Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE

Pavel Římský

Podpora procesorů UltraSPARC III, IV, T1 a T2 v HelenOS

Katedra softwarového inženýrství

Vedoucí diplomové práce: Mgr. Martin Děcký

Studijní program: Informatika, Softwarové systémy

I would like to thank my supervisor Martin Děcký for reading the text of the thesis, for his suggestions, for negotiating the purchase of the Niagara-based machine by MFF UK and for administering the machine. I would like to thank Jakub Jermář for supervising the thesis from the very beginning, for lots of useful suggestions and for carefully reading the text of the thesis. I would like to thank the Prague globalization division of Sun Microsystems, namely Petr Štembera and Pavel Heimlich, for lending the SunBlade 1500 machine. I would like to thank Tomáš Hrubý for testing the Niagara port of HelenOS on his machine, which helped in overcoming a problem with starting application processors on Niagara.

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

V Praze dne   Pavel Římský

Contents

List of Figures

List of Tables

Název práce: Podpora procesorů UltraSPARC III, IV, T1 a T2 v HelenOS
Autor: Pavel Římský
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí diplomové práce: Mgr. Martin Děcký
e-mail vedoucího: Martin.Decky@mff.cuni.cz
Abstrakt: Stručné představení operačního systému HelenOS. Přehled obecných vlastností 64-bitových procesorů SPARC (registry, trapy, jednotka správy paměti, OpenBoot PROM). Specifické vlastnosti procesorů vyhovujících JPS (Joint Programming Specification). Specifické vlastnosti procesorů Niagara a příbuzných procesorů (virtualizace, hypervisor, správa paměti). Představení původního portu systému HelenOS na procesory SPARC (bootování, preemptibilní handler trapů, správa paměti). Popis portu systému HelenOS na procesory vyhovující JPS (bootování, anomálie OBP, úpravy ve správci paměti, detekce konkrétního modelu procesoru, podpora pro dvoujádrové procesory, ovladač konzole na stroji Serengeti). Popis portu systému HelenOS na procesory Niagara (integrace s dalšími porty systému HelenOS na procesory SPARC, úpravy bootloaderu, převzetí TLB, volání hypervisoru, zpracování machine description, správa paměti, úpravy preemptibilního handleru trapů, ovladač vstupu a výstupu, podpora pro více procesorů, optimalizace plánovače). Přehled existujících portů systému HelenOS. Srovnání s původním portem systému HelenOS na procesory SPARC, srovnání s Linuxem a Solarisem.
Klíčová slova: operační systém, jádro, HelenOS, SPARC

Title: Support for UltraSPARC III, IV, T1 and T2 processors in HelenOS
Author: Pavel Římský
Department: Department of Software Engineering
Supervisor: Mgr. Martin Děcký
Supervisor's e-mail address: Martin.Decky@mff.cuni.cz
Abstract: Overview of the HelenOS operating system. General overview of the 64-bit SPARC processors (registers, traps, memory management unit, OpenBoot PROM). Overview of the specific features of the processors compliant with JPS (Joint Programming Specification). Overview of the specific features of the Niagara-based processors (virtualization, hypervisor, memory management). Overview of the original SPARC port of HelenOS (boot phase, preemptible trap handler, memory management). Description of the port of HelenOS onto the JPS-compliant processors (booting, OBP anomalies, modification of the memory management, detecting particular processor model, adding support for dual-core processors, driver of the Serengeti machine console). Description of the port of HelenOS onto the Niagara-based processors (integration with other SPARC HelenOS ports, bootloader modifications, taking over the TLB, hypercalls, parsing the machine description, memory management, preemptible trap handler modifications, input and output driver, multiprocessing, scheduler optimization). Overview of the existing HelenOS ports. Comparison with the original SPARC HelenOS port, comparison with Linux and Solaris.
Keywords: operating system, kernel, HelenOS, SPARC

Chapter 1

Introduction

1.1 Motivation

The world of today's computers is very diverse, spanning from small embedded devices to large servers. In the server machines, the maximum throughput and parallelism plays the most important role; such machines often contain tens of processors. In the past few years the producers have put an emphasis onto lowering the energy consumption, which lead to development of multicore processors where multiple central processing units are integrated in a single processor chip. Moreover, multithreaded processors have been introduced, which are able to execute multiple threads per one processor core in parallel, just by regularly switching among active register sets.

An interesting way how to investigate the features of the processors found in modern server machines is trying to add support for some selected family of those processors to some existing operating system which has not supported these processors so far.

This thesis describes the process of adding support for the newer 64-bit SPARC processors to the HelenOS operating system.

1.2 Goals

HelenOS is an experimental operating system being developed by volunteers – current and former students of Faculty of Mathematics and Physics of Charles University in Prague and a handful of external developers. It supports many processor architectures, among which there are the 64-bit SPARC processors. By the time the work on this thesis started, HelenOS had already supported some older 64-bit SPARC processor models. The aim of this thesis is to add support even for the newer 64-bit SPARC processor models so that all HelenOS features could be used on these models.

The processor models for which the support is intended are:

- the newer UltraSPARC III, IIIi, III+, IV and IV+ processors, and
- even newer UltraSPARC T1 and T2 processors.

The effort should result in a stable operating system with support for:

- multithreading,
- memory management,

- userspace,
- multiprocessor configurations.

This thesis introduces the newer SPARC processors, describes the process of porting HelenOS to those processors, and provides a comparison of the support for those processors in HelenOS with other operating systems.

1.3 Getting the Sources

The sources are maintained in a Subversion repository. To get the trunk, use:

```
svn checkout \
    svn://svn.helenos.org/HelenOS/trunk \
    HelenOS
```

Apart from the trunk, a separate branch dedicated to this thesis called `sparc` exists. It is recommended to view the SVN log of the branch instead of the trunk to get a detailed overview of the changes made to the sources by the author of this thesis. Moreover, by the time this thesis was written, some changes from the branch had not been merged into the trunk yet. To check out the branch, use:

```
svn checkout \
    svn://svn.helenos.org/HelenOS/branches/sparc \
    HelenOS
```

1.4 Acknowledgements

The work on HelenOS (initially called *SPARTAN Kernel*) was started by Jakub Jermář. His work was joined by a group of other students of Faculty of Mathematics and Physics who extended HelenOS considerably and added support for a lot of new architectures. Support for the 64-bit SPARC processors was added by Jakub Jermář in 2006. Concurrently with the work on this thesis some new features were added to HelenOS. Let us point out

- loader, dynamic linker and debugger (by Jiří Svoboda),
- simple shell (by Tim Post),
- support for real-world Itanium servers (by Jakub Váňa) and
- a lot of further enhancements (by Jakub Jermář, Martin Děcký, Jiří Svoboda and others).

The code written as a part of this thesis is intermixed with the code written mainly as a part of [jj_thesis]. It is contained mainly in the following directories:

- `boot/arch/sparc64/loader`,
- `contrib/util`,
- `kernel/arch/sparc64`,

- `uspace/srv/fb` and
- `uspace/srv/kbd`.

The code written as a part of this thesis is usually either located in a subdirectory called `sun4v` or it is inside a block delimited by `#if defined (US3)` and `#endif` directives. Some pieces of code which are a part of this thesis are, however, located in different places. Similarly some pieces of code which were *not* written as a part of this thesis have been borrowed and used in a `sun4v` directory or inside an `#if` .. `#endif` block. Thus, the only way how to reliably identify pieces of code written as a part of this thesis is inspecting the log of the `sparc` branch of the HelenOS SVN repository.

1.5 How to Read this Document

In the text of this thesis the particular processor models are described and compared, the environments are briefly introduced, the original SPARC port is introduced and the enhancements done as a part of this thesis are explained.

Chapter 2 introduces the HelenOS operating system and its architecture.

Chapter 3 describes the very basic properties of the SPARC 64-bit processors which are common for all CPU models HelenOS supports.

Chapter 4 focuses on UltraSPARC III, III+, IIIi, IV and IV+ processors. In the chapter properties specific to these processors are introduced and where suitable, a comparison with their predecessors (UltraSPARC I, II and IIi) is provided.

In Chapter 5 the newest family of the SPARC 64-bit processors (UltraSPARC T1, T2) is introduced. Its basic features and properties are described.

Chapter 6 summarizes the implementation of the original port of HelenOS onto the SPARC 64-bit processors made by Jakub Jermář.

Then, the two most important chapters follow:

Chapter 7 describes the implementation of the port of HelenOS onto newer SPARC 64-bit processors (UltraSPARC III, III+, IIIi, IV and IV+).

Chapter 8 describes the implementation of the port of HelenOS onto the newest SPARC 64-bit processors (UltraSPARC T1, T2).

Chapter 9 compares the port to the newer and the newest SPARC processors with other HelenOS ports and with other operating systems.

Chapter 10 summarizes what has been achieved and outlines the perspectives of the work.

In Glossary 11 the reader can find definitions of the terms used throughout this book.

Chapter 2

HelenOS Overview

HelenOS is composed of a tiny microkernel and of servers – a set of userspace tasks providing basic operating system services. The microkernel is responsible for the time management, scheduling and synchronization, and memory management. The userspace servers implement the filesystems and drivers of devices such as frame-buffer, keyboard or RAM disk.

HelenOS design is thoroughly described in [helenos][1]. In this chapter only the most significant design features are dealt with.

2.1 Time Management and Scheduling

Each processor regularly invokes a timer interrupt. HelenOS kernel handles these interrupts. Kernel uses these interrupts to

- keep track of the real time, and
- preempt the threads if they are running too long, so that no thread can usurp the whole CPU.

Upon a timer interrupt the kernel increments the variable representing the real time by a number of microseconds which have passed since the last timer interrupt.

2.1.1 Threads

A thread is a basic unit of execution. An illusion is made that the threads run in parallel. On uniprocessor systems this is achieved by switching regularly among active threads (round robin with multilevel feedback, separate run queues for each CPU). On multiprocessor systems the threads may really run in parallel, each thread on its own CPU; if the number of threads exceeds the number of CPUs, threads are alternated on a particular CPU. To make all the CPUs roughly equally busy, a balancing thread is running on background. Each CPU has its own load balancing thread; when it detects that there are fewer ready threads than on an average CPU, it steels threads from the busy CPUs.

Each thread has its own stack. User processes are called *tasks* in HelenOS. A task is a group of threads which share the same virtual address space.

[1]Unfortunately some portions of that document may be obsolete.

2.1.2 Fibrils

Fibrils are units of execution which are recognized purely by the userspace and which the kernel is not aware of. Several fibrils may be mapped onto one ordinary userspace thread.

2.2 Synchronization

2.2.1 Active Primitives

On the very low level, *spinlock* is used. The spinlock code tries to grab the lock by atomically setting the variable it is associated with using an architecture-dependent test-and-set instruction. If the variable is already set, the code actively waits (spins) until the variable becomes unset again, then it retries to grab the lock.

The spinlock code is usually optimized so that on uniprocessor systems only the preemption is disabled upon locking the spinlock, without spinning. On multiprocessors the preemption is disabled as well for the thread holding the lock, in order to prevent the priority inversion problem.

2.2.2 Passive Primitives

The basic passive synchronization primitive is called a *wait queue*. All other passive primitives build on it.

Wait queue allows a thread to sleep until an event associated with the queue occurs. If the event occurs before the thread starts to wait for it, it is recorded in a counter called *missed wakeups*. When some thread starts to wait for an event associated with a wait queue for which the missed wakeups variable is greater than zero, the missed wakeups variable is decremented by 1 and the thread does not have to sleep.

By pre-setting the missed wakeups to the number of threads which are allowed to enter a critical section, the wait queue will behave exactly as a semaphore (with the *wait* operation having the same effect as semaphore *down* operation and invoking the wait queue event having the same effect as the semaphore *up* operation).

Mutex is just a binary semaphore. Read-write lock is implemented in a way such that neither readers nor writers starve.

2.3 Memory Management

2.3.1 Frame Allocator

Upon startup, the kernel detects all available regions of the physical memory. The frame allocator allocates contiguous regions of physical memory, the size of the regions must be a power of two. For keeping track of all available regions of physical memory, buddy system is used.

2.3.2 Slab Allocator

Slab allocator allocates objects to be used by the kernel. A typical kernel object is small and kernel uses a lot of instances of the same type of object. Therefore it is reasonable to pre-allocate multiple objects of the same type at once, so that the average cost of an object allocation is low and memory fragmentation is minimized.

That is exactly what the slab allocator does. It pre-allocates multiple instances of objects of the same type. Pointers to those objects are stored in a cache and returned to the kernel when the kernel asks for the objects. Objects which the kernel does not use anymore are returned to the cache instead of being immediately deallocated.

If the kernel asks for an object type which the slab allocator has run out of, the slab allocator pre-allocates a new set of objects of that type. Similarly, if the kernel starts to run out of memory, the pre-allocated objects are deallocated.

The `malloc` function builds upon the slab allocator. The `malloc` function allocates pieces of memory whose sizes are multiples of two. Special 'pseudotypes' of objects are defined whose sizes are multiples of two. The `malloc` function then works by allocating instances of this 'pseudotype'.

2.3.3 Virtual-to-physical Mapping

HelenOS kernel makes use of paging. Segmentation is not used on any system which supports it, nor can the kernel run on MMU-less machines.

The virtual-to-physical mappings are stored in a structure called *page table*. On some architectures the page tables are managed by the hardware, while on others they are managed by the kernel. Anyway, the interface for accessing the page tables is unified in HelenOS. Three basic operations are defined: `page_mapping_insert`, `p-age_mapping_find` and `page_mapping_remove`. The actual implementation of these functions depends on the page table mechanism used for the given platform. HelenOS defines two page table mechanisms – global hashtable (where mappings of all the address spaces are stored) and hierarchical 4-level page tables.

2.4 Userspace Support

2.4.1 Passing Information from Kernel

Since kernel is able to gather some information about the host system which is not accessible by userspace, a mechanism for passing pieces of information from the kernel to userspace exists. The mechanism is called *sysinfo*. Kernel calls a special function, passing a string (as a key) and a number (as a value) to it. The userspace task then picks the value up by calling a function to which the key is passed. The sysinfo mechanism is commonly used by device drivers to pass the address where the device is mapped in the memory and other information.

2.4.2 IPC

Interprocess communication (IPC) is a mechanism via which the tasks communicate with each other. It is heavily used for communication between server tasks and ordinary applications. The IPC is asynchronous, analogous to phones (on the client side) and answerboxes (on the server side).

In order to make the IPC mechanism easily usable, a framework called *asynchronous framework* has been implemented. The asynchronous framework is described at the project wiki ([wiki]).

2.4.3 Important Servers

Name Server (`ns`)

The name server is a registry of all other servers. When a task needs to communicate with a certain server S, it asks the nameserver for a phone to S.

VFS Server (`vfs`) and Filesystem Servers

VFS server is a key server in the filesystem implementation. It accepts requests for filesystem operations from the tasks and dispatches them to servers which implement particular filesystems (TMPFS, FAT).

RAM Disk Server

The RAM disk server represents a virtual block device which uses physical memory as a storage. It is used by a filesystem implementation to store the initial user space environment (tasks, configuration, etc.).

Keyboard (`kbd`), Framebuffer (`fb`)

The keyboard server encapsulates drivers of several different keyboard models. The framebuffer server encapsulates

- drivers of several different framebuffer models, and
- drivers of several output devices which do not use a framebuffer (but for example a serial line).

The servers provide a unified interface to its clients, so that the clients do not have to bother with a particular type of input or output device.

Console

In the HelenOS user interface several virtual consoles may exist – each task that needs an output has its own virtual console. The user can switch between the consoles. The console server implements the virtual consoles mechanism.

Chapter 3

64-bit SPARC Processors Overview

This thesis deals with two architectures of the SPARC 64-bit processors. The first architecture conforms to the SPARC V9 specification [sparc_v9] and it will be referred to as *sun4u* throughout the thesis. The second architecture follows the newer UltraSPARC Architecture 2005 specification [us2005hp] and will be referred to as *sun4v* throughout the thesis. Strictly speaking the terms *sun4u* and *sun4v* are commonly used to denote architecture of the whole machine rather than architecture of the processor. For sake of simplicity, however, in this thesis these terms will be used to denote architecture of the processors.

The SPARC V9 specification defines a common subset of properties that all conforming processor models must have. On the other hand, some properties[1] are left undefined in the SPARC V9 specification – they are further defined in so called *implementation supplements*. The SPARC V9-conformant processors can be logically grouped into

- the older models UltraSPARC I, UltraSPARC II, UltraSPARC IIe and UltraSPARC IIi, and
- the newer models UltraSPARC III, UltraSPARC IIIi, UltraSPARC III+, UltraSPARC IV and UltraSPARC IV+.

Processors from the same group are very similar to each other. Since processors from the former group have already been thoroughly described in [jj_thesis], this thesis will only focus on processors from the latter group. Common properties of the processors from the latter group are summarized in a separate document called *Joint Programming Specification* [jps][2].

UltraSPARC Architecture 2005 specification defines common properties of the sun4v architecture processors. This thesis deals with the UltraSPARC T1 model (further described in UltraSPARC T1 Supplement [t1hp]) and the UltraSPARC T2 model.

This chapter summarizes common features of both the sun4u and the sun4v processors. Features which are specific to the particular architecture are summarized in Chapter 4 and Chapter 5.

[1]Examples of such properties are: memory management unit, timer support, inter-processor interrupts and some special registers and instructions.

[2]Joint Programming Specification is a common document created by Sun Microsystems and Fujitsu, as Fujitsu also manufactures JPS-compliant processors called *SPARC64® V*. Fujitsu models are, however, not covered in this thesis.

3.1 Registers

This section deals with general purpose registers and an assorted set of control registers.

3.1.1 General Purpose Registers

Code running on a 64-bit SPARC processor can access 32 non-privileged 64-bit integer registers at a given instant. These registers are denoted by symbols `%r0` to `%r31`. They are partitioned into *global registers* (`%r0` to `%r7`, commonly also denoted by symbols `%g0` to `%g7`) and *windowed registers* (`%r8` to `%r31`). Section 3.1.3 and Section 3.1.4 discuss both global and windowed registers thoroughly.

3.1.2 Control Registers

Control registers are privileged registers, which represent the current state of the CPU. They are discussed in Section 3.1.5.

3.1.3 Global Registers

Global register `%g0` always reads as zero and writes to it are ignored.

There are multiple sets of global registers, but only one set is active at any given moment. A *normal globals* set is active when no trap is being processed. When a trap occurs, the normal globals set is shadowed by an alternative set. There are several alternative sets of global registers; the actual number and names of those alternative sets, however, depend on the architecture. See Section 4.1 for a description of the alternative sets of the global registers on the sun4u processors and Section 5.4 for a description of the alternative sets of the global registers on the sun4v processors.

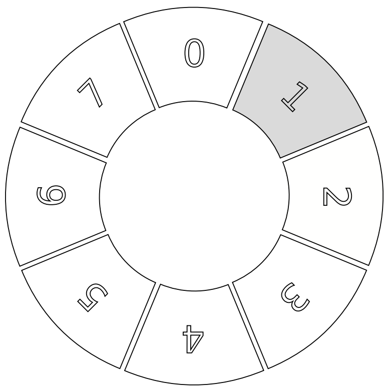
3.1.4 Windowed Registers

Registers `%r8` to `%r31` are called *windowed registers*. They are partitioned into

- output registers `%r8` to `%r15`, also denoted by `%o0` to `%o7`, which are used by a calling function to pass parameters to the callee and to read the return value,
- local registers `%r16` to `%r23`, also denoted by `%l0` to `%l7`, where the function can store values of its local variables, and
- input registers `%r24` to `%r31`, also denoted by `%i0` to `%i7`, which are used by a callee to read the parameters and pass the return value to the caller.

There are multiple sets of the windowed registers, but only one set is active at any given instant. A set of windowed registers is called a *register window*. The number of register windows is given by a CPU model-specific number NWINDOWS[3]. Register windows are numbered by numbers 0 to NWINDOWS - 1. The window which follows a window with number n has number $(n + 1)$ modulo NWINDOWS. Therefore after the (NWINDOWS - 1)-th window a 0-th window logically follows, as depicted on Figure 3.1.

[3]Which equals 8 on all processors mentioned in this thesis.



Register window with number 1 is the currently active window.

Figure 3.1: Register Windows Arrangement.

Functions store their local variables and call parameters to the active register window. During a function call a new register window is allocated, shadowing the register window of the caller. More precisely, in its prologue a function can issue the `SAVE` instruction, which results in deactivating the *n*-th register window and activating the (*n* + 1)-th (mod NWINDOWS) register window. Similarly, upon a return from a function call the current register window is deallocated and the register window of the caller becomes active again. More precisely, in its epilogue a function can issue the `RESTORE` instruction, which results in deactivating the (*n* + 1)-th window and re-activating the *n*-th window.

In order to make passing function parameters faster, output registers of the *n*-th register window overlap with the input registers of the (*n* + 1)-th register window. Hence performing the `SAVE` instruction causes that all values accessible via the output registers become accessible via the input registers. Overlapping of windows is depicted in Figure 3.2.

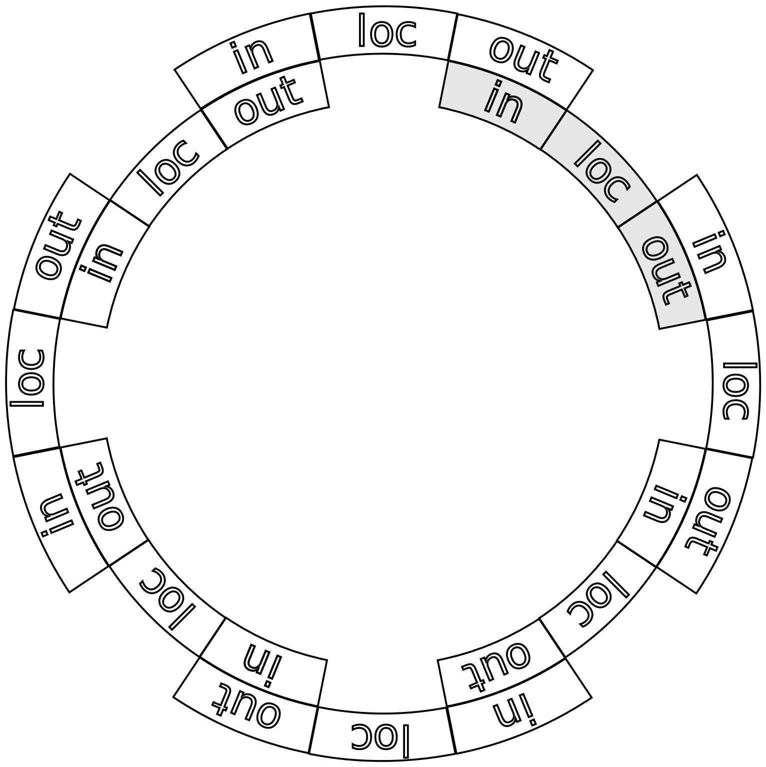


Figure 3.2: Overlapping of Registers.

If the `SAVE` instruction is issued and a new register window can not be allocated (there is a limited amount of register windows), a trap to the operating system is taken. Operating system copes with the situation by saving (in UltraSPARC terminology *spilling*) the oldest register window onto the stack. Similarly, if a `RESTORE` instruction is issued and the register window to which the code wishes to switch has been spilled to the stack, a trap to the operating system is taken. Operating system copes with the situation by reloading (in UltraSPARC terminology *filling*) the values from the stack to the register window.

There is a handful of control registers which describe the configuration of register windows:

CWP, current window pointer

Contains the number of the currently active register window. It can contain values between 0 and NWINDOWS - 1.

CANSAVE

The value in the `CANSAVE` register denotes how many times the `SAVE` instruction can be issued before a spill trap will be generated.

CANRESTORE

The value in the `CANRESTORE` register denotes how many times the `RESTORE` instruction can be issued before a fill trap will be generated.

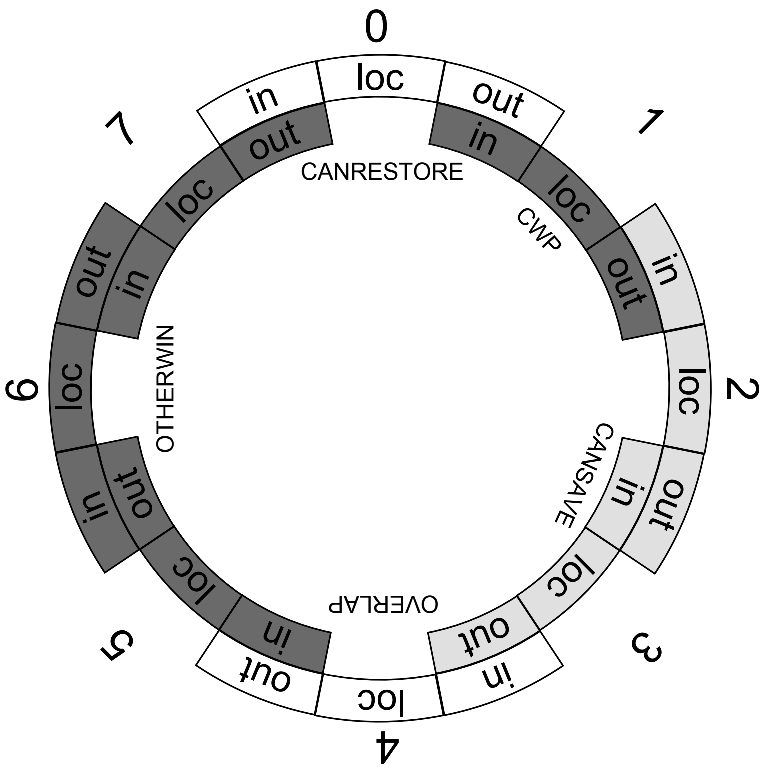
OTHERWIN

This register is used to mark a contiguous block of register windows which belong to a different address space and their spilling should be therefore performed by a different handler. Windows which belong to the different address space are called *other windows*. Windows which belong to the current address space are called *normal windows*.

The following equation holds:

CANSAVE + CANRESTORE + OTHERWIN + 1 + 1 = NWINDOWS.

The first "1" in the equation represents the current register window. The second "1" in the equation represents so called *overlap window*, which is a window whose output registers overlap with the input registers of the first occupied window and whose input registers, if all the windows had been occupied, would overlap with the output registers of the last window. An example register window configuration is depicted in figure Figure 3.3.



Window number 1 is the current window. Value of the CANSAVE register is 2, so the software can issue the SAVE instruction twice without invoking trap to the privileged software. Value of the CANRESTORE register is 1, which means that there is one more normal register window to which software can switch by issuing the RESTORE instruction without the need to reload register values from the memory stack. Value of the OTHERWIN register is 3, which means that there are three register windows which belong to a different address space and their spilling will be performed by a different spill handler. Window number 4 is an overlap window.

Figure 3.3: Example of Register Window Configuration.

Privileged software can define up to eight different spill handlers for normal windows and up to eight spill handlers for other windows. Similarly, privileged software can define up to eight different fill handlers for normal windows and up to eight fill handlers for other windows. The particular handler which will be used is selected according to the value of the `WSTATE` register[4].

The last thing related to windowed registers worth mentioning is a special meaning of some registers. The `%o6` register, as defined by the application binary interface (ABI), is a *stack pointer*, i.e. an address of the top of the stack[5]. When the `SAVE` instruction is executed, the original value of the `%o6` register becomes accessible via the `%i6` register. The `%i6` register is also called a *frame pointer*, because its value represents an address[6] of the stack frame (the top of the stack of the calling function). The `%o7` register is a register where a calling function stores the values of its program counter when performing the call.

3.1.5 Control Registers

Here is a list of the very basic control registers, whose knowledge is recommended for understanding the ongoing parts of this thesis.

PC, program counter

Contains the address of the instruction being currently executed.

NPC, next program counter

Contains the address of the instruction which is to be executed next.

PSTATE, processor state

Encapsulates values of several flags, such as the privilege level or the active set of global registers.

3.2 Traps

A *trap* is a transfer of control to the privileged software. Examples of traps on 64-bit SPARC processors include

- register window spills and fills (see Section 3.1.4),
- memory management exceptions,
- illegal instructions,
- hardware, inter-processor and timer interrupts.

3.2.1 Trap Levels

SPARC 64-bit processors allow nested traps. When a trap is being processed, another trap can come and be processed without destroying the state of the first trap. The depth of trap nesting is limited, though. The depth depends on the processor architecture, but it is usually not bigger than five levels. The current trap level is represented by the

[4]The `WSTATE` register has six significant bits. Bits 0 to 2 contain the index of the spill (fill) handler which will be used should the normal window trap. Bits 3 to 5 contain the index of the spill (fill) handler which will be used should the other window trap.

[5]Address of the top of the stack is actually equal to `%o6` + 0x7ff.

[6]Address of the top of the stack frame is actually equal to `%i6` + 0x7ff.

value of the `TL` register. When no trap is being processed, the `TL` register has value 0. When a trap comes, the value of the `TL` register is increased by one.

When a trap comes, the state of the processor is snapshotted to special registers so that it can be renewed as soon as the trap is processed. The registers are as follows:

TSTATE

Encapsulates the value of the `CWP` and `PSTATE` registers and other essential state valid at the time the trap was taken.

TPC

Contains the value of the `PC` register valid at the time the trap was taken.

TNPC

Contains the value of the `NPC` register valid at the time the trap was taken.

There are multiple sets of `TSTATE`, `TPC` and `TNPC` registers, the number of the sets equals to the number of trap levels above zero. If `TL` equals $n > 0$, the `TSTATE`, `TPC` and `TNPC` registers always contain the state of the CPU valid the last time when `TL` was equal to n - 1.

3.2.2 Trap Table

The type of the trap can be determined from the `TT` (*trap type*) register. The `TT` register contains a number between 0 and 511, which acts as an index to the trap table. Each entry in the trap table is a sequence of a limited number of instructions (32 for the most common types, 8 for other types). The table is partitioned into two parts, the first part contains trap handlers which will be invoked if the trap occurs when `TL` = 0, the second part contains handlers which will be invoked if the trap occurs when `TL` > 0.

Trap table is placed in the main memory, its address is set using the privileged `TBA` (trap base address) register.

3.2.3 Returning from a Trap

There are two instructions for returning from a trap: `DONE` and `RETRY`. They both decrease the trap level (`TL`) by one and restore the state saved in `TSTATE`, `TPC` and `TNPC` registers. After the `DONE` instruction the control is transferred to the instruction which follows the instruction last executed before the trap. After the `RETRY` instruction the control is transferred to the instruction last executed before the trap, which means that the instruction last executed before the trap is re-executed.

3.3 Memory

This section describes how memory is virtualized on UltraSPARC processors, introduces memory access instructions and mentions how to access special registers using these instructions.

3.3.1 Virtualization

UltraSPARC processors are equipped with a memory management unit which translates virtual page numbers onto physical frame numbers[7].The translations are managed entirely by software, hardware provides only a small but fast cache of translations called *translation lookaside buffer*. The MMU supports 8 kB, 64 kB, 512 kB, and 4 MB page sizes.

Address translation mechanism can be split so that one translation mechanism is used for instructions and a different one for data, even though the same physical memory is still used. Privileged software may, however, keep the same virtual-to-physical mappings for both instructions and data.

3.3.2 Memory Access Instructions

There are only a few instructions for memory access: load instructions, store instructions and load-store instructions (which are used for synchronization purposes). All arithmetical and logical operations are performed purely on registers without interacting with memory. Load and store instructions are used not only to access memory, but also to access I/O address spaces and some special registers.

3.3.3 Address Space Identifiers

When issuing a load or store instruction, an *address space identifier* (ASI) can be specified as one of the instruction arguments. Address space identifier is a number between 0 and 255. Generally speaking, the address space identifier determines how the load or store instruction is to be interpreted. Address space identifier influences

- whether an ordinary memory or a special register is accessed,
- whether accessing the address has a side effect (accessing I/O addresses),
- whether the data being loaded/stored are big-endian or little-endian,
- whether the memory management unit is bypassed (no virtual address translation),
- which virtual address space (*context* is SPARC terminology) is accessed,
- and other issues.

3.3.4 Memory Contexts

Virtual address spaces are called *contexts*. Memory management unit may concurrently keep track of up to 2^{13} contexts. Contexts are distinguished by *context identifiers*[8], numbers between 0 and 2^{13} - 1.

[7]On UltraSPARC Architecture 2005 it is a little bit more complicated.

[8]The term 'context' is also known as ASID on some architectures, do not confuse with ASI.

Context Registers

Three contexts may be active concurrently. Which one of the three contexts will be used for particular memory operations depends on the ASI used and on the current trap level.

NUCLEUS

This context is used when the trap level (`TL`) is greater than 0, i.e. when a trap is being processed. The context ID is hardwired to 0.

PRIMARY

Context used for normal memory access. Normal memory access is access with ASI equal to `ASI_PRIMARY`[9]. This context's ID is stored in the *Primary Context Register*.

SECONDARY

Context used for memory access using ASI equal to `ASI_SECONDARY`. Whether and how the access via `ASI_SECONDARY` is used depends purely on the software. This context's ID is stored in the *Secondary Context Register*.

3.3.5 Translation Lookaside Buffers

Translation tables are software-managed on the 64-bit SPARC processors. Hardware keeps track of a limited set of translations in a *translation lookaside buffer* (TLB). There are separate TLBs for instruction and data memory. The instruction TLB is commonly referred to as ITLB and the data TLB is commonly referred to as DTLB. Depending on the architecture, the TLB contains between 64 and 512 entries. On some architectures there may be more than one ITLB and more than one DTLB in one memory management unit.

Translation Table Entry

Translation table entry holds information for a single page mapping. It consists of a 64-bit *tag* and *data* words. The tag contains the context identifier and the virtual address. It is used to determine whether there is a hit. The data contains the physical address and flags of the mapping. The flags determine whether the mapping is locked (i.e. can not be automatically expelled from the TLB) or whether the mapped page is writable, privileged, cacheable or executable.

3.3.6 Memory Management Traps

If a page which is not contained in the TLB is accessed, a trap is invoked. Privileged software handles the trap by looking up the translations in its translation tables and (if found) reloading the translation to the TLB. The basic MMU traps are:

Fast Data Access MMU Miss

An instruction accessed data not mapped in DTLB.

[9]Implicit ASI, the value of the `ASI_PRIMARY` constant is 0x80.

Fast Data Access Protection

An instruction accessed data mapped as read-only in DTLB.

Fast Instruction Access MMU Miss

The code to be executed is not mapped in ITLB.

The traps are called 'fast', because in the trap table they can take up 32 instructions[10]. Thus, in most cases the trap can be handled purely by the code contained in the trap table, without branching.

3.3.7 Translation Storage Buffers

Translation storage buffer is an in-memory cache of page mappings managed by operating system. It is used to quickly find a translation which cannot be found in the TLB. Usage of TSB is optional, operating system can work completely without TSB. For each context a separate TSB[11] can be defined. The translation storage buffer is an array indexed by a sequence of the n least significant bits of a page number. Number n is defined by an operating system. An entry in the TSB consists of a TTE tag and TTE data. The size of the tag is 8 bytes, as well as the size of the data. The total number of entries in one TSB equals 2^n, which is usually a number significantly greater than the size of the TLB (moreover, each context has its own TSB, while the TLB is shared among contexts).

Reloading the translation from the TSB is slower than looking it up in the TLB; the SPARC processors, however, provide some hardware support for speeding it up: The privileged software just supplies some essential information to the hardware, such as the location of the TSB in the memory. Then, when the MMU trap comes, the hardware automatically pre-computes the address of the TSB entry which potentially contains the translation of the page which caused the trap. The privileged software then checks whether the tag matches the page that has caused the trap; if so, it copies the entry to the TLB, if not, it looks the translation up in its page tables.

3.4 Interrupts

This section briefly describes the most important interrupts – timer interrupts and inter-processor interrupts.

3.4.1 Timer Interrupts

64-bit SPARC timer support is based on two registers: `TICK` and `TICK_COMPARE`. Value represented by bits 0 to 62 of the `TICK` register is incremented by one at every clock cycle[12]. When the value of the `TICK` register equals to the value of the (privileged) `TICK_COMPARE` register, a tick interrupt is generated.

Newer UltraSPARC processors (JPS-compliant and sun4v) define two more registers: `STICK` and `STICK_COMPARE`. They have the same function as the `TICK` and `TIC-K_COMPARE` registers. The only difference is that the value of bits 0 to 62 of the `STICK`

[10]Unlike most other traps, whose handler can take up to 8 instructions only.

[11]More precisely a pair of TSBs: one instruction TSB and one data TSB.

[12]Bit 63 controls read access to this register for the non-privileged software.

register is not incremented at every clock cycle, but at a rate given by an external clock signal.

3.4.2 Inter-processor Interrupts

Inter-processor interrupt (IPI) is an interrupt invoked by one CPU and received by a different CPU. An example of its usage is TLB-shootdown[13] in a multiprocessor environment. During the inter-processor interrupt a short massage is passed from the sender to the receiver. The message is called an *interrupt vector*. The length of the vector depends on the processor (sub)architecture. Upon receiving the IPI, a trap is invoked on the receiving CPU. The trap handler can read the vector using special registers. What the sending CPU must do in order to invoke the IPI depends on the architecture – sun4u processors write the message contents to special registers, whereas sun4v processors will typically use a hypercall.

3.5 OpenBoot PROM

OpenBoot PROM (OBP) is not a part of the SPARC processors specifications, but it is present in almost all SPARC machines. Since operating systems may use some information OBP provides, this section will briefly introduce it.

OpenBoot PROM (OBP) is a firmware which is active from the moment the machine is switched on and its purpose is to boot the operating system. Apart from booting, OBP can also provide the operating system with some essential information. The information is structured in a tree-like structure called *OBP tree*. The operating system can investigate the tree via *OBP client interface*[14]. The root node of the OBP tree represents the whole machine. Nodes deeper in the tree represent devices, memory, CPUs, etc. With each node, a set of key-value pairs is associated, representing node properties. An example of a property of a `memory` node is the memory starting address. An example of a property of a `CPU` node is a clock frequency.

The root node's `compatible` property contains the name of the architecture, hence it can be used to detect whether the host platform is sun4u or sun4v.

[13]When a mapping is modified of removed or when a virtual address space identifier becomes invalid on one CPU, (portions of) TLB must be invalidated on other CPUs as well.

[14]When booted, the operating system has a pointer to a certain function in one of the registers. The function is an entry point of the OBP client interface, via the entry point all OBP services are accessible.

Chapter 4

JPS-compliant Processors

The *Joint Programming Specification* [jps] is a document created by Sun Microsystems and Fujitsu. It describes common features of processors belonging to the UltraSPARC III subarchitecture. This chapter focuses on the UltraSPARC III, UltraSPARC III+, UltraSPARC IIIi, UltraSPARC IV and UltraSPARC IV+ processors, but the information presented here should hold for the Fujitsu processor models as well.

Manuals for the particular processor models can be downloaded from the following site:

```
http://www.sun.com/processors/documentation.html
```

From the system programmer's point of view, there are only a few differences between particular models. The differences concern mainly TLB size and the OBP tree structure. The main difference between the UltraSPARC III and UltraSPARC IV series is that UltraSPARC IV contains dual-core processors.

4.1 Registers

In Section 3.1.3 it has been mentioned that there are multiple sets of global registers (`%g0` to `%g7`) and that the *normal globals* set is shadowed by an alternative set when processing a trap. It has also been mentioned that the number and nomenclature of the alternative sets depends on the (sub)architecture.

As for the JPS-compliant processors, there are *three* alternative sets of global registers.

alternate globals

This set is active when a trap is being processed, except for interrupts and memory management traps.

memory globals

This set is active when a memory management trap is being processed.

interrupt globals

This set is active when an interrupt is being processed.

4.1.1 Setting the Registers Set

The alternative global registers set is implicitly enabled upon the trap, similarly upon the return from the trap the *normal globals* set is reactivated. The privileged code may, however, activate an arbitrary global registers set explicitly. The currently active register set is specified by the `AG`, `MG` and `IG` bits of the `PSTATE` register (see Section 3.1.5). If all these bits are set to 0, *normal globals* set is used. If the `AG` bit is set to 1 and the remaining two bits are set to 0, *alternate globals* set is used. If the `MG` bit is set to 1 and the remaining two bits are set to 0, *memory globals* set is used. If the `IG` bit is set to 1 and the remaining two bits are set to 0, *interrupt globals* set is used.

4.2 Memory Management

The UltraSPARC III subarchitecture slightly differs from the older UltraSPARC[1] subarchitecture. The virtual address space, whose size was 2^{44} bytes on the older 64-bit SPARC processors, is extended to 2^{64} bytes on the UltraSPARC III subarchitecture. The physical addresses, which could be at most 41 bits long on older 64-bit SPARC processors, have been extended to 43 bits. There are multiple ITLBs and DTLBs in one memory management unit.

4.2.1 Translation Lookaside Buffers

On older 64-bit SPARC processors there was only one ITLB and one DTLB, each holding at most 64 entries. On the UltraSPARC III subarchitecture there are multiple DTLBs and multiple ITLBs.

There is *one* small DTLB which can hold *both locked and unlocked* entries.

There are *two* large DTLBs which can hold *unlocked* entries only.

There is *one* small ITLB which can hold both *locked and unlocked* entries.

There is *one* large ITLB which can hold *unlocked* entries only.

The size of the TLBs is slightly different on UltraSPARC III and UltraSPARC IV series. The sizes are summarized in Table 4.1.

TLB	**size on UltraSPARC III**	**size on UltraSPARC IV**
small DTLB	16	16
big DTLB	512	512
small ITLB	16	16
big ITLB	128	512

Table 4.1: TLB sizes

As the locked entries can be contained in the small TLB only, the maximum number of locked entries is limited to 16 on UltraSPARC JPS-compliant processors.

4.2.2 MMU Registers

This section describes how the translation table entry is reloaded to the TLB when the MMU miss occurs. The process is almost the same as on older 64-bit SPARC processors.

[1]UltraSPARC I, UltraSPARC II, UltraSPARC IIi, UltraSPARC IIe – they are not covered in this thesis.

TLB entries are manipulated via special registers, which are accessible through special ASIs (see Section 3.3.3). The ASI numbers and virtual addresses through which these special registers are accessible are defined in the Joint Programming Specification [jps]. All of the registers mentioned exist in two variations – one for the ITLB and one for the DTLB. For sake of simplicity, only one of those two registers is defined in the following list.

MMU Tag Access Register

To this register the privileged software can write the translation entry *tag* which is to be loaded into the TLB. The upper 51 bits contain bits 63 to 13 of the virtual address. The lower 13 bits contain the context identifier. Upon MMU miss, the MMU hardware implicitly writes both the virtual address and the context of the faulting page to this register, so that the operating system does not have to touch this register when processing the MMU miss. Nevertheless, the operating system is free to explicitly change the value of this register if it needs to.

Data In Register

To this register the privileged software can write the translation entry *data* which is to be loaded into the TLB. Writing to this register triggers an atomic write of both the tag (stored in the MMU Tag Access Register) and the data to the TLB. If any other entry must be evicted from the TLB, an automatic replacement algorithm is initiated. The algorithm depends on the CPU model.

MMU Demap

Three demap operations are provided: *demap page* demaps a single entry from the TLB, *demap context* demaps all entries with a given context identifier and *demap all* demaps all entries in the TLB[2]. The demap operation is performed by writing an arbitrary value via a special ASI (defined in [jps]) to the address of the format described in figure Figure 4.1.

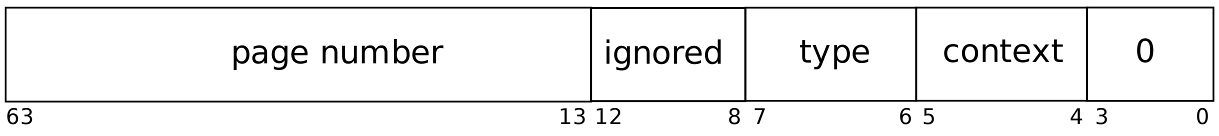


Figure 4.1: Demap Address Format.

Synchronous Fault Status Register (SFSR)

Instruction and Data Synchronous Fault Status Registers (SFSR) are privileged read-only registers which describe an MMU fault. They encapsulate the ASI which caused the fault and the exact condition that caused the fault (e.g. privilege violation, invalid ASI value).

[2]Demap all is present on JPS-compliant processors, but not on older UltraSPARC I and UltraSPARC II models.

Direct Access to TLB

UltraSPARC I to UltraSPARC IV+ processors define privileged ASI-accessible registers for direct diagnostic access to the TLBs. This feature is used for debugging purposes, such as printing the whole contents of the TLBs by the operating system.

4.2.3 Translation Storage Buffer

The basic principle of the translation storage buffer has been briefly described in Section 3.3.7. This section will shortly describe the interface provided by hardware to the operating system on the sun4u architecture and compare the interface of the older UltraSPARC processor models with the JPS-compliant models.

The size of the TSB is 2^n, where n is a privileged software-defined number greater or equal to 9 (which implies that the minimum number of TSB entries is 512). Hardware provides support for the 8 kB and 64 kB page sizes. Privileged software interfaces with the hardware by means of a handful of ASI registers. Again, each of the registers presented here exists in two variations – one for ITSB and one for DTSB.

TSB Tag Target Register

This register is used by the privileged software to find out whether the TSB entry tag matches the faulting page and context. Its format is the same as the format of the TSB entry tag. This register is updated automatically by hardware upon an MMU miss.

TSB Base Register

Privileged software sets the base address of the TSB using this register. Apart from the base address, this register also encapsulates other parameters of the TSB mechanism: the *size* of the TSB and whether the TSB is *split* (i.e. one half of the TSB contains entries for 8kB pages and the second half contains entries for the 64 kB page sizes). Figure Figure 4.2 illustrates the format of the register.

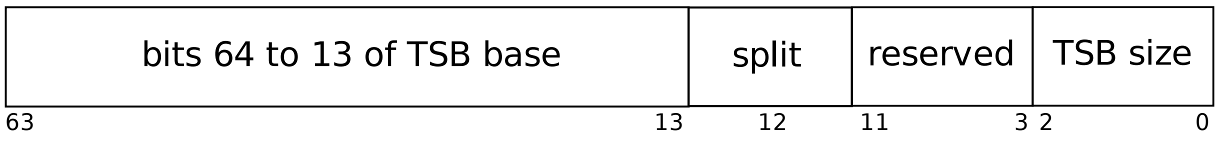


Figure 4.2: TSB Base Register format.

TSB 8kB Pointer Register

This register contains the precomputed address of the TSB entry whose tag potentially matches the faulting page. As entries in the TSB are indexed by a sequence of only n least significant bytes of the page number, collisions may occur in the TSB; therefore the privileged software must perform additional checks that the entry really matches the faulting page.

This register provides the entry address for the case of 8kB pages. Similarly, there exists an analogous register for 64kB page sizes.

Extension Registers

Up until now, all information about the TSB was valid for both the older UltraSPARC 64-bit processors and for the JPS-compliant processors. Extension registers are a feature introduced in the Joint Programming Specification, which enables privileged software to use three separate TSBs concurrently: for the primary, secondary and nucleus contexts.

It has already been stated that the operating system can define a separate TSB for every memory context (virtual address space). On the SPARC 64-bit processors three memory contexts can be concurrently active (primary, secondary and nucleus), yet only one TSB Base Register is present. Therefore, every time the software is about to make a memory access using `ASI_SECONDARY` and the TSB Base Address register still contains base address of the primary context TSB, the TSB Base Address register must be updated[3].

Joint Programming Specification solves this issue by introducing three more registers: *TSB Nucleus Extension Register*, *TSB Primary Extension Register*, and *TSB Secondary Extension Register*. The first two exist for both data and instruction address spaces, the last one exists for data address space only. Their format is exactly the same as the format of the TSB Base Address Register. When pre-computing a pointer to the TSB entry, hardware does not use TSB Base Address Register directly, but its value is first XOR-ed with the corresponding extension register. This enables a backward compatibility with older 64-bit SPARC processors: if the system is unwilling to use the extension registers, it sets them to 0, hence the address will be precomputed only using the TSB Base Address Register, i.e. the same way as on the older processor models.

4.2.4 Caches

SPARC 64-bit processors contain separate caches for instruction memory and data memory. Older processor models do not automatically keep instruction and data caches coherent. When a piece of code is modified during program execution[4], the `FLUSH` instruction is needed to synchronize data and instruction address spaces. The `FLUSH` instruction must be called for every doubleword[5] which has been modified.

Newer, JPS-compliant, processors automatically keep instruction and data caches coherent. The `FLUSH` instruction is still needed, though, but only in order to flush the instruction pipeline.

4.3 CPUs

4.3.1 CPU Identification

In a multiprocessor environment it is essential to distinguish between particular CPUs. Each CPU is identified by an integer number called *module ID* (`MID`) or *agent ID* (`AID`). The code can find the identifier of the CPU it is executed on by inspecting an *interconnect bus register*, which is called *UPA config register* on older sun4u processors and

[3]And similarly for other combinations of contexts.

[4]Code which modifies instruction stream during its execution is commonly referred to as *self modifying code* (SMC).

[5]Doubleword is 8 bytes large.

Fireplane config register on newer sun4u processors[6]. The register is accessible through a special ASI. The format of the register depends on the processor model, moreover the length of the `MID`/`AID` field may differ too. Table 4.2 summarizes the name of the processor ID, its size and name of the register it is accessible through in particular processor models.

UltraSPARC CPU model	**register name**	**field name**	**OBP property**	**length**
I, II, IIi	UPA Config	`MID`	upa-portid	5 bits
III, III+, IV, IV+	Fireplane Config	`MID`	portid	10 bits
IIIi	Fireplane Config	`AID`	cpuid	5 bits

Table 4.2: Processor IDs

CPU nodes in the OBP tree contain a property, whose value equals to the identifier of the CPU. The name of the property key varies among processor models, see table Table 4.2.

4.3.2 CPU Model Determination

Particular processor models are slightly different. Operating systems may detect the model by means of the `VERSION` register. The `VERSION` register is a privileged register which encapsulates the CPU manufacturer code and the CPU model code.

4.3.3 Dual-core UltraSPARC IV Processors

The main difference between the UltraSPARC IV series processors and the rest of the sun4u processors is that the UltraSPARC IV processors have two processor cores.

From the system programmer's point of view the processor core behaves like a standalone CPU. The only thing the operating system must be aware of is a different layout of the OBP tree. Single-core processor is represented by a single node in the OBP tree. All CPU properties accessible via OBP are associated with the node. Dual-core processor, however, is represented by an OBP node called 'cmp' (an acronym meaning *chip multiprocessing*), which has two subnodes representing the cores. The majority of the CPU properties accessible via OBP is associated with the subnodes.

Also the Agent ID (see Section 4.3.1) property is contained under the subnode. The Agent ID is a 10-bit integer. The most significant bit is 0 if the ID belongs to the first processor core and it is 1 if the ID belongs to the second core. The least significant bits identify the particular chip.

4.3.4 Inter-processor Interrupts

This section briefly describes the process of sending and receiving an interrupt vector.

[6]UPA and Fireplane are names of the interconnect buses on older and newer sun4u processors respectively.

Sending an Interrupt Vector

The sending CPU writes the vector data to special privileged ASI-accessible registers called *interrupt vector data* registers. On older sun4u architectures there are three interrupt vector data registers, on JPS-compliant processors there are eight such registers. The size of each register is 64 bits. The interrupt is triggered by writing to the *interrupt vector dispatch* register. One field of the interrupt vector dispatch register is the ID of the target CPU (see Section 4.3.1), other fields are hard-wired to a constant defined in [jps]. The status can be determined from the `BUSY` and `NACK` fields of the *interrupt dispatch status* register.

Receiving the Interrupt Vector

On the receiving CPU, a trap is invoked. The trap handler then reads the incoming data from the *interrupt data* registers. Analogously to the interrupt vector data, on older sun4u CPUs there are three interrupt data registers, on JPS-compliant CPUs there are eight interrupt data registers. Their size is 64 bits.

Chapter 5

UltraSPARC Architecture 2005

The UltraSPARC Architecture 2005 specification [us2005hp] is a document describing the newest architecture of the SPARC 64-bit processors, commonly referred to as *sun4v*. Even though the sun4v architecture has a lot of features common with the already described sun4u architecture (see Chapter 4), there are substantial differences the system programmer must be aware of.

This chapter briefly describes the general properties of the UltraSPARC Architecture 2005, focusing on two processor models – UltraSPARC T1 (also known under its codename *Niagara*) and UltraSPARC T2 (also known under its codename *Niagara 2*).

5.1 Virtual Processors

The UltraSPARC T1 and T2 processors are *multicore* and *multithreaded*. In the sun4u architecture there have already been multicore processors – UltraSPARC IV and UltraSPARC IV+. In addition, the sun4v processors are multithreaded – in each processor core there are multiple sets of all registers, each set belonging to one *strand* ('hardware thread'), and the processor core switches between running threads each clock cycle.

If there are 6 cores on one chip, each core able to run 4 strands (which is one of the possible configurations), to the software it appears as if there have been 24 processors. If four threads are running on a processor core, the throughput of a single thread is roughly 4 times smaller than the throughput of the whole processor core, but the overall throughput of the whole processor core is improved significantly:

- If multiple threads run on the system, they may run concurrently without the need of software context switching.
- This approach eliminates stalling of the processor during a cache miss. If a cache miss occurs in a thread, the thread is excluded from being switched to until the value is reloaded from the main memory, but the processor core keeps executing the code of the other threads.

The Niagara processors are fairly described in a Wikipedia article UltraSPARC T1 [wiki_t1].

5.2 Hypervisor

Hypervisor is an integral part of all UltraSPARC Architecture 2005 processors. It is thoroughly described in [hypervisor].

5.2.1 Overview

Hypervisor is a piece of firmware which lays between the sun4v hardware and the privileged software. Hypervisor is present on all sun4v machines and the operating system cannot disable it. Hypervisor makes porting the operating systems to other sun4v subarchitectures much easier, since the hypervisor isolates the operating system from hardware details. Hypervisor enables the system to be split into so called *logical domains*. From the operating system's perspective the logical domain behaves like a standalone machine[1]. A system administrator may allocate a group of virtual processors for one logical domain. In each logical domain a different operating system may run.

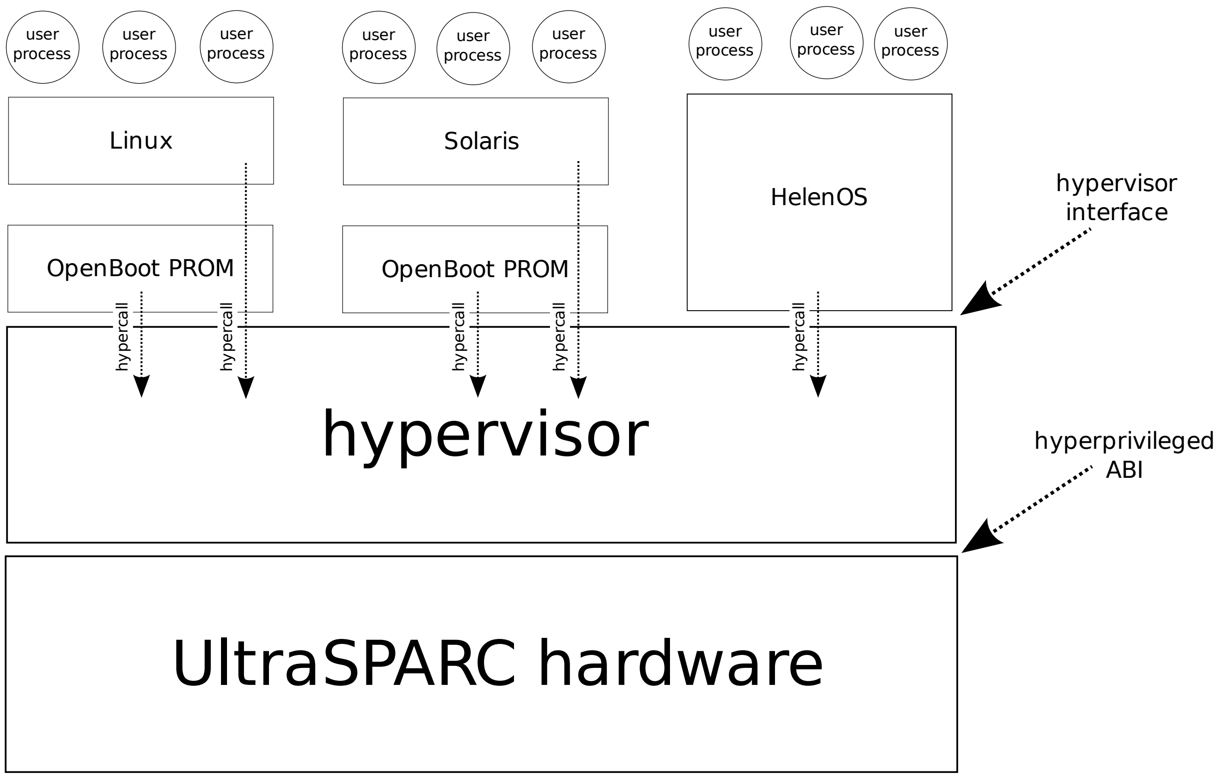


Figure 5.1: The sun4v Architecture

In order to protect the isolation of the logical domains and to make it impossible for the operating system to disable the hypervisor, the sun4v processors define some registers, traps and instructions as *hyperprivileged*. The term 'hyperprivileged' is analogous to terms *privileged* and *non-privileged*. The sun4v processors can, apart from the privileged and non-privileged protection modes, run in so called *hyperprivileged mode*. The

[1]This is a right place to mention, as a curiosity, the origin of the sun4u and sun4v abbreviations. Letter 'u' in the sun4u abbreviation comes from the name of the system bus in some sun4u processors – *UPA*. The system bus in the sun4v architecture is *virtualized*, hence the letter 'v'.

code running in the hyperprivileged mode can use all the registers, ASIs, instructions and traps of the CPU – hyperprivileged, privileged and non-privileged. The code running in the privileged mode can only access privileged and non-privileged ABI. The code running in the non-privileged mode can only access non-privileged ABI.

Only the hypervisor can run in the hyperprivileged mode. The operating system kernel will typically run in the privileged mode, whereas the user applications will only run in the non-privileged mode. The privileged ABI does not offer any means by which the operating system could perform operations such as adding a new page mapping or starting a CPU. These operations can only be performed using hypervisor API calls.

Figure 5.1 illustrates relations between the hypervisor and other parts of the sun4v-based system.

5.2.2 Hypercalls

A *hypercall* is analogous to a syscall. While using the syscall the user process can ask the kernel for some service, using the hypercall the kernel can ask the hypervisor for some service.

Calling Conventions

Calling conventions define how the hypercall parameters are passed from the privileged code to the hypervisor and how the return value is passed from the hypervisor to the privileged code.

The transfer of control from the privileged to the hyperprivileged code is done using the `ta`[2] instruction. The trap codes used are 0x80 and above. Traps with code 0x80 and above are always hyperprivileged.

Parameters of the hypercall are always passed in the output registers `%o0` to `%o4`. The error code of the hypercall can be subsequently read from the `%o0` register.

Hypercalls can be divided into *hyperfast*, *fast* and *core*. *Hyperfast hypercalls* are hypercalls for which the function number is encoded in the trap code. *Fast hypercalls* are hypercalls for which the function number is passed in a register; all fast hypercalls share the same trap number. *Core hypercall* is a special kind of hypercall which is used to set the version of the hypervisor API interface; it is guaranteed to remain the same in all versions of the hypervisor.

Hyperfast Hypercalls

The function number of the hyperfast hypercall is determined from the trap code of the TA instruction. *MMU Map Address* and *MMU Unmap Address* are examples of hyperfast hypercalls. See the example Example 5.1 to see how the hyperfast traps are used.

[2] `ta` stands for trap always.

```
set 0x4000, %o0   ! pass the hypercall parameters
                  ! in output registers
set 1, %o1
mov %g3, %o2
set 1, %o3

ta 0x83           ! perform the hypercall, 0x83 is the trap code
                  ! for the MMU_MAP_ADDR trap
```

Example 5.1: Hyperfast Hypercalls

Fast Hypercalls

The function number of the fast hypercall is passed in the `%o5` register, the trap code is always 0x80. *CPU Start*, *Console Putchar* and *MMU Demap All* are examples of the fast hypercalls. See the example Example 5.2 to see how the fast hypercalls are used.

```
set 0x46, %o0    ! set hypercall parameters to output registers
set 0x61, %o5    ! function number goes to the %o5 register

ta 0x80          ! call the hypervisor
```

Example 5.2: Fast Hypercalls

5.2.3 Machine Description

Machine description (MD) is a data structure provided by the hypervisor to the privileged software which describes the whole machine. It contains similar information as the OBP tree (see Section 3.5). The basic difference between the OBP tree and the machine description is that the OBP can be inactive, in which case the OBP tree cannot be retrieved (OBP is not isolated from the operating system, so it is easy for the operating system to disable it). Since the hypervisor is always active, the machine description can always be retrieved.

Machine description is a tree-like structure. The basic building block of the machine description is an *MD node*. Each node holds a set of properties. There are several types of nodes:

- *CPU node*, which holds information about a single CPU, such as its ID and clock frequency,
- *memory block node*, which holds information about a block of virtual domain's memory, such as its base address and size,
- *platform node*, which holds information about the platform the privileged software is running on,
- and others.

The machine description can be retrieved via a (fast) `MACH_DESC` hypercall. The hypercall takes an address of a buffer, to which the MD will be copied, as its parameter. The format of the MD is described in [hypervisor].

5.3 Memory Management

On classical processor architectures there is usually only one level of memory virtualization: mapping of virtual memory pages onto physical memory frames. The sun4v architecture adds one more level. Virtual memory is not mapped onto physical memory directly. Instead, virtual memory is mapped onto so called *real memory*. Real memory is also virtualized. Translation of the real memory pages onto physical memory pages is managed by the hyperprivileged software.

Real memory address spaces are called *partitions*, which is a term analogous to *context*. Each logical domain has its own partition. Partitions are identified by *partition IDs*. Figure 5.2 shows how pages of virtual memory are mapped onto real memory and how real memory is mapped onto physical memory.

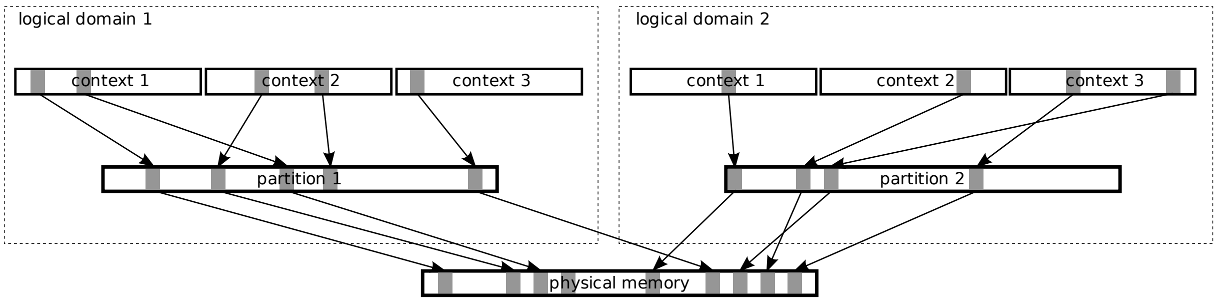


Figure 5.2: Two Levels of Memory Virtualization

Operating system kernel is fully isolated from the mechanism of this three-level memory virtualization. The kernel only takes care of the virtual-to-real mappings; the physical memory is completely hidden for the kernel.

5.3.1 Virtual to Physical Mapping

TLBs are hidden from the kernel. All registers for direct access to the TLB are hyperprivileged. The only way the kernel can manipulate page mappings is using the hypervisor API.

MMU Flags

Some hypervisor API calls which manipulate virtual-to-real mappings can operate on both instruction MMU and data MMU. Whether to operate on IMMU, DMMU, or both is specified by one of the hypercall parameters called *MMU flags*. MMU flags is an integer value; its bit number 0 is set if and only if the operation is to be applied on DMMU, its bit number 1 is set if and only if the operation is to be applied on IMMU, other bits are unset.

Installing a Mapping

The virtual-to-real mapping can be specified in one of three ways.

- Privileged software installs a non-permanent (not locked) mapping using the `MMU_MAP_ADDR` hypercall. The hypercall takes virtual address, context ID, TTE and MMU flags as its parameters.

- Privileged software installs a permanent (locked) mapping using the `MMU_MAP_-PERM_ADDR` hypercall. The hypercall takes virtual address, TTE and MMU flags as its parameters. Note that the hypercall does not take context ID as its argument – permanent mappings can only be installed for context number 0 (nucleus and kernel).
- Privileged software sets up a translation storage buffer (TSB). Upon an MMU miss the hypervisor automatically reloads the missing mapping from the TSB to the TLB.

Translation Storage Buffers

The sun4v translation storage buffers are slightly enhanced in comparison with the sun4u TSBs. On sun4v, multiple TSBs can be specified for a given context. Moreover, the TSBs can be *n*-way set associative, where *n* is a number between 1 and 4, inclusive. The privileged software need not make any use of these enhancements; if only one TSB is installed for each context and if the TSB is defined as direct mapped[3], the privileged code managing the TSBs will be compatible with the one for sun4u.

The TSBs are set, as usual on sun4v, using hypervisor API calls. There are two hypercalls installing the TSBs – `MMU_TSB_CTX0` installs the TSBs for context number 0, `MMU_TSB_CTXNON0` installs the TSBs for context other than 0. For both hypercalls, TSBs are described by an array of structures called *TSB description*. The hypercalls take the size of the array and the pointer to the array start as its arguments. The TSB description encapsulates

- the size of the pages mapped,
- TSB associativity,
- TSB size,
- real address of the TSB.

5.3.2 MMU Miss Handling

MMU miss traps are hyperprivileged. This means that when an MMU miss occurs, a trap to the hypervisor is taken, not to the kernel. The hypervisor tries to find the mapping in the TSB (if installed for the trapping context). If the mapping is found in the TSB, hypervisor automatically installs the mapping into the TLB and restarts the instruction which caused the trap, without an intervention of the kernel. If the TSB is not installed for the trapping context or the mapping is not found in the TSB, hypervisor jumps to the kernel trap table to the kernel MMU miss handler[4].

[3]Direct mapped means '1-way set associative'.

[4]Note that even though the MMU miss is hyperprivileged, the privileged software can still define its handler. The difference from the sun4u architecture is that the privileged handler is not used right upon the miss, but only if the hypervisor fails to find the mapping in the TSB.

5.3.3 MMU Fault Status Area

MMU Fault Status Area is an in-memory structure to which, upon an MMU fault, the hypervisor[5] stores the description of the fault. The structure is partitioned into two parts, one part describing instruction memory faults, the other one describing data memory faults. For each virtual processor a separate MMU Fault Status Area is specified. MMU Fault Status Area encapsulates

- fault type (MMU miss, protection violation, invalid real address, invalid virtual address, unaligned access,...),
- fault address,
- fault context.

Privileged software may specify the location of the MMU Fault Status Area using a hypervisor API call `MMU_FAULT_AREA_CONF`.

5.4 Traps

5.4.1 Privileged and Hyperprivileged Traps

The sun4v architecture recognizes two types of traps – *privileged* and *hyperprivileged*. The trap can be delivered either to the privileged or the hyperprivileged software, based on the type of the trap and the privilege level at which the trap occurred.

trapping privilege level	**trap type**	**delivered to**
non-privileged	privileged	privileged SW
non-privileged	hyperprivileged	hyperprivileged SW
privileged	privileged	privileged SW
privileged	hyperprivileged	hyperprivileged SW
hyperprivileged	privileged	hyperprivileged SW
hyperprivileged	hyperprivileged	hyperprivileged SW

Table 5.1: Trap Handling

5.4.2 Trap Levels

Trap levels have been introduced is Section 3.2.1. On sun4v processors there are *MAXTL* trap levels above level 0, where *MAXTL* is an implementation-dependant number[6]. The number of trap levels above zero at which the *privileged* software can run is *MAXPTL*, where *MAXPTL* is an implementation-dependant number[7]. Since *MAXTL* > *MAXPTL*, above *MAXPTL* there are always several trap levels at which only a hyperprivileged software can run. Should the trap occur in privileged code at level *MAXPTL*, the trap will be always delivered to the hypervisor.

[5]Or it can be the processor itself, if implemented so.

[6]It can be a number between 4 and 7.

[7]It can be a number between 2 and 6 and it is always less than *MAXTL*.

Global Registers

On 64-bit SPARC processors there are always multiple sets of global registers, one set being active when no trap is being processed, and a few other sets, one of which is active when a trap is being processed.

On the *sun4u* processors each alternative set is associated with the *type of the trap*: there is an alternative set for MMU traps, alternative set for interrupt traps and alternative set for other traps.

On the *sun4v* processors each alternative set is associated with the *trap level*, more precisely, the active global registers set is determined by the value of the `GL` register. Conventionally the value of the `GL` register is kept the same as the value of the `TL` register.

Higher Global Sets not Preserved

Due to the seamless activity of the hypervisor, the privileged software may never rely that the global registers from a greater level than the current one will be preserved. For instance, if the privileged software currently running at level `TL` = 0 writes a value to a global register from level `GL` = 1, the value may be lost any time. The reason is that during the MMU miss a hyperprivileged handler will be invoked (see Section 5.3.2) on `TL` = 1, thus using global set at `GL` = 1, potentially overwriting the value the privileged software stored there.

5.5 Miscellaneous

5.5.1 Scratchpad Registers

The UltraSPARC Architecture 2005 specification defines a set of eight privileged ASI-accessible registers which can be used for arbitrary purpose by the software. Four of them are guaranteed to be accessible by the privileged software. They are called *scratchpad registers*. All scratchpad registers are accessed using the same ASI, but a different virtual address.

The first scratchpad register is conventionally used to store the ID of the processor. The second scratchpad register is conventionally used to store the real address of the MMU Fault Status Area. Generally, the scratchpad registers are read in performance-critical trap handlers, where the number of instructions executed matters.

5.5.2 Some Hyperprivileged Registers

Some registers which are privileged in the sun4u architecture have been made hyperprivileged in the sun4v architecture.

`TICK` Register

The ticked register can be *read* from the *privileged* software, but *writing* to it is *hyperprivileged*. Hence, the system may not set this register to zero when initializing the timer. Instead, it must add the value of the `TICK` register at the time of timer initialization to the initial value of the `TICK_COMPARE` register.

`VERSION` register

The `VERSION` register cannot be read from on sun4v from the privileged code. Therefore, there is no way the privileged software could detect the particular processor model. Since the privileged software is isolated from the processor thanks to the hypervisor, information about the particular model is not that important for the privileged software.

Chapter 6

Original UltraSPARC HelenOS Port

This chapter briefly describes the original port of HelenOS onto the 64-bit SPARC processors. The original port, created by Jakub Jermář, supported UltraSPARC I, UltraSPARC II and UltraSPARC IIi processors. This chapter focuses on those aspects of the original port whose knowledge is essential for understanding the following two chapters: Chapter 7 describes the enhancements made to the original port to run HelenOS on the JPS-compliant processors, Chapter 8 describes enhancements made to the original port to run HelenOS on sun4v processors. For details on the original port see [jj_thesis].

6.1 Boot Process and Kernel Startup

The product of the HelenOS build process is a binary file, called `image.boot`, which is referred to as *bootable image*. The bootable image encapsulates

- kernel image,
- initial tasks (processes),
- initial RAM disk, and
- a 'small loader program', whose main purpose is to place the previous three components to a well-defined location in memory, and jump to the first instruction of the kernel image.

The bootable image is sufficient for booting HelenOS over network – the firmware loads the bootable image via the TFTP protocol and jumps to the first instruction of the 'small loader program'.

6.1.1 Bootable CD

For booting HelenOS from a disk (e.g. a CD-ROM), bootable image is not sufficient. A *bootable disk* image must be created. A bootable disk is a formated disk with a standard filesystem (ext2, ISO-9660) containing a special sector called *boot sector*. On the disk filesystem the bootable image is located. The boot sector contains a bootloader, a third party program, whose purpose is to pick up the bootable image from the disk's filesystem, load it to the memory and jump to its first instruction (in our case the first instruction of the 'small loader program').

The third party bootloader used in HelenOS bootable disk is called SILO. Basic information about SILO can be found at `http://silo.auxio.org`. The bootable CD is created using the mkisofs utility. The utility takes these parameters:

- binary file which will be copied to the boot sector, and
- a directory tree which will be copied to the bootable disk filesystem.

The former is a file distributed along with SILO, called `isofs.b`, the latter is a directory tree containing all the SILO binaries and the HelenOS bootable image.

6.1.2 Bootloader

This section describes the 'small loader program' mentioned in the previous section. From now on it will be referred to as *bootloader*.

The main purpose of the bootloader is to relocate the kernel, initial tasks and the RAM disk image to a well defined location, to make a copy of the OBP tree, and to jump to the first instruction of the kernel. The reason for relocating is that SILO has loaded the bootable image to address 0x4000, but the kernel expects to be located at 0x400000[1]. The reason for copying the OBP tree is that the HelenOS kernel will take a complete control over the machine, disabling the OBP, hence the kernel will have no chance to access the OBP tree via the OBP client interface. Since the HelenOS kernel will need some information contained in the OBP tree, a copy of the tree is made before starting the kernel.

Since the OBP resides in the same virtual address space as the bootloader, the bootloader needs to *claim* a piece of memory from the OBP. Claiming the memory means telling the OBP that the memory will be used by the bootloader and that OBP should not use it. HelenOS bootloader claims a block of memory starting at 0x400000 and big enough to incorporate kernel, init tasks, initial RAM disk, copy of the OBP tree and helper structures. On the UltraSPARC I, II and IIi processors claiming such a block of memory always succeeds.

One of the helper structures passed from the bootloader to the kernel is a *memory map*, which contains information about the staring address and size of each memory segment of the physical memory. The memory map is actually redundant, as the kernel could derive it from the copy of the OBP tree, but it is passed anyway for kernel's convenience.

Another important task of the bootloader is to wake the application (non-bootstrap) processors up.

6.1.3 Kernel Initialization

The basic thing the kernel must do is take over the TLBs and the trap table – so far the TLBs and the trap table have been managed by the OBP. After the takeover the OBP will be completely disabled, the traps will be handled purely by the HelenOS kernel and the kernel will have a complete control over the memory management.

[1] Address 0x400000 is nice, given it is the starting address of the second 4MB page. The kernel image is smaller than 4 MB, so if the kernel locks the second 4MB page in the TLB, the kernel code will be always mapped. The first 4MB page is intensionally left unlocked: if it was locked, then virtual address 0 would always be mapped and there would be no way to detect dereferencing a null pointer.

TLBs Takeover

During the takeover the kernel will remove any existing mappings from the TLB and install a locked entry mapping the second 4MB page – i.e. the page containing kernel code and data. Virtual address 0x400000 will be mapped onto a physical address C + 0x400000, where C is the starting address of the physical memory[2]. The takeover is performed in two stages: first, the DMMU is taken over, then the IMMU is taken over.

Taking over the DMMU is straightforward. The DTLB is invalidated and a locked entry mapping the second 4MB page is installed in context 0.

Taking over the ITLB is not that straightforward. The problem is that the ITLB is mapping the code being executed. The problem is tackled as follows: First, an entry mapping the second 4MB page is inserted in context 1. Then, the code switches to context 1. From context 1, it invalidates all entries in context 0 and installs a mapping for the second 4MB page in context 0. Then the code switches back to context 0.

8kB Page TTE Template

During the startup phase the kernel initializes a global 64-bit variable called `kern-el_8k_tlb_data_template`. The variable is a template of a valid 8-kB cacheable writable privileged translation table entry. When installing a mapping for the kernel in the MMU miss handler, the faulting page number is added to the template and the result is inserted to the TLB.

6.2 Preemptible Trap Handler

Preemptible Trap Handler is a routine which is called upon some traps[3]. It makes arrangements for calling a higher level service routine. Thanks to the preemptible trap handler the higher level service routine may be executed on trap level 0, the routine may be written in the C language, and it may call the scheduler. The preemptible trap handler ensures that the global registers and the valid register windows will not be corrupted and that the handling of the trap will be fully transparent for the thread which caused the trap. The design also ensures that the maximum number of trap levels will not be exceeded.

The handler is implemented as an assembly language macro. The handler expects the address of the higher level service routine to be in the `%g1` register and the argument of the service routine in the `%g2` register. The handler must be called from trap level 1. The exact steps performed by the handler depend on whether the trapping thread is a user thread or a kernel thread and whether the trap is a syscall or not.

6.2.1 Trapping from the Kernel

If the trapping thread is a kernel thread, the following steps are performed:

1. A new stack frame and new register window are allocated. If allocating the register windows causes a spill trap, the window will be spilled to the kernel stack.

[2]The starting address of the physical memory will usually be 0. On some machines, however, memory may start at non-zero addresses. An example situation when this may happen is that there is no dimm inserted into the first memory slot on the motherboard.

[3]The traps are: timer interrupts, hardware interrupts, MMU misses, syscalls, illegal operations.

2. Higher level service routine address is copied to the `%l0` register, its argument is copied to the `%o0` register.

3. The `TSTATE`, `TPC` and `TNPC` registers are backed up onto the kernel stack.

4. Trap level is set to 0.

5. Normal global register set is activated (i.e. the higher level service routine will not use alternate, interrupt nor memory globals).

6. Global registers `%g1` to `%g7` are backed up to local registers `%l1` to `%l7`. Registers `%l1` to `%l7` will not be corrupted, because the higher level service routine will allocate its own window.

7. Higher level service routine is called. Its address has been set to the `%l0` register in step 2.

8. Actions from steps 3 to 6 are reverted: global registers are restored, alternate globals set is activated, trap level is set to 1, trap state registers are restored.

9. If the `CWP` has changed since the time before calling the service routine[4], the current register window will be relocated to its original position.

10. The window allocated in step 1 is deallocated and the instruction causing the trap is restarted.

6.2.2 Trapping from the Userspace

If the thread which caused the trap is a userspace thread, the behavior of the preemptible trap handler will be much more complex. The register windows which belong to the thread causing the trap cannot be spilled onto the stack, as the userspace stack may not be mapped in the TLB. Since the preemptible trap handler does not want to risk a nested MMU trap here, the userspace windows are spilled to so called *userspace window buffer*. The userspace window buffer is a structure which exists for each userspace thread, it is big enough to incorporate all the register windows[5]. The buffer belongs to the kernel address space (context 0), whose MMU misses will not cause a nested MMU trap (see Section 6.3.1).

Upon a context switch to thread T, the address of the top of T's userspace window buffer is stored into the `%g7` register of the alternate globals set. The `%g7` register is updated upon window spills to the userspace window buffer. Upon a preemption of thread T the `%g7` register is snapshotted so that it can be re-set next time T is switched to again.

The exact steps the handler must do when the trap comes from userspace are as follows (the description assumes the trap is not a syscall):

[4]If the scheduler has not been called from the higher level service routine, the number of function calls and returns from function must be equal, so the `CWP` must be the same. This means that the `CWP` could only change if the scheduler has been called in the service routine. During the context switch in HelenOS, all register windows except the active one are always spilled to stack, so if the `CWP` changed, it means that there is *just one* active window.

[5]Given each register window contains 8 input and 8 local registers, there can be at most 7 valid register windows and each windowed register is 8 bytes long, the size of the buffer must be (8 + 8)*7*8 bytes.

1. The `WSTATE` register is configured such that all normal window spills will go to the userspace window buffer.
2. A new stack frame and new register window are allocated. If allocating the register windows causes a spill trap, the window will be spilled to the userspace window buffer.
3. All windows which belong to the trapping thread are marked as *other*, i.e. the value of the `OTHERWIN` register is set to the value of the `CANRESTORE` register.
4. Context 0 is set as a primary context (so far the context of the trapping thread has been the primary context) before leaving nucleus.
5. The `WSTATE` register is configured such that all normal window spills will go to the kernel stack and all other window spills will go to the userspace window buffer.
6. Service routine address and arguments are copied to the `%l0` and `%o0` registers, trap state registers are backed up onto a kernel stack, trap level is set to 0 (nucleus is left), global variables are saved.
7. The service routine is called.
8. Global variables are restored, trap level is set to 1, trap state registers are restored, the current register window is relocated to the location where it was before the trap (if needed).
9. The primary context register is set to the ID of the userspace context of the thread which caused the trap.
10. Register windows which have been spilled onto the userspace window buffer are restored.
11. The register window allocated in step 2 is deallocated and the instruction which caused the trap is restarted.

As for the syscalls, the handler is very similar. Instead of calling a higher level service routine, a syscall handler is called (which can take more parameters than just one). In the end, the instruction which caused the trap is not restarted, but the control is transferred to the subsequent instruction (instead of `RETRY`, `DONE` is issued).

6.3 Memory Management

6.3.1 Handling MMU Misses

The MMU miss handler behaves in a different way depending on whether the miss was caused by a kernel or a userspace thread.

Handling Kernel MMU Misses

If an MMU miss trap occurs in the kernel, the trapping page with virtual address *VA* is mapped onto a physical address *PA* = *VA* + *C*, where *C* is the starting address of the physical memory. The kernel MMU miss handler is written in the assembly language and its code fits into 32 instructions, so the handler fits into one trap table entry and branching is not needed (see Section 3.2.2). The handler does not use any higher level service routine when installing the mapping for the kernel, so the preemptible trap handler is not called.

Handling Userspace MMU Misses

If the MMU miss trap occurs in userspace, the handler first tries to find the mapping in the TSB. If the mapping is not found in the TSB, a higher level service routine is called (via the preemptible trap handler). The service routine finds the translation in the HelenOS translation table and installs the mapping to the TLB and TSB. The service routine takes the value of the MMU Tag Access Register as its argument.

When handling userspace data MMU miss, a certain anomaly may occur. Imagine a userspace thread issues a `SAVE` instruction. If `CANSAVE` equals 0, a spill trap is taken and the trap level is set to 1. The spill handler tries to save one of the register windows onto a userspace stack. If the stack is not mapped in the TLB, data access MMU miss occurs, in which case the trap level is set to 2. If the mapping is not found in TSB, a higher level service routine must be called via the preemptible trap handler. But the preemptible handler expects to be called from trap level 1, not 2!

In such a case we just lower the trap level to 1, pretending the MMU trap is not nested in a window spill trap, but that it is a standalone MMU trap which occurred on trap level 0. Then the higher level service routine is called (via the preemptible trap handler). Once the MMU trap is resolved, the `SAVE` instruction is restarted. It (again) causes a spill trap (as the spill trap has not been handled yet), but this time spilling the window succeeds, since the userspace stack is already mapped.

6.3.2 Caches and the Illegal Aliasing Problem

Illegal alias is an undesired state of page mappings which the system programmer must be aware of and which is a little tricky to avoid. *Alias* is a virtual-to-physical mapping which maps two virtual addresses *v1* and *v2* (belonging to the same virtual address space) onto the same physical address *p*. The next two paragraphs aim to explain what is meant by the term *illegal alias*.

Caches on the UltraSPARC I, II and IIi processors are *virtually indexed*, which means that the index where the entry will be placed in the cache is computed from the virtual address. They are *physically tagged*, which means that a physical address is used to recognize a hit. The 14 least significant bits of the address form an index to the cache; since the caches are direct mapped, the size of the cache is 2^{14} = 16 kB entries, twice as much as the size of an 8kB page. The pages whose bit 13 is 0 are depicted gray and the pages whose bit 13 bit is 1 are depicted black in Figure 6.1. In figure *(a)* the value 0xAA is saved on a physical address mapped by two virtual addresses. The value is cached at two different places of the cache (because the cache index is computed from the virtual address and the virtual addresses have different colors); this state is, however, illegal!

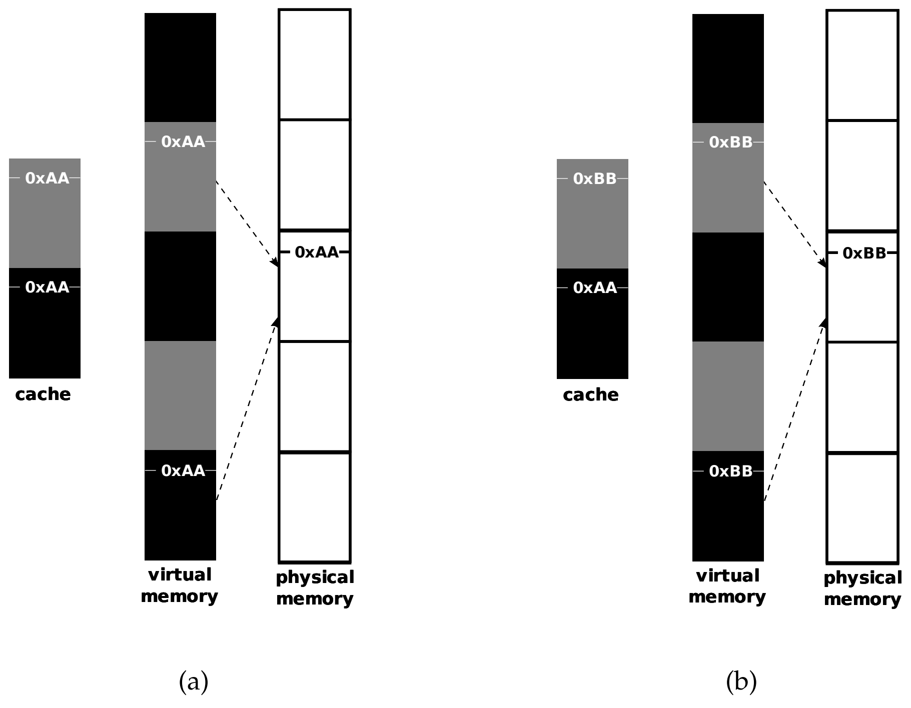


Figure 6.1: Illegal Alias

When a new value 0xBB is written to the first ('gray') virtual address, the value does not get propagated to the second ('black') entry in the cache. During the subsequent read from the 'black' virtual address the cached value will probably be used, hence the out-of-date value will be read.

HelenOS avoids illegal page mappings by emulating 16kB pages – when installing a new virtual-to-physical mapping, two neighboring 8kB pages are mapped concurrently (the first page having 'gray' color, the second one having 'black' color) onto physical frames with the same colors. Thus it may never happen that two pages of different colors get mapped onto the same physical frame, thus it may never happen that there are two entries in the cache caching value from the same physical address.

6.3.3 Other Memory Management Issues

Context Switch and the Secondary Context Register

The secondary context register is written to by a higher level service routine which performs the context switch. It is used to temporarily hold the context ID which will be copied to the primary context register upon jumping to userspace. The reason why the context ID is not written to the primary context register directly is that the context switch is performed by a higher level service routine on trap level 0 (i.e. not *nucleus*), thus writing to the primary context register would cause an immediate switch of memory context and the handler's memory would cease to be mapped. Jumping to userspace is performed from nucleus, so changing the value of the primary context register from there is safe.

Translation Storage Buffers

The translation storage buffers for a memory context take up 64 kB, thus fit into exactly one 64kB page. The page is split into two equally-sized parts, the first part holds the ITSB, the second one holds the DTSB. When the context the TSBs belong to is active, the TSB page is locked in the TLB, so accessing the TSBs never causes a nested MMU trap.

Kernel mappings (context 0) are not included in the translation storage buffer. Context 0 contains only identity mappings (or identity with displacement, if the physical memory starts at non-zero address) so the usage of the TSB makes little sense.

Locked Entries

On UltraSPARC I, II and IIi the TLB can contain 64 locked entries. HelenOS usually uses only a small amount of locked entries – the kernel is locked in the 4MB page and the active TSB is locked in the 64kB page.

6.4 Miscellaneous

6.4.1 Crosscalls

A *crosscall* is a kind of function call in which the caller runs on a different CPU than the callee. In HelenOS it is used for TLB shootdown[6]. The crosscalls are implemented using inter-processor interrupts (see Section 3.4.2). UltraSPARC I and II processors allow to pass three 64-bit values in interrupt vector data, but HelenOS uses only one 64-bit value: to pass the address of the function being called. The remaining two 64-bit values are unused.

6.4.2 Device Drivers

Information about the attached devices is obtained from the OBP tree. Each device is represented by a node in the OBP device tree. Among the node properties there are addresses and sizes of the device registers.

[6]When a mapping is modified or removed or when a virtual address space identifier becomes invalid on one CPU, (portions of) TLB must be invalidated on other CPUs as well.

Chapter 7

Porting HelenOS to JPS Processors

Enhancements which enable running HelenOS on UltraSPARC III, IIIi, III+, IV and IV+ processors were made by Pavel Římský in 2008 and 2009 as a part of his master thesis.

7.1 Overview

7.1.1 Supported Environments

HelenOS is able to run on two types of machines equipped with an UltraSPARC III series CPU: a Simics-simulated[1] Serengeti machine and a real SunBlade 1500 workstation. Serengeti is Sun Microsystems' mid-range server. It can contain up to 24 UltraSPARC III, III+, IV or IV+ processors. SunBlade 1500 is Sun Microsystems' workstation containing one UltraSPARC IIIi processor.

7.1.2 Enhancements Overview

Basically, two types of enhancements had to be done to make HelenOS run on JPS-compliant processors.

- Some registers have slightly different format and there are a few brand new registers. The formats of the registers had to be redefined in HelenOS sources. These changes are not very interesting, the fundamental part of the work was to carefully read the specifications and find out which differences affect HelenOS and how to cope with them.
- OpenBoot PROM tree layout is a little bit different and on some machines the firmware behaves in a very specific way. Even though the main goal was to add support for newer CPU models, the hardest work did not concern the CPUs, but the firmware. Especially the version of OBP found on the Serengeti machine has a lot of anomalies. This was further complicated by the fact that the firmware is very badly documented.

7.1.3 Limitations

HelenOS is able to run on a Simics-simulated Serengeti machine with up to four processor chips (and since the UltraSPARC IV and IV+ processors are dual-core, the max-

[1]Simics is a full system simulator.

imum number of CPUs is eight). The only reason why the number of CPUs is limited is that when the CPU on the fifth chip is being woken up via OBP, the simulator crashes. It occurs both on 3.0 and 4.0 versions of the Simics simulator. The reason is that the simulator probably does not implement *Instruction Breakpoint Register* (which is a JPS-mandatory ASI-accessible register) which the firmware (see Section 7.2) for some reason uses when waking up the CPUs.

Input and output is possible through a serial line on Serengeti. Theoretically a graphic card could be connected to the Serengeti machine; the simulator, however, is not able to simulate such a configuration reliably. Since Serengeti is a server machine, graphic output has only little importance.

HelenOS running on the SunBlade 1500 machine writes its output to the framebuffer. Input is not possible, since only a USB keyboard can be attached to the SunBlade 1500 machine and HelenOS has no USB support yet.

7.1.4 Integration with the Original UltraSPARC Port

In the kernel, the source code of the original port and the JPS port are mixed – typically one source file contains parts which are common for both CPU series, specific parts are enclosed in the `#ifdef` .. `#endif` block. Example 7.1 demonstrates how the code of the original port and the JPS port are separated.

```
/** TLB Demap Operation Address. */
union tlb_demap_addr {
  uint64_t value;
  struct {
    uint64_t vpn: 51;      /**< Virtual Address bits 63:13. */
#if defined (US)
    unsigned : 6;          /**< Ignored. */
    unsigned type : 1;     /**< The type of demap operation. */
#elif defined (US3)
    unsigned : 5;          /**< Ignored. */
    unsigned type: 2;      /**< The type of demap operation. */
#endif
    unsigned context : 2; /**< Context register selection. */
    unsigned : 4;          /**< Zero. */
  } __attribute__ ((packed));
};
typedef union tlb_demap_addr tlb_demap_addr_t;
```

Example 7.1: Separating Original and JPS Port Code

In the bootloader, the code of the original port and the JPS port are mixed too, but they are not separated by preprocessor directives. Instead, an autodetection is used to decide what actions are to be performed on the host CPU model. The autodetection is based on reading the value of the `VERSION` register, which tells the bootloader the exact host CPU model.

The autodetection enables the bootloader to be generic to some extent, so that it does not have to be recompiled every time the CPU model is changed. Since the kernel is much more complex and it contains far more CPU model-specific definitions, the preprocessor directives are used instead of the autodetection (which implies that the kernel must be recompiled every time the target CPU model is changed).

7.2 Serengeti and its Firmware

The Serengeti machine is equipped with an untypical version of the OpenBoot PROM. It is called COBP, because it is written mainly in the C language, while the standard versions of OBP are written in the Forth language. The firmware is based on the implementation called SmartFirmware implemented by the Codegen (`www.codegen.com`) company. The firmware affects mainly the way how HelenOS is booted. The version of the OBP on the referenced machine is 5.17.0.

7.2.1 SILO not Compatible with the Firmware

The basic problem is that SILO (version 1.4.11 is used) is not fully compatible with the Serengeti machine firmware. The Serengeti machine firmware supposes that in the bootable CD bootsector there will be a plain binary code. The `isofs.b` file, which is copied to the bootsector (see Section 6.1.1) is not a plain binary, but it contains an `a.out` header.

When booting from the bootable CD, without further provisions the following message would be printed instead of running SILO:

```
SmartFirmware, Copyright (C) 1996-2001. All rights reserved.
Boot path: /ssm@0,0/pci@19,700000/scsi@2/disk@6,0:f Boot args:
ERROR: Illegal instruction
debugger entered.
ok
```

The illegal instruction exception is actually caused by the CPU interpreting the first four bytes of the `a.out` header as an instruction.

The problem is solved by ripping out the `a.out` header from the `isofs.b` file before making the bootable CD (in other words, patching the SILO binaries). A cleaner solution would be to modify SILO source files and compile them along with HelenOS. The problem is that SILO is not cross-compilable (a SPARC machine is needed to compile SILO), whereas HelenOS is cross-compilable. Making SILO cross-compilable would be beyond the scope of the thesis.

The `isofs.b` file is patched using the `xdd` command line utility. The utility is able to convert given binary file to its hexadecimal representation and the binary representation back to the binary form. It is also able to skip a given number of bytes. The complete reference to `xdd` can be found in its manual pages.

A piece of code which downloads and patches the SILO binaries can be found in the `contrib/util/DownloadAndPatchSILO.sh` file. Ripping the `a.out` header from the `isofs.b` file is performed by this line:

```
(((xxd -p -l 512 isofs.b) && \
 (xxd -p -s 544 isofs.b)) | xxd -r -p) \
          > isofs.b.patched
```

The first 512 bytes are zero and they are kept zero even in the patched file. Bytes number 512 to 544 (the `a.out` header) are completely removed, so consequently the size of the file is decreased by 32 bytes.

7.2.2 Other Serengeti Firmware Properties

The anomaly mentioned in the previous paragraph is not the only property specific for the Serengeti firmware. This section briefly mentions the rest of the most important ones.

Constrained Memory to Claim

On standard (non-Serengeti) UltraSPARC machines OBP guarantees that it will restrict its usage of virtual memory such that the block of memory from address 8 kB to 10 MB will be free to be claimed by the bootloader. Thus the kernel image, init tasks, RAM disk and helper structures will easily fit into the memory which can be claimed.

Unfortunately, this is not the case of the Serengeti machine firmware. On Serengeti machine the firmware uses some bits of memory from address 0x600000 and higher, so the bootloader cannot claim more than 6 MB of contiguous memory which resides at the beginning of the address space.

Since HelenOS kernel does not use the OBP once booted, it may use the whole memory without claiming. During the boot phase, however, this feature of the firmware is very limiting. The bootable image, init tasks and the RAM disk must be kept sufficiently small. It is achieved by not compiling parts of code which are not necessarily needed (for example drivers of some devices which can not be attached to Serengeti), not including some tasks which are not essential or by using the TMPFS RAM disk instead of the FAT RAM disk. As soon as HelenOS has got a disk driver, there will be no need to use the RAM disk anymore and the problem will hopefully disappear.

It is also worth mentioning that SILO is not aware of the constrained memory problem. As a consequence, SILO can not be given the RAM disk in a separate file, otherwise it would try to load the RAM disk to the part of the memory which it could not claim. Therefore, the RAM disk is supplied in the same file as the bootable image (which SILO, fortunately, puts to the memory it can claim).

OBP Tree Layout

There are small differences in the OBP tree layout of the pre-JPS processors and the JPS processors which HelenOS must be aware of.

On non-Serengeti machines the CPU nodes are children of the root node, on Serengeti, they are children of the '/ssm' node[2]. The type of the machine is detected by looking whether the '/ssm' node is present. If so, the CPU nodes are looked up in the '/ssm' node, if not, they are looked up directly in the root node. The following code snippet comes from the `kernel/arch/sparc64/include/cpu_node.h` file:

```
parent = ofw_tree_find_child(ofw_tree_lookup("/"), "ssm@0,0");
if (parent == NULL)
    parent = ofw_tree_lookup("/");
```

As a curiosity, let us mention one anomaly of the Serengeti OBP client interface: if the client interface is asked to return all children of the root node, it surprisingly does not return '/ssm'. In HelenOS this is fixed by the bootloader by explicitly sticking the '/ssm' node under the root node of the OBP tree copy.

[2]SSM is a cache coherency protocol used on machines with a large number of CPUs.

Apart from the 'ssm' node, another difference in the OBP tree layout is the different name of the processor ID property as described in Section 4.3.1. In this case the code just tries all the three possibilities ('portid', 'upa-portid' and 'cpuid') and tests, which one is valid for the given OBP tree. The following code snippet comes from `kernel/arch/sparc64/src/cpu/cpu.c`:

```
/* Read the "upa-portid" OBP property,... */
prop = ofw_tree_getprop(node, "upa-portid");

/* ...if it is not present, try "portid",... */
if ((!prop) || (!prop->value))
    prop = ofw_tree_getprop(node, "portid");

/* ...if it is not present either, try "cpuid". */
if ((!prop) || (!prop->value))
    prop = ofw_tree_getprop(node, "cpuid");
```

`WSTATE` Register not Set to 0

While on standard (non-Serengeti) SPARC-based systems the value of the `WSTATE` register is set to zero by the time the control is passed from OBP to the kernel, on Serengeti it is surprisingly set to 5. In the original HelenOS port explicitly setting the `WSTATE` register to 0 was not needed, but on Serengeti it is essential. The `WSTATE` register is cleared very early after the kernel is passed control from the bootloader by the following instruction:

```
wrpr %g0, 0, %wstate
```

7.3 Memory Subsystem

The physical address space is larger on JPS-compliant processors in comparison with the older UltraSPARC models[3], which affects many things. First some parts of the code where the size of the physical memory address space had been hard-wired had to be rewritten and the size defined as a symbolic constant: in the kernel startup routine in the `start.S` file a new constant `PHYSMEM_ADDR_SIZE` has been defined, which represents the number of bits of the physical space address. Additionally a huge number of registers' formats had to be redefined.

7.3.1 Translation Lookaside Buffers

There are multiple ITLBs and DTLBs in one JPS-compliant MMU and they are basically larger than those of the older UltraSPARC processors. The TLBs within one MMU are identified by *TLB numbers*, which are symbolic constants in HelenOS:

- `TLB_DSMALL` (16-entry data TLB), `TLB_DBIG_0` (the first big data TLB), `TLB_DBIG_1` (the second big data TLB), and
- `TLB_ISMALL` (16-entry instruction TLB) and `TLB_IBIG` (the big instruction TLB).

[3] It uses 43-bit addresses instead of 41-bit.

The number of entries is returned by functions

- `tlb_dsmall_size`, which returns the size of the small DTLB (always 16),
- `tlb_dbig_size`, which returns the size of the big DTLB (always 512),
- `tlb_ismall_size`, which returns the size of the small ITLB (always 16), and
- `tlb_ibig_size`, which returns the size of the big ITLB – reads the `VERSION` register and then returns 128 (if running on UltraSPARC III, IIIi or III+ CPU) or 512 (if running on UltraSPARC IV or IV+ processor).

The sun4u processors enable a direct diagnostic access to the translation lookaside buffers. The registers for a direct access to the TLBs are ASI-accessible and their virtual address encodes the TLB number and the entry number (on the pre-JPS processors it encapsulated only the entry number).

HelenOS contains a handful of functions which directly access the TLBs. These function have been modified to reflect the TLBs on the JPS processors – they take a TLB number as an additional argument.

The original headers looked like this:

```
/* reading "data" portion of the ITLB entry */
static inline uint64_t itlb_data_access_read(index_t entry)

/* writing "data" portion of the ITLB entry */
static inline void itlb_data_access_write(
    index_t entry, uint64_t value)

/* ... */
```

In the JPS port they had to be adapted in this way (the TLB number was added):

```
static inline uint64_t itlb_data_access_read(
    int tlb, index_t entry)
static inline void itlb_data_access_write(int tlb, index_t entry,
    uint64_t value)

/* ... */
```

In the JPS port the body of the already mentioned functions contains (apart from the code borrowed from the original port) encoding the TLB number to the virtual address of the TLB register:

```
/* reg contains the virtual address */
reg.value = 0;

/* encode the TLB number (JPS only) */
reg.tlb_number = tlb;

/* encode the entry number */
reg.local_tlb_entry = entry;

/* write to the TLB register via the virtual address */
asi_u64_write(ASI_ITLB_DATA_ACCESS_REG, reg.value, value);
```

These functions are called from withing functions for invalidating the TLB entries and from a function for printing the contents of the whole TLB, which have been obviously modified too. Functions for invalidating the contents of the TLB must now invalidate entries in all the TLBs (while in the original port just one – the only present – TLB had to be invalidated). Analogously, the function for printing the contents of the TLB now prints the contents of all the TLBs. See the `kernel/arch/sparc64/src/-mm/tlb.c` source file for more details.

TLB Demap

In the HelenOS SPARC port a function called `tlb_invalidate_all` is defined to invalidate all the unlocked entries of the TLB.

As mentioned in Section 4.2.2, the JPS-compliant processors' demap operation is able to demap (apart from a particular mapping and a particular context) all the unlocked entries in the TLB. In the original port, demapping tho whole TLB was achieved by iterating through all the TLB entries using the registers for a direct diagnostic access and setting the 'valid' bit to `false` for all the unlocked entries. Demapping all unmapped entries has been optimized in the JPS port, now all the ITLB and DTLB entries are demapped by issuing a single operation.

7.3.2 Caches

Illegal Aliases

The illegal alias problem described in Section 6.3.2 appears on JPS processors as well – caches on JPS processors are virtually indexed too. The size of the cache is 64 kB, but the cache is 4-way set associative, which means that the index to the cache has the same size as on the older UltraSPARC processors. This means that exactly the same solution – emulating 16kB pages – can be used.

SMC Functions Optimization

SMC stands for *self-modifying code*. As mentioned in Section 4.2.4, on older UltraSPARC processors the `FLUSH` instruction had to be issued whenever the code modified the instruction stream, in order to synchronize the instruction cache with the data cache. The JPS processors keep the caches synchronized automatically, so the `FLUSH` instruction is only needed to flush the pipeline. The sparc64-specific functions which are called whenever the instruction stream is modified have been optimized in the JPS HelenOS port.

Before the change, each word of the piece of code which was modified had to be flushed:

```
/*
 * Synchronizes data and instruction caches for
 * memory block starting on address "a" and
 * being "l" bytes long.
 */
for (i = 0; i < l; i += FLUSH_INVAL_MIN)
    flush((void *) a + i);
```

Now on the JPS-compliant CPUs only the pipeline is flushed:

```
flush_pipeline();
```

Since flushing the pipeline takes the same amount of time as flushing a single instruction, if a block of length *L* is flushed on a JPS-compliant processor, the optimized code is *L* times faster than without the optimization.

7.3.3 Miscellaneous

Locked Entries

Locked entries can be contained only in the small TLB, which implies that the maximum number of locked entries on JPS processors is 16. This is not a big problem for HelenOS, because HelenOS uses only a very limited amount of locked entries (see Section 6.3.3).

The maximum number of locked entries in the data TLB is determined by the `DTLB_MAX_LOCKED_ENTRIES` constant, which is `#define`d as `DTLB_ENTRY_COUNT` for the older UltraSPARC CPUs and as `16` on the JPS-compliant CPUs.

Non-contiguous Physical Memory

Not only that the physical memory can start at non-zero addresses (see Section 6.1.3), but its address space may be discontinuous, i.e. 'contain holes'. On the SunBlade 1500 machine on which the author of this thesis tested the port it has been observed that the first block of the physical memory starts at address 0 and it is 2 GB large, and the second block of physical memory starts at address 8 GB and it is also 2 GB large. Physical addresses from 2 GB to 8 GB are not assigned to any physical memory, so writing to them causes an exception.

The SunBlade 1500 which HelenOS has been tested on is the only supported machine with such a weird memory layout.

As for now, HelenOS just ignores the second block of memory and uses only the first one. Port to the SunBlade 1500 machine has been meant only as a proof-of-concept implementation whose purpose was to show that the changes made to the code are not Serengeti-specific, but generic in the UltraSPARC III-world, so solving the non-contiguous memory issue has only a little importance.

TSB Extension Registers

The TSB extension registers (see Section 4.2.3) are set to zero. This means that the TSB code of the original UltraSPARC HelenOS port can be reused.

Usage of the nucleus extension register makes no sense, because in the nucleus the context 0 is active; context 0 (in HelenOS) bypasses the TSB.

Usage of the secondary context register makes no sense either, because in HelenOS there is no piece of code where two virtual address spaces are accessed alternately, both address spaces having their mappings in the TSB. In the `memcpy_from_userspace` and `memcpy_to_userspace` functions the code accesses both the kernel address space (context zero) and the userspace address space (secondary context), but the TSB is not used for the kernel so the TSB Base Register may contain the base address of the userspace TSB. Hence in HelenOS there is no need to keep track of the TSB base addresses for both the primary and the secondary context at the same time.

Functions for reading and writing to the extension registers have been implemented anyway, so if, in the future, someone decides to redesign HelenOS, the extension registers will be ready to use.

7.4 Processors

This section describes things related to CPUs, such as inter-processor interrupts, `TICK` interrupts, processor identification and processor models.

7.4.1 Interconnect Bus Configuration Register

Older UltraSPARC processors and the JPS-compliant processors use different models of the interconnect bus; the bus is called *UPA* on the older UltraSPARC processors and *Fireplane* on the newer UltraSPARC processors. From the kernel developer point of view the interconnect bus is not very important, since the kernel only uses one of the bus registers (UPA/Fireplane bus configuration register) to read the MID of the processor. Anyway, in the original UltraSPARC HelenOS port the constant for the interconnect bus configuration register ASI was called `ASI_UPA_CONFIG`, which has been changed to a more general name `ASI_ICBUS_CONFIG`.

7.4.2 Inter-processor Interrupts

Interrupt vector is composed of eight 64-bit registers, on older UltraSPARC processors it was composed of three 64-bit registers.

As well as in the original UltraSPARC HelenOS port (see Section 6.4.1), only one of these registers is used.

Constants defining the virtual addresses of the eight registers have been defined anyway, so the code is prepared for further extension. The constants are defined in the `kernel/arch/sparc64/src/smp/ipi.c` file and they are called `VA_INTR_W_DATA_0` to `VA_INTR_W_DATA_7`.

7.4.3 Timer Interrupts

JPS defines a new register: `STICK` (see Section 3.4.1). It is analogous to the `TICK` register, but unlike `TICK`, it is incremented at the rate determined by the external clock signal rather than the CPU clock.

HelenOS makes no use of the `STICK` register; it must be, however, aware of it. The kernel must, upon timer initialization, disable `STICK` interrupts and clear any pending ones, because the OBP may have used the `STICK` and `STICK_COMPARE` registers. Without doing so, it could happen that the value of the `STICK` register would become equal to the value of the `STICK_COMPARE` register, triggering a spurious tick interrupt.

Disabling the `STICK` interrupts and clearing the pending ones is done in the `tick_init` function in the `kernel/arch/sparc64/src/drivers/tick.c` file, right after the initialization of the tick interrupts.

7.4.4 CPU Models

HelenOS supports a large set of different UltraSPARC-based CPU models. There are not many differences between the particular CPU models the operating system must be aware of. Some of the few exceptions are the width of the `MID` register, TLB size or OBP layout. The CPU model is detected by inspecting the `VERSION` register.

Bootloader

In the bootloader, the model is determined in the `detect_subarchitecture` function in the `main.c` file. The `detect_subarchitecture` function reads the value of the `VERSION` register. First, it decides whether the model it is running on is one of the older models (UltraSPARC I, II, IIi) or a JPS-compliant model (UltraSPARC III, IIIi, III+, IV, IV+). For the older models, it sets the value of the `subarchitecture` variable to the `SUBARCH_US` value. For the JPS-compliant models, it sets the value of the `subarchitecture` variable to the `SUBARCH_US3` value. For the JPS-compliant processors it also checks whether the CPU model is UltraSPARC IIIi. If so, the `MID` register mask is set to (1 « 5) - 1, because on UltraSPARC IIIi the `MID` register is 5 bits wide. For the pre-JPS processors the `MID` mask is set to (1 « 5) - 1 as well. On the JPS processors other than UltraSPARC IIIi the `MID` mask is set to (1 « 10) - 1, because the `MID` is 10 bits wide on these processors.

In the bootloader, the information about the CPU model is further used from the assembly language routine which jumps to the kernel. Before jumping to the kernel, the instruction cache must be invalidated. Since invalidating the cache is an expensive operation and it is not needed on the JPS processors, it is skipped if the `subarchitecture` variable equals `SUBARCH_US3`.

The `MID` mask is used when waking up the application processors. In order to skip waking up the current processor, we must know the `MID` of the current processor. The `MID` is obtained by ANDing the value of the ASI-accessible `FIREPLANE_CONFIG` register with the `MID` mask.

Kernel

In the kernel, C language functions are defined:

- `is_us`, which returns `true` if and only if the CPU's model is UltraSPARC I, II or IIi,
- `is_us_iii`, which returns `true` if and only if the CPU's model is UltraSPARC III, IIIi or III+, and
- `is_us_iv`, which returns `true` if and only if the CPU's model is UltraSPARC IV or IV+.

The information returned by those functions is used when traversing the OBP tree copy in order to detect the number of CPUs, detect CPU frequencies or wake the application processors up. The most important issue is that the code must know whether the host processors are single or dual-core. As only the UltraSPARC IV and IV+ processors are dual-core, the code for detection of number of CPUs, detecting CPU frequencies and waking application processors up is slightly different for `is_us_iv` returning `true` and `is_us_iv` returning `false`. For more details on detecting the number of

CPUs, detecting their frequency and waking the application processors up, see Section 7.4.5.

To state the width of the `MID` register or the TLB size, a finer granularity of the CPU model detection is needed, because the width of the `MID` register is 5 bits on UltraSPARC IIIi, but 10 bits on UltraSPARC III. Since such a granularity is not needed on many places of the code, no special functions have been written for such purpose and the `VERSION` register is inspected directly at the place where it is needed. This is the case of the `tlb_ibig_size` function, which returns the size of the big instruction TLB (see Section 7.3.1) and of the `read_mid` function, which reads the MID of the current CPU (see the `kernel/arch/sparc64/include/sun4u/cpu.h` file).

7.4.5 Dual-core UltraSPARC IV Processors

UltraSPARC IV and UltraSPARC IV+ are the first SPARC-based processors which are dual-core. Because of that, the original UltraSPARC port (see Chapter 6) did not support dual-core processors at all and the support had to be added as a part of the JPS port.

From the system programmer's perspective, nothing special must be done to make HelenOS run on dual-core processors, the situation is almost the same as for the conventional multiprocessors. Only detecting the number of CPUs, detecting their frequency and waking the application processors up is affected by the difference in the OBP layout.

Detecting the Number of CPUs

The number of CPUs is detected in the `smp_init` function. The algorithm is slightly different for single-core and dual-core processors, due to differences in the OBP layout.

If HelenOS is running on a single-core CPU, it iterates through all the 'cpu' nodes and for each node it increases the counter of the CPUs by one, just as in the original port.

If HelenOS detects that it is running on an UltraSPARC IV or IV+ CPU, it iterates through all the 'cmp' nodes and for each 'cmp' node it increases the counter by two.

Detecting CPU Frequencies

CPU frequencies are detected in the `cpu_arch_init` function. Again, the algorithm is slightly different for single-core and dual-core processors. Algorithm for the single-core is the same as in the original port.

If HelenOS detects that it is running on an UltraSPARC IV or IV+ CPU, it iterates through all the 'cmp' nodes. For each 'cmp' node N it detects frequencies of CPUs represented by N's children called 'cpu@0' and 'cpu@1'.

Waking Processors Up

CPUs are woken up in the `kmp_init` function. Again, the algorithm is slightly different for single-core and dual-core processors. Algorithm for the single-core is the same as in the original port.

If HelenOS detects that it is running on an UltraSPARC IV or IV+ CPU, it iterates through all the 'cmp' nodes. For each 'cmp' node *N* it wakes up the CPUs represented by *N*'s children called 'cpu@0' and 'cpu@1'.

7.5 Device Drivers

For the Serengeti machine a driver of the serial input and output has been written. For the SunBlade 1500 machine a framebuffer driver had to be adapted slightly to work correctly.

7.5.1 SunBlade 1500 Framebuffer

In the SunBlade 1500 machine a graphics device called SUNW,XVR-100 is installed. The device has almost the same properties as the SUNW,m64B device, which had already been supported by HelenOS. Only two small changes had to be made.

- The driver had to be made aware that a device called SUNW,XVR-100 exists, so that it is able to recognize the device in the OBP tree.
- On most framebuffers the starting address of the framebuffer memory (as read from the OBP tree property) corresponds to the first pixel (left top corner). This is not the case of the SUNW,XVR-100 device, whose first pixel corresponds to an address *S* + 0x8000, where *S* is the starting address of the framebuffer memory. Addresses *S* to *S* + 0x7fff are valid, but writing to them has no effect.

Debugging Using Stripes

As a curiosity, let us mention that drawing to the framebuffer was, at some stage, the only way of debugging the HelenOS port on the SunBlade machine. If the kernel crashes before the output gets initialized, it is not possible to use `printf` to identify the exact point of failure. With some help of the OpenBoot PROM, however, it is possible to find the physical address of the framebuffer and draw something to the screen by storing a value to the framebuffer.

If there was a function `f` which was suspected of causing a failure, a piece of code which was drawing a stripe onto the screen was added to several places of the function body. By counting the number of stripes having been drawn it was possible to localize the place of failure inside the body of function `f`.

A drawback of this solution was that it did not work when `f` was (directly or indirectly) recursive – if `f` failed in the nested call but succeeded earlier in the call chain, all the stripes were drawn and it seemed as if `f` had not failed at all. This was fixed by introducing a static variable incremented each time `f` was called. The stripes were then drawn to different areas of the screen, depending on the value of the static variable. The nested call of `f` was therefore drawing the stripes onto a different area of the screen than the call preceding in the call chain.

7.5.2 Serengeti Console

The *Serengeti console* (SGCN) is a means of input and output on the Serengeti machine. SGCN is a non-standard chip to which a keyboard and output device can be attached using a serial line. Since the SGCN chip is non-standard, its documentation is not publicly available. Nevertheless, a functional implementation of the SGCN driver for HelenOS has been achieved.

The implementation is based on reverse engineering of the analogous driver for OpenSolaris.

SGCN Memory

The operating system communicates with the Serengeti console via a shared memory.

Within the address space of one of the Serengeti PCI buses[4] an SBBC nexus[5] address space is embedded. In the OBP tree SBBC is not present, so its address cannot be computed by standard means. The SBBC starting physical address could be certainly determined by investigating the PCI bus. As HelenOS has no full-featured PCI driver for the UltraSPARC processors yet, the SBBC nexus address is hardwired to 0x63000000000 – that is the address where the SBBC is mapped on the Simics-simulated Serengeti system.

Within the SBBC nexus address space, an SRAM address space is embedded. SRAM is a shared memory used for communication between the operating system and some devices (such as SGCN). The offset of the SRAM address space within the SBBC address space is 0x900000 + n, where n is a number read from the 'iosram-toc' property of the '/chosen' OBP tree node.

At the beginning of the SRAM there is a *table of contents* – an array of structures describing all the subareas of the SRAM. Each structure contains a key – a string which determines the purpose of the subarea described by the structure – and an offset within the SRAM of the subarea. Examples of the keys are 'SOLCONS' (standing for 'Solaris console'[6]) or 'OBPCONS' (standing for 'OpenBoot PROM console'). As for HelenOS, the 'OpenBoot PROM console' subarea is used for communication with SGCN. The reason is that the OBP, before it was shot down by the HelenOS kernel, had done all the necessary setup and the 'OBPCONS' subarea is ready to use. Theoretically the 'SOLCONS' could be used as well, but that would require doing some additional setup, which would not be an easy task given the SGCN is not documented.

HelenOS SGCN Driver Initialization

First, the address of the IOSRAM is computed in the `init_sram_begin` function. The function reads the `iosram-toc` property of the `/chosen` OBP node. Then it adds the value read to the SBBC nexus address (which is hard-wired to 0x900000 as described in the previous section). The physical memory at the computed address is immediately mapped to the kernel address space using the generic HelenOS function `hw_map`. When the IOSRAM memory is mapped, its virtual address is saved to a global variable called `sram_begin`.

After that, the `sgcn_buffer_begin_init` function computes the address of the SGCN buffer. It first asserts that the IOSRAM contains a magic at its beginning. After

[4]There can be multiple PCI buses.

[5]SBBC nexus is a PCI device, found in Serengeti servers, used for communication between the kernel (or OBP) and the Service Processor. The Service Processor is a 'small computer' used to control the Serengeti server.

[6]Which indicates that the designers of Serengeti probably expected that Serengeti would never run anything else than Solaris.

that it browses the table of contents and looks up the entry with the 'OBPCONS' key. From the entry it reads the offset to the 'OBPCONS' buffer and adds the offset to the IOSRAM address. The result is an absolute address to the 'OBPCONS' buffer, which is saved to the global `sgcn_buffer_begin` variable.

SGCN Buffers

The OBPCONS subarea contains two buffers: an output buffer used by the driver to print characters and an input buffer used by the driver to read characters typed on the keyboard.

The state of the buffers is described by header present at the beginning of the subarea. In HelenOS the header is accessed via a C structure. The header contains these main fields:

IN read pointer

Index of the first character in the input buffer which has not been read by the driver yet.

IN write pointer

Index of the last character which has been written to the input buffer by the controller.

OUT read pointer

Index of the first character in the output buffer which has not been read by the controller yet.

OUT write pointer

Index of the last character which has been written to the input buffer by the driver.

The buffer is addressed in a circular manner, i.e. when a character was written at the last index, the next character will be written at index 0.

Accessing the Buffer from HelenOS

A structure called `sgcn_buffer_header_t` has been defined, which represents the SGCN buffer header, as described in the previous section. For convenience, some macros have been `#define`d by means of which the buffer header or the buffer itself can be accessed. The macros compute the address within the buffer and convert the result to the pointer of a desired type.

Macro `SGCN_BUFFER` uses the value of the `sgcn_buffer_begin` which was computed during the HelenOS SGCN driver initialization:

```
/*
 * Returns a pointer to the object of a given type
 * which is placed at the given offset from the
 * console buffer beginning.
 */
#define SGCN_BUFFER(type, offset) \
    ((type *) (sgcn_buffer_begin + (offset)))
```

Upon this macro the `SGCN_BUFFER_HEADER` macro builds:

```
/** Returns a pointer to the console buffer header. */
#define SGCN_BUFFER_HEADER
    (SGCN_BUFFER(sgcn_buffer_header_t, 0))
```

The `SGCN_BUFFER_HEADER` macro is used when accessing the input/output read-/write pointers:

```
/* determines the size of the buffer */
uint32_t begin = SGCN_BUFFER_HEADER->out_begin;
uint32_t end = SGCN_BUFFER_HEADER->out_end;
uint32_t size = end - begin;
```

Reading from the Buffer

Since implementing the SGCN driver is not a focal point of this thesis, reading from the SGCN input is implemented in a simpler way: using polling, not the interrupts.

A background thread regularly checks whether the *IN read* and *IN write* pointers differ. If so, it reads all characters from the *IN read* position up until the *IN write* position and updates the *IN read* pointer.

Writing to the Buffer

When the SGCN driver would like to print a character, it first computes the index in the output buffer to which to store the character code. The index is

$$new\_ptr = (out\_write\_ptr + 1) \ \% \ buffer\_size$$

If at *new_ptr* there is a character which has not been read by the controller yet, the driver actively waits until the *new_ptr* position gets free.

```
while (out_read_pointer == new_ptr)
    ;
```

Active waiting is acceptable in this case, since

- the case when the *new_ptr* position is occupied by an unread character is rare, because the size of the buffer is big enough (several thousands of characters),
- it will take only a very short time before the controller picks the unread character up, and
- the thread which prints to the output would have to wait anyway.

Integrating the Driver into HelenOS

By the time the SGCN driver was written, HelenOS had already used the serial console input and output – in the MSIM (simulated MIPS) port. To prevent code duplication, portions of the MSIM console input and output which were not MSIM-specific were extracted. The portions are now used by both the SGCN driver and the MSIM console driver.

Userspace SGCN Driver

The kernel SGCN driver pre-computes the address and size of the 'OBPCONS' memory area and passes it to its userspace counterpart. The userspace driver then maps the area to its address space and works the same way as the kernel driver.

7.6 Miscellaneous

7.6.1 Setting Color Palette

If the color depth is set to 8 bits, each byte in the framebuffer acts as an index to the color palette. This is different from the 16-bit (24-bit) color depth, where the couple (triplet) of bytes encodes directly the red, green and blue components of the color.

In the previous implementation of the HelenOS SPARC port the color palette was not explicitly set, so the palette contained colors set by the OBP. Therefore, if the 8-bit color depth was configured, the framebuffer showed wrong colors.

During the work on this thesis the author discovered a way how to set the color palette using the OBP client interface. The respective OBP method is called `color!` and it is to be called from the `screen` node of the OBP tree for every color to be set in the palette. The method expects an index to the palette, intensity of red, intensity of green and intensity of blue on the stack.

The palette is set such that the most significant three bits of the color index encode the intensity of the red component, the following two bits encode the intensity of the green component and the least significant three bits of the index encode the intensity of the blue component. The HelenOS framebuffer driver can therefore, if the 8-bit depth is set, treat the framebuffer in a similar way as in the 16-bit or 24-bit mode.

The palette must be set in the bootloader, before the OBP is disabled.

This solves the problem pointed out in [jj_thesis] in section 4.3.7 – Handling I/O Devices, Summary of Hardware Support.

Chapter 8

Porting HelenOS to sun4v Processors

Enhancements which enable running HelenOS on the sun4v processors were made by Pavel Římský in 2008 and 2009 as a part of his master thesis.

8.1 Overview

8.1.1 Supported Environments

HelenOS is able to run on a Simics-simulated Sun Fire T2000 server with an UltraSPARC T1 processor and on a real Sun Fire T1000 Enterprise server with an UltraSPARC T1 processor. The machine simulated by Simics is a little bit simplified in comparison with its real-world counterpart: the processor frequency is hardwired to 5 MHz and the physical memory size is hardwired to 256 MB, but it is sufficient for development purposes.

8.1.2 Enhancements Overview

As well as during porting HelenOS to the JPS-compliant processors, some register formats had to be changed. This kind of changes, however, is not that interesting. While in the JPS port it was one of the few things to be changed, in the sun4v port redefining registers is only a small portion of the labor.

Porting HelenOS to the sun4v architecture required to write a new algorithm of TLBs takeover, to implement the hypercalls mechanism, to modify some memory management routines, to modify the preemptible trap handler, to write (at least simplistic) drivers of input and output and to make a handful of other minor changes.

In comparison with the JPS port, only a small amount of changes concerns OpenBoot PROM. Luckily, the sun4v port does not have to rely on the OpenBoot that much since a lot of actions can be performed via the hypervisor instead. Moreover, the hypervisor is perfectly documented (unlike OBP).

8.1.3 Integration with the sun4u Port

The kernel code of the sun4u port and the code of the sun4v port are separated as follows:

- If in the sun4u port there was a header file `path_to_file/xxx.h` which contained some definitions which had to be adapted for sun4v, the header file was split into
  - `path_to_file/sun4u/xxx.h`, which contains only definitions specific for sun4u,
  - `path_to_file/sun4v/xxx.h`, which contains only definitions specific for sun4v, and
  - `path_to_file/xxx.h`, which contains definitions common for both sun4u and sun4v, and which includes either `path_to_file/sun4u/xxx.h` or `path_to_file/sun4v/xxx.h` depending on symbolic constants SUN4U and SUN4V being defined. This is an example for file `cpu.h`:

```
#if defined (SUN4U)
    #include <arch/sun4u/cpu.h>
#elif defined (SUN4V)
    #include <arch/sun4v/cpu.h>
#endif

/* definitions common for both sun4u and sun4v */
```

Example 8.1: Example of a Common Header File

  Other files always include the common file, not the specific file, so that they do not have to worry about the particular UltraSPARC architecture – correct definitions will always be included. The symbolic constants SUN4U and SUN4V are defined in the Makefile.

- If in the sun4u port there was a C file `path_to_file/xxx.c` which contained some definitions which had to be adapted for sun4v, the file was split into
  - `path_to_file/sun4u/xxx.c`, which contains only definitions specific for sun4u,
  - `path_to_file/sun4v/xxx.c`, which contains only definitions specific for sun4v, and
  - `path_to_file/xxx.c`, which contains definitions common for both sun4u and sun4v.

  The Makefile ensures that always the correct files will be compiled and linked.

In the bootloader an autodetection is used – the OBP root node 'compatible' property indicates whether the CPU's architecture is sun4u or sun4v.

Comparison with Linux

The way how the sun4v-specific code is integrated into the SPARC port differs between Linux and HelenOS. The most notable difference is that Linux does not separate the sun4u-specific and sun4v-specific pieces of code by means of the preprocessor `#i-f` nor `#ifdef` clauses, but it uses a runtime autodetection of the architecture. The architecture is detected by analyzing the OBP tree. In the `arch/sparc64/kernel-/head.S` file the host CPU type is detected and if its architecture is sun4v, a global variable `is_sun4v` is set to 1. This variable is inspected from a plenty of places in the code.

The implantation of the sun4v code into Linux is sometimes not very straightforward. Let us have a look at one trick which is used to make certain MMU functions do a sun4v-specific job. Linux SPARC port defines several generic functions for handling TLB misses, which are overridden by the sun4v-specific functions if the host CPU's architecture is sun4v; for example, `tl0_iamiss` is overridden by `sun4v_itlb_mi-ss` or `tl1_iamiss` is overridden by `sun4v_itlb_miss` (to understand this text, it is not important to know what these functions do). An interesting thing is the mechanism how the functions are overridden. Actually, the generic functions are *patched*. If the detected architecture is sun4v, the very first instruction of each such generic function is replaced by a branch to the corresponding sun4v-specific function (by modifying the instruction memory) at runtime. The code doing this can be found in the `arch/spa-rc64/kernel/sun4v_tlb_miss.S` file.

In Linux the sun4u and sun4v sources are not so strictly separated as in HelenOS. In a big number of `*.c` and `*.h` files the sun4u-specific and sun4v-specific parts are intermixed. Some source files have 'sun4v' in their names, such as `sun4v_icev.S` – those ones contain only sun4v-specific pieces of code. The names of the sun4v-specific functions, structures, etc. usually start with 'sun4v'.

Due to the use of the `#if` and `#ifdef` clauses the way how architecture-specific parts of HelenOS code are separated is much more straightforward and clean – no such ugly things as patching the code of the generic functions is used. Because the sun4v HelenOS bootable image contains only sun4v-specific code (except for the bootloader), the image is small. The drawback is that the sun4u and sun4v HelenOS bootable images must be compiled separately.

Comparison with Solaris

The way how the sun4u and sun4v sources are separated in Solaris very much resembles the way how it is achieved in HelenOS. The majority of the sun4u-specific pieces of code is in the `uts/sun4u` directory, the majority of the sun4v-specific pieces of code is in the `uts/sun4v` directory. The `uts/sun4` directory contains code common for both sun4u and sun4v with a few pieces of code specific to one of these platforms (the pieces are separated by `#ifdef sun4u` .. `#endif` and `#ifdef sun4v` .. `#endif` clauses).

The difference between Solaris and HelenOS is the directory structure level at which the sun4u and sun4v sources are split. While in Solaris the `sun4u` and `sun4v` subdirectories are almost at the top of the directory structure, in HelenOS they are the leaf directories.

8.2 Bootloader and Kernel Startup

8.2.1 Bootloader

One of the very first modifications which had to be made to HelenOS to support the sun4v architecture was adding a piece of code which would detect whether the host

CPU architecture is sun4u or sun4v. The detection is performed in the `detect_architecture` function, which is called as soon as the HelenOS bootloader is passed control by SILO.

Whether the CPU architecture is sun4u or sun4v is determined by inspecting the OBP root node 'compatible' property. If the value of the property is 'sun4v', the bootloader concludes that the architecture is sun4v. Otherwise the bootloader concludes that the architecture is sun4u. The reason why the bootloader does not check that the property values is 'sun4u' before making the conclusion is that on some sun4u machines the property value may be different from 'sun4u'; for example, on the Serengeti machine the 'compatible' property has value 'SUNW,Serengeti'. Hence sun4u is considered to be the default. The information about the architecture is saved to the `architecture` variable. It contains either the value of the `COMPATIBLE_SUN4U` or `COMPATIBLE_SUN4V` symbolic constant.

The information about the architecture affects several issues of the bootloader. Copying kernel image, copying the initial tasks and allocating memory for the RAM disk is the same for both sun4u and sun4v, as well as passing the memory map to the kernel. On sun4u a copy of the OBP tree is made, while on sun4v the machine description is used instead of the OBP tree. On sun4u a color palette is set, while sun4v machines have no framebuffer, so there is no point in setting the palette. On sun4u the processor model is detected, while on sun4v there is no need to do it, since the privileged software is shadowed from the processor model details thanks to the hypervisor.

OBP-established Page Mapping

When SILO booted, the OBP had established a virtual-to-real memory mapping. This mapping is not an identity (because the real memory starts at a non-zero address), which is not surprising. However, the mapping even does not map virtual address 0 onto the starting address of the real memory, but onto an address which is 0x400000 bytes (4 MB) higher. The reason is that the OBP occupies the first 4 MB of the real memory (so neither SILO nor the bootable image may be placed there) and this real memory is mapped in a different way.

In order to keep the HelenOS bootloader (which expects identity virtual-to-real mapping) simple and to avoid code relocation, the problem is overcome in this manner: we pretend that the real memory starts 0x400000 bytes further than it actually does (and hence pretend that the real memory is 0x400000 bytes smaller). So the memory map is patched like this (note that the bootloader, for sake of simplicity, does not distinguish between the terms 'real' and 'physical' memory as the code is common for both the sun4u and sun4v architectures):

```
// pretend the physical memory starts 4 MB further
bootinfo.physmem_start += 0x400000;

// pretend the first zone starts 4 MB further
bootinfo.memmap.zones[0].start += 0x400000;

// pretend the first zone is 4 MB smaller
bootinfo.memmap.zones[0].size -= 0x400000;
```

8.2.2 Kernel Startup

The kernel is passed control from the bootloader in the `kernel/arch/sparc64/src/sun4v/start.S` file code. First, the very basic information passed by the bootloader – the starting address of the physical memory, the information whether the host CPU is a bootstrap CPU and the boot info structure – is processed and set aside to the local registers. Then the basic state registers (`CANSAVE`, `CANRESTORE`, `OTHERWIN`, `CLEANWIN`, `WSTATE`, `TL`, `PSTATE`) are initialized. After that the trap table base address register is set to point to the HelenOS trap table (before that it used to point to the OBP trap table). These steps are almost identical to those of the original UltraSPARC port. Then some more interesting steps come, as described in the following subsections.

MMU Takeover

The UltraSPARC Architecture 2005 specification states that the MMUs are disabled upon the system startup and that before using them they must be explicitly enabled by a hypercall. The kernel, however, does not have to take it into account, as the OpenBoot PROM has already enabled the MMUs. All it must do is take a full control over the TLBs from the OBP[1].

Taking over the TLBs works in almost the same way as on the sun4u architecture (see Section 6.1.3). The main difference is that the ITLB is taken over concurrently with the DTLB, which is easy since the hypercalls which install the mappings can operate on both the TLBs simultaneously.

The individual steps of the takeover are as follows:

1. The second 4MB page, where the kernel is loaded, is mapped in context 1 using the `MMU_MAP_ADDR` hypercall.

2. Context 1 is switched to.

3. From within context 1, all mappings in context 0 are demapped using the `MMU_DEMAP_CTX` hypercall.

4. A permanent mapping of the kernel (i.e. The second 4MB page) is installed to context 0.

5. Context 0 is switched to again.

6. From within context 0, all mappings in context 1 are demapped (cleanup).

Thanks to the fact that UltraSPARC Architecture 2005 processors do not require any memory synchronization instruction (`FLUSH` etc.) after a store to an MMU register, the code is much simpler than the analogous code for the sun4u architecture.

Setting MMU Fault Status Area

The MMU fault status area is defined in an assembly language file called `start.S`, which is the file in which the routine for taking over the MMU is defined. The size of the MMU fault status area is `MMU_FSA_SIZE` * `MAX_NUM_STRANDS`, where `MMU_FSA_SIZE`

[1]This is true only on the bootstrap processor. On the application processors the MMU must be enabled explicitly, see Section 8.8.7.

is the size of the MMU fault status area structure for one virtual processor and `MAX_-NUM_STRANDS` is the maximum number of virtual CPUs HelenOS supports. The MMU fault status area is set up just after taking over the MMUs using the `MMU_FAULT_AREA-_CONF` hypercall, which takes the real address of the area as its argument.

Scratchpad Registers

To the first scratchpad register an ID of the current CPU is written, so that the ID can be quickly read (without the need of performing a hypercall) in time-critical sections of the code, such as trap handlers. To the second scratchpad register the real address of the MMU fault status area is snapshotted for the same reason.

Once the first and second scratchpad registers are set, they will be only read from the HelenOS kernel, never written to.

When switching to a new thread, the address of the thread's kernel stack is snapshotted to the third scratchpad register and the address of the thread's userspace window buffer is snapshotted to the fourth scratchpad register.

Scratchpad Registers: Comparison with Linux and Solaris

Both Linux and Solaris use the scratchpad registers for the same purpose. The first and second scratchpad registers are used for the same purpose as in HelenOS, usage of the third and the fourth scratchpad registers differs from HelenOS.

In Linux and Solaris, the third scratchpad register (`SCRATCHPAD_UTSB1`) encapsulates the base address and the size of the first TSB of the active userspace process, the fourth scratchpad register (`SCRATCHPAD_UTSB2`) encapsulates the base address and the size of the second TSB of the active userspace process or it contains NULL if the process has only one TSB. These registers are written to during a context switch and they are read in the TLB miss handler when a translation is not found in the TSB. They speed up refilling the TSB with the translation.

In HelenOS only one TSB is used by each process (task) and its address and size is read directly from the in-memory structure which describes the TSBs of the process. The two spare scratchpad registers can then hold pointers to the kernel stack and the userspace window buffer.

Finishing the Startup

When the MMU fault status area is configured, 8kB TTE template is precomputed just as in Section 6.1.3, the stack is configured and a generic C language initialization routine is called.

8.3 Hypercalls

Hypercalls are used by almost all subsystems of the sun4v HelenOS kernel – by input and output drivers, by the memory management routines, for obtaining the machine description, for waking the application processor up and for operations related to interprocessor interrupts.

Symbolic constants have been defined for SW trap numbers, function numbers and return codes. Apart from that a few assembly language macros and C routines have been defined, which perform the hypercall.

8.3.1 Performing Hypercalls from Assembly

Before making a hypervisor API call an assembly language code must set the output registers properly. After that it just invokes one of these two macros:

- `__HYPERCALL_FAST(function_number)`, when a fast hypercall is to be performed, or
- `__HYPERCALL_HYPERFAST(sw_trap_number)`, when a hyperfast hypercall is to be performed.

In Example 8.2 calling the hypervisor API (more specifically demapping the whole context) is illustrated.

```
! demap all in context 0
set 0, %o0
set 0, %o1
set 0, %o2
set MMU_FLAG_DTLB | MMU_FLAG_ITLB, %o3
__HYPERCALL_FAST(MMU_DEMAP_CTX)
```

Example 8.2: Calling Hypervisor from Assembly

8.3.2 Performing Hypercalls from C

There are two groups of C routines that make a hypervisor API call.

The first one is used for hypercalls which, apart from the error code, produce no output. This group contains:

- Six routines for performing a fast hypercall, each routine takes a different number of input arguments. The name of each routine is `__hypercall_fast`*n*, where *n* is the number of the hypercall input arguments. For example, calling a fast hypervisor service which takes 3 arguments and returns no values (apart from the error code) would look like this:

  ```
  err = __hypercall_fast3(
      MMU_UNMAP_PERM_ADDR, tsb, 0, MMU_FLAG_DTLB);
  ```

- One routine for performing a hyperfast hypercall. The routine takes five arguments. Should the hypervisor service take less, the redundant arguments must be zero. For example, creating new MMU mapping would look like this:

  ```
  __hypercall_hyperfast(
      page, ASID_KERNEL, data.value, MMU_FLAG_DTLB, 0,
      MMU_MAP_ADDR);
  ```

The second group contains one routine for performing a fast hypercall which returns, apart from the error code, one output value. The routine takes five arguments. Should the hypervisor service take less, the redundant arguments must be zero. For example, obtaining the CPU ID would look like this:

```
err = __hypercall_fast_ret1(0, 0, 0, 0, 0, CPU_MYID, &myid);
```

As for the implementation of these functions, the only interesting thing is the way how the return value is retrieved in the `__hypercall_fast_ret1` function. First, the `__hypercall_fast` function is called, then the return value is picked up from the `%o1` register:

```
uint64_t errno = __hypercall_fast(
        p1, p2, p3, p4, p5, function_number);
if (ret1 != NULL) {
  asm volatile ("mov %%o1, %0\n" : "=r" (*ret1));
}
```

8.3.3 Comparison with Linux and Solaris

Neither Linux nor Solaris define generic functions for performing a hypercall; they both define separate wrapper functions for each particular hypercall type. To make a hypercall, the kernel calls the corresponding wrapper.

The wrapper functions are defined in assembly language as leaf-optimized. They use the fact that both the function calls and hypercalls use the output registers to pass the call parameters, so the wrapper functions do not have to touch the output registers (except for the `%o5` register which holds the function number of the fast hypercall), but can simply let the values passed in the wrapper function call be passed as parameters of the hypercall. The typical function making the hypercall just sets the function number, invokes the trap and returns. This is an example of the `CONS_PUTCHAR` wrapper in Linux:

```
    .globl sun4v_con_putchar
    .type sun4v_con_putchar, #function
sun4v_con_putchar:
    mov HV_FAST_CONS_PUTCHAR, %o5
    ta HV_FAST_TRAP
    retl
    .size sun4v_con_putchar, .-sun4v_con_putchar
```

In HelenOS the hypercall mechanism is defined consistently with the syscall mechanism. Since almost every hypercall type is used only once from the HelenOS code, defining a separate function for each hypercall type would produce a lot of redundant code which would have to be maintained.

8.4 CPUs

This section deals with the machine description traversal and with one issue related to the `TICK` register.

8.4.1 Machine Description

Machine description (MD) is a data structure provided by the hypervisor which describes the host system (see Section 5.2.3 for further details). The code for the machine description traversal can be found in the `kernel/arch/sparc64/src/sun4v/md-.c` file.

In HelenOS the machine description is used to retrieve some essential information about the CPUs – their count, their cpuids and their clock frequencies. It is also used to find the physical cores the particular virtual processors are backed by. The pieces of code the machine description traversal is used from can be found in the `kernel/-arch/sparc64/src/smp/sun4v/smp.c` and `kernel/arch/sparc64/src/cp-u/sun4v/cpu.c` files.

Let us briefly summarize the most important functions related to the machine description traversal. The `md_init` function initializes some helper data structures used for the MD traversal. The `md_get_root` function returns the root node of the MD. The `md_next_node` function takes the last visited node and a name of a node, it returns the first not-yet-visited node with the given name (the function is used to iterate through all the nodes with a given name, e.g. 'cpu', without bothering with the tree-like structure of the MD). The `md_get_child_iterator` function returns an iterator used to iterate through all the children of a given node. The `md_next_child` function takes a child iterator and it returns the first not-yet-visited child.

8.4.2 TICK Register

In the original UltraSPARC HelenOS port the `TICK` register was set to zero upon the initialization of the timer, and the `TICK_COMPARE` register was set to value t – the number of instructions after which the first timer interrupt should be triggered.

Since on sun4v the `TICK` register is hyperprivileged, it cannot be set to zero, so the original value v must be kept there. As a consequence, the `TICK_COMPARE` register must be set to $v + t$ then.

The `STICK` register is not used at all in the sun4v HelenOS port, as well as it is not used in the JPS port. On the UltraSPARC T1 processor the `STICK` register is actually an alias of the `TICK` register.

8.5 Memory Management

Memory management in the sun4v HelenOS port is similar to the one found in the original UltraSPARC port. The basic difference is that on sun4v the hypercalls are used instead of directly accessing the MMU registers.

8.5.1 Caches

Fortunately, the illegal alias problem (see Section 6.3.2) does not occur on sun4v. The size of the Niagara data cache is 8kB only. Thanks to that, the sun4v HelenOS port does not emulate the 16kB pages. Upon an MMU miss an 8kB page is installed.

8.5.2 MMU Miss Handler

The MMU miss handler is an assembly language routine entered from the privileged trap table. Unlike on sun4u, it contains no code for looking up the translation in the TSB. The reason is that once the privileged MMU miss handler is entered, the hypervisor has already unsuccessfully tried to look the translation up in the TSB, as described in Section 5.3.2.

The sun4v handler does not fit into 32 instructions, so a branch is needed. The reason why 32 instructions are not enough is mainly that the output registers must be snapshotted to the global registers (at `GL` = 1) before the they can be used to pass hypercall arguments, and they must be restored from the global registers once the trap is handled. Due to the fact that the hypercalls will use a total amount of four output registers, the handler will inevitably need eight instructions solely for saving and restoring the output registers. Note that allocating a new register window would be a risk here, as it could cause a nested trap.

The steps of the miss handler are very similar to those of the sun4u handler, with minor differences. This is what the sun4v handler actually does:

1. The real address of the MMU Fault Status Area is read from the second scratchpad register.

2. The faulting context is read from the MMU Fault Status Area. The context is read via the *MMU bypass ASI*, since a *real* address of the MMU Fault Status Area has been read from the second scratchpad register.

3. The faulting virtual page number is read from the MMU Fault Status Area. The page number is read via the *MMU bypass ASI*, since a *real* address of the MMU Fault Status Area has been read from the second scratchpad register.

4. If the faulting context is not a kernel context (zero), a higher level service routine is called.

5. If the faulting page has number zero, it means the kernel may be dereferencing a null pointer, so a higher level service routine is called.

6. For the kernel context and non-zero pages a kernel mapping is installed, as described in the following section.

Handling Kernel Misses

Mapping for the kernel is being installed in a separate assembly routine called `install_identity_mapping`, thus the number of instructions is not limited to 32. A jump to the `install_identity_mapping` routine is done directly in the MMU miss handler. The MMU miss handler leaves the virtual address in the `%g1` register from where the `install_identity_mapping` routine picks it up.

Before installing the kernel mapping, output registers are snapshotted to global registers (as already pointed out). Not all output registers need to be snapshotted, only those which will be used for passing parameters to hypercalls during installing the mapping, i.e. `%o0` to `%o3`.

```
mov %o0, %g3
...
mov %o3, %g6
```

Similarly as on sun4u, kernel MMU misses on sun4v are handled by installing a mapping obeying formula

$PA = VA + C + 0x400000,$

where C is the starting address of the physical memory. The constant 0x400000 has to be added due to the issue described in Section 8.2.1).

The TTE is composed by ADDing the faulting virtual address to the 8kB TTE template, which has been prepared during the kernel startup (see Section 8.2.2). The translation is installed via the `MMU_MAP_ADDR` hypercall.

When the mapping has been installed, registers `%o0` to `%o3` are restored from the globals and the instruction which caused the trap is restarted.

Kernel Misses: Comparison with Linux

Let us have a look how Linux copes with the fact that only 32 instructions may fit into the MMU miss entry in the trap table.

In Linux, the DMMU miss handler (`sun4v_dtlb_miss` in the `sun4v_tlb_miss.S` file) takes 37 instructions, so it (as well as in HelenOS) does not fit into one trap table entry. Linux uses branching too, but unlike in HelenOS, the branch is not very apparent.

In the Linux sparc64 port in the trap table (`arch/sparc64/kernel/ttable.S`) there is an entry called `tl0_damiss`, which handles the DMMU misses. In runtime the `tl0_damiss` is patched (as described in Section 8.1.3) so that when `tl0_damiss` is invoked, a branch is taken and the code of `sun4v_dtlb_miss` is executed. So the branch is not explicitly in the code, but it *is* effectively performed. Since `sun4v_dtlb_miss` resides outside the trap table, it may contain more than 32 instructions.

Handling Userspace Misses

Userspace MMU misses are handled in a similar way as in the sun4u port.

One thing which deserves a description is the way how the faulting context and address is passed from the (assembly language) MMU miss handler to the higher level service routine.

In the sun4u HelenOS port the higher level service routine is passed a value of the TLB Tag Access register (register from which the faulting context and address can be determined) as its parameter. Since on sun4v the faulting context and address is not read from the TLB Tag Access register but from the MMU Fault Status area, the information about the fault is passed in a different way to the service routine. Both the context and the address must be encoded into one 64-bit value, because only one 64-bit parameter may be passed to the service routine. Specifically, the faulting page address is encoded into bits 63 to 13 and the faulting context is encoded into bits 12 to 0 of the value being passed.

Another notable issue is a simplification of some functions due to the illegal alias problem not occurring on sun4v. The `itlb_pte_copy` function (which copies a TTE from the translation table to the TLB) takes the *index* parameter on sun4u, which is 0 if the lower 8kB sub-page of the simulated 16kB page is to be copied and 1 if the higher 8kB sub-page of the simulated 16kB page is to be copied to the TLB. On sun4v this is not necessary, as in the sun4v port the 16kB pages are not simulated.

8.5.3 Translation Storage Buffers

The sun4v HelenOS port makes use of the TSB for non-kernel mappings (the same way as the sun4u port does) – for each memory context there exists one TSB. The only difference from the sun4u port is that when the MMU miss trap comes, the hypervisor tries to automatically reload the missing mapping from the TSB (see Section 5.3.2), so in the sun4v port there is no code for reloading the mapping from the TSB. There is, however, code for the TSB initialization and for installing entries to the TSB. The code is very similar to the one of sun4u.

As pointed out in Section 5.3.1, translation storage buffers on sun4v have some extra features, such as multiple way set associativity. For the time being, the HelenOS port makes no use of these features. Only one direct mapped TSB is used for each context. The size of the TSB is 64 kilobytes, as in the sun4u port, so the code for adding entries to the TSB and for initializing the TSB is very similar to the analogous code for sun4u. Since the size of one TSB entry is 16 bytes, the total number of TSB entries is 4096.

TSB Description

The TSB Description is a structure, which is passed to the hypervisor when setting up the TSBs, describing the properties of a particular TSB (see Section 5.3.1 for more details). In HelenOS the structure is defined in the `kernel/arch/sparc64/include/mm/sun4v/tsb.h` file and it is called *tsb_descr_t*.

TSB Initialization

The TSB is being initialized in the constructor of the address space.

First, a 64kB page is allocated for the TSB. The page is *not* partitioned into instruction and data TSB (unlike on sun4u), but the whole page is used for both instruction and data translations.

Second, the TSB Description structure is initialized. The TSB Description structure is referenced from a structure called `as_arch_t`, which is a structure describing the platform-specific properties of a particular address space. In the TSB Descriptions the address of the just allocated 64kB page is set, TSB associativity is set to 1, the size of the pages being mapped is set to 8 kB and the number of entries is set to 4096.

At the end the TSB page is filled with zeros.

Installing the TSB

The TSB is installed via an `MMU_TSB_CTXNON0` hypervisor API call to which the TSB description is passed. This is done every time the TSB context is switched – in the `as_install_arch` function.

Manipulating the TSB

Functions which manipulate the TSBs (add new mappings to the TSB and invalidate the TSB) are located in the `kernel/arch/sparc64/src/mm/sun4v/as.c` and the `kernel/arch/sparc64/src/mm/sun4v/tsb.c` files. The basic things in which they differ from the analogous functions for the sun4u architecture are:

- They do not simulate the 16kB pages, since on sun4v the illegal alias problem does not occur. This simplifies implementation of some functions. For example, the `itsb_pte_copy`, which copies the instruction memory page table entry to the TSB, does not have to take the `index` parameter which would determine whether the lower 8kB entry or the higher 8kB page entry of the faulting 16kB page is to be copied. In the sun4u port the function looked as follows:

  ```
  void itsb_pte_copy(pte_t *t, index_t index)
  {
    /* ... */
    ASSERT(index <= 1);
    as = t->as;
    entry = ((t->page >> MMU_PAGE_WIDTH) + index)
        & TSB_INDEX_MASK;
    /* ... */
  }
  ```

  This has been simplified in the sun4v port:

  ```
  void itsb_pte_copy(pte_t *t)
  {
    /* ... */
    as = t->as;
    entry = (t->page >> MMU_PAGE_WIDTH) & TSB_INDEX_MASK;
    /* ... */
  }
  ```

- There is one common TSB for both the instruction and data memories (the hypervisor API provides no means how to instruct the hypervisor to use separate TSBs for instruction and data memories). Whereas in the sun4u port the 64kB page which held the TSB was split into the ITSB and DTSB parts, in the sun4v port one 64kB page holds TSB entries for both the instruction and data translations. As a consequence, the index to the TSB is calculated in a different way.

  The following example shows how the entries are retrieved inside the ITSB and DTSB respectively in the sun4u port:

  ```
  as->arch.itsb[entry_index & (ITSB_ENTRY_COUNT - 1)]
  as->arch.dtsb[entry_index & (DTSB_ENTRY_COUNT - 1)]
  ```

  And this is how the TSB entry is retrieved in the sun4v port:

  ```
  as->arch.tsb_description.tsb_base[
      entry_index & (TSB_ENTRY_COUNT - 1)]
  ```

- The way how the TSB entry is invalidated during its update is different. Generally, when updating a TSB entry, the entry must be invalidated first, then a memory barrier must be issued, then the entry can be updated, then the memory barrier must be issued again and finally the entry can be made valid again. In the sun4u port the entry is invalidated by setting the most significant bit of the TSB tag to 1 (this is our internal convention introduced in the original UltraSPARC HelenOS port). This bit is ignored by the HW, but it is used by the MMU miss handler to verify whether the entry is valid when reloading the entry from the TSB. Since on sun4v *hypervisor* is the one who reloads entries from the TSB, not the kernel MMU handler, setting the most significant bit would have no effect (recall it is just our internal convention). So in the sun4v port the TTE data's 'valid' bit is set to zero instead in order to invalidate a given entry.

TLB Miss Handling

When a TLB miss occurs, the hypervisor tries to reload the missing mapping from the TSB. If the TSB description has not been supplied to the hypervisor for the given memory context, the hypervisor calls the *Fast Data Access MMU Miss* or *Fast Instruction Access MMU Miss* handler from the privileged trap table. If the description has been supplied to the hypervisor for the given memory context, but the translation is not present in the TSB, the hypervisor calls the *Data Access MMU Miss* or *Instruction Access MMU Miss* handler from the privileged trap table. In the HelenOS privileged trap table the *Data Access MMU Miss* and *Instruction Access MMU Miss* handlers just unconditionally branch to the *Fast Data Access MMU Miss* and *Fast Instruction Access MMU Miss* handlers respectively. The miss is then handled as outlined in Section 8.5.2.

Comparison with Linux

Linux uses TSB for both the kernel and the userspace. If configured so, Linux uses two TSBs for one userspace process – one which holds mapping for 'small' pages and one which holds mappings for 'big' pages. How big the 'small' and 'big' pages are is defined in compile-time.

Linux, as well as HelenOS, uses only direct-mapped TSBs.

An interesting thing about Linux TSBs is that the TSBs can have dynamic size – Linux kernel is able to increase the size of a TSB on the fly. For further details, see [dyn_tsb].

8.6 Preemptible Trap Handler

The sun4u preemptible trap handler has been moved to the `arch/sparc64/src/trap/sun4u/trap_table.S` file, while a new file `arch/sparc64/src/trap/sun4v/trap_table.S` has been created to incorporate the preemptible trap handler for the sun4v architecture. Even though the sun4u and sun4v handlers do not differ much, they have been placed to separate files in order to preserve readability of such an important and error-prone piece of code.

Preemptible trap handlers for sun4u and sun4v have a lot of aspects in common and their code looks very similar. Handler for sun4v, however, must cope with three apparent problems which are not present on sun4u:

- The number of trap levels usable by the privileged software is smaller than on sun4u. Not more than two levels are possible above level zero.
- Alternate, memory and interrupt globals have been replaced by alternative globals depending on the value of the `GL` register.
- If a privileged software running on trap level `TL` = 0 stores some values to global registers at `GL` = 1, there is no guarantee that the values will not be overwritten by the hypervisor (as described in Section 5.4.2). This is different from sun4u where the privileged software could save anything to some of the alternative global sets and no-one could destroy such values.

8.6.1 Limited Trap Levels

Privileged software may use only two trap levels above level zero. HelenOS's usage of particular trap levels is as follows:

level 0

Normal execution, no trap being handled (or the trap is being handled by a higher level service routine).

level 1

Register window traps, MMU traps and interrupts.

level 2

Kernel MMU misses caused by the preemptible trap handler accessing the stack or the userspace window buffer.

Without additional modifications the preemptible trap handler could easily bring the CPU to trap level 3:

1. The preemptible handler is entered when the processor works on trap level 1.
2. The preemptible handler needs to issue the `SAVE` instruction in its prologue. This may lead to a spill trap, bringing the CPU to trap level 2.
3. If the stack is not mapped, handling the register window trap will cause a nested MMU miss trap, bringing the CPU to trap level 3.

Analogously the same situation may occur when issuing the `RESTORE` instruction in the preemptible trap handler's epilogue.

A solution of this problem is to avoid the spill and fill traps by proactively spilling the register window if the `CANSAVE` register equals 0 and proactively filling the window if the `CANRESTORE` register equals 0, just before issuing the `SAVE` or `RESTORE` instruction.

The following code snippet illustrates what must be done before issuing the `SAVE` instruction in the preemptible trap handler.

```
rdpr %cansave, %g3      ! find the value of the CANSAVE register
brnz %g3, 2f            ! skip spilling if CANSAVE is not zero
nop
INLINE_SPILL %g3, %g4   ! actively spill the register window

2:
```

```
/* ask for new register window */
save %sp, -PREEMPTIBLE_HANDLER_STACK_FRAME_SIZE, %sp
```

INLINE_SPILL is an assembly language macro which spills the register window at CWP + 2 (mod NWINDOWS) onto the stack. Therefore the subsequent SAVE instruction may not cause any trap, since it will always be issued with the CANSAVE register being non-zero. The macro is defined directly in the trap_table.S file.

Analogously there exists a macro for spilling the active register window to the userspace window buffer. The macro is called INLINE_SPILL_TO_WBUF. The macro is called from the preemptible trap handler if the trap occurred in a userspace task and a SAVE instruction would cause an undesired spill exception which would be handled by spilling the active window to the userspace window buffer. There is no need to implement a macro for filling the registers from the userspace window buffer, because the registers are filled in the preemptible trap handler epilogue.

8.6.2 Alternative Globals Set

On sun4u the alternate globals set was used during the execution of the preemptible trap handler. In the sun4v handler, the globals set from GL = 1 is used instead.

One problem is, however, connected with the alternative globals set. In Section 6.3.1 it has been mentioned that the address of the thread's kernel stack and the thread's userspace window buffer is saved to the alternative global registers %g6 and %g7 respectively. This is, however, not possible on sun4v due to the fact that hypervisor might overwrite the alternative globals (see Section 5.4.2).

Original Solution

There are many ways how the problem can be overcome. The author of this thesis first tried a solution in which the address of the kernel stack and the userspace window buffer was stored to the memory. The solution worked fine, but accessing the memory is slow, so it was later replaced with a different solution. First, let us explain the original solution.

In this solution the addresses of the stack and the window buffer are stored to the memory instead of registers. For each virtual processor an instance of a data structure defined like this exists:

```
typedef struct {
    uintptr_t kstack;
    uintptr_t wbuf;
} __attribute__ ((packed)) kstack_wbuf_ptr;
```

The kstack attribute contains the address of the top of the kernel stack of the thread, which is currently active on the given virtual processor. The wbuf attribute contains the address of the top of the userspace window buffer of the thread, which is currently active on the given virtual processor. The structures are arranged in an array defined in the start.S file[2].

Data in this array are updated during the context switch. They are read and updated by the preemptible trap handler and also by the register window spill handler, which is used for userspace windows marked as *other* (the rationale behind marking

[2]The file containing the first code to be executed when the kernel boots.

some windows as *other* is explained in Section 6.2.2). An assembly language macro is defined which finds an address of the `kstack_wbuf_ptr` for the current virtual processor. To understand the macro, see the inline comments:

```
/*
 * Computes the pointer to the kstack_wbuf_ptr structure of
 * the current CPU.
 *
 * Parameters:
 * tmpreg1    global register to be used for scratching purposes
 * result     register where the resulting pointer will be saved
 */
.macro get_kstack_wbuf_ptr tmpreg1, result
    ! load CPUID to tmpreg1
    or %g0, SCRATCHPAD_CPUID, \tmpreg1
    ldxa [\tmpreg1] ASI_SCRATCHPAD, \tmpreg1

    ! compute offset within the array of kstack_wbuf_ptr
    ! structures (each such structure is 16 bytes long)
    mulx \tmpreg1, KSTACK_WBUF_PTR_SIZE, \tmpreg1

    ! compute the pointer to the structure for the current CPU
    sethi %hi(kstack_wbuf_ptrs), \result
    or \result, %lo(kstack_wbuf_ptrs), \result
    add \result, \tmpreg1, \result
.endm
```

Note that the number of instructions of this routine is 6. This is important because this routine is also used from within the window spill handler, in which the maximum number of instructions is limited to 32.

The Current Implementation

In the current implementation the scratchpad registers are used. The third scratchpad register is used as a storage of the kernel stack address and the fourth scratchpad register is used as a storage of the userspace window buffer address.

As an example let us show how the kernel stack address is retrieved in the preemptible trap handler prologue:

```
set SCRATCHPAD_KSTACK, %g4
ldxa [%g4] ASI_SCRATCHPAD, %g6
save %g6, -PREEMPTIBLE_HANDLER_STACK_FRAME_SIZE, %sp
```

In the early stage of the porting effort the author first tried to avoid wasting the scratchpad registers for this purpose, in case they would be needed for some different purpose in the future. Anyway, when HelenOS started to work correctly on Niagara without the third and fourth scratchpad registers being touched, the author decided to optimize the kernel by using the scratchpad register for holding the stack and window buffer addresses.

Comparison with Solaris

In Solaris there is a routine analogous to the preemptible trap handler which is called `sys_trap`. The register windows which belong to the user process in which the trap occurred are saved to the userspace window buffer, which is called `wbuf` in Solaris. In Solaris the pointer to the kernel stack and the userspace window buffer is saved to an in-memory structure, similarly as in HelenOS in the original solution.

In Solaris a global array called `cpu` exists. The array is indexed by CPU IDs. Each element of the array is a pointer to the `struct cpu` structure. The structure contains a platform-specific part called `mcpu` ('m' standing for 'machine'). The `mcpu` structure contains

- a pointer to the kernel stack,
- a variable counting the number of register windows which are currently in the userspace window buffer,
- the userspace window buffer itself.

In Solaris macros for retrieving the pointer to the kernel stack and the window buffer from this in-memory structure are defined. They are used from the `sys_trap` routine.

In Solaris the scratchpad registers can not be used for snapshotting these pointers, since the scratchpad registers are used for different purpose, as described in Section 8.2.2.

8.7 Niagara Input and Output

Both the Simics-simulated Sun Fire T2000 server and the real Sun Fire T1000 Enterprise server contain no framebuffer. Input and output is possible only via a serial line. The hypervisor API provides hypercalls for putting a character to the serial output and for reading a character from the serial input. HelenOS uses purely these hypercalls for input and output, no special device driver is implemented (even though the input and output functionality is contained in a separate module in the `drivers` directory, it is not a real device driver, but a kind of pseudodriver).

8.7.1 Output from Kernel

Output from the kernel is very simple; to print a character, the kernel calls the `CONS_-PUTCHAR` hypercall, passing the character to be printed in the `%o0` register.

The only problem the pseudodriver must cope with is that if the output buffer is full, the hypercall will refuse to print the character and it will return the `EWOULDBLOCK` error code. Since it takes only a very short instant until the buffer ceases to be full, the pseudodriver just spins as long as the `EWOULDBLOCK` code is being returned. Once the `EOK` code is returned, the pseudodriver knows the character has been successfully printed.

Surprisingly, on the Simics-simulated machine HelenOS never noticed hypervisor returning the `EWOULDBLOCK` code, the output worked even without the spinning. On the real machine, on the other hand, even printing the very first character failed (probably because the output buffer became full due to OBP). Adding the spinning was one of the very few things which had to be adapted to make HelenOS run on a real Niagara machine. The author thinks that `EWOULDBLOCK` was never returned because Simics simulates a machine with a 5MHz processor, which is a far cry from the real processors, so the Simics machine is too slow to make the output buffer full.

8.7.2 Input to Kernel

To read a character from the serial line, HelenOS input pseudodriver issues the `CONS_GETCHAR` hypercall. The hypercall takes no input parameters. If there is a pending character in the input buffer, the hypercall picks the character from the buffer, returns `EOK` as an error code and returns the character read in the `%o1` register.

Since the input is performed by actively requesting the hypervisor, the input pseudodriver works in a polled mode. This is essential because the hypervisor may not notify the kernel via interrupts.

8.7.3 Userspace I/O Driver

HelenOS is a microkernel, so userspace device drivers are ordinary user processes (tasks). Before dwelling on the details of the userspace Niagara I/O driver, let us summarize how a HelenOS task can communicate with a device. The tasks may

- map a region of the device memory into their address space,
- register themselves to be notified of interrupts and
- ask the kernel to execute a few load and store operations upon an interrupt; the load and store operations to be executed are specified by means of so called *pseudocode*.

It is not possible, however, for a task to issue a hypercall. Without an additional support from the kernel, the userspace tasks have no way how to make input and output operations purely by hypercalls. In HelenOS the userspace I/O drivers have their kernel counterparts. The counterparts provide that additional support for the userspace drivers.

Input to Userspace

If the input is not grabbed by the kernel (i.e. the system is not switched to the kernel console) and the kernel detects that there is a character to be read, the kernel input driver reads the character, saves it to a temporary storage and notifies the userspace of a fictional interrupt. The value of the character read is passed to the userspace via a pseudocode (a new pseudocode instruction `CMD_NIAGARA_GETCHAR` is defined for this purpose).

Output from Userspace

Userspace output is done by writing the output to an area of memory shared between the userspace output driver and the kernel output driver. The area is owned by the kernel output driver, which publishes the area's real address and size and the userspace counterpart just maps this area into its own address space. The kernel output driver regularly checks the area (in the same thread as the keyboard polling takes place) and prints any characters to the output using hypercalls.

Conclusion

The way how the I/O operations are performed from the userspace is a little bit cumbersome. Anyway, it is one of few ways how it can be implemented if we do not want to bother with a specialized I/O device driver. On Niagara, the I/O device is called QCN, but its documentation is not publicly available. Implementing QCN device driver would require additional reverse engineering, as in case of the Serengeti console (see Section 7.5.2). Since the machines equipped with a Niagara processors are server machines, implementing the I/O driver is not a focal point. Thus the provisional I/O driver is pretty sufficient.

8.8 Multiprocessing

Niagara is a heavily parallel architecture, therefore support for multiprocessor configurations is a crucial issue of every operating system with Niagara support. It is important to emphasize that by *support for multiprocessors* on Niagara we mean the ability of an operating system to make use of all the *logical* processors (Niagara-based machines can contain only one physical processor chip). Every time we use the term *processor* or *CPU* in this chapter, we will always mean a *logical processor*.

Adding support for multiple processors to HelenOS was very tricky. The basic problem was to make the application (i.e. non-bootstrap) processors execute our code.

8.8.1 Waking the Application Processors up

In the original UltraSPARC HelenOS port the application processors are woken up by means of an OpenBoot client API call. Since the Niagara port tries to avoid using the OpenBoot and prefers the hypervisor API calls, the first thing the author of this thesis tried was issuing a `CPU_START` hypervisor API call. This call takes the following arguments: `cpuid` of the processor being woken up, the value of the `PC` register to be set on the processor being woken up, and the value to be written to the `%o0` register of the processor being woken up.

Unfortunately, the `CPU_START` call returned an `EINVAL` error code when issued. This means that the CPU which was about to be woken up was not in the *stopped* state, but in the *running* state. The `CPU_START` hypercall simply refused to wake a processor which was already running. The reason why the application processors are already running when the HelenOS bootloader is passed control remains a mystery. The official documentation states that after switching the machine on all the application processors must be stopped and only the bootstrap processor must be running. Maybe it is the OBP who wakes the processors up. Maybe the Niagara documentation and the actual implementation do not fully correspond. Anyway, in order to make the application processors execute HelenOS code, they must be stopped first and then woken up again (having their `PC` register set to some address in the instruction memory containing HelenOS code). Stopping the processor is unfortunately tricky as well.

8.8.2 Stopping the Application Processors

The hypervisor API provides developers with a `CPU_STOP` hypercall. The hypercall, as apparent from its name, should stop a running logical processor. However, when the hypercall is issued without special provisions, it always returns the `ENOTSUPPORTED` error code. The error code indicates that the hypercall is not supported by the platform.

Experiments with trying to stop (or directly wake up) the processors via the OBP failed.

8.8.3 Setting the API Version

In the hypervisor documentation [hypervisor] in chapter number 11 there is a mention about so called *API versioning*. It states that in order to be able to use certain hypervisor API calls (such as `CPU_STOP`), the hypervisor API must be *explicitly* set to a higher version first using a special hypercall. The `CPU_STOP` hypercall is a part of the 1.1 version of the hypervisor API, whereas the default hypervisor API version is set to 1.0. Neither setting the hypervisor API version to 1.1 helped, though.

8.8.4 Problem Setting the API Version on Simics

For setting the hypervisor API version a special trap with a dedicated trap number is used (so called *core trap*). Before issuing the `ta` instruction the desired version of the API must be encoded to the output registers.

On a real machine the core trap succeeds. On the Simics machine, however, it returns an `EBADTRAP` error code. The error code indicates that the trap number or the function number supplied is wrong. By inspecting the disassembled piece of memory of the hypervisor the author found out that the hypervisor on the Simics machine will always return the `EBADTRAP` error code, no matter which function number is supplied.

Due to this fact the author temporarily gave up porting HelenOS to the Simics-simulated Niagara machine and concentrated purely on the real machine. Eventually the way how to wake processors up on a Simics-simulated Niagara was discovered. See Section 8.8.9 for more details.

8.8.5 Problem Stopping CPUs on a Real Machine

The hypervisor API version can be set to 1.1 on a real machine without any problems. Nevertheless, even when the API was successfully set to 1.1, the `CPU_STOP` hypercall still returned `EINVAL` every time it was issued.

8.8.6 The Problem Solved

The author asked Tomáš Hrubý, the author of the Niagara port of the OK-L4 system, to try to run HelenOS on a different Niagara-based machine than the one which HelenOS was originally tested on. This time not only that the hypervisor API version could be successfully set to 1.1, but even stopping the CPU using the `CPU_STOP` hypercall succeeded. Consequently, the version of the hypervisor was to be suspicious of.

On the Niagara machine HelenOS was originally developed on, the 1.3.4 version of the hypervisor was installed. On the machine which belonged to Tomáš Hrubý, the 1.5.2 version of the hypervisor was installed.

Upgrading the firmware solved the problem. With the newest version of the firmware the CPUs can be stopped without any problems. Once stopped, they can be woken up by the `CPU_START` hypercall and made execute the HelenOS code.

8.8.7 Start of the Application Processors

The total number of CPUs is detected in the `smp_init` function by iterating through all the 'cpu' nodes of the machine description and adding 1 to the counter for each 'cpu' node encountered.

Waking the application processors up is done in the `kmp` function. First, all the application processors are stopped. The CPU does not stop immediately after issuing the `CPU_STOP` hypercall, but a short time later. After issuing the `CPU_STOP` hypercall, the kernel actively waits for the CPU to be brought into the *stopped* state. The state of the CPU can be tested via a `CPU_STATE` hypercall:

```
__hypercall_fast1(CPU_STOP, i);  // stop CPU with cpuid = i

// actively wait for the CPU to stop running
uint64_t state;
__hypercall_fast_ret1(i, 0, 0, 0, 0, CPU_STATE, &state);
while (state == CPU_STATE_RUNNING) {
  __hypercall_fast_ret1(i, 0, 0, 0, 0, CPU_STATE, &state);
}
```

Once the processors have stopped, they may be woken up. The program counter is set to the very first instruction of the code in the `start.S` file (i.e. the same instruction which is executed right after the control is passed to the kernel from the bootloader on the bootstrap CPU). In the `CPU_START` hypercall the start of the physical memory is passed. The application processor will find the value in its `%o0` register.

The steps made by the application processor are very similar to those made by the bootstrap processor. Let us summarize them.

1. The starting address of the physical memory is extracted, the basic runtime environment (the `PSTATE` register, the window configuration registers, etc.) is set up. The kernel trap table is set.

2. The initial mapping is set up in the MMU. This process is identical to taking over the MMU on the bootstrap CPU (even though a slightly simpler process would be sufficient, since the APs start with MMU disabled, so there is no danger that they would overwrite the mapping of the code they are executing).

3. The fault status area is set. The `cpuid` is written to the first scratchpad register, the address of the fault status area is written to the second scratchpad register.

4. The MMU is explicitly enabled. This is a difference from the bootstrap processor, where the MMU was already enabled by the OBP.

5. A temporary stack is configured. The temporary stack which was initially used by the bootstrap processor is reused by all the application processors during their startup phase (the application processors start up sequentially, so their startup phases never interleave).

6. Kernel jumps to the generic function called `main_ap`, which performs AP-specific initialization steps.

8.8.8 Waking CPUs up: Comparison with the Original Port

In the original UltraSPARC port the application processors are woken up by means of an OBP API call. There are no such issues as on Niagara – the processors are already stopped when HelenOS boots, so they can be easily woken up.

In the original port another issue arises, though. Since the OPB API may not be used once the kernel has booted (HelenOS kernel takes the full control over the machine, shooting down the OBP), it is the HelenOS bootloader who actually wakes the processors up. When the application processors start, they make a basic initialization and then continue spinning in a loop. A CPU stops spinning when a `waking_up_mid` variable becomes equal to the CPU's MID. The `waking_up_mid` variable is set by the kernel in the `kmp` function.

8.8.9 Waking Processors up on Simics

The application processors can be made execute HelenOS code even on Simics. The way how it is achieved is not very straightforward, it is a kind of trick.

As mentioned in Section 8.2.1, the initial virtual-to-real mapping established by the OBP does not map the virtual address zero onto the start of the real memory, but onto the address which is 4 M further. It has also been mentioned that the OBP stores some of its data structures to this 4MB piece of memory. Besides other things, the OBP puts its trap table into that area.

When the kernel boots, the application processors are already running (see Section 8.8.1), most probably looping in some piece of OBP code. In order to make the application processor execute HelenOS code, the bootstrap processor can:

- replace the code of the OBP inter-processor interrupt handler with a piece of HelenOS code (the OBP IPI handler is located somewhere in the first 4 M of the real memory), and
- send an inter-processor interrupt to the CPU being made execute HelenOS code.

The piece of code which is copied to the first 4MB area of the real memory is located in the `start.S` file. Its main job is to jump to the first instruction of the kernel.

As for now, the address of the OBP inter-processor interrupt handler is hardwired in the code. It was obtained by reverse-engineering using Simics. The only purpose of the SMP on Simics was to debug some issues which would be too hard to debug on a real machine. Only one AP can now be running on Simics. Should there ever be a need to implement a thorough support for SMP on the Simics-simulated Niagara machine, the mechanism would have to be adapted a little bit. Most probably the bootstrap processor would have to read the value of the `TBA` (trap base address) register and save it to a global variable, where the code for replacing the OBP inter-processor interrupt handler would pick it up from.

8.8.10 Inter-processor Interrupts

The registers which are used to send and analyze the inter-processor interrupts (IPIs) on sun4u have been made hyperprivileged on sun4v. The *Interrupt Vector Trap*, which is invoked upon the receipt of an IPI on sun4u, has also been made hyperprivileged on sun4v. On sun4v the inter-processor interrupts are sent and analyzed via hypervisor API calls and they are reported via a new trap called *CPU Mondo Trap*. *CPU mondo* is a term used to refer to a CPU-to-CPU interrupt message, as opposed to so called *Device mondo*, which is used to refer to an interrupt sent by an I/O device.

Initialization of the CPU Mondo Queues

In order to be able to receive the IPIs, each processor must configure so called *CPU mondo queue*, which is a buffer where the hypervisor stores IPI messages upon their receipt. The queues are declared in the `kernel/arch/sparc64/src/trap/sun4-v/interrupt.c` file: the global variable `cpu_mondo_queues` is a two-dimensional array, whose first dimension index is the ID of the CPU and whose second dimension index is the index of the particular 64-bit word inside the CPU's CPU mondo queue. The CPU mondo queues are then set up using a `CPU_QCONF` hypercall. HelenOS configures the CPU mondo queues to contain 8 entries. Since each entry (according to the hypervisor documentation) takes up 8 64-bit words, the size of a particular queue is 8 $\times$ 8 $\times$ 64 bits = 512 bytes.

Sending CPU Mondos

The CPU mondos (IPIs) are sent via a `CPU_MONDO_SEND` hypercall. The hypercall takes the CPU mondo message and a list of CPU IDs as its parameters (notice that on sun4u the IPI must be sent separately for each CPU, it is not possible to send as single IPI to a group of CPUs).

The code invoking the hypercall can be found in the `kernel/arch/sparc64/s-rc/smp/sun4v/ipi.c` file. The implementation is split into three functions. One of them sends the IPI to a given set of CPUs, another one sends the IPI to one particular CPU (this one is used to make an AP execute the HelenOS code on Simics, see Section 8.8.9) and the last one broadcasts the IPI to all the CPUs (this one is used to make a cross-call).

Receiving CPU Mondos

When a CPU receives a CPU mondo, the hypervisor stores the mondo message into the queue the processor has defined, and notifies the privileged code by jumping to the *CPU Mondo Trap* handler in the privileged trap table. The position at which the most up-to-date mondo message is located is indicated by the value of the `CPU_MO-NDO_QUEUE_TAIL` register. The `CPU_MONDO_QUEUE_TAIL` register is an ASI-accessible register which contains an offset to the first byte of the last mondo message received.

In HelenOS the trap is handled by the `cpu_mondo` function defined in the `ker-nel/arch/sparc64/src/trap/sun4v/interrupt.c` file. The function picks the mondo message up from the mondo queue. The position to which the next mondo message will be stored is represented by the value of the `CPU_MONDO_QUEUE_HEAD` register. The `CPU_MONDO_QUEUE_HEAD` register is an ASI-accessible register which contains an offset to the first byte of the next mondo message to be stored to the buffer. It is a responsibility of the `cpu_mondo` function to modify the `CPU_MONDO_QUEUE_HEAD` register after reading the mondo message. The `CPU_MONDO_QUEUE_HEAD` value is incremented by the size of the mondo message (8 bytes), modulo the size of the buffer (8×8 bytes).

8.8.11 Optimal Scheduling

The last issue having something to do with multiprocessing is preventing the threads from being unevenly distributed over the physical processor cores.

Motivation

As outlined in Section 5.1, the T1 processor chip consists of several physical processor cores, each core containing several logical processors. Whereas threads distributed over different physical cores are really running in parallel, threads distributed over different logical processors of the same physical core alternate with each other each processor cycle.

Imagine a processor chip has four physical cores, each core having four logical processors. Imagine there are exactly four threads running on the system. When those four threads are evenly distributed over all the physical cores, the total throughput is four times higher than when all the threads are distributed over different logical processors of the same physical core, thus having to share the performance of the single core.

Our goal will be to prevent the scheduler from distributing the threads unevenly among the processor cores.

Suboptimal Scheduling without Modifications

In HelenOS on each CPU there is a load balancing thread. The thread is sleeping most of the time, being periodically activated (in the current implementation it is activated once a second). Load balancing thread ensures that the number of ready threads on a particular CPU will be roughly the same as the average number of ready threads per CPU.

At first sight it may seem that thanks to the load balancing thread the threads will be evenly distributed over the physical cores – if there is a roughly equal number of threads on every logical CPU and there is the same number of logical processors per one physical core, there will be roughly the same number of threads per a physical core. It is, however, not so simple.

The problem is that if the average number of ready threads per CPU is not a whole number, there will always be (small) differences in the number of ready threads on particular CPUs. And if the CPUs which have the smaller number of ready threads belong the same core and/or the CPUs which have the greater number of ready threads belong the same core, the load of the physical cores will be imbalanced. In the example from the last section we had four threads in the system which where distributed over four logical processors, which seemed a perfectly even distribution. However, all the CPUs with one ready thread belonged to the same physical core and all the CPUs with no ready threads belonged to the rest of the physical cores. Thus a great load imbalance arose.

Solution

The solution is that the load balancing thread checks not only how many threads to steal from other CPUs in order to make the CPUs evenly loaded, but also how many threads to steal from other CPUs in order to make the *cores* evenly loaded. In the load balancing thread of a CPU `c` which belongs to core `core1` a calculation described by this pseudocode is performed:

```
average := readyThreadsTotal / physicalCoresCount
toSteal := average - core1.readyThreads
if toSteal <= 0
    return
for i := 1 to toSteal
    let CPU c be the most idle CPU of core1
      (when including the threads to steal)
    c.proposedThreadsToSteal++
```

When deciding how many threads to steal, a maximum of these two values is taken:

- average CPU's thread count - `c.readyThreads`, and
- `c.proposedThreadsToSteel`.

Thanks to taking the second value into account a thread will be migrated even though in the previous implementation it would not be migrated at all.

The implementation does not steal any threads for a processor if the processor belongs to a physical core which is overloaded. Stealing threads from other processor cores by an overloaded processor core makes no sense. Stealing threads from the same core is intensionally avoided in the overloaded cores, since a thread can not be migrated multiple times in HelenOS and we would like the threads of the overloaded core be stolen by other cores. So, the load balancing thread first ensures that all the cores are equally loaded and only then it ensures that the processors within a particular core are equally loaded.

Detecting the Physical Cores

The last topic concerning the threads distribution the reader might be interested in is the way how to detect which physical core a given logical processor belongs to. On Niagara each physical core has just one integer execution unit (which is not shared with other physical cores)[3]. Each integer execution unit is represented by its own node in the machine description. The execution unit nodes are children of the CPU nodes[4]. In the HelenOS kernel a data type called `exec_unit_t` has been defined, which encapsulates, apart from other things, list of all CPUs belonging to the same processor core (i.e. sharing the same integer execution unit). The execution units are detected in the `detect_execution_units` function in the `kernel/arch/sparc64/src/-smp/sun4v/smp.c` file. Apart from detecting the execution units the function also determines the total number of CPUs.

[3]There is, however, just one floating point unit per one Niagara processor chip, shared by all the physical cores, thus by all the logical processors.

[4]Note that since the execution units are shared by multiple logical processors, the machine description is rather a DAG than a tree.

8.9 UltraSPARC Architecture 2007

UltraSPARC Architecture 2007 is a descendant of the UltraSPARC Architecture 2005 specification. It is obeyed by the UltraSPARC T2 processor model.

There are certain changes between the T1 and T2 processor models, some of the changes affect the hyperprivileged mode, some of them affect the privileged mode. Changes affecting the hyperprivileged mode are not interesting for us, because operating system kernels never run in the hyperprivileged mode. What is interesting for HelenOS developers are the differences between the T1 and T2 processor models which affect the privileged mode. There are only two such differences:

- The first one concerns the *Floating-Point Status* register's *Floating-Point Trap Type* field, which is not used in HelenOS, so it is not important.
- The second one is slightly more interesting. The *Instruction Access Exception* and *Data Access Exception* traps have been replaced with more specific traps. For the *Data Access Exception* these are:
  - DAE_invalid_asi,
  - DAE_nc_page,
  - DAE_nfo_page,
  - DAE_privilege_violation, and
  - DAE_side_effect_page.

  For the *Instruction Access Exception* these are:
  - IAE_nfo_page,
  - IAE_privilege_violation, and
  - IAE_unauth_access.

  All these traps are described in the UltraSPARC Architecture 2007 specification.

All the new traps have been added to the HelenOS trap table. Body of all the entries of the DAE_* traps is the same as for the original *Data Access Exception* trap. Similarly, body of all the entries of the IAE_* traps is the same as for the original *Instruction Access Exception* trap. This implies that even though HelenOS is informed about the specific cause of the trap, the information is not used and HelenOS acts as if it received the old *Data Access Exception* or *Instruction Access Exception* traps.

HelenOS has not been tested on any machine equipped with an UltraSPARC T2 processor (simulated nor real). Nevertheless, the changes between the T1 and T2 models are so subtle that it is possible to claim that HelenOS (at least theoretically) supports the UltraSPARC T2 processor.

Chapter 9

Related Work

9.1 Original HelenOS SPARC Port

This thesis has been meant as a continuation of Jakub Jermář's master thesis [jj_thesis], which concerned porting HelenOS to the older 64-bit SPARC models (the implementation of the original port is briefly described in Chapter 6). It is therefore natural that the original port is the most influential work for this thesis. The source code of the port to the JPS-compliant processors is (except for the Serengeti console driver, see Section 7.5.2) in fact a modification of the source code of the original SPARC port. The source code of the port to the Niagara-based processors often required writing brand new pieces of code, but they were massively inspired by the original port as well, mainly the bootloader, taking over the TLBs, kernel MMU misses handling and the preemptible trap handler.

9.2 Other HelenOS Ports

9.2.1 Overview of HelenOS Ports

Table 9.1 summarizes and compares the existing HelenOS ports to different processor architectures.

architecture	**real HW**	**SMP**
amd64	yes	yes
arm32	no	no
ia32	yes	yes
ia64	yes	yes
mips32	no	no
ppc32	yes	no
sparc64	yes	yes

Table 9.1: Comparison of the Current HelenOS Ports

9.2.2 Inspiration by the Other Ports

Since the original SPARC port allows all the HelenOS features to be used on older UltraSPARC processors, there was little need of looking for inspiration in HelenOS ports to completely different architectures. Some of them, however, served as a source of inspiration for solution of particular problems.

In the source code of the *mips32* port of HelenOS (specifically the MSIM input and output driver) the author found how the serial input and output is controlled. Portions of the code have been borrowed and used by the input and output drivers of both the Serengeti machine and the Niagara-based machines.

Some pieces of inspiration also come from the *ppc32* port, specifically from the sources which are shared between the ppc32 and the sparc64 ports which concern mainly the OBP. A deep understanding of the way how the OBP client interface is accessed by the bootloader and the kernel was useful when the bootloader was being modified to work on the newer UltraSPARC machines and on the Niagara-based machines and also when the function for setting the color palette was being written (see Section 7.6.1).

9.3 Solaris

Solaris is the most mature operating system supporting the SPARC processors. It supports the widest range of SPARC processors and exploits the biggest set of features the processors provide. The majority of information about its kernel may be obtained from [sol_internals].

Solaris has also been a good source of inspiration during porting HelenOS to the newer SPARC processors. The most useful piece of code investigated was the driver of the Serengeti machine console (SGCN). Since the Serengeti machine console (SGCN) has no publicly accessible documentation, the author reverse-engineered the Solaris SGCN driver and used the information obtained to implement an analogous driver for HelenOS (see Section 7.5.2 for further details).

9.3.1 Comparison with HelenOS

Even though Solaris is a much more advanced operating system than HelenOS, there are a lot of problems both of the systems must cope with. Let us have a look at those which have some connection to the JPS-compliant and Niagara-based processors. For a comparison of HelenOS and Solaris which focuses the SPARC processors in general, see [jj_thesis].

Serengeti Console

Since the Solaris implementation acted as a source for reverse-engineering, the HelenOS implementation is naturally much simpler. The address where the shared buffers are present in the physical memory address space is hard-wired. The HelenOS implementation re-uses the shared buffers which have already been configured by the OBP and does not configure them on its own. It is not notified of the keystrokes via interrupts, but it uses polling instead. For more details, see Section 7.5.2.

Separation of the sun4u and sun4v Sources

There is a subtle difference between HelenOS and Solaris in a way how the source files for the sun4u and sun4v architectures are separated. Both systems put the architecture-specific source files to different subdirectories called sun4u and sun4v, but both make the branch on different level of the directory structure. In Solaris the sun4u and sun4v subdirectories are almost at the top of the directory structure, in HelenOS they are the leaf directories. Fore further details, see Section 8.1.3.

Usage of Scratchpad Registers

The sun4v processors offer a handful of read-write registers (called *scratchpad registers*, see Section 5.5.1), typically four of them can be used by the privileged software for an arbitrary purpose. In both Solaris and HelenOS to the first scratchpad register the ID of the CPU is written at the boot stage, which is read from the time-critical sections of the kernel code, such as the MMU miss handlers. In both systems the second scratchpad register holds the address of the MMU fault status area (memory area where the hypervisor stores a description of an MMU exception, see Section 5.3.3), which is also used from the time-critical sections of the kernel code. On the other hand, the third and fourth scratchpad registers are used for a different purpose. In HelenOS, they are used to hold the addresses of the kernel stack and the userspace window buffer respectively, in Solaris they hold the base address and the size of the first and the second TSB of the active userspace process respectively. For details see Section 8.2.2.

Hypercalls

HelenOS defines a set of functions in the forms of `__hypercall_fast`*n*, `__hypercall_hyperfast` or `__hypercall_fast_ret`, where *n* is the number of arguments the hypercall takes. These functions take the function number of the hypercall and the input arguments of the hypercall, they return the error code and (some of them) the output of the hypercall.

Solaris defines no such generic functions, but for each hypercall type it defines a separate wrapper function. For details, see Section 8.3.

Preemptible Trap Handler

In Solaris there is a routine analogous to the preemptible trap handler which is called `sys_trap`. Let us now concentrate purely on the differences connected to the sun4v architecture. For general comparison of the HelenOS and Solaris SPARC preemptible trap handler, see [jj_thesis].

The main difference is in the way how the kernel stack and the userspace window buffer of a thread are snapshotted when a trap occurs in userspace. Whereas Solaris stores the addresses of the stack and the buffer to an in-memory structure, HelenOS uses the third and the fourth scratchpad registers respectively for that purpose. See Section 8.6.2 for more details.

9.4 Linux

Linux is the second most mature operating system with the SPARC processors support, right after Solaris, with the second widest range of supported SPARC processor models. It has also been a good source of inspiration during porting HelenOS to the newer SPARC processor models, yet not that useful as Solaris, since Linux is poorly commented and documented. There is no crucial piece of Linux code which helped the author during the work on the port (unlike in Solaris, whose SGCN driver implementation helped in a considerable way). The majority of information about the Linux kernel may be found in its source files. For further information, it is recommended to visit the blog [davem] of David Miller (a person who ported the Linux kernel to the SPARC processors).

9.4.1 Comparison with HelenOS

Linux is much more advanced than HelenOS and it exploits a much wider set of SPARC processors features than HelenOS. Both systems, however, cope with a big set of similar problems. Let us now focus on the differences which concern the JPS-compliant and Niagara-based processors.

Separation of the sun4u and sun4v Sources

The way how the sun4u and sun4v source files are separated differs significantly between Linux and HelenOS. Whereas HelenOS separates the sun4u-specific and sun4v-specific source files by putting them into separate directories (keeping the generic sources in common directories), Linux does not separate the source files at all. The sun4u-specific and sun4v-specific pieces of code are intermixed and it is decided at runtime which ones will be executed. Sometimes the sun4v-specific subroutines are integrated into the existing code in a way which is not very clean. See Section 8.1.3.

Usage of Scratchpad Registers

The usage of the scratchpad registers is exactly the same as in Solaris. See Section 9.3 for an overview of differences between HelenOS and Solaris, see Section 8.2.2 for details on the scratchpad register usage in HelenOS and Linux.

Hypercalls

The way how functions performing the hypercalls are defined is exactly the same as in Solaris. See Section 9.3 for an overview of differences between HelenOS and Solaris, see Section 8.3 for a detailed description of the functions performing the hypercalls in HelenOS and Linux.

Handling Kernel MMU Misses

Operating systems must cope with the fact that the SPARC MMU miss handler must fit into 32 instructions, otherwise branching would be inevitable. Neither the HelenOS nor the Linux kernel MMU miss handlers fit into 32 instructions. Whereas in the HelenOS handler the branching is apparent and straightforward, in Linux it is not apparent at the first sight. In Linux the branch is done at the very first instruction of the handler code, but the branch instruction is added to the instruction memory at runtime, when the code is being patched (patching the source code at runtime is an ugly trick Linux uses to incorporate sun4v-specific pieces of code into the existing sources). For further details, see Section 8.5.2.

Translation Storage Buffers

Whilst HelenOS uses only one TSB for every memory context, there may be two TSBs in Linux, one for small and one for big pages. Moreover, Linux TSBs may have dynamic size. See Section 8.5.3 for further details.

Chapter 10

Conclusion

10.1 Enhancements

Support for the 64-bit SPARC processors was originally added to the HelenOS system by Jakub Jermář as a part of his master thesis [jj_thesis].

Table 10.1 summarizes which processor models were supported in the original port and which models have been added as a part of this thesis.

in original port	added thanks to this thesis
UltraSPARC I	UltraSPARC III
UltraSPARC II	UltraSPARC III+
UltraSPARC IIi	UltraSPARC IIIi
	UltraSPARC IV
	UltraSPARC IV+
	UltraSPARC T1
	UltraSPARC T2[1]

Table 10.1: Supported CPU models

Support for particular machines equipped with the SPARC processor has been extended as well. Table 10.2 summarizes which machines were supported in the original port and which ones have been added as a part of this thesis.

in original port	added thanks to this thesis
Sun Enterprise E6500	Sun Fire 6800 (Serengeti)
Ultra 5	SunBlade 1500
Ultra 60	Sun Fire T1000 Enterprise
	Sun Fire T2000

Table 10.2: Supported Environments

[1]Even though the UltraSPARC T2 processor is theoretically supported, the author had no chance to test it either on a simulator or on a real hardware. There are not many differences between the T1 and T2 processors in the non-privileged and privileged mode. The T2-specific properties have been reflected in the code; they are not tested, though. For more details on the UltraSPARC T2 support see Section 8.9.

10.2 Achievements

The author has achieved all the goals which have been outlined in Chapter 1. The number of new platforms supported is very wide, as outlined in Section 10.1, as well as the number of new processor models. SPARC architectures are known to be very diverse, which puts extra demands on the implementor. Despite this fact, HelenOS is now able to run on two different simulated and two different real machines. On the simulated Serengeti machine HelenOS works even with several configurations.

Despite its breadth, the implementation is deep in a sense that almost all the features of HelenOS are accessible on the newly supported machines (with a small number of exceptions as described in Section 7.1.3). A reasonably big portion of SPARC processors' features is exploited. The reason for not exploiting some features is that sometimes it would go behind the scope of this thesis, it would make the HelenOS code less generic (portable) or that the value added would be simply small in our case.

The text of this thesis provides a good overview of the features which the SPARC processors (especially the newer ones) provide the kernel programmer with. In three chapters the properties of the newly supported processor models are explained deeply enough so that the reader is able to understand the ongoing text. Apart from aspects important for the system programmer the text mentions aspects notable for a curious reader who is interested in low-level system architecture. The implementation chapters thoroughly explain the changes which had to be made to the original HelenOS code to support the new platforms, providing comparison with Linux and Solaris where useful and applicable.

10.3 Contributions

By porting HelenOS to the newer UltraSPARC processors, HelenOS has become one of the systems with the best support for the 64-bit SPARC processors, along with Linux, Solaris or the *BSD systems. It has been confirmed again (as many times before) that HelenOS is extremely generic and portable.

By investigating the newer processor models the author has contributed to the general 'know-how' of the HelenOS project team. The thesis has been a big opportunity to solve challenging problems, such as making HelenOS boot on machines with non-standard versions of firmware, reverse-engineering the Solaris driver of a non-standard I/O device, debugging the kernel on a real machine before any output was initialized or adding SMP support to the Niagara port.

Moreover, this thesis helps the academia by both the source code and this text. It will help students and researchers in understanding the low-level aspects of operating systems and in understanding the Sun machines' and processors' architecture.

10.4 Perspectives

Porting to the UltraSPARC T1 processor paved the way for support of other sun4v processors. The UltraSPARC T2 processor is theoretically supported even now, but testing on some machine equipped with a T2 processor (simulated or real) is needed.

By the time this thesis was written, Sun released Niagara descendant called *Victoria Falls*. It is possible that in the future Sun Microsystems will release another Niagara's descendant called *Rock*. Porting HelenOS to these processors should not be a big deal, given the hypervisor shadows the kernel programmer from the very low-level details of the processor by providing a relatively stable interface.

Also, the port to the JPS-compliant processors and the sun4v processors is not perfect and it would be very interesting to solve some issues connected with it, such as

- improving SILO so that it is able to boot on the non-standard firmware machine without patching SILO binaries,
- rewriting the bootloader so that it is not limited by the constrained amount of memory to be claimed on Serengeti,
- making the Serengeti console driver use interrupts instead of polling,
- improving HelenOS to be able to exploit all the physical memory even though the physical memory address space is non-contiguous,
- adding support for the keyboard on the SunBlade 1500 machine,
- improving the bootloader so that it is able to cope with the fact that the virtual-to-real mapping is biased by 0x400000,
- writing a driver of the Niagara console (QCN).

HelenOS is now undergoing a heavy development. Let us mention for instance that the implementation of the TCP stack is in progress or the filesystem is being improved. This opens interesting horizons.

Chapter 11

Glossary

architecture

When not speaking about the 64-bit SPARC processors, the term *architecture* denotes a group of processor models which have so many properties in common that a common compiler backend is used to generate code runnable on these models. IA32, AMD64, IA64, PPC32, ARM32, MIPS32 and SPARC64 are all examples of an architecture.

When speaking about the 64-bit SPARC processors, the term architecture is used to distinguish between the older UltraSPARC I, II, III and IV-series processors (referred to as *sun4u*) and the newer UltraSPARC T1 and T2 processors (referred to as *sun4v*).

identity mapping

A virtual-to-physical (or virtual-to-real) memory mapping in which the virtual address VA is mapped onto the physical (or real) address PA according to the following formula:

$$PA = VA + C,$$

where C is the starting address of the physical (or real) address space. C is often zero.

pre-JPS

Used to refer to the older sun4u CPU models that do not conform to the Joint Programming Specification: UltraSPARC I, II, IIe and IIi.

SMC

SMC stands for *self-modifying code*. A piece of code is *self-modifying*, if it modifies an instructions stream.

subarchitecture

This term is used to distinguish between the older *sun4u* processor models (UltraSPARC I and II series) and the newer *sun4u* processor models (UltraSPARC III and IV series).

sun4u

Generally this term denotes the architecture of the Sun machines equipped with an UltraSPARC I–IV-series processor. In this thesis, the term is used to denote the UltraSPARC I–IV-series processors themselves.

sun4v

Generally this term denotes the architecture of the Sun machines equipped with an UltraSPARC T1 or T2 processor. In this thesis, the term is used to denote the T1 and T2 processors themselves.

Chapter 12

Bibliography

[US_III]	*UltraSPARC III Cu User's Manual* `http://www.sun.com/processors/manuals/USIIIv2.pdf`
[US_III_i]	*UltraSPARC IIIi Processor User's Manual* `http://www.sun.com/processors/manuals/USIIIi_UM.pdf`
[US_II_i]	*UltraSPARC IIi User's Manual* `http://www.sun.com/processors/manuals/805-0087.pdf`
[US_IV]	*UltraSPARC IV Processor User's Manual Supplement* `http://www.sun.com/processors/manuals/USIV_v1.0.pdf`
[US_IV_plus]	*UltraSPARC IV+ Processor User's Manual Supplement* `http://www.sun.com/processors/manuals/USIVplus_v1.0.pdf`
[davem]	David Miller, *DaveM's Linux Networking BLOG* `http://vger.kernel.org/~davem/cgi-bin/blog.cgi`
[dyn_tsb]	David Miller, *Dynamic TSB sizing* `http://vger.kernel.org/~davem/cgi-bin/blog.cgi/2006/03/17`
[helenos]	*HelenOS design documentation* `http://www.helenos.eu/documentation`
[hypervisor]	*UltraSPARC Virtual Machine Specification (The Hypervisor API specification for Logical Domains)* `http://opensparc-t1.sunsource.net/specs/Hypervisor-api-current-draft.pdf`
[jj_thesis]	Jakub Jermář, *Porting SPARTAN kernel to SPARC V9 architecture* `http://www.helenos.eu/doc/theses/jj-thesis.pdf`
[jps]	*SPARC Joint Programming Specification (JPS1): Commonality* `http://www.fujitsu.com/downloads/PRMPWR/JPS1-R1.0.4-Common-pub.pdf`
[sol_internals]	Jim Mauro and Richard McDougall, *Solaris Internals*, 2nd Edition.
[sparc_v9]	*The SPARC Architecture Manual, Version 9* `http://www.sparc.com/standards/SPARCV9.pdf`

[t1hp] *UltraSPARC T1 Supplement to the UltraSPARC Architecture 2005, hyper-privileged edition*
http://opensparc-t1.sunsource.net/specs/UST1-UASuppl-current-draft-HP-EXT.pdf

[t1p] *UltraSPARC T1 Supplement to the UltraSPARC Architecture 2005, privileged edition*
http://opensparc-t1.sunsource.net/specs/UST1-UASuppl-current-draft-P-EXT.pdf

[us2005hp] *UltraSPARC Architecture 2005, hyperprivileged edition*
http://opensparc-t1.sunsource.net/specs/UA2005-current-draft-HP-EXT.pdf

[us2005p] *UltraSPARC Architecture 2005, privileged edition*
http://opensparc-t1.sunsource.net/specs/UA2005-current-draft-P-EXT.pdf

[wiki] *HelenOS project wiki*
http://trac.helenos.org/wiki

[wiki_t1] *UltraSPARC T1 article on Wikipedia*
http://en.wikipedia.org/wiki/UltraSPARC_T1