6

Virtual Memory Management

6.1 Introduction

Memory is central to the operation of modern computer systems such as PARAM. In order to execute a program (or collection of parallel tasks), it must be stored in main memory. Once the program is loaded in memory, program execution is initiated by supplying its starting address to the CPU (or to nodes in the multiprocessor system). The CPU then, sends address of instruction or data to be accessed to the memory unit. This source of this address is completely unknown to the memory unit and it does not distinguish CPU requests whether it is for accessing instruction or data. The source of these address may be PC (programmer counter) for instruction request, index (for array access), literal address (for fixed location such as a well know port address), and so on. To manage all these issues, operating system must provide a unit called memory manager. It is responsible for managing memory requests associated right from the end-user request for program execution to the program termination including dynamic memory requests generated by the program during its execution.

Modern computer systems demand execution of multiple programs at the same time. Hence, it is necessary to have multiple in the main memory in ready-to-run state and this concept is popularly called multiprogramming. Multiprogramming concept to work, it is necessary that program must be able to execute successful irrespective of memory location where it is loaded in main memory for execution. From this it can be inferred that, the memory access quests (address) generated by the program (logical- or virtual address— LA) is not the same as the one to be used to access (physical address— PA) information from the main memory. That is, program generates logical address whereas, memory unit request physical address. This view differences is managed by the virtual memory manager which maps logical memory to physical memory as shown in Figure 6.1.

Each program (hereafter called task) has an associated address map (maintained by kernel, hereafter known as microkernel) which control the translation of virtual addressees in the task's address space into physical addresses. As is true virtual memory systems, the contents of the entire address space of any given task is most likely not completely resident in physical memory at any given time. There must exist mechanisms to use physical memory as a cache for virtual address spaces of tasks. Unlike traditional virtual memory designs, the PARAS kernel does not implement all of the elements of this caching itself; it encourages to allow user mode tasks the ability to participate in these mechanism. These mechanisms will be supported through users mode task's what is popularly known as subsystems. However, such subsystems are not available under PARAS (and future PARAS might have this support. Note that, the MACH microkernel support virtual memory mechanisms through memory objects.)

This chapter discusses memory management supported by the PARAS microkernel. It includes regions, memory model, organization, data structures, interactions. The emphasis is laid on both machine dependent portion and machine-independent portion of memory manager. Hence, the discussion

on implementation of true virtual memory subsystems is beyond the scope and it is a subject of user mode services. Note that virtual memory can be much larger than physical memory and this can be realized only when supporting subsystem are available under PARAS operating environment.

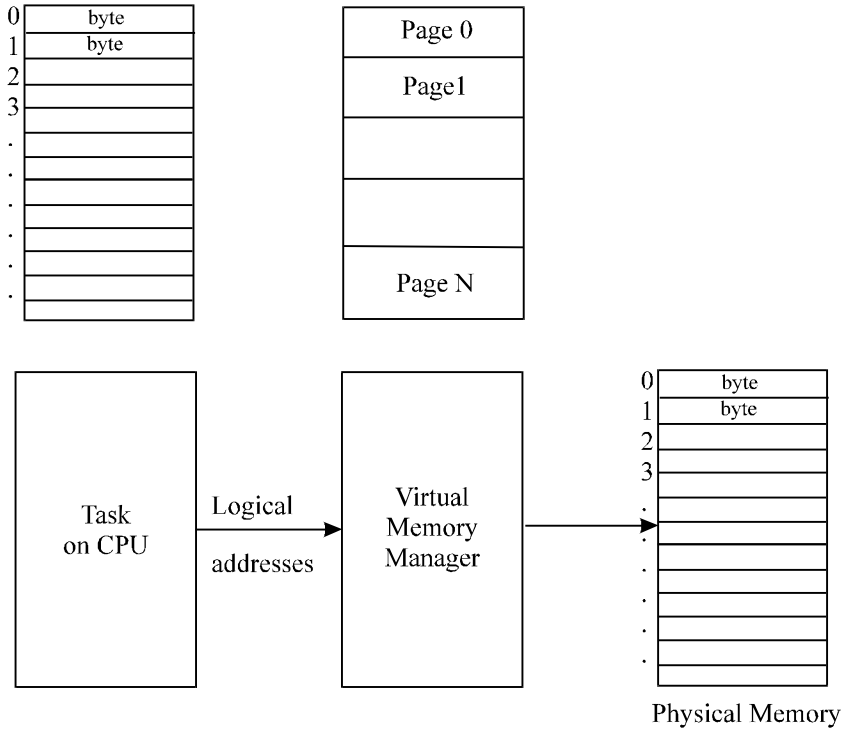

Figure 6.1: Mapping of Logical Address to Physical Memory Address

6.2 Regions

Each task has a large virtual address space within which its threads execute. The virtual address range may be sparse, with ranges of addresses which are not allocated, followed by ranges of addresses which are allocated. Address space is organized as collection of region. A “region” of an address space is that memory associated with a continuous range of addresses; that is, a start address and an end address. Regions consist of pages which may have different protection attributes, enabling a task to protect pages in its address space to allow/prevent access to that memory.

6.3 Memory model

Regions are not swapped, which means that tasks have to be resident in memory to run. This extremely simple memory model has been chosen for reasons of performance and simplicity. Having a task resident in memory allows faster interprocess communication. Also, a simple design without swapping makes the kernel simpler, smaller and more manageable.

6.4 User Virtual Memory Interface

The Microkernel virtual memory interface can be thought of as being divided into these functional groups (see Table 6.1):

- Address space manipulation including the allocation and deallocation of virtual memory
- Memory protection allowing flexible use of memory protection hardware
- Miscellaneous functions to access other tasks' virtual memory

Functional group	Primitives
Address space manipulation	□□□□□□□□□□□□□□□ and □□□□□□□□□□□□□□□□□
Protection	□□□□□□□□□□□□□□
Miscellaneous	□□□□□□□□□□□ and □□□□□□□□□□□□

Table 6.1: Virtual Memory Interfaces

Virtual memory is allocated using the primitives □□□□□□□□□□□□□□□□□ and deallocated using □□□□□□□□□□□□□□□□□□□. Freshly allocated memory is returned zero-filled.

Virtual memory protection is supported at the page level. Each allocated page in an address space has a protection attribute. This is a combination of read, write, and execute permissions. Protection codes are set using RegionProtect.

Reading/Writing to another task's address space is supported by the RegionRead and RegionWrite calls. These may be used by tasks (for example, the debugger) which want to examine and modify the address space of another task (the task being debugged).

Virtual Memory Operations

The virtual memory management operations that can be accessed by the microkernel user are discussed below:

□□□□□□□□□□□□□□□□□□□□□□**Allocate Virtual Memory**

Syntax:

□□

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□

□□□□□□□□□□□□□□□□□□□□□□

□□□□□□□□□□□□□□□□□□□□□□□□□□

Description:

□□□□□□□□□□□□□□□ allocates a region of virtual memory in the specified task's address space. If □□□□□□□□□ is false, an attempt is made to allocate virtual memory at □□□□□□□□. If □□□□□□□□ is not at the beginning of a page, it will be rounded down to one. If there is not enough space, no memory will be allocated. If □□□□□□□□□ is true, the input value of this address will be ignored, and the space will be allocated wherever it is available. In either case, the address at which the memory was actually allocated will be returned in □□□□□□□□.

Physical memory is immediately mapped for the allocated virtual memory. Initially, the pages of allocated memory will be protected to allow all forms of access. Subsequently, 'RegionProtect' may be used to change the protection attribute. The allocated region is zero-filled.

[illegible] Deallocate Virtual Memory

Syntax:

[illegible]

Description:

[illegible] deallocates a region of a task's address space, causing further access to that memory to fail. This address range will be available for reallocation. Note that because of the rounding to page boundaries, more than [illegible] bytes may be deallocated.

[illegible] Writing Virtual Memory

Syntax:

[illegible]

Description:

[illegible] allows a task's virtual memory to be written by another task. If either [illegible] bytes starting at [illegible] is not allocated, or if a portion of the address space is protected against writing, a partial write would have succeeded. The number of bytes of data actually written is returned in [illegible].

[illegible] Reading Virtual Memory

Syntax:

[illegible]

Description:

[illegible] allows one task's virtual memory to be read by another task. If either [illegible] bytes starting at [illegible] is not allocated, or if a portion of the address space is protected against reading, a partial read would have succeeded. The number of bytes of data actually read is returned in [illegible].

Protect Virtual Memory

Syntax:

Description:

sets the virtual memory access privileges for a range of allocated addresses in a task's virtual address space. The protection argument describes a combination of read, write, and execute accesses that should be permitted. The protection attributes are a set of: REGION_READ_ACCESS, REGION_WRITE_ACCESS, REGION_EXEC_ACCESS, and REGION_NO_ACCESS. On failure, protection attributes for pages in the specified address range would not have been modified.

The knowledge of processor support is very much essential for understanding and implementing memory manager.

6.5 Hardware Overview

This section is discusses machine dependent portion of memory manager. It discusses Memory Management Unit (MMU) of SUN's SPARC processor.

Overview of SPARC Reference MMU

The SPARC Reference MMU (SRMMU) provides translations from a 32-bit virtual address space to a 36-bit physical address space. All address translation information required by the SRMMU resides in physically addressed data structures in main memory. Mapping a virtual address is performed by using upto 3 levels of page tables, the first and second levels usually contain Page Table descriptors (PTP) which point to the next level page tables as shown in Figure 6.2. The third level entry is always a Page Table Entry (PTE) which points to a physical page.

The Root Pointer Page Table descriptor is unique to each process and is found in the context table discussed in the next (sub)section. The Root pointer is used to point to the root of the level 1 page table.

The Translation Lookaside Buffers (TLBs) of the MMU controller keep recently used translations cached close to the processor.

Contexts

The SRMMU can retain translations for several process address spaces at the same time, speeding up context switching between processes. Each address space is identified by a context number, which is used to index into the context table in main memory to find the root pointer of the page table hierarchy.

Caches

SuperSPARC has two large multi-way associative caches to speed up memory access (and provide high performance):

16K-Byte data cache

20K-byte instruction cache

The instruction cache is a physically addressed cache and is non-writable but is kept consistent with the data cache and external memory through extensive cache-coherence support.

The data cache is a physically addressed cache, and can be operated in either the write-back or write-through modes.

The MultiCache Controller (MXCC), the external cache controller implements a 1 MB, directly mapped, physically addressed external cache.

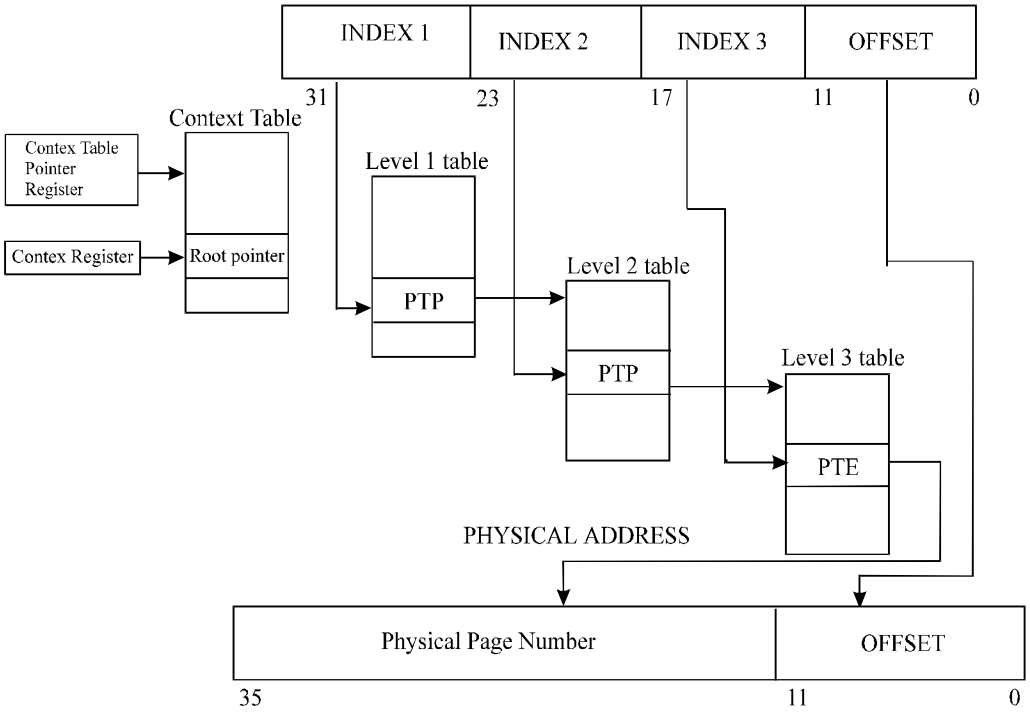

Figure 6.2: Address Translation Utilizing Four Levels of Page Tables

DVMA

The SBus is the primary I/O bus for machines conforming to the Sun4M system architecture - a system definition for a number of platforms including the SS-10 line of products, the class to which the PARAM node card falls into. Data transfers on the SBus are done through DVMA (Direct Virtual Memory Access). The advantage of DVMA is that pages need not be physically contiguous for DMA to take place; it is enough if they are virtually contiguous.

I/O Memory Management Unit (IOMMU)

The Sun4M IOMMU is used to perform address translations when Sbus masters request the bus. The IOMMU is a one-level memory based MMU which is similar to the SRMMU, but provides reduced functionality appropriate for DVMA needs.

There is a base pointer that points to a table in memory, and part of the DVMA virtual address is used to index into this table to access an IOPTE (Page Table Entry). Each IOPTE maps one 4KB page.

The table size and the corresponding DVMA virtual address range are configurable.

The IOMMU maps (of DVMA range) virtual address space for DVMA activity. DVMA addresses are always at the high end of the virtual address space.

The *DVMA base* is the highest address in the 32-bit virtual address space minus the DVMA range.

Address Space Identifiers (ASI's)

The processor modules contain items that are accessed in ASI space with virtual addresses. The Address Space Identifiers (ASI's) and virtual addresses from the SPARC IU are interpreted and used solely on these modules.

For ASI's 0x8, 0x9, 0xA and 0xB, the address is translated into a 36-bit physical address by the SRMMU; the bypass ASI's 0x20-0x2F generate the physical address without using the MMU. Other ASI's have particular functions associated with them. For example, the ASI 0x03 is used for flushing/ probing the SRMMU TLB; ASI's 0x36 and 0x37 are used to invalidate (flush clear) the contents of the instruction and data caches respectively. ASI spaces are accessed through use of the SPARC lda, sta and swapa instructions, which can be issued only in supervisor mode.

DVMA virtual addresses are translated by the IOMMU. The 36-bit physical address space is further broken down into sixteen 32-bit address spaces specified by PA (35:32).

6.6 Memory Organization

The organization of primary memory in PARAM 9000 is depicted in Figure 6.3. The microkernel uses some firmware services provided by OpenBoot (OBP). These include using OBP's serial port drivers for performing console I/O, scanning the device tree, accessing device properties, determining the amount of physical memory available on the system etc. Hence, it is necessary to allow OBP to remain resident and operational, by not reclaiming the resources (primarily memory) that is being used by the firmware.

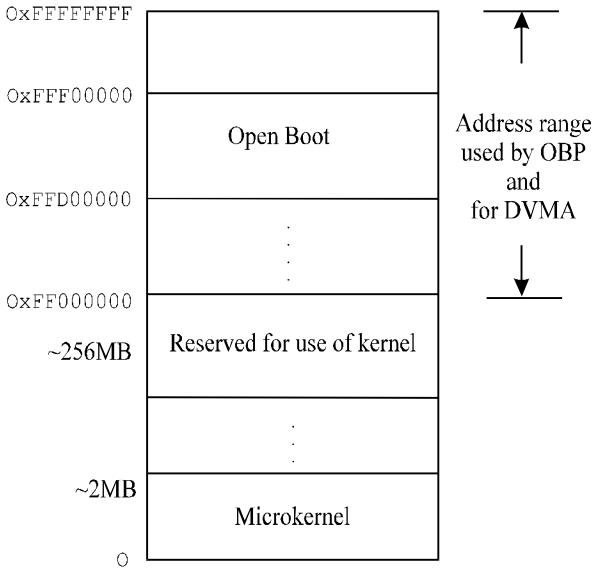

Figure 6.3: Memory Organisation in PARAM

6.7 Virtual Memory Data Structures

The microkernel sets up and uses the data structures needed for virtual address translation. The data structures needed to represent an address space are the level 1, level 2 and level 3 page tables. A context table, indexed by context number, points to the root level (level1) page table of an address space. The address space associated with a task is given by the context number that has been assigned to it on creation.

The kernel uses the context table and the page tables are for managing the SRMMU and an iommu page table for managing the IOMMU.

The context table and the iommu table are statically allocated data structures, aligned on the boundaries required by hardware while the page tables are allocated from zones. The microkernel uses 2 zones, one for level one page tables (which has 256 entries) and the other for allocating level 2 and level 3 page tables (which have 64 entries).

The page_address and page_valid arrays are maintained by the kernel for use by physical memory allocation/deallocation routines. The page_address array holds the addresses of physical pages and the page_valid array is used to determine if the corresponding physical page is free or is allocated. Both these arrays are indexed by physical page number.

6.8 Memory Management Routines - Undocumented

The pmap Interface

The pmap (acronym for physical map) is the machine dependent memory mapping data structure corresponding to the hardware representation of an address space. On SPARC, a pmap corresponds to page tables that represent an address space, and is identified by the context number associated with the address space.

The pmap is the handle with which machine-independent code communicates with machine-dependent code. All routines that operate on pmaps explicitly specify which pmap is to be operated upon.

All virtual addresses, offsets and sizes are passed to the pmap module as byte offsets. Nevertheless, operations at the page level (allocation, deallocation, changing protection attributes) are only meaningful on page boundaries.

The interface to the pmap module is split into 2 functional groups:

- pmap handle maintenance such as creation and destruction of pmaps
- Pmap range operations - primitives that change mappings for a range within a pmap.
- Use information, by which the kernel specifies which pmap is to be used on the processor.

Functional Group	Primitives
Pmap handles	pmap_create, pmap_destroy
Range operations	pmap_enter, pmap_remove
Use operation	pmap_activate

Table 6.2: Virtual Memory Interfaces

The □□□□□□□□□□□() is invoked when the system begin using a new address space. Since the kernel is part of every task's address space, the mappings for these will be set up in the new pmap also. The □□□□□□□□□□() is invoked to validate addresses. Address invalidation is performed by the □□□□□□□□□□□(). The □□□□□□□□□□□□() is used to change the protection attributes of an address range. A call to □□□□□□□□□□□□() releases all resources that are used by the pmap.

The next part of the pmap interface consists of a □□□□□□□□□□□□□□□□□□□□□□□□□□() that is used to notify the pmap module which address space is going to be used on the processor. This primitive sets up hardware page table registers, and other hardware specific registers and state related to memory management to effect the address space switch (as in context switching between threads belonging to different tasks). `pmap_activate` specifies which pmap is now going to be active on the processor.

Managing physical memory

Information about physical pages is maintained by the kernel in tables indexed by physical page number. These tables hold information of the page address and whether the page is free or is allocated. Physical memory allocation routines scan these lists to determine the address of a free page.

The pmap Operations

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

Syntax:

□□□□□□□ □ □□□□□□□□□□□()

Description:

Create and return a pmap. This routine is called to initialize a new address space. Since the kernel is part of every task's address space, its address translation information is put into the new pmap.

□□□□□□□□□□□□() - Destroy a pmap

Syntax:

□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□

Description:

Destroy the pmap and the resources that it has been using. This is called when a task dies.

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

Syntax:

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□□□□

Description:

Insert the given page (p) at the specified virtual address (v) in the specified physical map. The page is marked as a cacheable page if cacheable is set to TRUE.

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

□□□□□□
□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□

Description:

Remove the specified range of addresses from the specified pmap. The start and end addresses (s and e) are guaranteed to be rounded to page boundaries.

□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

Syntax:

□□□□□□□□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□□□□

Description:

Set the protection on the specified range in the pmap. The start and end addresses are specified by s and e. The desired protection is specified by prot. Like pmap_remove, the start and end addresses are rounded off to page boundaries.

□□□□□□□□□□□□□□□□□□□□□□□□□□

Syntax:

□□□□□□□□□□□□□□□□□
□□□□□□□□□□□□□□□

Description:

Signifies that the specified pmap is now active on the processor.

6.9 Generic OS support

The virtual memory management module of the microkernel uses the “zone package” feature - a generic kernel facility, for fast dynamic allocation of internal kernel data structures.

The zone package

This package facilitates fast allocation of memory for kernel data structures, some of which may be in time critical pieces of code. The zone package provides a simple, efficient mechanism for allocation and deallocation of many data structures used by the kernel.

Memory for these data structures is organized into zones. A zone is a collection of free data structures, managed as a linked list. If the zone is non-empty, an allocation requires a list removal operation, and deallocation a list insertion operation. It is assumed that a zone has enough elements to satisfy all requests that may arise at any point of time.

Each zone is associated with the following information:

- a name, a string used for debugging purposes
- a free list, a linked list of free elements available for allocation

- element size, the size of each element of the zone
- The maximum number of elements in that zone

The zone package is extensively used not only by the virtual memory implementation, but also by the rest of the PARAS microkernel.

Zone Operations

The zone package supports the following interfaces:

Syntax:

Description:

Create and initialize a new zone. The size of each element of the zone is specified by ‘element_size’. A maximum of ‘n_elements’ are present in this zone. The ‘name’ parameter provides a textual name that can be used for debugging purposes.

Puts the memory specified by ‘newmem’ and ‘size’ into the specified zone.

Syntax:

Description

Returns an element from the specified zone.

Syntax:

Description:

Places the specified element (‘elem’) onto the free list of the specified zone (‘zone’)

6.10 Kernel Page table initialization

When the kernel starts executing, the address translation will still be done using OBP’s page tables (set up as described in the section on Stand-alone environment). The kernel takes up control over memory

management after initializing all its data structures, including its own page tables. The following is the sequence of steps done while the kernel sets up its page tables.

1) Get the addresses of the kernel's context table and IOMMU table

These tables are statically allocated (they are in the bss segment of the kernel). These tables are (to be) aligned as required by the alignment restrictions of the processor.

2) Bring the MMU and the caches to a clean state

The contents of the on-chip instruction and data caches are invalidated (flash-cleared). The MMU TLB's contents are also invalidated. The contents of the external cache are written back to memory by performing a read of size 1 MB (the size of the cache) on some address.

3) Unmap regions of memory that are not being used by the kernel

As described elsewhere in this document, 10 MB of memory is mapped by firmware before the kernel starts executing. The portions of memory that are not used by the microkernel are now unmapped. In particular, the area from the end of kernel to 10 MB and the region between VA 0 and the start of our trap table (which is mapped just below the kernel start address) is unmapped.

4) Setup the kernel's page tables

The contents of OBPs page tables are copied into the kernel's page tables. This is done in order to retain the mappings that OBP has set up for itself, as the firmware has to be kept active. The kernel's page tables can be accessed as cacheable locations. However, all accesses to OBP's page tables (here and in step 3) have to bypass the MMU since these addresses are not virtually mapped.

5) Setup registers in MMU to start using our page tables

The microkernel is now responsible for all memory mapping operations.

6) Initialize data structures used for physical page allocation.

The amount of physical memory available and their addresses are obtained by making service calls to the firmware. This information is used to initialize the page_address and page_valid arrays.

7) Initialize the IOMMU

Memory for the CCP send and receive buffers is allocated in the DVMA range and the mappings for these are setup both in the SRMMU and the IOMMU. These buffers are treated as cacheable locations.

6.11 Operational Issues

The issues related to flushing, maintaining cache coherence, etc., are discussed below:

SRMMU Flushing

Flushing is used to purge stale translation from the TLB. The flush operation is implemented through the use of SPARC store alternate (sta) instruction in an alternate address space reserved for MMU probing/flushing. The address format encodes the type of flush (page, segment, region, context, entire) to be done.

Cache flushing

As the on-chip instruction and data caches and the external cache are physically addressed, there is no need to flush (write back) the contents of the cache to memory on a context switch between tasks, which

has to be done in systems which have virtually addressed caches. We need to invalidate the contents of the caches only once - on startup, to bring the system to a clean state.

DVMA and cache coherence

To ensure that both the processor and the DVMA device see the most recent copy of the information at an address, the hardware and the kernel work together to ensure this in one of two ways.

1) The page is marked as non-cacheable in both the SRMMU tables and the IOMMU table.

2) The page is marked as cacheable in both the SRMMU tables and the IOMMU table. Hardware will ensure that coherence is maintained, provided that the processor does not touch pages belonging to a stream data transfer during the transfer.

The microkernel follows the second model. The CCP transmit and receive buffers (allocated in DVMA space) are mapped as being cacheable in both the SRMMU tables and the IOMMU table.

6.12 Standalone Program Interface

Stand-alones programs are programs that are loaded into and execute from RAM. The OBP interface to the stand-alone programs consists of the specification of the machine environment that exists when the stand-alone program begins execution, and the set of services that OBP provides for the program's use.

Standalone Environment

Initial Program Counter

A stand-alone program is first loaded into memory at the address given by 'load-base'. If the program expects to execute from an address other than 'load-base', the program must begin with position-independent-code and copy itself to the expected address.

Stack

Stand-alone programs are invoked with a valid stack pointer, with at least 2K-bytes of memory available for stack growth. If the stand-alone requires more stack space, it should allocate a sufficient amount of stack space in its "bss" area, and setup the stack pointer appropriately.

Arguments

Arguments are passed to the stand-alone program using C subroutine argument passing conventions. The first argument is the base address of the ROMvec structure. The second and third arguments are the address and length of the array of bytes that is passed to the program from the program that booted it. For programs booted by OBP, the length is zero.

Memory and MMU

The stand-alone program may assume the presence of some amount of pre-mapped (one-to-one mapping) memory when it starts execution. On SuperSPARC, 10MB of memory is premapped by OBP. This is primarily intended for the use by the self-relocation code that moves the stand-alone to the location at which it is intended to execute.

The protection attributes of this pre-mapped memory is system dependent, but read, write and execute access from system state is guaranteed. Whether or not memory is marked cacheable is system dependent. Unused virtual addresses are marked invalid.

Interrupts

When a stand-alone program begins execution, interrupts are enabled but the processor priority level is set to its maximum value, denying all interrupts except non-maskable ones. When the program is ready, it should decrease the processor priority level to allow the user to interrupt the booting process from the keyboard.

The CPU's interrupt (trap) table will be valid, with OBP's clock interrupt, console input and output vectors and the breakpoint vectors appropriately installed. The stand-alone may setup it's own trap table, if required, but it should copy the clock, input, output and breakpoint vectors if it expects to use these services.

ROMvec Interface

The set of services provided to stand-alone programs by OBP is described by ROMvec. ROMvec is an array of data and subroutine addresses described by a C structure. ROMvec services include device tree access, console I/O, network I/O and other services.

6.13 Booting the microkernel

Diskless nodes running on the compute partition depend on the CCP network to load their OS. The network booting proceeds as follows: the diskless nodes use the "RARP/TFTP" protocols of the TCP/IP suite to load the image of the program that will run on the booting machine. Often, this is the secondary loader program, which then loads the OS, possibly using other protocols.

The microkernel loader is the secondary loader program used while booting the microkernel. It relocates itself to the address at which it is expected to execute, loads the image of the microkernel into memory using the TFTP protocol and then transfers control to the microkernel. The microkernel loader would have established one-to-one mapping for the microkernel.