The Sombrero Single Address Space Operating System Prototype
A Testbed for Evaluating
Distributed Persistent System Concepts and Implementation

Alan Skousen Donald Miller
Computer Science and Engineering Department
Arizona State University, Tempe, Arizona 85287, USA
E-mail: {alan.skousen,donald.miller}@asu.edu

Keywords: single-address-space operating systems, distributed operating systems, protection, persistent systems

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.

1. Introduction

One way to realize a distributed persistent system is to implement it with a Single Address Space Operating System (SASOS). Sombrero [1] [2] [3] is a SASOS in which a CPU-resident hardware protection cache especially designed so that an operating system can take advantage of the properties of a single address space provides the underlying protection. The Sombrero prototype is currently being completed [4]. Providing protection in hardware within the CPU makes possible a very general and simple protection domain implementation that eliminates the need for conventional capabilities to control access. The major benefit from this architecture is a considerable saving in the cost of software development which accrues from simplifications in the programs needed for communication, resource access and control transfer. Performance can be expected to increase modestly also because of these simplifications, but this is not the primary goal.

SASOSs by their nature provide a single persistent namespace that is a natural substrate for distribution of execution and data across a LAN of homogenous workstations. In a single address space virtual addresses (VAs) can be permanently and uniquely bound to all data and code objects, i.e., VAs are unique names. The VA namespace spans all levels of the storage hierarchy on every node. Since the VA namespace is directly manipulated by CPU memory and protection management hardware, the CPU can *directly* enforce access policy on all objects in the system. In contrast, in traditional process-oriented operating systems, e.g., UNIX and NT, each process runs in it's own VA space, making the reused VAs context dependent. Since addresses don't mean the same thing in different processes, communication, resource sharing and execution migration become much more complicated.

Most contemporary SASOSs have been designed to run on stock RISC 64-bit processors. These include Mungi [5] and Opal [6]. This means additional software communication protocols are required for protection. Typically capabilities are used, requiring operations in one or more additional namespaces. SASOSs with

protection and memory management hardware targeted at single address space operation include AS/400 [7], Monads [8] and Sombrero. Hardware support enables and/or significantly facilitates a flat modular system architecture based on a single page table where operating system services, environment servers and user programs are accessible to each other via ordinary procedure calls and where provision of all hardware protection is via one general mechanism. Protection can be provided at object rather than page granularity, providing system-level support for object-oriented programming and simplifying consistency management in a distributed system. Hardware support enables implicit protection domain crossing at all levels, simplifying programming and increasing performance.

2. Sombrero

Sombrero's basic abstractions are memory objects, protection domains, threads and principals [1]. There are general protection domains (GPDs) in which code and data and other GPDs are accessible to all threads within it and carrier protection domains (CPDs) that contain Data and GPDs privately accessible to an individual thread. Sombrero includes the design of CPU-resident protection hardware called a Range Protection Lookaside Buffer (RPLB) [2]. This hardware supports implicit protected resource access including protection domain crossing with ordinary instructions.

To allow this implicit protected access it's necessary to cache memory access and domain switching authorizations within the CPU in a way that the protection checks can be integrated into the instruction pipeline. To do this requires a cache for protection information and a pair of registers containing virtual addresses identifying the current general protection domain and a carrier protection domain associated with the currently executing thread. The cache needs to store triplets of the form:

<protection domain tag, resource accessed, access rights>

where the protection domain tag and resource accessed, which can be either a memory region or another protection domain, are represented by virtual addresses and the access rights are {*rwxs*} where *s* means a domain switch is allowed. The Sombrero RPLB is a protection cache that provides these features. The hardware functionality provided by the Sombrero RPLB includes: separation of address translation and protection functions, hardware caching of allowed protection domain crossings, protection domains for threads distinct from the general protection context, implicit domain crossing using ordinary instructions and protection definition for variable granularity object sizes. Essentially the RPLB contains the CPU-level representation of protection information contained in an access matrix. By doing so it co-locates in protection triples the same information normally contained in ACLs and capability lists. Software protection data structures resident within the protection service, cache the data needed to convert between user-level policy information in the form of resources, access rights and users, e.g., files, {rwx} privileges, user ids and resource ownership, and the CPU-level representation of these things in terms of protection domains, virtual address and access privileges.

As the Sombrero prototype implementation progressed, it became clear that a SASOS should have a *completely* flat but modular system structure. Because Sombrero can take advantage of the full set of characteristics of a single address space it can be designed as a set of peer level services. Hardware caching of protection domain object regions, allowable domain crossings and hardware-supported private thread protection domains are crucial to this structure. They allow Sombrero service requests by applications and upcalls by executive services to be implemented by ordinary procedure calls and also permit simple migration of control across machines.

Communication between the executive and user modules is the same as that between user programs. System calls implemented with trap instructions and the context switches associated with operating system intervention as occur in process-oriented operating systems are not required. Privileged instructions are bound directly to user modules so that Privilege level as defined by one or two mode bits in modern processors (e.g., kernel/user) is not needed. Operating system services and user programs run at the same privilege level. There is no central kernel. Services, implemented as

protection domains, are modules within the executive and can be replicated or migrated across machines. Lock registers, accessed via privileged instructions limit access of CPU resources to designated protection domains that provide the associated executive services. This removes the need for the orthogonal protection currently supplied by kernel/user bits replacing it with a more general mechanism that applies to all protection domains. As shown in Figure 1, a sketch of part of such a structure, each software module exists at the same CPU privilege level but is guarded by access policy, i.e., who can access what, as determined by protection information cached in the software protection data structures and the RPLB. For compatibility with available hardware our current implementation makes use of PAL mode to limit access to CPU resources to designated executive protection domains.

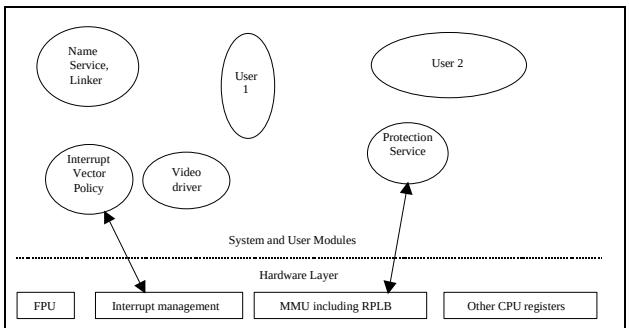


Figure 1. Sketch of Completely Peer-Level Single Address Space Operating System Structure. Arrows represent lock registers.

The Sombrero executive modules are instantiations of object classes that are implemented as GPDs as are all Sombrero programs. These services' methods can be extended and overridden by other services as established by a path in the name-service for the object instantiation linker. The name-service provides symbolic representation for memory objects, protection domains and threads as well as a naming hierarchy for establishing conventional directories and linker information.

3. Sombrero Prototype

The Sombrero prototype is nearly completed on two sets of two Alpha 21164 boxes. The prototype will contain a subset of the full design. The semantics of the interface to this subset are the same as anticipated in the final version. Sombrero applications can be run and the effects of the hardware supported single address space paradigm on the requirements and benefits of distributed persistent systems for software engineering can be evaluated. The actual implementation has a number of constraints. As there is currently no processor with an RPLB, the prototype has to emulate one, depending on the underlying TLB to provide initial protection faults. Time and manpower restrictions necessitate using an existing operating system, Windows NT, for file system and windowing. The reachable address space has been extended by specializing an NT process to include the full range available to the processor. This is done by modifying the NT PALCode to recognize a Sombrero process and forwarding TLB misses outside the normal NT range to a Sombrero extension of the PALCode. A pager in NT user space forms the correct TLB entry and handles the miss. With this structure available we extended the PALCode to emulate an RPLB that is triggered by misses in the TLB. Using this emulated RPLB hardware, individual threads are assigned to separate protection domains within the specialized process. This implementation provides a functional protected single address space similar to that which sombrero would provide.

4. Sombrero SASOS Benefits for Software Engineering

We are currently investigating whether and to what extent there is an inherent reduction in complexity of applications under the Sombrero

SASOS [9]. As indicated above, data structures and services typically available as a system service, via a context switch to kernel mode and/or message passing or remote procedure call in a process-oriented operating system, appear as methods and attributes of local objects in Sombrero. Sombrero is designed to have the appearance of a single large multithreaded process distributed over many nodes. As such, many system services can be accessed as if the service was part of the user-level application. The net result is that in most cases an access to a system service or a resource in another process is reduced to a local subroutine call. The efficacy of such an environment is to reduce the complexity of programs by replacing IPC with inter-domain communication (IDC) and hence reducing managing communication to managing protection. This provides the application level developer with new and often powerful (native) system services and abstractions. This environment also provides improved standard system service abstractions that minimize the effort and expertise required for developing many applications.

A comparison of the code required in a process-oriented client/server system for a file copy, [10] with that to perform the equivalent operations in a SASOS programming environment is shown in Figures 2 and 3. A few observations can be made about the differences in programming costs and performance in the two environments. In the process-oriented system, the message format shown in the header file, used to linearize or flatten the data, requires programmer time for argument specification and CPU time to execute the run-time marshalling/unmarshalling operations. The client and server must exchange two messages for each block of data copied. Client code requires different operations and naming conventions for accessing memory and secondary storage. File server code is actually much larger than shown. The reading, writing, creating and deleting routines consist of hundreds of lines of file system code not required in a SASOS and also take significant CPU time to execute.

In the SASOS program the *address* function calls the name service to resolve the name to a virtual address and return a data structure that contains the object's address and size. No *open*

//Header file used by the client and server:

```
/* Definitions needed by clients and servers. */
#define MAX_PATH  255        /* maximum length of a file name */
#define BUF_SIZE   1024      /* how much data to transfer at once */
#define FILE_SERVER  243         /* file server's network address */

/* Definitions of the allowed operations. */
#define CREATE        1 /* create a new file */
#define READ          2 /* read a piece of a file and return it */
#define WRITE         3 /* write a piece of a file */
#define DELETE        4 /* delete an existing file */

/* Error codes. */
#define OK            0 /* operation performed correctly */
#define E_BAD_OPCODE  -1       /* unknown operation requested */
#define E_BAD_PARAM   -2       /* error in a parameter */
#define E_10       -3     /* disk error or other I/O error */

/* Definition of the message format. */
struct message {
    long source;            /* sender's identity */
    long dest;          /* receiver's identity */
    long opcode;            /* which operation: CREATE, READ, etc. */
    long count;             /* how many bytes to transfer */
    long offset;            /* where in file to start reading or writing */
    long extra1;            /* extra field */
    long extra2;            /* extra field */
    long result;            /* result of the operation */
    char name[MAX_PATH];        /* name of the file being operated on */
    char data[BUF_SIZE];        /* data to be read or written */
};
```

// Process oriented file SERVER:

```
#include <header.h>
void main(void)
{
  struct message m1, m2;            /* incoming and outgoing messages */
  int r;                            /* result code */

 initialize();
 while (1) {
    receive (FILE_SERVER,&m1);       /* server runs forever */
    switch(m1.opcode) {
      case CREATE:  r = do_create(&m1, &m2);  break;
      case READ:    r = do_read(&m1, &m2);  break;
      case WRITE:   r = do_write(&m1, &m2);  break;
      case DELETE:  r = do_delete(&m1, &m2);  break;
      default:      r = E_BAD_OPCODE;
    }
    m2.result = r;                  /* return result to client */
    send(m1.source, &m2);           /* send reply */
  }
}
```

//Process oriented CLIENT uses file server to copy a file:

```
#include <header.h>
int copy(char *src, char *dst)          /* procedure to copy file using the server */
{
  struct message m1;                /* message buffer */
  long position;                    /* current file position */
  long client = 110;                    /* client's address */
            /* prepare for execution */
  position = 0;
  do {
    /* Get block of data from source file */
    m1.opcode = READ;               /* operation is a read */
    m1.offset = position;           /* current position in the file */
    m1.count = BUF_SIZE;            /* how many bytes to read */
    strcpy(&m1.name, src);          /* copy name of file to be read to message */
    send(FILE_SERVER, &m1);             /* send message to the file server */
    receive(client, &m1);           /* block waiting for the reply */

    /* Write the data just received to the destination file */
    m1.opcode = WRITE;              /* operation is a write */
    m1.offset = position;           /* current position in the file */
    m1.count = m1.result;           /* how many bytes to write */
    strcpy(&m1.name, dst);          /* copy name of file to be written to buf */
    send(FILE_SERVER, &m1);             /* send the message to the file server */
    receive(client, &m1);           /* block waiting for the reply */
    position += m1.result;          /* m1.result is the number of bytes written */
  } while (m1.result > 0);              /* iterate until done */
  return(m1.result >= 0 ? OK : m1.result);  /* return OK or error code */
}
```

Figure 2. File system server and client code for a file copy in a process-oriented Client/Server system.

```
//SASOS Memory object copy
int copy(char *src, char *dst)
{
  FILE_OBJECT *from, *to;
  //address() is a function that obtains an object's address from the nameservice
  from = address(src);
  // allocate a new memory object
  to = new FILE_OBJECT(dst, PERSIST, from->Size());
memcpy(to,  from,  from->Size());
}
```

Figure 3. Client/Server file copy performed as a memory object copy in a SASOS programming environment.

is required to set up a channel of data structures to the file within the operating system. *address* performs the SASOS equivalent of a directory lookup. *new* invokes a constructor function that creates a persistent memory object providing it the size of the object. The SASOS *memcpy* directive invokes a standard C library routine that copies blocks of memory from one location to another. Ultimately, to get to persistent store, disk storage must be accessed by the SASOS. However, this is transparent to the application and, as the secondary storage namespace is also the virtual address space, the access can be done with low level memory mapped I/O primitives assisted by an indexed inverted page table to convert virtual addresses to disk sectors. For distribution, virtual addresses can be used for network routing.

It can be seen that many functions now considered client/server operations are replaced by standard library operations in a SASOS. "Under the hood" a SASOS handles these efficiently by retrieving VA space descriptors across the net, transparently migrating code and data as required and performing the necessary VA space operations. In a SASOS, because threads can migrate, servers can be constructed as passive entities. They can exist in a separate protection domain on another node and, in a hardware supported SASOS, be entered and activated by remote clients' threads via ordinary procedure calls. Hardware support for a separate CPD and GPD adds flexibility to server design. Servers can be designed to either retain client information within their own GPD or to operate on this information when it is introduced as part of the calling client's CPD or both. In the former case, a copy of the server GPD would migrate to the client node. In the latter case, the client thread and its messages and other initialization data contained in a carrier protection domain can migrate to the server's node or server code can be cached on the client node for efficiency.

5. Sombrero DSM Consistency

The Sombrero design contains a distributed system of user-definable consistency semantics on a resource by resource basis called *Token Tracking*. Token Tracking refers to services that enforce consistency semantics between copies of objects on different nodes. It allows distributed resource management to occur at a very low level, the level at which resources are created. Object grained protection hardware directly supports object regions rather than pages as the base unit of network distribution. It also enables per-object copy-set management to replace the per-page copy-set management found in process-oriented systems and SASOSs built on stock hardware. Network data transmission granularity is therefore determined by access policy rather than by operating system virtual address translation mechanisms. Even more important, because the hardware enforced protection triples (PD, VA, access rights) are invariant and available throughout the network-wide virtual address space, every virtual address, and every memory object containing it, is tied to a specific access policy. Once established, access policy, i.e., who can access what and when, is implicit and transparently distributed across the network by the protection hardware resident on every node. Local and remote protection faults support object access policy in much the same way as local and remote page faults support memory sharing in conventional DSM systems. And as in DSM, the demand-paged virtual memory management present on every node automatically generates the object region address faults needed. Unlike DSM, object-region granularity enables simple scaleable, per-object data structures and pointers to be used. Token tracking structures contain the list pointers of simple network-wide graphs used to locate the most recent copy of an object's modified page, and control block tokens are used to enable access [3]. Only the data structures in use at a node need be present at it.

These services are a subset of services provided by a *Node Aware Service* [11]. Each object has a Node Aware Service assigned to it to implement the desired consistency policy. A

Node Aware Service is a per-object user policy manager that enforces consistency by maintaining an awareness of an object's state on a given computer node. Exactly when a *rwxs*-token is granted or revoked is a policy decision and is controlled by the Node Aware Service. The default set of Token Tracking services provided by Sombrero enforces a policy of sequential consistency with Single Writer, Multiple Reader semantics. Other consistency semantics such as Concurrent Read Exclusive Write (CREW) and Weak Semantics can be implemented with additional Node Aware Services. To simplify the task of writing Node Aware Services, the default Node Aware Service functions as a base class permitting programmers to build new or modified services by extending the base class. Node Aware services interact with the more privileged Protection Services, typically by invoking methods in memory object control blocks e.g., *set_read_token*, *unset_write_token*. On a protection fault, the copy of the protection service on the node will call the Node Aware service for that object to verify that the access is allowed, and if it is, to initiate the actions necessary to supply the needed page. In summary, Node Awareness services are user authorizing agents for *consistency* policy and the privileged Protection Service implements protection. A third service, the Domain Service acts as an authorized agent to provide *access* policy to the Protection Service.

6. Additional Observations on Distributed Persistent Concepts and Implementation

6.1. Persistent programming languages and type evolution

Thread and protection domain mobility has led us to modify and augment the conventional type scope definitions. In particular, since process scope does not exist in a SASOS, it has been replaced by a combination of global and domain scopes. Thread scope had to be included from the start; it could not be a work-around add-on as is done in run-time systems for process-oriented systems. We modified the C++ *new* to create persistent objects (see Figure 3). Entry point type designators (decorated entry points) were needed for linking. As noted elsewhere [6] [12], linking is significantly different in a SASOS. It's a one-time operation that has consequences on the structure of dynamic link library (DLL) modules. For example, distinct address regions must be allocated for private static data each time a DLL is linked to an object.

6.2. Fault tolerance and recovery strategies

In a single address space every level of storage on every node can be viewed as a cache for the virtual address space. Sombrero takes advantage of this cache hierarchy to place an archival fault tolerant layer at the bottom of the hierarchy. This layer is used for checkpointing portions of the database that are in a consistent state. Fault tolerance in a single address space is no harder than (and no easier than) that in process-oriented systems with a separate file system namespace for persistent storage. Similar logging and private workspace mechanisms apply. The ability for kernel structures to be replicated by surrogates on remote nodes enables Sombrero to provide support for RAIN (Redundant Array of Inexpensive Nodes). The remaining nodes can regenerate a failed node.

6.3. Distributed Scheduling

We have investigated thread scheduling in a distributed persistent system as a mechanism to achieve load balancing [13]. A simulation was constructed in which a Sombrero system was organized into hierarchies of clusters for scalable distributed scheduling. The best scheduling algorithm turned out to be sender initiated thread rescheduling at lower loads and receiver initiated thread rescheduling at higher loads.

7. Conclusions

A hardware-supported SASOS is a suitable vehicle to realize a Distributed Persistent System. It reduces software complexity, which should reduce the costs associated with software development and maintenance. Single address space operation requires extensions and modification to programming language constructs, compilers and linkers, the most

important of which is an extension to type scope attributes necessitated by the removal of encapsulating process boundaries. By its very nature a Sombrero is a superset of a DSM system, enabling simple, efficient and flexible mechanisms to be used for consistency management and client/service migration. The 'single level store' nature of a SASOS facilitates fault tolerant checkpointing.

REFERENCES

[1] 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 1998.

[2] Skousen, A., and Miller, D., "Using a Distributed Single Address Space Operating System to Support Modern Cluster Computing", Proceedings of *32nd Hawaii International Conference on System Sciences, HICSS-32*, January 1999.

[3] Skousen, A., and Miller, D., "Using a Single Address Space Operating System for Distributed Computing and High Performance", *Proceedings of 18th IEEE International Performance, Computing, and Communications Conference*, February 1999.

[4] Skousen, A. and Miller, D., "The Sombrero Distributed Single Address Space Operating System Project", *2nd USENIX Windows NT Symposium*, August 1998, p. 168.

[5] Heiser, G., Elphinstone, K., Vochteloo, J., Russell, S., and Liedtke, J., "The Mungi Single-Address Space Operating System", *Software-Practice and Experience*, VOL. 28(9), July 1998, pp. 901-928.

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

[7] Soltis, F. G., *Inside the AS/400*, Duke Press, 1995.

[8] Rosenberg, J., and. Abramson, D., "MONADS-PC - A Capability-Based Workstation to Support Software Engineering", *Proceedings of the Eighteenth Annual Hawaii International Conference on System Sciences*, pages 515-522, IEEE, 1985.

[9] Feigen, Ron, "Inherent Reduced Complexity of Applications under the Sombrero Single Address Space Distributed Operating System", Computer Science and Engineering Department, Arizona State University, *Masters Thesis Research Proposal*, February 2000.

[10] Tanenbaum, Andrew, *Distributed Operating Systems*, Prentice-Hall, 1995, pages 53 and 55.

[11] Olson, John, "Token Tracking Overview and Implementation Options in Sombrero", Computer Science and Engineering Department, Arizona State University, Internal Report, October 1998.

[12] Deller, Luke, and Heiser, Gernot, "Linking Programs in a Single Address Space", *Proceedings of 1999 USENIX Annual Technical Conference*, Monterey, CA, USA, June 1999, pp. 283-294.

[13] Patil, Milind, "Distributed Scheduling in Sombrero, a Single Address Space Distributed Operating System", Computer Science and Engineering Department, Arizona State University, *MS Thesis*, August 1999.