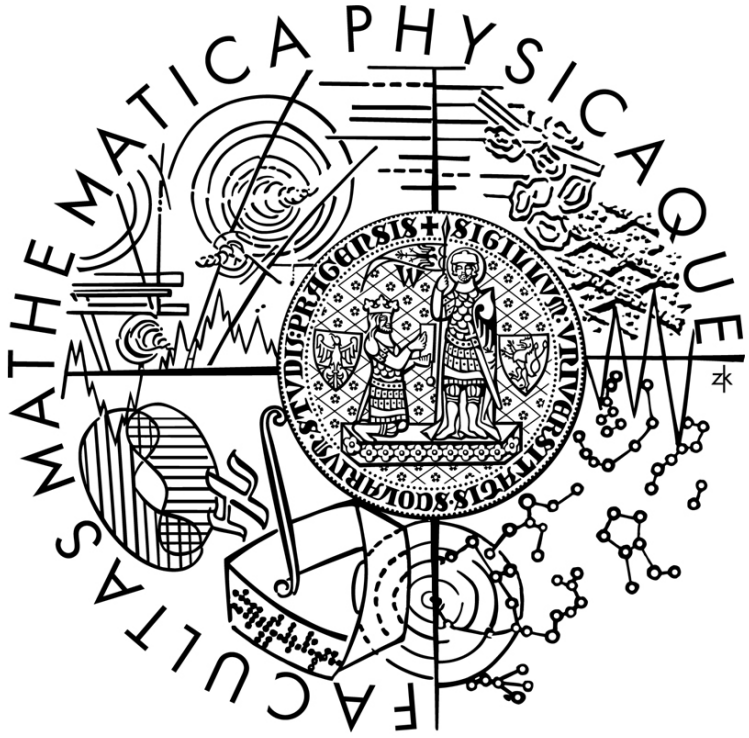
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Jakub Jermář

Porting SPARTAN kernel to SPARC V9 architecture

Department of Software Engineering
Supervisor: RNDr. Jakub Yaghob, Ph.D.

Study program: Computer science, software systems

2007

I would like to thank Mark Sweeney, BMath CSEEE, from Sun Microsystems for carefully and patiently reading and correcting my imperfect English in the entire text of the thesis.

I would also like to thank Mgr. Martin Děcký from MFF UK for running many tests for me and for reading preliminary versions of the thesis as well as giving me suggestions and feedback. I also thank Martin for being such a good HelenOS maintainer and a friend of mine.

I would also like to thank Ing. Lukáš Rovenský from Sun Microsystems for the support which made development and testing of the implementation much easier.

The same words of gratitude go to Mgr. Pavel Semerád from MFF UK who gave me another opportunity to test and develop HelenOS on an Ultra 60 box.

Last but not least, I thank my supervisor, RNDr. Jakub Yaghob, Ph.D. for not being satisfied with some of the first solutions for implementation problems I ran into and for encouraging me to find better ones.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne 3. 4. 2007 Jakub Jermář

Contents

List of Figures

List of Tables

Název práce: Porting SPARTAN kernel to SPARC V9 architecture
Autor: Jakub Jermář
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí diplomové práce: RNDr. Jakub Yaghob, P.h.D.
E-mail vedoucího: Jakub.Yaghob@mff.cuni.cz
Abstrakt: *Přehled vlastností operačního systému HelenOS (základní podpora kernelu, správa paměti, podpora uživatelských aplikací). Přehled vlastností 64-bitové architektury SPARC V9 (registry, trapy, jednotka správy paměti, ASI, hardwarová přerušení). Popis implementace jednotlivých částí systému HelenOS pro architekturu SPARC V9 (boot systému, práce s OpenFirmwarem, převzetí TLB a trap tabulky, přepínání kontextu a FPU context, správa registrových oken a preemptivní trap handler, správa zdroje času, správa vstupně/výstupních zařízení, přerušení, adresové prostory a překlad virtuálních adres pomocí hash tabulky, TSB, řešení problému vzniku ilegálních virtuálních aliasů, podpora systémových volání, podpora čistě uživatelských pseudovláken, start sekundárních procesorů, implementace spinlocku). Přehled a srovnání s implementacemi v operačních systémech Solaris a Linux.*
Klíčová slova: operační systém, jádro, SPARC, HelenOS

Title: Porting SPARTAN kernel to SPARC V9 architecture
Author: Jakub Jermář
Department: Department of Software Engineering
Supervisor: RNDr. Jakub Yaghob, P.h.D.
Supervisor's e-mail address: Jakub.Yaghob@mff.cuni.cz
Abstract: *Overview of the HelenOS operating system (basic kernel functionality, memory management, userspace support). Overview of the 64-bit SPARC V9 architecture (registers, traps, memory management unit, ASI, interrupts). Description of implementation of the HelenOS operating system for the SPARC V9 architecture (system boot, work with OpenFirmware, TLB and trap table take-over, context switching and FPU context, management of register windows and the preemptible trap handler, time management, handling I/O devices, interrupts, address spaces and virtual address translation using the page hash table, TSB, solution to the illegal virtual aliasing problem, system call support, pseudo threads, application processor start-up, spinlock implementation). Overview and comparison of Solaris and Linux implementations.*
Keywords: operating system, kernel, SPARC, HelenOS

Chapter 1

Introduction

1.1 Motivation

Operating systems play an essential role both in software engineering and everyday life. Increasingly demanding applications make users feel uncomfortable when their system doesn't live up to their expectations on performance, responsiveness and stability. Processor vendors have a long tradition in tuning their products to diminish this effect and to give the processors whatever advantage possible. The processor architectures are diverging from each other in considerable ways but still they preserve certain common features. On the other hand, there are software engineers who are striving to match the capabilities of the operating systems they develop with the rich feature sets of modern hardware, while still maintaining portability. This document describes the trade-off between feature completeness and portability on porting the SPARTAN kernel[19] to SPARC V9 architecture[3].

1.2 Goals

This thesis aims to illustrate design and implementation of the SPARTAN kernel[19] port to the UltraSPARC[5] incarnation of the 64-bit SPARC V9 architecture[3].

The porting effort will result in a working kernel with support for:

- basic kernel functionality,
- memory management,
- userspace,
- multiprocessor systems.

The thesis will discuss the above points in considerable detail and provide a comparison with already existing ports of the SPARTAN kernel. The analysis pays attention to places where the new port influences or even changes the generic kernel interfaces. Where considered useful and applicable, alternative approaches used in other operating system kernels will be presented.

1.3 Obtaining Source Code

Source code for the HelenOS operating system, including the sparc64 port, is generally available from the project homepage at `http://www.helenos.eu`. The recommended way of getting the most up-to-date version is via the Subversion repository at `svn://svn.helenos.eu/HelenOS`.

1.4 Acknowledgements

The SPARTAN kernel was initially written by Jakub Jermář. His effort was later joined by Ondřej Palkovský, Martin Děcký, Jakub Váňa, Josef Čejka and Sergey Bondari who formed the HelenOS project and adopted the SPARTAN kernel as a foundation for the HelenOS operating system[19]. The group has improved the kernel considerably and ported it to other processor architectures. The sparc64 port thereof is an exclusive work of Jakub Jermář.

The sparc64 bits of the HelenOS operating system can be found in the following places of the source tree:

- `boot/arch/sparc64`,
- `kernel/arch/sparc64`,
- `uspace/libc/arch/sparc64`,
- `uspace/kbd/arch/sparc64`.

Besides these architecture specific directories, there are also more or less generic contributions brought by the port:

- `boot/genarch/balloc.*`,
- `boot/genarch/ofw_tree.*`,
- `kernel/genarch/{src,include}/ofw`,
- `kernel/generic/{src,include}/ddi/irq.*`.

In addition, this thesis resulted in a great amount of changes throughout the kernel in places that are too numerous to mention. The project Subversion repository provides the entire history of source code changes.

Special recognition goes to Martin Děcký for running many tests of the sparc64 port on his Ultra 60 box and also for switching the amd64, ia32, ia32xen, mips32 and ppc32 architectures over to the new IRQ dispatcher.

1.4.1 License Issues

file, even in cases where the modification completely replaced the original code. Note that some sparc64-specific files were created from a skeleton taken from another port. When in doubt, the project Subversion repository provides information about the authorship of each individual change.

1.5 How to Read This Document

Before the central section describing how the port was actually done, there are two sections describing the characteristics of the SPARTAN kernel and the processor to which it was ported.

Chapter 2 is an architectural overview of the SPARTAN kernel; major subsystems are described.

Chapter 3 is an architectural overview of the SPARC V9 architecture. When it comes to implementation dependencies, the UltraSPARC processor implementation is used as a model.

Chapter 4 covers the design decisions and implementation details of the sparc64 port. It dedicates a large section to each goal of this thesis.

Chapter 5 goes through related work. It attempts to briefly evaluate similarities and differences between the sparc64 port of the SPARTAN kernel and major operating systems that run on 64-bit SPARC processors. A partial comparison is also made for non-SPARC ports of the SPARTAN kernel.

Chapter 6 concludes the thesis.

1.6 Definitions and Terminology

For the sake of brevity and easy readibility, the thesis sometimes uses a term that can be ambiguous or HelenOS-specific. This subsection attempts to identify such terms and provide definition for each use case.

1.6.1 SPARTAN Kernel vs. HelenOS

Throughout this text, two expressions can be freely used to refer to the kernel of the HelenOS operating system and in the context of referring to the kernel can be considered synonyms. These are *the SPARTAN kernel* and *HelenOS*. When the former is used, it never addresses non-kernel parts of the system.

1.6.2 Architectures

Throughout this text, the following terms can be used to refer either to the prospective processor architecture, the particular implementation of that processor architecture or the HelenOS port to that architecture. Depending on context, the following terms are defined:

amd64 — AMD64 Architecture, AMD64 Architecture processor, processor with EM64T technology or the HelenOS port to the AMD64 Architecture;

ia32 — IA-32 Intel Architecture, IA-32 Intel Architecture processor or the HelenOS port to the IA-32 Intel Architecture;

ia64 — Intel Itanium Architecture, Intel Itanium Architecture processor implementation or the HelenOS port to the Intel Itanium Architecture;

mips32 — MIPS 4kc Architecture and MIPS R4000 compatible processors, or the HelenOS port to the MIPS 4kc Architecture and MIPS R4000 compatible processors;

ppc32 — 32-bit PowerPC processors or the HelenOS port to 32-bit PowerPC processors;

sparc64 — SPARC V9 architecture or more narrowly the architecture of UltraSPARC I, UltraSPARC II and UltraSPARC IIi processors, or the HelenOS port to those processors.

Note that in HelenOS terminology, the word *architecture* is sometimes used instead of the word *port* and has the same meaning.

Chapter 2

Kernel Overview

The SPARTAN kernel has supported the following architectures so far: amd64, ia32, ia32xen, ia64, mips32 and ppc32. Among these, there are both 32-bit and 64-bit architectures, including both little-endian and big-endian. The level of support for each architecture differs as some of them run only in a simulator and some of them are relatively new ports. The situation is outlined in table 2.1.

Port	Creation	Endianness	Real HW	SMP	Comment
amd64	2005	little-endian	yes	yes	
ia32	2001	little-endian	yes	yes	
ia32xen	2006	little-endian	yes	no	basic kernel functionality
ia64	2005	little-endian	no	no	
mips32	2003	both	yes	no	
ppc32	2005	big-endian	no	no	

Table 2.1: Comparison of SPARTAN kernel ports.

The kernel is capable of symmetric service on shared-memory multiprocessor systems. There is no coarse-grained lock in the system synchronizing access to every global resource (e.g. threads, processors, tasks and address spaces). Instead, the kernel deploys myriads of smaller locks to achieve good concurrency. SMP enabled ports are amd64 and ia32.

The ultimate document describing generic kernel subsystems is [1] and the reader is encouraged to read it in order to learn about the operating system's internals in detail. The following overview covers topics referenced in chapter 4.

2.1 Basic Kernel Functionality

2.1.1 Time Keeping

The kernel handles hardware interrupts among which is the system clock interrupt. This interrupt is programmed to occur periodically in a fixed relation to the real time so that the kernel can translate the number of clock interrupts to the number of microseconds. The clock interrupt is the source of kernel preemption which is essential in ensuring that no thread, including a kernel thread, can monopolize the processor. Additionally, any part of the kernel can request delayed execution of a registered function. This capability is widely used by the synchronization primitives.

2.1.2 Threads and Scheduling

Basic scheduling entity is one thread. A thread is a light-weight execution unit with its own kernel stack and minimal state information. A task contains a set of threads and a pointer to its address space. There can be many tasks with many threads running in a system. The scheduler services threads in a round robin fashion with multilevel feedback. A running thread, including a thread executing in kernel context, can be preempted by the scheduler. Processor affinity of threads is accomplished by separate run queues for each processor. Starving is prevented by dedicated kernel threads that load balance system processors.

2.1.3 Synchronization

Synchronization in the SPARTAN kernel builds heavily on the implementation of wait queues. A wait queue is a SMP-safe synchronization primitive that can be used by threads to wait for a particular event. Threads are woken up in the order of arrival and their sleep in the wait queue can time out or be interrupted, if requested by the thread. A simple semantic change of the wait queue, pre-setting the `missed_wakeups` member of the wait queue to the initial semaphore value, results in the semaphore synchronization primitive. Furthermore, the mutex is just a binary semaphore. A combination of a mutex and a wait queue creates a condition variable, which can be also used for awaiting events. The most complicated synchronization primitive is a reader-writer lock. Reader-writer locks in the SPARTAN kernel are implemented in a way that will not starve either the readers or the writers.

Because all synchronization primitives described in the previous paragraph derive from the wait queue, all of them are SMP-safe and subject to timeouts and explicit interruptions.

2.2 Memory Management

2.2.1 Frame Allocator

Early during kernel initialization, the frame allocator sets up its structures in order to prepare to satisfy requests for allocations of contiguous pieces of physical memory address space. To

achieve this, the frame allocator makes use of the buddy system. Once initialized, it can be asked to allocate consecutive physical memory frames, in arbitrary powers of two, up to the size of the whole physical memory.

2.2.2 Slab Allocator

Although it is a client of the frame allocator, the slab allocator represents a very effective and cache-sensitive way of managing kernel memory allocations. The idea behind the slab allocator is that the kernel allocates objects of limited types and sizes. The allocator can therefore preallocate some space for each type of object. When the preallocated space drops below the low water mark, the slab allocator asks the frame allocator to allocate some more physical memory. Similarly, when the frame allocator starts to run out of memory, it asks the slab allocator to release some memory from its caches. In most cases, this allows the allocator to satisfy requests instantly.

Moreover, the slab allocator can save its client some time by caching unused but initialized objects. When such an object is allocated, it needn't be initialized again.

The slab allocator also elegantly imitates the heap allocator by satisfying `malloc()` allocation requests, automatically growing or shrinking the heap size.

2.2.3 Address Spaces

Most hardware architectures have support for memory contexts[1]. The main purpose behind memory contexts is to allow translations from multiple address spaces to coexist in translation lookaside buffers. In presence of this hardware feature, the operating system is required to implement a strategy for assigning unique memory context identifiers to address spaces. HelenOS defines an API for this purpose and supplies one implementation based on an in-array queue of unused identifiers. Address spaces are assigned memory context identifiers from the queue as long as the queue is not empty. When the queue empties, address space identifiers start being stolen from inactive address spaces, which have assigned memory context. Orphaned identifiers go back to the queue upon destruction of the respective address space.

2.2.4 Virtual Address Translation

The SPARTAN kernel provides an API through which its clients can manage mappings of virtual memory to physical. The API is independent from the actual implementation and the representation of the set of mappings. Currently, there are two implementations of the API: 4-level hierarchical page tables and the global page hash table. Some architectures enforce use of one mechanism (e.g. amd64 and ia32 are bound to the former) while other architectures give the implementor the freedom of choice. The API also makes it possible to add more implementations.

[1] Also known as address space identifiers on mips32 and region identifiers on ia64.

2.3 Userspace Support

2.3.1 `sysinfo` Subsytem

The SPARTAN kernel propagates essential information about the system to userspace via a tree-like data structure and subsystem called `sysinfo`. The kernel populates the sysinfo tree with key-value pairs. The textual representation of keys uses dotted notation. For example, `fb.address.physical` encodes the position in the tree of the key containing the physical address of the framebuffer.

2.3.2 Syscalls

There are 29 syscalls through which the userspace requests services from the kernel. Not all 29 syscalls are needed or implemented for all architectures. The biggest share of the syscalls is used for IPC. Other syscalls handle task and thread management, address space management, synchronization, security and reads from `sysinfo`.

2.3.3 IPC

Due to its microkernel design, communication among tasks is essential in HelenOS. Tasks can communicate either by sending very short messages that always fit into a few processor registers or by sharing memory. The memory sharing protocol is, however, also handled by the IPC. The IPC subsystem plays an important role for userspace drivers because it forwards interrupt notifications to tasks.

2.3.4 Pseudo Threads

The userspace IPC code makes heavy use of lightweight, purely userspace threads called pseudo threads in HelenOS terminology. Pseudo threads run in the context of the userspace task, on behalf of some thread and cooperatively switch among each other when the need arises. The kernel is not aware of pseudo threads at all.

Thread-Local Storage

Pseudo threads and IPC connection handling code exploit advantages of thread local storage[2]. Small portions of the architecture-dependent standard library code take care of proper allocation and deallocation, respectively, of the thread local storage area when a new thread is created and destroyed, respectively. The pseudo thread switching low-level assembly routines are required to accordingly switch the thread pointer register.

Chapter 3

Architecture Overview

There are several different types of SPARC V9 compliant processors. The SPARC V9 architecture, as described in [3], defines a common ground, on which all implementations build. This common ground is mainly composed of the definitions of the registers, traps, trap mechanisms and instructions. The SPARC V9 architecture deliberately does not cover certain details and leaves them undefined as numbered implementation dependencies for customization by the architect of the processor. These dependencies are then typically defined in processor architecture supplements or in some other subspecifications. Moreover, the SPARC V9 architecture differentiates between mandatory and optional features. Mandatory features are those that must not be changed nor omitted by the implementation. On the other hand, processor models are free to not implement optional pieces and replace them as implementation dependencies permit[1].

UltraSPARC I and UltraSPARC II processors share a common processor architecture supplement[5] to the SPARC V9 specification. Newer processors, starting with UltraSPARC III and Fujitsu's SPARC 64V, are based on the JPS1 subspecification[4] of the SPARC V9 architecture. The new Niagara processors have their own supplements. The UltraSPARC family of processors share a memory management unit implementation called Spitfire[2] MMU. These UltraSPARC supplements also define somewhat similar units used for receiving and dispatching interrupts.

Processor	Codename
UltraSPARC I	Spitfire
UltraSPARC II	Blackbird
UltraSPARC IIi	Sabre
UltraSPARC III	Cheetah

Table 3.1: UltraSPARC processor codenames.

[1] For example, SPARC V9 uses the optional `data_access_MMU_miss` trap to signal a data TLB miss, while UltraSPARC processors use the implementation-dependent `fast_data_access_MMU_miss` trap for the same purpose.

[2] Spitfire is a codename for the UltraSPARC I processor. See table 3.1 for a more complete list of UltraSPARC processor codenames.

Besides processor manuals, there is another technical document[6] relevant to SPARC V9. It defines the 64-bit ABI. The ABI determines, for example, what registers are preserved and what registers are scratch, the calling convention, and the intended structure and use of the stack frame.

If not stated otherwise, the following text will focus on the UltraSPARC II processor, which was the target processor for porting HelenOS to the sparc64 architecture.

The following sections cover several sparc64 features referenced in chapter 4. The reader is recommended to consult other sources to learn about all features of the SPARC V9 architecture. The third chapter of [3] contains a decent introduction into the architecture and the rest of the book provides the necessary details.

3.1 Registers

Integer general purpose registers are broken down into two groups. The first group is represented by global registers. The second group consists of so called windowed registers. The programmer is given the opportunity to access up to 32 general purpose registers (`r0–r31`) at a time. However, the architecture offers more than one needs and, in several ways further described below, provides far more registers.

3.1.1 Global Registers

At any time, there are 8 global registers (`r0–r7` or `g0–g7`) which the application is able to access, out of which seven can be used for storing arbitrary computation data. The `g0` register always reads as zero and writes to it are ignored. In order to reduce the size of the processor interrupt context (i.e. the amount of registers that must be saved by a trap handler), SPARC V9 architecture internally contains and switches between two sets of global registers: normal globals and alternate globals. All UltraSPARC and JPS1 processors add two more sets: interrupt globals and MMU globals. The normal globals are switched on by default when the processor is not servicing a trap. Other sets are automatically enabled when a trap occurs and shadow all other sets. Different traps switch to different sets of global registers.

The concept of several sets of global registers, out of which only one is accessible to unprivileged software, is similar to the one used on ia64, where there are two banks of registers `r16–r31`.

3.1.2 Windowed Registers

Windowed registers are more interesting from the perspective of an operating system writer as they require active support from the operating system kernel. Registers `o0–o7` or `r8–r15` are output registers and are used to pass argument values from the caller to the callee on a procedure call and to read return values on return from the callee. Local registers, `l0–l7` or `r16–r23`, are private to a procedure. Finally, `i0–i7` or `r24–r31` are called input registers and are used by the callee to read arguments passed by the caller on a procedure call and to write return values back to the caller on a procedure return. To avoid ineffective copying and to automate passing

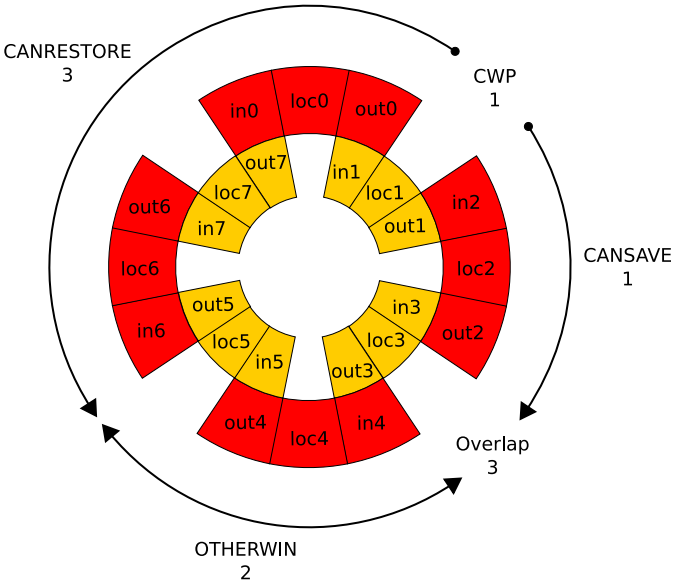


Figure 3.1: Register windows of the UltraSPARC processor.

arguments in the fashion described above, the output registers of the caller overlap with the input registers of the callee.

The output, local and input registers visible to an application compose one register window. Because the number of hardware register windows is limited (e.g. on UltraSPARC processors their number is 8), the last implemented window overlaps with the first implemented window. More precisely, the output registers of the last window overlap with the input registers of the first window and vice versa. The finite nature of the hardware register window set is turned into an effectively infinite stack of register windows by a register window overflow and underflow detection mechanism. This mechanism, programmable by the operating system, triggers dedicated traps that spill valid register windows to memory on overflows and fill invalid register windows from memory on underflows. For more information about register window traps, see subsection 3.1.3.

Figure 3.1 depicts a situation on the UltraSPARC processor. Each window has three compartments: one for the input registers, one for the local registers and one for the output registers. For example, `loc5` refers to all local registers in window 5 and `in0` refers to the input registers of window 0. At any time, one of the windows is the current window and its number is stored in the `CWP` register. The `CWP` register moves clockwise as the nesting of procedure calls deepens and `SAVE` instructions are being executed by the processor. Numerically, the `CWP` register increases modulo the number of windows. When procedures start to return and execute `RESTORE` instructions, its value decreases modulo the number of windows, and the movement becomes counter-clockwise.

Several other state registers influence the functionality of the windowing mechanism. These registers are called register window configuration registers. The `CANSAVE` register determines

how many executions of the SAVE instruction will not cause a window spill trap. Similarily, the CANRESTORE register determines how many times the RESTORE instruction can be used before a window fill trap. Through the OTHERWIN register, a group of windows from a different address space can be marked. Window spills and fills, respectively, of the other windows are serviced by a different set of spill and fill traps. Finally, due to the cyclic nature of the window file, one of the windows is inevitably overlapped from either side: its input registers overlap with the output registers of the last window in the CANSAVE area and its output registers overlap with the input registers of the first window in the OTHERWIN area or the last window in the CANRESTORE area. This window is called the overlap window.

The operating system is responsible for keeping the register window configuration registers in a consistent state. The consistent state is defined by the following equation:

$$\texttt{CANSAVE} + \texttt{CANRESTORE} + \texttt{OTHERWIN} = \texttt{NWINDOW} - 2,$$

where NWINDOW is the total number of register windows (i.e. 8 for the UltraSPARC processor).

Interestingly, as described on p. 329, section H. 1. 5., of [3], the operating system is completely free to decide whether to use windowing of windowed registers or not. If window management instructions[3] are avoided in procedure prologues and epilogues, both in the lowest level assembly code and in the compiler-generated code, the 32 general purpose registers can be used in the flat mode, similarly as on mips32 or ppc32 architectures. While it is easy for an operating system developer to avoid placing such instructions into hand-written assembly code, one must pass special options[4] to the compiler in order to reach the same goal. The trade-off between using the windowing mechanism versus not using it, is a better overall performance versus more deterministic latencies of procedure calls and simpler design.

Windowed registers on sparc64 bring another similarity with ia64 to mind. As described in chapter 6 of [11], ia64's registers r32–r127 are called stacked registers and are being used in a pretty analogous way to windowing to pass arguments between procedures and to store procedure local variables. They are also very complex and difficult to handle properly. Contrary to sparc64, the stacked registers of ia64 don't use windows, but deploy a specialized stack instead. The register stack engine (RSE) would, mostly independently of the program control flow, virtualize the set of available hardware registers by spilling and filling registers, which are currently out of the current stack frame, to or from the backing store (i.e. register stack in memory).

3.1.3 Register Window Traps

Each register window spill corresponds to either one of 8 spill_n_normal or one of 8 spill_n_other traps. Similarly, each register window fill corresponds to one of 8 fill_n_normal or one of 8 fill_n_other traps—in all cases, n ranges from 0 to 7. Table 3.2 shows criteria that influence the selection of the final trap. The WSTATE register plays an important role here. The operating system can determine the trap by programming its two parts (WSTATE.NORMAL and WSTATE.OTHER) according to its needs. WSTATE.NORMAL determines the trap in case OTHERWIN is zero. If OTHERWIN is non-zero, WSTATE.OTHER prevails.

[3] SAVE and RESTORE.

[4] gcc up to version 4.0.2 supported -mflat option for achieving this.

WSTATE.NORMAL	WSTATE.OTHER	OTHERWIN > 0	spill/fill	trap taken
x	y	no	spill	spill_x_normal
x	y	no	fill	fill_x_normal
x	y	yes	spill	spill_y_other
x	y	yes	fill	fill_y_other

Table 3.2: Register window traps.

Besides the traps generated during window overflows and underflows, the clean_window trap triggers every time during the SAVE instruction when the CLEANWIN register is zero. The idea is to prevent information leak from windows that still contain data from different flows of control or, more importantly, different address spaces. For a given flow control and when there are no modifications to this register done by the kernel, CLEANWIN starts at zero and grows until it reaches NWINDOWS $-$ 1. One entity that increments this register is the RESTORED instruction used at the end of the fill handler. Another place where CLEANWIN is supposed to increase is the clean_window trap handler itself. In this case, the operating system must assist the hardware and manually increment the register.

Finally, the FLUSHW instruction can be used by software to enforce spilling of all valid windows into memory. The mechanism triggers the respective spill trap handler until CANRESTORE and OTHERWIN are zero.

3.1.4 Stack Bias

The notable attribute of both the stack pointer and the frame pointer registers (i.e. o6 and i6), respectively, is that they are biased by 2047 bytes towards lower addresses. In other words, both of these registers point 2047 bytes below the actual top of the stack or the beginning of the frame, respectively. This changes the reach of stack relative addressing that uses a 13-bit signed immediate operand of load and store instructions. With the bias, the forward reach is sacrificed but the code can address more of the older stack content.

3.2 Traps

3.2.1 Trap Types

The architecture can recognize 512 different traps. Each trap is represented by a unique number—a trap type—ranging from 0 to 511. Not all trap types are used however and sometimes several trap types represent a single trap.

3.2.2 Trap Levels

A remarkable feature found on SPARC V9 processors is the trapping mechanism. Depending on implementation, the processor contains a hardware trap stack of a certain depth. For instance, the UltraSPARC's trap stack is 5 levels deep. In combination with multiple global register sets (as described in 3.1.1), the trap stack reduces, and for some trap types completely eliminates, the need to save the interrupted register state on the memory stack.

Normal code executes on trap level 0 as indicated by its `TL` register being equal to 0. When a trap occurs, `TL` is incremented and essential state registers (i.e. integer condition code register, `PSTATE`, `CWP`, `PC` and `NPC`) are saved in the `TSTATE`, `TPC` and `TNPC` registers, respectively (with `TSTATE` encapsulating the first three); the trap type is stored in `TT`. Registers `TSTATE`, `TPC`, `TNPC` and `TT` are defined for trap levels greater than 0 so that the processor can handle several nested traps without destroying any interrupted state. It would not be an error to picture these registers as register arrays indexed by `TL`. The trap level can be changed manually by writing to the `TL` register. `RETRY` and `DONE` instructions reverse actions taken when the trap occurred. Registers are restored from the respective `TSTATE`, `TPC` and `TNPC`, and both instructions decrement `TL`.

RED State

When a trap is taken on a trap level which is one level below the maximum trap level (i.e. 4 on UltraSPARC), the processor enters the RED[5] state. When this happens, control is transfered to a hardwired address, memory translation is turned off and the operating system irreversibly loses control over the machine.

3.2.3 Trap Table

The trap table is a code table with two times 32 bytes allocated for each trap type. In total, the trap table is 32K large. The first half of the table contains the beginning for trap handlers for all 512 trap types taken on trap level 0. Analogous to that, the second half is for traps taken on trap levels greater than or equal to 1. Most traps are associated with only one trap type and thus their in-table trap handlers span only 32 bytes, or 8 instructions, in each half. However, some traps are given 4 consecutive trap types (i.e. 32 instructions), which makes it possible to cram the whole handler into the trap table itself.

3.3 Spitfire Memory Management Unit

The MMU is split into the instruction memory management unit (IMMU) and the data memory management unit (DMMU). Each unit has its own registers and data structures.

[5] Reset, Error, Debug.

3.3.1 Memory Contexts

The Spitfire MMU supports 13-bit memory contexts. Memory contexts are identifiers that are used to isolate mappings from different address spaces within a translation look-aside buffer (see 3.3.2). Other architectures might use different names for the same device (e.g. ASID or RID). See table 3.3 for comparison of memory context widths on different architectures.

Architecture	Width	Name
amd64	0	N/A
ia32	0	N/A
ia32xen	0	N/A
ia64	18–24	RID (Region Identifier)
mips32	8	ASID (Address Space ID)
ppc32	24	VSID (Virtual Segment ID)
sparc64	13	Memory Context

Table 3.3: Memory context implementation comparison.

3.3.2 Translation Look-aside Buffers

Both instances of translation look-aside buffers (i.e. ITLB and DTLB) have 64 entries and are the only means the hardware uses for virtual address translation. Optionally, the operating system can make use of precomputed pointers into software managed translation storage buffers. Each TLB entry is 128-bits long and consists of two parts:

- TLB tag and
- TLB data.

A TLB tag is used by the hardware to recognize a TLB hit. It contains memory context, the global bit and virtual page number associated with the entry. The TLB data contains several bits of interest and the physical frame number. The interesting bits are:

valid — valid entries must have this bit set,

locked — locked entries are not automatically removed from the respective TLB,

size — size of the page mapped by the entry[6],

privileged — privileged pages can only be accessed by the operating system,

writable — writable pages can be written to.

[6]See table 3.4 for the list of supported page sizes.

Architecture	Page sizes
amd64	4K, 2M
ia32	4K, 2M, 4M
ia64	4K, 8K, 16K, 64K, 256K, 1M, 4M, 16M, 64M, 256M, 4G
mips32	4K, 16K, 64K, 256K, 1M, 4M, 16M
ppc32	4K, some powers of two between 8K and 256M
sparc64	8K, 64K, 512K, 4M

Table 3.4: Comparison of supported page sizes across architectures.

3.4 ASIs

Besides ordinary memory, the SPARC V9 processor can also access other address spaces. In this case, the word address space is not related to memory management but rather represents an addressable entity—memory or hardware register. The processor uses special identifiers, ASIs, to select what space is being accessed. For this to work, load and store instructions have versions that load and store, respectively, from an explicitly specified address space. Owing to ASIs, it is possible to have both the kernel and the userspace application use the whole 64-bit address space.

The Spitfire MMU is a grateful example of a unit which is exclusively controlled by programming registers residing in MMU ASIs.

3.4.1 Synchronization between Memory and ASIs

Stores to some of the ASIs are a little complicated due to the necessity to carry out certain synchronizing instructions after the store. Some of the ASIs require `RETRY`, `DONE` or `FLUSH` instructions while some others will be satisfied with a mere `MEMBAR #Sync`[7]. `FLUSH` is a heavyweight instruction used for flushing the instruction cache. Moreover, it behaves like a full memory barrier and its biggest disadvantage is that it takes an address operand. When the operand is not mapped by the DTLB, the instruction generates a DTLB miss. `MEMBAR` is the lightweight memory barrier instruction. It cannot cause a DTLB miss and its impact differs depending on the bitmask it takes as an operand. In this case, the `#Sync` bitmask requires that all instructions preceding the barrier are completed and effects of any traps are visible before any instruction following the barrier is executed.

3.5 Interrupt Unit

Interrupts from devices and other processors are received by the interrupt unit. The unit is also responsible for dispatching software-initiated inter-processor interrupts (IPIs) to other processors. On pre-JPS1 processors, the interrupt unit is composed of several receive and dispatch ASI

[7]The first two instructions were mentioned in subsection 3.2.2.

registers. Through these registers, interrupt packets can travel in both directions. There are three data registers for receiving and three data registers for dispatching interrupts. Dispatch registers of the sender will become receive registers of the destination processor upon receipt.

UltraSPARC-IIi can only make use of the first receive register. Other registers are hardwired to zero. This is justified by the fact that UltraSPARC-IIi cannot be used in SMP systems. On the other hand, the UltraSPARC-II processor can use the other data registers arbitrarily.

By hardware convention, the first data register contains an 11-bit unique identifier of the interrupter. Nevertheless, software can write wider values to that register.

Besides the data registers, the unit is further composed of a receive register that combines status and control functionality and a pair of status and control dispatch registers.

When the unit receives an interrupt, it generates the `interrupt_vector_trap` trap.

Chapter 4

Design and Implementation

The efforts associated with this master thesis did not result only in the sparc64 port limited to the SPARTAN kernel, but also in the complete port of the broader HelenOS system, including boot infrastructure and the userspace layer.

4.1 Supported Environments

The sparc64 port of HelenOS was developed in its entirety on the simulated model of the SunFire server Sun Enterprise E6500. The virtual environment was provided by the Simics simulator, which was also used during the development of the majority of other HelenOS ports. Even though it is virtual, the simulated machine can be considered a faithful and complete model of the real SunFire server. In order to prove viability of the sparc64 port on real hardware, the author has also made sure that HelenOS functions on a Sun Ultra 5 and two slightly different Sun Ultra 60 machines. Except for two minor issues which will be discussed later, this goal has been accomplished. Table 4.1 summarizes the main differences between the four environments.

Machine	CPU	SMP	Memory base	Framebuffer	Keyboard controller
Enterprise E6500	UltraSPARC II	yes	0G	ATI 3D Rage XL	Zilog 8530
Ultra 5	UltraSPARC IIi	no	0G	ATI 3D Rage Pro	UART 16550
Ultra 60	UltraSPARC II	yes	2G	Creator 3D	UART 16550
Ultra 60	UltraSPARC II	yes	2.5G	Creator 3D	UART 16550

Table 4.1: Summary of supported environments.

4.2 Boot Process

In a diskless environment that does not understand the notion of files and filesystems, loading a microkernel and its essential userspace layer can be somewhat challenging. The task can be

further complicated by the fact that the kernel is known to take complete control over the machine very early during its initialization and that it needs to be passed certain information about the system—information which it is unable or unwilling to determine itself.

In order to overcome the outlined difficulties and contrary to the situation on the amd64, ia32 and ia32xen architectures, where a suitable third party Multiboot specification compliant boot loader exists[1], the sparc64 port of HelenOS borrows an idea from the custom boot loaders found in the ppc32 and mips32 ports: it self-extracts from a single loadable image.

HelenOS can be booted over the network using the TFTP protocol directly by OpenFirmware or, alternatively, from a disk or some removable media using the SILO loader. Using SILO allows the system to be loaded from a large palette of filesystems.

4.2.1 Interfacing with SILO

The sparc64 bootable image contains a small loader program and linked-in binaries of the kernel and all the userspace tasks, each residing in a dedicated ELF section. The image is supposed to be loaded by the SILO primary boot loader into an identity-mapped region[2] of memory starting at a fixed virtual address of 0x4000. SILO is primarily intended for loading the Linux kernel and as such needs to be manipulated into doing the right thing for HelenOS. To prevent SILO from loading the image to an unknown and non-identically mapped virtual location, the bootable image must contain a special header used by Linux kernels. Via information contained in the header, HelenOS presents itself to SILO as an old version of Linux which cannot relocate itself. The assembly language form of the header is shown in figure 4.1.

```
.ascii "HdrS"
.word 0
.half 0
```

Figure 4.1: Fragment of Linux header used in the loader.

4.2.2 Copying Kernel and Userspace Tasks

After being successfully loaded by SILO, the HelenOS loader starts executing at virtual address 0x4000. It then copies the kernel to memory at virtual address 4M. The userspace tasks, are, one by one, copied to page-aligned locations following the end of the kernel image.

4.2.3 `bootinfo` Structure

When the software pieces are in place, the loader is responsible for collecting essential information about the system and storing it in the `bootinfo` structure. The address of the bootinfo

[1] When the GRUB2 boot loader matures enough, there will probably be such a loader for sparc64 as well.

[2] In case the physical memory starts at a non-zero address, the mapping is not identity, but is biased by the physical memory start address.

structure will be passed to the kernel along with control once the boot phase is over. Currently, the bootinfo contains these items:

1. starting address of physical memory,
2. map of userspace tasks,
3. boot allocator info,
4. memory map,
5. canonical copy of the OpenFirmware device tree.

The map of userspace tasks contains an array of records describing the memory location and size of each task. It also includes information about the number of records in the array.

The memory map tells the kernel what memory regions are available. Technically, the map is found in the `reg` property of the `/memory` node of the OpenFirmware device tree. Therefore it is not absolutely necessary to precompute the memory map for the kernel in the loader—the canonical copy of the device tree is passed to the kernel and contains all memory map information as well. On the other hand, the ppc32 port deals with the OpenFirmware too, but does not copy the device tree to the kernel. Thus, processing memory map data in the loader is the only way for the ppc32 port. The sparc64 loader is based on the ppc32 version and simply reuses the same code.

4.2.4 OpenFirmware Device Tree

The information contained in the OpenFirmware device tree is very important for the proper functionality of the kernel. Every system equipped with OpenFirmware provides access to this information via a set of OpenFirmware calls. Curiously, the 64-bit kernel avoids direct communication with the 32-bit OpenFirmware[3]. The outcome of this is that the kernel does not use the OpenFirmware interface at all and the device tree is therefore not directly accessible to it. Instead, it depends on the information passed from the loader. The loader is thus required to do a full device tree traversal and copy all discovered nodes and edges into memory. Nodes are copied along with their properties. Figure 4.2 shows a portion of a device tree found on the Enterprise machine.

However, the tree must be traversed with care. Due to the organization of the tree[4], the loader must not use recursion when processing peer nodes. Typically, there will be enough peer nodes in the first level of the tree to overflow the stack if a recursive algorithm was used. It is essential that the loader iterates over the list of peers instead. As for the child nodes, the tree is typically not deeper than a few levels, so the recursive descent algorithm is applicable on the 8K stack.

HelenOS is not the first operating system to work with a memory copy of the OpenFirmware device tree. For example, Linux and xnu[5] also work with an in-memory representation of the device tree.

[3] There are more reasons for this. Besides different addressing capabilities, the correctness of some OpenFirmware implementations is sometimes at least questionable.

[4] Each node contains a child node pointer and a peer node pointer.

[5] Kernel of Mac OS X.

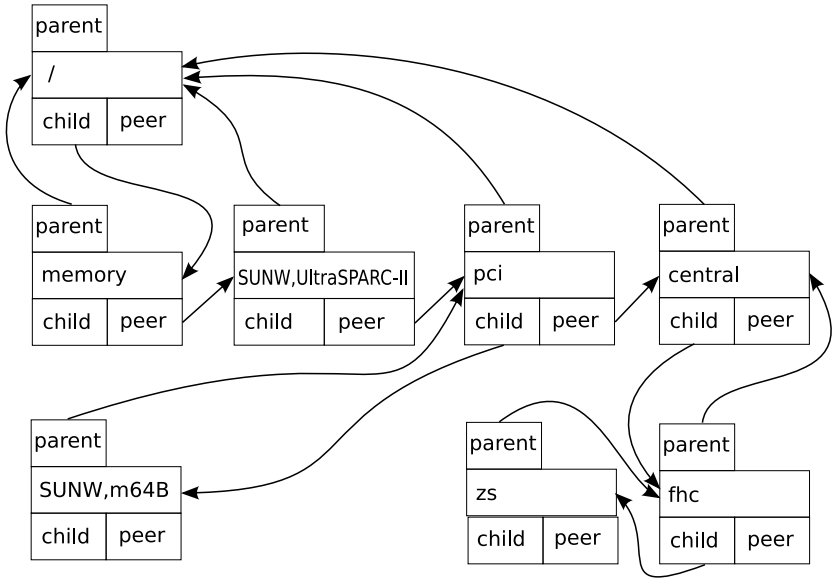


Figure 4.2: Example of canonical copy of the OpenFirmware device tree.

Moreover, the advantage of having the device tree nodes in memory seems to be more straightforward code. For example, the programmer does not have to use a function call in order to reference either the parent, child or sibling of an arbitrary node. Instead, he or she uses pointers for the same purpose.

4.2.5 Boot Memory Allocator

An OpenFirmware device tree is a dynamic structure with a theoretically unlimited number of nodes. It is therefore imperative that the loader allocates memory for the device tree nodes dynamically and on an on-demand basis. For this reason a specialized allocator was implemented. The allocator only knows how to allocate memory—memory passed to the kernel is never deallocated by the loader. Its heap is placed after the end of the last userspace task loaded and is aligned to a page boundary. Each allocation request is satisfied with respect to previous allocation requests and according to the demanded size and alignment constraints. Thus, the top of the heap grows towards higher addresses.

4.2.6 Secondary Processors

Secondary UltraSPARC processors are started via an UltraSPARC-specific OpenFirmware call `SUNW,start-cpu`. Since the kernel does not use the OpenFirmware interface, the processors must be started by the loader.

Regardless of whether the kernel is compiled with support for SMP or not, the loader, if configured so during compile time, detects all secondary processors by searching the OpenFirmware device tree and starts them up one by one. Each processor is told the address at which to start

executing and the initial value of its `o0` register. Contrary to the bootstrap processor, the register is not logically OR'ed with the `BSP_FLAG` in the case of secondary processors. The flag distinguishes between the bootstrap and application processors to the kernel startup code. Due to this design, the secondary processors have the same kernel entry point address as the boot processor.

4.3 Basic Kernel Functionality

4.3.1 TLB and Trap Table Takeover

This subsection refers to the part of kernel initialization which is specific to the sparc64 architecture. The kernel starts executing at virtual address 0x400000. The most important steps the kernel makes in this stage are:

1. taking over the trap table,
2. taking over DTLB and ITLB.

Both of these steps aim to take over complete control over the system from the OpenFirmware, which the kernel does not trust. In the beginning, the TLBs are managed by the firmware and contain various entries, including locked ones. The trap table contains exclusively OpenFirmware trap handlers. Upon completion of the steps outlined above, there will be no way to use OpenFirmware's services, including its command line interface. The TLBs will be populated solely by kernel mappings and the `TBA` register will point to kernel trap table. Interestingly, Solaris and also Linux both coexist with the firmware and can pass control back to it on demand.

After initializing control registers and disabling interrupts, the kernel switches to the kernel trap table. It must ensure that no trap will occur until the trap handlers[6] can dependably work.

The process of taking over the TLBs is not very straightforward and is very error prone. For example, the author had to replace the first implementation that worked on the simulated E6500 after it became clear that the solution was not functional on the real Ultra 5. The first implementation simply disabled each MMU, relying on the fact that the kernel was running from an identity-mapped memory, set up each of the TLBs and enabled MMU again. The reason why this didn't work on the Ultra 5 machine still remains a mystery. In the case of the Ultra 60 machines, this solution would not work either due to the non-zero start address of physical memory.

Nevertheless, a functional implementation was achieved. The kernel first takes over the DTLB. The whole kernel startup code is written in the assembly language so there is no danger of unknowingly causing a DTLB miss when the DTLB is being invalidated. Moreover, in case of writes to data MMU ASIs, which are inevitable during MMU operations, the `FLUSH` instruction is not necessary and a mere `MEMBAR #Sync` can be used to make the ASI write visible. This is very important because, contrary to the `FLUSH` instruction, the `MEMBAR` instruction cannot cause a DTLB miss. When the contents of the DTLB are invalidated, the kernel installs a locked 4-megabyte DTLB entry identically mapping its first 4 megabytes in memory context 0. A non-locked 4-megabyte entry mapping the same memory is installed in memory context 1. The latter is used during the takeover of the ITLB.

[6] Especially the register window trap handlers and MMU trap handlers.

The ITLB takeover is inspired by the OpenBSD approach and the kernel attempts it after it takes over the DTLB as described in the previous paragraph. While running in memory context 0, the kernel installs a non-locked 4-megabyte ITLB entry, mapping its first 4 megabytes in memory context 1, and switches to that context. Running from context 1, it then demaps context 0 and installs a locked 4-megabyte ITLB entry for the kernel there. After that, it switches back to memory context 0. During this trampolining, the kernel needs to issue a `FLUSH` instruction after each write to an instruction MMU ASI in order to propagate the changes. The instruction has one operand which is an address mapped by the DTLB. A fatal DTLB miss is avoided by picking the operand from the address range covered by the kernel DTLB mappings in contexts 0 and 1.

At this point, the TLBs and the trap table are taken over and the kernel is now ready to service and, if possible, survive any trap. Moreover, because the kernel is expected to fit into 4 megabytes, there should be no kernel ITLB misses. The 4-megabyte DTLB kernel entry limits kernel DTLB misses only to virtual addresses outside the kernel image large page. Those misses are resolved by the MMU traps. Instruction and data mappings that are now left over in context 1 will be invalidated during higher-level TLB initialization.

4.3.2 Register Window Traps

Besides the `clean_window` trap, which always stays the same, the preemptible trap handler (which is described in 4.3.5) requires the kernel to service the following register window traps:

- `spill_0_normal` and `fill_0_normal`,
- `spill_1_normal` and `fill_1_normal`,
- `spill_2_normal`,
- `spill_0_other`.

Kernel windows are handled by `spill_0_normal` and `fill_0_normal` traps. Register window spills and fills originating in userspace are serviced by `spill_1_normal` and `fill_1_normal` trap handlers. All of these handlers work with the register window save area, which is part of the memory stack, as described in[6]. On a spill, the local and input registers are saved on the stack. On a fill, the same registers are reloaded back from the stack. The only difference between the 0 and 1 versions is in the ASI used. Kernel window handlers access the stack with full privileges while the userspace window handlers use the `ASI_AIUP` which lowers the privileges for the stack accesses to those of userspace code.

The `spill_2_normal` and `spill_0_other` traps are used for spills of userspace windows initiated by the kernel. Contrary to the previous spill traps, these two don't save registers onto the memory stack but spill them to a dedicated userspace window buffer instead. The userspace window buffer is a page-aligned area large enough to accommodate 7 register windows, which is 896 bytes. The first of the traps can be generated only by the `SAVE` instruction in the preemptible trap handler. The second trap is then triggered every time `CANSAVE` is zero and `OTHERWIN` non-zero.

Note that due to the design of the preemptible trap handler, there is no trap for filling a register window from the userspace window buffer.

4.3.3 Context Support

The sparc64 kernel implements the context switching functions `context_save_arch()` and `context_restore_arch()`. Their intent is to enable the kernel to remember the register context of one thread of execution into a memory structure and to return to this context sometime later. One of the biggest clients of these two functions is the scheduler, another one is the `waitq` implementation.

The implementation is optimized so that only the preserved general purpose registers, as identified by the SCD[6], participate in the context[7]. One exception to this is the `pri` member, which is used by all architectures to preserve the processor interrupt priority. As a side effect on sparc64, the `pri` member absorbs the whole `PSTATE` register.

Register	Alternative name	Comment
`l0–l7`		
`i0–i5`		
`i6`	`fp`	frame pointer
`i7`		return address - 8
`o6`	`sp`	stack pointer

Table 4.2: SPARC V9 registers that compose register context.

Moreover, both functions are leaf-optimized. Leaf-optimization avoids any extra register window spills and, in this case, leads to more easily understandable code because both `context_save_arch()` and `context_restore_arch()` work in the register window which is being saved or restored.

To this point, the sparc64 has been aligned with many other HelenOS ports. However, the register window mechanism complicates the matter. The problem is caused by two factors: `CWP` itself is not preserved across context switches and the save `CWP` is usually different than the restore `CWP`. In addition, the restore operation must not clobber any unsaved windows of the current context.

As illustrated in figure 4.3 on an example register window configuration, the restore function restores the saved window (window 1) in the current window (window 5). That alone would destroy the output registers, especially the stack pointer, of the preceding window (window 4). The solution is to issue the `FLUSHW` instruction and flush all windows, including window 4, in the `CANRESTORE` area to memory stack. Besides protecting these windows against damage, it also makes the preceding window (window 4) the overlap window. With the overlap window immediately preceding the current window, it is safe to restore the frame pointer in the current

[7] See table 4.2.

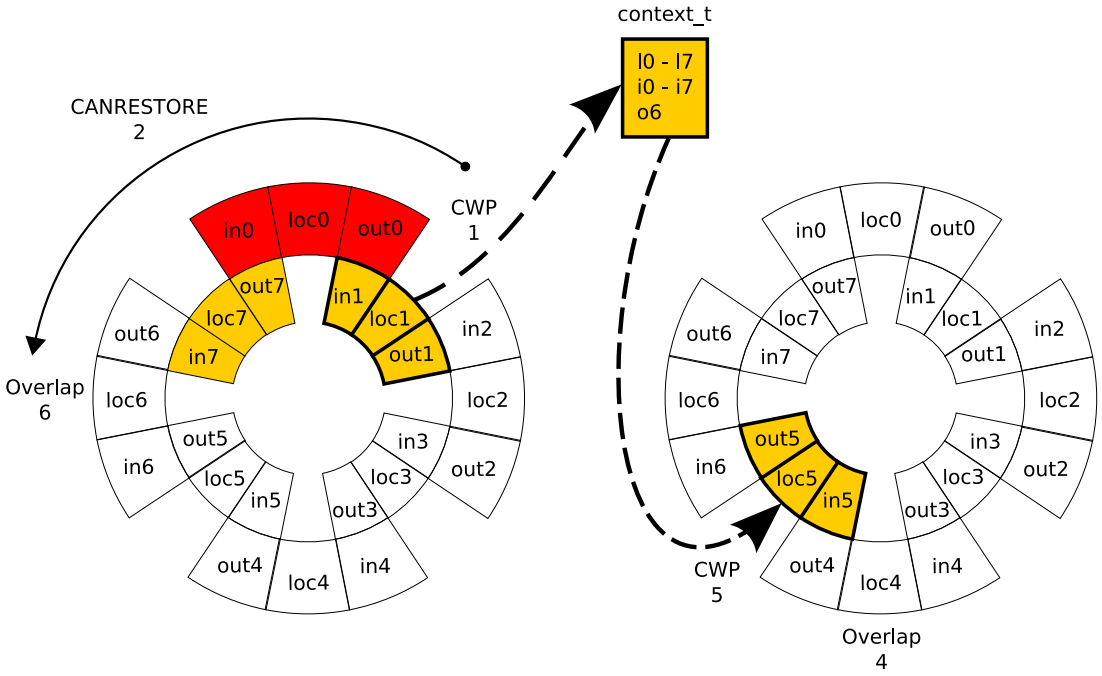


Figure 4.3: Register window interactions during context save and restore.

window and the stack pointer in the preceding window[8]. After these steps, `CANRESTORE` will be 0 and `CANSAVE` will be `NWINDOWS` - 2. Should the `RESTORE` instruction be used in the current window now, the correct content of the saved context register windows (window 0 and window 7, respectively) will be restored by the register window fill trap handler (in window 4 and window 3, respectively).

4.3.4 FPU Context Support

The sparc64 kernel supports both lazy FPU context switching and normal non-deferred FPU context switching. The mode is selected during compilation of the kernel. Both mechanisms make use of the generic FPU context switching framework and supply only architecture-specific hooks. These include functions for disabling and enabling the FPU and functions for saving and restoring FPU registers to and from memory. The `fpu_context_save()` function is used to save the FPU context and the `fpu_context_restore()` function is used to restore the FPU context. Both of these functions work with the full set of FPU registers. The context is composed of 32 double-precision general purpose FPU registers[9] and the `FSR` register.

In lazy mode, the FPU context switch request is triggered via the `fpu_disabled` trap when the processor touches any FPU register when either of the `PSTATE.PEF` or `FPRS.FEF` is zero.

[8] Due to overlap of the windows, it is physically one register.

[9] Which overlap in part or in their entirety with 32 single-precision or 16 quad-precision general purpose FPU registers.

While the PSTATE register is privileged and any attempt to read or write to it with only user privileges leads to a protection violation and punitive measures taken by the kernel against the offending task, the FPRS register can be accessed by non-privileged software. HelenOS does not support userspace controlling FPU context switching. The kernel therefore effectively ignores this bit. On each fpu_disabled trap, the kernel checks whether FPRS.FEF was disabled by the userspace. If that is the case, it reenables it again and quiesces the trap. With this strategy, only PSTATE.PEF really determines whether a lazy FPU context switch is necessary.

Besides FEF, the FPRS register contains two other interesting fields: DU and DL. The former is set when the upper half of the general purpose FPU registers is dirty and the latter is set when the lower half of the general purpose FPU registers is dirty. Had the FPRS register been privileged or had these fields been part of another privileged register, it would be possible to implement the save and restore functions in a way in which only dirty parts of the FPU context were saved or restored.

FPU context switching issues inherent to the environment of purely userspace pseudo threads are discussed in section 4.5.2.

4.3.5 Preemptible Trap Handler

This section deals about the preemptible trap handler. The preemptible trap handler is used to handle all traps with a higher-level service routines that either lower interrupt priority level of the processor or have a call to the scheduler in their codepath. Among these traps, there are all syscalls, the tick interrupt trap, hardware interrupt trap, memory management traps and traps that can lead to forceful termination of the task. When a trap is being processed, the preemptive handler arranges the processor state so that other traps can be taken and serviced before the current one finishes. The preemptive trap handler needs to anticipate all its possible uses, especially with regard to different register window configurations. With all the above mentioned attributes, it becomes the focal point of the whole SPARC V9 port.

Implementation

The preemptible trap handler is implemented as an assembly language macro. The macro produces slightly different code for syscalls and for other, non-syscall, traps. In each case, the handler is invoked from the trap's entry in the trap table. On invocation, the caller must initialize respective global registers with parameters for the preemptible trap handler. The parameters include the address of the higher level service routine and its argument. Registers g6 and g7 from the alternate and interrupt sets contain previously stored addresses of the thread's kernel stack and the thread's buffer for spilling userspace register windows, respectively.

A trap switches the processor's global register set to one of the alternate sets. The preemptible trap handler expects either the alternate global set or the interrupt global set. Memory management traps work with the MMU globals and need to switch to one of these prior to passing control to the handler. However, they are free to use the MMU set before that. The design of the preemptible trap handler guarantees that nested traps will not clobber a global set which was not previously saved by the handler.

The UltraSPARC processor implements five trap levels above trap level 0, on which the processor operates when no trap is in progress. When a trap occurs, the trap level increases by one and the necessary processor state is saved by hardware. This scheme allows trap nesting up to trap level 4. When the processor receives a trap on that trap level, it enters the RED state. For description of the RED state, please, refer to section 3.2.2.

HelenOS is designed to make use of five trap levels without having the machine enter the RED state[10]. Under normal conditions, the kernel will be using trap levels from 0 through 3. Trap level 3 is reached pretty rarely. Nevertheless, there is a small window for this to happen in which several conditions are required to be true[11]. The effect of hitting this window was observed in lab conditions. Refer to table 4.3 to see transitions between processor trap levels involved in the scenario.

TL	Activity	Trap cause
0	Processor executes a kernel thread	Hardware interrupt
1	Processing of `interrupt_vector_trap` trap	`SAVE` instruction
2	Processing of `spill_0_normal` trap	`STX` instruction
3	Processing of `fast_data_access_MMU_miss`	

Table 4.3: Example scenario of reaching TL 3.

In this scenario, the kernel executes a kernel thread with interrupts enabled. A hardware interrupt comes and the processor enters trap level 1. Next, if there are no more register windows that the preemptible handler, which is now servicing the interrupt, can claim, the TL will go to TL 2 upon execution of the `SAVE` instruction. As described later in this text, the preemptible handler executes that instruction in its prologue. Further, if the kernel stack of the interrupted thread is not DTLB-resident, the register window spill handler will cause a TLB miss during an attempt to spill a register window to the kernel stack. The TLB miss will be handled on TL 3.

The author sees a similar scenario, in which an arbitrary thread executing in userspace replaces the role of the kernel thread and the userspace window buffer replaces the role of the kernel stack.

Finally, the last usable trap level, trap level 4, is reserved for civilized servicing of kernel and hardware errors that occur during trap level 3.

Note that the maximum reachable trap level can be lowered by one if the kernel always locks the DTLB entries mapping the kernel stack and the userspace window buffer before a thread is run. The trade-off here is between two spared DTLB entries and faster trap processing.

Despite the simpler design, which allows more code to be written in C, the preemptible trap handler is still pretty complex. The handler expects to run on trap level 1. In case the trap comes from a higher trap level, the trap table code needs to make arrangements to invoke the preemptible trap handler from the proper level.

[10] Like Solaris and Linux do.

[11] `CANSAVE` must be 0 and the kernel stack must not be mapped by the DTLB.

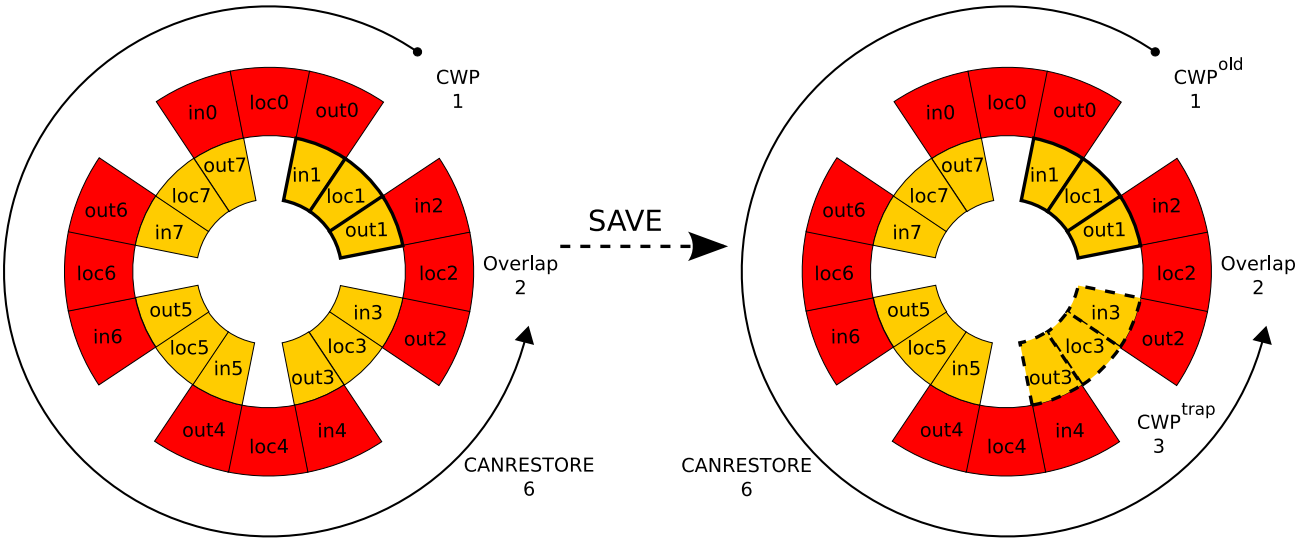


Figure 4.4: CWP change on window spill trap.

Prologue

Once the preemptible handler is entered, it performs a check to see whether the trap level is right. If that is the case, it proceeds and right from the start needs to cope with one of the anomalies.

Imagine that a userspace thread issued the SAVE instruction or the RESTORE instruction when the CANSAVE register or the CANRESTORE register, respectively, was 0. The processor receives the spill_1_normal trap or the fill_1_normal trap, respectively. When the page with the userspace stack is not DTLB-resident, a nested fast_data_MMU_miss trap is taken in an attempt to spill or fill, respectively, the content of the trapping window. Had the original trap not been a register window spill or fill trap, there would be no problem. However, in case of these two traps, the nested trap inherits a wrong CWP register value because the CWP they use is different from the CWP in which the trap happened first.[12] The MMU trap table code invokes the preemptible trap handler and forces trap level 1. The idea is that the preemptible trap handler calls code that refills the DTLB and returns from the trap so that the faulting instruction can be restarted. However, due to the wrong CWP, the outcome of this would be complete chaos—the preemptible handler would start running in the incorrect register window. The situation is best viewed from figures 4.4 and 4.5. Figure 4.4 shows how CWP is incremented by 2 (modulo NWINDOWS) for the spill trap and figure 4.5 depicts how CWP moves one window back for the fill trap. In order to mitigate the potential problem, the handler proactively sets the CWP register to the value that existed in the previous trap level.

When the CWP is correct, the handler looks at the privileges of the previous trap level. If the processor had been executing privileged code before the trap, it is already using the kernel stack and a new register window can be allocated via the SAVE instruction. If the trap came from the

[12] Unlike in case of most of the other traps.

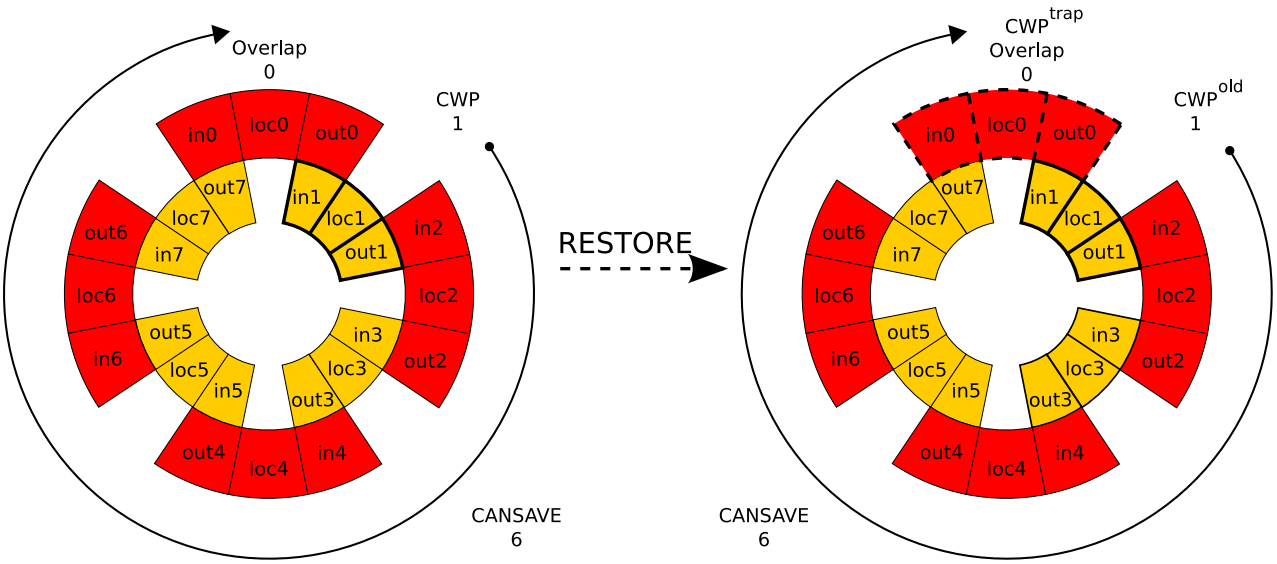


Figure 4.5: `CWP` change on window fill trap.

userspace, the stack needs to be switched to the address found in the `g6` register, but before that happens, some other provisions need to be made.

Trapping from Userspace

Initially, the kernel needs to reconfigure the `WSTATE` register to change the selection of the register window spill vector in favor of the version that spills windows to the userspace window buffer which is referenced by the `g7` register. It is also a good idea to set the `CLEANWIN` register to `NWINDOWS` - 1 at this point so that the kernel will not incur needless `clean_window` traps. The kernel then asks for a register window and switches to the kernel stack via the `SAVE` instruction. If the instruction traps, a register window will be spilled to the userspace window buffer.

At this point, another anomaly can come into existence. Figures 4.6 and 4.7 show two different register window configurations transitioning into a single end register window set configuration via the `SAVE` instruction. In the case of figure 4.6, `CANSAVE` is 0 and the `SAVE` instruction will generate the `spill_2_normal` trap. The userspace window buffer will be populated by the content of window 3 and register `g7` will be incremented to point to the next available position in the buffer. In the case of figure 4.7, `CANSAVE` is 1 and the following `SAVE` will not cause the spill trap. The userspace buffer will stay intact. In both cases, the end configuration of the register window set will have `CANSAVE` equal to 0 and `CANRESTORE` equal to `NWINDOWS` - 2. `CWP` will be the same and the only difference will be the configuration of the userspace window buffer. The anomaly will remain latent until the code in the epilogue tries to refill register windows from the userspace window buffer.

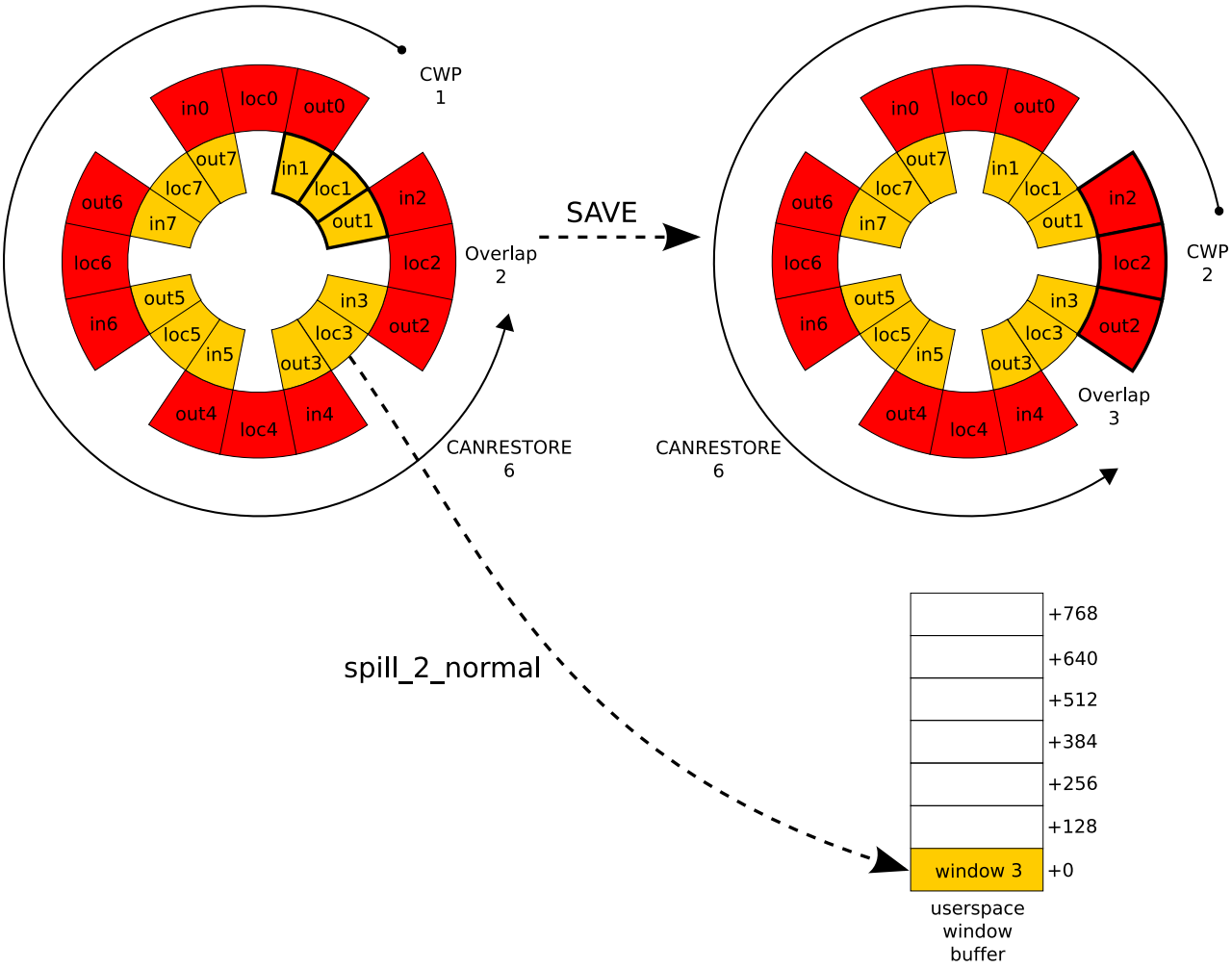


Figure 4.6: SAVE instruction interactions in preemptible trap handler (spill).

Now, when the kernel operates from a dedicated window, it swaps CANRESTORE with OTHERWIN[13] so that the valid userspace windows are marked as other windows and become subject to spilling to the userspace window buffer.

Finally, when running from trap level 1, all memory is mapped from the nucleus memory context[14]. However, the intent of the preemptible trap handler is to fire the higher level service routine from trap level 0. On trap level 0, the default memory context will be the primary memory context so the kernel writes 0 to the Primary Context register.

[13] Which is 0.
[14] Which is 0.

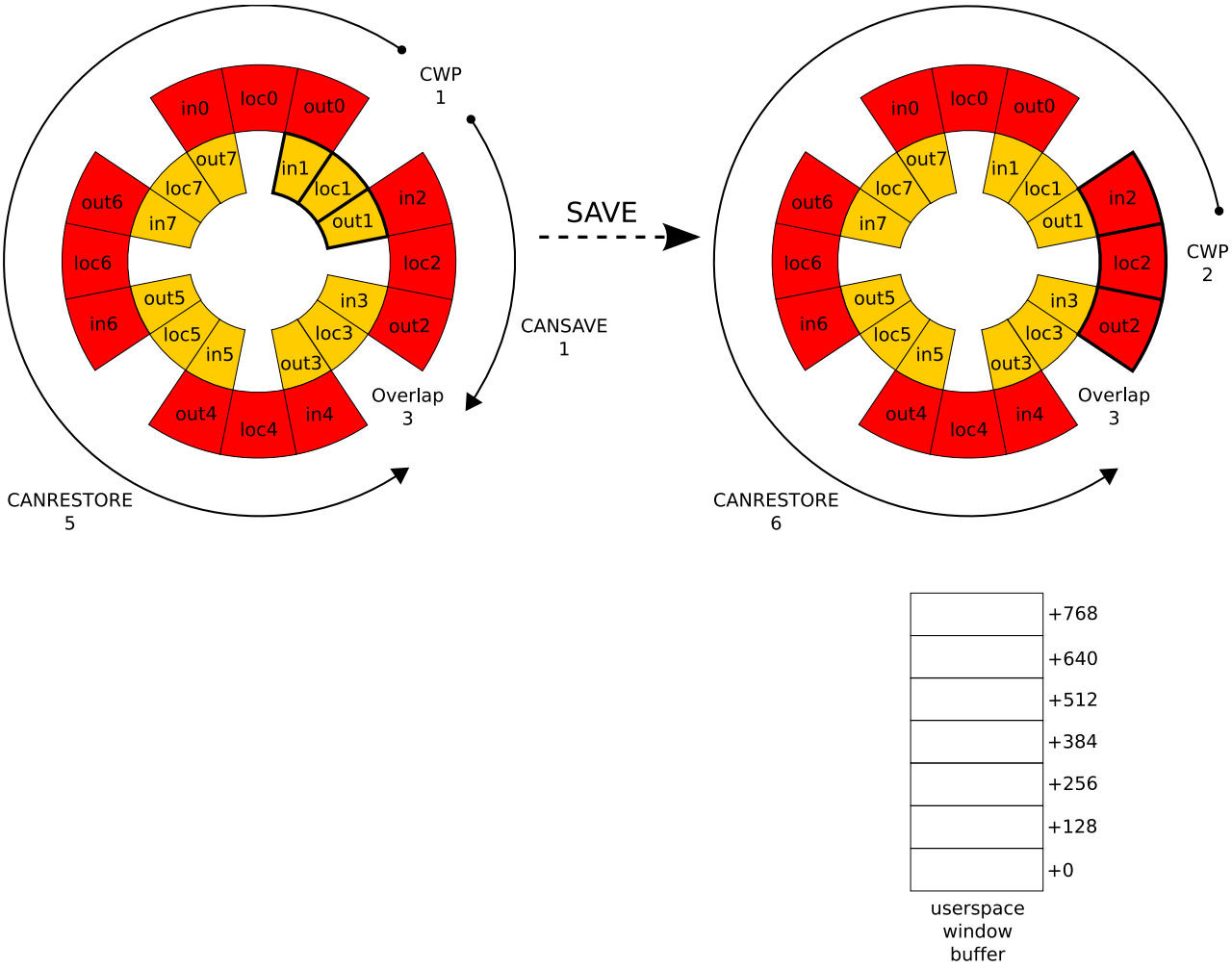


Figure 4.7: SAVE instruction interactions in preemptible trap handler (no trap).

Trapping from Kernel

This case is much simpler than the previous one. The kernel is already using the kernel stack, so the handler only asks for a new register window through the `SAVE` instruction. Should the instruction trap, a register window will be spilled to the kernel stack as usual. The memory context registers already contain the right setting and no modifications are needed.

Middle Part

Now, when the trap handler resolves the two different invocation scenarios, it proceeds with rearranging the register window configuration registers. By further modifying the `WSTATE` register, the spill and fill traps that occur in normal windows go to and from, respectively, the kernel stack. For other windows, spill traps go to the userspace window buffer.

Further, the handler saves the `TSTATE`, `TPC` and `TNPC` registers from the current trap level on the kernel stack. Besides these three, the `Y` register is saved as well.

The `Y` register deserves a dedicated paragraph. The SPARC V9 specification classifies `Y` as deprecated and urges software writers to refrain from using it as well as the instructions that read or write it. Nevertheless, `gcc` generates code that does not comply with this rather strong recommendation. As a result, the kernel must preserve the deprecated `Y` register in the preemptible trap handler as well.

The trap that is being virtually followed is now almost half-way processed. The trap level can drop to 0 now and switch to the normal global register set. The kernel explicitly enables the FPU and after saving all global registers into local registers (`g1` into `l1`, ... , `g7` into `l7`), the higher-level service routine is finally called and is passed its argument. Due to the ceaseless activity of the windowing mechanism, the act of preserving the globals in the locals can be viewed as a deferred spill on the kernel stack.

After the higher-level service routine eventually returns, the whole process of state saving needs to be reverted. The normal global registers are reloaded from the locals and the processor switches to the alternate global set. The choice of the alternate global set is hardcoded on purpose—it is absolutely necessary because the userspace window buffer pointer, the `g7` register, is, for consistency reasons, only maintained in the alternate set. The trap level increases to 1 and the `TSTATE`, `TPC`, `TNPC` and `Y` registers are restored from the stack. Special provisions are made to record the `PSTATE.PEF` bit changes made by the service routine into the trap level 1 `TSTATE.PEF`.

Nevertheless, returning from the service routine has its own issues. By inspecting the value of the `OTHERWIN` register, the handler can tell whether the scheduler was called by the service routine or not. In case `OTHERWIN` is zero, the scheduler was probably called and chances are that the handler resumed execution in a different register window (i.e. `CWP` has changed). This assumption is based on the effect of the scheduler calling `context_restore()`. This function, besides other things, spills all active register windows to memory, effectively clearing the `CANRESTORE` or `OTHERWIN` registers. Therefore, if `OTHERWIN` is non-zero, the scheduler was certainly not called and the register window is correct. In case of discrepancy between the current and the expected value of `CWP`, the handler has to perform a remedy by relocating itself and the

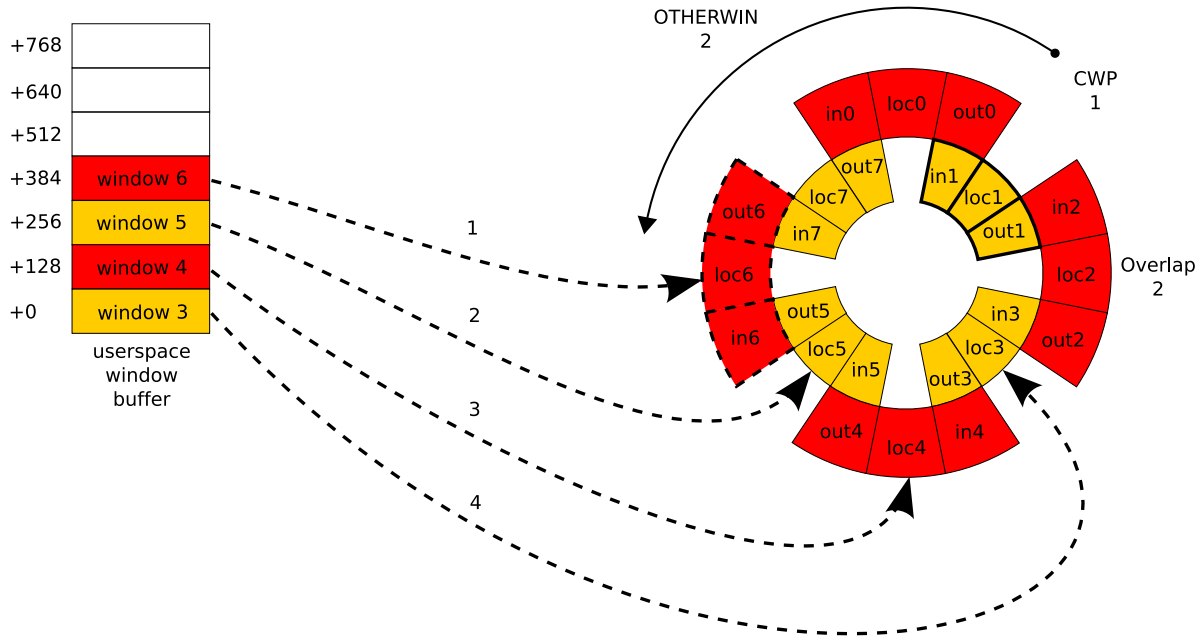


Figure 4.8: Refilling register windows from the userspace window buffer.

input registers to the expected register window. The stack is used as temporary storage for the input registers during the relocation. The input registers are important, because they represent the output registers of the window in which the trap occurred. The handler is responsible for preserving them and they must be copied to the proper window at this point.

Epilogue

Once the preemptible handler runs in the correct window, it makes a check to see whether it is returning to userspace or to the kernel. While returning to kernel is not a big deal, returning to userspace is somewhat challenging.

Returning to Userspace

First, the handler resets the `WSTATE` register so that window fills and spills are handled by the normal handlers again. Second, it reloads the primary context register by the value stored in the secondary context register[15].

Most likely, some userspace register windows got spilled to the userspace window buffer when the handler and the higher level service routine executed. These windows are now refilled from the buffer, as illustrated in figure 4.8. The algorithm increments a counter when a window is refilled from the buffer. At the end of the algorithm, the counter and the `OTHERWIN` register should add up to the `CANRESTORE` register value as it existed before the trap. Let the sum be

[15]The secondary context register functioned as a backup for the primary context register during the trap.

called *regsum*. Knowing *regsum*, the register window configuration registers needed for returning to userspace are easily determined:

$$\texttt{CANRESTORE} = regsum,$$

$$\texttt{CANSAVE} = (\texttt{NWINDOWS} - 2) - regsum,$$

$$\texttt{CLEANWINDOW} = regsum,$$

$$\texttt{OTHERWIN} = 0.$$

Under certain circumstances, however, there can be more valid windows than the CANRESTORE register can accommodate. This anomaly occurs when there is a window spill trap on the SAVE instruction at the beginning of the preemptible trap handler. In that case, one extra window is spilled to the userspace window buffer (recall figures 4.6 and 4.7). Without any provisions, the handler would put the register window configuration registers into an inconsistent state, in which:

$$\texttt{CANRESTORE} = \texttt{NWINDOWS} - 1.$$

The kernel detects such an attempt and manually[16] switches to the preceding window and sets the register window configuration registers as follows:

$$\texttt{OTHERWIN} = \texttt{CANRESTORE} = 0,$$

$$\texttt{CLEANWIN} = \texttt{CANSAVE} = \texttt{NWINDOWS} - 2.$$

Returning to Kernel

No register windows need to be explicitly filled, register window configuration registers contain proper values and the primary context register continues to map the kernel. The RESTORE followed by either the RETRY or DONE instructions is enough for the preemptible trap handler to return to the interrupted context.

Comparison with Solaris

In Solaris, there is a variation on the preemptible trap handler called sys_trap. Generally, it has a different structure compared to HelenOS, but deals with similar issues. To highlight one, let us focus on the way Solaris copes with the userspace register windows.

Register window spills of userspace windows generated in the Solaris kernel go preferably to the userspace stack. This is significantly different in HelenOS, where the kernel doesn't want to risk the DTLB miss on a userspace address and spills directly to the userspace window buffer. In Solaris, some of the spills might succeed and save the kernel from additional work needed to restore the windows from other places. However, attempts to spill to the userspace stack will fail if the stack is not mapped by the DTLB. Solaris deploys a mechanism which detects this and falls back to spill to Solaris' incarnation of the userspace window buffer—wbuf.

[16] As opposed to via the RESTORE instruction. RESTORE is not executed in this case.

Because not all userspace windows go to `wbuf`, Solaris has harder times housekeeping the state of the buffer. For instance, it has to remember stack pointers of windows actually stored in the buffer.

When returning to userspace, Solaris again considerably diverges from HelenOS. HelenOS can benefit from not-yet-spilled userspace windows and loads the content of the userspace window buffer directly into the register window set. On the other hand, Solaris makes sure all userspace windows are either on the userspace stack or in `wbuf`. Knowing stack pointers of all buffered windows, it copies those windows directly to the userspace stack. Once a window is on the stack, standard register window fill traps can work with it.

As a result, Solaris might have to deal with a smaller buffer for userspace register windows than HelenOS. When it comes to restoring windows from the buffer, HelenOS is more straightforward and refills register windows faster than Solaris.

4.3.6 Timer Support

The UltraSPARC and UltraSPARC II processors have one source of time integrated directly on-chip as a couple of registers: `TICK` and `TICK_Compare`. The former register is incremented on a timely basis with a known relation to the processor speed. The relation is found in each `cpu` node of the OpenFirmware device tree canonical copy in the `clock-frequency` property. Its value determines the increase of the `TICK` register in a second. The `TICK_Compare` register contains the match value. When `TICK` reaches the value in `TICK_Compare`, a level 14 interrupt is generated and the kernel notices the event.

Advantage of this implementation is that the kernel is self-sufficient and does not need to look for another time signal. The disadvantages are only theoretical. Should the processor support frequency scaling, the `TICK` register would become unfeasible for the task.

As for the implementation, the `TICK` register is initialized to 0 and never reinitialized so it keeps monotonely growing. On the other hand, the `TICK_Compare` register is periodically reprogrammed every time the tick interrupt is processed. With the speed of the UltraSPARC II processors, the `TICK` register would take more than 800 years before it overflows. In case of a theoretical 2GHz UltraSPARC processor, the overflow would take more than 100 years to happen. The overflow condition is therefore not addressed by the kernel. Due to `TICK` growing monotonely, the register can be used for measuring performance.

4.3.7 Handling I/O Devices

Owing to its micro architecture, the SPARTAN kernel itself doesn't do a lot of device handling. It does, however, handle several devices such as the keyboard and the framebuffer. Unlike on ia32 machines, these devices are not standardized, so there is nothing like the VESA framebuffer or an i8042 compatible keyboard controller on each sparc64 box. Moreover, there is no single interrupt controller like the i8259 (i.e. PIC) or APIC. Instead, devices within an OpenFirmware device subtree use their own dedicated interrupt controller which typically corresponds to a predecessor node of the subtree.

Memory Mapped Devices

Framebuffers, keyboard controllers and bus controllers are the three types of memory mapped devices that the kernel needs to support. The `/aliases` node of the OpenFirmware device tree provides the `screen` and `keyboard` properties that contain absolute tree paths for the respective devices. Table 4.4 shows console aliases used on the Enterprise machine.

Alias	Path
screen	/pci@4,2000/SUNW,m64B@2
keyboard	/central/fhc/zs@0,904000

Table 4.4: Console aliases found on the Enterprise machine.

Register addresses reside in the `reg` property of the device. Besides addresses, the property also includes sizes of each memory mapped register area. There are two problems with this hierarchical scheme. The first is that the `reg` property is, in fact, an array of several register addresses and sizes. The second one is that addresses in the `reg` property are relative to the beginning of the parent node's address space—they are not necessarily system bus addresses.

The first problem requires that there are small device drivers for all devices that the kernel intends to support. The device driver must be aware of the device's OpenFirmware binding in order to map the registers to correct array indices. The driver also has to know what bus the device can attach to.

The second shortcoming is fixed algorithmically by walking the OpenFirmware device tree from the device node up to the root node. During the walk, the register address associated with the child node is relative to the parent's address space. The goal is to transform that address into an address from the root node address space, which makes it meaningful to the system bus. The mapping for one step is embedded in bus node's `ranges` property, which is also an array—this time an array of ranges of addresses. Along the path to the root node, the kernel applies the `ranges` property of the bus node to an entry in the child `reg` property format, transforming it to the `reg` property format of the bus node. The path usually involves several types of buses with incompatible formats and sizes of `reg` and `ranges` properties. A bus-specific transformation function must be therefore implemented for each of them. Currently, HelenOS supports the following buses:

- FHC Central,
- FHC,
- PCI,
- EBus.

Example of Applying the `ranges` Property

Suppose that the kernel is trying to determine the physical address of the Zilog 8530 keyboard controller. By investigating properties of the `zs` node shown in table 4.5, the kernel knows that

the registers of the device are located at address 0x904000 relative to the parent node address space. Moreover, the registers are only 8 bytes long.

Property	Value
keyboard	
interrupts	00000039
reg	00000000 00904000 00000008
name	zs

Table 4.5: Selected properties of the `zs` node.

In the next step, the kernel ascends to the parent of the `zs` node, which is the `fhc` node. Table 4.5 lists important properties of that node. The `ranges` property has only one record; the record matches the `reg` property of `zs`[17]. By adding 0x904000 and 0xf80000000, the kernel gets 0xf8904000, which is the address of `zs`'s registers relative to `fhc`'s parent.

Property	Value
ranges	00000000 00000000 00000000 f8000000 08000000
name	fhc

Table 4.6: Selected properties of the `fhc` node.

The parent of the `fhc` node is the `central` node. Its properties are listed in table 4.7. Also the `central` node has only one record in the `ranges` property. By inspecting the record, the kernel comes to the conclusion that address 0xf8904000 fits in the interval determined by the first two numbers (i.e. child address space base) and the last number (i.e. size). It therefore uses the third and the fourth number (parent address space base), 0x1fff8000000, and adds it with 0xf8904000. The child address space base in the `ranges` property is already offset by 0xf8000000, so it has to subtract that value from the result. Finally, due to the `central` node being the child of the OpenFirmware device tree's root node, no more ranges need to be applied and 0x1fff8904000 is the system-wide physical address of the memory mapped registers of the Zilog 8530 controller.

Property	Value
ranges	00000000 f8000000 000001ff f8000000 08000000
name	central

Table 4.7: Selected properties of the `central` node.

[17]The first pair of numbers represents the base address in the child address space, the second pair of numbers represents the base address in the parent address space and the last number is the size of the area accessible from the parent space.

Mapping Interrupts

When a device generates an interrupt, its respective OpenFirmware device tree node contains a record in the `interrupts` property. Similarly to the `reg` property, entries of the `interrupts` property are not absolute values. In order to map the device interrupt number to the system wide interrupt vector, the kernel needs to do another device tree traversal. Moreover, the traversal yields a device-specific interrupt controller node[18]. The controller node contains the `IMAP` and `ICLR` registers for each interrupt source. The interrupt is enabled and initialized in the former and cleared in the latter register. The device interrupt must be enabled both in the device and in its interrupt controller.

During the interrupt mapping traversal, the child node passes a `reg` entry and an interrupt number to its parent node. The parent node, depending on the underlying bus, might contain a `interrupt-map-mask` and `interrupt-map` properties. Buses that don't have these properties (e.g. FHC) follow some wired-in behaviour. For instance, the FHC bus functions as the interrupt controller for its devices itself and does not translate the interrupt number. On the other hand, the EBus uses both of these properties. The `reg` entry fields and the interrupt number are logically multiplied with the bit mask stored in the fields of the `interrupt-map-mask` property. The results of the logical multiplication are then compared with the entries of `interrupt-map`, which is in an analogous relation of the `reg` property to the `ranges` property. If there is a hit, the matching entry contains the OpenFirmware handle of the device tree node functioning as the interrupt controller. In case of the forementioned EBus, the interrupt controller is one of several possible PCI controllers. Once the interrupt controller is known, a new translation step starts in that node. The list of supported interrupt controllers follows:

- Sabre PCI controller,
- Psycho PCI controller,
- FHC.

Summary of Hardware Support

The Sabre PCI controller is found on UltraSPARC IIi based systems such as the Ultra 5 machine. The Psycho PCI controller is used in UltraSPARC II machines such as Ultra 60. The FHC[19] is, as of now, Sun Microsystem's proprietary hardware without publicly available documentation[20]. It is used in SunFire servers like the Enterprise E6500.

The combination of device specific drivers and the generic device tree traversing algorithm allowed the implementation to support several different framebuffers, keyboard controllers and interrupt controllers (see table 4.1).

Nevertheless, the support is sometimes problematic as there are known insufficiencies. In the case of framebuffers, color depths of 16bpp and better are generally recommended because the

[18] Either explicitly or implicitly.

[19] FireHose Controller.

[20] HelenOS support for FHC is a result of reverse engineering the debug trace output of the Simics simulator when running the Linux kernel on the simulated SunFire server.

respective drivers are not capable of 8bpp color palette initialization. The Ultra 5's framebuffer displays wrong colors even when in 24bpp color mode. The author believes this could be fixed by providing a more sophisticated device driver for the device. The other two framebuffers work as expected in the 24bpp regime.

The interrupt mapping is supported on three interrupt controllers, but the keyboard interrupt is being generated only on the Enterprise E6500 machine, which has the FHC and the Zilog 8530 keyboard controller. The other two machines with PCI interrupt controllers and UART 16550 keyboard controllers use polling to read the keyboard.

4.3.8 New IRQ Dispatcher

A major flaw in the design of the HelenOS IRQ dispatching mechanism preventing the kernel from handling shared interrupts was discovered during the implementation phase of this thesis. The kernel IRQ registration API didn't allow more than one handler per interrupt number. Furthermore, the userspace IRQ registration API was oriented on interrupt numbers instead of device numbers. It therefore suffered the same problem as the kernel. For this reason, the flaw had to be fixed in both places—fixing only the kernel would not help. The notion of a unique device number assigned by the kernel did not exist at all.

A new generic mechanism was developed and completely replaced the old one. The new scheme deploys a hash table of IRQ structures. Each IRQ structure contains a possibly ambiguous interrupt number and a unique device number. There are also two virtual functions in each structure: claim function and the handler. For the sake of userspace IRQ notifications, the IRQ structure also embeds notification configuration data.

Upon registration, a filled IRQ structure is passed to `irq_register()`. The function hashes on the interrupt number and inserts the structure into the respective bucket.

When a hardware interrupt comes, the interrupt trap handler calls the `irq_dispatch_and_lock()` routine and passes it the interrupt number. The routine then hashes on the interrupt number to locate the only possible bucket. The bucket is then searched for exact match on the interrupt number and true return value of the claim function. The claim function typically polls the device in order to find out whether the interrupt was generated by it or another device. When the match is found, the handler function is called. In case there is no-one to claim the interrupt, the interrupt is spurious and a warning message is printed.

The `SYS_IPC_REGISTER_IRQ` and `SYS_IPC_UNREGISTER_IRQ` syscalls use the `irq_find_and_lock()` routine for searching the hash collision chains. In this case, an interrupt number and a device number are passed to the routine, which in turn hashes on these values. The device number must be assigned to the device by a call to `device_assign_devno()`. Kernel assigns device numbers by incrementing a counter and returning its previous value. Note the fundamental difference between the old and the new behaviour for userspace tasks. With the new API, the task receives IPC notifications about interrupts generated by specific devices; with the old API, this semantics was not supported.

4.3.9 Interrupt Priorities

Although the architecture supports 15 prioritized interrupt levels, these are not used by the kernel except for the tick interrupt[21]. This behaviour is in line with all the other ports. In HelenOS, all interrupts are either enabled or disabled in the processor—in `PSTATE.IE` in case of the SPARC V9 architecture. The `PIL` register is initialized to zero and stays like that during the whole run of the operating system. Should the kernel support level-interrupts in the future, the `PIL` register will have to be preserved by the preemptible trap handler.

4.3.10 Scheduler Hooks

The generic scheduler code is instrumented with several architecture specific hooks. On the sparc64 architecture, these hooks are used to pin important pieces of the kernel address space in the data TLB and to exchange vital pointers between the kernel data structures (i.e. the address space structure and the thread structure) and hardware registers.

Thread Hooks

Before a thread is scheduled, the `before_thread_runs_arch()` hook fires. If the thread runs also in userspace[22], the hook reads the address of the kernel stack from the thread structure and writes it into both the alternate and interrupt version of the `g6` register. It also fills the alternate `g7` with the address of the top of the userspace window buffer.

After a thread is preempted or calls `scheduler()` explicitly, the `after_thread_ran_arch()` hook is called. The hook samples the current top of the userspace window buffer back to the thread structure.

Note that the userspace window buffer pointer is only copied to and from the alternate `g7` register. The attentive reader might wonder why this is sufficient. The reasoning is noteworthy. Register windows can be written to the userspace window buffer only from a spill trap, but all spill traps start executing in the alternate global register set. Similarly, register windows are read from the userspace window buffer only in the preemptible trap handler, especially in its epilogue. However, as described in 4.3.5, the middle part of the handler switches to the alternate global set and never leaves it again. Finally, since there is only one register holding the pointer, the kernel doesn't have to guess what register contains the valid copy.

Address Space Hooks

When threads from different address spaces are scheduled, the scheduler calls out `as_deinstall_arch()` to clean up after the old address space and `as_install_arch()` to prepare environment for the new address space. On sparc64, the latter function writes the assigned ASID to the secondary context register. If applicable[23] and necessary[24], the page with the

[21] The hardware always generates the tick interrupt as a level 14 interrupt

[22] It has userspace context.

[23] TSB support is compiled in.

[24] TSB is not mapped by the 4M locked DTLB entry.

TSB is locked in the DTLB. This is essential for avoiding nested MMU traps, which would lead to irrecoverable loss of the content of the memory management set of global registers. In the spirit of similar hooks, the former routine demaps the previously locked entry from the DTLB.

4.4 Memory Management

In general, there is less architecture specific code related to memory management because the sparc64 port uses the generic mechanisms as much as possible. However, it does use sparc64 specific low-level code in order to overcome certain bottlenecks of the generic code. There are also several low-level parts that must be written in the assembly code.

4.4.1 Address Spaces

The memory context identifiers are 13 bits wide on SPARC so one memory context identifier can fit into a 16-bit short integer. In total, all available memory context identifiers cram into a 16K large dynamically allocated array and are therefore ideally handled by the generic ASID[25] FIFO allocator. The allocator keeps the inactive memory context identifiers in an in-array FIFO. The FIFO makes the allocation and deallocation work bound by a constant time. Besides sparc64, the same mechanism is used by other architectures that have hardware support for address space IDs. The ASID allocation strategy is in its entirety best described in [1].

4.4.2 Virtual Address Translation

If not enforced by hardware, HelenOS ports can basically choose between two generic virtual address translation mechanisms: the hierarchical 4-level page tables and the global page hash table. The sparc64 port makes use of the page hash table mechanism, which is the same as the ia64 port. Architecture specific functions, `itlb_pte_copy()`, `dtlb_pte_copy()`, `itsb_pte_copy()` and `dtsb_pte_copy()` take care of propagating the software PTE content to both TLBs and TSBs, respectively. As with the memory context identifier allocator, both schemes are thoroughly described in [1].

To get a picture of how the translation works, let us now follow the processing of a DTLB miss[26]. When the processor references memory which is not mapped by the DTLB, the kernel starts servicing the `fast_data_access_MMU_miss` trap. If the trap is not a kernel memory miss, it cannot be resolved by a simple identity mapping function. If it also cannot be solved by consulting the DTSB (see 4.4.3), the mapping has to be looked up in the global page hash table.

When the mapping is not found even in the global page hash table, the very high-level address space handler is called to resolve the page fault. Otherwise the mapping is found and gets inserted into the DTLB and, possibly, into the DTSB as well.

The hardware doesn't support *accessed* and *modified* bits in the TLB and TSB translation entry (TTE) format. On the other hand, TTEs contain the *writable* bit and the kernel uses it to emulate

[25] Address Space ID—not to be confused with SPARC V9 ASI's.

[26] The ITLB miss processing is very similar.

the *modified* bit in software page table entries (PTE). Whenever a writable mapping is inserted, the kernel deliberately inserts the mapping without the writable bit set. When the processor later attempts to write to that page, the `fast_data_access_protection` trap is generated. The kernel then checks whether the page should be writable by looking into the respective PTE. If the PTE signals a writable mapping, the TTE is updated to writable and the dirty bit in PTE is set. The *accessed* bit software emulation is more straightforward. The kernel sets this bit in the PTE on every MMU trap related to the respective TTE. Despite all this support, HelenOS is not yet ready to make use of this information[27] so neither of the bits is ever cleared.

On sparc64, the kernel implements non-executable pages by strictly enforcing that the ITLB can be refilled only from PTEs that have the *executable* bit set.

Global Page Hash Table Scalability

When the global page hash table mechanism was developed for the ia64 port, its scalability bottlenecks were not developers' first priority. The main purpose for adding a new mechanism to the already existing generic 4-level hierarchical page tables was to evolve the virtual address translation API to the state in which the underlying mechanism is irrelevant[28].

It is deemed to be in the very nature of the global page table to be a potential performance bottleneck: it is a shared global structure and as such must be protected by a mutex. The mutex serializes every access to it, both searches and updates, thus reducing concurrency in one of the most critical and hot paths in the system. One of the options of how to make the algorithm scale better, would be to use a synchronization primitive that does not harm concurrency as much as the mutex. Reader/writer locks come automatically to mind. Nevertheless, three problems discourage this simple solution. The first is that the reader/writer locks have bigger overhead than mutexes. The second is that the virtual address translation API does not distinguish between read-only and write accesses. And the third is that the reader/writer lock would still serialize access to the page hash table to some extent. Alternatively, the global page hash table could be split into several hash tables so that each address space would have its own private one—much like the hierarchical page table mechanism.

Another possible solution of the page hash table lock bottleneck, the solution used in HelenOS, is obvious since it is supported directly by the hardware. Using the translation storage buffer, which is explained in next subsection, the operating system can significantly speed up virtual address translation lookups and still use the global page hash table.

4.4.3 Translation Storage Buffer

The Translation Storage Buffers, commonly abbreviated to TSBs, can be viewed as optional software extensions of the TLBs. While there are only 64 entries in each TLB, the number of TSB entries can be of several times higher magnitude. TLBs and TSBs use the same format of entries and with a little assistance from the hardware, the TLBs can be refilled from TSBs in about ten instructions.

[27] Swap is not supported.

[28] In fact, other mechanisms could be easily added.

Although several TSB configurations are supported by hardware, HelenOS can only be configured and built to deploy 32K instruction and 32K data TSBs. The data TSB physically follows the instruction TSB in one 64K page[29]. One TSB entry is 16 bytes large so there are 2048 entries in each TSB. This is a fair-sized number, considering that each address space has its own pair of TSBs.

The TSBs are allocated upon creation of each address space. In order to prevent nested TLB misses, the kernel locks the TSB page mapping in the DTLB every time the address space becomes active on a processor and removes the mapping every time it gets deinstalled from the processor. For the sake of simplicity, both TSBs are also allocated for the kernel, even though it doesn't use them.

Resolving TLB misses from TSB

In case of the `fast_instruction_access_MMU_miss`, the trap handler reads the ITSB[30] Tag Target and ITSB 8K Pointer registers. The tag register contains the virtual page number and memory context identifier. This information will later be used to distinguish between a ITSB miss or hit. The pointer register is precalculated by the processor and is offset from the ITSB Base register which was previously set during installation of the address space; it points inside the ITSB at the only position which can contain the respective TTE.

Afterwards, the handler atomically reads the entire 16-byte TTE entry from the ITSB into a pair of registers. The register holding the TTE tag is compared with the ITSB tag register. If the registers match, the ITSB contains the entry that was missing in the ITLB. The other register contains the TTE data and is ready to be inserted into the ITLB using one `STXA` instruction. If the registers don't match, the ITSB does not contain the right entry and a higher level TLB miss routine must be invoked. If the higher level routine succeeds, it automatically refills the ITSB with the respective entry so that the miss will not occur next time.

The `fast_data_access_MMU_miss` case is almost identical to the ITSB related code of the ITLB miss except that additional checks are required. The kernel memory is handled by the mechanism described in section 4.4.4 so the kernel itself must be excluded from the DTSB[31] handling. The DTSB related code within the DTLB miss handler checks the memory context identifier read from the DTSB Tag register and if it reads as zero, further DTSB processing is skipped. Otherwise the DTLB miss proceeds comparably to an ITLB miss.

TSB Synchronization

The TSB design is remarkable in that it requires no locking for synchronization of readers (i.e. assembly language parts of TSB miss handlers). Due to this reason, the TSB mechanism is a perfect complement to the global page hash table and a possible solution to its forementioned bottleneck. Writes to the TSBs are serialized by the address space mutex—a lock with still finer granularity than the global page hash table mutex.

[29] Alternatively, HelenOS can work completely without TSB support.
[30] Instruction TSB.
[31] Data TSB.

Recall that each TSB-enabled TLB miss handler reads the entire TTE atomically. The UltraSPARC processors have a dedicated instruction[32] and an ASI for this purpose.

A valid TTE can be recognized by inspecting the three most significant bits of the TTE tag. The TSB Tag Target register has these bits hardwired to zero. In order to invalidate a TSB entry, it is therefore sufficient to set any of these bits to one.

Updating a TSB entry is done in C, using a special protocol. First, the entry is invalidated. A combined write memory and a compiler reorder barrier is then issued to prevent any code reordering. Second, the in-TSB TTE is updated, followed by another combined write memory and compiler reorder barrier. Finally, the most significant bit is cleared and the TTE becomes valid again. The reader is guaranteed to observe either an invalid entry or the updated and valid entry.

4.4.4 Kernel Memory

The kernel maintains an identity mapping for itself so that it doesn't need to look at the page tables, unlike the other address spaces. More precisely, each virtual memory page maps to the physical memory frame with the same address. On systems where the physical memory starts at a non-zero address, the transformation function includes a constant for correction. The `KA2PA()` function that translates a kernel address to a physical address is therefore defined as follows:

$$\texttt{KA2PA}(ka) = ka + C,$$

where ka is the virtual kernel address and C is the address of the beginning of physical memory.

The kernel organizes its virtual memory in this way in order to be able to resolve kernel page faults without going to the higher level TLB miss handler. On a kernel DTLB miss, the trap table resident TLB miss handler notices that the associated memory context belongs to the kernel and instead of redirecting to the C language `fast_data_access_mmu_miss()` routine through the preemptible trap handler, it merely applies the `KA2PA()` transformation on the missing page address and immediately inserts the resulting mapping back to the DTLB. The virtue of not going to the higher level TLB miss handler is essential because it avoids nested kernel TLB misses during the global page hash table searches. Having kernel mappings in the global page hash table would lead to deadlocks on nested faults caused by accesses to the global page hash table.

Memory Mapped Devices

One exception to the identically mapped kernel memory is the mapping of memory mapped devices' registers. When a memory mapped device is initialized, the kernel allocates unused pages after the end of physical memory and maps that virtual region to device's registers. For this reason, the memory mapped devices are excluded from the identity mapping. The kernel takes care of this singularity by locking the device's memory mapped registers directly into the DTLB so that accesses to them will never cause a DTLB miss. For a small amount of devices,

[32] `LDDA`.

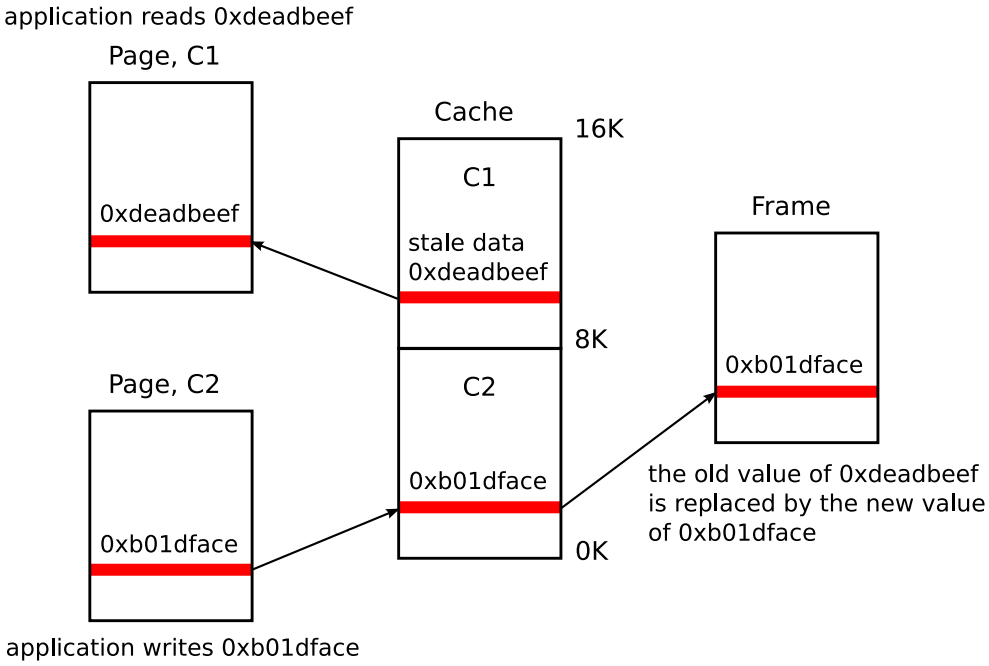


Figure 4.9: Application reading stale data from the D-cache.

this is acceptable as there are 64 entries in the DTLB. HelenOS currently uses three or four DTLB entries for this: one or two for the framebuffer, one for the keyboard registers, and one is also used for each interrupt controller. Should there be more devices accessed by the kernel in the future, another mechanism will have to be adopted.

4.4.5 Data Cache

The UltraSPARC processors have a direct-mapped, virtually indexed and physically tagged L1 write-through data cache. The cache is twice as big as the minimal page size (i.e. 16K of cache vs. 8K of virtual page), which, in combination with the forementioned characteristics, causes problems to the operating system. The problem here is that virtually aliased data can end up in different cachelines, depending on the virtual color of the respective pages. The virtual color is determined by the address bits up to the cache size boundary. Bits between the page and cache size boundaries define the virtual page color. On UltraSPARC, only one bit, bit 13, participates in forming the page color. Using the color terminology, a 32 byte long naturally aligned block of physical memory will take up two cachelines, as opposed to one, if there are at least two virtual pages of different color that map the block. Such an address alias is illegal and must be either avoided or maintained by the kernel.

Figure 4.9 depicts a situation in which a piece of physical memory was first written with the value of 0xdeadbeef using an address with virtual color *C1*. This initial write filled the respective

color *C1* cacheline with the new value. Sometime later, the same piece of physical memory was written from an address with virtual color *C2*. This time the value written was 0xb01dface and the write filled the color *C2* cacheline. At this time, the cache contains an illegal virtual alias. Note that the content of the color *C1* cacheline differs from the real content of memory. Anyone reading that location from a *C1* color virtual address will read stale data.

Over the course of the implementation phase, the author came to the conclusion that it pays off to address the virtual aliasing problem at its very origin rather than trying to fix the caches by preventively flushing them or disabling cacheability of pages. In order to recapitulate, the origin of the problem is that the virtually indexed cache is twice as big as the standard page size used by the system. It would therefore be sufficient if the smallest page size was at least as big as the cache and was a whole multiple of the cache size. This combined with the natural alignment of page virtual addresses would guarantee that illegal virtual aliases do not occur.

Skipping the base 8K page size, the next page size supported by the MMU which is bigger than the 16K D-cache is 64K. Picking this page size for the task of avoiding illegal virtual aliases would reduce the TLB refill rate approximately to one eighth. On the other hand, more memory would be wasted because the granularity of physical frame allocations would be also reduced to one eighth.

Instead, the author implemented a solution in which the page size is increased to 16K for the generic parts of the kernel (e.g. frame allocator, global page hash table and address space code) and the entire userspace. Since 16K is not a page size supported by the MMU, it is internally emulated by a pair of adjacent 8K subpages on the TLB and the TSB level. When a TLB or a TSB miss occurs, the kernel refills the TLB or the TSB, respectively, with the mapping for the 8K subpage which triggered the miss[33]. Similarly, write protection misses on one of the two 8K subpages will result in the kernel updating the *dirty* bit of the corresponding emulated 16K page's PTE, which resides in the global page hash table; updating the *accessed* bit is analogous.

4.5 Userspace Support

4.5.1 Syscalls

Syscalls are implemented with the aid of the `TA`[34] instruction. Given an immediate operand, the instruction unconditionally generates the `trap_instruction` trap with a trap type of 0x100 plus the number stored in the operand. The operand is 7 bits wide, but HelenOS needs only 5 bits as there are only 29 syscalls.

On the userspace side, the standard library provides the usual syscall API for user tasks. Nevertheless, the implementation of the sparc64 version of the `__syscall()` routine differs from other architectures in that the function is defined inline. Due to this fact, the syscall can, in an ideal case, translate into a single instruction and in an average case into a few more instructions.

On the kernel side, there are 29 traps[35] that branch into the respective higher level syscall

[33] Provided that the TLB or the TSB, respectively, can be refilled with a valid mapping at all.

[34] Trap Always.

[35] And 3 currently unused.

routines. The trap taken depends on the operand of the `TA` instruction. Owing to the syscall number being passed in the operand and being encoded in the trap taken, it doesn't have to be passed in a register.

Preemptible Trap Handler Modifications

The preemptible trap handler, as described in 4.3.5, differentiates between non-syscall traps and syscalls. In case of syscalls, the handler can be optimized because some tests can be evaluated during compile time. Tests that are always false for syscalls and code guarded by these tests are simply left out. To illustrate this, the trap always comes from userspace[36] and the anomaly of the shifted `CWP` doesn't occur[37]. Similarly, the syscall has to always switch to the kernel stack. One of the most important changes is the method of returning from the trap. The non-syscall version of the handler uses the `RETRY` instruction. The syscall version needs to skip the trapping instruction so it always returns via the `DONE` instruction.

Besides dead code elimination and a different return path from the trap, the syscall also adds a few extra lines to the preemptible trap handler. These lines take care of copying syscall arguments in `o0`, `o1`, `o2` and `o3` to the new working register window[38] and also of copying the syscall return value from the working register window back to the `o0` register of the register window in which the trap occurred.

Even though the handling of register windows with userspace content has already been described in 4.3.5, it can be considered an integral part of the userspace support.

4.5.2 Pseudo Thread Support

As in other architectures, sparc64 userspace pseudo threads need a certain amount of low-level support from the standard library. In order to facilitate cooperative switching among the pseudo threads, userspace functionality equivalent to the kernel's `context_save()` and `context_restore()` is desired. Fortunately, the switching functions could have been ported almost verbatim from the kernel. The only difference is that instead of the processor's interrupt priority level, the userspace version stores the TLS[39] pointer.

The kernel preserves the processor interrupt priority level, along with the whole `PSTATE` register, in the context structure in order to support calls to `waitq_sleep()` when interrupts are both enabled and disabled. Since interrupts are always enabled for user tasks and because that functionality is purely kernel specific, there is no sense in storing the interrupt priority level in the userspace pseudo thread context.

[36] The syscall handler need not check it.
[37] The syscall always traps at TL 1.
[38] Recall how this window is allocated in 4.3.5.
[39] Thread Local Storage.

Thread Local Storage

On the other hand, HelenOS supports thread local storage so the spared space in the context structure is used for the thread-private data pointer. The pointer is, according to the TLS[2] specification, stored in the `g7` preserved register. The `context_save()` and `context_restore()` functions take care of saving and restoring the register, respectively.

The TLS model used on sparc64 is almost identical to that of the ia32 and [2] refers to this model as variant II. Both models were developed by Sun Microsystems and due to their similitude, the code that allocates and later on deallocates the TLS was taken from the ia32 port without any changes.

FPU Context Issues

In general, userspace pseudo threads have one substantial disadvantage with respect to floating-point operations: due to cooperative switching among pseudo threads running within one kernel-visible userspace thread, lazy FPU context switching doesn't take place and the pseudo thread FPU context must be preserved by other means. Nevertheless, the situation is not that bad if the architecture ABI defines the floating point registers as volatile or scratch. In the case of scratch registers, the caller only saves the scratch registers with valid content prior to a procedure call[40]. On the other hand, preserved registers should be saved and restored by the callee, which has no clue about what registers are in use. The pseudo thread context switch code would therefore have to do a full, non-lazy, FPU context switch.

Example of ports that have to do the non-lazy FPU switch are mips32 or ia64. Floating point registers of these architectures have to be preserved across function calls. Ports that are not affected by this are, for instance, amd64, ia32 and sparc64.

In the case of the sparc64 architecture, [6] specifies that the general purpose floating point registers are all scratch. Two remaining registers that compose the FPU context, `FSR` and `FPRS`, are at least partially volatile. In the case of `FPRS`, the only problematic bit is `FEF`. Nevertheless, the kernel effectively neutralizes effects of this bit so its content really doesn't matter to applications. In the case of `FSR`, there are three preserved bits: `RD`, `TEM` and `NS`, respectively. These are a source of problems due to their potential to influence FPU rounding direction, reported FPU exceptions and normality of operands, respectively. Userspace tasks that have ambitions to switch between different settings of these three bits have to take care of their proper saving and restoring. Nevertheless, this is almost a non-issue because the problem would be local to a task that doesn't obey this guidance. Moreover, applications using different settings of `FSR.RD`, `FSR.TEM` and `FSR.NS` for different pseudo threads are rather hypothetical.

4.6 SMP Support

On sparc64, the low-level mechanism of starting up application processors is hidden from the operating system behind the Sun-specific OpenFirmware service `SUNW,start-cpu`. This

[40] In this case, the procedure call represents a switch to another pseudo thread.

complicates the process of multiprocessor startup for HelenOS. Recall that the kernel refrains from using the OpenFirmware interface so the only piece of software that can start additional processors is the boot loader. Section 4.2.6 describes the process of application processor startup up to the point the processors reach kernel code.

SMP support for the sparc64 port is optional and is configured during the build process.

4.6.1 Kernel Startup

The bootstrap processor shares a remarkable portion of the kernel startup code with the application processors. The codepaths start to differ after the processors take over both TLBs and the trap table. The bootstrap processor then heads towards the `main_bsp()` C routine while the application processors enter a loop where they wait until the bootstrap processor picks them up later.

4.6.2 Application Processor Pick-up

Unlike on the ia32, where the application processors are halted until the kernel executing on the bootstrap processors wakes each one of them up by sending an IPI[41], the application processors on the sparc64 linger in a loop, actively checking the `waking_up_mid` global variable. When the bootstrap processor sets the variable to the `MID`[42] of an application processor, the respective processor notices this by observing the `waking_up_mid` becoming equal to its own `MID`. When this happens, the processor leaves the active loop, switches to the kernel stack and calls the `main_ap()` function.

The bootstrap processor looks for processors to pick up in the canonical copy of the OpenFirmware device tree. Discovered processors are unblocked, one at a time. Serialization of the process is necessary so that the starting processors don't destroy another processor's stack. Following the ia32 template, the algorithm runs in the context of the `kmp` kernel thread. After setting `waking_up_mid` to the MID of the starting processor, the thread goes to sleep in the `ap_completion_wq` wait queue until it is either woken up by the started processor or the sleep times out after the processor does not start within one second.

4.6.3 Application Processor DTLB Initialization

Shortly after an application processor starts executing higher-level kernel code, the sparc64 code comes across the MMU initialization and its architecture-specific hooks. In case of a SMP kernel, the `page_arch_init()` hook needs to synchronize the DTLB of the processor with the bootstrap processor as far as locked entries of memory mapped devices are concerned.

For this reason, the bootstrap processor creates an initialization entry for each memory mapped device in the `hw_map()` function. Each application processor then walks the array of these ini-

[41] Inter-Processor Interrupt.

[42] Unique number identifying the processor.

tialization entries and recreates the mappings found there in its local DTLB. Thus, all processors' DTLBs contain the same mappings for all memory mapped devices.

4.6.4 Spinlock Implementation

The spinlock algorithm is closely modeled after appendix J.6 of [3]. It makes use of the `CASX` atomic instruction, which has the compare-and-swap semantics. On entry to `spinlock_lock()`, the `CASX` instruction tries to atomically exchange an anticipated zero stored in the in-memory spinlock variable with a non-zero stored in a register. If the lock was not held, the instruction succeeds and the lock ownership is given to the caller of `spinlock_lock()`. If the memory location does not contain zero, the lock is already held and the operation fails. The spinlock acquisition then falls back to a non-atomic monitoring loop. In the monitoring loop, the algorithm repeatedly reads the memory location over and over again until it observes zero. It then transfers control back to the beginning of the procedure so that another atomic attempt to gain the lock can be made. Note that the read-only operation in the monitoring loop is more lightweight and cache-friendly than an atomic instruction could ever be. The read-write nature of the `CASX` instruction could lead to perpetual cacheline migration from processor to processor.

The spinlock is eventually acquired by the caller but the locking algorithm is still not done. In order to prevent instruction permeation described in [1], memory barriers need to be issued. Even though the whole kernel operates in the strongest memory model (TSO[43]), spinlocks use memory barrier protection required by the weakest memory model (RMO[44]). Thus, it is guaranteed that if the kernel switches to a weaker model, spinlock synchronization will continue to function. Another reason for using the relaxed memory model is that UltraSPARC documentation is a little unclear about what memory barriers are needed for the two stronger supported models. In RMO, all stores and loads are required to wait until the loads preceding the memory barrier are finished. At this point, it really doesn't matter whether the barrier waits for loads or stores to finish because `CASX` counts as both.

The unlock operation is a simple store of zero to the in-memory lock variable. Even here the assignment of zero to memory must be preceded by a memory barrier. This time, the store representing the assignment will not proceed until all previous loads and stores from the critical section run to completion.

4.6.5 Inter-Processor Interrupts

Inter-processor interrupts use the same mechanism as ordinary hardware interrupts. The complete mechanism is described in [5]. In case of IPIs, HelenOS stores the destination function address in the interrupt vector unit dispatch register 0. On the receiving side, the trap handler checks the data receive register 0 to distinguish between IPIs and normal interrupts, which are 11-bit numbers. Moreover, the trap handler compares the value stored in data register 0 with list of functions that can be called via an IPI and refuses to call anything which is not on the list.

[43] Total Store Order
[44] Relaxed Memory Order

Each IPI is always sent to a particular MID. A broadcast IPI, as found on the ia32 and amd64, must therefore be emulated by the kernel.

Chapter 5

Related Work

5.1 Other HelenOS Ports

The SPARTAN kernel has been previously ported to six architectures. It is therefore natural that a lot of inspiration comes directly from the kernel itself. Because of certain similarities and shared implementations, the mips32 and ia64 ports were most influential. The related parts were identified as the register stack engine of ia64 and the memory management unit of both ia64 and mips32. In addition, the ppc32 architecture shares the OpenFirmware architecture with the sparc64 architecture.

5.2 Solaris

The Solaris[20] kernel has the longest tradition of supporting SPARC family processors. It is the most mature operating system for SPARC architectures to date. Traditionally, Solaris has run on vast variety of both 32-bit and 64-bit SPARC processors. The most recent versions thereof are now supported only on 64-bit processors (i.e. UltraSPARC and models from Fujitsu). Nevertheless, the kernel still supports 32-bit applications.

5.2.1 Publications and Resources

Solaris is apparently the only kernel running on SPARC for which there exists a comprehensive reference primarily targeted on the SPARC platform[7]. The scope of the reference is, of course, exceeding the scope of this document. However, because of its large scope, it tends to be sometimes too brief on the low-level front. In 2006, an updated revision of this book has been published[8]. This second edition is targeted at features found in Solaris 10 and together with its companion book[9] tends to gravitate around the unique observability features of this operating system.

5.2.2 Implementation

Featurewise, Solaris is much more advanced than HelenOS. However, some technical approaches used in Solaris were also used in HelenOS. Let us now look at some interesting similarities and differences between HelenOS and Solaris 10.

Basic Kernel Functionality

As stated in the previous chapter, the SPARTAN kernel makes no use of the OpenFirmware and completely takes over the control of the system from it. After the kernel starts running, no OpenFirmware code is executed. On the other hand, Solaris coexists with the OpenFirmware and it is possible to temporarily halt execution of the kernel and drop to the OpenFirmware prompt. This Solaris feature is very convenient for remote networkless interaction.

Solaris uses the `sys_trap()` routine, which is somewhat similar to the preemptible handler used in HelenOS. Nevertheless, this similarity is, at best, at a surface level. The author documented some interesting differences, notably the handling of the userspace register windows, in this thesis (see 4.3.5).

Memory Management

The HelenOS low-level memory management subsytem internally uses 8K pages and emulates 16K pages for higher levels. Solaris has support for all page sizes listed in table 3.4 for sparc64.

Both Solaris and HelenOS make use of the Translation Storage Buffers (abbreviated as TSBs) but significantly differ in the implementation and the use of this hardware-assisted aid to virtual address translation. Both operating systems deploy TSBs that are private to an address space. HelenOS only uses one fixed size of the TSB for the 8K page size. Contrary to that, Solaris supports up to two TSBs per process—one for 8K pages and one for 4M pages—and can shrink or grow each TSB as the situation and process virtual memory utilization requires. Solaris TSBs are described in section 12.2.4 of [8].

On the page table level, both systems use page hash tables. HelenOS maintains only one page hash table for the whole system while Solaris maintains two: one for the kernel and one for userspace mappings. For HelenOS, page hash table entries are software PTEs that don't mirror the actual sparc64 TTE format. For Solaris, page table entries are the `hme_blk`[1] structures. Each `hme_blk` groups either up to eight native 8K TTEs or one TTE of the respective page size if the `hme_blk` maps a page bigger than 8K. Solaris has slightly finer locking policy compared to HelenOS when it comes to page hash tables. HelenOS locks the entire page hash table while Solaris implements locking on the hash table bucket level. Solaris sparc64 page tables are described in section 12.2.3 of [8].

The way HelenOS handles the kernel virtual memory via `KA2PA()` and `PA2KA()` is based on similar principles on which Solaris' `seg_kpm` segment driver works. The `seg_kpm` segment driver is mentioned in section 14.7.3 of [8].

[1] Standing for a block of HAT (Hardware Address Translation) mapping entries.

It is also worth noting that the slab allocator used in HelenOS, which is described in [1], was modelled after its implementation in Solaris (see section 11.2.3 of [8]).

Userspace Support

Section 2.8 of [8] describes the implementation of system calls in Solaris. It is also invaluable in that it mentions syscall interactions with the `sys_trap` trap handler. There are few differences between HelenOS and Solaris. The first is that Solaris supports 32-bit userspace—a non-goal for HelenOS. The second is that Solaris passes the syscall number in a register rather than as part of the `ta` instruction op-code. The third difference is that Solaris passes syscall arguments that don't fit into registers on the stack. HelenOS never has to pass syscall arguments on the stack because all its system calls were designed to either work with four[2] register arguments or to accept a pointer to a user structure with an arbitrary number of arguments.

5.3 Linux

After Solaris, Linux[21] has had the second most mature support for SPARC processors. The current kernel (as of this writing version 2.6.20) still supports 32-bit SPARC processors and JPS1 processors produced by Sun are supported. Linux was second to support Niagara.

Even though the sparc64 portion of the Linux kernel adheres to higher coding style standards than some other parts of the kernel, the author doesn't consider Linux good study material. This is due to lack of comments, which unfortunately affects even the SPARC-specific code, and the lack of any concise literature, which could fill this gap.

5.3.1 Publications and Resources

As mentioned in the previous paragraph, Linux on SPARC is missing a good compendium which is available, for example, for the Itanium port[12]. Currently, the sources of information about the implementation are mostly limited to the Linux source code.

Besides the source code, there is a summary[13] of David Miller's[3] talk describing the initial porting effort. There is also a somewhat related[14] document about porting the User Mode Linux to UltraSPARC.

Interestingly, some useful information about the architecture and the SPARC Linux port itself can be found in David Miller's web blog[16]. The author used this source to learn more about handling the `reg`, `ranges`, and `interrupts` OpenFirmware device tree properties.

Some notable entries in [16] reveal methods used by Linux when dealing with virtually indexed caches[17] and translation storage buffers[18].

Finally, there are many books that deal with the generic Linux kernel or some of its subsystems. For example, [15] thoroughly describes the Linux memory management layer and provides

[2] This limitation is given by the ia32 architecture register file size constraints.

[3] David Miller is the author and maintainer of the SPARC Linux port.

some architecture-specific information. The book is primarily focused on Linux 2.4 memory management, but includes description of early 2.6 features.

5.3.2 Implementation

For obvious reasons, Linux is also much more complex than HelenOS on sparc64. The following subsection will emphasize some interesting similarities and differences between HelenOS and Linux version 2.6.20.

Basic Kernel Functionality

Similarly to Solaris and contrary to HelenOS, Linux coexists with the OpenFirmware and supports switching to its command prompt. Having this kind of support in the kernel increases the risk of a failure due to errors in the OpenFirmware code.

Linux has its own implementation of the preemptible trap handler: the `etrap` and `rtrap` routines. Handling of the userspace windows is surprisingly very similar to that of Solaris: if possible, userspace windows are spilled to the userspace stack; otherwise they are spilled to a kernel structure called `reg_window` from where they are eventually copied back to the userspace stack. What is interesting on this trap handler is its activity on trap levels above TL 1. When the trap handler is invoked from a trap level higher than 1, it saves the state of all trap levels onto the stack. Neither HelenOS nor Solaris do this. In line with Solaris, Linux remembers TL 0 global registers by copying them onto the kernel stack. In HelenOS, this is achieved by preserving the globals in the corresponding local registers.

Memory Management

Like HelenOS, Linux supports only one page size[4], but the supported page size can be configured at compile time.

As for page tables, Linux generally, and on sparc64 specifically, uses exactly the same mechanism as HelenOS uses for some other architectures: generic 4-level hierarchical page tables. The older, 3-level, version of the Linux implementation is described in chapter 3 of [15]. In order to be able to map the entire 44-bit sparc64 virtual address space, the sparc64 Linux page tables are internally 3-level.

Much like other operating systems, Linux also uses TSBs to cache TLB-hot translations in a directly accessed memory array. In this field, Linux is somewhere between HelenOS and Solaris. It can grow each process' TSB when the demand arises, but it apparently cannot shrink it. Growing of TSBs requires some extra synchronization. The idea is outlined in [18].

Together with Solaris, Linux is an example of an operating system which tries to fight the illegal virtual aliases by preventively flushing the D-cache in places that can introduce an illegal virtual alias. For the `mmap` syscall, Linux uses a similar approach to HelenOS—it places the virtual address of the shared region at a 16K page boundary and thus avoids the illegal virtual

[4] If we don't count the Huge TLB filesystem support, which is described in [15].

alias completely. HelenOS uses an emulated 16K page size, so it is never possible for an illegal virtual alias to be created. More on the Linux solution to illegal aliases can be found in [17].

Userspace Support

The 64-bit Linux SPARC kernel supports 32-bit applications. Again, this is more than what HelenOS was designed for. When it comes to system calls, Linux uses only a few trap table entry points and in this regard appears like Solaris; the syscall number is passed in a register. This is different from HelenOS, where the system call number is part of the operand of the `ta` instruction.

5.4 BSD

NetBSD[23] and FreeBSD[22] have all ported their systems to some subset of the SPARC V9 architecture on their own. The OpenBSD[24] port of sparc64 is derived from NetBSD/sparc64. These ports support fewer processor models than Solaris and Linux. The supported processors are basically the same as those supported by HelenOS. However, OpenBSD has recently added support for UltraSPARC III and FreeBSD has begun to support the Niagara processors. OpenBSD and NetBSD also support the old 32-bit processors.

The author will not attempt to provide a more detailed comparison of HelenOS and any of the BSDs on sparc64 as he did in case of Solaris and Linux. This is not because the BSDs were not interesting for such a comparison, but because it goes beyond the scope of this thesis to compare HelenOS with all available interesting operating systems.

Nevertheless, it must be acknowledged here that the OpenBSD implementation of the TLB takeover greatly inspired the author, resulting in a direct reimplementation of the idea in HelenOS. See subsection 4.3.1.

Chapter 6

Conclusion

6.1 Achievements

The implementation meets all goals outlined in 1.2 and even goes beyond those goals. Each milestone from the list in 1.2 has a dedicated section in this thesis which discusses interesting implementation details. The text provides a comparison with other HelenOS ports and in several cases even with other operating systems.

The SPARC V9 architecture in general, and specifically UltraSPARC processors, offers wide variety of capabilities and hardware optimizations. HelenOS makes use of a reasonable number of them, but does not go as far as some other operating systems in order to get the maximum utilization out of the processor. This allowed the sparc64 port to find its own golden mean in the trade-off between features and portability. If a hardware feature is not exploited by the implementation, the gain is always in simplicity and generic code re-use.

Finally, the target architecture is very diverse. What is controlled by a single chip on the ia32, would be typically controlled by many different chips on different Sun boxes with one or more UltraSPARC family processors. Therefore, it is quite notable that three significantly different configurations are supported by the implementation[1].

6.2 Contributions

By porting HelenOS to yet another processor architecture, the author, in the first place, contributed to the HelenOS project. The port helped to get the IRQ dispatching right and also facilitated the forming of the global page hash table virtual address translation mechanism. Some new memory management techniques were tried out, which demonstrated the feasibility of their implementation also in other ports (e.g. ia64's VHPT or a TSB variation on the mips32). The port was a unique opportunity to try to solve challenging problems such as supporting the virtually indexed data cache in the unusual way of growing the page size to the D-cache size. It once again examined and confirmed an excellent potential, as mentioned by Děcký in [10], for being ported

[1] Even though there are known and documented deficiencies. See 4.3.7.

to another processor family.

The author believes that work on this thesis also contributed to academia by providing students and researchers with an improved version of HelenOS—a comprehensible and portable open source operating system.

6.3 Perspectives

The port to the UltraSPARC II processor paved the way for future development in this area. Besides UltraSPARC II and UltraSPARC IIi processors, HelenOS could be made to also run on newer Sun Blade machines with UltraSPARC IIe processors. The UltraSPARC IIe is very similar to the UltraSPARC IIi and is most likely supported even now. However, in order to get full support, the OpenFirmware device tree probing would need to be adapted and device drivers for newer Blade hardware added.

Support for JPS1 processors (i.e. processors from UltraSPARC III to UltraSPARC VI+) would require additional work due to substantial differences in architecture. This is even more true for the newest UltraSPARC T1 processor, which was introduced during the work on this thesis.

Interesting new horizons open for the sparc64 port in connection with the planned development of the filesystem subsystem and the networking stack for the whole project. To make this possible, the HelenOS device driver architecture will probably have to be extended and the sparc64 port will have to reflect those changes. In this regard, it is essential that a community continues to build around the project and takes over some of the tasks.

Bibliography

[1] HelenOS 0.2.0 design documentation.

[2] Drepper U.: ELF Handling For Thread-Local Storage, 2005.

[3] The SPARC Architecture Manual, Version 9.

[4] SPARC Joint Programming Specification (JPS1): Commonality.

[5] UltraSPARC User's Manual, UltraSPARC-I, UltraSPARC-II.

[6] SPARC COMPLIANCE DEFINITION 2.4.

[7] McDougall R., Mauro J.: Solaris Internals: Core Kernel Architecture, Prentice Hall, 2000.

[8] McDougall R., Mauro J.: Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture, Prentice Hall, 2006.

[9] McDougall R., Mauro J., Gregg B.: Solaris Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris, Prentice Hall, 2006.

[10] Děcký M.: Mechanismy virtualizace běhu operačních systémů, 2006.

[11] Intel Itanium Architecture Software Developer's Manual, Volume 2: System Architecture, 2002.

[12] Mosberger D., Eranian S.: IA-64 Linux Kernel: Design and Implementation, Prentice Hall, 2002.

[13] Galligher G.: The SPARC Port of Linux (summary), USENIX Annual Technical Conference Anaheim, California, 1997.

[14] Katta S.: Porting User Mode Linux to Ultrasparc Architecture, http://www.csee.wvu.edu/ katta/uml/.

[15] Gorman M.: Understanding the Linux Virtual Memory Management, Prentice Hall PTR, 2004.

[16] Miller D.: DaveM's Linux Networking BLOG, http://vger.kernel.org/˜davem/cgi-bin/blog.cgi.

[17] Miller D.: How to find a bug..., http://vger.kernel.org/˜davem/cgi-bin/blog.cgi/2006/06/09#bug_adventure.

[18] Miller D.: Dynamic TSB sizing, http://vger.kernel.org/˜davem/cgi-bin/blog.cgi/2006/03/17#dynamic_tsb.

[19] HelenOS project, http://www.helenos.eu.

[20] OpenSolaris, http://www.opensolaris.org.

[21] Linux, http://www.kernel.org.

[22] FreeBSD, http://www.freebsd.org.

[23] NetBSD, http://www.netbsd.net.

[24] OpenBSD, http://www.openbsd.org.