Porting the Chorus *Supervisor* and Related Low-level Functions to the PA-RISC

Ravindranath Konuru,
Marion Hakanson,
Jon Inouye,
Jonathan Walpole*,

Department of Computer Science and Engineering
Oregon Graduate Institute of Science and Technology

January 27, 1992

Abstract

This document is part of a series of reports describing the design decisions made in porting the Chorus Operating System to the Hewlett-Packard 9000 Series 800 workstation.

The *Supervisor* is the name given by Chorus to a collection of low-level functions that are machine dependent and have to be implemented when Chorus is ported from one machine to another. The *Supervisor* is responsible for interrupt, trap and exception handling, managing low-level thread initialization, context switch, kernel initialization, managing simple devices (timer and console) and offering a low-level debugger [7]. This document describes the port of the *Supervisor* and related low-level functions.

The information contained in this paper will be of interest to people who wish to understand:

- The main characteristics of Chorus and PA-RISC architecture that are useful in understanding the port of the Chorus *Supervisor* .
- The requirements and implementation of the Chorus *Supervisor* .
- The requirements and implementation of Chorus page fault interface
- The requirements and implementation Chorus System Call Interface
- The requirements and implementation of *mutex* interface which is a part of the Chorus system call interface for efficient thread synchronization.
- Reasons for the modifications to the portable layers of Chorus kernel to implement the above requirements. A summary of the modifications is also presented.

It is useful to read the port overview [17] before reading this document. It is also a good idea to have the *Precision Architecture and Instruction Set Reference Manual* [10] and Chorus v3.3 implementation guide[7] on hand although it is not absolutely necessary.

*This research is supported by the Hewlett-Packard Company (HP), Chorus Systémes, and Oregon Advanced Computing Institute (OACIS).

Contents

1 Introduction

This document is part of a series of reports describing the design decisions made in porting the Chorus Operating System to the Hewlett-Packard 9000 Series 800 workstation.

Chorus is horizontally divided into a machine independent layer and a machine dependent layer. The machine dependent layer exports a machine *independent* interface that is expected to remain unchanged as the operating system is ported from one machine to another. The machine dependent layer is divided vertically into two major partitions: the *Supervisor* and the *mmu* (memory management unit). The *mmu* is responsible for the implementing the machine dependent memory management functions [1]. This document deals with the implementation of the *Supervisor* and other related low-level functions. The port of the *mmu* is discussed elsewhere [11].

An overview of the port of the Chorus *Supervisor* and related functions is given in section 1.1. Brief reviews of some of the characteristics of Chorus and the PA-RISC architecture are given in sections 1.2 and 1.3 respectively. The purpose of these reviews is to give sufficient background for discussing the machine dependent layer. For detailed information about Chorus, refer to the Chorus technical reports CS/TR-90-71 [7] and CS/TR-89-36.1 [2]. For information about PA-RISC, refer to [15, 10].

The *Supervisor* requirements and implementation are presented in section 2. The Chorus page fault interface is presented in section 3, the System call interface in section 4, and the *mutex* interface in section 5.

The main reasons for the modifications to the portable layers of the Chorus kernel and a summary of the modifications is given in section 6. Future work is presented in section 7.

1.1 Supervisor Port Overview

We started our ground work for the port in Sep 90. The operating system as well as the architecture were completely new to us at that time. We spent about a month reading the documentation and papers on PA-RISC architecture [10, 15, 16, 13] and Chorus operating system [2, 1, 4]. The **Tut** books [8, 3] documenting the mach 2.0 port by HP to PA-RISC proved valuable sources of information.

In Oct 90 we had a 1-week course on porting Chorus at Chorus Systémes, France. Various components were identified. As Chorus personnel were also not familiar with PA-RISC, the Chorus port to Motorola 88000 was used as a case study to explain the various machine dependent components and the porting process. This proved useful for the design of the Chorus *Supervisor* .

Assembly language programs were written to understand the PA-RISC architecture especially with respect to nullification, delayed branches, procedure calling conventions and the usage of adb, the assembly language debugger.

The following basic principles of design were applied as often as possible:

- Use 32 bit addresses. Initially, we considered using 64-bit addresses. However, it would have caused extensive changes in the portable layers of the kernel and is was not clear how to design an interface with 64-bit address parameters. In any case, it would have increased the time of the port. We left this for future work.

- Keep the design as simple as possible. The aim was to get the first working port as quickly as possible. This was one of the principles that was reiterated during our course at France. We whole heartedly agreed with that.

- Use the available **Tut** code for the machine dependent layer implementation. The goal again was to get the port up as quickly as possible. For example the code for initialization would have taken us a long time to figure out, write and debug if did not use the **Tut** code albeit with modifications.

The design and implementation of the *Supervisor* did not prove very difficult once we had a good grasp of the architecture and the Chorus machine dependent layer. The availability of **Tut** code was also very beneficial.

Chorus provides a kernel test suite[9] for validating the kernel. This was the only method we employed to validate our kernel port.

1.2 Chorus Overview

Chorus is a message based micro-kernel that supports the following abstractions[1]:

- Actor
- Thread
- Message
- Port

An **Actor** forms the unit of resource allocation and identifies a protected address space. An address space is split into a **user address space** and **system address space**. On a given site[2], each actor's system address space is identical and its access is restricted to privileged levels of execution. An actor in Chorus can be a Supervisor actor or User Actor. A supervisor actor lives in the system address space along with the kernel. Supervisor actors have higher privilege than user Actors.

A **thread** is the basic unit of execution and runs in the context of an **Actor**. A thread is a sequential flow of control and is characterized by a thread context corresponding to the state of the processor at any given point during the execution of thread. There can be multiple threads per **actor**.

Threads communicate and synchronize by exchanging **messages** between their actors' **ports**. Threads sharing the same address space can use share memory for communication and synchronization. **Semaphores** and **Mutexes** provided by the Chorus interface are useful for this purpose.

A thread belonging to a user Actor is called a user thread. However during a system call, it becomes a supervisor thread. A user thread has 2 stacks: a user stack for executing user code and a system stack for executing system calls, traps, and storing the context of the thread when the thread is blocked. A thread belonging to a supervisor actor is called a supervisor thread. Since a supervisor thread lives entirely in the system address space, it has only a system stack and no user stack.

1.3 PA-RISC

This section consists of extracts from the PA-RISC architecture reference manual useful for understanding the machine dependent layer implementation. For more details see the cited references.

[1] Chorus is written in an object oriented language C++. These abstractions are implemented as C++ classes

[2] A site is a grouping of tightly-coupled resources controlled by a single Chorus Nucleus[6]

PA-RISC Architecture is the frame work for Hewlett-Packard's HP3000/900, HP9000/800, and HP9000/700 series computer systems.

It is based on the principles of RISC and has 140 fixed length instructions. It employs a virtually addressed cache and the I/O sub-system is memory mapped. PA-RISC supports 48-bit, 56-bit or 64-bit virtual addresses and provides some hardware protection support. The global virtual memory is organised as a set of linear spaces with each space being 4 gigabytes (2^{32}) long. Each space is specified with a space identifier.

PA-RISC supports 4 privilage levels numbered 0-3. The highest privilege level is 0 and the lowest privilege level is 3.

PA-RISC architecture has the following resources:

- 32 General Registers. GR0 is tied permanently to zero. GR1 is the target of *Addil* instructions. GR31 is the link register for an inter-space branch and link external (*Ble*) instruction. GR27 used as the base pointer for data accesses. This is specified by the procedure calling conventions of the architecture.

- 25 Control Registers. CR1-CR7 do not exist. Control registers are discussed in more detail in the section 1.3.1.

- 8 Space Registers. SR0 is the instruction address space link Register for Ble instruction. SR0-SR4 can be modified at any privilege level. SR5-SR7 can be modified at privilege level 0. The usage of the space registers is left to the operating system. The space registers are 16-bit long on a level 1 PA-RISC, 24-bit long on a level 1.5 PA-RISC and 32-bit long on a level 2 PA-RISC. On a level 0 PA-RISC, the space registers do not exist. A level 0 PA-RISC supports absolute addressing only.

- Processor Status Word (PSW) The processor state is encoded in a 32-bit register PSW. PSW does not appear as an operand in instructions. When an *interruption* [3] occurs, the old value of the PSW is saved in the IPSW register(CR22). Some of the bits in the PSW are reserved. It is software's responsibility that these are zero when written. The PSW is set from IPSW by a *return from* interruption instruction.

  The PSW bits that are important for the discussion are:

  - C bit (PSW_C) Code (instruction) address translation enable. When 1, instruction addresses are translated and access rights checked.
  - Q bit (PSW_Q) Interruption Collection Enable. When 1, *interruption* state is collected. When an *interruption* occurs the details of the instruction being executed are recorded in the control registers (see 1.3.1).
  - P bit (PSW_P) Protection Identifier enable. When this bit and the C-bit are both 1, instruction references check for valid protection identifiers(PIDs). When this bit and the D-bit are both 1, data references check for valid PIDs. When this bit is 1, probe instructions check for valid PIDs.
  - D bit (PSW_D) Data address translation enable. When 1, data addresses are translated and access rights checked.
  - I bit (PSW_I) External interrupt, power failure interrupt, and low-priority machine check interrupt unmask. When 1, these interrupts are unmasked and can cause an interrupt. when 0 the interrupts are held pending.

[3] An *interruption* is PA-RISC specific term. An *interruption* is a trap or an interrupt that can occur on PA-RISC.

- Instruction Address queues.

  The instruction Address queues hold the address of the currently executing instruction and the address of the instruction that will be executed after the current instruction, termed the *following* instruction. There are 2 queues: Instruction Address Space Queue (IASQ) and the Instruction Address Offset Queue(IAOQ). Each queue is 2 elements deep. The elements are referred to IAOQ_FRONT, IAOQ_BACK, IASQ_FRONT and IASQ_BACK. The 2-deep queues are used to support the delayed branching capability.

1.3.1 Control Registers

This section defines the main registers used in the implementation:

- Protection Identifier Registers: PID1, PID2, PID3, PID4, aliases for CRs 8, 9, 12, and 13. These registers designate up to four groups of pages accessible to the currently executing process. When translation is enabled, the four protection identifiers (PIDs) are compared with a page access identifier to validate access. If access is not valid trap is raised.

- Coprocessor Configuration Register (CR10 alias CCR) is an 8-bit register which records the presence and usability of coprocesors. A bit is 1 implies the coprocessor corresponding to that bit is present and operational. Else it is logically decoupled. In the current implementation the entire CCR is set to 0.

- *interruption* Vector Address Register (CR14 alias IVA) contains the **absolute address** of the base of an array of service procedures assigned to the interruption classes. This address must be a multiple of 1024.

- External Interrupt Enable Mask (CR15 alias EIEM) is a 32-bit register containing a bit for each of the 32 external interrupts. When 0, bits in the EIEM mask interrupts pending for the external interrupts corresponding to those bit positions.

- External Interrupt Request Register (CR23 alias EIRR) is a 32-bit register containing a bit for each external interrupt. When 1, a bit designates that an interrupt is pending for the corresponding external interrupt. Both the PSW_I bit and the corresponding bit position in the EIEM must be 1 for an interrupt to occur.

- Interval Timer Register (CR16 alias ITMR) consists of 2 internal registers. One of the internal registers is continually counting up by 1. Reading the ITMR gives the value of this internal register. Writing to ITMR updates the other (comparison) register. When the two registers have identical values, an external interrupt is raised and bit 0 of EIRR is set to 1.

- *interruption* Instruction Address Space and Offset Queues (CR17 alias IIASQ, CR18 alias IIAOQ): Two offset registers and two space registers are used to save the instruction address and and privilege level information for use in processing interruptions. The registers are arranged as two two-element deep queues. The queues generally contain the address(including the privilege level field in the rightmost two bits of the offset part) of the two instructions in the IA queues at the time of the interruption.

  The IIA queues are continually updated whenever the PSW_Q bit is 1 and are frozen by an interruption (PSW_Q) bit becomes 0. After such an interruption these registers contain copies of the IA queues. These queue elements will also be referred to as PCOQH, PCOQT, PCSQH and PCSQT in the context of the implementation.

- Interruption parameter registers are the Interruption Instruction Register (CR19 alias IIR), Interruption Space Register (CR20 alias ISR) and Interruption Offset Register (CR21 alias IOR). As the names indicate, these registers contain interrupted instruction and the virtual address the instruction was attempting to access.

1.3.2 Interruptions

Table 1: PA-RISC Interruption

Interruption #	Description
1	High-priority machine check
2	Power failure interrupt
3	Recovery counter trap
4	External interrupt
5	Low-priority machine check
6	Instruction TLB miss fault
7	Instruction memory protection trap
8	Illegal instruction trap
9	Break instruction trap
10	Privileged operation trap
11	Privileged register trap
12	Overflow trap
13	Conditional trap
14	Assist exception trap
15	Data TLB miss fault
16	Non-access instruction TLB miss fault
17	Non-access data TLB miss fault
18	Data memory protection trap/Unaligned data reference trap
19	Data memory break trap
20	TLB dirty bit trap
21	Page reference trap
22	Assist emulation trap
23	Higher-privilege transfer trap
24	Lower-privilege transfer trap
25	Taken Branch trap

All *interruption*s (traps or interrupts) on PA-RISC are precise, i.e., the software sees a single unpipelined processor executing one instruction at a time. PA-RISC supports 25 interruptions divided into 4 priority groups, with group 1 having the highest priority and group 4 the lowest. The interruptions are listed in table 1.

Interruption 1 belongs to group 1. Interruptions 2-5 belong to group 2. Interruptions 6-22 belong to group 3 and the rest to group 4.

1.3.3 Memory Management Support

Like most microprocessor architectures, the PA-RISC contains some form of memory management unit (MMU).

This section describes the features of the PA-RISC that are used to support virtual memory operations. These features include a translation look-aside buffer (TLB) for transforming virtual addresses to physical addresses, bit traps for memory management support, and memory protection mechanisms. The material presented in this section is covered in more detail in chapter 3 of the *Precision Architecture and Instruction Set Reference Manual* [10].

Page Tables and the TLB :

The PA-RISC (along with the MIPS R2000/R3000) is unusual in that it requires software to handle TLB misses[4]. By allowing software to perform TLB loads, the PA-RISC architecture gives the operating system lots of flexibility in the format of page tables. Normally, architectures specify some page table format to follow so the hardware can perform TLB loads.

Rather than develop our own page table design for the initial port, we decide to use the *Physical Page Directory (PDIR)* format suggested by the PA-RISC architecture manual [10]. We made this decision because it allowed us to reuse a great deal of **Tut** code for the low level TLB miss handlers. Figure 1 shows the structure of a Physical Page Directory (PDIR) entry.

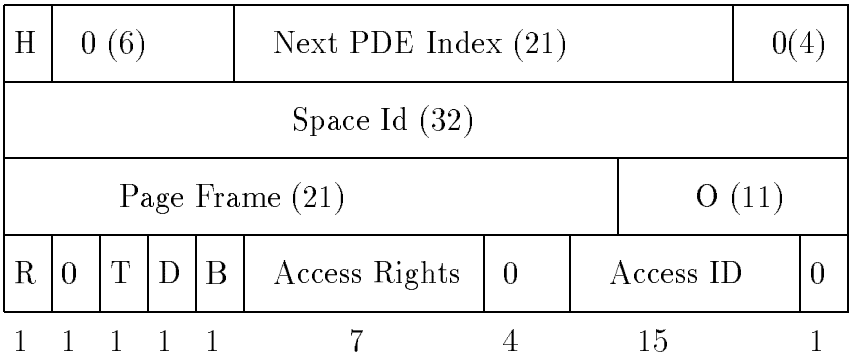

Figure 1: PDIR Entry (PDE)

Bit Flags :

The TLB and PDIR contain a variety of bit flags which can be used to generate traps. The following information describes the function of each of the 1-bit fields.

T Page Reference Trap. When 1, data references using this translation cause a page reference trap interruption. The T-bit is most commonly used for program debugging.

D Dirty. When 0, store and semaphore instructions cause a TLB dirty bit trap on systems with software TLB miss handling. When 0, store and semaphore instructions cause the D-bit in

[4]The *PA-RISC Architecture and Instruction Set Manual* mentions that hardware implementations can exist but to our knowledge no such implementation exists at this time.

the DTLB entry and the PDIR to be set to 1 on systems with hardware TLB miss handling. When 1, no trap or update occurs. The D-bit may be used by the operating system to determine which pages have been modified.

B Break. When 1, instructions that could modify data using this translation cause a data memory break trap interruption, if enabled. Store instructions, the PURGE DATA CACHE instruction, and semaphore instructions are the only instructions that potentially modify data. The B-bit is most commonly used for program debugging.

R is the reference bit (only present in the PDIR entry). If R = 1, the page has been accessed (read, write, execute, or non-access) by a processor since the bit was last set to 0. For systems with software TLB miss handling, this bit is managed by the software and not directly set by the hardware.[5]

Memory Protection :

The TLB is also responsible for enforcing memory protection. The PA-RISC protection mechanisms are disabled when physical addressing is used or when the PSW_P bit is disabled. The TLB maintains protection information in two fields: the *access rights* and the *access ID*. The 7-bit access right field encodes the allowed access types and privilege levels into three sub-fields: *type*, privilege level 1 (*PL1*), and privilege level 2 (*PL2*). The *access ID* is a 15-bit field that can be thought of as a capability. This field must match one of the four protection ID's in the PA-RISC's control registers (CR8,9,12,13).

Logical Page Replacement :

The PA-RISC allows the software to operate on a logical page size of 2K, 4K, 8K, or 16K bytes. When operating on a logical page size greater than 2K bytes, the TLB miss handling procedures may insert all translations for that page group provided that the translation for the faulting page is inserted last. This is probably because the software has no ability to know which TLB entry is invalidated to make room for a new insertion. By inserting the faulting page entry last, the software ensures that upon return, the TLB miss has been satisfied.

PA-RISC Memory Management Traps :

Out of the 25 *interruption*s that can occur on PA-RISC, 9 of the interruptions are traps to be dealt by the memory management unit of the operating system. These memory management traps are listed in table 2.

These traps can be partitioned into four groups: TLB miss faults, non-access TLB miss faults, meory protection faults, and bit flag traps.

TLB Miss Faults (#6,#15) :

The PA-RISC architecture allows both software and hardware TLB miss handling. The HP 9000/834, the target processor for the port, does not have hardware TLB miss handling. It has separate traps for instruction and data TLB misses with the hardware making no distinction between TLB misses and page faults. When a TLB miss fault occurs, the handler must determine

[5]The unused bit is used by some implementations. This **A** bit acts similarly to the R bit except non-access faults will not set it.

Table 2: PA-RISC Memory Management Exceptions

Trap #	Description
6	Instruction TLB miss fault
7	Instruction memory protection trap
15	Data TLB miss fault
16	Non-access instruction TLB miss fault
17	Non-access data TLB miss fault
18	Data memory protection trap/Unaligned data reference trap
19	Data memory break trap
20	TLB dirty bit trap
21	Page reference trap

whether or not the missing page is in memory. One disadvantage of an inverted page table(i.e, PDIR) is that it is more expensive to determine whether a particular virtual page is in memory. We use a hashing function and linked list search to determine whether a virtual page entry is present in the PDIR. The handler hashes the faulting (virtual) address to obtain an offset into a hash table. This hash table contains a reference to the PDIR list that represents the hash bin. This bin is organized as a linked list of PDIR entries. The handler then sequentially searches this list for the desired virtual page. A successful match results in the entry being placed in the TLB. A failure in the matching process results in a page fault. Figure 2 presents a flow chart of the steps for handling a TLB miss.

Non-access TLB miss faults (#16,#17) :

The PA-RISC architecture also has the notion of non-access TLB faults which differ from other TLB faults in that the faulting page need not be loaded into memory. Our platform requires both instruction and data non-access TLB miss faults to be handled by software.

Non-access data TLB miss faults are caused by LOAD PHYSICAL ADDRESS (LPA), PROBE, and PURGE/FLUSH DATA CACHE instructions. When the requested page entry is not present in the PDIR, the action of the trap handler depends on the type of instruction causing the fault. For LPA and PROBES, zero is returned if the desired page cannot be found in the PDIR. There is a problem with the PROBE instruction that is covered in more detail in section 3.2.

In HP-UX and **Tut** , cache PURGE and FLUSH instructions that cause non-access TLB miss faults are handled as if a TLB miss occurred, i.e. the page is loaded into physical memory and the page descriptor is inserted into both the PDIR and TLB. Non-access instruction TLB miss faults are caused by FLUSH INSTRUCTION CACHE (FIC) instructions. These are handled similar to other cache non-access faults described above.

Memory Protection Traps (#7,#18) :

The PA-RISC has two traps used to detect memory protection violations. The *instruction memory protection trap (7)* is the result of invalid access rights or invalid protection IDs for an instruction fetch[6]. The *data memory protection trap (18)* is the result of an invalid access right or protection ID for any load, store, semaphore, and PURGE DATA CACHE instruction. This

[6]Protection ID checking is only done when the PSW P-bit is set.

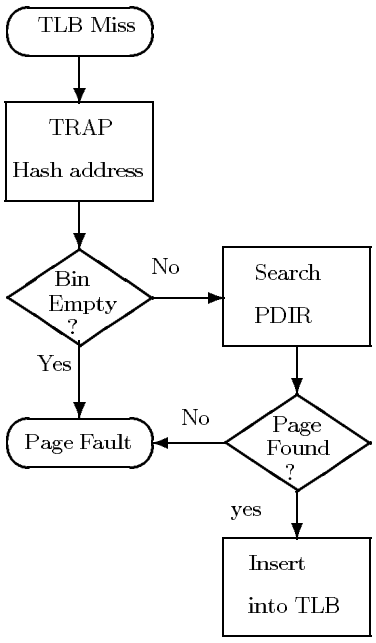

Figure 2: TLB Miss Handling

trap is also caused by any load or store to addresses not aligned at the boundaries required by the instructions. Detection of unaligned addresses is performed by examining the least significant bits of the virtual address.

TLB Dirty, Page reference and Data memory break traps (#20,#21,#19) :

The HP 9000 Series 834 workstation does not have a hardware supported TLB, so the manipulation of the D (dirty) and R (reference) bit flags is left to the operating system.

When the D bit is 0, stores and semaphore operations will cause a *TLB dirty bit trap (20).* The trap handler must then set the D bit in both the PDIR and TLB entry. Once the D bit has been set, further modifications to that page are ignored. If the T bit is set, data reference using the translation causes a *page reference trap (21).* The *data memory break trap (19)* is triggered when instructions that could possibly modify data require the translation and the B bit in the Processor Status Word (PSW) is 1. When software loads an entry into the TLB, it should set the R bit to indicate that the page has been referenced.

2 Supervisor

The *Supervisor* is the component that directly interacts with the underlying hardware. It is responsible for managing interrupts, traps and exceptions and other machine dependent functions. The *Supervisor* along with the *mmu* layer forms the machine dependent layer and is expected to offer a machine independent interface to the portable layers of the Chorus kernel. The requirements to be satisfied by the *Supervisor* layer are detailed in section 2.1 and the implementation is detailed in section 2.2.

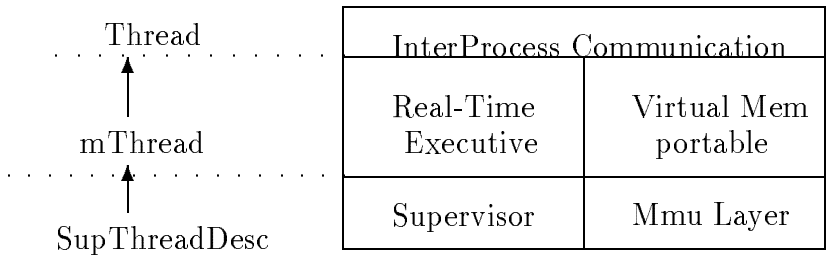

Figure 3: thread Class hierarchy

2.1 *Supervisor* requirements

The Chorus *Supervisor* is expected to export a specified machine independent interface, and is responsible for interrupt, trap and exception handling, timer and console management, kernel initialization, and offering a low-level debugger. The *Supervisor* interface is detailed in section 2.1.1. The sections on event handling, timer and console management and low-level debugger regroup the functions in the interface according to their functionality and provide the requirements for the function group as a whole.

In addition to the above functions, the *Supervisor* is responsible for defining two fundamental structure types : `KnThreadCtx`, and `SupThreadDesc`. `KnThreadCtx` defines the register context frame that is used to save state during interrupts, traps, exceptions and context switches. `SupThreadDesc` defines the machine dependent thread descriptor. As mentioned in section 1.2, a thread in Chorus has a user stack and a system stack. The descriptor `SupThreadDesc` keeps track of the stacks and other machine dependent thread attributes (if any) of the thread and is the base class for the `Thread` class. The `Thread` class hierarchy is shown in fig 3. The dotted lines show the levels of definition and management of base and derived classes. Variables and pointers of type `KnThreadCtx` and `SupThreadDesc` get defined and passed in the portable layers of the kernel but are treated as black boxes. Functions are defined in the *Supervisor* interface (see section 2.1.1) that allow the portable layers to query and update the contents of the data structures in a machine independent manner.

A portion of the Chorus Interface to supervisor actors allows handlers to be attached for interrupts, traps, exceptions and time-outs. Invocation of some of these handlers is the responsibility of the *Supervisor* . This requirement is detailed in section 2.1.2.

2.1.1 *Supervisor* interface requirements

The following functions must be implemented by the *Supervisor* .

SupCtxInit(): Build the initial context frame on the system stack of the new thread and initialize it's machine dependent thread descriptor `SupThreadDesc`. The initial values inserted into the context frame on the system stack are used by **SupCtxSwitch()** when switching to the new thread. **SupCtxInit()** should build the frame as if the thread is returning from an exception. This function takes the following parameters:

- The system stack bottom, `unsigned char *stackbot`.
- Thread parameters descriptor, `KnThreadDesc *threadParams`. This descriptor has the entry point of the thread, the thread privilege, priority, the user stack bottom, and the initial execution status. The user stack bottom is used only when the thread is a user thread. In the case of a supervisor thread, this field is ignored.

- Pointer to the thread's machine dependent thread descriptor, `SupThreadDesc *ptThreadDesc`.
- Pointer to the virtual address space descriptor of the actor in which the thread will be created, `context *ptContext`. Note that `context` is a class used by the virtual memory system and is not the same as the machine dependent thread context which is basically a set of registers.

SupCtxSwitch(): Switch thread machine dependent context. This function takes the following parameters:

- Pointer to the old thread, `SupThreadDesc* oldThread`
- Pointer to the new thread, `SupThreadDesc* newThread`

SupGetUserCtx(): Return a Pointer to a thread's saved context, `KnThreadCtx* SupGetUserCtx(...)`. This function takes the following parameter:

- Pointer to the machine dependent thread context descriptor, `SupThreadDesc* desc`.

SupCtxReset(): Reset thread's context frame on the stack by the values given in the machine dependent thread context descriptor. This function takes the following parameters:

- Pointer to thread machine dependent context, `SupThreadDesc* desc`
- Pointer to exception context frame on the stack, `KnThreadCtx* ctx`

SupCtxIsUserMod(): Return *true* if thread execution is in User mode else *false*. This function takes the following parameters:

- Pointer to a context frame, `KnThreadCtx* ctx`.

SupCallConnect(): Connect a vector of handlers to a trap. This function takes the following parameters:

- The trap number, `unsigned trapNb`
- Pointer to the vector of handlers, `KnCallEntry* hdlVect`
- Number of elements in the vector, `unsigned NoHdl`
- The privilege level `unsigned sup`. Basically there are two privilege levels: **Supervisor** and **User**. If `sup` is **Supervisor** in this call then this vector is executed for supervisor actors causing a trap equal to trapNb. If a user Actor causes a trap equal to trapNb, this vector will not be executed unless another SupCallConnect has been explicitly called with the same parameters and `sup` is set to **User**.

SupCallDisConnect(): Disconnect a Vector of trap handlers. This function takes the following parameters:

- The trap number, `unsigned trapNb`
- The privilege level, `unsigned sup`

SupItConnect(): Connect a handler to an interrupt. This function takes the following parameters:

- The interrupt number, `unsigned intrNb`

- The handler to be executed on the interrupt occurrence, `KnHdl hdl`.
- The privilege level, `unsigned sup`

SupItDisconnect(): Disconnect a Interrupt handler. This function takes the following parameters:

- The interrupt number, `unsigned intrNb`
- The handler to be executed on the interrupt occurrence, `KnHdl hdl`.

The handler parameter is required since there can be a list of interrupt handlers connected to the interrupt. The (intrNb, hdl) pair uniquely identifies the element to be removed from the list.

SupItLevel(): Return the current interrupt nesting level. This function takes no parameters.

svMask(): Set the interrupt level. All interrupts equal or less than this level are masked. Returns previous interrupt level. This function takes the following parameters:

- Interrupt level mask, `int intLvlMask`.

svUnMask(): Reset the interrupt level. All interrupts equal or less than this level are unmasked. Returns previous interrupt level. This function takes the following parameters:

- Interrupt level unmask, `int intLvlUnMask`.

svMaskAll(): Mask all interrupts. This function has no parameters.

svUnMaskAll(): Unmask all interrupts. This function has no parameters.

svCopyIn(): copy from User space into kernel space. This function takes the following parameters:

- Source address in user space, `char* src`
- Destination address in kernel space, `char* dst`
- Size of transfer in bytes, `unsigned int count`.

svCopyOut(): copy from Kernel space to User space. This function takes exactly the same parameters as svCopyIn(), only that the source and destination spaces are reversed.

The functions **sv*()** are also part of the Chorus Supervisor actor interface.

SupTrapConnect(): Connect a handler to a trap. This function takes the following parameters:

- The trap number, `unsigned trapNb`
- The handler to be executed on the trap occurrence, `KnHdl hdl`.

SupTrapDisConnect(): Disconnect a Trap handler. This function takes the following parameters:

- The trap number, `unsigned trapNb`
- The handler to be executed on the trap occurrence, `KnHdl hdl`.

SupPanic(): Fatal abort. This function takes no parameters.

SupDebugger(): Call the debugger. This function takes the following parameters:

- The exception context frame pointer, `KnThreadCtx* ctx`
- The trap or exception number, `unsigned no`

SupPreciseTime(): Return the current precise time. This function takes no parameters.

SupPutChar(): Write a character on the console device. This is a synchronous operation, i.e., the write returns only after the output is completed. This function takes the following parameters:

- the character to be written, `int c`

SupGetChar(): Returns a character from the input device. This is a synchronous operation. This function takes no parameters.

SupPollChar(): Poll the input device. This function returns 0 if no input is waiting else it returns the character. This function takes no parameters.

In addition to exporting the interface, the *Supervisor* is expected to make up-calls into the kernel upper layers for various synchronous and asynchronous events. The calls are:

KnDebugEnter(): The *Supervisor* is expected to call this function to inform the portable layers whenever it enters the debugger. This functions informs the portable layers not to perform context switching when the debugger is entered. This function takes no parameters.

KnDebugLeave(): The *Supervisor* is expected to call this function to inform the portable layers whenever it leaves the debugger. This function takes no parameters.

KnLock(): Lock the kernel. This function takes no parameters.

KnUnLock(): Lock the kernel. This function takes no parameters.

KnHandler(): Exception Handler of the kernel. This function should be called for all unrecoverable exceptions. This function executes the actor specific exception handler if present else calls KnIpcHandler() to abort the thread. This function takes the following parameters:

- Pointer to the exception frame on the stack, `KnThreadCtx* ctx`
- Exception number `int excNb`

KnItRetSup(): Return from interrupt to supervisor thread. The supervisor after executing the interrupt handlers connected by SupItConnect() prepares to return from the interrupt. This function should be called by the *Supervisor* if the thread executing at the time of the interrupt was a supervisor thread. This function takes no parameters. A supervisor thread can be preempted only if there is a supervisor thread of higher priority ready to run.

KnItRetUser(): Return from interrupt to supervisor thread. The supervisor after executing the interrupt handlers connected by SupItConnect() prepares to return from the interrupt. This function should be called by the *Supervisor* if the thread executing at the time of the interrupt was a user thread. This function takes no parameters. This function can cause preemption of the user thread.

KnAbortHandler(): Abort Handler. If the thread is found to be aborted while returning from an interrupt, then KnAbortHandler() is called. This function takes the following parameters:

- The exception frame on the stack, `KnThreadCtx* ctx` as parameter.

KnTimeIn(): Record a clock tick. This function should be called by the *Supervisor* every time a clock interrupt occurs. This routine increments the Chorus software clock and executes any routines that have reached their timeout period. This function takes the following parameters:

- The execution mode at the time of the clock interrupt, `int supOrUsr`.
- The program counter at the time of the clock interrupt, `int pc`.

2.1.2 Supervisor Actor Interface

A portion of the Chorus interface is available only to supervisor actors and would be referred to as the *supervisor actor interface*. Some of the functions of the supervisor actor interface get directly mapped to corresponding functions of the supervisor interface and the rest of the functions are handled in the portable layers of the Chorus kernel. Ideally, all the calls of the supervisor actor interface except **svCheckUserSpace()**, **svCopy[In/Out]()**, **sv[Un]Mask[All]()** are expected to be implemented in the portable layers of the Chorus kernel by calling the appropriate functions in the *Supervisor* interface. However, due to the way in which instructions are generated on the PA-RISC by the compiler, additional work and portable layer modifications were required to implement this functionality (See section 2.2.9 for details and functionality implementation).

Only those functions of the supervisor actor interface that needed additional implementation are specified below. Note that **svCopy[In/Out]()**, **sv[Un]Mask[All]()** have already been covered under the *Supervisor* interface.

svAbortHandler(): Define an abort handler for the Actor. This function takes the following parameters:

- Actor Capability, `KnCap *actcap`.
- Abort Handler, `KnHdl routine`.

This function is expected to be entirely implemented in the portable layers of the kernel.

svCallConnect(): Exactly the same function and parameters as **SupCallConnect()** (see section 2.1.1). This function is expected to be entirely implemented in the portable layers of the kernel.

svCheckUserSpace(): verify that an address is within the user address space. This function takes the following parameters:

- the address to be checked, `char* addr`.

This function is expected to be implemented during the port to the target architecture.

svExcHandler(): define an exception handler for the Actor. This function takes the following parameters:

- Actor Capability, `KnCap *actcap`.
- Exception Handler, `KnHdl routine`.

This function is expected to be entirely implemented in the portable layers of the kernel.

svItConnect(): Exactly the same function and parameters as **SupItConnect()** (see section 2.1.1). This function is expected to be entirely implemented in the portable layers of the kernel.

svTrapConnect(): Exactly the same function and parameters as **SupTrapConnect()** (see section 2.1.1). This function is expected to be entirely implemented in the portable layers of the kernel.

svTimeOut(): set a time out and call the given routine when the time-out occurs. This function takes the following parameters:

- The routine to be called by kernel on time out, `KnToHdl routine`.
- The parameter to be passed to `routine`, `void param`
- TimeOut period in milliseconds, `unsigned int delay`.

This function is expected to be entirely implemented in the portable layers of the kernel.

2.1.3 Event (Interrupt, Trap and Exception) Handling

The *Supervisor* is expected to save the register context on the stack, call the appropriate handlers and restore register context when required. The functions in the *Supervisor* interface that fall in this group are **SupTrap[Dis]Connect()**, **SupIt[Dis]Connect()**, **SupCall[Dis]Connect()**, **SupItLevel()**, and **sv[Un]Mask[All]()**. The *Supervisor* implements the data structures and code for these functions and calls the appropriate connected handlers. In the case of interrupts, the *Supervisor* should execute the list of handlers in the decreasing order of priority and acknowledges the interrupt to the external device raising the interrupt. In all cases, up-calls should be made at the precise points in execution as identified by the supervisor interface. The general algorithms to be used for interrupt, trap, and exception handling are detailed in the Chorus implementation guide [7].

2.1.4 Timer and Console Management

The *Supervisor* manages the timer and console devices. It programs the timer device so that it generates clock ticks at a frequency defined by the K_CHZ constant defined in **include/chorusConf.h**. Each time a timer interrupt is received, the supervisor calls the **KnTimeIn()** function (see section 2.1.1). The functions of the *Supervisor* interface that fall under this group are **SupPutChar()**, **SupGetChar()**, **SupPollChar()**.

The *Supervisor* is responsible for connecting, at least, **SupPutChar()** and **SupGetChar()** behind a trap. This trap is used in the implementation of library functions **PutChar()** and **GetChar()**.

2.1.5 Low-level Debugging facility

The *Supervisor* is responsible for implementing the kernel debugger. The function that implements the debugger is **SupDebugger()** (see section 2.1.1).

The *Supervisor* is responsible for connecting the debugger entry point to a trap number. This trap number will be used by the implementation of the **callDebug()** library function. The **callDebug()** function is part of the Chorus kernel interface exported to Chorus Actors.

The *Supervisor* should call **KnDebugEnter()** and **KnDebugLeave()** when entering or leaving the debugger. This avoids context switches when in the debugger.

2.1.6 Kernel initialization

The *Supervisor* implements the function (usually called **start()**) that performs the kernel initialization. This function performs all the machine dependent and machine independent initialization necessary for calling the portable layers of the kernel. The function **start()** forms the entry point of the Chorus kernel image. Transfer of control to this entry point is performed by the boot program portion of the **boot archive** loaded by the resident boot monitor. For more details on the **boot archive** and Chorus booting procedures see the PA-Chorus booting document[12].

The kernel initialization function is responsible for:

- Initialization of processor specific data like interrupt vector, setting the process status word for appropriate execution mode, etc.
- Static constructors' invocation. Chorus is written in C++, an object oriented language and the static constructors for the various static objects of the kernel must be called.
- Initialization of memory management, by calling **VmInit()**.
- Initialization of various devices and connection of device handlers and trap handlers. This function is embedded in the routine **SupBoardInit()**.
- Calling **KnInit()**, a function that initializes the portable part of the kernel. This includes scheduler data structure initialization, connection of system call handlers, and creation of the first thread of the system. This first thread is the transformation of the kernel initialization code being executed into a Chorus abstraction. **KnInit()** returns the new stack pointer to be used by the executing first thread.
- Switching to the new stack pointer and call **knMain()** which is the main routine of the kernel. **knMain()** never returns.

2.2 *Supervisor* Implementation

The fundamental data structures `KnThreadCtx` and `SupThreadDesc` manipulated by the *Supervisor* code are defined first in sections 2.2.1 and 2.2.2 respectively. This will establish the background to detail the implementation of the *Supervisor* in the rest of the sub-sections.

2.2.1 Thread Register Context

The thread register context is basically is the set of general registers and control registers of the processor and any other information that is needed for monitoring, manipulating and resuming the thread at a later stage. The thread register context is required to be typedefined as `KnThreadCtx` and is declared for PA-RISC in **include/PARISC/threadCtx.h**. The following are the elements of the `KnThreadCtx` structure:

- `state_flags`, a software register used to track current status of the thread, ex: in-system-call, in-trap, etc.

- General registers `gr1,..,gr31`. PA-RISC has only 31 32-bit general registers. Gr0 is permanently tied to 0.
- Control registers `cr0,cr8,..,cr31`. Cr1-cr7 do not exist.
- Instruction space queue tail `pcsqe`(alias PCSQT), instruction offset queue tail `pcoqe`(alias PCOQT). These fields contain the address (space and offset) of the next instruction to be executed.
- Kernel stack pointer `ksp`, this field is a software register. It is 0 when running on the kernel stack and contains the stack pointer to the kernel stack when running on the user stack in user mode.
- Space registers `sr0..sr7`.
- Floating point registers `fr0..fr15`.
- Special functional unit status registers, `mdhi, mdlo, mdov`, keep track of the status of the special functional units, emulated or actual hardware.

The floating point registers and special functional unit fields are ignored in the current implementation. This implies that code having floating point instructions or special function instructions will currently abort. The next version of the implementation will have floating point and special function unit emulation.

Discussion :

The definition of the thread context follows from our design objective of reusing as much of the **Tut** code as possible. The **Tut** project was done in two phases. First the HP-UX virtual memory system was replaced by Mach virtual memory system. In the second phase, HP-UX was modified to provide the mach thread abstraction and interface. In the case of the **Tut** kernel with threads, there are 3 different structures used to store the thread context depending on the execution mode of the thread and the purpose of accessing the context.

The purpose of each of the structures of the **Tut** kernel is given below:

- `save_state` structure is used when the thread enters the kernel mode through system calls, traps and interrupts.
- `PCB` structure is used when the thread was executing in kernel mode.
- `hppa_thread_state` structure is the context visible to the user for interrogation and modification.

In the case of Chorus, the machine independent layers recognize only one structure for the thread context, i.e., `KnThreadCtx`. For the PA-Chorus port, we defined `KnThreadCtx` structure as the union of the three structures. This enabled us to use the same structure uniformly through out the kernel and allowed us to use the low-level **Tut** code for the system-call interface, interrupt and trap handling as our starting point and make the Chorus specific modifications relatively easily. Further, we saw no reason to have distinct structures as a single structure can be used to store different levels and types of information.

```
typedef struct {
    unsigned    reserved;   /* reserved  for simple links */
    long        typeCtx;    /* Supervisor or User Thread */
    KnThreadCtx *currCtx;   /* indirect pointer to saved context of
                             * thread */
    KnThreadCtx *userCtx;   /* indirect pointer to initial context
                             * of the thread */
} SupThreadDesc;
```

Figure 4: Machine dependent thread descriptor: `SupThreadDesc`

2.2.2 Machine Dependent Thread Descriptor

The machine dependent thread descriptor is typedefined as `SupThreadDesc`. As described in section 2.1, this descriptor is used to keep track of the thread's system stack, user stack and the thread's context. In the case of PA-RISC, `SupThreadDesc` is defined in **include/PARISC/sv.h** as in fig 4.

The fields of `SupThreadDesc`, except the link field, get initialized in **SupCtxInit()** (see section 2.2.3), and remain fixed during the life time of the thread.

2.2.3 SupCtxInit()

This function is implemented in **kern/PARISC/sv.cxx**. The initialization of the new thread is performed in the following manner:

1. If the thread is the first thread of the kernel then exit from the function. The first thread of the kernel is nothing but the kernel initialization code being made part of the thread abstraction and recognizable by the Chorus portable layers. This thread ultimately becomes the idle thread of the system. Since this "thread" was already executing before it was created, there is nothing to be done at this stage. The machine dependent initialization for the first thread would have been already performed in **start()** in **kern/PARISC/sv.cxx** during kernel initialization.

2. Force the allocation of the system stack of the thread. At the point of calling SupCtxInit(), the system stack of the new thread is mapped, but physical memory is not allocated, by the virtual memory layers. It is necessary for the system stack to be actually allocated in physical memory before starting up the new thread since traps caused by the new thread must be handled on its system stack and this would cause recursive traps if the system stack is not physically allocated.

3. Allocate two frames of type `KnThreadCtx*`: `userFrame` and `switchFrame` on the system stack (see fig 5).

4. Initialize the `userFrame` as follows:

   (a) If the thread is a user thread then initialize the stack pointer as follows:

   userFrame→sp = threadParams→sp + FM_SIZE.

   FM_SIZE is the frame size needed to satisfy the PA-RISC procedure calling conventions.

(b) If the thread is a supervisor thread then initialize the stack pointer as follows:

`fr_size` = sizeof(`KnThreadCtx`) + FM_SIZE + FM_FIXED_ARG_SIZE.
userFrame→sp = stackbot + 2 * `fr_size`.

The FM* operands above are needed to satisfy the PA-RISC procedure calling conventions.

(c) Initialize the thread's Processor Status Word. As mentioned earlier, the thread frame should be initialized as if the thread is returning from exception. So the required *interruption* parameter registers are updated as follow:

`userFrame`→`ipsw` = Q + C + D + I,
`userFrame`→`eiem` = *Enable-all-interrupts*

(d) Initialize the thread's protection identity registers as follows:

`userFrame`→`pid1` = 0,
`userFrame`→`pid3` = 0,
`userFrame`→`pid4` = 0,
`userFrame`→`pid2` = Protection Id of the Actor's context.

(e) Initialize the thread's space registers and the instruction address queues as follows:

`userFrame`→`sr4` = spaceId of thread's Actor,
`userFrame`→`sr5` = spaceId of thread's Actor,
`userFrame`→`pcsqh` = spaceId of thread's Actor,
`userFrame`→`pcsqt` = spaceId of thread's Actor,
`userFrame`→`sr6` = KernelSpaceID.
`userFrame`→`sr7` = KernelSpaceID.
`userFrame`→`pcoqh` = threadParams→pc,
`userFrame`→`pcoqt` = `userFrame`→`pcoqh` + Instruction length (4 bytes).

(f) Initialization of the data pointer (`dp`) of the thread is performed as follows:

i. If the thread is a kernel thread then set `dp` as follows:

`userFrame`→`dp` = `data_pointer` (the kernel's data pointer).

ii. If the thread belongs to a user Actor then set `dp` as follows:

`userFrame`→`dp` = 0x40000000 (the absolute virtual address of a user Actor's data pointer).

iii. If the thread is not the first thread of the supervisor actor then initialize the `dp` from the datapointer value in the saved context of the first thread of the supervisor actor. This value can be found by looking at the thread list attached to the thread's actor.

iv. If none of the above cases is true, then do nothing to initialize the `dp`. This is the case when the new thread is the first thread of the supervisor actor. Since this thread is the main thread of the actor, the start up sequence will be similar to a unix process, i.e., the execution starts at an entry point in a crt0.o equivalent and then branches to main() after some initialization. The `dp` in this case would be set by the code in crt0.o. The code for the crt0.o equivalent is in **ktests/PARISC/kt_ass.s**.

It is important for the `dp` to be set before any part of the main program gets executed since instructions produced by the compiler are generated with respect to the `dp`.

5. The variable `switchFrame` points to the frame that is equivalent to a context frame saved by the scheduler during a context switch operation. The fields are initialized such that on a context switch to the new thread, control is transferred to the kernel procedure **SupThreadStart()** which executes in privileged mode without preemption. **SupThreadStart()** is implemented in **kern/PARISC/supctx.s**. This routine loads the values from the `userFrame` portion of the system stack and performs a return from exception sequence to transfer control to the new thread's actual entry point. The return from exception sequence is described in section 2.2.11. The frame pointed by `switchFrame` is initialized as follows:

   (a) Initialize the Processor status word as follows:

   `switchFrame`→`ipsw` = Q + C + D, Note that Interrupts are not enabled.

   (b) Initialize the data pointer `dp` to the kernel's data pointer:

   `switchFrame`→`dp` = `data_pointer`

   (c) Initialize the space registers:

   `switchFrame`→`sr4` = KernelSpaceId;
   `switchFrame`→`sr5` = KernelSpaceId;
   `switchFrame`→`sr6` = KernelSpaceId;
   `switchFrame`→`sr7` = KernelSpaceId;

   (d) Initialize the instruction address queues:

   `switchFrame`→`pcsqh` = KernelSpaceId;
   `switchFrame`→`pcsqt` = KernelSpaceId;
   `switchFrame`→`pcoqh` = **SupThreadStart**,
   `switchFrame`→`pcoqt` = **SupThreadStart** + Instruction length (4 bytes).

6. Thread descriptor `ptThreadDesc` fields are initialized as follows:

   (a) `typeCtx` = privilege value passed in threadParams.

   (b) `currCtx` = `stackbot` + 2 * `fr_size`.

   (c) `userCtx` = `stackbot` + `fr_size`.

Discussion :

Note that to change a value in the switchFrame, we have to subtract `fr_size` bytes from currCtx and then use the resultant address as `KnThreadCtx*`. The same argument is true for the userFrame. The pointers currCtx and userCtx are fixed for the life time of the thread. There is an inefficiency in space usage and time of access to the context by this definition. First, 2*`fr_size` bytes are lost in the system stack. To access a register in the current context, one has to first get the context pointer from the context pointed to by currCtx and then access the register. This would not have been necessary if currCtx was not fixed but pointed directly to the current context. The advantage of the current approach is in debugging. Since the currCtx is always available at a fixed position relative to the bottom of the system stack, it offers an easy way of looking up the current context of the thread during memory dumps.

The initialization of the `dp` is complicated by the fact that the `dp` is not always available to the kernel at the time of performing the machine dependent initialization.

This is a consequence of the fact that we did not have a data pointer field as part of the machine dependent context `mmuContext`. Having a `dp` field in the class `mmuContext` works fine as long as

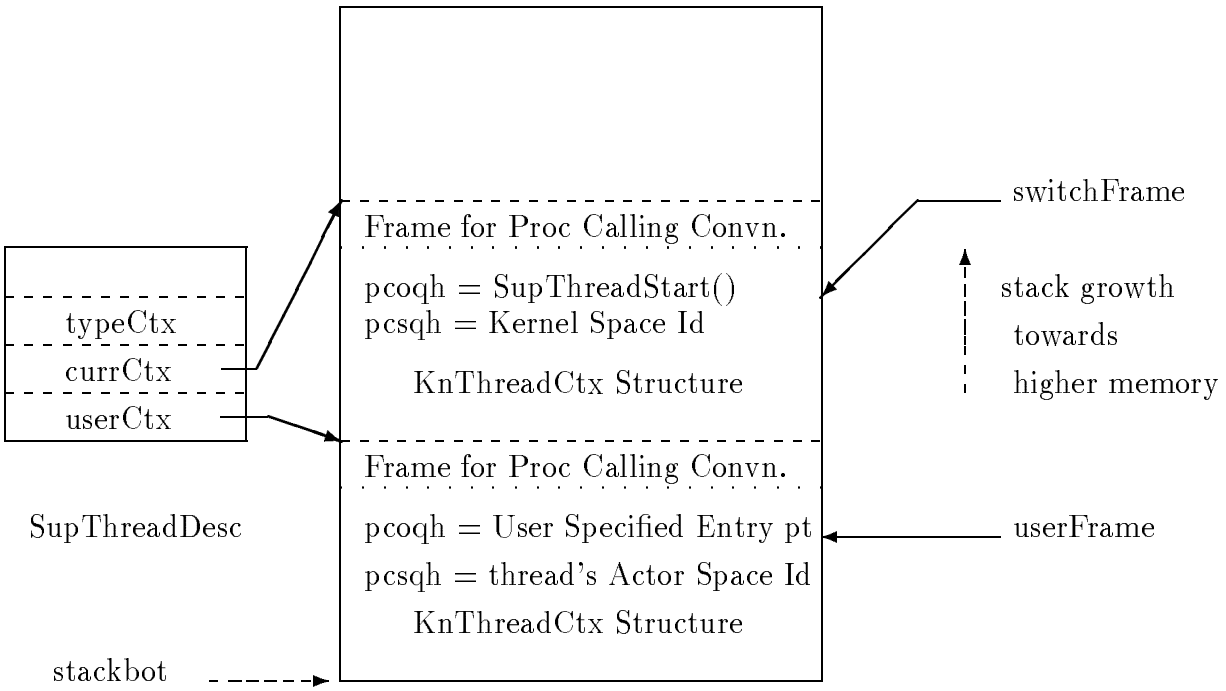

Figure 5: Machine dependent Thread Initialization

there is one executable image per virtual address space. In the case of Chorus all supervisor actors, which are independent executable images, share the same context, i.e., `KernelContext`. In such a case, it is no longer sufficient to have one `dp` field per `mmuContext`. One solution to simplifying the initialization code for `dp` is to have a `dp` in the `actor` and have this field initialized during the startup sequence of the main thread of the actor. Then for all the threads created then onwards in this actor, this field can be used to initialize the `dp` in their startup thread context. Note that in this case, a thread of actor `A` cannot create any other thread other than the main thread of actor `B` unless the main thread of actor `B` has already executed its startup sequence and initialized it's actor specific `dp`. The most elegant solution would be to do the `dp` initialization at the time of creating an actor. But there is no clear way of initializing the data pointer of an actor in the portable layers.

2.2.4 SupCtxSwitch()

SupCtxSwitch(), the thread machine dependent context switch function, is implemented in **kern/PARISC/supctx.s**. This function performs the switch in the following manner[7].

1. Allocate a `fr_size` frame on the system stack of the old thread from the current stack pointer. Save general registers including the current sp and thread specific control registers. The return address of SupCtxSwitch() becomes the new point of resumption for the oldThread. This implies that when a context switch loads back the old thread, it will be as though it has returned from SupCtxSwitch(). To achieve this `pcoqh` and `pcoqt` are set to RP and RP+4

[7]Note that oldThread is really the running thread and the purpose of **SupCtxSwitch()** is to save the register context of the running thread and switch to the new thread

respectively. Update the stack pointer value in the fixed switchFrame pointed to by currCtx [8].

2. Find the position of the save context frame from the stack pointer field in the fixed context frame pointed to by `newThread→currCtx`. Restore general registers and some control registers. Perform a return from exception sequence (see step (3f) of trap handling) to set the instruction queue registers and the process status word.

Discussion :

Doing a return from exception sequence is not necessary to implement the context switch. An alternate way is to do a procedure call return into the context of the new thread (the thread to be scheduled). This can be achieved by performing a branch to the value in the RP of the context of the new thread. The reason the return from exception sequence was chosen is to have more control over the PSW bits during debugging. The PSW bits can be changed in the saved context of the blocked thread and thus enable any debug traps if required. The disadvantage is the inefficiency in this method of implementation due to the greater number of operations that need to be performed.

2.2.5 SupGetUserCtx()

This function is implemented in **kern/PARISC/sv.h**.

2.2.6 SupCtxReset()

This function is implemented in **kern/PARISC/sv.cxx**. The function blindly overwrites the context frame on the stack by the user context frame portion of the `SupThreadDesc`.

Discussion :

This function needs to be changed to perform some sanity and protection checks before modifying the thread context.

2.2.7 SupCtxIsUserMod()

This fucntion is implemented in **kern/PARISC/sv.h**. The function returns *true* if the `sr4` of the context is not the same as the kernel's space id. Other wise it returns false.

Discussion :

One of the experiments which we want to do is to take the supervisor actors from the same space as the kernel and use the multiple privilege levels. One of the interesting aspects of the architecture is the cheap mechanism for system calls and the multiple privilege levels. This function would definitely break if such a separation is done.

[8] Actually `currCtx` points to the top of the switchFrame. Therefore, `fr_size` bytes have to be subtracted from `currCtx` before accessing a field in the KnThreadCtx structure portion of the switchFrame

2.2.8 The various connect and disconnect functions

This section details the implementation of **SupCall[Dis]Connect()**, **SupIt[Dis]Connect()** and **SupTrap[Dis]Connect()**. These three pairs of functions are implemented in **svConnect.cxx** in the directory **kern/PARISC**.

These functions are provided by Chorus so that the portable layers of the Chorus kernel can attach handlers to traps, interrupts, system calls in a machine independent manner. The chorus kernel maps the **Connect()** system calls provided to supervisor actors to the corresponding **Connect()** calls of the *Supervisor* interface. We found that we needed to have more information than the specified parameters for the **Connect()** calls to satisfy the system call interface for supervisor actors. This problem is discussed in detail in section 2.2.9. In this section, we will discuss the different semantics that apply to some of the parameters to these ***Connect()** calls on PA-RISC in contrast to what Chorus originally envisaged. The semantic differences are as follows:

- The Chorus interface expects the first parameter to **SupCallConnect()** and **SupTrapConnect()** to be a trap number. **SupCallConnect()** is used in the Chorus kernel to connect a vector of handlers behind a trap. In the case of implementing UNIX as a collection of actors (*sub-system*), a **SupCallConnect()** is made from the *sub-system* actor called the PM to connect a vector of system call routines behind a specified trap number to handle UNIX system calls. In the case of PA-RISC, it is not necessary to cause a trap to perform a system call. There is an efficient *gateway* mechanism (see section 4) by which a controlled transfer between privilege levels can be achieved. This method is the proper mechanism for making system calls on this architecture (HP-UX and MPE use the same approach) and has been used in the implementation of system calls for PA-Chorus. The consequence is that calling the first parameter to **SupCallConnect()** as `trapNb` is not quite correct. Since the purpose of having **SupCallConnect()** and **SupTrapConnect()** is to build sub-systems (including Chorus as the base case), this number is distingushed in the *Supervisor* layer as a real hardware trap number or a *sub-system* number to which a vector of handlers (**SupCallConnect()**) or a single handler (**SupTrapConnect()**) should be connected. Hence an appropriate name for the first parameter would be subSysOrTrapNb. Additional constants were added in **include/PARISC/syscall.h** to map symbolic constants for sub-systems to numbers. For example:CHORUS_SUBSYS is defined to be 31. The range of hardware trap numbers does not overlap with the range of sub-system numbers possible. This makes differentiation between a sub-system number and a trap number simple.

- The Chorus interface expects the first parameter to **SupItConnect()** to be a hardware interrupt number. In the case of PA-RISC, all the external devices including the clock raise the same interrupt #4 (External Interrupt). We found it more elegant to pass the number of the external device as a parameter to SupItConnect() rather than the interrupt number. For example:To connect clock and disk handlers, two calls SupItConnect(CLOCK, clockHdl, clockprio), SupItConnect(DISK01, diskHdl, disk01prio) are required. The interface is not changed, only the meaning of the first parameter is slightly different. However, it is possible to connect all the device handlers to the external interrupt #4. Hence an appropriate name for the first parameter would be DeviceOrIntrNb. An include file **include/PARISC/extern_intr.h** was created to map symbolic constants for devices to mapped to integers. The numbers possible for hardware interrupts do not overlap with that of the devices. This makes differentiation between a device number and a interrupt number simple.

The basic data structures that have been used for the implementation are presented in figure 6.

```
typedef struct {
    unsigned long   funcNb;        /* No of functions in the array being connected */
    KnCallEntry*    calls;         /* address of the array */
} vector_desc;

typedef struct {
    unsigned int    connectType;   /* Array or function connected */
    union {
        vector_desc vector;
        KnHdl       hdl;
    };
    VmAddr          dataPointer;   /* The $global$ of the Supervisor Actor */
} supCallTbl;

/* MAX_SYS_NUM = 25 from include/PARISC/syscall.h; the maximum number *
 * of sub-systems that can simultaneously run on top of Chorus          */
   supCallTbl userTrapVect[MAX_SYS_NUM]; /* For User Actor System Calls */
   supCallTbl kernTrapVect[MAX_SYS_NUM]; /* For Sup  Actor System Calls */
```

Figure 6: Data structures for System Call Handling

- The structures `userTrapVect` and `kernTrapVect` are used by **SupCallConnect()**, **SupTrapConnect()**.**SupCallHandler()** finds and executes the routine attached by the **Connect()** calls during system call execution.
- `vect`, `DeviceVect` are updated by **SupItConnect()** and **SupTrapConnect()**. The routines attached to the various interrupts and traps by the **Connect()** calls are executed by **SupItHandler()**,**SupItSelector()** or **SupTrapHandler()** on the occurrence of the those *interruption*s . The data structures for *interruption* handling are presented in figure 7. In addition to the 25 PA-RISC interruptions, there are 3 psuedo-interruptions generated by the low-level event handling layers. Therefore, we maintain an INTRMAX element array (25+3 elements) and allow handlers to be attached to one single vector `vect`. Currently the psuedo-interruptions are handled in the kernel itself.

  PA-RISC has one external interrupt for all external devices including the ITMR. This implies that all handlers for different external devices would have to be connected to the same position in `vect`. To avoid this, `DeviceVect` is introduced to keep the interrupt handlers device-specific rather than connecting all the device handlers to one position in the `vect` array.

The implementation of the various **Connect()** calls is now presented:

SupItConnect(itNum, hdl) : calls **SupItConnectParisc()** with kernel's **$global$** as additional parameter. **SupItConnectParisc()** allocates a `itLink` structure from the `itPool`, stores the parameters of the function in the structure and attaches it to `vect` or `DeviceVect` depending on the actual interrupt number parameter (`itNum`) being a PA-RISC interrupt number or a symbolic device name. Two or more handlers for the same interrupt or device are linked in descending order of priority.

```
typedef struct {
    itLink* lnk;
    KnHdl routine;
    unsigned long priority;
    VmAddr dataPointer;
} itLink;

typedef struct {
    KnHdl routine;
    VmAddr dataPointer;
} KnHdlEntry;

typedef union {
    itLink* lnk;
    KnHdlEntry hdlEnt;
} VectEntry;

VectEntry vect[INTRMAX];
VectEntry  DeviceVect[NumOfExternalDevices]
```

Figure 7: Data Structures for *Interruption* handling

The advantage of having a separate table for external devices is for efficiency of search during interrupt handling.

SupTrapConnect(trapOrSubSysNum, hdl) : calls **SupTrapConnectParisc()** with the kernel's **$global$** as an additional parameter. **SupItConnectParisc()** does the following:

- If `trapOrSubSysNum` is a PA-RISC trap, then store the dataPointer and `hdl` at `vect[trapOrSubSysNum]`.
- If `trapOrSubSysNum` is sub-system number, then this implies a single routine interface for system calls in contrast to a vector of handlers connected by **SupCallConnect()**. Update both `userTrapVect` and `kernTrapVect` by the same parameters and update `connectType` to be FUNC_TYPE.
- If neither of the above condition holds, then return illegal value status.

As explained in the beginning of this section, it is not necessary to cause a trap to perform a system call on this architecture. If a sub-system manager requires a single routine to handle all the system calls instead of a vector of handlers, there are now two ways of doing it:

- **SupTrapConnect()** with subsystem number instead of trap number as first parameter and the handler as second parameter. This is the interface used by the UNIX sub-system implementation to provide UNIX system call services to the actors of the sub-system.
- **SupCallConnect()** with size of the array equal to one. In this case the stub library should always have the system call number equal to 1, and the single handler in `vectorOfHandlers` responsible for distingushing various system calls of the actors running on that sub-system.

SupCallConnect(no, vectorOfHandlers, NoHdl, privilege) : calls **SupCallConnectParisc()** with the kernel's **$global$** as an additional parameter. **SupCallConnectParisc()** connects the `vectorOfHandlers` to `userTrapVect[no]` if `privilage` is *User* or to `kernTrapVect[no]` if `privilage` is *Supervisor*. The `connectType` is set to VECTOR_TYPE.

DisConnect functions : All the disconnect functions are straight forward and basically reset the corresponding locations to NULL or deallocate the allocated structure as in the case of interrupts.

2.2.9 Supervisor Actor Interface Implementation

In the case of PA-RISC, the compiler generates instructions that access data relative to general register 27 (DP or DataPointer). During the startup of a thread this register is set to **$global$** (of the address space) before the actual code gets executed. The **$global$** refers to the starting address of the the **$DATA$** section of a typical UNIX process. Chorus requires the kernel and supervisor actor to live in the kernel address space. Chorus assumes that it is possible to make a simple procedure call to a procedure in the the supervisor actor from the kernel as the actors are in the same address space, i.e., the kernel address space, even though the two actors are two separate executable images. During the port, this requirement that the supervisor actors should live in the same context as the kernel has been satisfied by laying out the supervisor actors including the kernel in distinct regions in the 30-bit virtual address space corresponding to the same spaceId=KernelSpaceId. Because of this, the **$global$** address is different for each of the supervisor actors and the kernel. The consequence of this design decision is that it is no more a simple procedure call from the kernel into the supervisor actors. If a procedure p_s of a supervisor actor s has to be called in the interrupt handling sequence, then the procedure p_s can be called only after the DP register has been updated to that of the supervisor actor. In addition the kernel's `dp` should be restored when returning from p_s. We considered the following implementation approaches:

- DP of the supervisor actor should be known by the kernel at the time of calling the interrupt handler.
- The routine should know that it should set the DP to its **$global$** and restore the kernel's **$global$** at the end of the routine.
- The routine should know that it should the set the DP to its **$global$** and the kernel should restore its own DP after returning from the call.

The second approach was rejected because, even if submerged, the setting and restoring of DP using static variables in the system call stub at the time of the **Connect()**, it is not a robust mechanism and can be broken by a misbehaving supervisor actor. Of course, as supervisor actors are trusted, there are other ways in which a misbehaving supervisor can crash the kernel but we did not want add more ways.

The third approach required modification in the kernel interface and the stubs. It was rejected because using static variables in system call stubs did not appear to be elegant and it required kernel modification.

We examined two alternatives of obtaining supervisor actor's DP at time of calling the handler:

- Modify the Supervisor Actor interface and the *Supervisor* interface to pass the datapointer as an additional parameter to all those functions that required a routine in the supervisor actor to be called.

- Pass the DP as a hidden parameter during the system call and keep the machine dependency in the the machine dependent layers as much as possible.

The first approach was rejected because it would modify the machine independent interface of the Chorus Kernel and would require modifications in the supervisor actors already written.

In the current implementation, the DP of the supervisor actor is passed as a hidden parameter during the system call. For all the calls given in the section 2.1.2, the following scheme has been adopted. If the system call name is *scf* then this would call a PA-RISC specific function *scfParisc* which takes the DP of the actor as an additional argument. The stub will be generated for *scfParisc* rather than for *scf*. For example: **svTimeOut(routine, param, delay)** calls **svTimeOutParisc(routine, param, delay, get_dp())**. **get_dp()** returns the DP of the actor. The system call stub is generated for **svTimeOutParisc()** rather than for **svTimeOut()**.

In the kernel, **kern/scSystem.cxx** is modified. All the kernel routines which now require the knowledge of the DP of the requesting supervisor actor are replaced by functions that have the same name with **Parisc** suffix. For example: **KnTimeOut()** is replaced **KnTimeOutParisc()** and the number of arguments field is incremented by 1.

The following is a list of the changes at the supervisor stub library level:

- **svAbortHandler(actcap, routine)** calls **svAbortHandlerParisc(actcap, routine, get_dp())**. A stub is generated for **svAbortHandlerParisc()**.
- **svExcHandler(actcap, routine)** calls **svExcHandlerParisc(actcap, routine, get_dp())**. A stub is generated for **svExcHandlerParisc()**.
- **svCallConnect(trapNo, hdlVect, NoHdl)** calls **svCallConnect(trapNo, hdlVect, NoHdl, get_dp())**. A stub is generated for **svCallConnectParisc()**.
- **svItConnect(trapNo, hdlVect, NoHdl)** calls **svItConnect(trapNo, hdlVect, NoHdl, get_dp())**. A stub is generated for **svItConnectParisc()**.
- **svTrapConnect(trapNo, hdlVect, NoHdl)** calls **svTrapConnect(trapNo, hdlVect, NoHdl, get_dp())**. A stub is generated for **svTrapConnectParisc()**.
- **svTimeOut(routine, param, delay)** calls **svTimeOutParisc(routine, param, delay, get_dp())**. A stub is generated for **svTimeOutParisc()**.

The following is a list of changes in **kern/scSystem.cxx**: All the modifications are done under compilation flag PARISC.

- Replace **scSvAbortHandler()** taking 2 parameters by **scSvAbortHandler()** that takes an additional parameter DP.
  The call `workActor→setAbortHdl(f)` is changed to `workActor→setAbortHdl(f, dataPointer)`.
- Replace **scSvExcHandler()** taking 2 parameters by **scSvExcHandler()** that takes an additional DP parameter.
  The call `workActor→setExcHdl(f)` is changed to `workActor→setExcHdl(f, dataPointer)`.
- Replace **SupCallConnect()** taking 2 parameters by **SupCallConnectParisc()** that takes an additional DP parameter.

- Replace **SupItConnect()** taking 2 parameters by **SupItConnectParisc()** that takes an additional DP parameter.
- Replace **SupTrapConnect()** taking 2 parameters by **SupTrapConnectParisc()** that takes an additional DP parameter.
- Replace **KnTimeOut()** taking 2 parameters by **KnTimeOutParisc()** that takes an additional DP parameter.

The following is a list of changes in **kern/knMk.hxx**: All the modifications are done under compilation flag PARISC.

- Add two fields acPariscExcDp, acPariscAboDp for recording the DPs of the supervisor actors that have performed **svExcHandler()**, **svAbortHandler()** respectively.
- Extend the parameter list of **setExcHdl()**, **setAboHdl()** to take DP as a parameter. Add an additional assignment of the DP to acPariscExcDp and acPariscAboDp respectively.
- Modify **execExcHdl()/execAboHdl()** to call **SupTrapStub()** with exception context, exception number, exception/abort handler to be called and the DP of the exception/abort handler routine as parameters.

The following is a list of changes in **kern/knMk.cxx**. All the modifications are done under compilation flag PARISC.

- The `timeOutItem` structure has an additional field: `DataPointer` that gives the **$global$** of the supervisor actor to which the routine belongs.
- **KnTimeOut()** calls **KnTimeOutParisc()** which has an additional parameter DP. The DP is the **$global$** of the kernel.
- **KnTimeOutParisc()** is exactly the same as **KnTimeOut()** but takes an additional parameter and performs the data pointer assignment into the timeOutItem structure.
- **KnProcessTimeOuts()** has been modified at the point of calling the timeOut routine. **SupTrapStub()** is called to take care of switching to the data pointer of the timeOut routine before executing the routine and then restoring the data pointer of the kernel when returning to the kernel.

SupTrapStub() is implemented in **kern/PARISC/SupTrapStub.s**. It allows the kernel to call the supervisor actor routines that have instructions generated with a different **$global$** . It takes the following parameters:

- Pointer to the thread's context, `KnThreadCtx* ctx`
- Interruption number, `int no`
- Routine address, `int (*fnPtr)();`
- **$global$** of the actor to which the routine belongs, `VmAddr global`.

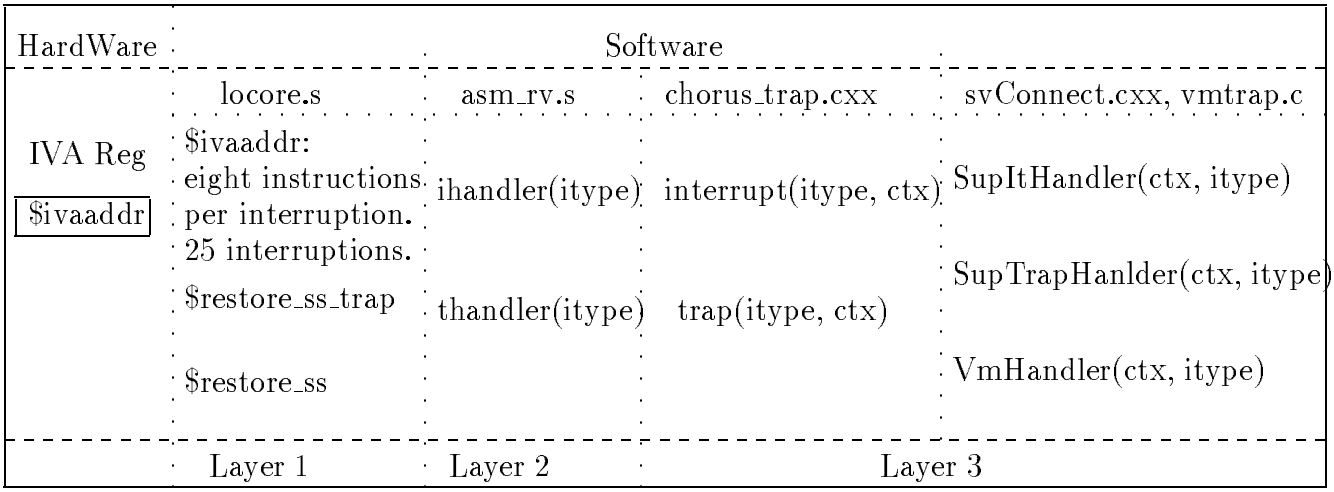

Figure 8: Chorus Event Handling Sequence

2.2.10 Interrupt masking and monitoring functions

The section details the implementation of SupItLevel(), svMask[All]() and svUnMask[All](). SupItLevel() is implemented in **kern/PARISC/sv.h** and the functions sv[Un]Mask[All]() are implemented in **lib/HP800/svMask.s** The mask/unmasking functions are implemented by calling the appropriate **spl()** routines of Tut code implemented in **lib/PARISC/asm_utl.s**.

2.2.11 Event Handling

The event handling sequence has been implemented in 3 layers and is shown in fig 8.

The IVA control register is first initialized with the code addr **$ivaaddr** in **kern/PARISC/locore.s** during the kernel Initialization phase. This address is page-aligned and is therefore 1024-byte aligned as required by the PA-RISC interrupt architecture. This will be referred to as the **interrupt vector table** in the following discussion.

PA-RISC interruptions are classified in the following manner:

- PA-RISC Interruptions #1, #2, #4 and #5 have been classified as interrupts.
- The remaining PA-RISC Interruptions are handled as traps. Of these traps, #6, #7, and #15 to #21 pertain to memory management. HP 9000/834 (the target architecture) has software TLB handling. So the interruptions #6, #15, #16, #17 are actually TLB misses but may result in page faults if the page is not in the PDIR (see section 1.3.3).

Recall that on an interruption, hardware branches to the code address given by the following relation:

code address = *IVA* + 8*4**interruption number*.

Depending on the interruption, the interruption parameter registers are updated by hardware. The processor is in physical mode. Interrupts are turned off. The PSW_Q bit in the PSW is disabled.

All the interrupt and trap handling code resides in the directory **kern/PARISC**. An overview of the interrupt and trap handling is as follows:

- Hardware branches to the effective address in the **interrupt vector table** as detailed above. At this level a few registers are saved in control registers to obtain some working registers. A branch is performed to **ihandler()** for handling interrupts or to **thandler()** for handling traps[9].

- **ihandler()** and **thandler()** save the context of the executing thread. **thandler()** always stores the context on the system stack of the current thread and executes on the same. **ihandler()** uses the system stack of the current thread for storing the context and for execution if interrupt nesting level is zero. Otherwise, the function uses the *interrupt control stack* (ICS). In the general case **thandler()** calls **trap()** and **ihandler()** calls **interrupt()**.

- **interrupt()** executes the interrupt routines attached by {**Sup, sv**}**ItConnect()** by calling **SupItHandler()**. **trap()** directs all the memory management traps to **VmHandler()** and non-memory management traps to **SupTrapHandler()**.

- **ihandler()**, after returning from the call to **interrupt()**, checks if rescheduling or aborting the thread is necessary, and by default, restores the context of the current thread. Depending on whether **ihandler()** was running on the ICS or on the system stack of the current thread, it has to perform a different restore sequence to return to normal execution. The reason for the difference is that threads' system stacks are not equivalently mapped, whereas the ICS is equivalently mapped, i.e., virtual and physical addresses are the same. **$restore_ss** is the final restore sequence used on the ICS and **$restore_ss_trap** is the final restore sequence used on the system stack. **$restore_ss_trap** and **$restore_ss** are text addresses in **locore.s**. A branch is taken to this code to cause a potential TLB refill and then the code gets executed. This branch is necessary because code updating the interruption instruction queues and IPSW should not cause TLB misses as the PSW_Q bit is turned off.

  thandler() after returning from the call to **trap()** restores the context not connected with instruction queues and PSW etc, and performs the final restore sequence to restart executing the code of the current thread by branching to **restore_ss_trap**.

This completes the overview of the implementation of trap and interrupt handling.

A detailed description of the control flow of interrupt handling is now presented. The variable `istackptr` contains the pointer to the bottom of the ICS if the code is not executing on the ICS. `istackptr` is set to zero whenever the code is running on the ICS. The variable `nbit` indicates the current interrupt nesting level.

The detailed flow of control for interrupt handling is given below.

1. Hardware branches to one of the first level interrupt handlers in **locore.s**. The processor is in physical mode. Interrupts and the PSW_Q bit are disabled.

2. The first-level handler saves registers ARG0, SP, T1 which will be used as scratch and branches to **ihandler(itype)** in **asm_rv.s**

3. **ihandler()** performs the following operations:

[9] PA-RISC interruptions #6, #15, #16, #17 are are treated slightly differently. All TLB misses are first handled in **kern/PARISC/locore.s**. If the page is found in the PDIR then the tlb is refilled and control is returned to the executing thread. If the page is not found in the PDIR (see Inouye [11]), then **thandler()** is called with the appropriate page fault type (Instruction [non-access] page fault or Data [non-access] page fault) as the case may be.

(a) Increment `nbit`. If `nbit` equals two, then switch to the ICS.

(b) If `nbit` is greater than two, continue using the current SP as we will be definitely on the ICS.

(c) If `nbit` equals one, then check for which one of the following cases is true and execute the corresponding code:

   i. Already on ICS. This case was possible as interrupts were turned on during booting. This case can arise only if interrupts are enabled during booting. The interrupts are not enabled during booting in the normal execution. In any case, if interrupts are re-enabled during booting, the current SP is used.

   ii. Check if we are running on the user stack or system stack of the thread. The state has to be saved on the system stack in this case. This requires saving the interruption parameter registers in the equivalently mapped `tmp_save_state` structure, turning on virtual memory and checking `CurThread`'s `ksp`. If `ksp` equals zero, the thread was executing on its system stack and so keep the SP unchanged. If `ksp` is not equal to zero, the thread was executing on its user stack and `ksp` contains the pointer to its system stack. Set SP equal to `ksp` and set `ksp` to 0.

(d) Make sure there is enough space on the selected stacks. If executing on the system stack and system stack overflow is detected, switch to ICS and change interrupt number to *kernel stack overflow* pseudo-interrupt. If executing on the ICS and ICS overflow is detected, change the protections of the overflow page available just after the ICS. Change the interruption number to *ICS overflow* psuedo-interrupt. Both these events are currently unrecoverable. **interrupt()** passes them to **trap()** which gives a panic message and crashes.

(e) Allocate the context frame of size `fr_size`(see section 2.2.3) on the selected stack.

(f) save context on the stack, set DP to **$global$** of the kernel and call **interrupt()** in **chorus_trap.cxx** with arguments context pointer and interrupt number.

   i. There are two ways **interrupt()** could be called: If a trap occurs on the ICS or a genuine interrupt has occurred. In the first case, redirect the parameters to **trap()**. In the latter case, call **SupItHandler()**.

   ii. **SupItHandler()** calls routines attached by the **ItConnect()** calls. The various interrupts are handled as follows:

   - *External Interrupt* #4: the attached routine is **SupItSelector()**. This routine is attached in **SupBoardInit()** in **svBoard.c** during kernel initialization. The clock handlers connected in **SupBoardInit()** get executed by **SupItSelector()** every time interrupt #4 occurs and the EIRR 0 bit is set. These handlers are part of the kernel and are necessary to satisfy the implementation specification.
   - *Power failure interrupt* #2, *Low-priority machine check* #5: These interrupts do not have a real handler in the current implementation.
   - *High priority Machine check* #1: This interrupt is handled at the first level itself in **locore.s** and is unchanged from the **Tut** implementation.

(g) Disable interrupts. Decrement `nbit`. If `nbit` equals one, then restore `istackptr` to ICS stack bottom. This is because the interrupt handling at nesting level 0 is performed on the system stack of the current thread.

(h) If `nbit` equals zero then check for execution mode of the interrupted thread. if the thread was executing supervisor mode then call **KnRetSup()** and go to step (3j. Otherwise call **KnRetUser()**.

(i) If **KnRetUser()** returns non-zero, call **KnAbortHandler()**.

(j) Restore saved context by loading all the context except those related with PC queues, SP, PSW, and interruption parameter registers. If executing on the interrupt stack, then branch to **$restore_ss** in **locore.s**. Otherwise, branch to **$restore_ss_trap** in **locore.s**.

4. In **$restore_ss** interruption instruction queues and parameter registers are going to be written into and since the Q-bit will be disabled, there can not be any TLB misses. Disable interrupts. Turn off VM and Q bit and Protection bits. Restore the remainder of the state from the previous interrupt after calculating the new `psw`. Restore SP and perform *rfi* instruction. The new psw is given by the following relation:

   New psw = (GLOBAL_VAR_MASK && `ipsw`) || global_psw

   The operations at **$restore_ss_trap** will be described in the trap handling description.

Once the restore sequence is done, the normal mode of execution is resumed. A detailed flow of execution for trap handling is now presented:

1. Hardware branches to one of the first level trap handlers in **locore.s**.

2. The first-level handler saves registers ARG0, SP, T1 which will be used as scratch registers and branches to **thandler(itype)** in **asm_rv.s**

3. **thandler()** performs the following operations:

   (a) If currently executing on the ICS, then allocate a context frame on the stack, store the interruption parameter registers and branch to step (3f) in **ihandler()**. Increment `nbit` to ensure compatibility with interrupt handler code to execute correctly.

   (b) If not executing on the ICS, then the context has to be saved on the system stack of the thread. Find stack on which the thread was executing. This requires the VM to be enabled. Before turning on VM, store the registers that might get trampled by turning on VM due to tlb misses into the equivalently mapped `tmp_save_state` structure. If current thread's `ksp` equals zero, the thread is executing on system stack. Otherwise, `ksp` contains the system stack pointer of the thread. Obtain the stack pointer if `ksp` is non-zero and set the SP register to `ksp`. Otherwise, do nothing.

   (c) If sufficient space is unavailable on kernel stack, switch to the ICS and change interruption number to the psuedo interruption *kernel-stack-overflow* (I_KS_OVFL) and allow the interruption number to float up to the next layer just as in **ihandler()**.

   (d) Allocate space on the selected stack, store the context onto the stack. Set DP to **$global$** of kernel and call **trap()**.

   (e) **trap()** passes all the memory management traps to **VmHandler()**. The traps *Break*, *High Privilege Transfer*, *Low Privilege Transfer*, *Taken Branch* are passed to **SupTrapHandler()**. The traps *Assist Emulation* and *Assist Exception* are not currently supported. The default behavior for un-supported or irrecoverable errors is to panic with a message.

SupTrapHandler() calls the connected handler if a handler to the trap is connected. Otherwise, it calls **KnHandler()**. **SupBrkHandler()** is connected to the *Break* trap. This checks the parameter of the *break* instruction available in the context and calls the Kernel debugger if the condition succeeds otherwise, it calls **KnHandler()**.

VmHandler() checks the type of trap and the space (user or system) in which the trap has occurred. For some of combinations that are unrecoverable (for example:Instruction page fault in the system space), the function panics. For the various trap and space pairs for which the portable layers can make a decision, **VmHandler()** calls **fault_handle()** which calls **execPageFault()**. The Chorus page fault interface requires **execPageFault()** to be implemented by the machine dependent layer (see section 3). The handling of various memory management traps is outlined below:

TLB miss faults : Before performing any other action, the Chorus memory management data structures need to be consulted. **fault_handle()** calls **execPageFault()** to determine whether the page has been mapped in the portable structures but not yet allocated by the machine-dependent layer.

Non-access TLB miss faults : For non-access TLB misses, PDIR search failure for LPA and PROBE instructions does not result in a page being brought into memory. Therefore, fault handling for these cases also ends in stage one. For LPA, we may have to modify the base register (if specified in the faulting instruction). PROBE and LPA handling ends by setting the N bit which nullifies the next instruction.

Memory Protection Faults : Protection and alignment faults are handled exclusively in stage three. Alignment faults result when a either a store or a load instruction access an address which is not aligned as per the requirements of the specific instruction. Alignment faults are exclusively due to bad code and their handling ends in stage three by sending an error message to the user process. Protection faults may occur due to illegal accesses to pages or due to copy-on-write violations. The latter necessitates a call to **execPageFault()**.

(f) Begin the return from exception sequence. Restore most of the context from the context frame except those pertaining to interruption address queues, `psw` and working registers. Branch to **restore_ss_trap**.

4. **$restore_ss_trap** is more complicated than **restore_ss** since we are dealing with non-equivalently mapped system stacks. We need to first copy the context to be restored in to the equivalently mapped `tmp_save_state` structure before we can turn off VM. Operations similar to **restore_ss** are performed. In addition, if the thread is returning to user mode, the `ksp` of the thread is updated. This check can be done by checking the nesting level of the traps on the system stack.

Discussion :

We reused most of the first and second level handling code from the **Tut** project and this greatly simplified the implementation. The first level of interruption handling in **locore.s** is unchanged from Tut. The second level of Trap handling code **asm_rv.s** remained basically the same except for a few modifications related to the accessing the current thread and its descriptor. The offsets had to be modified to access the appropriate fields in the `KnThreadCtx` structure. Interrupt handling code required extensive changes. Chorus requires that on the first interrupt (nest level 0), the context of the current thread is saved on the system stack and not on the ICS as in the case of

Tut . This means all the problems of non-equivalently mapped stacks that arise in trap handling also find their way into interrupt handling. Chorus requires the current interrupt nesting level to be maintained by the machine dependent layers. This was one of the additions. The third level of handling is more operating system specific and had to be written for Chorus, although some pieces of **Tut** code were reused.

2.2.12 Timer and Console Management

Timer management is implemented in **kern/PARISC/svBoard.c**.

The function **SupBoardInit()** connects clock handler **clock()** and **clock_ack()** to the clock interrupt in descending order of priority. **clock()** calls the portable kernel exported function **KnTimeIn()** and returns. **clock_ack()** acknowledges the interrrupt by resetting the clock interrupt bit in EIRR and reschedules the interrupt by writing (`currentTime` + rescheduling interval) into the ITMR register.

SupPreciseTime() is trivially implemented in the same file by returning the value in the ITMR register.

SupPutChar(), **SupGetChar()**, **SupPollChar()** are also implemented in **kern/PARISC/svBoard.c**. For details on their implementation, see [14].

The *Supervisor* is responsible for connecting at least **SupPutChar()** and **SupGetChar()** behind a trap so that library functions can be implemented. In the case of PA-RISC, three system calls **PutChar()**, **GetChar()**, **PollChar()** are implemented using the system call interface and can be called from user or supervisor actors.

2.2.13 Debugger

The debugger function **SupDebugger()** is implemented in **kern/PARISC/debug.c**. Most of the code to implement the debugger has been ported from Chorus 3.3 sources for the compaq386. The debugger is minimal and can perform the following functions:

- Show what commands are available and syntax (help facility).
- Recover from a break instruction. The debugger does not have the capability of setting a break point. Currently the debugging is done by having an explicit *break* instruction in the source code.
- Modify data
- Hex dump of memory
- Show interruption context if the debugger is called during the interruption handling phase.
- Change debug trace level
- Toggle the more option in during traces. Setting this option would cause traces to pause for input after every 24 lines.
- Show the context switch history of whole system or that of a particular thread. The history displays the following information about the switch:
  - Is the switch voluntary or caused by preemption
  - cause of preemption

– The thread descriptor of the thread which is the destination thread of the context switch.

The history is maintained in a circular buffer.

- Show the history of the interrupts and traps. The histories of traps and interrupts are maintained in separate circular buffers.
- Visualize the scheduler, actor, thread, message and port data structures. This functionality is provided by the portable layers of the kernel (**kern/knPrint.cxx**, **kern/knMk.cxx**) and the appropriate functions were called from **SupDebugger()**

The debugger has been connected behind the PA-RISC *break Trap* (#9) . The instruction *break* causes a *break Trap*. The *break* instruction takes two parameters that can be used in resolving the break instruction processing. The instruction *break* **BI1_DEBUG,0** causes the program to enter the debugger. To achieve this, a generic break handler function **SupBrkHandler()** is first connected to the *Break* trap using **SupTrapConnect()**. See **SupBoardInit()** in **kern/PARISC/svBoard.c**. This handler is invoked by the event handling code(see section 2.2.11). **SupBrkHandler()** calls **SupDebugger()** if the first parameter of the break instruction is BI1_DEBUG.

The library function **callDebug()** is implemented in **lib/PARISC/utDbg.s** and basically contains the *break* instruction with BI1_DEBUG as it's first parameter.

2.2.14 Kernel Initialization

The function **start()** is implemented in **kern/PARISC/sv.cxx** but is not the entry point of the kernel image. In most other Chorus implementations, virtual memory initialization is done entirely in the boot program portion of the boot archive and the kernel has to perform only its own initialization. In the case of PA-RISC, because of the reuse of **Tut** code, it was easier having the kernel do all the initialization in one, mostly unchanged, procedure than try to break up and modularize the low-level code. For more details on booting, see [12].

Control is transferred to the kernel entry point **rdb_bootstrap** in **kern/PARISC/locore.s** from the boot program part of the boot archive. At this point interrupts are disabled and the processor is in physical mode. DP, EIRR, IVA, SP, space registers and some global variables are initialized followed by a call to **realmain()** in **kern/PARISC/vm_machdep.c**. **realmain()** maps the kernel and returns the next available physical page. Then virtual memory is turned on in **kern/PARISC/locore.s** and control transfers to **start()**.

start() performs all the functions specified in section 2.1.6 in addition to disabling the scheduler and initializing the `CurThread` variable that points to the current executing thread. The initialization of `CurThread` is necessary so that trap and interrupt handling code that refer to `CurThread` see a legal value even though the thread has not actually been created. **start()** calls **KnInit()** which makes the executing kernel initialization code the first thread of the operating system. **KnInit()** returns the system stack pointer to be used by the first thread. At this point, the first thread's descriptor gets manually built, in a manner similar to that by **SupCtxInit()**. This is necessary because the thread is already running. It is like bootstrapping the thread abstraction. Then a stack switch from the interrupt control stack (ICS) to the allocated system stack is done followed by a branch to **KnMain()**. This function never returns.

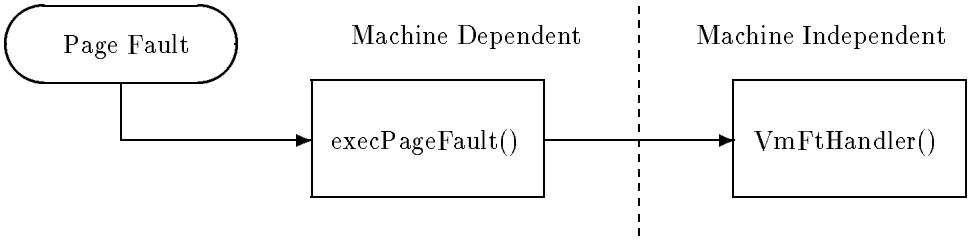

Figure 9: Chorus Page Fault Interface

3 Chorus Page Fault Interface

As there is little documentation on porting the Chorus virtual memory unit, most of this information was gathered through word-of-mouth and assumptions made from reading Chorus source code.

3.1 Requirements

Figure 9 shows the procedural interface between the machine-dependent code and the machine-independent code. The routine **execPageFault()** should be called by all the low-level trap handlers requesting access to the portable layers. This procedure represents the machine specific end of the bridge between the machine dependent (MMU) and the machine independent (VM) layers. The routine **VmFtHandler()** represents the VM end. The file **kern/vm/pvm/pvm.hxx** contains a fault descriptor structure (**gmiPullInArgs**) (shown in Figure 10) that is used to pass information between these two layers and is the only parameter passed to **VmFtHandler()**. The descriptor is created in **execPageFault()**. The machine-dependent layer is responsible for filling in the fields for *ftAddr, ftAccess, nonAccess* and *prContext*. The field *ftAddr* contains the address which caused the fault. The type of access i.e., read or write, is specified in the field *ftAccess*. A pointer to the faulting context is inserted into *prContext*. Non-access page faults are indicated when the *nonAccess* flag is non-zero. The nonAccess flag indicates that the portable layers are only being consulted about page protections and that the faulting page should not be swapped back into memory.

If the page fault is resolved by the upper layers then **VmFtHandler()** returns **K_OK**. The *prPage* field should now contain a refererence to the mmuPage descriptor that represents the desired page. The *prProt* field represents the protections that should be assigned to the page. It is then the responsibility of the machine-dependent section to load the page into the proper context.

3.2 Implementation

The routine **VmHandler(KnThreadCtx* ssp, int type)** implements the memory management trap handlers. All the handlers in stage three which need access to **execPageFault()** call **fault_handle()** , which performs additional checks on the space ID and performs certain recovery actions as described below.

execPageFault() is implemented in **kern/PARISC/mmu.cxx**. This routine sets up the fault descriptor and calls the portable layer.

The Chorus portable layer, specifically **VmFtHandler()**, is called by **execPageFault()** and is passed a pointer to a fault descriptor. Should the page fail to be found, then the kernel exception

```
struct gmiFaultArgs {
    gmiAddr                 ftAddr;         // Set in MMU layer
    gmiFlags                ftAccess;       // Set in MMU layer
    gmiOffset               ftOffset;
    gmiCache*               ftCache;
    gmiFlags                ftFlags;
};

struct gmiPullInArgs : gmiFaultArgs {
    int                     nonAccess;      // Set in MMU layer
    gmiContext*             prContext;      // Set in MMU layer
    vmPage*                 prPage;         // Returned to MMU layer
    gmiFlags                prProt;         // Returned to MMU layer
    gmiOffset               prOffset;
    gmiSize                 prSize;
    operationDesc*          prOper;
    gmiCache*               prCache;
    int                     mapWasOut;
    gmiFlags                mpRequeredAccess;
    gmiOffset               mpAccessOffset;
    gmiSize                 mpDataSize;
    gmiCache*               mpTransitSegment;
    gmiOffset               mpTransitOffset;
    gmiFlags                mpGrantedAccess;
};
```

Figure 10: Fault Descriptor

handler, **KnHandler()**, must be called. In the M88K sources, this is not done in **execPageFault()** but by either **codefault()** or **dataFault()**. This is probably done in this manner because **KnHandler()** locks the kernel, but **execPageFault()** runs with the kernel locked.

The routine **execPageFault()** is responsible for determining the faulting context, faulting address, and the faulting page and loading this information in a fault descriptor.

The fault address is passed to this routine by the low-level trap handlers i.e., **VmHandler()**. The page address can be found by masking off the lower page offset bits from the faulting address. The faulting context can be determined by examining the address space in which the fault occurred. If the address space is SID0, then the KernelContext is responsible for generating the fault.

Discussion :

There are loopholes that still need to be plugged.

One serious problem is the PROBE instruction. This instruction presents a problem in the presence of copy-on-write pages. The non-access TLB miss routines can return the correct values if they are called, but if a page is in memory and marked copy-on-write then the PROBE instruction

that checks for write access will fail. The way the Tut group solved this problem was to track down all the occurences of PROBE instructions in the kernel and add another procedure call when PROBE fails. This call would check with the Mach portable layers and is similar to our execPageFault() routine when the nonAccess flag is set. Unfortunately, user programs which use the PROBE instruction are on their own. This issue is not addressed in the current port. It might be wise for future PA-RISC implementations to implement the PROBE instruction as a software trap which would allow user programs to receive the correct treatment of the instruction.

The presence of non-access TLB miss faults requires that certain additions be made to the page fault handler in the portable layers. Non-access TLB miss faults are not supposed to cause the faulting page to be brought into memory. Since the portable layer is called to resolve access rights in the case of non-access faults caused by PA-RISC *probe* instructions, it was necessary to make a few changes to the interface so that the faulting page was not swapped back into memory.

Rather than change the number of arguments passed to each procedure in the fault handling sequence, a field, `nonAccess`, was added to the fault descriptor, i.e.,the structure `gmiPullInArgs`. This modification results in not having to change the format of any procedure call in the portable layer. When the `nonAccess` field is non-zero, it indicates that the current fault should be handled as a non-access fault and the swapper should not be called.

4 System Call Interface

The system call interface deals with the code and control flow that occurs during the execution of a system call. The purpose of a system call is to gain higher privilege so that a user can execute privileged operations in a controlled fashion.

The usual method of making a system call on many architectures is to execute a trap instruction. Some amount of state gets stored, at which point the operating system recognizes the trap as a request for a system call. Then the user parameters get copied into kernel space and the system call routine is performed in privileged mode. The return values are then copied to user space and registers are appropriately set. Finally, a return to user mode occurs as with an exception.

The Chorus system call interface requirements and system call stub generation environment is detailed in section 4.1 and the implementation on PA-RISC is detailed in section 4.2.

4.1 Requirements

Chorus supports two types of actors: supervisor actors and user actors. Supervisor actors are privileged and live in the kernel space. Although Chorus can conceptually use the same system call interface for user and supervisor actors, it is expected that different system call interface is implemented for each type of actor. The rationale is that supervisor actors do not require the protection checks and the copying that is needed for user actors and thus can have a more streamlined interface. It is important to note that this is only an implementation decision and that the same interface can be used for both user and supervisor actors if so desired. Our goal was to implement both types of interfaces.

Chorus provides a general frame work for writing the stubs for supervisor and user system calls. The stubs are expected to be generated by the utilities **mk[s]lib()**. The file **lib/mklib.c** is expected to produce the executable **mkslib** for generating supervisor system call stubs when compiled with -DSUP_CALLS flag, or produce the executable **mklib** by default for generating user system call stubs. The executables **mklib** and **mkslib** have the following command line syntax.

mk[s]lib *system-call-name system-call-number*

The standard output for **mk[s]lib** is **stdout**.

All the compiled stubs for the supervisor actors are expected to be in **lib/chorusSv.a** and those for user actors in **lib/chorus.a**.

The kernel attaches the system call routines for user and supervisor actors by executing **scSystemInit()** and **scUserInit()** which, in turn, call **SupCallConnect()** with appropriate parameters. It is the responsibility of the *Supervisor* and the system call interface implementation to call the correct system call routine inside the kernel with the parameters for the system call given by the user or supervisor thread.

4.2 Implementation

In the case of PA-RISC, there are two mechanisms that can be used for implementing a system call interface for a user.

- By causing a trap in the system call stub. This method is similar to that described in the introduction of system call interface (section 4. An example for PA-RISC is to have a *break* instruction with an appropriate parameter value as the last instruction in the stub.
- By using the *gateway* mechanism. PA-RISC provides a *GATE* instruction [10] to perform a controlled transition from a lower privilege level to a higher privilege level. Pages can be mapped with special access control information and are called **gateway** pages. A gate instruction executed in these pages promotes the privilege level of the code that is executing. The privilege level obtained depends on the access control information for that page.

The gateway mechanism was chosen in PA-RISC for implementing the system call interface for user actors. The advantage of this mechanism over trap-based system calls is efficiency because no saving and restoring of full user state is necessary (as for any trap or interrupt) before it is realized that the trap is a deliberate mechanism to enter the kernel to perform privileged operations. **Tut** code uses the same mechanism for implementing system calls. The stub interface for user actors is detailed in section 4.2.1.

For most Chorus implementations, the supervisor actor system call stubs make a procedure call to the required system routine in the kernel. The address of the system routine was calculated from the starting address of the kernel's vector of system routines for supervisor actor system calls. This address is made available in the *Root structure* for all the supervisor actors and is set by the kernel during kernel initialization. We decided to adopt the same approach. The implementation of this interface turned out to be more complicated than most chorus implementations on other architectures and is detailed in section 4.2.2.

In our port, code executes at privilege level 0 (the highest privilege) or at privilege level 3 (lowest privilege). Levels 1 and 2 are not used. Code in supervisor actors and kernel executes at privilege level 0 and that of user actors at privilege level 3.

4.2.1 System call interface for user actors

The system call interface for user actors is implemented in **lib/mklib.c**, **kern/PARISC/locore.s**, **kern/PARISC/asm_scall.s**, and **kern/PARISC/svConnect.cxx**.

The gateway pages are mapped during the kernel initialization phase and are set up such that promotion to privilege level 0 occurs at the target address of the *gate* instruction.

A new gateway page has been defined exclusively for chorus system calls that mimics the HP-UX gateway page. It is physically contiguous with the HP-UX page but is mapped at virtual address

```
;;;;;;;;  begining  of stub ;;;;;;;;;;;;;
    .code
    .export threadCreate,code
threadCreate
    ldil L%0xc0006004,r1
    ldi 31,r21 ; Sub System No
    ble R%0xc0006004(sr7,r1)
    ldi 30,r22 ; Call No for  threadCreate
    bv,n r0(rp)
    nop
;;;;;;;;  end of stub ;;;;;;;;;;;;;
```

Figure 11: User Actor System call stub example

CHORUS_SYSCALLGATE defined in **include/PARISC/syscall.h**. This virtual address is six 4K pages greater than SYSCALLGATE, the virtual address of the HP-UX gateway page. The address assignment is based on the following constraints:

- The new gateway page address should not clash or overlap with HP-UX gateway pages.
- There should be sufficient room for growth in the virtual address space for more HP-UX gateway pages.
- The address should be in the fourth quadrant.

lib/mklib.c has been modified to produce the user and supervisor stubs for various system calls. The assembly language stub for the system call threadCreate is shown in figure 11 as an example.

The stubs for the user actor system calls are similar to the system call stubs for HP-UX except that one more temporary register has a dedicated use. In HP-UX , the system call number is loaded into gr22 (referred to as CN). In Chorus, HP-UX or unix-like operating systems are expected to be implemented as sub-systems on top of the micro-kernel. Therefore, the system call stub should specify the sub-system which should handle the system call. Hence, in PA-Chorus, the sub-system number is also passed (in register gr21) to the kernel. The HP-UX gateway page has been retained for the long term goal of maintaining HP-UX binary compatibility. The idea is that when a binary image makes a system call using a standard HP-UX stub and branches to the HP-UX gateway page, we just branch to the Chorus gateway page with the hpux subsystem number set. The system call can then be handled in the same way for all system calls, whether from Chorus or from other subsystems.

The control flow during a system call is as follows:

- A user actor makes a system call by executing the corresponding labeled stub in **lib/chorus.a**.
- The system call stub loads the sub-system number in gr21, the system call number in gr22, and performs an inter-space branch and link (*ble*) to the virtual address CHORUS_SYSCALLGATE. The user arguments to the system call are in grs 23-26 and/or on the user stack. The Chorus gateway page is implemented in **kern/PARISC/locore.s**.

- The code in the Chorus gateway page raises the privilege level to 0 and performs a vectored branch to a potentially different label based on the sub-system number. In the current implementation, a single label (**chorus_syscallinit**) is used for system call handling for all sub-system numbers. This is a hook in case system call handling needs to be performed differently based on the sub-system number. **chorus_syscallinit** is in **kern/PARISC/asm_scall.s**.

- At **chorus_syscallinit**, the following operations are performed:

  - Switch to the current thread's kernel stack to perform the system call. To achieve the switch, the system stack pointer is read from the `ksp` field of the current thread's `currCtx` structure.
  - Allocate a context frame on the kernel stack by incrementing the system stack pointer by **fr_size**. Mark that we are on the kernel stack by zeroing the `ksp` field in the context frame. The begining of context frame would be referred to as `ssp`.
  - Save thread specific registers SP, DP, GR31 (contains user stub return address), RP, SR4, in the and mark that this is the first frame on the system stack.
  - Mark that we are performing a system call:

    `ssp`→`state_flags` = TCB_INSYSCALL,
  - Pass the system call number in the context by setting a temporary register in `ssp`:

    `ssp`→TCB_RET1 = CN

    This system call number is used by **SupCallHandler()**.
  - Copy `arg0-arg3` into the context frame. These arguments are passed to the system routine by **SupCallHandler()**. Call **SupCallHandler()** (**kern/PARISC/svConnect.cxx**) with the pointer to saved context (ssp) and sub-system number.

- **SupCallHandler()** performs the following operations:

  - Get system call number from `ssp`:

    SysCallNo = `ssp`→TCB_RET1.
  - Check the `userTrapVect` table. Check if a handler is present.
  - If a handler is present then check if the number of arguments to the system call is greater than four. If true, then copy the extra arguments from the user stack using **svCopyIn()**. Otherwise do nothing. Call the handler with all the parameters. The decision to copy the extra user arguments into a temporary space before calling the handler is make use of the compiler in creating the stack frame to call the handler routine. Copying of parameters that are passed by reference from user space to kernel space is left to the individual system calls.
  - If a handler is not present, then call **KnHandler()**.

- Check if system call needs to perform a complete restore sequence. This check was needed in HP-UX to handling signals, It was retained since it could be a useful facility for sub-system managers. This check is done by checking `state_flags` field of `ssp` for the TCB_DORFI bit. If the bit is set, then perform a full restore sequence that is very similar to returning from an exception. Otherwise, the thread specific registers that have been stored at **chorus_syscallinit** are restored, the `ksp` field in the thread's `currCtx` is updated, and an external branch is performed to the user actor's return address, simultaneously lowering the privilege level to 3.

This completes the outline of User actor system call interface.

Discussion :

This implementation is another instance where **Tut** code was reused. The main file of reuse is **asm_scall.s**. We started from this file and modified to suit Chorus calling conventions and thread access.

4.2.2 System call interface for supervisor actors

In the case of system call interface for supervisor actors, the system call stub is similar to a procedure call in most Chorus implementations. This was possible as the requirement of Chorus that supervisor actors live in the kernel context was sufficient to perform this optimization.

In the case of PA-RISC, this was not quite the case. This is because of the same data pointer (**$global$**) problem mentioned in section 2.2.9. Since each image in the kernel space has its own datapointer (**$global$**), calling a routine directly in another executable is not possible even though all the supervisor actors share the same space id. To execute a procedure of the kernel from a supervisor actor, the processor's DP register should be set to that of the kernel's **$global$** before calling the kernel's procedure and restore the supervisor actor's **$global$** on return from the kernel procedure. This is exactly what the supervisor stub performs. An example of a supervisor system call stub generated by **lib/mkslib** is given in figure 12. The stub is for the system call **threadCreate()**.

The stub performs the following actions:

- Save the current DP and RP in the frame marker allocated by the calling conventions,
- Initialize the DP register by the Kernel's DP available in the *Root Structure* [12] which is mapped, privilege level 0 read-write, into quadrant 4.
- Get the address of the Supervisor system call table from the *Root Structure* and calculate the offset of the system call routine associated with the system call number.
- invoke the system routine
- Restore the supervisor actor's DP and RP from the frame marker and return to the supervisor actor code.

By branching directly to the system routine, the parameters to the system call can be directly reused by the system routine thus avoiding the copying of parameters on the system stack.

Discussion :

The system call table is an array of structures. The structure has 2 elements: a function pointer and the number of arguments. TABL_ELMT_SIZE and FUNC_OFFSET define the size of the structure and offset to the function pointer respectively. These definitions are fragile and must be automatically generated from the structure definitions.

```
;;;;;;;;  begining  of stub ;;;;;;;;;;;;;
    .code
    .export threadCreate,code
threadCreate
#define TABL_ELMT_SIZE 8
#define FUNC_OFFSET 4
    stw dp,-32(sp) ; fm_edp posn in the frame
    stw rp,-24(sp) ; fm_erp posn in the frame
    ldil L%0xd0000448,dp
    ldw R%0xd0000448(dp),dp
    ldil L%0xd0000024,r1
    ldw R%0xd0000024(r1),r1
    ldw 30 * TABL_ELMT_SIZE + FUNC_OFFSET(r1),r1
    blr r0,rp
    bv r0(r1)
    nop
    ldw -32(sp),dp ; fm_edp posn in the frame
    ldw -24(sp),rp ; fm_erp posn in the frame
    bv r0(rp)
    nop
#undef TABL_ELMT_SIZE
#undef FUNC_OFFSET
;;;;;;;;  end of stub ;;;;;;;;;;;;;
```

Figure 12: Supervisor Actor System call stub example

5 Mutex Interface

In Chorus, Semaphores and Mutexes are data structures that are defined by an actor in it's address space. The kernel is invoked for all semaphore operations. For mutex operations, the kernel is invoked only only when the threads have to be blocked behind a mutex or unblocked. In the ideal case, where the threads never attempt to enter a critical region while another thread is in its critical region, the kernel will never have to be invoked. Thus, mutexes provide an efficient means of synchronization at the cost of fairness. Semaphores, in contrast, guarantee fairness at the cost of efficiency.

In the case of supervisor actor, the semaphore or mutex data structure is directly accessed, whereas in the case of user actor, the semaphore or mutex data structure is copied into the kernel space (as in any user system call).

5.1 Requirements

The mutex interface consists of the following functions. All the functions take the address of mutex (`KnMutex *mutex`) as parameter. The specification is taken from the Chorus Programmer's manual[5].

mutexInit() : The mutex is initialized to *free*.

mutexTry() : Acquire a mutex. If mutex is free, then mutex is locked, returns value 1 and execution proceeds normally. If mutex is locked, then the call returns 0.

mutexGet() : Acquire a mutex. If mutex is free, then mutex is locked, returns value 1 and execution proceeds normally. If mutex is locked, block the thread until the mutex becomes free.

mutexRel() : Release a mutex. If threads are blocked on the mutex, one of them is awakened.

5.2 Implementation

The following atomic read-modify-write instructions are available on PA-RISC. These are basically load and clear instructions:

- *ldcwx*: Load and Clear Word Indexed
- *ldcws*: Load and Clear Word Short; Short refers to using a short displacement parameter rather than a short word.

Both instructions clear the location at the effective address and the previous contents of the memory location are loaded into the destination register. For details on the instructions, see the PA-RISC architecture manual[10]. Any one of the instructions can be used for mutex implementation. The *ldcwx* instruction was chosen for PA-Chorus.

The factors to be resolved in implementing mutexes were:

1. PA-RISC load and clear instructions require the effective address of the memory location to be aligned on a 16-byte boundary.
2. Since mutexes are declared in an actor's address space, there is no kernel control over what the alignment will be.
3. The mutex structure is a black box to the user or supervisor actor and all operations are performed through kernel exported functions.
4. There should not be any change in the system call interface.
5. The portable layers of the kernel involved in implementing the kernel part of the mutex operations assume that a value of 1 indicates **locked** and 0 indicates **unlocked**. In the case of PA-RISC it is not possible to have a value 1 for **locked** as the instructions provided are load and clear instructions.

The first alternative was to enter the kernel for every mutex system call. In the kernel, the system call routine would disable interrupts, perform the read and write operations in separate steps, re-enable interrupts and return. This alternative was rejected since it decreased the performance of mutex operations greatly. It made them as costly as semaphore operations which defeated the purpose of using mutexes.

The following alternative was adopted. The Mutex structure definition given in the chorus public interface file **include/chorus.h** has been modified. The structure *prior* to modification is given in figure 13.

The atomic operations for mutexes need to be performed on the `lock` field of the KnMutex/KnSem structure. The other members come into play only when the mutex is not free or there are threads to be released.

```
typedef struct {
    int         lock;       /* used by the mutex operations only */
    SemQueue*   threads;    /* pointer to blocked thread queue */
    int         count;      /* Semaphore Count value */
    unsigned    key;        /* Semaphore key */
} KnSem;

typedef KnSem KnMutex;
```

Figure 13: KnSem structure definition before modification

```
typedef struct {
    int         lock[4];    /* ARRAY used by the mutex operations only */
    SemQueue*   threads;    /* pointer to blocked thread queue */
    int         count;      /* Semaphore Count value */
    unsigned    key;        /* Semaphore key */
} KnSem;

typedef KnSem KnMutex;
```

Figure 14: KnSem structure definition after the modification

To guarantee a 16-byte aligned lock field, the KnMutex structure definition was modified as shown in figure 14.

Since `lock` is now 4 words or 16 bytes long, there will be one word that is 16-byte aligned among the 4 words allocated for `lock`.

The implementation of the various mutex functions is given below.

mutexInit() : The stub is generated by **/lib/mk[s]lib**. The corresponding system call routines **ScUsMutexInit()** and **ScMutexInit()** in **kern/scUser.cxx** and **kern/scSystem.cxx** have been modified to handle the changed data type of `lock` and the `lock` status values.

mutexTry() : This function is implemented in **lib/PARISC/mutex.c**. This function performs the following operations:

- Find the address of the aligned word in the `lock array`. This is done by the macro `alignLock`.
- Pass this address to the assembly routine **low_mutexTry()** in **lib/PARISC/chorusSync.s** and return the return value.

This function is entirely an user level library function. No kernel invocation is needed.

mutexGet() : This function implemented in terms of **mutexTry()** in **lib/PARISC/mutex.c**. If the mutex is available, then call returns immediately. Otherwise, kernel is invoked to block the thread. The blocking call returns when the mutex becomes free and the procedure is repeated again until the mutex becomes available. The call returns to the user only when the mutex is obtained by the stub.

mutexRel() : This function is implemented in **lib/PARISC/mutex.c**. This function writes MUTEX_UNLOCKED into the 16-byte aligned word of the `lock` array. If threads are waiting on this mutex, then the system call **k_mutexRel()** is invoked to release the threads. The function then returns to user.

The stubs for k_mutexGet and k_mutexRel are generated from **lib/mk[s]lib**. Modifications were made to **kern/scSystem.cxx** and **kern/scUser.cxx** to deal with the change of `lock` from an `int` to an `int array` and to address the 16-byte aligned word of the `lock` array. Note that in the case of the kernel routines to handler user-actor system calls there is an added factor to be considered. Since the user data structure is copied into a corresponding temporary kernel data structure, finding the aligned word in the kernel copy of the mutex may not be the same as that of the user mutex. Additional code was required to take care of the potential difference in alignment.

6 Modifications to the Chorus Portable Layers

Several small modifications were made to the "portable" layers of Chorus to carry out our port to PA-RISC. This section summarizes these modifications and outlines the reasons for making them:

- Calling the handlers attached by supervisor actors: Chorus requires the kernel and supervisor actors to share the system address space. The initial design that all supervisor actors and the kernel have the same space identity meant that each one of them had a separate **$global$** . This complicated the calling of attached handlers as Chorus assumed in its portable layers that these routines can be called directly ((*fn)(...)) as the supervisor actors are in the same system space. This cannot be done on PA-RISC (see section 2.2.9). This accounted for most of our modifications to the portable layers. See the Supervisor actor interface implementation section for a full list of the modifications and additional functions implemented.

  Of course, even if the kernel and supervisor actors were put in different spaces with different space Ids but with the same value for **$global$** (For example: all user actors on Chorus have their **$global$** equal to 0x40000000), we will have to deal with other problems of inter space linkages similar to a system call. In either case it entails some modification in the kernel and/or the Chorus interface.

- Stack direction: The portable layers of Chorus assume that stacks grows towards lower addresses. For example, the system stack bottom is taken as the end of the stack area, whereas on PA-RISC it is the begining of the stack area. This is actually a problem in the Chorus interface definition for threadCreate which takes stack bottom as parameter. The stack bottom would be different if the stack direction is different. A better definition would be to specify two parameters:

  - Address of the buffer allocated for the stack.
  - Size of the buffer

  On PA-RISC, the second parameter could be ignored. On architectures on which the stack grows towards lower addresses the size parameter could be used to locate the bottom of the stack.

- Stack initialization: In addition to finding the stack bottom, a frame must be allocated on the stack for PA-RISC in accordance with procedure calling conventions of the architecture.

For example, for user thread stack initialization, it is 48 bytes. There is one instance in **knMain.cxx** where Chorus does not submerge this function in the machine dependent layers.

- Stack allocation: The system stack has to be physically allocated during the thread initialization. There is one instance where the portable layers of Chorus assume that the size of the system stack is one physical page. In our port it was 4 logical pages. This could have been done in a portable manner by using the symbolic constants exported by the included machine dependent header files.

- 16-byte alignment constraint on addresses for atomic load and clear instructions: This required a modification in the mutex data structure definition in the Chorus public interface file. This was the only modification necessary to guarantee a 16-byte aligned address for mutexes. The user treats the mutex structure as a black box and this feature aided in maintaining exactly the same interface to the user, with very little degradation in space utilization and performance.

- Values for mutex **unlocked** and **locked** status: This is far more serious than the earlier problem. The code in the kernel implementing the mutex operations assumes that `lock = 1` implies a locked mutex and `lock=0` implied an unlocked mutex. With load and clear instructions, this assignment will not guarantee synchronization. A better approach would have been to import a machine dependent mutex header file and use the values exported by this header file in checking for mutex status. The list of modifications is specified in section 5.

- Non-access TLB miss faults: These faults do not require a page to be brought into memory. The portable layers need to know that they need not have to bring in the page. This resulted in some modification. See section 3.

The modifications that involved stack direction, allocation and initialization are as follows. All have been done under the compilation flag PARISC:

- **KnInit()** in **kern/knMain.cxx**. Corrected for stack direction and allocation.

- **ActorInit()** in **kern/knMain.cxx**. Corrected for stack initialization and direction.

- Member function **init()** of class **mThread**. Corrected for stack direction.

A qualitative evaluation of Chorus on HP PA-RISC is presented in[18].

7 Future Work

The first item that will receive the highest priority is the floating point and coprocessor emulation. Right now, any thread performing those instructions is aborted. This has to be rectified.

The next step is to enable the floating point coprocessor and handle the various exceptions. The Tut code should be useful in reaching these two short-term goals.

Some of the more interesting experiments we would like to do are:

- Faster context switches based on recognizing the thread's type and status. For example: A system thread uses no floating point coprocessor. This characteristic can be used in making a faster context switch. More generally, we would like to recognize the characteristics of the thread that can be utilized to provide efficient context switches. Using the compiler information about the thread is also an interesting possibility.

- Taking the supervisor actors out of the same space as the kernel and put them in separate spaces. Multiple privilege levels and protection ids can be used to control the access of these actors. Inter space calls are very cheap on PA-RISC. System call costs are marginally higher but approximately 3 orders of magnitude better than using a trap to implement system calls on PA-RISC. PA-RISC seems to be suitable for decoupling of functions because of the global virtual memory, multiple privilege levels, orthogonal protections, etc. It would be interesting to get some experimental results with various configurations and interfaces.

- Evaluate the use of 64 bit addresses and to determine where the operating system interfaces need to be broadened in the interests of globally addressable memory, efficiency and identification.

8 Acknowledgements

We thank Chorus for their sources and the valuable time spent in making us understand the machine dependent layers of Chorus. We especially thank Jean-Jacqués Germond, Frédérick Hermann, and Vadim Abrossimov for being very responsive and helpful.

Our sincere thanks to Bart Sears for providing us with the Tut sources and his visit which saved us a considerable amount of time. We thank Ahmed Ezzat for answering the 'help!!' questions at the time we didn't have the low-level documentation. We would also like to thank the other members of the Tut project who provided us with such high quality information.

Srikanth Kambhatla of OGI modified the original Tut **trap.c** file so that it could compile in our environment.

Finally, the project would not have reached this stage without helpful discussion and input with the other members of the PA-Chorus group.

References

[1] Vadim Abrossimov, Marc Rozier, and Michel Gien. Virtual Memory Management in Chorus. In *Proceedings of Progress in Distributed Operating Systems and Distributed Systems Management*. Springer Verlag, April 1989. Also published as technical report CS/TR-89-30.

[2] François Armand, Michel Gien, Frédéric Herrmann, and Marc Rozier. Revolution 89 or "Distributing UNIX Brings it Back to its Original Virtues". In *Proceedings of the Workshop on Experiences with Building Distributed and Multiprocessor Systems*, October 5-6 1989. Also published as technical report CS/TR-89-36.

[3] Chia Chao, Milon Mackey, and Bart Sears. Tut Threads Book. Technical Report HPL-DSD-90-23, Hewlett-Packard Laboratories, 1990.

[4] CHORUS Kernel v3.2 Implementation Guide. Technical Report CS/TR-90-5, Chorus Systèmes, 1990.

[5] CHORUS v3.3 Programmers Reference Manual. Technical Report CS/TR-90-59, Chorus Systèmes, 1990.

[6] Overview of the CHORUS Distributed Operating System. Technical Report CS/TR-90-25, Chorus Systèmes, 1990.

[7] CHORUS Kernel v3.3 Implementation Guide. Technical Report CS/TR-90-71, Chorus Systèmes, 1991.

[8] Ahmed Ezzat, Chia Chao, Milon Mackey, and Bart Sears. Tut VM Book. Technical Report HPL-DSD-89-32, Hewlett-Packard Laboratories, 1989.

[9] Jean-Jacques Germond. Specifications of the CHORUS/MiX Kernel v3.2 Test Suites. Technical Report CS/TR-90-27, Chorus Systèmes, 1990.

[10] Hewlett-Packard. *Precision Architecture and Instruction Set Reference Manual*, third edition, April 1989.

[11] Jon Inouye, Marion Hakanson, Ravindranath Konuru, and Jonathan Walpole. Porting Chorus to the PA-RISC: Virtual Memory Manager. Technical Report CSE-91-5, Oregon Graduate Institute, January 1992.

[12] Jon Inouye, Marion Hakanson, Ravindranath Konuru, and Jonathan Walpole. Porting Chorus to the PA-RISC: Booting. Technical Report CSE-91-4, Oregon Graduate Institute, 1992.

[13] David V. James, Stephen G. Burger, and Robert D. Odineal. Hewlett-Packard Precision Architecture: The Input/Output System. *Hewlett-Packard Journal*, 37(8):23–30, August 1986.

[14] Ravindranath Konuru, Marion Hakanson, Jon Inouye, and Jonathan Walpole. Porting Chorus to the PA-RISC: Building, Debugging, Testing and Validation. Technical Report CSE-92-7, Oregon Graduate Institute, January 1992.

[15] Ruby B. Lee. Precision Architecture. *IEEE Computer*, 22(1):78–91, January 1989.

[16] Michael J. Mahon, Ruby Bei-Loh Lee, Terrence C. Miller, Jerome C. Huck, and William R. Bryg. Hewlett-Packard Precision Architecture: The Processor. *Hewlett-Packard Journal*, 37(8):4–22, August 1986.

[17] Jonathan Walpole, Marion Hakanson, Jon Inouye, and Ravindranath Konuru. Porting Chorus to the PA-RISC: Project Overview. Technical Report CSE-92-3, Oregon Graduate Institute, 1992.

[18] Jonathan Walpole, Marion Hakanson, Jon Inouye, and Ravindranath Konuru. Porting Chorus to the PA-RISC: Overall Evaluation. Technical Report CSE-92-8, Oregon Graduate Institute, January 1992.