Sombrero System Interface

Mark Carnes

12/12/96

Table Of Contents

1 Overview 1

1.1 System call structure and philosophy 1

1.1.1 Protection of system wide data structures 1

1.1.2 Building block for higher OS functions 2

1.1.3 Existing OS extension 2

2 Library Interface 4

2.1 Overview 4

2.1.1 Policy and Authorization 4

2.1.2 Gateways and Signal Stubs 5

2.1.3 Transfer of Control 6

2.1.4 System Errors 7

2.1.5 Conventions 7

2.2 SysObject 8

2.2.1 Description 8

2.2.2 C++ Interface 8

2.2.3 C Interface 9

2.3 SysMemory 10

2.3.1 Description 10

2.3.2 C++ Interface 11

2.3.3 C Interface 17

2.4 SysThread 19

2.4.1 Description 19

2.4.2 C++ Interface 19

2.4.3 C Interface 24

2.5 SysDomain 26

2.5.1 Description 26

2.5.2 C++ Interface 26

2.5.3 C Interface 28

2.6 SysPrincipal 29

2.6.1 Description 29

2.6.2 C++ Interface 29

2.6.3 C Interface 31

2.7 SysNode 32
2.7.1 Description 32
2.7.2 C++ Interface 32
2.7.3 C Interface 35
3 OS Extensions Interface 37
3.2 Accessing Traditional OS Services 38
4 Usage Scenarios 39
4.1 Native Programs 39
4.1.1 User Perspective 39
4.1.2 Developer Perspective 40
4.1.3 Administrator Perspective 41
5 Comparison with other system call interfaces 43
5.1 Opal 43
5.1.1 Interface Comparison 43
5.2 Solaris 2.4 46
5.2.1 Command Comparison 46
6 References 64

1 Overview

This document describes the system call interface to the proposed Sombrero Operating System. A good description of the overall architecture and philosophy of Sombrero can be found in [Sk*96a], [Sk*96b], and [Sk*96c]. Sombrero is a single address space operating system. It is also a distributed operating system, where multiple nodes behave as a single, non-uniform access multiple processor machine. All nodes share the same address space. Communication between nodes, and between threads of execution, is accomplished through this shared address space. Special hardware provides protection of address ranges from unauthorized threads.

This simplifies the essential elements of an operating system, eliminating many aspects normally considered part of a traditional operating system and allowing portions to be pushed into user space or to be eliminated.

The Sombrero System Interface is the lowest layer interface to the operating system.

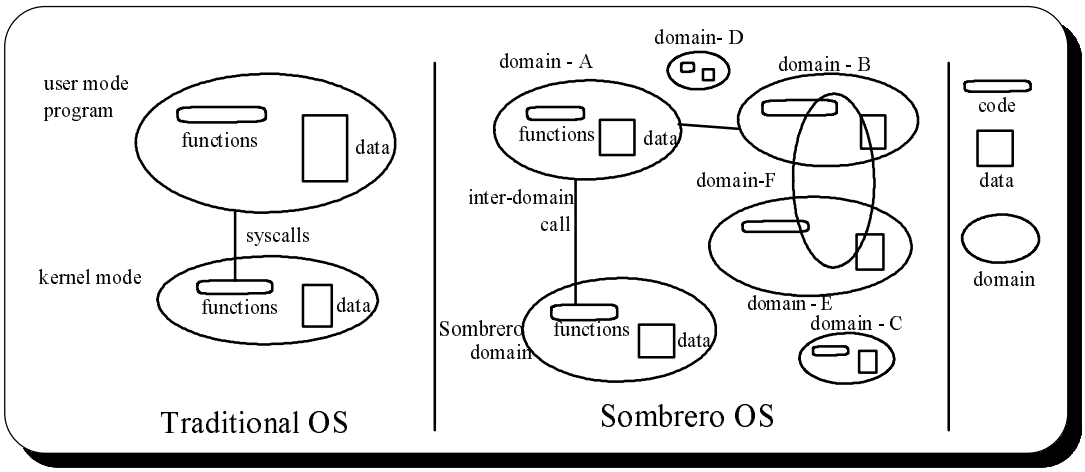
1.1 System call structure and philosophy

The system calls in Sombrero are documented in the form of C++ class definitions. This is a logical representation of these functions since it models the basic data abstractions in Sombrero very naturally. Any language could have an interface to these underlying structures and examples of a C interface are also described in this document.

1.1.1 Protection of system wide data structures

Normally a system call in traditional operating systems causes a switch to a different protection mode called "kernel mode". By having this mode, data structures can be protected by hardware. This concept is extended in Sombrero so that there are any number of "modes" that can be hardware-protected from access by other processes. The mechanism providing this hardware protection implements what is called a "protection domain". The system functions and data structures that are required in the Sombrero system are called the "system interface functions". Using these building blocks, any higher layer of operating system environment can be generated.

The diagram below compares the differences between a traditional OS and Sombrero. The ovals represent protection domains. Note that there is one protection domain per process in a traditional OS, and it is the virtual address space. The only communication allowed between domains is through the kernel. In Sombrero, there are many protection domains and communication can be between any of those domains (including the domain acting on behalf of the Sombrero System Services).

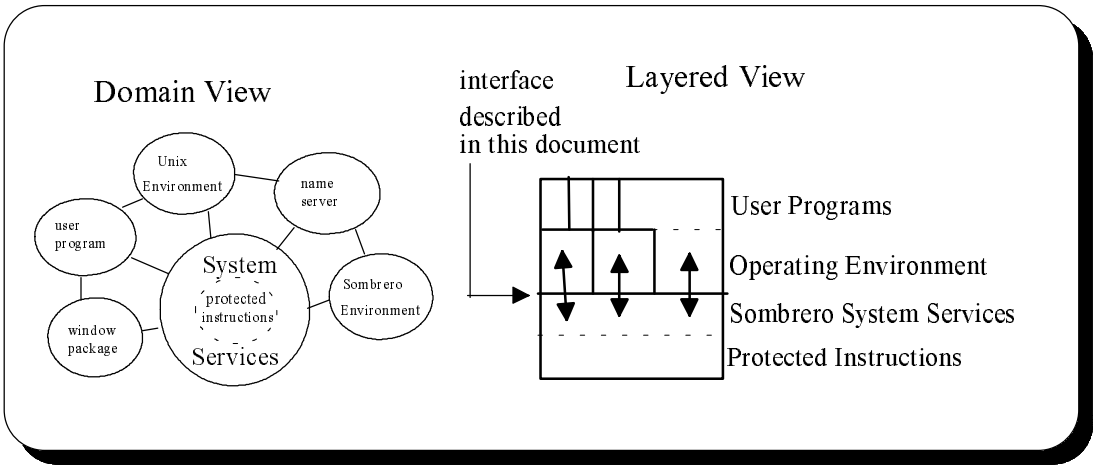


1.1.2 Building block for higher OS functions

The system interface for Sombrero is general and simple. It is not intended to be the standard programming interface for typical user applications. While it can be used by any program, a more likely scenario is that this interface will be used to make a higher level interface that gives more familiar facilities to its users.

The Sombrero OS will undoubtedly have interfaces to OS environments that provide the same programming interface as current legacy OS environments, such as Windows NT or UNIX. This will allow programs to be recompiled and linked for a Sombrero system with little or no code changes. While binary support is conceptually possible, a significant degree of emulation would have to be added to capture the implicit semantics of a many address space operating system. Since Sombrero is designed to be run in 64bit OS environments, the issue of binary compatibility with legacy DOS and Windows programs where no source is available is not as big an issue.

The interface described in this document is the lowest layer accessible to either an operating environment or a user program. An operating environment could be built to allow or deny access to lower layer functions. In all likelihood any operating environment constructed would be built using shared pieces between environments like a windowing package, or name server. The "Sombrero Environment" shown below is an example of an operating environment that shares the name server domain with the "Unix Environment".



1.1.3 Existing OS extension

It is not necessary to have Sombrero be the core operating system, with all compatible operating system functions built above it. An existing OS could also incorporate a Sombrero extension that provided a single shared address space for all Sombrero applications. In this scenario an existing multiple-address space operating system with many threads may have threads that have their own virtual address space and threads that share a virtual address space.

This is similar to a single multi-threaded process in a traditional OS, but existing operating systems do not allow new, untrusted code to be incorporated into an executing process, with any amount of protection. Current operating systems also do not normally spread the process' address space across multiple nodes.

A Sombrero extension would allow new programs to be created and deleted within the Sombrero space with as much protection as desired by the users. A Sombrero application would have the same access to old-style operating system facilities as any other application, (including the file system, devices, etc.) but there would be no increase in efficiency that accessing facilities within the Sombrero-space would have.

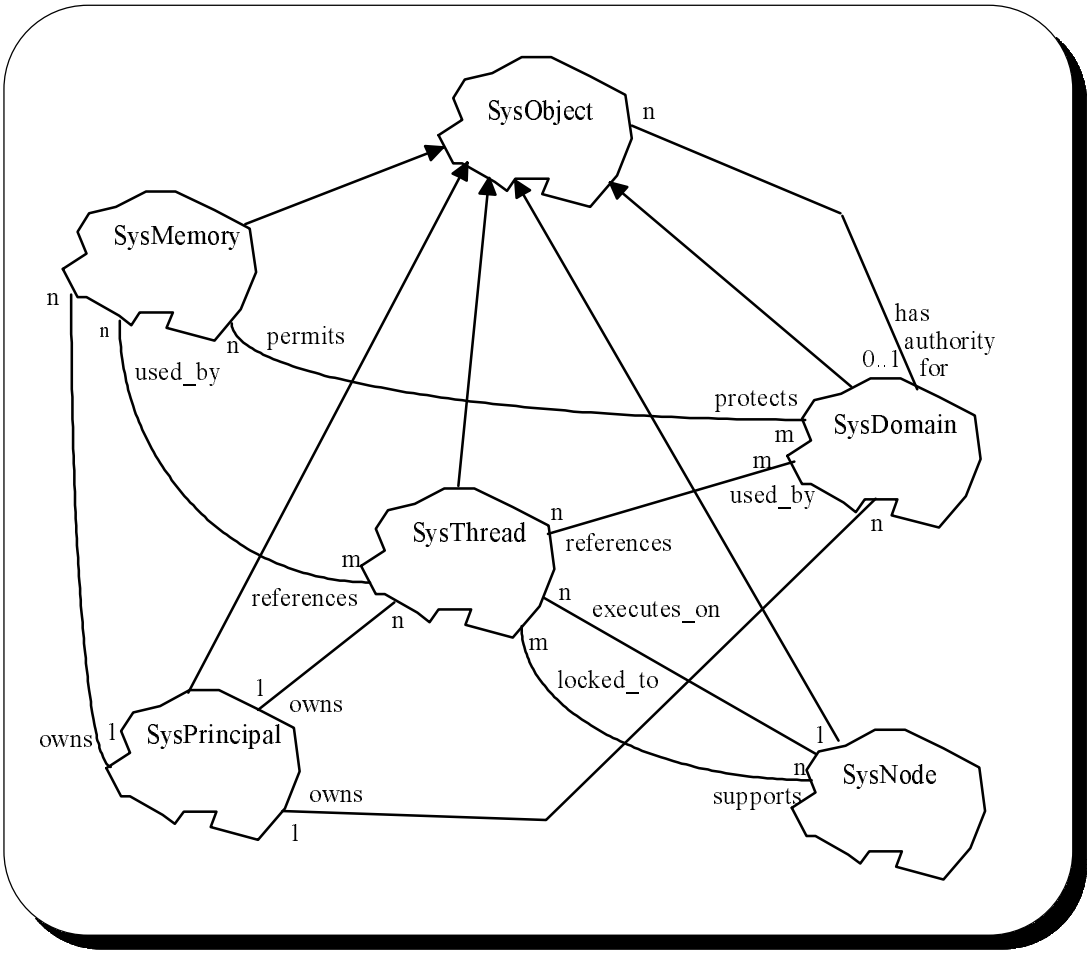
Supporting both multiple address spaces and a single address space requires hardware to support both arrangements. A TLB must be present within the processor as well as the RPLB. This may make the hardware slightly less efficient and certainly more complex, but retains the easy migration path from the current many address space operating system programs.

2 Library Interface

2.1 Overview

The objects described below are always accessed explicitly. This means that if there is an operation on an object, a pointer to that object must be obtained. Any thread can do any operation on any object in the system, including manipulating other threads. This is an important feature, so that operations used by a system administrator can be implemented (like killing a thread). It does require a mechanism to authorize the system calls being performed on an object.

The following diagram (using Booch notation [Bo*91]) illustrates the primary system interface objects. These objects all provide functions that allow information to be set and retrieved, including information about relationships with other objects.



2.1.1 Policy and Authorization

Throughout Sombrero, there are policy issues that are controllable by programs running in an authorized protection domain outside the Sombrero System Services. Authorized protection domains serve as agents for system objects. An authorized protection domain is a domain that has a gateway registered. A gateway is an entry point into code within the domain that can handle any authorization checks or signals on behalf of the object this domain acts as an agent for.

These domains will contain the code and data for operating environments, although different operating environments may share domains with other operating environments (such as a name server).

Only a thread executing in an authorized domain can perform the system calls identified in this document. It is sufficient to be in any authorized domain (not necessarily the authorized domain identified for this object) since an authorized domain is implicitly trusted. Threads not executing in an authorized domain will be redirected to the gateway of the authorized domain for the object the system call is being performed on.

An arbitrary thread does not control the redirection to this authorized domain's gateway, and therefore implicitly trusts any authorized domain. Since ultimately the "root" user is guaranteed to be trusted, only code running in the "root" domain can create an authorized domain. A domain may explicitly block transfer to an authorized domain by marking itself as an "isolated" domain.

A thread in an isolated domain will only transfer to a domain through its gateway if the current domain is also the object's authorized domain.

We need these special rules for authorized domain switching (outside of using explicit "set_switch" calls) so that the policy handler that decides whether a switch is allowed can be accessed without having to check the policy handler first. Without this policy for policy handlers just described, a thread could fault on access and pass control to the policy handler, but since this thread was not permitted access to the policy handler, it would fault again, and again, and again, eventually terminating the thread because the user's stack overflowed.

Another benefit of this simple policy for policy handlers may let switches to operating environment code be made without consuming PRAL resources (since an entry in the PRAL was not necessary to identify the switch rights from one domain to the other).

2.1.2 Gateways and Signal Stubs

Gateways are simply entry points that can be specified for a domain so that if this domain serves as an authorizing domain, the Sombrero System Services can transfer control to that point. There is only a single entry point, but part of the data passed to this entry point is an index identifying which of the possible events occurred. This index can then be used in a jump table or switch statement to do special processing based on which event caused the transfer.

Events such as accessing a protected memory address for which access rights are not currently in the PRAL can cause the gateway to be accessed as well as many other events detailed in the next section.

Each event that can be generated by the Sombrero System Services is documented within the following class definitions as a "signal stub". A "signal stub" is described for the class whose authorized domain's gateway would be called to implement the functionality. The "signal stub" function is not exposed to the Sombrero System Services directly, control is always passed through the gateway function.

The Sombrero System Services does not retain pointers to each of the "signal stub" functions implemented behind the gateway, only the gateway itself. It is up to the gateway to implement that table or case statement using the stub_id.

A gateway function always has the same prototype and it is as follows:

```
typedef void GatewayFunc(SysObject * generator,
                                 int stub_id,
                                 int syscall_id)
```

The generator is the object that caused the transfer, and the stub_id indicates the stub routine ID of the event. When there is no specific object (such as for class system calls) the generator will be the authorized domain of the thread. The syscall_id is always 0 for events other than authorization_failure.

There is no defined return location from a gateway function, it is up to the gateway to explicitly transfer control to the appropriate location after completing. Control never resumes to within the Sombrero System Services.

2.1.3 Transfer of Control

A thread is normally under complete control of the code it executes. It controls its program counter and performs explicit jumps to new addresses as a program is executed. A thread can have control transferred outside the normal program flow because of an event or signal.

Transfer is always to the gateway of the authorized domain, and it is performed by the Sombrero System Services. The thread the signal occurred against is the thread that is transferred to the authorized domain. This is very important for the security of the Sombrero System Services. The Sombrero System Services never assume that a thread will return. All code outside the Sombrero System Services code is untrusted.

The authorized domain of a thread is used for:

- interrupts sent by other threads
- exceptions caused by the thread itself
- monitored state changes of other threads
- unauthorized system calls directed to this thread

The authorized domain of a memory object is used for:

- access to a virtual address that is not currently permitted
- unauthorized system calls directed to this memory object.

The authorized domain of a domain, principal, or node is only used for:

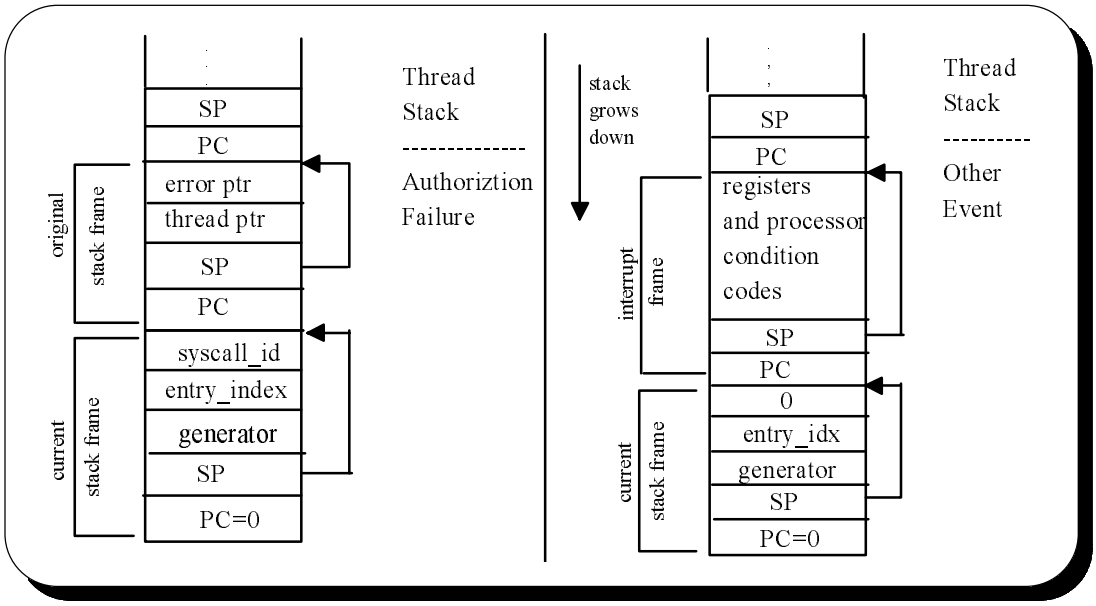
- unauthorized system calls directed to a particular domain, principal, or node.

By exploiting the ability of threads to do a passive transfer, the Sombrero System Services can completely protect itself. The thread simply has had its program counter and domain switched to a new address, just as if the thread had explicitly set its program counter to the new address itself.

The following diagram describes the two different ways the thread stack could appear after the gateway function had been transferred to.

The thread stack for an authorization failure includes the syscall_id as the third argument and is used by the gateway function to determine what call caused the authorization failure. The Sombrero System Services will have unrolled any stack frames on the user's stack that may have been placed there during the system call's execution. This prevents the code in the authorized domain from seeing any Sombrero System Services data and it makes the original call's stack frame available in a well known location. The original stack frame can be consulted by the gateway function to determine what the arguments to the call actually were.

The thread stack for other events will include a syscall_id of 0. A fictitious stack frame with the registers and processor condition codes at the time of interrupt, sits above the gateway stack frame so that the gateway function can determine where the code was interrupted.



A user level library function or macro (but not a system call) will allow the gateway to re-arrange the call stack so that it can return directly to the original caller rather than the Sombrero System Services.

To return from other events requires a slightly different library function or macro (but still not a system call) to restore the context and resume execution at the interrupted location.

2.1.4 System Errors

Errors may always occur due to an improper invocation of the system call and therefore need to be returned to the invoking procedure. This is passed back as an explicit "by reference" parameter on every system interface function.

Since system errors are not normally expected and under many conditions should be treated as an exception rather than providing a return code, the following special option is provided.

When a SysError value of USE_EXCEPTION is passed into the routine, any system error occurring as part of that system call will cause a signal to be generated. The "signal stub" corresponding to a system error will be executed.

2.1.5 Conventions

Throughout this document, there are system calls that fill data into an array of some data type. This array is always specified with an array size, and the array is expected to be allocated to that size. The system calls that fill out the array will always return the number of actual entries that were available. The number available can be less than, greater than, or equal to the size of the array. The order that data is placed into these arrays is undefined and there is no guarantee that repeated calls will return the entries in the same order.

2.2 SysObject

2.2.1 Description

This class represents any of the system objects. It is a superclass of all system objects. It provides the authorization facility that is common to all objects.

2.2.1.1 Usage

The SysObject object will be used during authorization of a request on a system service.

Example: A user wishes to terminate a thread that is in an infinite loop. A second thread attempts to invoke the terminate function on the thread to kill it. If the second thread was not currently executing code in an authorized domain, the system services will call the gateway function for the thread being terminated. The gateway function (which is authorized) will then perform the operation on behalf of the calling thread (if it decides the calling thread is allowed to).

The SysObject object will be used to hold important information a user program needs to know about this system object.

Example: An operating environment keeps an access control list expressed in terms of users and access rights in each memory object by storing a pointer to this list in the memento field.

2.2.2 C++ Interface

Class: SysObject

Identification	

ObjectType
object_type
(SysError & err)
	This returns an value indicating which of the various types of system objects this object really is:
	
	enum ObjectType {
	SYS_MEMORY,
	SYS_THREAD,
	SYS_DOMAIN,
	SYS_PRINCIPAL,
	SYS_NODE
	}

Instance Authorization	
void authorize (SysDomain * domain, SysError & err)	This function specifies the identified domain as the domain to act as the authorized domain. This domain must have a gateway. By default, created objects always have the creating thread's authorized domain as the authorized domain.
Mementos	
void set_memento (void * memento, SysError & err)	This routine stores an address within this object, that can be retrieved by the authorized domain to aid in its processing of any of the functions behind the gateway. The kernel does nothing with this value. By default this value is NULL.
void * get_memento (SysError & err)	This returns the address of the memento stored in this object.

Instance Authorization Query	
int access_allowed (SysError & err)	This function returns 1 if a system call against this object would be allowed and 0 if it is not.
SysDomain * get_authorized_domain (SysError & err)	This returns the domain whose gateway will be invoked for events against this object..
Authorization Stubs	
authorization_failure	This stub identifies a routine that is implemented within an authorized domain. The gateway is invoked with this stub_id when a thread tries to manipulate this object and is not running in an authorized domain.

2.2.3 C Interface

The following definitions correspond to the C++ functions defined above and provide identical functionality. Note that all references to system structures are through opaque types.

#include <sombrero/sys_object.h>

Identification

object_type **so_object_type**(*sys_object* o, *sys_error &* e);

Instance Authorization

void **so_authorize**(*sys_object* o, *sys_domain* d, *sys_error &* e);

Mementos

void **so_set_memento**(*sys_object* o, *void* * memento, *sys_error &* e);
void * **so_get_memento**(*sys_object* o, *sys_error &* e);

Instance Authorization Query

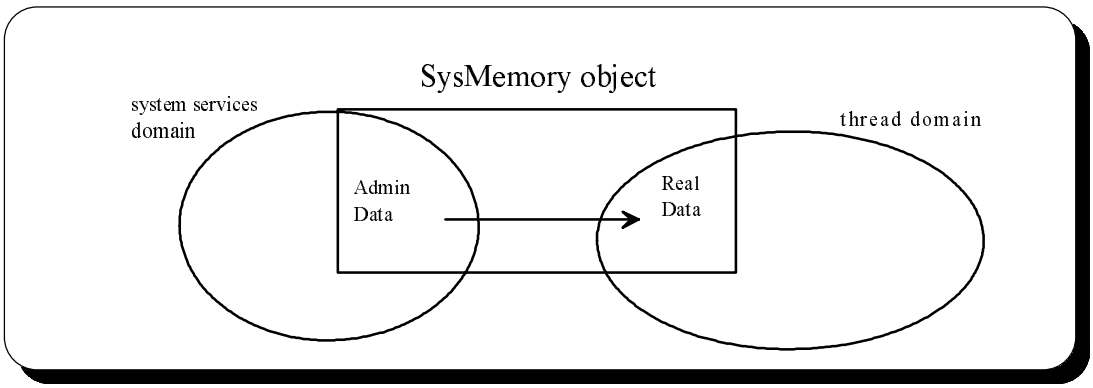
int **so_access_allowed**(*sys_object* o, *sys_error &* e);
sys_domain **so_get_authorized_domain**(*sys_object* o, *sys_error &* e)

2.3 SysMemory

2.3.1 Description

This class represents the memory objects within the Sombrero OS. A memory object is simply a range of contiguous addresses. A memory object is always accessed by its 64bit address. It is simply a block of memory to be manipulated. It can be used to represent much more complicated entities. A SysMemory object will identify the memory mapped locations for physical devices (such as a modem, printer, or video display card). It will also identify objects that will remain persistent across the life of a single thread of execution (what traditional operating systems model as files).

A SysMemory object actually exists in more than one protection domain. The administrative information used to keep track of allocation within the 64bit address space is located within the "system services" protection domain. The data itself is located within the protection domain of the initially owning thread. There is a pointer from the administrative data to the real data, but not from the real data to the administrative data. If a higher level interface wanted to add a back reference it may, but for operation of the system services it is not necessary.



While the manipulation of the "real data" is protected by the thread protection domain, the "admin data" is protected by the system services domain, and updates are allowed only through the system interface functions defined in this document.

2.3.1.1 Usage

The SysMemory object could be used when an algorithm requires dynamically allocated memory.

Example: A hash table is needed and a chunk of memory must be allocated to hold the hash table which is created based on the size of the data being hashed.

The SysMemory object could be used when data must be stored persistently.

Example: A new document is created to explain the vagaries of Martian life forms, so a memory object is created and associated with a name server domain.

The SysMemory object could be used as an execution stack for a new thread.

Example: A shell program causes a new program to be executed. A SysMemory object is allocated for the stack. The size is allocated very large (e.g., 4GB), but the needed size is set to 4k. The set_typical_region() and set_typical_permission() methods are invoked to allow access to the allocated portion. The memory object serving as the stack will be marked in the operating environment as a stack, so that access outside its read/write range will cause backing store to be added by the SysMemory's authorized domain.

2.3.2 C++ Interface

Class: SysMemory

Creation	
static SysMemory * alloc (int size, int needed_size, SysDomain * domain, SysError & err)	This is used to create a new memory object in the system. The memory object will be associated with the domain specified. If this domain is ever removed and no other domain references this memory object it will be freed. The default owner is the owner of the thread which created this memory object. The needed_size affects the backing store associated with this memory object (as opposed to the address range allocated). A needed_size of 0 means the entire address range gets backing store. A negative number means to start allocating from the end of the address range. A positive number means to start allocating from the beginning of the address range. If the allocation fails the function would return a NULL pointer, in addition to setting the error field.
void reference (SysDomain * domain, SysError & err)	This function will cause this memory object to be referenced from the specified domain, indicating that the va_space can not be reclaimed. The virtual address space of this object will be included against the quota for the owner of this domain, but not the backing store (that will only be counted against its real owner).
void dereference (SysDomain * domain, SysError & err)	This will remove the reference to this memory object from the specified domain. If no other domain references this memory object, it is freed.
Splitting	
SysMemory * split (void * start_address, SysError & err)	This splits the memory object, creating a new memory object at the start_address (which must be within the address range of this memory object). The current memory object will end at "start_address-1". All associated protection already specified against this memory object will be retained. Where protection regions already specified for this object span both objects, the regions will be divided as well.
void glue (SysMemory * hi_mem, SysError & err)	This will glue two memory objects together. They must have adjacent virtual address ranges. The lower addressed memory object will be retained with the higher addressed memory object being removed from the system. Any regions identified on the higher memory object as providing access will be retained and now will be against the lower addressed memory object.

Access	
void * data(SysError & err)	This returns a void * pointer to the start of the data of this memory object. The pointer to the SysMemory object and the pointer returned by this function are NOT the same.
int length(SysError & err)	This returns the size of the virtual address range described by this memory object.
static SysMemory * containing (void * address, SysError & err)	This call returns a pointer to the SysMemory object that contains the "address" within its virtual address range.
Recoverability	
void flush(SysError & err)	This will cause the object to be written to a persistent storage device immediately. This is important if the memory object represents data that must be consistent in case of a power crash or other failure.
Allocation	
void set_needed_size_ra (void * start, void * end, SysError & err)	This specifies that physical backing store is currently needed for the memory object from start to end. The entire virtual address range does not have to have backing store if it contains no information. This call can increase or decrease the amount of backing store required for this memory object. Setting this option removes any range defined using a range mask.
void set_needed_size_rm (RangeMask mask, SysError & err)	This will cause the object to have backing store for the ranges specified within the range mask. This allows discontiguous regions of memory to be provided backing store. Setting this option removes any range set using a specific start and end address range.
void get_needed_size_ra (void * & start, void * & end, SysError & err)	This returns the current range of addresses that have backing store associated with this SysMemory object. This will produce an error if a range mask is set rather than a specific range.
void get_needed_size_rm (RangeMask & mask, SysError & err)	This returns the range mask that defines the backing store associated with this SysMemory object. This will produce an error if a specific range is set rather than a range mask.

Protection Domains	
void set_typical_region (void * start, void * end, Mode mode, AccessType type, SysError & err)	This command defines the typical region for this memory object. A typical region is a special region that a domain can be attached to without specifying specific address ranges. Changing the typical region effectively causes all domains associated with this typical region to have their accessibility to this memory object changed. The mode indicates whether the domain is allowed access or denied access for this address range. The type indicates, whether this region describes read, write, or execute access.
void set_typical_region_masks (RangeMask masks[], int num_masks, AccessType type, SysError & err)	This identifies a set of range masks, which identify a possibly discontiguous set of regions that define permitted access to this memory object. A domain can be attached to this typical region without specifying specific address ranges. Changing the typical region effectively causes all domains associated with this typical region to have their accessibility to this memory object changed. The type indicates whether this region describes read, write, and/or execute access.
void set_typical_permission (SysDomain * domain, SysError & err)	This causes the domain specified to have access to the typical protection provided by this memory object. If the typical region is changed the accessibility of the domain to this memory object is changed. If the domain had any specific permission values those permissions are discarded. A private thread domain cannot have any execute permissions.
void set_specific_region_permission (SysDomain * domain, void * start_region, void * end_region Mode mode, AccessType type, SysError & err)	This identifies which region of addresses within the SysMemory object is accessible by the protection domain specified. If the domain had been marked as having typical protection, this replaces all typical permission values. This call adds or removes permission from the existing set of permissions. The range must be within the data range of the SysMemory object or an error will be returned. The mode indicates whether the domain is allowed access or denied access in that range. Internally a range could be split by enabling a large range and afterwards clearing a piece in the middle. The type indicates read, write, and/or execute access permissions. A private thread domain cannot have any execute permissions.

void set_specific_region_masks (SysDomain * domain, RangeMask masks[] int num_masks, AccessType type, SysError & err)	This identifies a set of region masks, which identify a discontiguous set of ranges that define permitted access by this protection domain to this memory object. If the domain had been marked as typical, those permissions will be replaced. This call replaces the entire set of permissions. The range masks must only indicate addresses within the data range. The type indicates whether this is a setting for read, write, and/or execute access permissions. A private thread domain may not have execute permissions.
Protection Domain Switching	
int set_switch (SysDomain * active, SysDomain * new, void * address, Mode mode, AccessType type, SysError & err)	This sets an address as causing a protection switch when that address is accessed. The switch only occurs if executing in the "active" domain. After accessing the switch address, the domain will be changed to the "new" domain. The mode indicates whether the switch is being allowed or disallowed. The type indicates whether the switch will occur upon read, write, and/or execute access to the specified address. When allowing a switch, an error will be returned if a switch already exists at this address for the same active domain and with the same access type.
Protection Information	
int num_regions(SysError & err)	This returns the number of protection regions associated with this memory object.
int get_region_list(Region regions[], int max_size, SysError & err)	This copies up to max_size regions into the array of regions. Typical region definitions will appear as entries with no accessing_domain. Domains that are defined in terms of a "typical" region will also have the accessing_domain and the start_address and end_address of the typical region. Domains that use a specific region also have the accessing_domain, a start_address, and an end_address. Switch addresses include the accessing_domain, the switch_domain, and the address.

	A Region structure contains the following entries:
	
	struct {
	  enum RegionType region_type; // typical, specific, switch
	  SysDomain * accessing_domain
	  enum AccessType type; // read, write, and/or execute
	  union {
	  struct {
	  SysDomain * switch_to_domain
	  void * address
	  } switch
	  struct {
	  void * start_address
	  void * end_address
	  } region
	  } u
	}
	
	The number of regions currently in the SysMemory object is returned.

int num_region_masks (SysError & err)	The current number of active region masks for this memory object is returned.

int
get_region_mask_list(
  RegionMask masks[],
  int max_size,
  SysError & err)
	This copies up to the max_size region masks into the array of region masks.
	
	Typical region mask definitions will appear as entries with no accessing_domain.
	
	Domains that are defined in terms of a "typical" region mask will also have the accessing_domain and a region mask for the typical region.
	
	Domains that use a specific region also have the accessing_domain, and a region mask.
	
	Switch addresses include the accessing_domain, the switch_domain, and the address.
	
	A RegionMask structure contains of the following entries:
	struct {
	  enum RegionType region_type; // typical, specific, switch
	  SysDomain * accessing_domain
	  enum AccessType type; // read, write, and/or execute
	  union {
	  struct {
	  SysDomain * switch_to_domain
	  void * address
	  } switch
	  RangeMask mask
	  } u
	}
	
	The number of region masks currently in the SysMemory object is returned.

Principals	
SysPrincipal * owner(SysError & err)	This function returns the owning SysPrincipal of this SysMemory object. A principal can become null, if the principal owning this memory object is ever removed. This causes all objects to be removed from the system, unless some domain is explicitly referencing this memory.
SysPrincipal * set_owner(SysPrincipal * p, SysError & err)	This function allows a thread to transfer ownership of a memory object to another principal. An object can be disowned by setting a NULL pointer as the principal.
Multiple Node Consistency	

void set_consistency (ConsistencyMode c, SysError & err)	This sets the multiple node consistency semantics for this memory object to be one of the following:
	
	enum ConsistencyMode {
	one_reader_writer
	one_writer_many_reader
	many_writer_many_reader
	}

ConsistencyMode consistency (SysError & err)	This gets the current consistency semantics for this memory object.
Replication	
void set_replication_mode (ReplicationMode mode, SysError & err)	This identifies the object as supporting "system defined" replication or "user controlled" replication. An object that only has user controlled replication will never create or remove a surrogate automatically, instead the authorized domain will be consulted.
ReplicationMode replication_mode (SysError & err)	This retrieves the current replication mode from the memory object.
void add_surrogate (SysNode * node, SysError & err);	This will create a surrogate on the node specified. There is no guarantee that this surrogate will remain on the node, unless this object is marked for explicit replication.
void remove_surrogate (SysNode * node, SysError & err)	This will remove a surrogate from the node specified. There is no guarantee that a surrogate will not be created later, unless this object is marked for explicit replication.
int has_surrogate (SysNode * node, SysError & err)	This will return non-zero if there is a surrogate for this memory object on the node specified.

Memory Object Stubs	
replication_needed	This stub identifies the routine that is implemented within an authorized domain. The gateway is invoked with this stub id whenever this memory object is set for explicit replication and the memory object may need to be moved.
invalid_read_access	This stub identifies the routine that is implemented within an authorized domain. The gateway is invoked with this stub id whenever an address is read within this memory object by a thread not executing in a domain explicitly allowed read access by this memory object.
invalid_write_access	Same as invalid_read_access() but for write.
invalid_execute_access	Same as invalid_read_access() but for execute.

2.3.3 C Interface

The following definitions correspond to the C++ functions defined above and provide identical functionality. Note that all references to system structures are through opaque types.

#include <sombrero/sys_memory.h>

Creation

sys_memory **sm_alloc**(*int* size, *int* needed_size,*sys_domain* d, *sys_error &* e);
void **sm_reference**(*sys_memory* m, *sys_domain* d, *sys_error* & e);
void **sm_dereference**(*sys_memory* m, *sys_domain* d, *sys_error* & e);

Splitting

void **sm_split**(*sys_memory* m, *void* * start_address, *sys_error* & e);
void **sm_glue**(*sys_memory* m1, *sys_memory* m2, *sys_error* & e);

Access

void * **sm_data**(*sys_memory* m, *sys_error* & e);
int **sm_length**(*sys_memory* m, *sys_error* & e);
sys_memory **sm_containing**(*void* * address, *sys_error* & e);

Recoverability

void **sm_flush**(*sys_memory* m, *sys_error* & e)

Allocation

void **sm_set_needed_size_ra**(*sys_memory* m, *void* * start, *void* * end, *sys_error & e)*
void **sm_set_needed_size_rm**(*sys_memory* m, *RangeMask* mask, *sys_error & e)*
void **sm_get_needed_size_ra**(*sys_memory* m, *void * &* start, *void * &* end, *sys_error & e)*
void **sm_get_needed_size_rm**(*sys_memory* m, *RangeMask* & mask, *sys_error & e)*

Protection Domain Access

void **sm_set_typical_region**(*sys_memory* m, *void* * start, *void* * end,
Mode mode, *AccessType* type, *sys_error* & e)
void **sm_set_typical_region_masks**(*sys_memory* m, *RangeMask* masks[], *int* num_masks,
AccessType type, *sys_error* & e)
void **sm_set_typical_permission**(*sys_memory* m, *sys_domain* domain, *sys_error* & e)
void **sm_set_specific_region_permission**(*sys_memory* m, *sys_domain* domain,
void * start, *void* * end, *Mode* mode,
AccessType type, *sys_error* & e)
void **sm_set_specific_region_masks**(*sys_memory* m, *sys_domain* domain,
RangeMask masks[], *int* num_masks,
AccessType type, *sys_error* & e)

Protection Domain Switching

void **sm_set_switch**(*sys_memory* m, *sys_domain* active, *sys_domain* new,
void * address, *Mode* mode, *AccessType* type, *sys_error* & e)

Protection Information

int **sm_num_regions**(*sys_memory* m, *sys_error* & e)
int **sm_get_region_list**(*sys_memory* m, *struct Region* r[], *int* max_size, *sys_error* & e)

int **sm_num_region_masks**(*sys_memory* m, *sys_error* & e)
int **sm_get_region_mask_list**(*sys_memory* m, *struct RegionMask* masks[],
int max_size, *sys_error* & e)

Principals

sys_principal **sm_owner**(*sys_memory* m, *sys_error* & e)
void **sm_set_owner**(*sys_memory* m, *sys_principal* p, *sys_error* & e)

Distributed Sharing

void **sm_set_share_boundaries**(*sys_memory* m, *void* * boundaries[],
int num_boundaries, *sys_error* & e)
void **sm_num_share_boundaries**(*sys_memory* m, *sys_error* & e)
int **sm_get_share_boundaries**(*sys_memory* m, *void* * boundaries[],
int num_boundaries, *sys_error* & e)

Consistency

void **sm_set_consistency**(*sys_memory* m, *ConsistencyMode* c, *sys_error* & e)
ConsistencyMode **sm_consistency**(*sys_memory* m, *sys_error* & e)

Replication

void **sm_add_surrogate**(*sys_memory* m, *sys_node* n, *sys_error* & e)
void **sm_remove_surrogate**(*sys_memory* m, *sys_node* n, *sys_error* & e)
int **sm_has_surrogate**(*sys_memory* m, *sys_node* n, *sys_error* & e)
void **sm_set_replication_mode**(*sys_memory* m, *ReplicationMode* mode, *sys_error* & e)
ReplicationMode **sm_replication_mode**(*sys_memory* m, *sys_error* & e)

2.4 SysThread

2.4.1 Description

This class represents a thread of execution within the system. A thread logically consists of a program counter, a stack, a principal, an active protection domain, and a set of memory objects. A thread can travel between protection domains. A switch between protection domains happens simply as the result of execution occurring at an address which within the current protection domain, indicates a switch to another protection domain.

Note that threads using shared memory (a typical scenario) will require synchronization. There is no specific Sombrero support for semaphores or other forms of locking. These can be implemented using standard hardware test and set instructions, and are expected to be in runtime packages and are therefore out of the scope of this discussion.

Synchronization of multiple threads is done through the shared memory between the threads, but there is still a need for the system to notify one or more threads if the thread state changes (for example, an unexpected termination) since other threads may have been expecting a master thread to wake up the others. This is accomplished using the monitoring functions.

2.4.1.1 Usage

The SysThread object will be used when an algorithm requires a new thread of control, or when the current thread of control must be manipulated.

Example: A word processor program is initiated from the window manager, so a new thread is created.

2.4.2 C++ Interface

Class SysThread

Creation	
static SysThread * create(void *exec_address, void * stack_address, const wait_flag, SysError & err)	This creates a new thread which will begin executing at the exec_address specified during creation. The stack will begin at the stack address and move downward. Typically, a SysMemory object that is big enough to hold the largest stack ever desired will be allocated prior to creating this thread. The initial size of the SysMemory object will be set to a non-zero value and a routine set to resize the backing store of the associated memory object. The thread may wait for its parent to request status from it, or it may automatically be freed when it exits. A thread always begins in the suspended state and within the general protection domain of the creating thread.
void free(SysError & err)	This causes the thread to be freed from the system. The pointer value that represents this thread will no longer be valid. The thread must have exited.
Access	
static SysThread * current_thread(SysError & err)	This allows the currently executing thread to get a pointer to itself.

struct t_state thread_state(SysError & err)	This returns information about the state of the thread, including total execution time (clock and CPU) as well as its current execution state (i.e., EXITED, SUSPENDED, RUNNING) and secondary state (i.e., BACKING STORE, MAIN STORE, etc.) and exit conditions.
void get_context(ContextRec & c, SysError & err);	This gets the context of all registers, including special registers - PC (program counter), PGPD (previous general protection domain), GPD (current general protection domain), CPD (carrier protection domain), and SP(stack pointer), of this thread. This is only allowed if the thread is suspended
void set_context(ContextRec c, SysError & err);	This sets the context of all writable registers, including special registers - PC and SP, but not PGPD, GPD, or CPD, of this thread. This is only allowed if the thread is suspended
Thread Control	
void suspend (int timeout, SysError & err)	This will cause the thread to stop executing instructions until it is awakened, or until the timeout has expired. The timeout is in microseconds. It is guaranteed to awaken sometime after the timeout has expired. A timeout of -1 will sleep indefinitely. A timeout of 0 will just cause the thread to be rescheduled (effectively yielding the processor prior to expiration of its time-quanta).
void resume(SysError & err)	This will cause the thread to begin executing the next time it has an available time slice.
void exit(SysError & err)	This causes the thread to stop execution and go into a state where it is waiting to be freed by another thread before being removed from the system.
void abort(SysError & err)	This causes the thread to be interrupted with the expected result being an abnormal termination. Transfer to the authorized domain for this thread will occur, and that code is expected to abort the thread in a meaningful way for a debugger.
void terminate(SysError & err)	This will cause this thread to be interrupted with the expected result being a normal termination. Transfer to the authorized domain will occur and the authorized domain's code is expected to do cleanup prior to termination.
void interrupt(SysError & err)	This will cause the thread to be interrupted. The transfer of control will be to the authorized domain and the code appropriate for this interrupt processed. The type of interrupt occurring will typically be communicated directly to the authorized domain.

Execution Interval	
void set_max_exec_time (int interrupt_at_cycles, int total_cycles, SysError & err)	This specifies a maximum limit on the number of CPU cycles that this thread will accumulate before an interrupt will occur. The timer_expired stub will be executed when the number of cycles is exceeded. This is the same number of cycles as can be retrieved from thread_state. The total_cycles are the number of cycles including the time in the interrupt handler before the thread is exited.
void get_max_exec_time (int & interrupt_at_cycles, int & total_cycles, SysError & err)	This retrieves the number of cycles at which an interrupt will occur, and the total cycles at which the thread will be exited.
Domain	
SysDomain * activeDomain(SysError & err)	This returns the general protection domain active for this thread. This normally would be the primary domain for the owning SysPrincipal but could change over time as domains are entered and left.
SysDomain * threadDomain(SysError & err)	This returns the domain associated with the currently executing thread. This domain was created automatically when the thread was created, and will be destroyed automatically. This domain remains constant for the life of the thread. Memory objects can be associated with a private thread domain using one of the following commands: set_typical_permission(), set_specific_region_permission(), or set_specific_region_masks() Even after a general protection domain switch has occurred, the thread domain will still allow access to the designated memory objects.
Principal	
SysPrincipal * owner(SysError & err)	This returns the SysPrincipal who owns the thread.
void transfer (SysPrincipal * new_owner, SysError & err)	This allows the ownership of the thread to transfer. to a new owner.

Thread Monitoring	

void
monitor_thread
(MonitorData monitors[],
int num_monitors,
SysError & err)
	The change in state of any of the specified threads will cause the monitored_thread_change stub for this thread to be executed.
	
	The monitor array indicates the threads and the states it is interested in.
	
	struct MonitorData {
	SysThread * thread,
	int state_set_mask
	}

int num_monitoring (SysError & err)	This returns the number of monitors registered for this thread.
int query_monitor (MonitorData monitors[] int num_monitors, SysError & err)	This returns the list of monitor data associated with this thread. At most num_monitors will be retrieved into the monitors array. The actual number of monitors registered at the time of this call will be returned.
Scheduling	
void set_multi_task (int priority SysError & err)	This will set the thread as being a multitask, where the thread has the specified priority. A thread that is classed as a multitask will be pre-empted by queue based threads or cyclic threads.
void set_queue_task (SysError & err)	A thread that is a queue based task, will run to completion unless a cyclic task pre-empts it.
void set_cyclic_task (int interval int percentage, SysError & err)	A thread that is a cyclic task, will always run at the specified interval, but cannot take longer than the specified percentage of that interval or it will be aborted. The interval is in microseconds, and specifies that start time of the thread. A thread with an interval of 1 second (1,000,000us) would execute exactly 1 second after it started executing last time.
Node	
void modify_node_list(SysNode *node, int mode, SysError & err)	This allows nodes to be added and removed (based on the mode) to the possible nodes that can support this thread. This allows policy mechanisms to be developed to keep a thread on a node or set of nodes. This would even let a heterogeneous Sombrero system be created, where threads compiled for a given processor architecture would be identified and limited to those nodes with that processor architecture.
SysNode * current_node(SysError & err)	This returns the node this thread is currently executing on.
void migrate(SysNode * node, SysError & err)	This transfers a thread from one node to another. This would be useful for a load balancing thread which could monitor each node's processor capacity and transfer threads to other nodes accordingly.

Thread Stubs	
terminate_thread	This stub identifies the routine that is implemented within an authorized domain. The gateway function is invoked with this stub_id whenever a thread has its terminate() function called.
interrupt_thread	This stub identifies the routine that is implemented within an authorized domain. The gateway function is invoked with this stub_id whenever a thread has its interrupt() function called.
abort_thread	This stub identifies the routine that is implemented within an authorized domain. The gateway function is invoked with this stub_id whenever a thread has its abort() function called.
illegal_instruction	This stub identifies the routine that is implemented within an authorized domain. The gateway function is invoked with this stub_id whenever a thread executes an illegal instruction.
timer_expired	This stub identifies the routine that is implemented within an authorized domain. The gateway function is invoked with this stub_id whenever the maximum execution time for this thread is exceeded.
math_error	This stub identifies the routine that is implemented within an authorized domain. The gateway function is invoked with this stub_id whenever a math error is encountered such as divide by 0.
syscall_error	This stub identifies the routine that is implemented within an authorized domain. The gateway function is invoked with this stub_id whenever a system call fails and the SysError was specified on input as USE_EXCEPTION.
monitored_thread_change	This stub identifies the routine that is implemented within an authorized domain. The gateway function is invoked with this stub_id whenever some thread which was being monitored by this thread changes state.

2.4.3 C Interface

The following definitions correspond to the C++ functions defined above and provide identical functionality. Note that all references to system structures are through opaque types.

#include <sombrero/sys_thread.h>

Creation

sys_thread **st_create**(*void* * exec_address, *void* * stack_address, *int* wait, *sys_error* & e);
void **st_free**(*sys_thread* t, *sys_error* & e);

Access

sys_thread **st_current_thread**(*sys_error* & e);
struct t_state **st_thread_state**(*sys_thread* t, *sys_error* & e);
void **st_get_context**(*sys_thread* t, *ContextRec & c, sys_error* & e);
void **st_set_context**(*sys_thread* t, *ContextRec c, sys_error* & e);

Thread Control

void **st_suspend**(*sys_thread* t, *int* timeout, *sys_error* & e);
void **st_resume**(*sys_thread* t, *sys_error* & e);
void **st_exit**(*sys_thread* t, *sys_error* & e);
void **st_abort**(*sys_thread* t, *sys_error* & e);
void **st_terminate**(*sys_thread* t, *sys_error* & e);
void **st_interrupt**(*sys_thread* t, *sys_error* & e);

Execution Interval

void **st_set_max_exec_time**(*sys_thread* t, *int* int_at_cycles, *int* total_cycles, *sys_error* & e);
void **st_get_max_exec_time**(*sys_thread* t, *int* & int_at_cycles, *int* & total_cycles,
sys_error & e);

Domains

sys_domain **st_active_domain**(*sys_thread* t, *sys_error* & e);
sys_domain **st_thread_domain**(*sys_thread* t, *sys_error* & e);

Principal

sys_principal **st_owner**(*sys_thread* t, *sys_error* & e);
void **st_transfer**(*sys_thread* t, *sys_principal* new_owner, *sys_error* & e);

Thread Monitoring

int **st_monitor_thread**(*sys_thread* t, *MonitorData* monitors[], *int* num_monitors,
sys_error & e);
int **st_num_monitoring**(*sys_thread* t, *sys_error* & e);
int **st_query_monitor**(*sys_thread* t, *MonitorData* monitors[], *int* num_monitors,
sys_error & e);

Scheduling

void **st_set_multi_task**(*sys_thread* t, *int* priority, *sys_error* & e);
void **st_set_queue_task**(*sys_thread* t, *sys_error* & e);
void **st_set_cyclic_task**(*sys_thread* t, *int* interval, *int* percentage, *sys_error* & e);

Node

void **st_modify_node_list**(*sys_thread* t, *sys_node* n, *int* mode, *sys_error* & e);
sys_node **st_curr_node**(*sys_thread* t, *sys_error* & e);
void **st_migrate**(*sys_thread* t, *sys_node* n, *sys_error* & e);

2.5 SysDomain

2.5.1 Description

The SysDomain object indicates a set of regions within memory objects and the access rights to those regions. The set of regions becomes a map of address ranges that allow read, write, execute permissions, or switches to another domain if a read, write, or execute occurs at a particular address. The declaration of the regions is done using SysMemory calls. A SysDomain also defines the context that a thread executes within.

When a SysMemory object is created, a domain is always specified. A SysMemory without a referencing domain will be reclaimed.

2.5.1.1 Usage

The SysDomain object will be created when a set of protection rights needs to be given independently of the initial domain.

Example: A domain will be created when a set of documents are given a second set of passwords so that only by entering the correct password can the data be read.

2.5.2 C++ Interface

Class: SysDomain

Create	
static SysDomain * declare(SysPrincipal *owner, SysError & err)	This is the standard interface to create a new global protection domain. Specifying the owner during create is simply a convenience since a call to set_owner can always be done later.
void destroy(SysError & err)	This will remove the domain from the system. This function is only allowed for general domains and not thread domains (which are always removed implicitly by the free()'ing of the owning thread).
Ownership	
void set_owner (SysPrincipal * owner, SysError & err)	This changes the owner from the current owner to a new owner.
SysPrincipal * owner (SysError & err)	This returns the current owner of this domain.
Memory Objects	
int num_objects(SysError & err)	This simply returns the number of objects referenced by this domain, either explicitly through a call to reference() or implicitly through a call to: set_typical_permission(), set_specific_region_permission(), set_specific_region_masks(), or set_switch().

int get_objects(SysMemory * objects[], int max_num, SysError & err)	This fills in the objects array with pointers to the memory objects that this domain is currently referencing. This function returns the number of memory objects this domain currently references.
Isolation	
IsolationMode isolation(SysError & err)	This returns whether threads executing in this domain are running in an isolated mode (i.e. they will not automatically change to execute in a domain other than their authorized domain.
void set_isolation(IsolationMode m, SysError & err)	This allows the domain to be set to an isolation mode.
Gateway	
void set_gateway (void * gateway_address, SysError & err)	This function identifies the address where the gateway function is located, if there is a gateway. Identifying a gateway address for a domain causes an implicit reference to the memory object that contains that address. Setting the gateway address to NULL removes the gateway. Only code executing in the "root" domain can make this call. This is special because this is the only way a thread can implicitly be transferred to unknown code.
void * gateway(SysError & err)	This returns the address of the gateway routine that this domain provides.

2.5.3 C Interface

The following definitions correspond to the C++ functions defined above and provide identical functionality. Note that all references to system structures are through opaque types.

#include <sombrero/sys_domain.h>

Create

sys_domain **sd_declare**(*sys_principal* owner, *sys_error* & e);
void **sd_destroy**(*sys_domain* d, *sys_error* & e);

Ownership

void **sd_set_owner**(*sys_domain d, sys_principal* p, *sys_error* & e);
sys_principal **sd_owner**(*sys_domain* d, *sys_error* & e);

Memory Objects

int **sd_num_objects**(*sys_domain d, sys_error* & e);
int **sd_get_objects**(*sys_domain* d, *sys_memory* objects[]*, int* max_num*, sys_error* & e);

Isolation

IsolationMode **sd_isolation**(*sys_domain* d*, sys_error* & e);
void **sd_set_isolation**(*sys_domain* d*, IsolationMode* mode*, sys_error* & e);

Gateway
void **sd_set_gateway**(*sys_domain d, void* * gateway_address,*sys_error* & e);
void * **sd_gateway**(*sys_domain d, sys_error* & e);

2.6 SysPrincipal

2.6.1 Description

The SysPrincipal object represents a user within the Sombrero operating system. A SysPrincipal is the ultimate owner of protection domains, and has threads operating on its behalf. Resource charging, accounting, and quotas can be set up against a SysPrincipal.

2.6.1.1 Usage

The SysPrincipal object will be created when a new user is added.

Example: The "add_user" command would create a principal.

2.6.2 C++ Interface

Class: SysPrincipal

Create	
static SysPrincipal * add(SysError & err)	This routine creates a new SysPrincipal in the system. A new SysPrincipal should normally gets its own protection domain, but this is not automatic. There is always a "root" SysPrincipal who exists in the system and can never be deleted. That special principal has authority to make any system call.
void remove(SysError & err)	This routine removes the SysPrincipal from the system. This has no effect on the "root" SysPrincipal. Removing a principal removes all memory, threads, and domains owned by this principal. Threads owned by this principal but executing in another domain will terminate after leaving that domain.
Global Access	
static int num_principals(SysError & err)	This returns the number of principals currently allocated in the system.
static int get_principals (SysPrincipal * principals[], int max_num, SysError & err)	This fills the array up to max_num with the principals currently allocated in the system. The actual number of principals in the system when this function was called is returned as well.
Accounting	
void set_quotas(int va_space, int backing_store, int num_mem_objects, int num_threads, int num_domains, SysError & err)	This routine sets quotas for all these resources. These totals are tracked for a SysPrincipal based on the associated threads, domains, and objects owned by it. Setting any of these values to 0 implies there is no limit.

void
get_quotas(
  int & va_space,
  int & backing_store,
  int & num_mem_objects,
  int & num_threads,
  int & num_domains,
  SysError & err)
	This routine returns the values that the various quotas are set to.

void
get_current_stats(
  int & va_space,
  int & backing_store,
  int & num_mem_objects,
  int & num_threads,
  int & num_domains,
  SysError & err)
	This routine returns the current usage by a SysPrincipal of each of the attributes that can have quotas set against them.

Thread Info	
int num_threads(SysError & err)	This routine simply returns the number of threads currently owned by this SysPrincipal.

int
get_threads(
  SysThread * threads[],
  int max_num,
  SysError & err)
	This routine will get up to the max_num SysThreads currently owned by this SysPrincipal. The current number of owned threads will be returned.

Memory Info	
int num_objects(SysError & err)	This routine simply returns the number of SysMemory objects owned by this SysPrincipal.

int
get_objects(
  SysMemory * objects[],
  int max_num,
  SysError & err)
	This routine will get up to the max_num SysMemory objects currently owned by this SysPrincipal. The current number of objects will be returned.

Domain Info	
int num_domains(SysError & err)	This routine simply returns the number of SysDomain objects owned by this SysPrincipal.

int
get_domains(
  SysDomain * domains[],
  int max_num,
  SysError & err)
	This routine will get up to the max_num SysDomain objects currently owned by this SysPrincipal. The current number of objects will be returned.

2.6.3 C Interface

The following definitions correspond to the C++ functions defined above and provide identical functionality. Note that all references to system structures are through opaque types.

```
#include <sombrero/sys_principal.h>
```

Create

```
sys_principal  sp_add(sys_error & e);
void  sp_remove(sys_principal p, sys_error & e);
```

Global Access

```
void  sp_num_principals(sys_error & e);
void  sp_get_principals(sys_principal principals[], int max_num, sys_error & e);
```

Accounting

```
void  sp_set_quotas(sys_principal p, int va_space, int backing_store,
                    int num_mem_objects, int num_threads, int num_domains,
                    sys_error & e);
void  sp_get_quotas(sys_principal p, int & va_space, int & backing_store,
                    int & num_mem_objects, int & num_threads, int & num_domains,
                    sys_error & e);
void  sp_get_current_stats(sys_principal p, int & va_space, int & backing_store,
                           int & num_mem_objects,int & num_threads,
                           int & num_domains, sys_error & e);
```

Thread Info

```
int  sp_num_threads(sys_principal p, sys_error & e);
int  sp_get_threads(sys_principal p, sys_thread threads[], int max_num, sys_error & e);
```

Memory Info

```
int  sp_num_objects(sys_principal p, sys_error & e);
int  sp_get_objects(sys_principal p, sys_memory objects[], int max_num, sys_error & e);
```

Domain Info

```
int  sp_num_domains(sys_principal p, sys_error & e);
int  sp_get_domains(sys_principal p, sys_domain domains[], int max_num, sys_error & e);
```

2.7 SysNode

2.7.1 Description

The SysNode object represents a physical computing resource, and provides statistics and allows the basic Sombrero resources to be manipulated. This includes the address space, the backing store, the hardware memory mapping and hardware interrupt vectors.

2.7.1.1 Usage

The SysNode object will be created when a new workstation is added.

Example: The "add_node" command would create a node.

2.7.2 C++ Interface

Class: SysNode

Creation	
static SysNode * add_node(const char * nsap_octet_string, SysError & err)	This will create a node within the system that can be assigned threads based on internal algorithms or direction from the threads themselves. The nsap_octet_string is the Network Service Access Point identifier which can be for whatever underlying network transport mechanism (TCP/IP, X.25, etc.) is used to identify nodes within the LAN. Normally a node will come up and if it is not attached to the network allow itself to go into stand-alone mode. In stand-alone mode it becomes its own Sombrero-space. It can also be put in stand-by mode to wait for it to be added to the Sombrero network. Once a node has been added, it will always try to rejoin the network, by negotiating with network. If no network is present it can go back into stand-alone mode. If the node hadn't been shutdown with the "no-storage" option, the data that was stored on this node would be lost.
void shutdown(int mode, SysError & err)	This will shutdown this node such that the OS will migrate all threads off this node to other processors, or if no processors remain, those threads will be exited, or suspended depending on the mode. The modes are as follows: POWER_DOWN - migrate or suspend threads REBOOT - migrate or exit threads NO_STORAGE - migrate threads and all memory objects
void remove_node(SysError & err)	A node can be removed only if it is shutdown. The last node in the system can never be removed.

Resources	
void set_memory_map (SysMemory * map, void * physical_address, SysError & err)	This function allows a memory object to have a specific physical range of addresses associated with it, starting at physical_address. The range of physical addresses is equal to the size of the virtual address space identified in this memory object. This function is critical to allow devices to be added to the system. It allows a physical memory map to be associated with a memory object so that it can be protected by a protection domain that contains the device driver.
void add_backing_store (int device_id, int size, SysError & err)	This adds a new disk or disk partition to this node, so this node can provide backing store to the virtual addresses that require backing store on this machine. The size defines the size of the partition.
void add_main_store (int size, SysError & err)	This adds more memory to this machine. This physical memory can be used to map more virtual memory.
void remove_backing_store (int device_id, SysError & err)	This removes the disk or disk partition from this node (at least in that it provides backing store). All data currently stored here is migrated to other disks or disk partitions or migrated off this node.
void remove_main_store (int size, SysError & err)	This reduces the physical memory on this node. All data currently using this physical memory is copied to some other pages or migrated off this node.
Node Info	
int num_backing_store_devices (SysError & err)	This returns the number of devices providing backing store present on this node.

int
get_backing_store_devices
(struct back_store data[],
int max_num,
SysError & err)
	Up to max_num backing store records will be filled into the data array.
	
	The back_store structure looks as follows:
	
	struct back_store {
	int device_id
	int size
	}
	
	The current number of devices is returned.

int memory_size (SysError & err)	This returns the current size of physical memory for this node.

void info (struct proc_info & proc_data, char os_version[], char nsap[], char serial_num[], SysError & err)	This returns information on the processor for this node. The proc_info structure includes information on processor type, FPU information, etc. Other information on the current OS version, as well as its network service access point and serial number, are also filled into the provided buffers (each of these fields has a known fixed length).
Virtual Memory Management	
int core_map_size(SysMemory * memory, SysError & err)	This returns the number of pages within this memory object that are residing in main store when this function was called.
int get_core_map(SysMemory * memory, void * core_map[], int core_map_size, SysError & err)	This call updates the core_map array with the address of each page residing in memory as part of the memory object. The pages dedicated to the memory object, may be the base memory object's pages or the surrogate memory object's pages. The value returned is the actual number of pages in core. This number will most likely be different than the core_map_size.
void lock_memory_pages(SysMemory * memory, void * page_list[], int num_pages, SysError & err)	This function will lock the list of pages into memory. The list of pages are those pages within the memory object that will never be swapped out of main memory. The addresses in the page_list array are the addresses of the start of the page. The page must exist at least partially within the memory object.
void unlock_memory_pages(SysMemory * memory, void * page_list[], int num_pages, SysError & err)	This function will unlock the list of pages from being present in main memory. The addresses in the page_list array are the addresses of the start of the page. The page must exist at least partially within the memory object.
Events	
void set_event_thread (int event_num, SysThread * thread, SysError & err)	The thread specified is designated to handle the event identified by event_num. An event is an asynchronous hardware event from some peripheral device that is handled by a hardware interrupt vector. Only interrupts that are unused by the underlying operating system (like parallel ports, serial ports, etc.) are given event_num values. When an event occurs, that thread will be resume'd if it is suspended.
SysThread * event_thread (int event_num, SysError & err)	This returns the thread associated with the specific event number.

Query	
static int get_nodes(SysNode * nodes[], int max_num, SysError & err)	This function will return the number of nodes in the system. Up to max_num will be stored in the nodes array. The actual number of nodes will be returned.
static int num_nodes(SysError & err)	This returns the number of nodes in the system.
n_state node_state(SysError & err)	This returns the state the node currently is in: ACTIVE - currently has threads SHUTDOWN - cannot be communicated with
Threads	
int num_threads(SysError & err)	This returns the number of threads currently executing on this node.
int get_threads(SysThread * threads[], int max_num, SysError & err)	This returns the threads that are executing on this node, up to the max_num listed. The actual number currently executing is also returned.
Statistics	
int statistics (int & percent_idle, int & percent_main_store, int & percent_backing_store, int & main_store_capacity, int & backing_store_capacity, SysError & err)	This returns information about the processing, utilization and capacity of this node. This information could be used to determine when threads should migrate from this node to another.

2.7.3 C Interface

The following definitions correspond to the C++ functions defined above and provide identical functionality. Note that all references to system structures are through opaque types.

#include <sombrero/sys_node.h>

Creation

sys_node **sn_add_node**(*const char* * nsap_octet_string, *sys_error* & e);
void **sn_shutdown**(*sys_node* n, *int* mode, *sys_error* & e);
void **sn_remove_node**(*sys_node* n, *sys_error* & e);

Resources

void **sn_set_memory_map**(*sys_node* n, *sys_memory* map,
void * physical_address, *sys_error* & e);
void **sn_add_backing_store**(*sys_node* n, *int* device_id, *int* size, *sys_error* & e);
void **sn_add_main_store**(*sys_node* n, *int* size, *sys_error* & e);
void **sn_remove_backing_store**(*sys_node* n, *int* device_id, *sys_error* & e);
void **sn_remove_main_store**(*sys_node* n, *int* size, *sys_error* & e);

Node Info

int **sn_num_backing_store_devices**(*sys_node* n, *sys_error* & e);
int **sn_get_backing_store_devices**(*sys_node* n, *struct back_store* data[], *int* max_num,
sys_error & e);
int **sn_memory_size**(*sys_node* n, *sys_error* & e);
void **sn_info**(*sys_node* n, *struct proc_info &* proc_data, *char* os_version[],
char nsap[], *char* serial_num[], *sys_error* & e);

Memory Management

int **sn_core_map_size**(*sys_node* n, *sys_memory* m, *sys_error* & e);
int **sn_get_core_map**(*sys_node* n, *sys_memory* m,
void * core_map, *int* core_map_size, *sys_error* & e);
void **sn_lock_memory_pages**(*sys_node* n, *sys_memory* m,
void * page_list[], *int* num_pages, *sys_error* & e);
void **sn_unlock_memory_pages**(*sys_node* n, *sys_memory* m,
void * page_list[], *int* num_pages, *sys_error* & e);

Events

void **sn_set_event_thread**(*sys_node* n, *int* event_num, *sys_thread* t, *sys_error* & e);
sys_thread **sn_event_thread**(*sys_node* n, *int* event_num, *sys_error* & e);

Query

int **sn_num_nodes**(*sys_error* & e);
int **sn_get_nodes**(*sys_node* nodes[], *int* max_num, *sys_error* & e);
struct n_state **sn_node_state**(*sys_node n, sys_error* & e);

Threads

int **sn_num_threads**(*sys_node* n, *sys_error* & e);
int **sn_get_threads**(*sys_node* n, *sys_thread* threads[], *int* max_num, *sys_error* & e);

Statistics

int **sn_statistics**(*sys_node* n, *int &* percent_idle, *int* & percent_main_store,
int & percent_backing_store, *int* & main_store_capacity,
int & backing_store_capacity, *sys_error* & e);

3 OS Extensions Interface

Sombrero could be implemented as an extension to an existing OS, rather than a completely new OS. In that case, it will have to co-habitate with the existing operating system functions and allow communication between Sombrero's single address space and the per-process address space.

3.1 Accessing Sombrero-space

The key extension to allow a process to be loaded and created in Sombrero-space is to modify the 'exec' or equivalent OS functions. These functions would recognize a new 'magic number' within the file that was to be executed which indicated that the file was a Sombrero program.

There also needs to be a mapping between a "user" and a SysPrincipal. For every user in the base OS there will be a SysPrincipal, and the user who executed the Sombrero program will become the SysPrincipal associated with that new thread.

To allow a Sombrero program to be accessed from Sombrero-space by an external program, the desired Sombrero operating environment's name space must be presented externally. A simple technique would be to present the Sombrero operating environment's naming tree as an externally mounted file system or separate drive. A Sombrero operating environment's name-space server would act as a device driver for the external file system. This device driver would present memory objects as files. File protection would be presented as if a SysThread operating in the file server domain and owned by the SysPrincipal corresponding to the user doing the file access was doing the read and/or write to this memory object.

A Sombrero program and an external program can generally communicate in the same way as any program communicates with an external program, which is through a device. For example, to communicate with a remote external program, an RPC interface running over TCP/IP could be used. The key is that communication occurs through a device driver over a physical medium that is shared between each program.

In the case of a Sombrero program communicating with a co-resident program in the same CPU, the mechanism for communication can be the basic shared physical medium (the physical memory). This means that a Sombrero program may attach to shared memory (in this case shared with a non-Sombrero program) to pass data between itself and a non-Sombrero program.

Extensions to existing OS calls	
exec(...)	In addition to any existing "magic numbers", this system call will also check for a new "magic number" indicating that this is a Sombrero program, and create a SysThread in Sombrero space with this user as its SysPrincipal.
create(...) open(...) read(...) write(...) close(...) ...	The system calls that support the file system access device drivers for the disk, floppy drive, tty, etc. will now be able to access the device driver for a Sombrero operating environment name space.

A Sombrero program must be able to launch non-Sombrero programs. This is achieved by simply invoking the 'exec' command or equivalent on the file specified. This file must be truly in the file system (as opposed to a memory object in a Sombrero operating environment name service).

3.2 Accessing Traditional OS Services

A Sombrero program must also be able to access the old file system, and it does so using the standard file system commands. In essence, all old style system calls are available to a Sombrero program just as they would be available to any program.

Most existing system calls would be identical but some would have slightly different semantics or be illegal for a Sombrero-space program. An example of modified calls would be those which create new threads. An example of an inappropriate call is "brk" since it implicitly assumes a private virtual address space.

Accessing the new system calls presented in this document is only allowed from programs running in Sombrero-space.

4 Usage Scenarios

4.1 Native Programs

The following scenarios describe how a system that had a native Sombrero operating system (i.e. was not an extension of an existing OS such as NT) could implement the basic functionality that a user would expect to see. For details on Sombrero as an extension to an existing many-address space operating system, see the previous section.

4.1.1 User Perspective

4.1.1.1 Logging On

Sombrero will be running an operating environment that has threads that process logon requests by users. The operating environment could implement a logon thread (or threads) as follows:

The logon thread would register with the device handler thread that was monitoring every device that can support logon sessions (this would include the keyboard of a particular node, as well as modem lines, or other external communication lines).

The device handler thread (probably set up as a "cyclic" task using deviceThread -> SysThread::set_cyclic_task(), so that it will not miss events) will detect a logon request.

The logon thread would then be informed of the event using logonThread -> SysThread::interrupt().

The now awakened logonThread will then retrieve the data from the device handler thread. The logonThread now determines that the password is correct and determines who it is, and what SysPrincipal they are.

A new thread is created to manage the initial shell, desktop, or other user interface program that will now communicate over the device being logged in over.

The thread will be created using SysThread::create(), and will be set to have the appropriate SysPrincipal as its owner using SysThread::set_owner(). The stack will be created for the thread using SysMemory::alloc(), and the location of the initial "user interface program" will be determined.

When the thread has been prepared, the thread will be started using newThread -> SysThread::resume(). This new thread will be executing in the domain of the "user interface program", and be owned by the SysPrincipal who just logged on, and communicating with the device that the person just logged in over.

The user is logged on.

4.1.1.2 Executing Programs

When launching a new program the calling program can either wait or continue, depending on the desire of the user and the needs of the application. The simplest case is when the calling program waits on the response. This would be implemented as a simple procedure call within the calling program. No new thread is created, and the old program performs no additional processing until the new program is complete.

If the calling program needs to continue processing while the other program is running, a new thread would be created using SysThread::create() with the starting address of the new program specified and a new stack. The stack would be set up using SysMemory::alloc(). This new thread would then execute the program desired.

4.1.1.3 Communicating With Devices

Communicating with device drivers is simply a matter of calling the appropriate routine defined within the device driver domain. An operating environment should provide device drivers with a unifying input/output model (like a STREAMS head).

4.1.1.4 Logging Off

When a user exits his "user interface program", that thread will exit using SysThread::exit(). The "user interface program" will no longer be communicating to the logon device, and the user is off the system.

4.1.2 Developer Perspective

4.1.2.1 Designing Programs

A developer who is designing a program within the Sombrero OS has additional choices to make compared to traditional OSs.

The most important choice is a choice of trust.

If this program is designed to rely on other libraries or routines (particularly user-specifiable programs), the question of how important protection is, must be decided. If the called program is untrustworthy (maybe buggy, maybe malicious) then a new thread context must be created (including a new stack) and all access to the current threads memory must be denied to the new thread. A safe means of re-entering this domain must be provided, and the return location of the new thread must be set to that location. It is assumed that library functions or macros can be used to define a system call as an untrusted call. If the program was trustworthy, simply calling the routine specified is sufficient.

If this program is designed to be used by others, the question of security must again be asked. If the calling program is untrustworthy (buggy or malicious) a new domain is required for this program (or routine). Entering at this entry-point will cause a domain switch and then all the input data can be verified and the security of the program's internal data can be maintained. A domain switch back to the calling program will be required and will be set up using SysMemory::set_switch(). If the program doesn't care who calls it, the memory object can just be created so that any domain can include the memory object with execute permissions. The trust relationship is defined at the domain crossing boundary.

Other choices about communication need consideration. The basic communication technique is shared memory, but higher level libraries may give the designer other semantic choices (although they will all ultimately use shared memory, the implementation is hidden behind a protection domain).

4.1.2.2 Compiling Programs

Compiling programs in Sombrero is subtly different than in a multi-address space operating system. The programs will likely be compiled into object code in a form where all static memory references are symbolic references that will be resolved at link time. Options will likely be provided on the compiler that puts constants into a separate memory object with a different protection domain. Other options may include flags to specify how heap space is maintained (i.e., within one memory object, many objects, etc.). There will also need to be options to tell the compiler which functions are providing entry points, and which functions are can be late-binding (at run time, or on access failure). Late-binding functions will have additional data to indicate the symbol (typically a name that could be looked up in the name server) and the actual address. When the symbol is resolved, the address is filled out. If the instruction is executed and the location specified is no longer executable, the policy handling code for this memory or thread would check the symbol (still retained in the code space), re-ask the name-server and, if found, modify the linked program with the new address.

4.1.2.3 Linking Programs

Linking programs involves the creation of at least one, but probably more, entry points into this program. These entry points are then published via the name server so that other programs can determine the address of each routine. At the same time, all the corresponding object modules are combined and the symbolic references resolved. It is the linker that creates the memory object that will give a fixed address that can be used to resolve all the symbolic references throughout the different object modules.

Linkers could be written in such a way as to create a single large memory object will all the data from the object modules copied into it and the symbols resolved, or it could be written to reference the object modules through the name server, and resolve the symbols when an appropriate function is called.

The linker will also have options to create a new domain for this executable or attach it to an existing domain. It will also have options to tell the operating environment who has permissions to execute its entry points.

There is nothing stopping a user from individually linking each object separately and having appropriate entry points in each object module that the other object modules call, but this may be unnecessary name space pollution for functions that should not be called externally.

4.1.2.4 Shipping and Installing Programs

Once a program is developed, it needs to be shipped. During the link process it is possible just to merge object files such that a single object file is created from the set of object files. The symbolic names would still be resolved. This is the code that would be distributed to another Sombrero system. When the new program is installed, the linker would be invoked and a new executable created at a new location, resolving the symbols accordingly.

4.1.3 Administrator Perspective

4.1.3.1 Adding and Removing Users

Adding a user is done by doing a SysPrincipal::add(). In addition to the principal, other information would be needed by the operating environment, including this user's logon name, password, initial "user interface program", etc. It would also be at this time that quotas for this user would be defined using SysPrincipal::set_quotas(). It is likely that most operating environments will create a new domain for this principal so that his or her data can be protected independently from all other users by doing SysDomain::declare().

A user is removed by removing the associated principal with the SysPrincipal::remove() call. The domains, threads, and objects owned by this principal will automatically be removed.

4.1.3.2 Adding and Removing Devices

Device drivers that are not required by the Sombrero System Services will be built outside the Sombrero System Services. This includes all devices other than those serving as network connections to other Sombrero nodes, or those acting as main memory or backing store. Examples of devices defined at user level are: keyboards, mice, video display terminals, printers, modems, external network interface cards, and computer controlled machinery.

These device drivers will be created as separate threads in the system (per device or set of devices). The hardware itself is memory mapped using SysMemory::alloc() and SysNode::set_memory_map(). The memory object is protected using a new domain created using SysDomain::declare() and then associated with the memory using SysMemory::set_typcial_region() and SysMemory::set_typcial_permission() such that the new domain is the only domain with access to this memory. This new domain will get another memory object associated with it using SysMemory::create(), which will hold the

thread code, including entry points to routines that support device requests and event notification. These entry points are set up using SysMemory::set_switch(). These entry points will be called by applications communicating over the device.

Removing devices is simply a matter of deleting the domain representing this device, using SysDomain::destroy(). Removing the domain will remove any memory that isn't otherwise referenced and kill any threads running in that domain.

4.1.3.3 Adding and Removing Nodes

When administrators want to add a new node to the existing Sombrero network, they would use SysNode::add_node(). This would specify the network identity and other information.

Shutting down a node would be accomplished using SysNode::shutdown(), and removing it is accomplished using SysNode::remove_node().

4.1.3.4 Adding and Removing Software

Adding new software is simply a matter of linking the appropriate object files. A new memory object with the appropriate domain, and entry points is created.

Removing software is simply a matter of dereferencing the memory object that holds the linked program. This would normally be accomplished by a request to the operating environment name server, which would call SysMemory::dereference(), to disassociate the memory object. If the memory object was still being used by some other domain the memory object would remain.

5 Comparison with other system call interfaces

5.1 Opal

5.1.1 Interface Comparison

The Opal interface is documented in general terms with example system calls, but is not specified completely. This section compares what is provided in Opal to the structures in Sombrero. For a complete reference on Opal see [Ch*95].

5.1.1.1 Data Storage

In Opal, storage is provided with a "Segment". The calls discussed are:

Opal	Sombrero	Description
SegmentRef ::NewSegment()	SysMemory ::alloc()	The memory object is the entity that holds the virtual addresses, just as the Segment in Opal holds the virtual addresses.
SegmentRef ::FirstAddress()	SysMemory ::data()	Both of these are used to get the starting address of the range of addresses in this memory object.
SegmentRef ::LastAddress()	SysMemory ::length()	Rather than the last address, the length (in bytes) is provided by Sombrero.
SegmentRef ::WriteEnabled()	SysMemory ::num_regions() ::get_region_list()	In Sombrero, a direct test of accessibility is expected to be implemented in the operating environment. The tools to show what the kernel currently allows are the num_regions() and get_region_list(). This may be insufficient without operating environment information, because the operating environment policy code may set up regions on demand (lazy evaluation) rather than upon creation of the segment.

In general a Segment is roughly equivalent to a memory object. A memory object is Sombrero also manages the reference functionality rather than the ResourceGroups in Opal. A SysMemory in Sombrero also provides facilities to divide and merge memory objects, facilities for recoverability, backing storage, association with protection domains, ownership, multiple node consistency and replication.

5.1.1.2 Execution

In Opal, execution facilities are provided with "Thread". The calls discussed are:

Opal	Sombrero	Description
Thread ::NewThread()	SysThread ::create()	This simply creates a new thread in both systems. In Sombrero it is assumed more manipulation of the thread will be performed and so the thread begins in the suspended state.
Thread ::Join()	SysThread ::monitor_thread() ::monitored_thread_change() ::suspend()	Rather than an explicit wait, which is what Opal provides, Sombrero allows a set of threads to be monitored (including for exit status) and then to suspend itself until the event occurs.

A Thread in Opal provides a subset of what a SysThread in Sombrero supplies at least with respect to modeling an executing entity. A thread in Sombrero also has the following additional functionality: retrieving thread state information, controlling the thread (through suspend, resume, terminate, etc), interrupt processing and control, private domains, ownership, thread monitoring (providing thread synchronization and multiplexing), scheduling facilities (real time scheduling), and node migration.

5.1.1.3 Protection

In Opal, protection facilities are provided with a "Domain". The calls discussed are:

Opal	Sombrero	Description
Domain ::NewDomain()	SysDomain ::declare()	Both Sombrero and Opal have a domain that identifies access privileges to memory areas.
Domain ::Attach()	SysMemory ::set_specific_permission() ::set_typical_region() ::set_typical_permission()	Unlike Opal, the granting of rights in Sombrero is done specifying the memory object and referencing the domain, instead of specifying the domain and referencing the memory object. Also unlike Opal, different regions within a single memory object can have different permissions. Otherwise, the facilities are logically equivalent.
Domain ::Detach()	SysMemory ::set_specific_permission() ::set_typical_permission()	In Sombrero, the same system calls that allow access rights for a given domain, also can deny access rights for a given domain.
Domain ::Destroy()	SysMemory ::destroy()	Both Sombrero and Opal allow the removal of a protection domain from the system.

A Domain in Opal and a SysDomain are very similar. They both describe what domain permits access to which memory. A SysDomain in Sombrero also supports ownership, as well as a special gateway entry point to let a domain act as an agent on behalf of an object in

Sombrero. It does not include mechanisms to attach or detach memory; that functionality is performed by the memory object granting or revoking permission for domains.

5.1.1.4 Communication

In Opal communication facilities are provided with a "Portal". The calls discussed are:

Opal	Sombrero	Description
Domain ::registerPortal()	SysMemory ::set_switch()	In Sombrero, there is no independent entity that represents something that causes a protection domain switch; it is just an address. When that address is accessed by execute, read, or write, a switch occurs because of the RPLB hardware.
PortalSwitch()	none	A switch to another domain is simply executing a subroutine in Sombrero.
Domain ::RebindPortal()	SysMemory ::set_switch()	The same function that allows an address to cause a switch is also the function that can remove the switch.
Domain ::UnregisterPortal()	SysMemory ::set_switch()	The address causing the switch can be removed with these system calls.

Opal has a special entity to provide switches between domains. In Sombrero the entity is just an address within a SysMemory. A call is still required to set up the switch, but the hardware takes care of actually performing the switch (an explicit switch is not required). This is the fundamental difference between Opal and Sombrero.

5.1.1.5 Resource Control

In Opal resource control facilities are provided with a "ResourceGroup". The calls discussed are:

Opal	Sombrero	Description
ResourceGroup ::NewResourceGroup()	SysPrincipal ::create()	A principal in Sombrero provides a grouping and access mechanism to all the resources in the system.
Thread ::PushResourceGroup()	SysThread ::set_owner()	These set the owner of the thread, and objects created by this thread default to this owner.
Thread ::PopResourceGroup()		
ResourceGroup ::Destroy()	SysPrincipal ::remove()	Both of these remove the owning entity.

In Opal, ownership for resource accounting is provided by the ResourceGroup. In Sombrero it is provided by the Principal. In Opal, there can be persistence of an object just by being part of a resource group. This is not true in Sombrero. In Sombrero, a memory object must be referenced by a protection domain to remain in the system. If it ever becomes unreferenced, it will be removed from the system and the Principal will no longer have that memory object. There is no need for an object to persist if it has no references to it. If a object that currently has no access permissions is desired to be persistent , the SysMemory::reference() call can be used to associate the domain with the memory.

5.2 Solaris 2.4

Solaris 2.4 is a complete industrial operating system. Its system call interface includes many major facilities that have been included in Unix variants as Unix as evolved. These system calls support thousands of applications.

This section compares the system calls provided in Solaris to equivalent facilities in Sombrero. For a complete reference see [SUN*94]. The functions listed in Sombrero either perform a similar function or can be put together to perform a similar function. The comments against each system call indicate how Sombrero differs and why.

The Sombrero system call interface is complete enough to allow all the functionality in Solaris to be implemented using the Sombrero system calls.

5.2.1 Command Comparison

Solaris 2.4	Sombrero	Comment
_exit	SysThread::exit()	Both cause a thread to exit .
_lwp_cond_broadcast	SysThread::resume() SysThread::suspend()	Condition variables and mutex variables are not implemented in the Sombrero System Services. They must be implemented in the operating environment. The broadcast and signal functionality can be implemented by the operating environment by maintaining the list of threads waiting on the condition variable and invoking resume on the correct thread or threads. The wait and timed wait can be implemented by the operating environment by putting the thread on the list it maintains and suspending the thread. Mutex variables are implemented using the test-and-set instructions in the processor.
_lwp_cond_signal		
_lwp_cond_timedwait		
_lwp_cond_wait		
_lwp_continue	SysThread::resume()	Both resume the thread.
_lwp_create	SysThread::create()	Creates the thread in the system. The wait_flag in Sombrero is equivalent to the DETATCHED flag in Solaris. In Sombrero, all threads begin in the suspended state and must be explicitly resumed.
_lwp_exit	SysThread::exit()	Causes the thread to exit.
_lwp_getprivate	SysThread::get_memento() or void *	Supporting private data specifically associated with a kernel structure is supported generically in Sombrero by the memento.

		An operating environment could also build a higher level construct using a SysThread and then store private data for its purposes there.
_lwp_kill	SysThread::terminate() SysThread::abort() SysThread::exit() SysDomain::destroy()	These all cause the thread to stop executing. Only exit is guaranteed to stop the thread immediately; terminate and abort can both call handlers first. To protect a domain's data, a domain may limit who is authorized to kill a thread. A runaway thread in a domain that has limited authorization requires killing the thread from within the domain it's in, or requires the domain to be deleted.
_lwp_makecontext	SysThread ::create() ::activeDomain() ::threadDomain() SysMemory ::alloc() ::set_typical_region() ::set_typical_permission()	This sets up the initial state for a new thread. In Sombrero, the operating environment will package the creation of a new thread using the basic system calls as follows: Create the thread and get its new protection domain. Associate a new stack with it by creating a memory object and giving that thread's protection domain permission to access it. Initialization of this startup state can be as simple or as complex as the operating environment desires.
_lwp_mutex_lock	none	These commands are intended to be implemented in the operating environment since they require no system services and can be implemented using the processor's test-and-set instruction.
_lwp_mutex_trylock		
_lwp_mutex_unlock		
_lwp_self	SysThread ::currentThread()	These just return the thread currently executing.
_lwp_sema_init	SysThread::resume() SysThread::suspend()	Like condition variables, these functions are expected to be built on top of the resume and suspend thread operations, with the operating environment maintaining the lists of interested threads, and controlling the implementation of the semaphore.
_lwp_sema_post		
_lwp_sema_wait		
_lwp_setprivate	SysThread	This just stores a pointer to data

	::set_memento()	associated with this thread (as can be done for any Sombrero object).
_lwp_suspend	SysThread::suspend()	Both suspend the thread, but in Sombrero the thread can resume executing upon an event other than receiving a resume (such as monitoring a thread, or receiving an exit, etc.)
_lwp_wait	SysThread::thread_state() SysThread::free()	This gets statistics on a thread (including an exited thread) and explicitly frees the thread from the system.
access	SysMemory ::containing() ::num_regions() ::get_region_list() ::get_memento() SysThread ::activeDomain() ::threadDomain() ::get_memento()	The access command and many other commands in Solaris are related to the file system, which is one of the main shared resources in the Solaris environment. In Sombrero, the only shared resource is memory, and so file-like mechanisms will be built on top of the memory object. The operating environment will still want to validate access and allow a pre-check to determine if access is allowed. The operating environment has many tools from which to make this decision, including regions already associated with the memory object being addressed, and any data the operating environment may have associated with this thread or memory (for instance pointers to ACLs and/or capabilities).
acct	SysThread ::authorize() ::exit() SysDomain ::set_gateway()	Sombrero can support arbitrary accounting methods including (as Solaris does) writing out information when a thread terminates, by developing this code in the operating environment. An operating environment which wanted to perform extra processing on a system call would deny that system call to threads created in its environment, and would specify a protection domain that it trusted as the authorized domain. That domain would have a gateway that the Sombrero System Services would invoke upon execution of the exit system call by an unauthorized

		thread. The code within the authorized domain would then perform the additional accounting and perform the exit on the thread. This ability to extend the functionality of a system call is performed uniformly throughout Sombrero, and this is just an example of how it could be used.
adjtime	SysThread ::set_cyclic_task()	Sombrero in and of itself does not care about the time of day. The facility to keep the time of day is required in an operating environment, however. Implement ing an accurate clock in an operating environment requires a thread that is guaranteed to be executed at a given time slice. This is supported by Sombrero through the use of a cyclic task. The data this task maintains can be used to provide time of day services, including adjusting the time of day.
alarm	SysThread ::set_cyclic_task() ::interrupt()	To implement an alarm service an operating environment could create a domain which had a cyclic task that could interrupt the thread interested in the alarm.
audit	SysDomain ::set_gateway()	Auditing functions trap events that someone may want to track. This type of functionality is meant to be developed in a Sombrero operating environment, not in the services themselves. To track access to system calls, the authorized domain for any of the objects controlled by this operating environment should be a domain that is part of the operating environment. This allows control of all system calls to be funneled through this operating environment, and allows counting or auditing to be performed.
auditon		
auditsvc		
brk	SysMemory ::set_needed_size()	In Solaris, a process has a 32bit virtual address space that the process owns completely and which by using "brk" can be set to increase the accessible range of addresses, which when accessed would get

		actual physical storage behind it. In Sombrero, the memory object has some virtual address space (user defined) and the set_needed_size() sets the explicit backing store requirements.
chdir	none	Sombrero does not have a notion of a file system.
chmod	SysMemory ::set_typical_region() ::set_typical_permission() ::set_specific_region_permissions()	Protection in Sombrero is at a memory object level instead of at a file level as in Solaris. The same kinds of read, write, or execute permissions can be granted, but rather than to fixed entities such as world, group, and owner, they are given to specific protection domains.
chown	SysMemory ::set_owner()	This transfers ownership of the memory object to another user. Unlike Solaris, there is no relationship between ownership and protection. Ownership is strictly a resource accounting function.
chroot	none	This is purely a file system concept.
close	none	The need to open and close files is purely a file system concept. Access to memory objects is always available within Sombrero and is purely restricted by the protection domains. Having a pointer is enough to access the memory.
creat	SysMemory ::alloc() ::reference()	While this function is for files, the need to create persistent data or programs is still required. In Sombrero, this is achieved by creating a memory object and referencing that memory object by a domain (which could be a domain implementing a name service).
dup	none	This is purely a mechanism to move identifiers for a single file. In Sombrero, all memory is simply accessed by its address.

exec	SysThread::create()	These functions all load a new context for a given thread so that it begins executing a new program. The starting address is specified during the creation of the thread by the creator. See the discussion on _lwp_create.
execl		
execle		
execlp		
execv		
execve		
execvp		
exit	SysThread::exit()	Exits the thread.
fchdir	none	See comments for chdir.
fchmod	SysMemory ::set_typical_region() ::set_typical_permission() ::set_specific_region_permissions()	See comments for chmod.
fchown	SysMemory ::set_owner()	See comments for chown.
fchroot	none	See comments for chroot.
fcntl	none	The fcntl command does a mixed bag of file control activities, including locking regions of a file, freeing space in a file, manipulating file descriptors, and controlling file close semantics. In Sombrero, the locking functions would be implemented by the user programs (using mutex test-and-set type instructions). Backing store is controlled by set_needed_size (see discussion on brk) and the other functions are purely file related.
fork	SysThread::create() SysMemory::alloc()	The fork command has no direct mapping, since much of its semantics are bundled with the inherent new virtual address space achieved by a new process. In Sombrero, however, a new thread is explicitly created. If it needs its own memory space, it will have to use alloc to get a new memory object.
fork1	SysThread::create() SysMemory::alloc	See fork comments.
fpathconf	none	Purely a file system query function.
fstat	SysMemory ::length() ::get_needed_size() ::num_regions() ::get_region_list()	This function returns information about files. Files don't exist in Sombrero, but there will be memory objects holding data. These memory objects do have owners

	::get_owner() ::replication_mode() ::has_surrogate()	and other properties that can be queried.
fstatvfs	none	This is purely a file system function, which is not present in Sombrero.
getaudit	none	Sombrero supports audit functionality in the operating environment. See the discussion on audit.
getauid	none	An audit user ID could be implemented in the operating environment by including additional information in the thread using the memento mechanism, even if the actual owner was transferred.
getcontext	SysThread ::get_context()	Sombrero provides an explicit system call for the retrieval of the thread state information. During a thread interrupt (for whatever reason) the former context of this thread at its the interrupted location is available on the thread's stack.
getdents	none	This is purely a file system and name service function, which is an operating environment feature, in Sombrero.
getegid	none	Group IDs are a policy concept that can be implemented at the operating environment level in Sombrero.
getgroups	none	Groups which are membership lists are implemented at the operating environment level in Sombrero.
getitimer	SysThread ::get_max_exec_time()	There is an overlap of the interval timer functionality and the alarm signal. See the discussion on alarm for how Sombrero could provide real-time interrupts. For process time related interrupts, the get_max_exec_time returns the amount of time the thread will execute before an interrupt is performed. This time is measured in total thread CPU cycles. If the thread desires to continue execution the max_exec_time would have to be bumped up, since there is no real timer to reset. Unlike Solaris, there is no distinction between thread time and system time on behalf of the

		thread in Sombrero.
getmsg	SysThread ::set_queue_task() ::set_cyclic_task() SysNode ::set_event_thread	The semantics of reading from an event stream such as a STREAMS head is implemented in Sombrero using shared memory and/or a protected interface provided by the device driver in cooperation with the operating environment. The Sombrero System Services do not have device drivers implemented within it. It does allow device drivers to be written outside the Sombrero System Services using the facilities to add cyclic based threads and queue based threads, as well as specific threads to be updated on hardware interrupt events that the Sombrero System Services do not care about.
getpgid	none	There is no concept of process groups, within the Sombrero system services. Any kind of grouping needed can be developed at the operating environment level.
getpgrp	none	See comments on getpgid.
getpid	SysThread ::currentThread()	The currentThread function returns a pointer to the SysThread of the currently executing thread.
getpmsg	none	See comments on getmsg.
getppid	none	There is no concept of parent processes in Sombrero. Any kind of hierarchy would need to be developed at the operating environment level.
getrlimit	none	The resource limit in Sombrero is the 64bit address space.
getsid	none	Process groups, sessions, and controlling terminals are implemented if needed at the operating environment level.
getuid	SysThread::owner()	Both indicate the owner of the thread (i.e., on whose behalf it is executing and whose quota its counted against).
ioctl	none	This is basically the dumping ground for device-related commands. Some functions are common to multiple types of devices or classes of devices (such as STREAMS, or terminals) but basically there is a

		wide variety of device control functions. In Sombrero, these device control functions can be implemented at the operating environment level by providing a protection domain that limits access to the hardware and protects the buffers and control structures. Sombrero allows queue-based and cyclic threads, so that I/O will not be lost. It even supports threads that are awakened due to a hardware interrupt. These tools allow arbitrary device drivers to be added without affecting the Sombrero System Services, or requiring control functions be embedded inside a single command (such as ioctl). A simple uniform access to these devices with identical semantics to other devices is still important, it just becomes an operating environment issue.
kill	SysThread::exit()	This simply exits the thread.
lchown	SysMemory::set_owner()	See comments for chown.
link	none	This is for creating aliases in the file system, which does not exist at the Sombrero System Services.
llseek lseek	none	Positioning within a file is as simple as specifying the correct address in Sombrero, since all data is mapped to the 64bit virtual address space.
lstat	SysMemory ::length() ::get_needed_size() ::num_regions() ::get_region_list() ::get_owner() ::get_replication_mode() ::has_surrogate()	See comments on fstat
mincore	SysNode ::core_map_size() ::get_core_map()	Information about what memory pages are in core can be retrieved with these functions.
mkdir	none	Directories are a function of a naming service, which is not provided in the Sombrero system services but rather in an operating environment.

mknod	SysNode ::set_memory_map()	This is used for creating a hook for adding a new device. This is provided in Sombrero by making a specific memory map and assigning physical addresses.
mmap	SysMemory ::alloc()	Rather than being a strange case, the memory mapping of data that can persist indefinitely is the only case in Sombrero. A memory object is a range of addresses that will remain as long as it is referenced by a domain. No file is necessary.
mount	none	Mounting a file system (either local or remote) is a function that would be provided by a Sombrero operating environment module. It would be implemented in much the same way that a device driver is implemented. See discussion under ioctl.
mprotect	SysMemory ::set_typical_region() ::set_typcial_permission() ::set_specific_region_permissions()	Both these functions provide protection for access of memory within this memory region. In both cases the protection is provided by hardware, but on Solaris protection is only in reference to one virtual address space.
msgctl	none	These are used for setting up, maintaining, and using message queues. In Sombrero, all messages are sent using shared memory. Higher level libraries can be built with semantics similar to message queues using shared memory, but message queues are not necessary in the Sombrero System Services.
msgget		
msgop		
msgrcv		
msgsnd		
munmap	SysMemory ::dereference()	This is the explicit method to remove a memory range. A SysMemory object will also be implicitly removed if it is no longer referenced by any protection domain.
nice	SysThread ::set_multitask()	These allow simple changes in a tasks priority.
open	none	There is no need to open a file in

		Sombrero, a process may simply access the appropriate address. If file-like semantics were desired, a higher level library could be built using the existing system calls.
p_online	SysNode ::shutdown() ::add_node()	This is how a node is shut down or turned up. Multiple processors on a single workstation are currently assumed to be separate SysNodes. This may have to be reconsidered later.
pathconf	none	See discussion under fpathconf.
pause	SysThread ::suspend()	This simply suspends the thread until it is either explicitly resumed or an event causes it to be restarted.
pipe	none	A communication mechanism with the semantics of a pipe is not provided in Sombrero. A communication mechanism like this could be built using shared memory.
plock	SysNode ::lock_memory_pages()	This allows specific pages within a memory object to be permanently locked into the main store.
poll	none	I/O multiplexing is supported by Sombrero in the operating environment. Since device drivers are implemented at the operating environment layer, techniques to wait on a set of devices for an event should be developed at that layer. It is expected that in calling a "poll"-like function, the thread doing the calling will pass into another protection domain and have itself registered. The shared memory that provided the communication mechanism would be scanned, and if no events were present, the thread would be suspended with a timeout. When the event came in, the registered thread would be resume()'ed.
pread	none	Reading data in Sombrero is as simple as doing a read from the address at which the data is located. No special call is necessary and no copying is necessary.
priocntl	SysThread ::set_multitask() ::set_queue_task() ::set_cyclic_task()	These options control thread scheduling. The options within Solaris are for REAL_TIME and TIME_SHARE classes of threads.

		These roughly correspond to a thread that has had called set_cyclic_task() or set_multitask() respectively. The addition of set_queue_task() is to make processing high priority event queues from devices easier. The restrictions on a set_cyclic_task() make it possible to create keyboard-reset tasks to kill queue based tasks that somehow get into an infinite loop, and to provide exact timing (within some bounded slop-time). This area probably deserves more research but these types of scheduling are needed for threads in Sombrero that implement device drivers.
priocntlset	SysThread ::set_multitask() ::set_queue_task() ::set_cyclic_task()	See comments for priocntl.
processor_bind	SysThread ::modify_node_list()	This allows threads to be associated with a single processor. In Sombrero, a set of nodes can be specified, where a node is assumed to be a single processor.
processor_info	SysNode ::info()	These function calls return information about the processor. In Sombrero information about the backing and main stores is also returned.
profil	none	A kernel level histogram is not provided in Sombrero. Profiling work can be done by other techniques (such as auto-code instrumentation, or execution interval timers).
ptrace	none	The ptrace system call is purely for debugging programs. Debugging in a single address space environment is trickier, since intrusive debugging (like inserting break point instructions) affects all threads executing this code. An intrusive debugging facility can be developed using the existing Sombrero system calls. To support getting notification of a thread hitting a breakpoint, the debugger would ask the authorized domain (i.e.,

		operating environment) to pass all interrupts to the debugger. The debugger can then examine the target thread's state to get its registers (using interrupt_stack()). Using the PC, the instruction being executed can be determined by just reading the address (there is only one address space!). The debugger can also explicitly resume the thread to "continue" the program.
putmsg	none	Message queues are not provided in Sombrero. All communication is through shared memory. A communication mechanism with the same semantics as message queues can be developed using shared memory.
putpmsg	none	See comments for putmsg.
pwrite	none	Writing data at a position is as simple as writing to an address in Sombrero.
read	none	Reading data in Sombrero is as simple as doing a read from the address at which the data is located. No special call is necessary and no copying is necessary.
readlink	none	Symbolic links are a function of naming but a name server is intended to be developed on top of these system calls.
readv	none	Reading data in Sombrero is as simple as doing a read from the address the data is located at. No special call is necessary and no copying is necessary.
rename	none	Changing the name of a file is a name service function, which is not in the Sombrero system services.
rmdir	none	Directories are also a name service function and not in the Sombrero System Services.
sbrk	SysMemory ::set_needed_size()	See comments for brk.

semctl	none	Semaphores are not provided by the basic Sombrero System Services. They can be implemented using shared memory and the processor's test-and-set instruction.
semget		
semop		
setaudit	none	See comments on getaudit.
setauid	none	See comments on getauid.
setcontext	SysThread ::set_context()	Sombrero provides a function to build and set a thread context (PC, registers, condition codes).
setegid	none	Effective group IDs are a policy concept that can be implemented at the operating environment level in Sombrero.
seteuid	none	Effective user ids are a policy concept that can be implemented at the operating environment level.
setgid	none	Group IDs are a policy concept that could be developed at the operating environment level if desired.
setgroups	none	Groups which designate membership lists are not present in Sombrero but could be implemented at the operating environment level if desired.
setitimer	SysThread ::set_max_exec_time()	For process time related interrupts the set_max_exec_time specifies the total thread CPU cycles before an interrupt will occur.
setpgid	none	There is no concept of process groups within the Sombrero System Services. Any grouping could be done at the operating environment level.
setpgrp	none	See comments on setpgid.
setrlimit	none	The resource limit in Sombrero is the 64bit address space.
setsid	none	Process groups, sessions, and controlling terminals are implemented if needed at the operating environment level.
setuid	SysThread ::transfer()	This is how ownership of a thread is changed.

shmat	SysMemory ::alloc() ::set_typical_region() ::set_typical_permission() ::set_specific_region_permissions()	These shared memory functions in Solaris are implemented as the one and only communication method in Sombrero. Because Sombrero is a single address space operating system no shared memory IDs, or special instructions to attach and detach it are necessary. The memory is simply present, and it may or may not be accessible.
shmctl		
shmdt		
shmget		
shmop		
sigaction	SysDomain ::set_gateway()	In Sombrero, all events (things normally thought of as signals in Unix) are sent to the authorized domain's gateway function. The stubs documented against each class are for each of the types of events that can occur. The action taken by the code in the authorized domain can be simple or complex. It could also include executing thread specific routines that the thread originally registered with the authorized domain (i.e. operating environment).
sigaltstack	none	No alternate stack is provided for a thread processing an interrupt. Custom stacks can be implemented where more than one stack is in a memory object but a protection domain should be set to deny read or write access from the area dedicated to this thread's stack (so an inadvertent overflow does not occur).

sighold	SysDomain ::set_gateway() SysThread ::suspend()	Autonomous events (or just unexpected events) can be detected by the Sombrero System Services. These events are always routed to the authorized domain of the appropriate system object (events are indicated throughout this document as stubs against the class where the event would occur). It is up to the authorized domain to provide routines to ignore an event, mask events, and provide ways to look at what events are pending for its clients. To wait simply requires the thread to be suspended.
sigignore		
signal		
sigpause		
sigpending		
sigprocmask		
sigrelse		
sigsend		
sigsendset		
sigset		
sigsuspend		
sigwait		
stat	SysMemory ::length() ::get_needed_size() ::num_regions() ::get_region_list() ::get_owner() ::get_replication_mode() ::has_surrogate()	See comments on fstat.
statvfs	none	See fstatvfs comments.
stime	none	System time is not maintained by the Sombrero System Services.
swapctl	SysNode ::add_backing_store() ::remove_backing_store() ::info()	In Sombrero, there really isn't swap space, but there is backing store for all objects. The backing store is maintained using the node manipulation calls.
symlink	none	Symbolic links are part of a name service and that is outside of the Sombrero System Services.
sync	SysMemory ::flush() SysNode ::shutdown()	In Sombrero, each memory object can be flushed to disk separately. If the whole system is shutdown (depending on the mode), all the memory objects on this node will be flushed.
sysfs	none	This is purely a file system function, which does not apply to Sombrero.
sysinfo	SysNode ::info()	This gets processor specific information (such as OS version, network identifier, and serial number)
time	none	Sombrero does not track the calendar date.
times	SysThread::thread_state()	These both get the real time and

		CPU usage time from the thread.
uadmin	SysNode::shutdown()	This allows a node in Sombrero to be turned off or rebooted.
ulimit	SysPrincipal ::set_quotas() ::get_quotas()	The means to limit system resource usage can be enforced in Sombrero by setting quotas on various system resources for a principal.
umask	none	Specification of default policy for newly created entities is an operating environment issue and is not defined in Sombrero.
umount	none	Sombrero leaves higher level concepts such as file systems at the operating environment level. Those features can be developed using memory objects and building appropriate device drivers.
uname	SysNode ::info()	A string representing the system name can be constructed from the information returned by the "info" function.
unlink	SysMemory::dereference()	While files as such do not exist in Sombrero, file-like structures will be implemented using SysMemory. When the "file" is freed the "dereference" call will be used to clean up the space used.
ustat	none	This is completely file system related, and file systems do not appear in the Sombrero System Services.
utime	none	Setting a timestamp when a file-like structure is updated or created is an important feature, but is left to the operating environment to create, when it creates the file semantics.
utimes	none	See comments for utime.
vfork	SysThread ::create() ::alloc()	See comments on fork.
vhangup	none	Controlling terminals are issues that will be dealt with at the operating environment level.

wait	SysThread ::thread_state() ::free() ::monitor_thread() ::suspend()	A thread can wait for the completion of another thread by monitoring that other thread for an exit state, and suspending itself. When that other thread does exit, it can then retrieve the other thread's state and free the thread that exited.
waitid		
waitpid		
write	none	Writing is as simple as setting a value at a memory location. There is no need for a write system call.
writev		
yield	SysThread::suspend()	The suspend call in Sombrero can be used to implement functionality equivalent to yield, by specifying the correct parameter.

6 References

[Sk*96a] Skousen, A. C., Miller, D. S. and Feigen R., "The Sombrero Operating System an Operating System for a Distributed Single Very Large Address Space, General Introduction", ASU 64-bit OS Group White Paper 0, ASU Computer Science and Engineering Department Technical Report TR-95-006, April 1996, also available at http://www.eas.asu.edu/~sasos.

[Sk*96b] 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 White Paper 2, ASU Computer Science and Engineering Department Technical Report TR-95-013, January 1996.

[Sk*96c] Skousen, A. C. and Miller, D. S., "Implementing A Very Large Single Address Space Across Multiple Nodes: Memory Partitioning, Protection Domain Migration, Kernel Replication, Consistency and Fault Tolerance", ASU 64-bit OS Group White Paper 10, ASU Computer Science and Engineering Department Technical Report TR-95-021, January 1996.

[Bo*91] Booch, G. *Object Oriented Design with Applications*. Benjamin/Cummings Publishing Company, Inc., 1991.

[Ch*95] 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).

[SUN*94] Sun Microsystems Inc. *Solaris 2.4 Manual Pages*, 1994.