass(void*, char *cname)
{ . . .
```

Figure 3 Class header and entry point example from SOSLinker.

When multiple class header declarations appear in a LIBRARY, one of them must be the DEFAULT class header and the others must be declared LOCALCLASS. By this means, class libraries can be constructed.

LOCALCLASS informs the compiler that the declared class is not a program class. Instead it refers to a normal C++ class that is instantiated in the program as a declared object of type class, but methods for the object are stored in the library. This allows libraries to contain code to implement class based data types such as heaps, trees and queues. The methods for these classes are resolved to class library entry points, but the class library functions treat the objects passed to them in the *this* pointer as normal C++ objects of type class. Entry points that are not declared as a member of a LOCALCLASS become part of the DEFAULT program class.

NOOID is used by LOCALCLASSEes to discard a variable embedded in the class by the compiler that contains the OID of the service calling the library. This variable has not proved very useful and will probably be dropped making NOOID obsolete.

BOOTINIT insures that the class will be instantiated by the boot loader.

5.2.2 Entry Point Implementation

The //ENTRYPOINT directive (Figure 3c) declares that the next line, which must be a function declaration, marks an entry point for the program. The function is cataloged and modified to act as an entry point during compile. It is recorded as a method of the project class to be included in the header file output. A signature for the entry point is placed in the program that will be used by the run time linker to resolve first time and updated entry point requests from other instantiations.

The modification on the function declaration referred to in the last paragraph enables the entry point function to preserve two return addresses. The first return is to the Return Point of the calling IPO, the other, the auxiliary return, is the return pointer to the initial method call in the calling instantiation (Figure 4). The reason for two return pointers is covered in the description of exit stubs and return points below. Entry points can be called without the auxiliary return pointer, as long as they don't return to a Return Point.

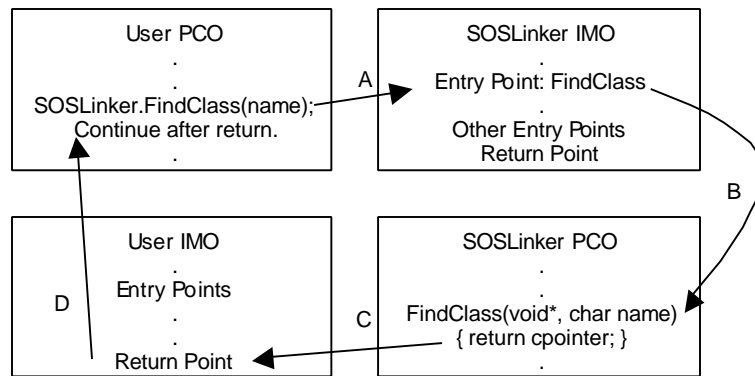
The Entry point directive has additional directives that follow on the same line. Currently there are two: PUBLIC and PRIVATE. These are used to create default access policy for allowing entry to an entry point. PRIVATE allows entry only by callers from the same GPD, PUBLIC from any GPD. The use of these two directives is a somewhat different semantic than the common usage for public and private in C++.

C++ class methods normally have an implied first argument that is always a pointer to its data structure in memory. It is commonly known as the “*this*” pointer in C++. Since normal static function definitions declared as entry points are being converted to class method entries by the compiler the first argument is discarded to accommodate the C++ method syntax on the caller side. Compare the entry in Figure 3c for FindClass with the corresponding entry in Figure 3b. The meaning of “*this*” in the proxy class instantiation on the caller side remains the same. However, the first argument of the entry point function on the callee side is generally never used unless the class was declared LOCALCLASS. In that case, the meaning of the first argument on the callee side and the

accompanying “*this*” pointer are as expected in C++, and the memory “*this*” points to is also expected to be accessible in the callee’s protection domain.

One of the most important behaviors of an entry point is that it loads the GP register with the OID for the instantiation that is being entered. This allows the global pointers to be properly used. The return point restores the GP register to the OID of the calling instantiation on return. It can also be seen in Figure 4 that entry and return points are part of the data instantiation (IMO) instead of the program (PCO). Storing the entry and return points in the data instantiation is necessary not only to provide a unique per instantiation OID to the entry and return points, but also to provide a unique address for each entry point for each instantiation so that no ambiguity exists on an implied GPD switch.

On another issue it is important to note that entry and return points are designed to work the same whether a GPD boundary is crossed or not. This permits programs to function properly even when the RPLB is disabled, especially during the boot phase, and takes advantage of the transparent and implied nature of the RPLB switch.



SOSLinker is a proxy. FindClass is one of its methods: A) Call entry point; B) GP register is updated followed by Jump to actual code; C) Function return is to caller return point. Restore user OID to GP register; D) Final return using Auxiliary return.

Figure 4 Runtime depiction of a proxy method invocation.

5.2.3 Exit Stub Implementation

Exit stubs are stubs added to a program in order to facilitate exit from the program and set up the return and auxiliary pointers. Exit stub implementation permits normal C++ “.” (class member) and “->” (pointer to class member) syntax to be used to call entry points in other instantiations. The use of normal syntax to access external entry points is done by way of a proxy class instantiation. The project class header files generated by the compiler provide class types that are used by a program to create internal proxies for real external instantiations whose entry points the calling program may want to access. For example to access FindClass in the run time linker:

```
#include <SOSLinker.h> // this includes the linker's program class header file
SOSLinker SOSLinker; // this declares a proxy for the linker
cpointer = SOSLinker.FindClass(cname); // access the FindClass entry point
```

When the compiler detects external entry point calls in the program source code it generates a local stub function, the exit stub. At run time, the stub function intercepts the external call in the caller and checks for linker resolution. If the internally stored pointer is unresolved or outdated the run time linker is called. When the external entry point pointer is valid the return point is loaded from the global table (Figure 2 `returnpoint`) and put in the return pointer register. The real return pointer is placed in an auxiliary return pointer register, and the method call to the callee's entry point is continued.

The local proxies that represent external program instantiations serve three purposes. First, to perform the initial link resolution. Second, to update the link resolution if it becomes outdated. Third, to associate a path with link resolution so that a particular program class instantiation will be located during resolution. The path and update features will not be tested until a directory is in place for Sombrero.

Exit stubs may differ from the normal behavior described above depending on the Sombrero directives that have been used. The directives `LOCALCLASS`, `NOPROXY` and `NODEAWARE` are mutually exclusive. `LOCALCLASS`, as explained earlier, takes precedence, then `NOPROXY`, and finally `NODEAWARE`.

`NOPROXY` allows an external instantiation's `OID` to be cast as a pointer to the external class instead of having a local proxy for the external instantiation. When `NOPROXY` is used the `OID` cannot be automatically updated by the system if the external class is destroyed and reinstantiated, as occurs in debugging a service, nor can a path be associated with the class. This is useful however when the current program

creates one or more instantiations from the same program and has direct control over the validity of the pointers.

NODEAWARE allows migrating threads and data to always locate the system modules associated with the local node. Locating local node services is important, for example, to the consistency service for locating the node protection service and pager. This will not be tested until the distributed features of Sombrero are implemented.

5.2.4 Return Point

The return point is primarily used to restore the callers OID on return from an external entry point as well as to provide an unambiguous address for protection domain switches. The thread then uses the auxiliary return to return to the instruction following original method invocation. All method invocations that go through caller exit stubs return through the return point.

5.2.5 Constructor and Static Constructor Entry Point

A remaining feature of the Sombrero program relating to the compiler is the use of a static constructor entry point. This entry point is only used during program instantiation by the loader as a means to locate the static constructors for a program class. There may eventually be a static destructor entry point.

Following the object oriented model further, Sombrero programs have an optional Standard or Main Constructor that, if it has no arguments, is called by the loader as a last step in the instantiation procedure.

5.2.6 Standard OID

The Standard OID (SOID) is the object identification of the most recent calling standard user. SOIDS are pushed onto the tail stack between tail switches by the exit stub to provide the identity of the user that initiated the current method invocation. Libraries do not push or pop SOIDS, so service calls via a library will identify the correct initiating user. Only the most recent SOID may be read by a call to the PALCode. Pushing or popping them improperly can corrupt SOIDS, but the tail switches are not affected and the most recent tail acts as a boundary to the corruption.

5.3 Program Class Object Instantiation

Program (PCO) instantiation in Sombbrero, as in other systems, is a non-trivial process. Sombbrero programs are compiled using the Executable and Linking Format (ELF) object file format. As yet, use of the detailed initialization and debug information provided for in ELF is not fully exploited, especially not the debug features. However, the symbol table and relocation information stored by ELF must be processed to properly initialize and bind a new data instantiation (IMO) to its program, and to create the global table pointed at by the GP.

In addition to initializing data and pointers in the data instantiation that will be bound to the program, the ELF symbol table is used to locate the Entry Point list and code, the Return point code, the program class name, the initialization type (LIBRARY, BOOTINIT), and the static constructors.

When initialization in a data instantiation is complete the loader will have created the entry points, the return point, the heap, the tuple space, and the global table, and

bound the data instantiation to its program forming a complete instantiation. For a new GPD, it will have created the GPD and recorded the access rights to the program, and the data instantiation for the new GPD.

To complete the instantiation process the static constructors are called, and finally the standard/main constructor is called to allow the new instantiation immediate access to a thread.

It is important to note that data instantiations should not share execution access rights outside of the owning GPD, as this would introduce ambiguities during the RPLB switch.

5.3.1 Grouping Class Instantiations

When a new service is instantiated, it will typically have need of the services of one or more libraries. The string library, the stdlib, and the classlib are some of the libraries currently available. None of them are complete. Libraries have properties that make it useful for them to share memory with the caller. In particular, the classlib needs to manipulate C++ classes instantiated in the caller data space. In addition, arguments, such as string pointers, often point to data that does not always conveniently appear on the thread stack. Putting data on the thread stack makes the pointed to data portable with the thread via the CPD. Otherwise, the pointed to data may become inaccessible if a GPD crossing occurs.

To accommodate the need for library instantiations to be in the same GPD as the user and to provide separate library static data for each user, libraries are co-instantiated in each new data instantiation that references them. Other strategies besides co-

instantiation can be used, such as read only sharing of a library data instantiation. However, read only sharing has not yet been experimented with and would not allow for per-user static initialization in a library. Co-instantiating a user's libraries in the same memory object (MO) as the user considerably reduces the number of required domain crossings, access descriptors, and memory objects. Co-instantiated data instantiations sharing the same memory object are linked in a list that the run time linker uses to resolve entry point symbols (Figure 2 oidnext pointer). The first OID of the chain of data instantiations is stored in the access descriptor of the data instantiation chain's MO for the GPD.

The Boot Loader instantiates all of the system programs declared with BOOTINIT in a single large MO and links them in the aforementioned OID chain. This is convenient for now. It may change later.

5.4 Symbol Resolution

Symbol resolution in Sombrero follows an ordering that first resolves external entry point requests to entry points found in the OID chain constructed by the loader. It then looks for other OID chains in the same GPD. Finally, the node system GPD is searched followed by a directory search. As Sombrero matures this is subject to change.

5.5 Interdomain Communication (IDC) and Protection

Interdomain Communication between GPDs is the mechanism by which threads change GPD context (switch) to have access to convey information to and obtain services from previously unreachable code and data. In Sombrero this was designed to be a very inexpensive mechanism. The proposed protection mechanism, with its ability to switch

transparently between GPDs when the need is implied, is somewhat new in nature but has limitations. One problem is that like a branch or a jump there is no implied return to a switch; it just works in one direction. When combined with a method invocation to an entry point this leaves the return side of the permission orphaned. To prevent security holes an implied return switch must be granted only when a legal method call has been made, and then only once. The return switch must also be made in the correct order to properly unwind the thread stack and prevent untoward services from fishing for a hole.

5.5.1 Tail Switch

A solution implemented in Sombrero to resolve the one sided GPD switch is the Tail Switch. The Tail Switch does not come without cost. The previously costless switch can now be flagged to generate a tail stack entry that is automatically detected and popped by a thread when it attempts to return across a GPD boundary. The costless switch can still be used but the return permission must be set up in advance by creating a switch for it at the caller's return point.

The tail stack is inaccessible except to push, pop, and read SOIDS. Tail switches cannot be read or discovered in any way. They are protected by SOSThreads, the service that creates and destroys threads. A register containing the most recent tail switch is available to the RPLB as it checks attempted instruction execution. A match on the address and protection domain stored in the tail registers initiates a switch, and the tail is popped from the stack. In doing so, the tail switch also forces the auxiliary return to the

value it had at the original call, so it is tailored to be used with the entry point/return point mechanism and provide security⁷.

5.5.2 RPLB Implementation

In Sombbrero, protection and domain crossing are implemented in an emulated RPLB. When this document refers to costless GPD switching, the caveat “if the RPLB wasn’t emulated” must be applied. If RPLB performance measurement were an objective of this dissertation the cost of RPLB emulation would certainly be backed out of the performance measurement to arrive at useful information. In this section, the implementation of the RPLB emulation is described to give a greater understanding of the Sombbrero prototype.

The RPLB is implemented wholly in physical memory so it has no translation dependencies. The current implementation of the RPLB is split into two parts; the DRPLB to match against data access, and the IRPLB to match against instruction access. Both are tables that appear after the PALCode in physical memory. The PALCode part of the RPLB is entered on a miss in the Alpha architecture TLBs so the smallest resolution that can be managed by the emulated RPLB is one page. The emulation operates by comparing the current VA access, GPD name, CPD name, and access type to the entries in the DRPLB or IRPLB table depending on the type of TLB miss. On a match, the entry in the RPLB is forwarded to the pager to allow that page with the all the allowable access rights to be entered as a translation in the respective instruction or data TLB. If the match

⁷ Tails bear only a vague resemblance to the x86 architecture access descriptors and are much simpler since multiple privilege levels and multiple segment types and multiple control transfer types are not an issue [AMD02].

in the RPLB is a switch the GPD register is updated to point to the new GPD and the RPLB is searched again in the new context. If the RPLB miss was for access to an instruction it is also checked against the Tail Register and handled accordingly.

On a miss in the RPLB the attempted access is forwarded to the PAL Cache for PRAL entries that holds a much larger list for updating the RPLB. If a hit occurs in the PAL Cache the RPLB is updated and searched again.

The PAL Cache table follows the RPLB tables in the physical memory and is an interleaved circular list of capability trees that is updated on PAL Cache misses.

For PAL Cache misses a set of miss handler threads are set aside. When the miss occurs the context of a miss handling thread is updated with the parameters of the miss and its status is changed to allow it to run. A fault or forced context switch then occurs on the miss handling thread starting it at the entry point of SOSAccess, the PRAL manager, to handle the PAL Cache misses. The original user thread's status is changed to a wait state waiting on the miss handling thread. The PRAL is searched and on a hit the RPLB and PAL Cache are updated and the original waiting thread is scheduled to start next so it can finish out its quantum. The miss thread then suspends itself awaiting the next miss.

5.6 PAL Code Support

The Alpha architecture supports a Privileged Architecture Library (PAL) mechanism that allows the Alpha Reduced Instruction Set Computer (RISC) architecture to maintain its level of simplicity by moving some tasks into a special programmable mode. This PAL mode handles all faults, exceptions, interrupts, and an array of programmable instructions as vectors into executable code. The array of programmable

instructions are PAL codes and the programming for the PAL codes are collectively known as PAL Code. PAL Code is distinguished from kernel or other privileged modes in that it runs only in physical address space with data access to the VA space and has access to special CPU registers.

The PAL Code in Sombrero is used not only to emulate the RPLB, but to provide an array of Sombrero support registers, perform context switches, schedule interrupts on blocking threads, manage wait queues for locks and semaphores, manage the tail stack, and respond to breakpoints. Some of the Sombrero PAL Code features, like the breakpoint vectors, are temporary, and as the implementation matures others no doubt will be added.

A goal, and it has not been reached yet, is to eventually determine whether all features currently placed in the PAL Code, as well as those to be added, can be implemented either as hardware or pushed up into a protection domain. If it can be implemented in hardware our current implementation of that feature is a hardware emulation. If a protected layer must exist to support some feature then the concept of a special kernel mode must remain, albeit a very simplified kernel.

5.6.1 Faults and Interrupts

So far, interrupt handling is managed by blocking threads in interrupt handlers on any permissible interrupt vector. This causes a thread to be signaled on an interrupt causing a context switch. The cost is nominal enough for most purposes. Additional experimentation is required to determine whether the thread executing the code when the interrupt occurs would best handle some of these services for performance reasons. In

any case, some level of thread context will have to be saved and restored to enter an interrupt handler. If the user thread is used to service an interrupt then a table of explicit entry point switches with tails will need to be created to vector the user thread to the proper handler entry point and GPD.

Threads are given permission to block on an interrupt vector by `SOSInterrupt`. The PAL Code places the thread in the interrupt wait queue in physical address space when it executes the `blkintth` PAL code instruction. If the GPD the thread is running in has no prior permission created by `SOSInterrupt` it fails to block and does not have interrupt handler status. Eventually, when Sombrero has a good exception handler, the thread will go to the exception handler when no permission exists. Currently, the only active interrupt is the system clock.

Faults are handled somewhat differently than interrupts. A fault handler, like the interrupt mechanism, has a thread or threads allocated for the purpose of providing a context for fault handling. By this means the user context can be preserved. Unlike interrupt handling threads, fault-handling threads do not currently block in the fault handler. Instead, the fault mechanism must set up the context for the fault-handling thread and start it at the handler entry point.

Currently the only active fault mechanism is the RPLB PAL Cache miss handler described earlier, and is implemented in PAL Code.

5.6.2 Wait Queue

Sombrero has a wait queue in PAL Code designed to handle semaphores and locks. PAL Code handles the semaphores and locks via PAL Code instructions. To

protect semaphores and locks from illegal access they are accessed in PAL Code by their virtual address to cause faults on permission violations. When a thread blocks on a semaphore or lock it is linked to a queue headed by the name of the semaphore or lock. The queue provides a first in first out mechanism for blocking on an event. The queue design benefits greatly from the use of the single address space allowing any read/write address accessible to the thread to be used as a global semaphore or lock. Semaphores are invoked using the *waitsemaphore* and *signalsemaphore* PAL Code instructions. Locks are accessed using the *waitlock* and *signallock* PAL code instructions. Spin locks are also available in Sombrero.

5.6.3 Context Switch

The context switch implemented in PAL Code performs a full context switch on Sombrero threads by executing the *contextswitch* PAL code instruction. All integer registers and all floating point registers are saved as well as a number of Sombrero registers including the thread name, current GPD name, current execution address and previous GPD name. Any scheduled thread is allowed to perform a context switch that will give up control to the next highest priority thread, which may be itself. The scheduler, of course, uses *contextswitch* extensively. The interrupt handler, RPLB PAL Cache miss handler, semaphores, and locks also use the context switch directly in PAL Code.

5.6.4 New Registers

Sombrero requires some additional hardware registers to support the RPLB and other features. The Tail Switch registers have already been mentioned, as have the RPLB

GPD and CPD registers used to match against entries in the RPLB. Additional registers that store the previous GPD (whenever a GPD is switched the previous GPD register gets the old GPD), the name of the idle thread, the next scheduled thread for the context switcher, base registers for the interrupt permission list, PAL Cache pointer, head and tail registers for a reschedule list, and the system interrupt mask are required. These all play important but fairly simple roles.

5.7 Sombrero System Modules

The current set of Sombrero system modules is, of course, incomplete. As such any attempt to develop a Sombrero API at this point would be incomplete. As mentioned earlier, the strategy for developing Sombrero is to build a workable prototype operating system and then write a formal specification, including a system API, for a well-understood version of Sombrero. The current set of system modules includes:

- SOSLinker – Run time linker. The Boot Loader places an entry point pointer to `SOSLinker::EPSymbolResolution` in a special linker register that is returned when the PAL code instruction *symbolres* is executed. The run time linker switches its search from the directory of boot objects to the symbol resolution sequence documented in section 5.4 at the end of Boot Loader execution. SOSLinker is the first system module to be enabled. `SOSLinker::FindClass`, used in previous examples, locates the memory object for a class if it is in the boot list.
- SOSLoader – Instantiates a program. It currently assumes a new GPD is required for each call to `SOSLoader::Instantiate`. SOSLoader will need to become more

sophisticated including searching for programs on the remote host and allowing instantiations into existing GPDs.

- SOSMem – Allocates and validates memory objects from a node's memory allocation range heap. SOSMem is frequently accessed to refill heaps, allocate thread stacks, and allocate new instantiated memory objects. SOSMem automatically calls SOSAccess to associate access descriptors with new memory objects.
- SOSAccess – Allocates access descriptors in the PRAL. It has an entry point for the RPLB miss handler.
- SOSInterrupt – Manages access to interrupt vectors. Interrupts are handled by signaling threads that SOSInterrupt allows to block in an interrupt handler.
- SOSNet – Provides access to the NIC. The Sombrero end of the remote debugger also resides in SOSNet since the debugger is currently the only user of the NIC after system boot.
- SOSProtection – Allocates and validates GPD control blocks.
- SOSThreads – Allocates and validates thread/CPD control blocks.
- SOSScheduler – Schedules threads on a set of priority run queues. SOSScheduler also supports queues for interrupt, wait and suspended threads. It accepts start thread and create thread requests. The system interval clock handler thread runs in SOSScheduler as does the system idle thread. It supports a system sleep function as well.

- SOSUser – Still in early stages of development. Eventually each user will have a SOSUser instantiation that will provide access policy and a directory interface for an individual user.

A number of other system modules that are not active are also under development and are not cataloged here.

5.8 Sombrero Libraries

Sombrero libraries are incomplete having additions made to them as demand occurs. Some basic string and memory functions are supported in the string library. *stdio* supports *sprintf*, *stdlib* supports *malloc*, *calloc*, and *free*. The code for these was obtained largely from publicly available Linux library sources and adapted to Sombrero.

classlib is far more populated supporting a class library for tree, queue, heap, and tuple space classes. These are the supporting classes for all of the system data structures. They represent fairly sophisticated data functions. The trees balance themselves, the heaps track fragments and merge neighboring fragments, and the queues grow themselves as needed, making management of the system data structures fairly trouble free. The tuple space available in each service is similar in function and purpose to the registry in Windows NT. The sources for the class library were adapted by the author from his own past work.

6 TOOLS FOR SOMBRERO IMPLEMENTATION

Implementing a new operating system from the ground up requires an eclectic gathering of tools both in hardware and software. They are chosen for the resources they provide and the time that can be saved in not having to reinvent them. The tools gathered here run on three different operating systems besides Sombrero.

6.1 Hardware

The Sombrero implementation makes use of existing tools as well as constructing new ones tailored to the needs of the project. The first consideration revolves around the platform of choice and what it supports, the second the development language of choice. The processor of choice is an Alpha 21164 processor based system since it supports a large enough address space to make experimentation practicable. It was also the most widely supported and advanced 64-bit CPU at the time the implementation was begun. The key consideration is the availability of the DEC Privileged Architecture Library (PAL) and PALCode facility [DEC 1997] that would permit the emulation of the proposed RPLB and other Sombrero hardware requirements. The specific platform eventually chosen, due to cost and availability, was the Alpha 21164PC from Samsung with the 164sx motherboard. The NIC of choice is the 3Com 3C905 since it is almost ubiquitous in its availability and its hardware documentation is available too.

6.2 Language

The language of choice for implementing Sombrero system modules is C++ since it is widely available and efficient and provides object-oriented features through classes

that can be used to model the Sombrero system modules. C and Alpha assembler (*asaxp*) on Alpha NT are used for the Boot loader and PALCode compilers.

6.3 Compilers

Sombrero modules are compiled using GCC on Red Hat Linux 7.1 for the Alpha processor. The assembler output of the compiler is further processed by custom programs to make the final object code compatible with Sombrero. The compiler steps are performed on an Alpha-based Linux Development machine (Figure 5). The Microsoft C (MSC) command line compiler under Alpha NT compiles and links the Boot Loader using *link*. The PAL Code is compiled by a combination of the MSC command line precompiler, the GNU Assembler (GAS) and *asaxp* followed by *link32*. These steps are performed on an alpha-based NT development machine (Figure 5). Some of the tools for the PALCode compile come from the DEC Evaluation Board Software Development Kit [DEC 1996].

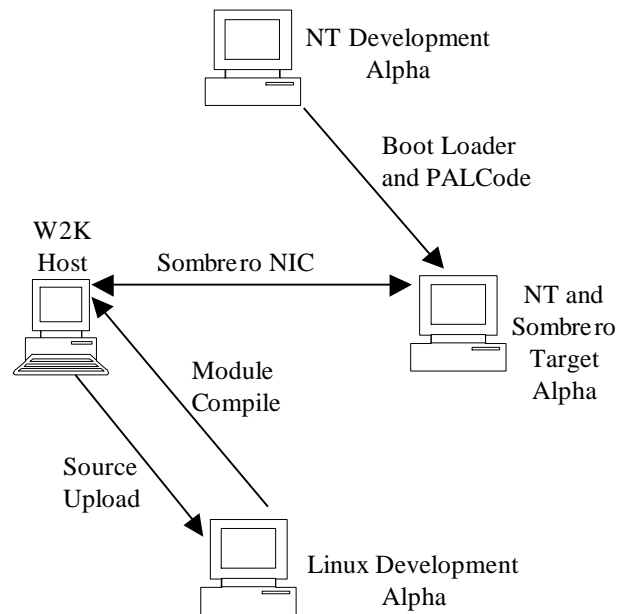


Figure 5 Layout of Development and Host Computers.

6.4 Development and Host Computers

As can be seen, there are a number of platforms that participate in the Sombbrero development environment (Figure 5). The complete set is:

- A Windows 2000 PC (W2K) Host computer for editing, remote debugging and User interface hosting. The Sombbrero Host that manages network traffic also runs here.
- An Alpha based Windows NT 4.0 development computer for editing and compiling the Boot Loader and PALCode.
- An Alpha based Red Hat Linux 7.1 development computer for editing and compiling the tools for the custom GCC Sombbrero compiler. This platform also compiles the actual Sombbrero executable modules from the source that the host uploads to it using Samba.

- The target Alpha PC running NT 4.0 and Sombrero. NT is used to transfer the PALCode and Boot Loader from the NT development Alpha to the target so it can boot into Sombrero. Sombrero is booted and runs on this machine.

6.5 Host Custom Tools and Services

It was necessary to develop a number of supporting programs that run on the host W2K system. These include the host side of the NIC connection to the Sombrero target, as well as a program to manage the Sombrero compiler running on the Linux Development Alpha. The host tools and services are:

- SOSHostDll - a dynamic linked library module designed to directly support the Sombrero NIC connection. Supports a protocol built over UDP/IP that allows messages and bulk transfers to be exchanged with a Sombrero computer using identifying addresses in the Sombrero VA namespace.
- SOSHost – Hosts Sombrero computers on a network so that Sombrero can launch or respond to user interfaces running on the host computer. Provides the initial set of boot system modules at the request of the Sombrero Boot Loader over the Sombrero NIC connection. Also will provide routing services to other instances of SOSHost running on the Internet to propagate the Sombrero VA namespace.
- SOSDebug – Remote debug user interface built over SOSHostDll and launched by SOSHost at the behest of a Sombrero Boot Loader. One instance of SOSDebug is launched for each Sombrero computer hosted by SOSHost. SOSDebug provides a disassembler, monitors registers, and memory, and supports breakpoints and tracing peculiar to Sombrero modules.

- SOSRBuild – Provides services on the W2K host to remotely build Sombrero modules on the Linux Development Alpha using an embedded version of rsh.

6.6 Sombrero Compiler Custom Tools and Services on Linux

The compiler for Sombrero on the Linux Development machine uses the standard issue GCC compiler [GNU 2002] in combination with a number of programs designed to alter the output of GCC. This strategy was used for now, instead of constructing a Sombrero cross compiler, to avoid introducing new, less apparent bugs by altering and possibly destabilizing GCC.

Three programs and a script were designed to support the Sombrero compile process on the Linux Development machine. The programs were written in C++ using Kdevelop 1.4 – the Linux KDE IDE:

- gcc-script – Actually builds the Sombrero module in stages using three gcc passes and two of the custom programs, buldsxe and catdebug.
- sosbuild - accepts arguments from SOSRBuild on the W2k host and begins a bash shell session for gcc-script.
- buldsxe – Executes on the Linux Development machine after a first pass error free compile from gcc that generates an assembler listing concatenating all of the output from the .cpp and assembler sources for this module. *buldsxe* processes the assembler listing to output a compilable assembler source file that resolves all local references and converts unresolved references into calls to other Sombrero modules. Produces the project header file.

- `catdebug` – Processes the final `gcc` object output on the Linux Development machine assigning offsets in the text for the global table pointers and concatenates debug source information to the object file. Produces the Program Class Object (PCO).

6.7 Other Tools

Additional tools including Visual C++ 6.0 (VC++) are used in the Sombrero compile process. VC++ is used, just for its editor, on the NT Development Alpha for the Boot Loader and PALCode source code.

6.8 Applying the Tools

Most of the software tools are directed a compiled output from project sources. This section details the compile of Sombrero modules, the Boot Loader and PAL Code.

6.8.1 Sombrero Module Compile

On the W2K host VC++ is used to create a utility project for each Sombrero module. One `.cpp` file for each project, containing the default project class header file information in the form of comments, is designated for custom build by `SOSRBuild`. The remaining files in the project are marked for exclusion from the build process. When 'Build All' is invoked in VC++, `SOSRBuild` is passed the name of a `.cfg` (configuration file), the directory for the project, the name of the final program file, and a `DEBUG` flag. The final output of a Sombrero compile is a `.sxe` program file and a `.h` header file containing a class type to represent the new Sombrero object.

The `.cfg` file identifies the name of the upload directory on Samba, the directory for the final destination of the object module, and the directory for the `.h` library for the

Sombrero generated classes on the W2K Host machine. It also identifies the name and an account on the remote Linux Development Alpha machine that SOSRBuild should connect to to perform the build.

The whole sequence for the build is:

1. SOSRBuild on the W2K Host uses data from its command line and the .cfg file to copy all files from the project directory on the host to a corresponding directory on the Linux Development Alpha samba shared directory.
2. A connection to the rsh server on the Linux Development Alpha is then established.
3. Once a connection has been established, `sosbuild` is invoked on the Linux Alpha machine and
4. `SOSbuild` begins a bash shell to run the `gcc-script`.
5. `gcc-script` makes a first pass with `gcc` to identify any syntax errors. All standard output and standard error output appears on the VC++ output screen on the W2K Host in a format that allows the user to link to errors in the source code.
6. If the first pass of `gcc` is error free, it is then again invoked to generate an assembler listing containing debug information and a list of unresolved labels.
7. The assembler, labels, and debug information are extracted by `buildsxe`
8. `buildsxe` then generates assembler output suitable for a final pass from `gcc`.
9. `catdebug` is invoked to assign global pointer offsets in the program object file and append indexed sources of the assembler and .cpp files for tracing execution in the debugger.

10. Finally SOSRBuild on the W2K host copies the Sombrero-ready module and .h file back to itself using the specified directory and name from the .cfg file.

6.8.2 Boot Loader and PALCode Build

The Sombrero Boot Loader is built under the Windows NT DDK Free Build Environment using nmake on the NT Development Alpha. It currently consists of two programs: somload.exe, and sombrero.exe which run on the target Alpha machine. somload.exe simply loads sombrero.exe and will eventually be discarded. Sombrero.exe and somload.exe must be manually dragged and dropped into the correct folder on the shared boot directory of any target Alpha. The Boot Loader also requires the installation of the Alpha Bios SDK, asdk, on the NT Development Alpha for it to compile properly.

The PALCode is compiled using the script PalArcB.bat on the Alpha NT Development machine. PalArcB.bat automatically places this PALCode in the correct directory with the correct settings on the Target Alpha. Additional target machines require dragging and dropping.

7 SOMBRERO BOOT SEQUENCE

Every LAN used with Sombrero requires the presence of a Host Windows machine. As already noted, W2K has been the host operating system for the implementation but other versions of Windows operating systems may work as well. Before Sombrero is booted on a target machine SOSHost.exe must be invoked on the designated host. At this time it requires no setup. Just start it and it listens. The host also requires a static Address Resolution Protocol (ARP) entry for any target machines.

The Target Alpha needs to have had a Sombrero OS boot selection created in the Alpha Bios. The following paragraph refers to settings in the Alpha Bios boot selection for Sombrero.

Somload.exe and its directory, if any, should be specified as the boot file. The OS path is ignored. 'OS Options' in the Target Alpha machine's Alpha Bios needs to include the following sequence separated by spaces: IP of the target machine, Net Mask of the target machine, IP of the host machine, MAC address of the host machine, and UDP listening port number on the host. The UDP listening port is currently set at 1010. See the following example:

```
OS OPTIONS: 192.168.1.203 255.255.255.0 192.168.1.200 00-03-6d-16-51-7f 1010
```

The Sombrero Boot Loader when invoked locates and initializes some hardware including the 3C905 NIC. It then loads the new Sombrero PALCode and activates it. Next, it registers itself with the host to acquire an address space partition from which it can allocate memory objects. If it is the first Sombrero machine to register it is assigned the entire address space. The Sombrero boot loader can now load the set of compiled boot

modules from the host machine. When complete, a directory of the modules is also acquired.

At this point, all of the Sombrero modules selected for instantiation are assigned memory so that global pointers can be assigned, one to each module, and any heaps can be initialized. This is a complex process involving processing the symbol table of the object files to resolve pointers and discover the memory requirements of the different sections of the object files.

When all is complete the Boot Loader currently invokes the remote debugger on the W2K Host by issuing a command to the host to start the debugger. It is done at this stage so that the debugger can participate in the instantiation sequence.

With the debug user interface now started on the host, the user has complete control of the remaining process. Typically breakpoints are set and then the Boot Load constructor on the Target Alpha machine is called from the debug control panel on the W2K Host.

7.1 Boot Construction Phase

In the construction phase each module is initialized in turn so that it is not initialized before a service module on which it is dependent. In each case a module's static constructors are called and then its standard constructor. Static constructors are the set of constructors for any C++ class objects that may have been declared as static variables. The standard constructor is the main constructor and gives the programmer an optional opportunity to run some code at instantiation time.

The linker SOSLinker is constructed first to provide run time name resolution to the rest of the boot sequence. Next, library modules are initialized to provide string, tree, queue, heap access, and other kinds of support. The remaining order of construction is SOSMem for memory allocation, SOSAccess to assign access descriptors to the memory objects, SOSThreads for thread allocation, SOSProtection for Protection domain creation, SOSInterrupt to manage hardware interrupts, SOSScheduler to establish time quantum, and an idle thread. Finally, all remaining constructors are called including constructors for SOSNet that provides services for network access and is temporarily the Sombrero side of the remote debugger interface.

After module construction is completed all Memory Objects are associated with access descriptors and protection domains if they were not already. Finally, fault-handling threads are assigned for RPLB misses and the RPLB is enabled. The last step identifies the Target Alpha machine user (or super user in UNIX parlance).

When all is completed control is transferred to the remote debugger in the SOSNet module on the target Alpha to wait on the user on the W2K host for a command. The Boot Loader is now retired.

8 CONTRIBUTIONS AND REMAINING WORK

A great deal of work remains for this experimental Sombrero implementation. Existing code needs to be exercised by building on top of it to reveal further issues, a process referred to earlier in this document as differentiation. New code needs to include the extension of the pager to a persistent store and the distribution of the VA space. Finally, a document will be produced that formally proposes the specification for the Sombrero implementation.

8.1 Contributions

Specific contributions made by this research include the development of ideas that have been suggested or modeled elsewhere as well as original material. This topic is presented here, near the end of the dissertation, to allow the full use of the vocabulary developed in earlier sections. Contributions include the following items:

- Protection Model – The protection model used in Sombrero is based on the legacy protection model found in TLBs and the extensions suggested by Kolding [1992] for the PLB in a single address space. I propose a protection model called the RPLB, which is further extended, compared to the PLB, to provide object based rather than page based protection using region masks. This has been implemented in emulation and supported in system software.
- Carrier Protection Domain – A protection domain associated with the thread of execution. I propose that the state associated with a thread of execution be extended to allow it to access its own data store no matter what GPD it is executing in. This is

important in trusted relationships to reduce the overhead associated with giving a thread access to its stack or other buffers regardless of the GPD association. CPDs are not allowed to have execute access except for associated GPD switches. This has been implemented.

- Domain Switching – Domain switching is a well-understood concept but is usually performed explicitly. I propose an implicit domain switch built into the RPLB that is costless. Use of an implied domain switch is a major contributor to program simplification in Sombrero. This too has been implemented in hardware emulation and supported in system software.
- Kernelless Architecture – Making use of the new protection model to provide the boundary between system and user allows a simpler model for system design. I propose a non-existent or nearly non-existent kernel. The absolute need for a kernel or something like it such as PAL Code has yet to be determined, but it is already apparent that the kernel can be greatly simplified in a SASOS. Sombrero is built around the kernelless architecture.
- Binding Hardware Resources to Protection Domains – I propose binding CPU and other hardware resources to individual protection domains to provide the protection normally provided by a kernel. This has been emulated.
- Programmable Hierarchies – By using the Object model, including inheritance and access to base classes, I propose a system model that can be molded via policy management to suit a particular environment such as real time or embedded systems

- vs. a general computing environment. Inheritance has not been fully tested yet nor has the policy mechanism been fully implemented.
- **Reduced Complexity** – Compared to traditional MASOS systems and even proposed SASOS systems, I have carefully tried to take advantage of the single name space to fold complexities that are normally addressed in the program model into the default behavior of the operating system. I'm sure I have missed a few opportunities that remain to be found, but this effort has been largely successful at reducing complexity.
 - **Entry/Return Point Mechanism** – I propose an entry/return point mechanism that supports both a local instantiation context switch as well as a protection domain switch. The entry/return point is designed to be unaware of its participation in a GPD switch, relying on the *implied* GPD switch mechanism, so that it can operate the same in interdomain or intradomain services requests. This generic behavior contributes to reduced complexity. The entry/return point mechanism is operational.
 - **Tail switch** – I propose the Tail Switch, an implied switch that provides an automatic return switch to a protection domain from a service in another protection domain. Tail Switches are pushed at entry points only when an actual GPD switch occurs, remaining transparent to the program. Tail Switches are popped at return points with the same advantage of transparency as GPD switches. Tail Switches have been successfully implemented in Sombrero.
 - **Compiler Support** – I successfully provided compiler support to allow the C++ program language model to be used to program access to external services via entry/return points.

- Semaphores and Locks – I successfully provided semaphores and locks that take advantage of the single address space. These do not need to be registered in the operating system to be globally available.
- Passive System Services – Instead of passing data between threads passive services are animated directly by user threads. This avoids both the thread communication protocol and a thread context switch. Passive services are not new to Sombrero but Sombrero makes more extensive use of the concept by allowing system services to be passive subject to trust relationships.
- Blocking Interrupts – I propose the use of blocking interrupts that allow threads to block in a registered GPD and provide the context for handling the interrupt. Return from interrupt is a context switch back to the original user thread. This allows user threads to block directly in device handlers supporting the concept of passive services. Blocking Interrupts have been implemented.
- Distributed Consistency in a SASOS – I proposed a mechanism to manage consistency and distribute data across the network on an object based rather than a page based granularity. Pages still provide the basic granularity of storage, but pages are not individually tracked rather the objects associated with the pages. Referred to as token tracking; not yet implemented.
- Surrogate Pointers – I proposed the use of surrogates for system control blocks that would allow network routing to be based on the address of the pointers to the surrogates; not yet implemented.

8.2 Remaining Work

Middle level issues that remain to be addressed include the development of the user interface, a system of directories, an upper level access policy store to handle PRAL misses, extension of the network driver, implementing destructors, building a compiler to run directly on Sombrero, experiments with fault tolerance on a single node and then in the distributed address space, exception handling, management of policy hierarchies, extension of the debugger, completion of the versioning system for obsolescing run time links and linking to the correct version of an entry point, experiments in data and thread migration, and others not listed. It is useful to describe some ways the implementation of a few of these issues can be done.

8.2.1 User Interface, Access Policy, and Directories

The user is the focal point of access policy and has a need for symbolic references such as a directory. Each user, on any system, has a view of resources to which they are permitted access. Users also have organizational relationships to other users, typically a hierarchy of groups of users. For example, a group of departments, where each department has a group of employees, can form a hierarchy. Sub hierarchies can be formed within the departments. It seems fair to experiment with allowing users (a user is an instance of a `SOSUser` instantiation) to create users and to associate policy inheritance with these hierarchies. More interestingly, binding directories to users and allowing users to inherit access to higher-level directories, as with access policy, may ultimately prove more scalable giving each user only what they need to see. Access misses in the PRAL and directory searches can be associated with a user and its inherited rights and

directories, limiting the scope of what must be searched. Most common popular systems have a flat namespace for the users. Scaling requirements are forcing more innovative approaches producing forests and trees (Microsoft Enterprise Server).

The user interface needs to include the manner in which the user can interact with Sombrero. The one currently operational interface is the remote debugger. However, the connection between the host and Sombrero was designed to allow a Sombrero application to launch a windows interface on the host. This has only been used with the debugger so far, but when it is fully operational Sombrero programs can interact with host GUI programs to provide a flexible user interface. A command line console is a good first step.

8.2.2 Versioning System

A versioning system on Sombrero has two goals. One to allow the run time linker to resolve access to the correct entry points, the other to allow an instantiation of a program to be destroyed and reinstantiated, possibly at a new address requiring users to re-resolve entry point links. Callers will be able to correctly detect the need to resolve links in either case. This means that callers must detect that an entry point pointer has become obsolete and call the run time linker to update the pointer to the correct new address. The versioning system, as it stands, is incorporated in the proxy classes and exit function stubs. A master version number at the head of the global table informs callers that links are obsolete. The mechanism has not yet been tested.

8.2.3 Fault Tolerance

Fault tolerance on a single node is needed to provide a workable level of operation to take full advantage of the persistent store. Various lines of defense can be drawn depending on the nature of the program and data. The first line of defense toward fault tolerance is the protection service already implemented. This provides a barrier that isolates most kinds of failures in the VA space. A failure in a single non-critical GPD will allow that domain to be discarded and its resources reclaimed analogous to killing a process in UNIX. Next, GPDs are thread independent so if a thread has an exception, but has not corrupted the service it is executing in, the thread can be discarded without consequence to the service.

For more critical conditions the service may need to store its data by submitting it to a data service that flushes its dirty pages back to the persistent store only when it is consistent. A consistent database would allow a failed service to be rebuilt from its last consistent state. Two phase commit, journaling, etc. are techniques that can be used to assure the consistent state of an update while at the same time providing open memory access to read only operations.

Protection of the memory object control blocks and the GPD control blocks is even more critical since these actually define the address space. The thread control blocks can probably be reconstructed as needed. Submission to a data service should work for the critical control blocks as well though we don't yet know what all of the issues will be.

The final most critical issue is the preservation of the persistent store page tables themselves. Unpredictable corruption in the page tables would destroy the contents of the VA space itself similar to the effects of directory corruption in a file system.

It should be noticed that as the level of fault tolerance rises the complexity of programs will increase though it can probably be assumed that consistent store to a data service will be less complex than use of a file system. This can be assumed because the data does not need to be flattened (replacement of pointers and other VA dependent features) and for read only purposes it can still be accessed directly.

9 FUTURE USES FOR SOMBRERO

Should Sombrero eventually prove sufficiently stable to enter a commercial phase I believe the first best step would be to emulate popular hardware, such as the x86 processor, on top of Sombrero to enable the installation of other operating systems and their applications. This would have the immediate effect of distributing these operating systems and their applications across a network of Sombrero computers. While this may involve some licensing issues it will enable legacy applications to take on the Sombrero properties of distribution, load balancing, and data migration. Further development for such systems can then take place in the Sombrero environment with its reduced complexity and cost benefits.

Areas where Sombrero may prove useful include large distributed applications involving many interacting participants, such as an engineering project with its component database. The reduced complexity should allow updates to the component database to occur more efficiently using permanent pointers for data classification. This would reduce delays and the probability of programming errors.

Rendering farms that produce the graphics for the animation industry would certainly benefit from a distributed common VA space in which to carry out very large computations. Web server farms can benefit in much the same way.

Embedded and real time systems should benefit from reduced complexity to meet time constraints through improved performance. Likewise protection domains contribute

by adding stability through protection without the use of the more complex MASOS processes.

Remote systems such as satellite systems can take advantage of the large multiprocessor model. By providing an array of processing units with multiple communication paths, processing units can fail and be removed from operation without losing service. The overall program would continue to run without interruption allowing a graceful degradation.

10 CONCLUSIONS

Typical issues that traditionally add to the complexity of program design in the MASOS paradigm are folded into the default behavior of Sombrero relieving the application developer of the necessity of dealing with them. The complexities of application design and implementation no longer required in Sombrero include the file system, Inter-Process Communication, distributed communication, data and execution migration, and CPU load balancing. This is made possible by building the operating system over a common substrate, the VA name space, on which supporting algorithms can be implemented without the need to perform multiple and unwieldy name space translations.

In its current state the Sombrero modular system design and the supporting protection mechanism have been largely realized.

REFERENCES

- Advanced Micro Devices, Inc., “AMD x86-64 Architecture PROGRAMMER’S MANUAL, Volume 2, SYSTEM Programming”, Publication # 24593 Rev: 3.00, January 2002.
- Atkinson, M. P., Bailey, P., Chisholm, K. J., Cockshott, W. P., and Morrison, R., “An Approach to Persistent Programming”, *The Computer Journal*, (26) (4), pp. 360-365, Nov. 1983.
- Carnes, M., “Sombrero System Interface”, MCS Project, Computer Science and Engineering Department, Arizona State University, December 1996.
- Chase, J., "An Operating System Structure for Wide-Address Architectures", *Tech. Report 95-08-06*, Department of Computer Science, University of Washington, August 1995, Ph.D. Dissertation.
- Digital Equipment Corporation, *The Alpha Evaluation Board Software Developers Kit (EBSDK)*, Release 2.1 Beta 5, August 1996.
- Digital Equipment Corporation, *Digital Semiconductor 21164 Alpha Microprocessor Hardware Reference Manual*, Order Number: EC-QP99B-TE, January 1997.
- Deller, L., Heiser, G., “Linking Programs in a Single Address Space”, *Proceedings of 1999 USENIX Annual Technical Conference*, Monterey, CA, USA, June 1999, p 283-294.
- Feigen, R., “Reduction of Software Development Costs under Sombrero, A Single Address Space Distributed Operating System”, Computer Science and Engineering Department, Arizona State University, December 2001, MS Thesis.
- Feigen R., Skousen A., Miller D., “Reduction of Software Development Costs under the Sombrero Distributed Single Address Space Operating System”, *The 2002 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2002)*, Las Vegas, NV, USA, June 2002, p 127-134.
- Khatri, S., “Protection Structures in the Sombrero Operating System”, Computer Science and Engineering Department, Arizona State University, December 1997, MS Thesis.
- Koldinger, E. J., Chase, J. S., and Eggers, S. J., "Architectural Support for Single Address Space Operating Systems", *Proceedings of 5th International Conference on Architectural Support for Programming Languages and Operating Systems, ACM SIGOPS Operating Systems Review*, October 1992.
- GNU Compiler Collection, GNU Project, Free Software Foundation,
http://www.gnu.org/directory/Software_development/Compilers/gcc.html, 2002

- Olson, J. "Distributed Consistency Management in Sombrero, a Single Address Space Distributed Operating System", Computer Science and Engineering Department, Arizona State University, December 2002, MS Thesis. (To be published).
- Patil, M. H., "Distributed Scheduling in Sombrero, A Single Address Space Distributed Operating System", Computer Science and Engineering Department, Arizona State University, December 1999, MS Thesis.
- Soltis, F. G., *Inside the AS/400*, Duke Press, 1995.
- Skousen, A. C., "Sombrero: A Very Large Single Address Space Distributed Operating System", Department of Computer Science and Engineering, Arizona State University, MS Thesis, December 1994.
- Skousen, A. C. and Miller, D. S., "Resource Access and Protection in Sombrero: Protection Model, Software Protection Data Structures and Hardware Range Protection Lookaside Buffer, ASU 64-bit OS Group WP 2", Computer Science and Engineering Department, Arizona State University, *TR-96-013*, May 1996a.
- Skousen, A. C. and Miller, D. S., "Implementing a Single Very Large Address Space across Multiple Nodes: Memory Partitioning, Protection Domain Migration, Kernel Replication, Consistency and Fault Tolerance, ASU 64-bit OS Group WP 10", Computer Science and Engineering Department, Arizona State University, *TR-96-021*, May 1996b.
- Skousen, A. and Miller, D., "The Sombrero Distributed Single Address Space Operating System Project", *2nd USENIX Windows NT Symposium*, August 1998c, p. 168.
- Skousen, A. and Miller, D., "Operating System Structure and Processor Architecture for a Large Distributed Single Address Space", *Proceedings of PDCS'98: 10th International Conference on Parallel and Distributed Computing Systems*, October 1998d.
- Torla, M. J., "RPLB Design Project Report", MCS Project, Computer Science and Engineering Department, Arizona State University, July 1997.
- Vochtelo, J., "Design, implementation and performance of protection in the Mungi single-address-space operating system", UNSW, 1998, PhD Thesis.

APPENDIX A

PUBLICATIONS IN SUPPORT OF THE SOMBRERO PROJECT

A.1 THE SOMBRERO DISTRIBUTED SINGLE ADDRESS SPACE OPERATING SYSTEM PROJECT

Published:

A. Skousen and D. Miller, *2nd USENIX Windows NT Symposium*, August 1998, page 168.

A.2 OPERATING SYSTEM STRUCTURE AND PROCESSOR ARCHITECTURE FOR A LARGE DISTRIBUTED SINGLE ADDRESS SPACE

Published:

A. Skousen and D. Miller, *10th IASTED Parallel and Distributed Computing Conference (PDCS98)*, October 1998, pages 631-634.

ABSTRACT

Recent 64-bit microprocessors have made a huge 18.4 quintillion byte address space potentially available to applications and the operating system. Because current process-oriented operating systems and their underlying page-based protection model were designed to conserve virtual addresses by reusing them, this four billion-fold increase in address space has led to investigations of alternative models. The fundamental issue is how to structure an operating system so that it provides a simple program development environment and high performance in a computer system that supports a very large address space where address space conservation is no longer a driving issue. One approach is to establish a *single* network-wide address space in which all code and data reside, and then design an operating system specifically to run in this environment. This paper presents a model and the design of a Single Address Space Operating System (SASOS) called Sombrero. It includes a design of proposed changes to CPU protection and memory management architecture to support its abstractions.

A.3 USING A DISTRIBUTED SINGLE ADDRESS SPACE OPERATING SYSTEM TO SUPPORT MODERN CLUSTER COMPUTING

Published:

A. Skousen and D. Miller, *Proceedings of the 32nd Hawaii International Conference on System Sciences (HICSS-32)* January 1999.

ABSTRACT

Recent 64-bit microprocessors have made a huge 18.4 quintillion byte address space potentially available to programs. This has led to the design of Operating Systems that provide a single virtual address space in which all code and data reside in and across all levels of storage and all nodes of a distributed system. These operating systems, called SASOSs, have characteristics that can be used to support modern cluster computing in a distributed system in ways that provide an improved program development environment and higher performance than available from conventional operating systems. Sombrero, our SASOS design, makes use of its hardware support for object-grained protection, separate thread related protection domains and implicit protection domain crossing to provide support for modern cluster computing not available in SASOSs built on stock processors. Its design, which provides direct system level support for object oriented programming, includes a number of features targeted specifically at modern cluster computing.

A.4 USING A SINGLE ADDRESS SPACE OPERATING SYSTEM FOR DISTRIBUTED COMPUTING AND HIGH PERFORMANCE

Published:

A. Skousen and D. Miller, *18th IEEE International Performance, Computing, and Communications Conference*, February 1999, pages 8-14.

ABSTRACT

Recent 64-bit microprocessors have made a huge 18.4 quintillion byte address space potentially available to programs. This has led to the design of Operating Systems that provide a single virtual address space in which all code and data reside in and that spans all levels of storage and all nodes of a distributed system. These operating systems, called SASOSs, have characteristics that can be used to support synchronization and coherency in a distributed system in ways that provide an improved program development environment and higher performance than that available from conventional operating systems. Sombrero, our SASOS design, makes use of its hardware support for object-grained protection, separate thread related protection domains and implicit protection domain crossing to provide synchronization and coherency support for distributed object copy set management not available in SASOSs built on stock processors. Its design, which provides direct system level support for object oriented programming includes a number of system architectural features targeted specifically at modern distributed computing environments.

A.5 THE SOMBRERO SINGLE ADDRESS SPACE OPERATING SYSTEM PROTOTYPE A TESTBED FOR EVALUATING DISTRIBUTED PERSISTENT SYSTEM CONCEPTS AND IMPLEMENTATION

Published:

A. Skousen and D. Miller, *The 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000)*, June 2000.

ABSTRACT

The Sombrero distributed single address space operating system prototype is currently being implemented. The prototype includes an emulation of CPU-resident protection hardware necessary to fully utilize the properties of a single address space. The primary objective of the Sombrero project is to demonstrate that software development costs are sharply reduced in a single address space environment with no reduction in performance. This paper presents Sombrero design decisions and implementation mechanisms developed in order to support protection, communication, resource access and control transfer across a very large persistent distributed address space. Included are changes to operating system architecture, evolution of programming types and a new approach to distributed consistency management in a single address space based-DSM system that is being integrated into the prototype.

A.6 REDUCTION OF SOFTWARE DEVELOPMENT COSTS UNDER THE SOMBRERO DISTRIBUTED SINGLE ADDRESS SPACE OPERATING SYSTEM

Published:

R. Feigen, A. Skousen and D. Miller, *The 2002 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2002)*, June 2002.

ABSTRACT

Sombrero is a distributed single address space operating system (SASOS). DSM and persistence are part of the fundamental nature of a distributed SASOS because the common virtual address space is distributed, shared and persistent. We believe that the major reason for using a SASOS is that many useful applications would be less complex if written to run on a SASOS rather than under a conventional process-oriented multiple address space operating system (MASOS). This paper examines the potential for reduction in software complexity of applications developed under the Single Address Space Operating System Sombrero versus those developed under a conventional process-oriented operating system, Windows 2000. To test Sombrero's impact on development costs, a set of sample database applications with varying functionality was developed under Windows 2000 and Sombrero. Using accepted software engineering metrics, the software complexity of these systems was compared. The results obtained indicate a substantial reduction of software complexity and software development costs in a single address space environment such as that provided by Sombrero. Reasons for this based on the code required to perform the same tasks in the Sombrero and Windows 2000 environments are presented.