IMPLEMENTATION OF A LOWER LEVEL ARCHITECTURE

FOR THE

SOMBRERO SINGLE ADDRESS SPACE OPERATING SYSTEM

by

Donald B. White

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

ARIZONA STATE UNIVERSITY

December 2005

IMPLEMENTATION OF A LOWER LEVEL ARCHITECTURE

FOR THE

SOMBRERO SINGLE ADDRESS SPACE OPERATING SYSTEM

by

Donald B. White

has been approved

August 2005

APPROVED:

_____, Chair

_____

_____

Supervisory Committee

ACCEPTED:

_____

Department Chair

_____

Dean, Division of Graduate Studies

ABSTRACT

Prior research indicates that the Sombrero Operating System offers a model for computing that reduces complexity in the implementation of both system and application software. Much of this research, though not all, relied on simulation. This thesis reports on an extensive effort to develop and implement for Sombrero common features found in more conventional operating systems. The goal was to render a judgment, if only in a preliminary manner, as to whether or not the benefits of Sombrero demonstrated in theory were realizable in practice.

A good software development environment facilitates good software development. The effort to install a Sombrero development system has been reduced and thereby the coding and testing of Sombrero system modules was made easier and previously undocumented aspects of Sombrero have been documented. This is an area where Sombrero will continue to lag because of the small development community.

This effort also exploits Sombrero's object-oriented development model to provide an implementation of a lower-level architecture for the Alpha platform. Software representations of the hardware system's major components have been implemented. The existing middle level architectural modules now use the lower-level services.

Two device drivers were implemented: a real-time clock driver and serial port driver. A standard input-output library was developed to ease use of the serial port driver. Address resolution protocol and internet control message protocol were added to the Sombrero protocol stack. The performance of the protocol stack was closely examined with attention to its passive service architecture and thread switching method.

This is dedicated to my wife, Diane. She has had to endure almost a decade of lost weekends and evenings as I pursued this degree. I would have quit years ago without her encouragement and support.

ACKNOWLEDGMENTS

Thanks to Dr. Donald Miller and Dr. Alan Skousen. They kept me motivated through the long learning and development effort that lead to this thesis. Thanks to the committee for their suggestions and encouragement. Thanks to the other members of the research group for their contributions.

TABLE OF CONTENTS

Page

LIST OF FIGURES xiii

CHAPTER

1 INTRODUCTION 1

2 DEVELOPMENT ENVIRONMENT 4

2.1 Common Development Environment 4

2.1.1 Boot Loader 5

2.1.2 PALCode 5

2.2 Development Tools 6

2.2.1 Object Strip 6

2.2.2 Arc 6

2.2.3 ARCDOS 6

2.2.4 SOSRMake 7

2.3 Sombrero Ethereal Dissector 7

2.4 Revision Control 7

2.4.1 Central Repository 8

2.4.2 Remote Access to Repository 8

2.4.2.1 Apache with WebDAV 8

2.4.2.2 TortoiseSVN 8

2.5 Simplified Configuration 9

2.5.1 Enhanced Sombrero Boot Loader 9

2.5.2 Peripheral Component Interconnect Bus Scan 9

CHAPTER Page

2.5.2.1 Polled UART Driver and Standard Input-Output Library 10

2.5.2.2 Simple Real-Time Clock 10

2.5.2.3 Address Resolution Protocol Support 10

3 LOWER LEVEL SOMBRERO ARCHITECTURE 11

3.1 Mainboard 12

3.1.1 Service map_irq 12

3.1.2 Service map_irq_to_ipl 12

3.2 Processor 12

3.2.1 Service read_ICSR 12

3.2.2 Service write_ICSR 13

3.2.3 Service read_MCSR 13

3.2.4 Service write_MCSR 13

3.2.5 Service read_amask 13

3.3 System Core Logic 13

3.3.1 Service start_timer 13

3.3.2 Service enable_irq 14

3.3.3 Service disable_irq 14

3.3.4 Service get_isa_interrupt 14

3.3.5 Service pci_cfg_io_setup 14

3.3.6 Service pci_cfg_io_status 15

3.4 ISA Bus Controller 15

3.4.1 Service init8259as 15

CHAPTER Page

3.4.2 Service enable_isa_irq 15
3.4.3 Service disable_isa_irq 15
3.4.4 Service ack_isa_irq 16
3.4.5 Service set_irq_trigger_mode 16
4 SOMBRERO INTERRUPT PROCESSING ARCHITECTURE 17
4.1 Interrupt as Synchronization Object 17
4.2 Interrupt Threads 19
4.3 PALCode Support 20
4.3.1 Schedule Interrupt Thread 20
4.3.2 Block Interrupt Thread 20
4.3.2.1 Promote Thread 21
4.3.2.2 Block Thread and Demote 21
4.3.2.3 Block Thread and Switch 21
4.3.2.4 Demote Thread 22
4.3.3 Unblock Interrupt Thread 22
4.4 Interrupt Thread Models 22
4.4.1 Active Service Model 22
4.4.2 Passive Service Model 23
4.4.3 Model Selection 23
4.5 Evaluation 24
4.5.1 Latency 24
4.5.2 Thread to IRQ Mapping 25

CHAPTER Page

4.5.3 Shared Interrupts 25

4.5.4 Multiplexed Interrupts 25

5 DEVICE DRIVER ARCHITECTURE 27

5.1 PCI Bus 28

5.1.1 Service reserve_device 28

5.1.2 Configuration Address Space Access 28

5.1.3 Input-Output Address Space Access 28

5.2 Network Card Driver 28

5.2.1 Service DRV3C905Send 29

5.2.2 Service DRV3C905Receive 30

5.2.3 Service GetMacAddress 31

5.3 Real Time Clock 31

5.3.1 Service get_time 31

5.4 Industry Standard Architecture Bus 32

5.4.1 Service register_isa_irq 32

5.4.2 Service unregister_isa_irq 32

5.4.3 Service wait_for_interrupt 32

5.5 Serial Port 33

5.5.1 Service ioctl 34

5.5.2 Service read 34

5.5.3 Service write 35

6 SOMBRERO PROTOCOL COMPONENT ARCHITECTURE 36

CHAPTER Page

6.1 Link Layer Protocol 37

6.1.1 Service EtherSend 37

6.1.2 Service EtherRecv 38

6.1.3 Service GetMacAddress 38

6.2 Address Resolution Protocol 38

6.2.1 Service NetMACLookup 39

6.3 Internet Protocol 39

6.3.1 Service IPOut 39

6.3.2 Service IPIn 39

6.4 Internet Control Message Protocol 40

6.5 User Datagram Protocol 40

6.5.1 Service UDPOut 40

6.5.2 Service UDPIn 41

6.6 Sombrero Protocol Driver 41

6.6.1 Service MsgToHost 41

6.6.2 Service WaitForMsgFromHost 42

6.6.3 Service MsgFromHostTimer 43

6.6.4 Service DataToHost 43

6.6.5 Service DataFromHost 44

7 SOMBRERO PROTOCOL DESCRIPTION AND EVALUATION 45

7.1 Protocol Stack Assembly 45

7.2 Message Protocol Description 46

CHAPTER Page

7.3 Burst Protocol Description 46

7.4 Evaluation 48

7.4.1 Performance 48

7.4.2 Thread Switching 49

7.4.3 Thread Interaction 50

7.4.4 Data Movement 50

7.4.5 Transmit to Receive Transition 51

8 STANDARD INPUT/OUTPUT LIBRARY 53

8.1 Service gets 53

8.2 Service puts 53

8.3 Service printf 53

8.4 Service scanf 53

8.5 Service sprintf 53

8.6 Service sscanf 54

9 CONTRIBUTIONS AND CONCLUSIONS 55

10 REMAINING WORK 56

REFERENCES 58

APPENDIX

A SOMBRERO DRIVER NUMBERING SCHEMA 60

B SOMBRERO PROTOCOL THREAD SWITCHING LOGIC 62

C SOMBRERO DEVELOPMENT ENVIRONMENT 64

CHAPTER Page

D BOOTLOADER DEBUGGER 71

E INTERRUPT THREADS 76

LIST OF FIGURES

Figure Page

1. Original Sombrero Development Environment 4
2. Current Sombrero Development Environment 5
3. Lower Level Architecture 11
4. Device Driver Architecture 27
5. Sombrero Protocol Stack 37

1 INTRODUCTION

In [15 p1], Skousen states his decision to "build it [Sombrero] directly on the processor." He chose not to follow other researchers who have "built their experimental operating systems on top of one of these MASOSs [Multiple Address Space Operating Systems] to take advantage of the services they provide so as to not reinvent the wheel." Skousen's objective was "to better take advantage of SASOS [Single Address Space Operating System] features" and "to achieve an object oriented modular system design." Skousen describes the "middle level architecture put in place to achieve the current goal." This thesis investigates the implementation implications of the Sombrero middle level architecture.

These three characteristics, run on the hardware, object oriented modular system and middle level architecture, provide the motivation for some of the research underlying this thesis. Sombrero [17][13] needs many wheels to be reinvented. Among these wheels, Sombrero needs an object oriented modular lower level architecture. This thesis begins the work of providing one. The design of the lower level architecture takes a direct approach. Software objects were defined which mirror the components of the Sombrero hardware platform. The relationships between the software objects reflect the physical interconnections of the components. The final step was to "port" the middle level architecture to this new software platform.

The development of the lower level architecture itself motivated a further area of research. The installation, configuration and use of a Sombrero development system exhibited little of the ability of Sombrero "to reduce the complexity of system and application software" [15 piii] found elsewhere in the code. Some, but not all, of the

difficulties have been removed. This thesis reports on implementation progress in this area.

Sombrero uses an unconventional interrupt processing architecture. In Sombrero, interrupts are synchronization objects much like semaphores or locks. A thread desiring to service an interrupt simply waits on the interrupt object. When the hardware interrupt occurs, the interrupt object is signaled and the thread resumes execution. Thus, this architecture permits interrupts to be serviced within a thread context. At the time of [15], Sombrero included an interrupt object and thread only for the system clock. Subsequently, Skousen replaced the polled network interface card driver with an interrupt-driven version using an interrupt synchronization object. Additionally, this driver incorporated the Sombrero concept of a passive service. This thesis examines critically and documents the resulting Sombrero protocol stack.

This thesis includes and describes the design and implementation of an assorted collection of “reinvented wheels.” There is a real-time or time-of-day clock driver. There is a serial port driver. The serial port driver is an active service that uses its own, not the caller’s, thread to service interrupts. This provides an alternative model for device driver implementation compared to the network interface card driver. A simple standard input-output library uses the services of the serial port driver to provide formatted reads and writes to a terminal device.

Finally, this thesis documents some previously undocumented aspects of Sombrero. There is a description of the interactions by which the Sombrero protocol

layer drivers assemble themselves into a protocol stack. The message and data burst protocols provided by the Sombrero protocol stack are also described.

2 DEVELOPMENT ENVIRONMENT

Skousen [15 p50] describes the Sombrero development environment. The environment is shown in Figure 1. This environment requires two separate development platforms: one for the boot loader and PALCode and one for the Sombrero program class objects. Moreover, while the boot loader and PALCode were developed on the same Windows NT for Alpha platform, each required its own set of development tools. The following paragraphs describe enhancements to the original development environment that were part of this research.

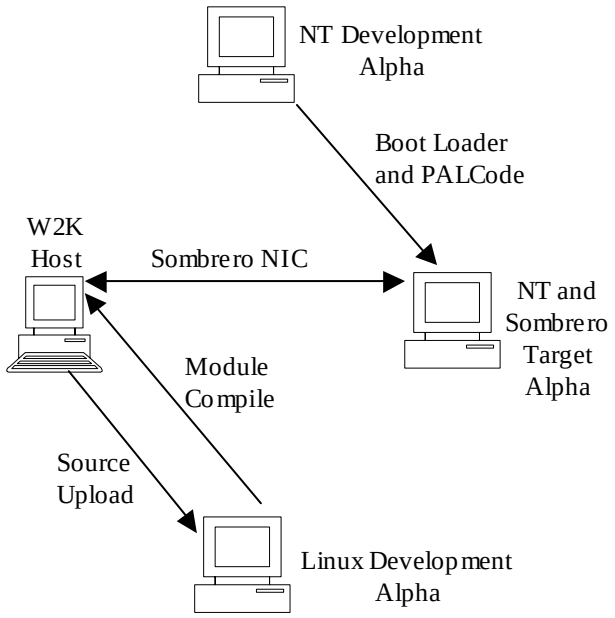


Figure 1: Original Sombrero Development Environment

2.1 Common Development Environment

The need for a Windows NT for Alpha platform was removed from the Sombrero development environment. The simplified environment is shown in Figure 2. This simplification was accomplished through the adaptation of tools from the Alpha Mini-Loader (MILO) project. MILO is boot loader developed to boot Linux for Alpha.

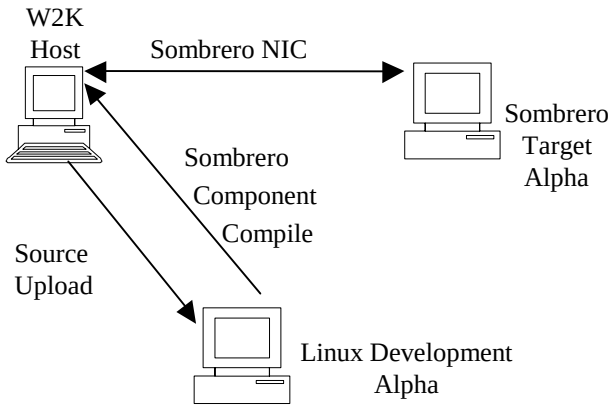


Figure 2: Current Sombrero Development Environment

2.1.1 Boot Loader

Skousen [15 p57] describes the boot loader development environment. This environment used the Windows NT Device Driver Development Kit (DDK) and the Advanced RISC Computing (ARC) Software Development Kit (ASDK) from Microsoft. These development kits provide header files and object libraries for creating programs that are directly executable in the AlphaBIOS runtime environment. While functional, this environment had two limitations that negatively impacted the development of the Sombrero boot loader. Firstly, this environment provided basically a 32-bit compiler with non-standard 64-bit extensions. Secondly, it provided an ARC [14] rather than a C language runtime environment. The boot loader development environment was replaced with the GNU C compiler [10] on the Linux development Alpha. A small object library was developed to provide a limited but standard C runtime environment.

2.1.2 PALCode

Skousen [15 p57] describes the original Sombrero PALCode development environment. This environment used the Evaluation Board Software Development Kit (EBDSK) from Digital Equipment Corporation [6]. The EBSDK provided a port of the

GNU C compiler to Windows NT. The EBSDK was replaced with the GNU C language compiler. All extensions present in the EBSDK compiler for generating PALCode were now present in the compiler version provided with the Linux Development Alpha.

2.2 Development Tools

The replacement of the ASDK and the EBSDK required the development of some new tools and the adaptation of several open-source development tools. These tools support the development of ARC programs on a Linux Alpha system. The following tools were developed or ported as part of this research.

2.2.1 Object Strip

Object strip is a tool for extracting the text and data sections as a binary file from an Executable and Linkable Format (ELF) executable. This permits the compiled code to be repackaged within another object file format. This tool was adapted from MILO.

2.2.2 Arc

Arc is a tool for repackaging a stripped object file into the ARC executable format that the AlphaBIOS can load and execute in the ARC runtime environment. This tool was adapted from MILO.

2.2.3 ARCDOS

ARCDOS is a standalone ARC program that provides simple services such as copying files from a floppy disk to a hard disk. This program permits the Sombrero boot loader and PALCode to be installed on the Sombrero target Alpha without another operating system installed. This tool was a demonstration program in the ASDK.

2.2.4 SOSRMake

SOSRMake is a Windows command script that transfers a collection of source files and a makefile from the Windows Host system to the Linux Development Alpha, executes the makefile and transfers the created executables back to the Windows Host system. This script is used as a custom build command within the Visual C++ development system to compile programs on the Linux Development Alpha. It is similar to the SOSRBuild [15 p55] utility used to compile Sombrero executables. With an appropriate makefile, it is used to compile the boot loader, the PALCode or the previously described utility programs. This script was developed as part of this research.

2.3 Sombrero Ethereal Dissector

Ethereal is an open-source protocol analyzer. In the Sombrero environment it is used on the Sombrero host to record message traffic between the Sombrero host and the Sombrero target. It is a valuable tool for investigating the correctness and performance the Sombrero LAN protocol. Ethereal is extensible through the addition of dynamically loaded dissector modules. A module was developed as part of this research that dissects both the Sombrero message protocol and the Sombrero data-burst protocol.

2.4 Revision Control

Prior versions of Sombrero were distributed as a directory tree contained in a compressed file. The revision history consisted of a numbered series of these compressed files. A better technique was required to support further Sombrero development by multiple researchers. The following paragraphs describe development capabilities that were implemented as part of this research.

2.4.1 Central Repository

A central Sombrero repository was created using the open-source Subversion version control system [3]. This repository holds the complete Sombrero source. Additionally, it holds all the tools, scripts and instructions necessary to create a Sombrero development platform. This repository permits all further changes to Sombrero to be controlled and recorded. Within the repository, each researcher may create branches for private or provisional lines of development. Successful efforts are merged to the Sombrero mainline.

2.4.2 Remote Access to Repository

This repository is located on a server maintained currently by this researcher [19]. All members of the Sombrero research group have access to this repository. Access is controlled through use of the Secure Socket Layer protocol. All interactions with the Sombrero repository server are encrypted.

2.4.2.1 Apache with WebDAV

The Subversion version control system includes a WebDAV (Web-based Distributed Authoring and Versioning) interface module for use with the Apache web server. WebDAV is a set of extensions to the HTTP protocol that allows users to edit and manage files on web servers. This interface provides the translation of HTTP commands and URLs into actions on the repository. This capability provides Sombrero researchers with full access to the repository using a suitable client.

2.4.2.2 TortoiseSVN

The open-source TortoiseSVN [12] project provides client access to the Sombrero repository. TortoiseSVN extends Windows Explorer include access to Subversion

through HTTP or HTTPS protocols. TortoiseSVN adds additional menu items supporting manipulation of controlled objects. Files and directories may be checked out, modified and committed to the repository.

2.5 Simplified Configuration

The configuration of the Sombrero development and execution environments was simplified. This simplification was accomplished through two means. Sombrero was enhanced to probe the target system to determine its hardware environment. Sombrero was enhanced to use common network protocols to determine the Sombrero network environment.

2.5.1 Enhanced Sombrero Boot Loader

Skousen [15 p58] describes the Sombrero boot sequence. With the exception of the removal from the sequence of the somload.exe program, the Sombrero boot sequence remains unchanged. However, many refinements have been made as part of this research to the Sombrero boot loader itself. These refinements make the Sombrero system easier to configure and use. The following paragraphs describe these enhancements.

2.5.2 Peripheral Component Interconnect Bus Scan

The boot loader was enhanced to perform a Peripheral Component Interconnect (PCI) bus scan to locate the network interface card. After locating the PCI bus slot containing the network card, the boot loader can read the PCI configuration address space for the network card and correctly configure the driver for the network card. Previously, a Sombrero target system was required to have its network card in a designated bus slot.

2.5.2.1 Polled UART Driver and Standard Input-Output Library

A simple serial driver and input-output library were developed. This allowed the boot loader to output debug information using standard C runtime “printf” statements. This output is captured on the host through a null modem cable using a host-based terminal emulation program. This facility enables debugging of the Sombrero protocol stack. Previously, debug information was written to the console display and, therefore, difficult to capture.

2.5.2.2 Simple Real-Time Clock

A simple clock driver was developed. This driver provides a clock tick at one-second intervals. This facility enables the boot loader to time requests for user input. Boot loading can continue after timer expiration using default values.

2.5.2.3 Address Resolution Protocol Support

The protocol stack in the boot loader was enhanced to both generate and respond to address resolution protocol (ARP) queries. In support, the network card driver provides the ARP protocol with the Ethernet address programmed into the network card during manufacture. This facility enables the Sombrero host and target to dynamically discover the other’s Ethernet address. Previously, these addresses were statically configured within each system.

3 LOWER LEVEL SOMBRERO ARCHITECTURE

Skousen [16] presents the fundamental concepts of Sombrero: its single address space and protection mechanisms. Skousen [15 p21] describes a middle level architecture for Sombrero. Moreover, Skousen chose "to build it [Sombrero] directly on the processor in order to better take advantage of SASOS features." This choice motivated this researcher's efforts to develop a lower level software architecture to support Sombrero directly on the hardware and, in doing so, to investigate the implementation implications of the Sombrero middle level architecture. The lower level architecture consists of a collection of Sombrero services representing the components of the hardware environment.

The following paragraphs describe the elements in this collection. In these descriptions a class refers to a program class object (PCO). The Sombrero compilation process creates a PCO. An instance refers to an instantiated program object (IPO). The boot loader creates the IPO from the PCO during system startup. A service refers to a callable entry point into the IPO. A service performs some processing for the calling

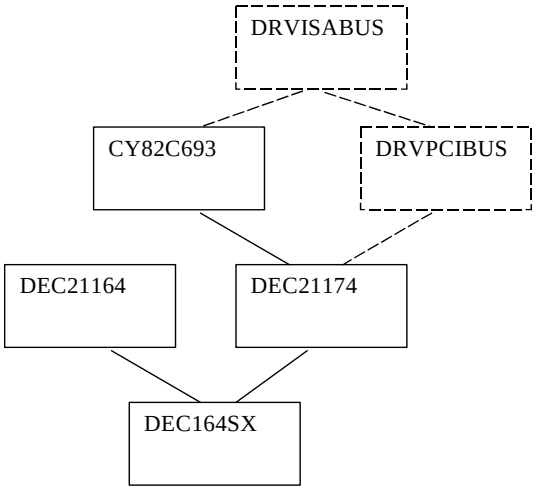


Figure 3: Lower Level Architecture

thread. The relationships of the classes in the lower level architecture are shown graphically in Figure 3.

3.1 Mainboard

The Sombrero target system uses an AlphaPC 164SX mainboard. This board is described in [4]. Within Sombrero, an instance the DEC164SX class represents this mainboard. This instance is an abstraction of the inter-chip wiring.

3.1.1 Service map_irq

This service maps a device on the PCI bus to a system interrupt number. This reflects the wiring between the PCI slot and the 21174 core logic chip.

3.1.2 Service map_irq_to_ipl

This service maps a system interrupt number to system interrupt priority level. This reflects the interrupt routing established by the AlphaBIOS and the wiring between the processor and the core system logic. The interrupt priority level associated with an interrupt request is called the targeted interrupt priority level.

3.2 Processor

The Sombrero target system uses a 21164PC processor. This processor is described in [7]. Within Sombrero, an instance of the DEC21164 class represents this processor. This instance provides access to the processor's configuration and status registers.

3.2.1 Service read_ICSR

This service returns the current value of the processor's instruction and decode unit (IDU) control and status register (ICSR). From the Sombrero perspective, this register controls superpage mapping, the enabling of PAL operation codes in kernel

mode, and the enabling of the PAL shadow registers. Superpage mapping is an addressing mode especially for Windows NT kernel mode.

3.2.2 Service write_ICSR

This service writes a new value to the processor's IDU control and status register.

3.2.3 Service read_MCSR

This service returns the current value of the processor's memory address translation unit (MTU) control register (MCSR). From the Sombrero perspective, this register controls superpage mapping.

3.2.4 Service write_MCSR

This service writes a new value to the processor's MTU control register.

3.2.5 Service read_amask

This service returns the results of the execution of an architecture mask instruction. This instruction reports the Alpha architecture features that are present in the processor.

3.3 System Core Logic

The Sombrero target system uses the DEC 21174 core logic chip. This chip is described in [8]. Within Sombrero, an instance of the DEC21174 class represents the system core logic. The system core logic includes the system interrupt controller, PCI bus controller, the memory controller and the system clock.

3.3.1 Service start_timer

This service initializes and starts a periodic system timer with a 10-millisecond interval. This timer provides the system clock tick to the Sombrero thread scheduler.

3.3.2 Service enable_irq

This service enables a system interrupt request number. The Sombrero middle level component SOSInterrupt [15 p48] uses this service to enable a system interrupt for which there is a registered driver.

3.3.3 Service disable_irq

This service disables a system interrupt request number. The Sombrero middle level component SOSInterrupt [15 p48] uses this service to disable a system interrupt for which there is no registered driver.

3.3.4 Service get_isa_interrupt

This service returns the number of the last industry standard architecture (ISA) bus interrupt request. In the Sombrero target system, the fifteen ISA bus interrupts are multiplexed through system interrupt request seven. The core logic chip has logic to query the ISA bus controller for the interrupting ISA interrupt request number. This service sends an acknowledgement to and reads ISA interrupt request number from the ISA bus controller.

3.3.5 Service pci_cfg_io_setup

The service configures the action that the core logic will take upon a fault during access to the PCI configuration address space. The Sombrero PCI bus driver, described below, uses this service during its initial scan of the PCI bus to disable faults. This permits the reliable detection of empty bus slots without the installation of a fault handler.

3.3.6 Service pci_cfg_io_status

This service returns any accumulated error status from prior accesses to the PCI configuration address space. The Sombrero PCI bus driver also uses this service during its initial scan of the PCI bus to determine if a read of a PCI configuration address space address returned valid data.

3.4 ISA Bus Controller

The Sombrero target system hardware provides an ISA bus through use of a PCI to ISA bus bridge chip, the CY82C693U [5]. This chip includes a pair of cascaded, 8259A compatible, programmable interrupt controllers and a real time clock. In the Sombrero target system hardware configuration, the real time clock interrupt is not connected to an ISA bus interrupt controller. It is connected to system interrupt request six. This differs from a typical industry-standard personal computer. Within Sombrero, an instance of the CY82C693 class represents the ISA bus controller.

3.4.1 Service init8259as

This service initializes the two cascaded programmable interrupt controllers. Initially, all ISA interrupts are configured as edge triggered and disabled.

3.4.2 Service enable_isa_irq

This service enables an interrupt within the appropriate programmable interrupt controller.

3.4.3 Service disable_isa_irq

This service disables an interrupt within the appropriate programmable interrupt controller.

3.4.4 Service ack_isa_irq

This service issues a specific interrupt acknowledge command to the appropriate programmable interrupt controller.

3.4.5 Service set_irq_trigger_mode

This service sets the interrupt trigger mode to edge or level mode.

4 SOMBRERO INTERRUPT PROCESSING ARCHITECTURE

Skousen [15 p64] proposes “the use of blocking interrupts that allow threads to block in a registered GPD [General Protection Domain] and provide the context for handling the interrupt. Return from interrupt is a context switch back to the original user thread. This allows user threads to block directly in device handlers supporting the concept of passive services.” At the time, only the Sombrero thread scheduler, SOSScheduler, used this approach to service the system clock interrupt. Subsequently, Skousen developed a passive Sombrero protocol service in which user threads block in the network card driver waiting to service both transmit and receive side interrupts.

This architecture establishes a control structure within which interrupt threads execute. The interrupt thread must cooperate with the Sombrero PALCode to properly service device interrupts. While device drivers are confined within a GPD, a poorly written driver can still affect negatively overall system behavior. The following sections describe more fully the Sombrero interrupt processing architecture. Finally, there is an evaluation of its effectiveness.

4.1 Interrupt as Synchronization Object

Sombrero treats interrupts similar to a synchronization object such as a semaphore. While the analogy is not exact, threads wait on interrupts similar to the manner in which threads wait on semaphores. When the interrupt occurs, the system schedules the first waiting thread similar to signaling a semaphore. Interrupt processing must allow for interrupt sharing, raising the interrupt priority level (IPL), and raising the thread priority level. These are features not normally required in semaphore management.

For each triple of system interrupt request number, targeted interrupt priority level and general protection domain, Sombrero creates an instance of an interrupt vector data structure. This data structure provides the data items necessary to manage the threads waiting for the interrupt. These data structures form a sparse three-dimensional matrix. At each populated entry, there is potentially a list of waiting threads to service interrupts.

The instance of the class SOSInterrupt provides the services to add and delete entries within the interrupt vector matrix. Skousen [15 p48] provides a brief description of this Sombrero system module. As GPDs register to service interrupts, SOSInterrupt computes a system interrupt priority level. The system interrupt priority level is lower than the lowest targeted interrupt priority level for which there may be an interrupt with an associated interrupt thread. That is, the system interrupt priority level is an interrupt priority level at which all hardware interrupts are enabled. It is the interrupt priority level at which the system normally runs when it is not servicing interrupts.

As noted above in the description of the mainboard system module, each system interrupt has an associated interrupt priority level that is called the targeted interrupt priority level. The processor maintains in an internal register the current or actual interrupt priority level at which the system is executing. The processor only recognizes interrupts with a targeted interrupt priority level higher than the current interrupt priority level. Thus, the raising and lowering of the current interrupt priority level within the processor masks and unmasks interrupts based on their associated targeted interrupt priority level. The PALCode can set the processor's current interrupt priority level.

4.2 Interrupt Threads

Servicing interrupts within a thread context is not unique to Sombrero. The use of kernel threads to handle interrupts in the Solaris 2 kernel is described in [11]. The Solaris 2 kernel preallocates partly initialized kernel threads, called interrupt threads. The kernel then converts an interrupt into an executing instance of one of these interrupt threads. "This allows a single synchronization model to be used throughout the kernel" [11]. Providing the interrupt handler with a thread context also permits the full use of kernel functions, even those that can potentially block the caller. This removes a traditional constraint on the logic of interrupt handlers.

In Sombrero, interrupt threads provide the same benefits as in Solaris 2. Additionally, because of its single address space architecture, Sombrero removes the Solaris 2 constraint limiting interrupt threads to kernel threads. Solaris 2 imposes this constraint because "[s]witching between kernel threads does not require a change of virtual memory address space" [11]. Obviously, Sombrero never requires a change of virtual memory address space. Moreover, since in Sombrero any thread can be an interrupt thread, the interrupt thread can possess the complete application context. This enables the passive service model described below. The FreeBSD Architecture Handbook [9], Section 8.3.1 Interrupt Handling, describes a more recent, though less sophisticated, application of interrupt threads.

The Sombrero model inverts the control flow relative to the Solaris 2 model. A Sombrero interrupt thread returns from an interrupt to an existing context. A Solaris 2 interrupt thread begins execution as a function call within a newly created context. A Sombrero interrupt thread invokes the Demote Thread service described below to

indicate the end of interrupt processing. A Solaris 2 interrupt thread returns to its invoking mechanism at the completion of interrupt processing. Interestingly, Solaris 2 provides a Release Interrupt service [11]. After an interrupt thread invokes this service, the thread continues execution as a normal kernel thread. This does not permit the thread, however, to service another interrupt as in Sombrero.

4.3 PALCode Support

The Sombrero PALCode implements the interrupt servicing architecture. There are three primary services: block interrupt thread, schedule interrupt thread and unblock interrupt thread. Device drivers use Block Interrupt Thread to request one of four actions described below. The PALCode uses Schedule Interrupt Thread to select and schedule an interrupt thread. The system thread scheduler uses Unblock Interrupt Thread to schedule a thread that is blocked waiting for interrupt and has an expired timer.

4.3.1 Schedule Interrupt Thread

This service is internal to the PALCode. The processor transfers control to this service when it recognizes an interrupt. This service validates that the current interrupt priority level is lower than the target priority level of the interrupt. It then examines the interrupt vector matrix to find the appropriate entry. If the entry contains a blocked thread, the current interrupt priority level is raised to the targeted interrupt priority of this interrupt and the thread priority is raised to prevent the thread from being preempted while the system executes at this interrupt priority level.

4.3.2 Block Interrupt Thread

An interrupt thread calls this service to change the current interrupt priority level, to lower its thread priority, to wait for an interrupt and to request that the system schedule

another thread to service an interrupt. The following sections describe each of these four actions.

4.3.2.1 Promote Thread

Through the block interrupt thread service, an interrupt thread can request that the current interrupt priority level be raised to the level associated with the interrupt that the thread services. This permits the thread to prepare the device to generate interrupts without the system recognizing the interrupt. Without this action, there would be a race between a thread blocking to service an interrupt it just enabled and an occurrence of the newly enabled interrupt. The scenario is this: 1) the thread enables the device, 2) the device interrupts, 3) the system discards the interrupt because there is no waiting thread and 4) the thread blocks, possibly forever.

4.3.2.2 Block Thread and Demote

Through the block interrupt thread service, an interrupt thread can request that it be suspended to wait for an interrupt. The thread's context is saved and the thread's thread control block is appended to the list of waiting threads for the appropriate entry in the interrupt vector matrix. The processor's current interrupt priority level is lowered if necessary to unmask the interrupt on which the thread is waiting.

4.3.2.3 Block Thread and Switch

Through the block interrupt thread service, an interrupt thread can indicate that it did not service the interrupt and that another thread, if waiting, should be given the opportunity to service the interrupt. If there is another waiting thread, the current thread is appended to the list of waiting threads, the first waiting thread is removed from the list of waiting threads and a context switch is performed to give control to this first waiting

thread. If there is no waiting thread, this service returns immediately with return value indicating that there is no other waiting thread. This allows the current thread to take an alternate action.

4.3.2.4 Demote Thread

Through the block interrupt thread service, an interrupt thread can indicate that it has completed servicing the interrupt. At this point, both the thread priority level and the current interrupt priority level can be lowered. The thread continues to execute but can now perform "bottom-half" processing without masking interrupts. It is important that a device driver quickly lower its interrupt priority level as all PCI and ISA interrupts are processed at the same interrupt priority level.

4.3.3 Unblock Interrupt Thread

When it blocks, an interrupt thread may specify a maximum time that it is to remain blocked. When this is the case, the PALCode makes an entry for the thread in the thread scheduler's timer queue. The thread scheduler examines this queue during servicing of the system clock interrupt. When it finds a thread with an expired timer, the thread scheduler calls this PALCode entry to unlink the thread from the interrupt and reschedule the thread.

4.4 Interrupt Thread Models

The Sombrero interrupt processing architecture supports two models for device driver organization.

4.4.1 Active Service Model

In the active service model, the device driver during system initialization creates a dedicated thread(s) to service device interrupts. User threads interact with the interrupt

thread through input and output queues. Semaphores may be used to block user threads while waiting for the interrupt thread to complete device interactions. The serial port and real-time clock drivers, described in following sections, use this model.

4.4.2 Passive Service Model

In the passive service model, the calling user thread becomes the interrupt thread as it enters the device driver system module. If no user threads are reading or writing the device, then device interrupts may be disabled or ignored. The Sombrero protocol stack and the network card driver, described in a following section, use this model.

4.4.3 Model Selection

The selection of the appropriate model depends on device characteristics and the processing environment. Device characteristics found to be important were status reporting and buffering. The processing environment conditions found to be important were the presence or absence of an activating thread and whether the service was to be blocking or non-blocking.

With the passive model and multiple threads, separate threads may be waiting for transmit and receive interrupts from the device. In this case, driver implementation is facilitated if the device reports transmit and receive status separately so that only control needs to be transferred from one thread to the other. The active model may be more appropriate for a device that reports status once using a destructive read such as the serial port. In this case, the device driver's active thread can then service both transmit and receive interrupts. Some devices have sophisticated support for buffering data. These types of devices can transfer data with little program intervention. An example is the 3C905 network interface card with its support for linked input and output buffers in

system memory. A contrary example is the serial port. The serial port can sustain neither input nor output operations without processor intervention due to its small internal buffers.

The passive model requires, obviously, that there be an activating thread at the times when the device requires service. This may not be the case when the device generates unsolicited events or messages and the active model is more appropriate. The real-time clock driver described below could have been implemented using either model. This researcher decided the service should be non-blocking. This decision resulted in an active model implementation.

4.5 Evaluation

The Sombrero interrupt processing architecture is feasible. Several device drivers were implemented using the architecture. While the Sombrero interrupt architecture differs from more conventional operating systems such as Linux, the architecture itself presents no obstacles to driver implementation.

4.5.1 Latency

The desirability of avoiding context switches conflicts with the need to have a waiting interrupt thread. The interrupt thread cannot be both waiting for the next interrupt and performing its own pre-interrupt device setup processing and post-interrupt bottom-half processing. Using another thread for bottom-half processing reintroduces the context switch that the architecture sought to avoid. In practice the impact may be minimal depending on the device. With respect to pre-interrupt device processing, for example, the 3Com 3C905 [1] network card used in the Sombrero target system maintains a queue of the last 31 transmit status values and stores the receive status in the

buffer descriptor for the received data. The driver can process all stored status values prior to blocking for an interrupt.

4.5.2 Thread to IRQ Mapping

A thread can block for only one interrupt. If the system has multiple devices of the same type, there must be multiple interrupt threads, one for each device. For example, the serial port driver starts two threads, one for each of the serial ports on the Sombrero target system.

4.5.3 Shared Interrupts

The architecture supports the blocking of multiple threads on a single interrupt. Device drivers can use this feature to support multiple devices sharing a single interrupt. The network interface card driver uses this feature. The driver is implemented as a passive service. Different user threads may be active for the transmit and receive sides of the driver. When the device interrupts, the first blocked thread is restarted which may be unable to service the interrupt. The driver will then forward the interrupt to the other waiting thread.

4.5.4 Multiplexed Interrupts

The Sombrero Alpha system hardware multiplexes all ISA bus interrupts through a single PCI interrupt. The Sombrero interrupt processing architecture's support for shared interrupts described previously does not address this situation. The ISA bus controller returns the interrupt number once when the PCI interrupt is acknowledged. This prevents chaining the interrupt threads. It would be inefficient to have all ISA device drivers query either their devices or the interrupt controller whenever any ISA

interrupt occurs. The ISA bus driver described in a following section is a solution to this problem.

5 DEVICE DRIVER ARCHITECTURE

The Sombrero device driver architecture implemented as part of this thesis builds on lower level architecture services and Skousen's interrupt processing architecture to support the development of drivers for devices attached to a Sombrero target systems PCI and ISA buses.

In these descriptions a class refers to a program class object (PCO). The Sombrero compilation process creates a PCO. An instance refers to an instantiated program object (IPO). The boot loader creates the IPO from the PCO during system startup. A service refers to a callable entry point into the IPO. A service performs some processing for the calling thread. The relationships of the classes in the device driver architecture are shown graphically in Figure 4.

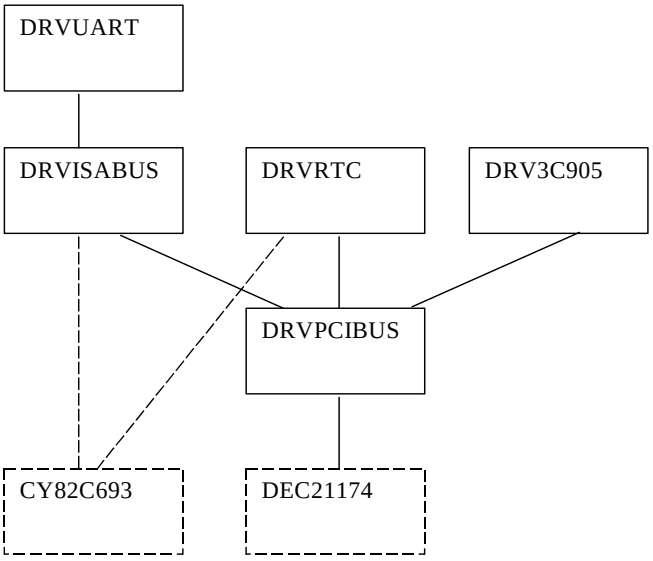


Figure 4: Device Driver Architecture

5.1 PCI Bus

Within Sombrero, an instance of the DRVPCIBUS class represents the PCI bus. This object is the initial owner of the virtual addresses corresponding to the PCI configuration and input-output address spaces. During system initialization, this object enumerates the devices on the PCI bus and validates input-output address ranges assigned to the devices by the AlphaBIOS.

5.1.1 Service reserve_device

This service allows a device driver to locate a device if present on the PCI bus. If the device is present, the service transfers ownership of the device's PCI address ranges and interrupt number to the caller's GPD. This insures that there is at most one driver for each device.

5.1.2 Configuration Address Space Access

This instance provides a collection of services for accessing PCI configuration address space addresses. These services provide device drivers with the ability to read and write to bytes, words and double words within the PCI configuration address space.

5.1.3 Input-Output Address Space Access

This class provides a header file with inline function definitions for accessing PCI input-output address space addresses. These services provide device drivers with the ability to read and write to bytes, words and double words within the PCI input-output address space.

5.2 Network Card Driver

Within Sombrero, an instance of the class DRV3C905 provides a driver for the 3Com 3C905 network interface card. Skousen developed the initial version of this

driver. This researcher enhanced the driver to use the PCI bus driver services. This allowed the network card to reside in any PCI bus slot.

During construction of the instance, the driver creates two rings of buffers: one for transmit and one for receive. These rings are organized as download (from system memory to the network interface card memory) and upload (from network interface card memory to system memory) lists. The 3C905 uses these lists to transfer data to and from its internal transmit and receive buffers.

5.2.1 Service DRV3C905Send

This service accepts a linked list of buffer descriptors. Each buffer descriptor contains the address of the buffer, the number of bytes to be transferred from the buffer and the address of the next buffer descriptor. A null value for the address of the next buffer descriptor indicates that the current descriptor is the last descriptor in the list. Within Sombrero, a list of buffer descriptors is called a buffer chain.

This service first stalls the download function within the 3C905. If there is an empty buffer in the transmit buffer ring, the driver copies the contents of the buffer descriptor list into the first empty buffer in the transmit buffer ring. The driver then restarts the download function in the 3C905 and returns to the caller.

If the transmit buffer ring contains no empty buffers, the driver enables transmit interrupts, restarts the download function in the 3C905 and blocks. When the interrupt occurs the driver examines the transmit status. If no transmit interrupt has occurred, the driver requests that this thread again block and that another thread be scheduled to service this interrupt.

If the status indicates that a transmit interrupt occurred, the driver stalls the download function in the 3C905, copies the contents of the buffer descriptor list to the newly emptied ring buffer, disables transmit interrupts, restarts the download function and returns to the caller.

5.2.2 Service DRV3C905Receive

This service accepts a linked list of buffer descriptors. Each buffer descriptor contains the address of the buffer, the size of the buffer in bytes and the address of the next buffer descriptor. A null value for the address of the next buffer descriptor indicates that the current descriptor is the last descriptor in the list.

This service first stalls the upload function within the 3C905. If there is a full buffer in the receive buffer ring, the driver copies the contents of the first full buffer into the buffers of buffer descriptor list and marks the ring buffer empty. The driver then restarts the upload function in the 3C905 and returns to the caller.

If the receive buffer ring contains no full buffers, the driver enables receive interrupts, restarts the upload function in the 3C905 and blocks. When the interrupt occurs the driver examines the receive status. If no receive interrupt has occurred, the driver requests that this thread again block and that another thread be scheduled to service this interrupt.

If the status indicates that a receive interrupt occurred, the driver stalls the upload function in the 3C905, copies the contents of the newly full ring buffer to the buffers of the buffer descriptor list, marks the ring buffer empty, disables interrupts, restarts the upload function and returns to the caller.

5.2.3 Service GetMacAddress

This service returns the Ethernet address of the network card associated with this instance of the driver. This service was added to support the Ethernet protocol driver.

5.3 Real Time Clock

An instance of the DRVRTC class represents the real-time or time-of-day clock. The clock hardware resides on the CY82C693U chip. The clock interrupt line is connected to system interrupt six. The clock is compatible with the Dallas Semiconductor DS1287 found in industry standard personal computers. This is an example of an active service.

During system initialization, the driver creates an interrupt thread. This thread configures the clock to generate an end-of-update interrupt. This interrupt occurs at one-second intervals. The thread then loops forever. Within the loop, the thread first blocks waiting for the clock interrupt. When the interrupt occurs, the current time is read from the clock chip into local storage. The loop then repeats.

This driver could have been easily implemented as a passive service. A consequence would be that some calls to the get_time service would be delayed. The time is not available from the chip during timer rollover. During these periods, the calling thread would need to wait for the time to become available. The implementation as an active service allows the caller of the get_time service to obtain the last reported time of day without delay.

5.3.1 Service get_time

This service copies the current time from local storage to the caller's storage. The service then returns to the caller.

5.4 Industry Standard Architecture Bus

An instance of the DRVISABUS class represents the industry standard architecture (ISA) bus. This class provides services to support development of drivers for devices on the ISA bus. This driver uses the services of the instances of the CY82C693 and DEC21174 classes described previously to provide a high-level ISA bus abstraction. This driver is a passive service that demultiplexes the ISA interrupts multiplexed over system interrupt seven. The ISA bus driver is basically a system interrupt seven driver.

5.4.1 Service register_isa_irq

This service registers the calling GPD as the owner of the specified ISA bus interrupt request number. It also configures the interrupt to be either edge or level triggered.

5.4.2 Service unregister_isa_irq

This service removes the registration of the calling GPD as the owner of the specified ISA bus interrupt request number.

5.4.3 Service wait_for_interrupt

This service first verifies that the specified interrupt is valid and that the calling thread belongs to the registered GPD for the specified interrupt. This service then checks if it has a pending interrupt that matches the caller's request. If so, it marks the pending interrupt as complete and returns to the caller.

If not, the service unmasks the interrupt in the appropriate programmable interrupt controller (PIC) and blocks waiting for system interrupt seven. The calling device driver is responsible for previously enabling interrupts at the device level. When the system interrupt occurs, this service calls the DEC21174 class instance to acknowledge the

system interrupt and obtain the ISA interrupt number. The service then both acknowledges the ISA interrupt and masks it in the appropriate PIC. If this ISA interrupt number matches the caller's request, the thread returns to the caller. If not, the service marks the ISA interrupt as pending and switches to another thread waiting for an ISA interrupt. When a thread is found that is waiting for a pending interrupt, the service marks the interrupt complete and returns to the caller.

The thread switching logic is similar to that in Appendix B except that this logic uses neither buffer descriptors nor data movement. Pending ISA interrupts are managed as bits in a shadow register.

5.5 Serial Port

An instance of the DRVUART class represents the serial communication ports. The serial port hardware is compatible with the National Semiconductor NS16550A chip found in industry-standard personal computers. The serial port hardware resides in the SMC FDC37C699 chip [18] on the mainboard. The interrupt lines from the serial port hardware are tied to inputs IRQ3 and IRQ4 on the CY82C693U chip. The driver is implemented following the active service model. During system initialization, the driver creates two interrupt threads, one for each serial port.

The decision to implement active services resulted from considerations during data output. Serial devices are slow. Writing to a serial port should not routinely block the user thread. The serial port has a limited output buffer (16 byte FIFO) and requires the processor to refill the FIFO to sustain output. The solution adopted was to allocate a relatively large circular output buffer. The write service normally just copies the user buffer into the output buffer, starts the data transfer and returns. The user thread is

blocked only when there is insufficient space in the output buffer. In that case, the user thread waits on a semaphore for output buffer space to be available. The interrupt thread signals the semaphore when space is available.

On the receive side, the interrupt thread copies the contents of the serial port receive FIFO to a relatively large circular input buffer. The read service satisfies user requests by copying from the input buffer to the user buffer. The user thread waits on a semaphore only when there is insufficient data available. The interrupt thread signals the semaphore when data is available.

5.5.1 Service ioctl

This service allows the caller to query and set serial port parameters. The caller may set the bit rate and the transfer mode. There are two transfer modes: binary and line. In binary mode, the driver transfers data without alteration between the serial port and the user thread. In line mode, the driver transfers lines of text between the serial port and the user thread. During transfer, the driver performs the following actions. On output, the driver replaces new-line sequences with carriage-return, line-feed sequences. On input, the driver only transfers complete lines to the user thread. It replaces carriage-returns with new-lines. It echoes input characters to the terminal if requested. The backspace character erases the last character stored in the input buffer. An input control-D character is transferred to the user thread as an end-of-file indication.

5.5.2 Service read

This service transfers bytes from the serial port to the caller's buffer. The caller specifies the maximum length of the transfer. If the maximum length is greater than the

length of the next complete line stored in the input buffer, then only the line is transferred followed by a new-line character; otherwise, a partial line is transferred.

5.5.3 Service write

This service transfers bytes from the caller's buffer to the serial port. The caller specifies the number of bytes to transfer.

6 SOMBRERO PROTOCOL COMPONENT ARCHITECTURE

Subsequent to [15], Skousen distributed an enhanced implementation of the Sombrero protocol stack. This protocol stack was implemented as a passive service supporting both the Sombrero message protocol and the Sombrero data-burst protocol. These protocols are described more fully later.

This protocol stack had several major refinements. Firstly, it used the interrupt driven version of the network interface card driver described earlier. Secondly, it was more modular with the Ethernet, UDP/IP and Sombrero protocol layers separated into individual services. Thirdly, it introduced the SOSBroker, a new Sombrero service, to facilitate the dynamic assembly of a protocol stack. Fourthly, it introduced common logic to perform the thread switching required in a passive service used concurrently by multiple threads.

This researcher made further enhancements to this protocol stack. Two new components were added: an internet control message protocol (ICMP) module and an address resolution protocol (ARP) service. The former responds to echo request packets (pings) from other systems. The latter responds to ARP queries from other system and generates ARP queries when required to resolve IP addresses for outgoing packets. Because the ICMP service required an IP service, the UDP/IP service was divided into separate UDP and IP services. The resulting Sombrero protocol stack is shown graphically in Figure 5.

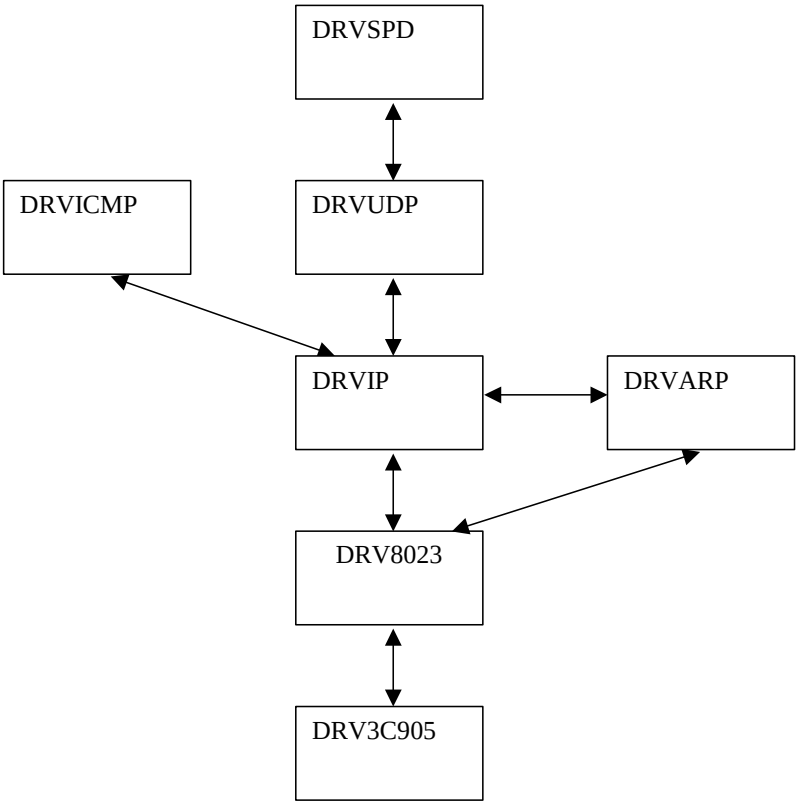


Figure 5: Sombrero Protocol Stack

6.1 Link Layer Protocol

An instance of the class DRV8023 provides the link layer protocol. The link protocol is a simple implementation of the Ethernet packet framing protocol following RFC 894.

6.1.1 Service EtherSend

This service accepts a destination Ethernet address, a frame type and a list of buffer descriptors. It allocates a new buffer and buffer descriptor. The service creates an Ethernet frame header for the packet in the buffer and inserts the buffer descriptor at the head of the buffer descriptor list. The service then calls the DRV3C905Send service passing the buffer chain.

6.1.2 Service EtherRecv

This service accepts a frame type and a list of buffer descriptors. It allocates a new buffer and buffer descriptor. The buffer is the size of an Ethernet frame header. It inserts the buffer descriptor at the head of the buffer descriptor list. The service then calls the DRV3C905Receive service passing the buffer chain. When this call returns, it compares the caller's frame type to that in the received frame. If they match, the service returns to the caller. If they do not, the service switches to another waiting thread. When a thread is found that is waiting for the frame type in the received frame, the service returns to the caller. If there is no suitable waiting thread, the frame is discarded and the DRV3C905Receive service is called again.

6.1.3 Service GetMacAddress

This service returns the local Ethernet address to the caller. This service was added to support the ARP protocol.

6.2 Address Resolution Protocol

An instance of the DRVARP class provides support for the ARP protocol. During protocol stack initialization, an ARP thread is created. This thread loops forever calling the EtherRecv service to receive ARP protocol packets. If a received ARP packet requests the Ethernet address of the Sombrero system as identified by its IP address, the ARP thread sends an ARP reply packet containing the system's Ethernet and IP addresses. If the received packet is an ARP reply to an ARP request sent by this system, the ARP thread adds the IP and Ethernet address pair to the ARP cache. If there is a user thread waiting for an ARP cache update, the ARP thread signals the user thread.

6.2.1 Service NetMACLookup

This service accepts an IP address from the caller and returns the associated Ethernet address. The service first examines the ARP cache. If the requested address is present in the cache, the service immediately returns it to the caller. If the request cannot be satisfied from the cache, this service broadcasts an ARP request for the information. The calling thread is then blocked on a semaphore. The ARP thread unblocks the user thread when an ARP reply arrives. The cache is then reexamined. If the requested Ethernet address is now present, the service returns it to the caller; otherwise, the service returns a failure status to the caller.

6.3 Internet Protocol

An instance of the DRVIP class provides internet protocol (IP) services.

6.3.1 Service IPOut

This service accepts an IP protocol number, a destination IP address and a list of buffer descriptors. The service allocates a buffer and associated buffer descriptor. In this buffer it constructs an IP packet header. It inserts the buffer descriptor at the head of the buffer descriptor list. It obtains the required Ethernet address from the NetMACLookup service. It then calls the EtherSend service to send the packet to its destination and returns to the caller.

6.3.2 Service IPIn

This service accepts an IP protocol number, an address of caller allocated storage for a source IP address and a list of buffer descriptors. The service allocates a buffer and associated buffer descriptor. This buffer is the size an IP packet header. It inserts the buffer descriptor at the head of the buffer descriptor list. The service then calls the

EtherRecv service. When this call returns, it compares the caller's IP protocol number to that in the received packet. If they match, the service returns to the caller. If they do not, the service switches to another waiting thread. When a thread is found that is waiting for the IP protocol number in the received packet, the service copies the source IP address from the IP header buffer to the caller's storage and returns to the caller. If there is no suitable waiting thread, the packet is discarded and the EtherRecv service is called again.

6.4 Internet Control Message Protocol

An instance of the DRVICMP class provides internet control message protocol (ICMP) support. During protocol stack initialization, an ICMP thread is created. This thread loops calling the IPIn to receive ICMP packets. If the packet is an ICMP echo request, the thread allocates a buffer and buffer descriptor. It formats the buffer as an ICMP echo response packet and calls the IPOut service. Otherwise, the thread discards the input packet. This class provides no callable services.

6.5 User Datagram Protocol

An instance of the DRVUDP class provides basic user datagram protocol (UDP) support.

6.5.1 Service UDPOut

This service accepts an IP address, a source UDP port number, a destination UDP port number and a list of buffer descriptors. The service allocates a buffer and associated buffer descriptor. In this buffer it constructs a UDP datagram header. It inserts the buffer descriptor at the head of the buffer descriptor list. It then calls the IPOut service to send the datagram to its destination and returns to the caller.

6.5.2 Service UDPIn

This service accepts an address of caller allocated storage for a source IP address, an address of caller allocated storage for a source UDP port number, a destination UDP port number and a list of buffer descriptors. The service allocates a buffer and associated buffer descriptor. This buffer is the size a UDP datagram header. It inserts the buffer descriptor at the head of the buffer descriptor list. The service then calls the IPIn service.

When the IPIn call returns, the service compares the caller's destination UDP port number to that in the received datagram header. If they match, the service copies the source IP address and the source UDP port number to the caller's storage and then returns to the caller. If they do not match, the service switches to another waiting thread.

When a thread is found that is waiting for the destination UDP port number in the received datagram header, the service copies the source IP address and the source UDP port number to the caller's storage and returns to the caller. If there is no suitable waiting thread, the datagram is discarded and the IPIn service is called again.

6.6 Sombrero Protocol Driver

An instance of the class DRVSPD provides Sombrero protocol (SPD) services. This is a Sombrero specific protocol. It is used between the Sombrero host and target systems. In a distributed Sombrero configuration, it is used between Sombrero target systems.

6.6.1 Service MsgToHost

This service accepts a Sombrero source address, a Sombrero destination address, a message type, a flag indicating whether or not an acknowledgement is expected and a list of buffer descriptors. The service allocates a buffer and a buffer descriptor. It

formats the buffer as an SPD header and inserts the buffer descriptor at the head of the buffer descriptor list. The service assigns a unique sequence number to every outgoing Sombrero message.

The Sombrero message is sent using the UDPOut service. The destination IP address is the configured IP address of the Sombrero host computer. The source UDP port number is the configured Sombrero UDP port number. The destination UDP port number is encoded within the Sombrero destination address.

If the caller requests an acknowledgement, the service waits for an acknowledgement message from the correspondent. If no acknowledgement is received after a short interval, the service retransmits the message. The service uses the assigned sequence number to identify the acknowledgement message.

6.6.2 Service WaitForMsgFromHost

This service accepts a Sombrero destination (local) address, a Sombrero source (remote) address, a message type and a list of buffer descriptors. The service allocates a buffer and a buffer descriptor. The buffer is the size of an SPD header. The service inserts the buffer descriptor at the head of the buffer descriptor list. The service then calls the UDPIn service to receive the message. The destination (local) UDP port is the configured Sombrero UDP port number.

When the UDPIn call returns, the service compares the caller's Sombrero addresses and message type to those in the received SPD header. If they match, the service sends an acknowledgement message to the correspondent and returns to the caller. If they do not match, the service switches to another waiting thread.

When a thread is found that is waiting for the Sombrero addresses and message type in the received SPD header, the service sends an acknowledgement message to the correspondent and returns to the caller. If there is no suitable waiting thread, the message is discarded and the UDPIn service is called again.

6.6.3 Service MsgFromHostTimer

This service is like the WaitForMsgFromHost service with the addition that the caller may specify a time-out value. If the time-out value expires, the service returns to the caller indicating that no message was available.

6.6.4 Service DataToHost

This service accepts a Sombrero source address, a Sombrero destination address and a buffer descriptor. The buffer descriptor may describe a contiguous region in memory up to 32 megabytes in length. The service allocates a fragment buffer and buffer descriptor. Using this fragment buffer, the service transfers the region by sending a series of fragments using the UDPOut service of the DRVUDP instance.

Periodically, the service sends an acknowledgement request message to the correspondent and waits for the acknowledgement with a small timeout. If the timer expires, the acknowledgement request is resent. The acknowledgement indicates using a bitmap which fragments were successfully received. The service then retransmits missed fragments and transmits additional new fragments if needed. This continues until the memory region is transferred. The service then sends the correspondent a data transfer complete message. When the correspondent acknowledges this message, the transfer is complete.

6.6.5 Service DataFromHost

This service accepts a Sombrero destination (local) address and a Sombrero source (remote) address. It allocates two buffers and two buffer descriptors. One buffer is the size of an SPD header and the other is the size of a fragment. It creates a list of buffer descriptors by creating a link from the SPD header buffer descriptor to the fragment buffer descriptor. Using the UDPIn service, the service receives 1024 byte fragments. It copies each fragment to memory at the destination addresses plus a fragment offset contained in the SPD header and marks the fragment received in its burst acknowledgement bitmap. When requested by the corresponding system, it sends this bitmap and after receiving an acknowledgement, clears the bitmap storage area in preparation for receiving another burst. When it receives the message indicating transmission complete, the service sends an acknowledgement message and returns to the caller.

This service has a potential “buffer overflow” problem. The calling interface includes only a destination address that is the base address for storing the received fragments. The caller can specify neither a maximum transfer length nor a maximum transfer address.

7 SOMBRERO PROTOCOL DESCRIPTION AND EVALUATION

As Sombrero starts execution, there are three phases that are important for understanding the assembly of the Sombrero protocol stack. These are instantiation, construction and initialization. The boot loader performs instantiation. During instantiation, an instance, IPO, of each class object, PCO, is created. This is a generic process that prepares the shareable code contained within the classes for execution.

During construction, the boot loader invokes the constructor service or entry point for each instance. The constructor entry point is simply an entry point with a well-known name: <class_name>_CONSTRUCTOR. It is the responsibility of this entry point to perform internal initialization of the instance. At this time, the instance prepares itself to provide the services of its public interfaces. The constructor entry point may only call a limited number of system services that were constructed early. The boot loader contains a list of the classes that must be constructed early and the order in which it may safely invoke their constructor entry points.

During initialization, hardware and protocol drivers dynamically locate each other and invoke one another's initialization services. The initialization service is again an entry point with a well-known name: Initialize. The initialization phase starts when the boot loader transfers control to the Sombrero debugger, SOSDebug. SOS debug invokes the Initialize entry point of the instance of DRVSPD class prior to entering it main message processing loop.

7.1 Protocol Stack Assembly

The binding together of the Sombrero protocol stack happens in two steps. During construction, each instance calls the Register service of the SOSBroker instance passing its object identifier (OID) and assigned driver number. The OID is the instance's

this pointer. Appendix A contains the current driver assignments. At the end of the construction phase, SOSBroker has a database containing driver numbers and associated OIDs for all drivers present in the system.

The initialization phase starts when SOSDebug invokes the Initialize service of the DRVSPD instance. This service calls the FindBest service of the SOSBroker instance passing the driver number of a protocol on which it is dependent. In this case it is DRVUDP. FindBest returns the OID of DRVUDP. Using this OID, the services of the DRVSPD instance can invoke the services of the DRVUDP instance. To continue the initialization process, it now invokes the Initialize service of the DRVUDP instance that in turn resolves its dependencies. This process of instances invoking the Initialize services of dependent instances repeats until all drivers have satisfied all of their dependencies. When it concludes, the Sombrero protocol stack is assembled and ready to process service requests.

7.2 Message Protocol Description

The Sombrero message protocol is a simple stop-and-wait protocol. It is capable of transparently transferring messages up to 1024 octets in length. The protocol assigns a unique sequence number to each message. When a message is sent, the protocol starts a short acknowledgement timer. If no acknowledgement with the matching sequence number arrives before the timer expires, the protocol resends the message.

7.3 Burst Protocol Description

The Sombrero burst protocol is a retransmit-n protocol. It is capable of transparently transferring large virtual address regions. The Sombrero implementation limits this to contiguous regions with a maximum size of 32 megabytes, but these

limitations are not a restriction of the protocol. The protocol assigns a unique sequence number to each region transfer. The protocol sends the region in bursts up to 8 megabytes in length. Each burst consists of up to 8192 fragments. Each fragment contains 1024 bytes. The fragment message header contains the transfer sequence number to identify the transfer to which the fragment belongs, the fragment number within the burst, the memory offset from the destination address at which the fragment is to be stored and the fragment length in bytes.

When all the fragments in a burst are transmitted, the sender transmits an acknowledgement request message to the receiver and starts an acknowledgement timer. The protocol assigns a unique sequence number to the acknowledgement request message. This message contains transfer sequence number. If the timer expires, the sender retransmits the acknowledgement request message. The receiver replies with an acknowledgement bitmap. The bits in the bitmap correspond to the fragment numbers within a burst. A set bit in the bitmap indicates that the fragment was successfully received. Using the bitmap, the sender then transmits another burst that may include any required retransmissions from the prior burst and new fragments up to the 8192- fragment limit.

When the region transfer is complete, the sender transmits a transmission complete message to the receiver and starts an acknowledgement timer. If the timer expires, the message is retransmitted. The receiver replies with an acknowledgement concluding the transfer.

7.4 Evaluation

Sombrero is an unconventional research operating system currently in an early prototype stage. As such it will invariably have architectural, logical design, implementation mechanism and bug problems. This section assumes the reader is familiar from earlier sections and [15] with the advantages gained from the Sombrero design's usage of single address space properties including such things as reduced code complexity, low-level object oriented design, etc. These are not repeated here nor are speculations on improvements via changes to design or implementation mechanisms, often called "future work". This section simply presents this researcher's evaluation of the current unfinished state of the Sombrero protocol stack at the conclusion of this research work, that is, the "Sombrero 39" release available at [19].

7.4.1 Performance

There is a test application for the Sombrero message protocol. The application creates up to 256 threads that will each send to the Sombrero host up to 256 messages requiring acknowledgement. A terminal emulation program on the Sombrero host captures debug information from the protocol layers through the serial port. The Ethereal protocol analyzer captures the message traffic. The goal is to have this application run reliably at maximum load. This goal has not been achieved. Below 8 threads, the application always completes. Above about 32 active threads, the application fails to complete. The protocol stack reaches an equilibrium condition in which messages are exchanged but no progress is made. Sometimes pinging the Sombrero system will perturb the equilibrium and allow the application to progress.

7.4.2 Thread Switching

In the description of the various layers of the Sombrero protocol, there is often a reference to switching between threads in order to find a waiting thread to accept the current message. The need to do this switching is a consequence of the passive service implementation. There are multiple threads wanting to receive a message. One of the threads becomes the receiving thread. As the thread returns up the receive side of the protocol stack (and call stack), each protocol layer checks if the message meets the thread's requirements. This meeting of requirements is no more than matching calling parameter values against values in the received message's protocol header for the current layer. A protocol layer's input service can only interpret the contents of the buffer that it inserted at the head of the buffer descriptor list prior to calling the lower layer. It has no access to the headers of lower layer protocols and no knowledge of the structure of buffers intended for higher layers.

If the message meets its requirements, the thread returns to the next higher layer. If at some layer the message no longer meets its requirements, the thread cannot simply switch to a new message. The current message cannot be discarded because the wrong thread happened to service the receive interrupt. The layer executing under the current thread must facilitate the switch to a new thread. This is possible if there are other threads waiting at this layer. All other threads waiting at this layer have the same message requirements for layers below the current layer. A new thread can just begin matching at this layer. Since thread switching must be done at every layer, the logic to perform the thread switching has become fairly standard. This logic is shown in

Appendix B. Observe that only one thread descends to the next lower layer. All other threads using a layer's service are blocked.

The concepts of interrupts as synchronization objects and passive services promised greater efficiency through reduced context switches. The outbound side of the Sombrero protocol stack achieved this goal. The inbound side of the Sombrero protocol stack achieved this goal only for the important case where a single thread traverses the protocol stack and reduces context switching to zero.

7.4.3 Thread Interaction

All threads calling the Sombrero protocol stack to receive a message must allocate a message buffer sufficient for the largest possible receive message. A message is always received initially into some thread's buffer descriptor list. The message may later move through a protocol layer's common buffer to another thread's buffer descriptor list. This can result in a truncated message. An error by one thread should not be able to negatively impact the execution of another thread.

A thread receiving a short message may have a long message copied at some layer into its buffer descriptor list. The thread may access this data. This is an unacceptable security violation.

7.4.4 Data Movement

One technique for improving the performance of a protocol stack is to minimize data movement. A single address space operating system (SASOS) seems to offer much potential in this area since all threads reside in the same address space. The current protocol stack implementation is deficient by this measure. Messages are copied from one thread's calling stack to another through common layer buffers. Moreover as the

number of threads increases, the data movement problem worsens. This is a consequence of the standard thread switching logic shown in Appendix B. When at some layer the receiving thread examines the message and determines that the message does not belong to it, the receiving thread copies the message from its buffer chain to that layer's common buffer and relinquishes control. Subsequently, as they gain control, each other thread waiting at that layer must first copy the common buffer to its buffer chain in order to examine the message. Thus, the data movement increases as the number of waiting threads increases.

7.4.5 Transmit to Receive Transition

After it sends a message, a thread often needs to receive an acknowledgement or other response. The protocol stack sometimes fails to perform this transition reliably. The addition of ARP and ICMP protocols to the protocol stack made this behavior clearly visible. These protocols introduced thread switching at the Ethernet and IP protocol layers.

There are two threads present at the Ethernet layer: the ARP thread and the thread holding the IP layer lock. Of these two, one holds the lock and the other waits. ARP frames are rare, but half of the time the ARP thread receives the frame and there must be a thread switch. There are two threads active at the IP layer: the ICMP thread and the thread holding the UDP layer lock. Of these two, one holds the lock and the other waits. ICMP packets are rare but half of the time the ICMP thread receives the packet and there must be a thread switch. Thus, there is an average of one context switch per message received just considering the IP layer and below. There is currently no thread switching at the UDP layer because only the SPD layer uses its services.

Trace information collected using the serial port sometimes shows the following scenario. The user thread acknowledgement timer expires while the thread is waiting for the IP layer lock. Meanwhile, the ARP thread repeatedly releases and acquires the Ethernet layer lock discarding in the process all non-ARP frames including the message desired by the thread blocked at the higher layer. There are also similar cases where the host system just responds too quickly and the protocol stack rightly discards the message because there is no thread waiting for it. In both cases the Ethereal protocol analyzer confirms that no messages are lost in transmission. The protocol stack requires a more robust and scalable method of matching messages with threads.

8 STANDARD INPUT/OUTPUT LIBRARY

To facilitate use of the serial port driver, this researcher developed a limited standard input-output library. In [15 p49], Skousen describes the development of Sombrero library classes.

8.1 Service gets

This service uses the read service of the DRVUART instance to read a new-line terminated string from the first serial port into the caller's buffer.

8.2 Service puts

This service uses the write service of the DRVUART instance to write a string from the caller's buffer to the first serial port.

8.3 Service printf

This service uses the write service of the DRVUART instance to write a formatted string to the first serial port. This service uses an open source version of vsnprintf.

8.4 Service scanf

This service uses the read service of the DRVUART instance to read a formatted string from the first serial port into the caller's buffer. This service uses an open source version of vsnscanf.

8.5 Service sprintf

This service generates a formatted string in the caller's buffer from a format specification string and a list of variables. This service uses an open source version of vsnprintf.

8.6 Service sscanf

This service decomposes a formatted string into a list of variables using a format specification string. This service uses an open source version of vsnscanf.

9 CONTRIBUTIONS AND CONCLUSIONS

This research provided Sombrero and its future researchers with an improved development environment. This environment facilitates the development and management of Sombrero system modules.

This research provided documentation of previously undocumented features of the Sombrero design and implementation. This documentation should be helpful to future investigators.

This research provided Sombrero with an implementation of a lower level architecture to complement the middle level architecture developed by Skousen. The lower level architecture exploited the Sombrero object oriented software organization. This implementation effort provided an investigation of implementation implications of the Sombrero middle level architecture.

This research demonstrated that Sombrero's unique interrupt processing architecture could be used successfully to build drivers for devices commonly found in computer systems. The techniques used in the serial port driver could be easily extended to a mouse or keyboard device.

The Sombrero protocol stack implementation is not yet robust enough to support planned developments for Sombrero. These developments include support for network-based storage and a web server. Both will require dependable network services. The stacking of passive services creates both context switching and data movement problems. The architecture of the protocol stack may need to be reconsidered.

10 REMAINING WORK

The development of functionality within the Sombrero boot loader and PALCode remains too difficult. A contributing factor is the transfer of the binary files from the Sombrero host system to the Sombrero target system disk. The development of a preliminary loader to transfer the PALCode and boot loader, configure and boot the target system would reduce development cycle time in this area. This preliminary loader could use standard protocols such as Dynamic Host Configuration Protocol [DHCP] and Trivial File Transfer Protocol [TFTP].

It is difficult to study the execution order and timing of events within the Sombrero system. This makes difficult to understand and correct anomalous behavior and evaluate the performance of alternate implementations of system features. The development of a system trace facility accessible to both PALCode and system modules would permit the quantitative study of system interactions during interrupt servicing and thread scheduling.

Since all Sombrero components are now compiled with gcc, the development environment could be further simplified through the use of cross compilation. This would permit the replacement of the Alpha Linux development system with an industry standard architecture system. Additionally, the host-based Sombrero components could be ported from Windows to Linux further simplifying the development environment.

Sombrero requires a storage mechanism to support the implementation of persistent memory. The implementation of a client for ATA-over-Ethernet [Advanced Technology Attachment, AoE] protocol [2] could provide Sombrero with connectivity to a large storage area network [SAN].

The Sombrero middle level architecture still requires implementation of persistent memory and distributed operation.

REFERENCES

[1] 3Com Corporation. *3C90x and 3C90xB NICs.* Technical Reference Manual, 89-0766-000, 1998.

[2] B. Coile and S. Hopkins. The ATA over Ethernet Protocol. http://www.coraid.com/documents/AoEDescription.pdf (30 July 2005).

[3] B. Collins-Sussman, B. Fitzpatrick and C. Pilato. *Version Control with Subversion*. O'Reilly, 2004.

[4] Compaq Computer Corporation. *Alpha PC 164SX Motherboard.* Technical Reference Manual, EC-R57EB-TE, 1998.

[5] Cypress Semiconductor Corporation. *hyperCache / Stand-Alone PCI Peripheral Controller with USB.* Technical Manual (preliminary), CY82C693U, 1997.

[6] Digital Equipment Corporation. *The Alpha Evaluation Board Software Developers Kit*. Release 2.1 Beta 5, August 1996.

[7] Digital Equipment Corporation. *Digital Semiconductor Alpha 21164PC Microprocessor*. Hardware Reference Manual, EC-R2W0A-TE, September 1997.

[8] Digital Equipment Corporation. *Digital Semiconductor 21174 Core Logic Chip.* Technical Reference Manual (preliminary), EC-R12GC-TE, 1997.

[9] *FreeBSD Architecture Handbook*. http://www.freebsd.org/doc/en_US.ISO8859-1/books/arch-handbook/index.html (30 July 2005).

[10] GNU Compiler Collection. GNU Project, Free Software Foundation. http://www.gnu.org/directory/Software_development/Compilers/gcc.html (6 July 2005).

[11] S. Kleiman and J. Eykholt. Interrupts as threads. *Operating Systems Review* 29(2):21-26, Apr. 1995.

[12] S. Kung and L. Onken. *TortoiseSVN: a Subversion Client for Windows*. http://tortoisesvn.tigris.org/ (6 July 2005).

[13] D. Miller. The Sombrero Project. http://www.eas.asu.edu/~sasos/sombrero.htm (13 July 2005).

[14] MIPS Technology Inc. *Advanced RISC Computing Specification.* Version 1.2. 1992.

[15] A.C. Skousen. *Sombrero: Implementation of a Single Address Space Paradigm for Distributed Computing Exhibiting Reduced Complexity*. PhD Thesis, Arizona State University, 2002.

[16] A.C. Skousen. *Sombrero : A Very Large Single Address Space Distributed Operating System*. MS Thesis, Arizona State University, 1994.

[17] A. Skousen and D. Miller. Operating System Structure and Processor Architecture for a Large Distributed Single Address Space. In *Proceedings of PDCS'98: 10th International Conference on Parallel and Distributed Computing Systems*, October 1998.

[18] Standard Microsystems Corporation. *PC 95 Compatible Super I/O Floppy Disk Controller with Infrared Support*. Technical Reference Manual, FDC37C669, 1996.

[19] D. White. Sombrero Repository. https://sombrero.kicks-ass.org/sos/ (14 July 2005).

APPENDIX A

SOMBRERO DRIVER NUMBERING SCHEMA

Hardware and Software Drivers use prefix DRV for DriVer. Drivers are classified by services rendered and registered with the SOSBroker. The following is the schema for the SOSBroker device driver numbering. To get a classification string, just follow a path from the major classification to the leaf item of interest. For example the IP driver is 2.1.1.3

Major classifications are:

1 Storage Devices
- 1 Block
- 2 Stream

2 Communications Devices
- 1 Protocol Drivers
  - 1 Abstract protocols
    - 1 DRVSPD - Sombrero Protocol Driver
    - 2 DRVUDP - UDP Driver
    - 3 DRVIP - IP Driver
    - 4 ICMP - Internet Control Message Protocol
    - 5 ARP - Address Resolution Protocol
  - 2 Base protocols
    - 1 DRVE8023 - Ethernet 802.3 Protocol Driver
- 2 Hardware Drivers
  - 1 DRV3C905 - 3C905 Hardware Driver
  - 2 DRVRTC - Real Time Clock Driver
  - 3 DRVUART - UART Driver
  - 4 DRVTIMER - Periodic Timer Driver

3 Output Devices
- 1 Raster
- 2 Stream

4 Input Devices
- 1 Byte stream

5 Device Managers
- 1 Bus devices
  - 1 unused - available
  - 2 DRVPCIBUS - PCI Bus Driver
  - 3 DRVISABUS - ISA Bus Driver

APPENDIX B

SOMBRERO PROTOCOL THREAD SWITCHING LOGIC

The following is the general form of the logic implemented on the receive side in each of the layers of the Sombrero protocol stack. Each layer has a lock and a common buffer. Each layer applies layer specific criteria to select the receiving thread.

```
input_service(buffer_descriptor_list)

    allocate this layer's protocol header buffer
    allocate buffer descriptor
    insert this layer's buffer descriptor
        at head of caller's buffer descriptor list

    acquire the protocol layer lock

    forever
        if this layer's buffer is owned
            copy the message in the buffer
                into the caller's buffer descriptor list
        else
            call the lower layer's input service
                to get a message in the caller's
                buffer descriptor list
        end if

        if the message meets the caller's requirements
            break out of the loop
        end if

        if the stored message buffer is unowned
            copy the caller's buffer descriptor list
                into this layer's buffer
            mark the caller as the owner of the
                stored message in this layer's buffer
            release the protocol layer lock

            <other threads may now run>

            acquire the protocol layer lock
            if this thread is still the owner of the buffer
                mark the buffer as unowned
            end if
        end if
    end forever

    mark this layer's buffer as unowned
    release the protocol layer lock

    return to caller
```

APPENDIX C
SOMBRERO DEVELOPMENT ENVIRONMENT

Sombrero Technical Note
Sombrero Development Environment

1.0 Introduction

This document describes the setup of the Sombrero development environment. This environment includes a W2K or XP Sombrero host system and an Alpha Linux system to function as a Sombrero compile engine. I have not confirmed every step as my Sombrero compile engine has evolved over time and I did not assemble a new one just to test these instructions.

In the following I will refer to the Windows systems on which the Sombrero source code resides as the host system and the Alpha Linux system on which the Sombrero code is complied as the target system.

2.0 Alpha Compile Engine

2.1 rshd Configuration

The `rshd` service must be enabled so that the Sombrero host can execute programs on the target compile engine. The disable directive in the file `/etc/xinetd.d/rsh` controls this service. The contents of the file should be as follows:

```
# default: on
# description: The rshd server is the server for the
# rcmd(3) routine and,consequently, for the rsh(1)
# program.  The server provides remote execution facilities
# with authentication based on privileged port numbers from
# trusted hosts.
service shell
{
      disable           = no
      socket_type       = stream
      wait              = no
      user              = root
      log_on_success    += USERID
      log_on_failure    += USERID
      server            = /usr/sbin/in.rshd
}
```

After changing the file, one must restart the `xinetd` daemon to have the change take effect. The following script, executed as root, restarts the `xinetd` daemon.

```
#/etc/init.d/xinetd restart
```

Create an account on the system besides root. I will refer to this account as `<target_user_name>` in the following. Create the file `/home/<target_user_name>/.rhosts` with the following contents:.

```
<host_system_name>      <host_user_name>
```

The field `<host_system_name>` is the name of the Sombrero host system. The field `<host_user_name>` is the Windows login name on the Sombrero host that is used when doing Sombrero development. The permissions on this file must be 644.

To enable the compile engine to resolve the `<host_system_name>` add the following line to the end of the `/etc/hosts` file:

```
<host_ip_address> <host_system_name>
```

At this point one can test that the target system is correctly configured. Execute the following command at the command prompt in Windows:

```
>rsh <target_system_name> “ls”
```

It may be necessary to add the `<target_ip_address>` and `<target_system_name>` to the `c:\winnt\system32\drivers\etc\hosts` file.

2.2 Samba Configuration

The next task is to create a Samba share within the `<target_user_name>` home directory. First, create the directory `/home/<target_user_name>/Sombrero`. This is the directory that will be shared. Second, one must configure the target system to run Samba and, third, one must configure Samba to share the directory.

The following steps configure the system. In the file `/etc/xinetd.d/swat`, change "`localhost`" to "`127.0.0.1`" and "`disable = yes`" to "`disable = no`". Alternately, one can just comment out the offending lines. This is to allow the `xinetd` daemon to start the SWAT web-based administration tool for Samba. Restart the `xinetd` daemon as described above.

By default the Samba init script `/etc/rc.d/init.d/smb` is off for all run levels. I ran

```
#chkconfig --level 5 smb on
```

so that `init` would start the two Samba daemons in runlevel 5. The script `/etc/init.d/smb` takes the command line arguments `start`, `stop`, `restart` and `status` which is handy until the `/etc/samba/smb.conf` is sorted out.

The desktop menu item programs->system->Samba Configuration starts the SWAT Samba configuration tool. Add a workgroup name and server name. The workgroup name must match the configuration on the Sombrero host system. The server name is the name of the compile engine system. Be sure to add the server name to `/etc/hosts` so that Samba knows what interface with which to bind.

Define shares and users. Just be sure the share paths exist. One can map NetBIOS users to Unix users in `/etc/samba/smbusers`, but I don't think I had to do this.

I am unsure of the SWAT commands used to configure my compile engine. The result was the following share definition in the `/etc/samba/smb.conf` file.

```
# workgroup = NT-Domain-Name or Workgroup-Name
workgroup = <your_workgroup_name>

# server string is the equivalent of the NT Description field
server string = Samba Server

[Sombrero]
path = /home/<target_user_name>/Sombrero
writeable = yes
force user = <target_user_name>
guest ok = yes
```

There are other entries that I found useful:

```
encrypt passwords = yes
local master = no

[<target_user_name]
path = /home/<target_user_name>
writeable = yes
force user = <target_user_name>
guest ok = yes
```

I think "`encrypt passwords = yes`" was required because W2K sends encrypted passwords to Samba. Setting the "`local master = no`" prevents the compile engine from gaining control of my local network and breaking printer sharing. Making a share for the `<target_user_name>` home directory makes it easier to copy the Sombrero compile scripts to the target system.

It is necessary to create a Samba user and assign a password. I used the name <host_user_name> and assigned a password which was the same as the password on the host system. It is convenient if the values for the Sombrero host and target machines match, but this is not required. If not, one will need to explicitly provide the Samba user name and password when mapping the Samba share.

```
#smbpasswd –a <host_user_name> <password>
```

Start or restart the Samba daemons with the /etc/initd.d/smb script and test from windows.

```
#/etc/initd.d/smb restart
```

At this point the host system should be able to see the share from the target system. It should be visible in the network neighborhood. One can test the correctness of the configuration by mapping the share to a drive letter and verifying that one can create, write, read and delete files on the mapped drive.

2.3 Target System Sombrero Compiler Setup

On the host system, the directory c:\Backup\Linux\Sombrero Compile contains three subdirectories: buildsxe, catdebug and sosbuild. Copy these three directories and their contents to the <target_user_name> home directory on the target system. The home directory share created above facilitates this.

On the target system, create the directory /home/<target_user_name>/bin. Within this directory, create three symbolic links.

```
$cd /home/<target_user_name>/bin
$ln –s /home/<target_user_name>/buildsxe/buildsxe/buildsxe buildsxe
$ln -s /home/<target_user_name>/catdebug/catdebug/catdebug catdebug
$ln -s /home/<target_user_name>/sosbuild/sosbuild/sosbuild sosbuild
```

It may be necessary to insure that the link targets are executable.

3.0 Host System Sombrero Setup

Several things to do:

Install VC++ 6.0.

Copy Sombrero source code into the directory c:\Backup. The source code is in the Sombrero repository.

Update `c:\Backup\Sombrero\os\rbuild.cfg` and `rbuildboot.cfg` to include `<target_user_name>` and `<target_system_name>`.

Within VC++, from the Tools menu select “Options…” to bring up the Options dialog box. Select the Directories tab. For the Win32 platform and the include directories, click on the New button and then on the Browse button. Browse to the `c:\Backup\Sombrero\OS\SOSInclude` directory and click the OK button. Click the New button again and then click on the Browse button. Browse to the `c:\Program Files\Microsoft Visual Studio\VC98\MFC\SRC` directory and click the OK button. Click the OK button to close the Options dialog box.

Copy the file `c:\Backup\Sombrero\os\lib\SOSHostDLL.dll` to `c:\WINNT\system32`.

Copy the file `c:\Backup\Sombrero\os\SOSRBuild\debug\SOSRBuild.exe` to `c:\WINNT\system32`.

4.0 Sombrero System Setup

The Sombrero system is a DEC 164SX mainboard with AlphaBIOS installed. Sombrero supports only a 3Com 3C905TX Ethernet adapter.

4.1 AlphaBIOS Setup

To setup the Sombrero boot option in AlphaBIOS, power on the Sombrero Alpha system and press F2 during AlphaBIOS initialization to enter setup.

Select “Utilities” from the main menu.

Select “OS Selection Setup” from the Utilities menu.

Press the insert key to create a new selection.

Enter a boot name such as “Sombrero”.
For the boot file, select the disk and partition where the Sombrero bootloader and PALcode are stored. On my system, this is disk 0, partition 1. Enter the path to the Sombrero bootloader. On my system this is `\arc\sombrero.exe`.

The OS path is not used. It can be set to CD with a path of `\arc\`.

The OS options are “`<Sombrero-system-IP-address> <network-mask> <Sombrero-host-IP-address> 1010`”. These items are specified in dotted-decimal notation.

Press F8 to make this selection primary if desired.

Press F10 to save changes to the NVRAM.

The AlphaBIOS can automatically boot the primary selection. From the setup main menu, select “CMOS Setup”. Next, enable the “Auto Start” option and set the “Auto Start Count” if desired.

4.2 Bootloader Setup

Boot the Sombrero system using Windows NT. NT will assign a drive letter, `<d:>`, to the small boot partition. The drive letter depends on how the drive was partitioned and to which partition NT was installed.

In the small FAT boot partition, create a directory `<d:>\arc`.

Copy `arcpal.pal` from the Sombrero host system or compile engine system to `<d:>\arc\arcpal.pal`.

Copy `sombrero.exe` from the Sombrero host system or compile engine system to `<d:>\arc\sombrero.exe`.

APPENDIX D
BOOTLOADER DEBUGGER

Sombrero Technical Note
Boot Loader Debugger.

Introduction

The Sombrero boot-loader debugger is a simple debugger that supports debugging of the construction of the Sombrero operating system objects. Once all objects are constructed, the bootloader debugger transfers control to the Sombrero system debugger, SOSDebug, which supports debugging of Sombrero services. The two debuggers are technically similar and make use of the same PALcode services.

The Sombrero boot-loader debugger consists of the function sombrero.c::debugger, the assembly language functions StoreInitialContext(), CallBreakPointEntry(), and RestoreBreakPointContext() in the module jtopal.S and the Sombrero PAL code services StoreBreakPointVextors (0x81), PerformBreakPointContextSwitch (0x82), and ResumeDebugExecution (0x83).

The high-level control flow is as follows. The boot-loader debugger requests that the Sombrero host start the host debugger GUI application. The host debugger application sends a message to the boot-loader debugger so that the debugger knows that the host debugger application is executing. The boot-loader debugger then creates two execution contexts: one for itself and one for code to be debugged. These two contexts are named respectively, the debugger context and the breakpoint context.

The debugger then enters a loop receiving messages containing commands from the host and executing them. These commands include reading and setting registers, reading memory areas, setting and clearing breakpoints and transferring control to functions. This loop continues until the either control is transferred to SOSDebug or the host GUI application terminates.

In general, the host debugger first commands the boot-loader debugger to setup the breakpoint context and then commands the debugger to transfer control to the breakpoint context. When the code encounters a breakpoint the debugger sends to breakpoint notification to the host debugger. Control returns to the top of the debugger message loop and the debugger waits for further commands.

The following examines in detail the content of the context areas and the support functions.

Execution Context

Each execution context is an array of 128 64-bit unsigned integers. The locations 0 to 31 store the 32 integer registers. The locations 32 to 63 store the 32 floating point registers. The location 64 stores the PC and the location 65 stores the FLAGS. The location 127

stores the count of entries in the breakpoint list. The breakpoint list starts at location 125 and expands downward toward location 66. Each breakpoint entry consists of two 64-bit words, the first is the displaced instruction at context[2n] and the second is the address of the instruction at context[2n + 1]. The two arrays are contiguous in memory.

On the Alpha, some registers have conventional uses. The return address (ra) register is $26. The stack pointer (sp) is $30. The first argument register (a0) is $16. When calling functions within the debugger, the Alpha gcc compiler calling convention requires the caller to set $27 to the address of the called procedure.

StoreInitialContext

This function initializes the debugger and the breakpoint context areas. The debugger calls this function after starting the Host debug application, but before entering the command-processing loop. In function, this function is similar to the Unix setjmp() function, in that it establishes a context at which execution may repeatedly resume. Like setjmp(), StoreInitialContext() is called once but returns many times. If StoreInitialContext() set $0, the return value register, to 0 prior to returning, then it would behave like setjmp() when returning directly. Instead, the boot-loader debugger uses the volatile automatic variable `init` to distinguish between the first and subsequent returns from StoreInitialContext(). The volatile attribute forces the compiler to allocate this variable on the stack instead of a register and thus preserve its value independently of the register context restoration.

This function stores the current execution environment into both context areas. In the breakpoint area, it sets the program counter and the stack pointer to 0, indicating that the host debugger must initialize these registers. It sets the return address register to the address of a call_pal 0x82 instruction. Thus, a program executing in the breakpoint context must at least generate a breakpoint when it exits.

In the debugger context, the function sets the program counter to the address of the CallBreakPointEntry() function. It executes a call_pal 0x81 to store the physical and virtual addresses of the context save area so that these addresses are available to the PALcode. The return address remains the value on entry, that is, the instruction past the initial call to StoreInitialContext(). This is where the debugger always resumes execution.

CallBreakPointEntry

This function stores the current execution context in the breakpoint context area. It restores the execution context from the debugger context area. It sets the program counter to the value of the return address register which is the value at the call to StoreInitialContext(). Execution continues then as if StoreInitialContext() just completed. If CallBreakPointEntry() set $0, the return value register, to a non-zero value

prior to returning, then StoreInitialContext() would behave like setjmp() retuning from a longjmp(). As noted above, the variable `init` is used to distinguish these two cases. In function, this function is similar to the Unix function longjmp().

RestoreBreakPointContext

The debugger calls this function in its context to execute a procedure in the breakpoint context. Before calling this function, the debugger must set the program counter and stack pointer in the breakpoint context. Using the breakpoint list in the breakpoint context, it sets any specified breakpoints in the code. This function then restores the execution environment from the breakpoint context and executes a call_pal 0x83.

StoreBreakPointVectors, call_pal 0x81

The service stores the physical and virtual addresses of the context storage area in PAL storage.

PerformBreakPointContextSwitch, call_pal 0x82

Executing code invokes this PAL code service when it encounters a breakpoint. This service retrieves the physical address of the context save area and stores the program counter and flags of the breakpoint in the breakpoint context. It sets the a0 register, $16, to the virtual address of the context save area. It sets the program counter to the saved value in the debugger context which is the address of CallBreakPointEntry() and resumes execution.

ResumeDebugExecution, call_pal 0x83

This PAL code service restores the program counter and flags from the breakpoint context. The procedure being debugged resumes execution.

Pseudocode

```
debugger()
  call StoreInitialContext(context)
  if return is from a breakpoint
    if <Alan what is the meaning of this condition???>
      call RestoreBreakPointContext(context)
    send breakpoint notification to the host debugger
  while(host is active)
    receive command from host debugger
    if command is SEND_BOOTLIST
      send the host a copy of the bootlist and datalist
    else if command is CONSTRUCTOR
      set breakpoint context PC to the Constructors function
      set breakpoint context $27 to the Constructors function
      call RestoreBreakPointContext(context)
    else if command is CONTROL_TRANSFER
      in the breakpoint context use the breakpoint list
        to set any breakpoint in the code
      call RestoreBreakPointContext(context)
    else if command is SEND_REGISTERS
      send the breakpoint context area to the host
    else if command is SEND_STRING
      send the requested string to the host
    else if command is SEND_MEMORY_WINDOW
      send the requested 1KB memory region to the host
    else if the command is RESTORE_REGISTERS
      copy the supplied values to the breakpoint context
    else if the command is EXIT_DEBUGGER
      break out the command processing loop
  return to AlphaBIOS
```

APPENDIX E
INTERRUPT THREADS

Sombrero Technical Note
Interrupt Threads

1.0 Introduction

Unlike many commonly available operating systems, Sombrero services interrupts within a thread context. Sombrero treats interrupts similar to semaphores. A thread waits for an interrupt to occur similar to pending on a semaphore. When the interrupt occurs, the system unblocks the thread similar to signaling a semaphore.

There are four steps involved in Sombrero's interrupt service. Firstly, the appropriate data structures are created to allow the PALCode to identify an appropriate thread. Secondly, at runtime, a thread must indicate to the PALCode that it is waiting for the interrupt. Thirdly, at the occurrence of the interrupt, the PALCode must map the interrupt to the waiting thread, perform the context switch and dispatch the waiting thread. Lastly, the interrupt thread must indicate to the PALCode either that it has completed servicing the interrupt or that it did not service the interrupt as in the case for shared interrupts.

2.0 Data Structures and Initialization

2.1 Data Structures

SOSInterrupt::SOSInterrupt_CONSTRUCTOR allocates the space for and initializes a vector for managing interrupts. The vector contains thirty-two entries: one for each Interrupt Priority Level, 0 to 31. Each vector entry is the following structure:

```
struct intvecpair {
    UINT64            pvec;
    struct intvector* vvec;
    UINT64            prophylactic;
    UINT64            unused;
}
```

SOSInterrupt_CONSTRUCTOR stores the virtual address of this vector in an external variable, `VectorTable`. After initialization, SOSInterrupt_CONSTRUCTOR passes the physical address of the vector to the PALCode using a call to the function intbase() in the file Interrupt.S. This function contains an assembly language call to the PALCode entry 0xa0. This entry point stores the caller specified value in the INTERRUPTBASEVECTOR Sombrero static register.

The prophylactic field is an OR of all the request bits in the 21174 IRR for the entries in the list chained to this vector entry. The PALCode uses this field to acknowledge interrupts for which there is no thread waiting.

The pvec and vvec fields are an anchor to a list, referred to as a chain, of struct intvector data items. The field pvec is a physical address for use by the PALCode and the field vvec is a virtual address for use by SOSInterrupt. Each chain entry is the following structure:

```
struct intvector {
    UINT64 cyclethread;
    UINT64 unuseddb;
    UINT64 GPD;
    UINT64 request;
    UINT64 hqlink;
    UINT64 tqlink;
    struct intvector *vchain;
    UINT64* pchain;
}
```

SOSInterrupt_CONSTRUCTOR allocates one struct intvector for each struct intvecpair. This initial allocated structure serves as an end of chain entry and has the GPD field set to zero.

The cyclethread field is the physical name of the thread that began the current queue search. This field is initialized to zero. The PALCode uses this field.

The GPD field is the general protection domain allowed to schedule on this (vector, request) pair. This is not a configuration item. The value of this field is taken from the first thread to register for this pair. This field is initialized to zero and indicates that this entry is the last in the chain. The board object should perhaps validate the (vector, request) pair for the hardware platform.

The request field is the interrupt request number for this (vector, request) pair. This field is initialized to zero. In use this field takes a value from 0 to 63. The value is the number of the bit of the corresponding interrupt in the 21174 Interrupt Request Register.

The hqlink field is a physical address of the head of the threads, TCBs, queued on this (vector, request) pair. This field is initialized to zero. The PALCode uses this field.

The tqlink field is a physical address of the tail of the threads, TCBs, queued on this (vector, request) pair. This field is initialized to zero. The PALCode uses this field.

The vchain field is the virtual address of the next struct intvector at this interrupt priority level. This field is initialized to the virtual address of the initial entry. This makes the list circular.

The pchain field is the physical address of the next struct intvector at this interrupt priority level. This field is initialized to the virtual address of the initial entry. This makes the list circular.

2.2 General Protection Domain Registration

A thread calls SOSInterrupt::AddInterruptGPD to register to process an interrupt. This function takes two arguments. The first specifies the interrupt vector as described in the previous section. The vector is the interrupt priority level, 0 to 31, at which the interrupt will occur. The second specifies the interrupt request number, 0 to 63. This is the number of the corresponding bit in the 21174 IRR. This registration allows any thread from the same general protection domain to service the (vector, request) pair. The caller insures that the hardware actually generates the specified interrupt at the specified priority level.

The function systemintmask() is in Interupt.S. The function contains an assembly language call the PALCode entry point 0xb0 which is described in a following section.

The current logic as described below does not reflect the intent of Sombrero. General Protection Domains should be permitted to share a vector and request as two peripherals may share a vector and request.

Description of Logic

```
AddInterruptGPD(vector, request)
  if the specified vector is valid
    while there are entries in the vector's list of struct intvector
      if the current entry is for the same level and interrupt
        if it is for the same GPD
          this is a duplicate registration, allow it
          return TRUE
        else
          only one GPD for each (vector, request), disallow it
          return FALSE
    call systemintmask() to set the new system IPL
    if the previous IPL is lower than the new IPL
      call systemintmask() to restore the previous IPL
    add the request to the prophylactic field
    allocate a new struct intvector
    link new entry before tail entry
    call the 21174 service to enable the hardware interrupt
    return TRUE
  else
    return FALSE
```

3.0 The PALCode

There are several sections of PALCode relevant to interrupt processing. These sections perform four functions. The first, intbase and systemintmask, support initialization. The second, blkintth, allows a thread to queue waiting for an interrupt. The third, unblkintth, supports the system timer thread in providing a timeout for threads queued for interrupts.

The fourth, Interruptcheck, services interrupts. For each I will provide a description of the logic using pseudo-code.

The PALcode is written in assembly language. It does not exhibit the block structure of a higher-level language. The following describes the logic and, to a lesser extent, the structure of the PALcode. It does not describe the implementation.

3.1 Common Routines

To start, I will describe two "subroutines": staticregistertable and PalAccess.

3.1.1 staticregistertable

This routine calculates the address of the static table within the allocated address region of the PALCode and returns this address to the caller. The static register table is a storage area within the PALCode. This area stores Sombrero specific information of interest to the PALCode. The header file ArcSombreroPal.h defines the offsets or static registers within this area. It that file they are referred to as PAL pseudoregisters. There are several offsets that are important to this presentation:

> THREADNAME – the address of the TCB of the running thread
> GPDOMAIN – the general protection domain of the running thread.
> INTERRUPTBASEVECTOR – the physical address of the interrupt vector table
> INTERRUPTCOUNTER – count of processed interrupts
> SYSTEMINTMASK - the system interrupt priority level IPL as calculated by SOSInterrupt

Description of Logic

```
staticregistertable()
  compute address of static table
  return computed address
```

3.1.2 PalAccess

This routine determines whether or not a caller may invoke a PALCode entry point. It makes this determination by examining the contents of the access table. The access table contains any entry for each PAL entry point. If the entry point is locked, the table entry holds the GPD of the locking thread. Otherwise, the table entry is zero. The access table follows the static table in memory.

Description of Logic

PalAccess()

```
  call staticregistertable()
  from address of the static table,
    compute the address of the access table
  from the caller's return address, compute the PAL entry point
  from the entry point, compute the index into the access table
  if the content of the table entry equals zero OR
     the content equals GPDOMAIN
    return
  go to invalid instruction processing or something
```

3.2 PALCode Entry Points

3.2.1 Intbase

This PALCode entry point stores the physical address of the interrupt vector table within the Sombrero static registers.

Description of Logic

```
Intbase()
  Call PalAccess()
  Remember value of static register INTERRUPTBASEVECTOR
  Store caller's value for INTERRUPTBASEVECTOR
  Return remembered value
```

3.2.2 Systemintmask

This entry point stores the caller specified value for the system IPL in the SYSTEMINTMASK Sombrero static register and returns the previous value of that register

Description of Logic

```
Systemintmask()
  Call PalAccess()
  Remember value of static register SYSTEMINTMASK
  Store caller's value for SYSTEMINTMASK
  Return remembered value
```

3.2.3 InterruptCheck

This is the entry point for processing an interrupt. It primarily checks to see if the system should process the interrupt. It does some direct servicing of the timer counter interrupt to reduce clock jitter.

Description of Logic

```
InteruptCheck()
  Call staticregistertable
  Increment the static register INTERRUPTCOUNTER
  If the interrupt IPL is higher than the system IPL
    If the interrupt IPL is 22
      If the 21174 timer counter is > 666000
        Reset the timer counter
      Else
        Reset the timer counter interrupt in the 21174 IRR
      Return to the interrupted thread
    Set the system IPL to the interrupt IPL
    goto ScheduleInterruptThread()
  Else
    return to interrupted thread
```

ScheduleInterruptThread

This is the primary interrupt processing logic. It attempts to find an interrupt thread to service the current interrupt request. It searches the interrupt vector table created by SOSInterrupt.

If this logic does not find an interrupt thread, the system IPL is not lowered. This logic should perhaps also match on the highest, left-most, bit set in the IRR although it is unlikely that more than one bit is ever set.

Description of Logic

```
ScheduleInterruptThread()
  Recover the address of the interrupt vector table
    from the static table
  Using the interrupt IPL, compute the interrupt vector table index
  Compute the start of the vector list
  While there entries in the vector list to examine
    If the entry's request field corresponds to a set bit in the IRR
      Set the cyclethread field to zero
      If there is a queued thread
        Reset the request bit in the IRR
        If there are more threads in the list
          Make the next thread the new request list head
        Else
          Set the list head and tail to zero
        For the interrupt thread, set tcb.tqueue.data to zero
        For the interrupt thread, set tcb.tqueue.eventname to zero
        If the state of the current thread is interrupt state
          Store its TCB address in the TCBINTTHREAD field
                        of the interrupt thread
          Set the state of the interrupt thread to interrupt state
          Call reschedulequeue
          Go to ConectSwithInt
      Else
        Continue walking the vector list
    Else
        Continue walking vector list
  Reset all interrupts at this vector level
  Return to interrupted thread
```

3.2.4 blkintth

This is the PALCode entry point 0xA1. A thread invokes this entry point for one of several reasons. A thread can indicate to the PALCode that it wants to block and wait for an interrupt. It can indicate to the PALCode that it did not process the current interrupt and that the PALCode should pass the interrupt to another waiting thread. A thread can indicate that it serviced the interrupt and should now be demoted.

There are five parameters: the interrupt vector number, the interrupt request number, the strategy, the time and the timestamp.

The strategies are as follows:

0: block waiting for interrupt
1: block and start next thread in same request list
2: block and start next thread in vector list
4: promote caller to INTERRUPT state, ignore other strategy bits
8: demote caller to RUN state, apply strategies 1 and 2 if selected

Strategies 4, 2 and 1 are exclusive. Only one is applied in the order 4, 2, 1.

The time is the duration before the call times out. I am unsure if this is in clock ticks or milliseconds.

The timestamp is the system time from which the timeout duration begins.

This routine does not appear to return an error code

Description of Logic

```
Blkintth()
  Call AccessPal()
  If there is an interrupt vector table
    If there is a vector list
      Set the return value to zero
      While there are entries in the vector list to examine
        If the GPD and the request of the entry match the caller's
          Set the caller's TCBSTATE to INTERRUPTSTATE
          Set the caller's TCBBASESTATE to INTERRUPTSTATE
          Set the caller's TCBINTERRUPTMASK
            to the caller's requested vector
          If strategy is 4, promotion only
            Set the system IPL to the caller's TCBINTERRUPTMASK
            Return from interrupt
          If strategy is 8, demotion
            Call ScheduleNextQueue()
          Else
            if caller requests a timeout
              Call TimeoutRequested()
            Call NoTimer()
            Call ScheduleNextQueue()
      // matching entry not found
      Set caller's TCBSTATE to TREADRUNSTATE
      Set caller's TCBBASESTATE to THREADRUNSTATE
      Call reschedulequeue()
  Return from interrupt
```

```
ScheduleNextQueue()
  Set return value to zero
  If strategy is 1, block and start next thread in same list
    If there is NOT an existing cycle thread
      Store the caller as the cycle thread
    If the request list head matches the cycle thread
      Set the return value to one
      reset cycle thread
    If there is a list of waiting threads
      Call Handoff()
    Else if strategy is 2, block and start next thread in vector list
        Call ChainScan()
    Else if caller requests demotion
      Call Demote()
    Else
      Set return value to 1
      Set caller's TCBSTATE to caller's TCBBASESTATE
      Call reschedulequeue()
      Call unblkintthEnter()


TimeoutRequested()
  get current system tick count
  convert tick count to milliseconds
  compute elapsed time from the timestamp
  if elapsed time is greater than time
    set return value to SOS_WAIT_TIMEOUT
    return to caller
  store remaining time in QTCBQUEUEDATA
  store –1 in QTCBEVENTNAME
  store vector number in QTCBQUEUEPHEAD
  store request number in QTCBQUEUEPTAIL
  store GPD in QTCBQUEUEPNEXT

NoTimer()
  call reschedulequeue
  if there is a request list
    add caller to list tail
  else
    make caller head and tail of new list
```

```
Handoff(old TCB, new TCB)
  reset new thread's QTCBQUEUEDATA
  reset new thread's QTCBEVENTNAME
  unlink new thread from TCBINTERRUPTQUEUE
  unlink new thread from request list
  if new thread is last in list
    set list head pointer to zero
  if old TCB is NOT equal to new TCB
    move TCBINTTHREAD from old TCB to new TCB
    set new thread state to INTERRUPT
    call reschedulequeue
  if caller requests demotion
    set old thread state to RUN
    call rescheldulequeue
    set TCBINTTHREAD in caller's TCB to zero
    go to ContextSwitch
  if old TCB NOT equal new TCB
    set TCBINTTHREAD in caller's TCB to zero
    go to ContextSwitch
  set old thread state to INTERRUPT
  return from interrupt

ChainScan()
  While there are entries in the vector list
    If entry's request matches caller's
      If there is a waiting thread in the request list
        Set new TCB to first thread in list
        Call Handoff()
  // at end of vector list
  Call NoChain()

NoChain()
  Set return value to 1
  Set caller's TCBSTATE to TCBBASESTATE
  Set new TCB to old TCB
  Call reschedulequeue
  Go to UnblkintthEnter

DoNotChain()
  If TCBINTTHREAD holds the address of the TCB of an interrupted thread
    Make that TCB the new TCB
    Set TCBINTTHREAD to zero
    Call ContextSwitch()
  Else
    If there is a new thread to which to switch
      Call ContextSwitch()
    Else if static registers have the address of the IDLE thread
      Make the new thread the IDLE thread
      Call ContextSwitch()
    else
      Return from interrupt to caller
```

```
Demote()
  Set caller's TCBSTATE to THREADRUNSTATE
  Set caller's TCBBASESTATE to THREADRUNSTATE
  Set system IPL to value of SYSTEMINTMASK
  Make caller's thread the new thread
  Call reschedulequeue()
  If the caller's TCBINTTHREAD points to an interrupted thread
    Set TCBINTTHREAD to zero
    Call ContextSwitch() to switch to interrupted thread
  Else
    Return from interrupt to caller
```

3.2.5 unblkintth

This is the PALCode entry point 0xA2. The function SOSScheduler::TickSleepQueue() calls this entry point when a timeout expires. The caller passes five arguments to this entry point in registers.

a0: interrupt vector number
a1: interrupt request number
a2: address of TCB of thread to remove
a3: GPD of thread to remove
a4: flag – if TRUE run thread, else suspend thread

Description of Logic

```
unblkintth()
  Call PalAccess()
  Call Enter()
  Return from interrupt

Enter()
  If there is an interrupt vector table
    Convert the caller's vector number to a table index
    If there is a vector chain for this vector number
      Call unblkintth()

Unblkintth()
  While there are entries in the vector chain
    If the entry's GPD matches the caller specified GPD
      If the entry's IRQ matches the caller specified IRQ
        If there is a request list
          Call xxx()
```

```
Xxx()
  While there are entries in the request list
    If this entry's TCBNAME matches the caller specified TCBNAME
      Unlink TCB from request list
      Update list's head and tail pointers as required
      If caller requests thread suspension
        Set entry's TCBSTATE to THREADSUSPENDSTATE
        Set entry' TCBBASESTATE to THREADSUSPENDSTATE
        Call reschedulequeue()
      Else
        Set entry's TCBSTATE to INTERRUPTSTATE
        Set entry's TCBBASESTATE to INTERRUPTSTATE
        Call reschedulequeue()
        Set entry's QTCBQUEDATA to zero
        Set entry's QTCBEVENTNAME to zero
        If caller's TCBSTATE is INTERRUPTSTATE
          Link the entry's TCB to the end of the caller's
            TCBINTTHREAD list
        Else
          Call ContextSwitch() to switch to new thread
```