1-1992

Porting Chorus to the PA-RISC: Booting

Jon Inouye
Oregon Graduate Institute of Science & Technology

Ravi Konuru
Oregon Graduate Institute of Science & Technology

Jonathan Walpole
Oregon Graduate Institute of Science & Technology

Marion Hakanson
Oregon Graduate Institute of Science & Technology

Citation Details

"Porting Chorus to the PA-RISC: Booting," Jon Inouye, Marion Hakanson, Ravi Konuru and Jonathan Walpole, OGI Technical Report No. CS/E-92-004, January 1992.

Porting Chorus to the PA-RISC: Booting

Jon Inouye
Marion Hakanson
Ravindranath Konuru
Jonathan Walpole

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

January 1992

This document describes the steps involved in booting Chorus on the Hewlett-Packard Precision Architecture RISC (PA-RISC) workstation. The information contained in this paper will be of interest to people who:

- intend to boot an operating system on PA-RISC machines.

The reader is strongly encouraged to read the following PA-Chorus documents before reading this document:

- Technical Report CSE-92-3, *Porting Chorus to the PA-RISC: Project Overview*

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

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. This document describes the steps involved in booting Chorus on the Hewlett-Packard 9000/834 workstation, which uses the Precision Architecture (PA-RISC) processor.

We started out with the low level Tut (HP-UX 2.0) boot code. One of our goals was to reuse as much of this code as possible, which would reduce the amount of low level code we would have to debug. This was very important, especially since the HP 9000/834 has a very complex I/O architecture and we lacked any sophisticated debugging tools. Writing the PA-Chorus boot code involved modifying the Tut code to match the Chorus startup sequence. In the remainder of this section, we present an overview of the PA-RISC boot mechanisms and the Chorus startup sequence. Section 2 describes the Chorus utility used to build a boot image. Section 3 concentrates on the Chorus boot program and section 4 covers Chorus kernel initialization. Section 5 describes the PA-Chorus memory map and how it evolved from HP-UX 2.0 (Tut). The modifications to the Tut code are discussed in section 6. Finally, we evaluate our approach to booting Chorus on the PA-RISC in section 7.

1.1 PA-RISC Boot Sequence

Boot can be triggered by several mechanisms: power on, the reset button, software "reboot", and the access port. Figure 1 outlines the low-level PA-RISC boot sequence. These steps are generally followed by all operating systems running on PA-RISC platforms. (For more information on the Hewlett-Packard PA-RISC boot sequence, consult [1].)

The first utility to run is the *Processor Dependent Code (PDC)*. This firmware initializes the hardware and sets up the registers before loading the first software, the *Initial Program Loader (IPL)*. In our case (series 800), the IPL is the *Initial System Loader (ISL)*. The ISL allows a user to get the boot-up characteristics of the system, modify these characteristics, or load and execute a program. Once the ISL is finished, the boot image is loaded by a special loader. The loader is responsible for loading the boot image and executing it. Chorus assumes the existence of a loader on the target platform and we used the standard HP-UX loader.

We were able to use the PA-RISC boot sequence without any modifications. In addition to the boot hardware/firmware, the PA-RISC I/O architecture also provides a uniform interface to the processor and I/O modules. The *Processor Dependent Code* (PDC) procedures act as an interface to processor-dependent operations. The I/O Dependent Code (IODC) routines provide a mechanism to obtain module-type dependent information from an I/O module. We used IODC calls to implement very simple device drivers which enabled us to write to the console and read from the keyboard.

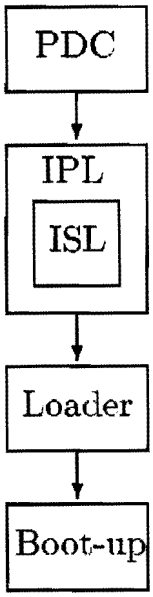

Figure 1: PA-RISC Boot-up Sequence

1.2 Chorus Startup Sequence

The Chorus startup sequence is composed of two components: boot and the kernel initialization. The boot component is responsible for setting up the machine specific environment that is relatively operating system independent. Kernel initialization deals with the initialization of both the machine-dependent and machine-independent sections of the Chorus operating system. Typically, ports of Chorus use two actors to implement these tasks. The boot actor runs the boot program and the kernel actor performs the kernel initialization. (For more information on the Chorus startup sequence consult [2].)

The Chorus boot and kernel initialization takes place in four major phases:

1. Creating the boot image
2. Loading the boot image
3. Running the boot program
4. Initializing the kernel

2 Creating the boot image

The utility program **btChorus** is used to create the boot archive. Its usage is a follows:

btChorus -l *load* -b *boot* -k *kernel* [-s *supActor*]* [-u *usrActor*]*

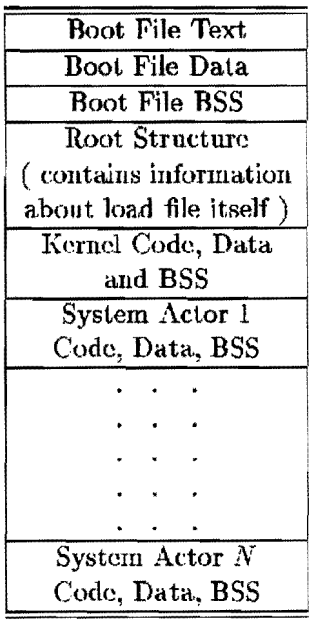

Figure 2: Structure of Generated *load* file

This utility is a UNIX program that should be easily ported to any UNIX system that is compliant with either the standard **a.out** or **COFF** format. Since HP-UX has its own binary format we needed to port this utility in addition to the boot code.

btChorus takes executable *boot*, *kernel*, supervisor *(supAct)*, and user *(usrAct)* actors and creates the boot archive, *load*, that looks like a single executable image to the loader. The archive is formed by extending the BSS[1] section of the *boot* actor to include the binary images of all the other actors. Thus, the first instruction that gets executed in *load* is the same as the first instruction that would get executed in *boot*. The structure of the executable file *load* is shown in figure 2.

2.1 Structure of the HP-UX Executable

The structure of an HP-UX executable file is defined by the Spectrum Object Module (SOM) format specification [3]. An **a.out** file obeying the SOM format has structure shown in figure 3. Except for the first header, all the headers need not occur in the given order.

2.1.1 Root Structure

The Root structure is defined in **include/HP800/load.h**. The Root structure consists of descriptors for the *load* image, i.e., the kernel and the actors that comprise the boot archive. This structure is used to pass information between **btChorus**, the boot program, and the Chorus kernel.

[1] Uninitialized data.

File Header
Exec header
Version String
Space dictionary
SubSpace dictionary
Initialization array
Loader fixup array
Space strings block
Symbol dictionary
Fixuprequest array
Symbol strings space
Compiler dictionary
Text (page aligned)
Data (page aligned) (Zero or Part of BSS in the final Data Page fragment)
Optional Unloadable Space

Figure 3: HP-UX a.out structure

Some of the structures and fields relevant to **btChorus** are shown in figure 4. The LOAD structure is a special data section within the ROOT structure that contains machine dependent information on all the actors in the boot image.

btChorus performs the following main steps in creating the boot archive. The actual names of headers have been omitted to give an overall picture.

1. Read the header of the *boot* file and update the bt_entry point in the load structure. Create the file *load* and copy the header of the *boot* into the *load* file. Copy the text and data segments of the *boot* into *load*. Explicitly allocate the BSS space of *boot* in *load*.

2. Allocate one physical page size space in the file for the Root/Load structure in *Load* and start copying the kernel and the boot actors.

3. Read the headers from *kernel* and update the kernel descriptor in the load structure. Copy the text and data segments from *kernel* into *load*. Allocate space equal to kernel BSS size (round to the nearest multiple of physical page size) in *load* and set the BSS size in the kernel descriptor to zero.

4. Repeat step 3 for each of the boot actors except for allocating BSS and changing the header. (The BSS sections of the boot actors are allocated during kernel initialization.)

5. Now seek to the beginning of the space in *load* which was reserved earlier for Root structure and copy the Root structure from memory into the file. Update the header of *Load* to indicate the data size of *load*. This would be equal to *boot* data + BSS + sizeof(Root Structure) +

```
typedef struct {
        long            v_addr;         /* Starting Virtual address */
          .
          .
        unsigned long   size;           /* Size of this section in bytes */
} desc_sect;

typedef struct {
        desc_sect       code;           /* code descriptor */
        desc_sect       data;           /* data descriptor */
        unsigned long   size_bss;       /* size of the BSS in bytes */
        long            pt_entry;       /* entry point in the actor's text segment */
} desc_bin;

typedef struct {
        char            name[16];
        desc_bin        act;
        long            cap[4];         /* Actors' capability */
        int             userActor;      /* 1=> user Actor, 0=>system Actor */
} desc_act; /* Actor descriptor */

typedef struct {
        desc_kern       kern;
        desc_act        act[10];        /* Maximum number of boot actors */
        short           nb_act;         /* Number of actors */
        long            bt_entry;       /* entry point of the load image */
        .
        .
        .
} KnLoad; /* LOAD structure */

typedef struct {
        .
        .
        .
        KnLoad Load;
} KnRoot; /* ROOT structure */
```

Figure 4: Root Structure

kernel size + boot actors' sizes. Set the BSS size field in the header of *load* to be zero since everything has been explicitly allocated in *load*. The file is now ready to be executed.

The standard HP-UX loader loads the bootable image into memory and transfers control to the entry point of the *load*. This is the code of the original *boot* program. This code knows that the Root structure starts at the first page aligned address after its data+BSS. It accesses the Load Structure for the kernel descriptor finds the entry point and transfers control.

3 The Boot Program

The boot program is responsible for initializing the environment so the kernel initialization can take place. It should:

- Set up a stack
- Install the kernel and system actors at specific locations in the target virtual memory system.
- Enable virtual addressing.
- Fill in certain sections of the Chorus LOAD structure.
- Transfer control to the kernel initialization procedure.

From the Chorus point of view, the major tasks of the boot program are to set up the stack (so that you can call C routines) and to initialize the machine dependent memory structures (to turn on virtual address translation). Normally, the boot program resides in a separate executable file. The Chorus startup sequence was designed to be separated into two actors.[2] The boot actor's primary task is to run the boot program, i.e., set up a stack to run C code and enabled virtual address translation. Unfortunately, the HP-UX startup code makes no distinction between boot program tasks and kernel initialization tasks. Code that is used to enable address translation also initializes several kernel variables. This is a problem because if the boot and kernel functionality is placed in separate actors they cannot directly address each others data![3] Rather than pass data through the LOAD structure, we decided to leave the boot program in the kernel. We then built a dummy boot actor whose only task was to jump into the kernel. Our kernel actor code now executes both the boot program and the kernel initialization. While this method may have wasted some physical memory (since Chorus normally deallocates the pages used by the boot actor after boot has completed) it saved us from making extensive modifications to the HP-UX code.

Before any C procedures can be called, the stack and certain processor registers must be set up. This task is described in the supervisor document [8].

Before virtual address translation can be enabled, the page tables and processor context must be initialized. Current PA-RISC implementations require translation look-aside buffer (TLB) misses

[2]This allowed the space used by the boot actor to be reclaimed during kernel initialization. It also allows the kernel to be recompiled without recompiling the boot actor.

[3]Using 32-bit pointers.

to be handled by software. Because of this, all exception handling code for TLB misses must be set up before address translation is enabled.

The two tables used to handle TLB misses are the hash table (HT) and the physical page directory (PDIR). These are the tables that are searched to determine whether a particular virtual address mapping can be satisfied without a page fault, i.e., they are used during TLB miss handling [4]. Before address translation is enabled, the page table entries corresponding to the boot actors (boot, kernel, systemActor1, etc.) should be initialized to the correct translations and protections.

Since the boot program often moves system actors around, it must specify the physical address location of each system actor before branching into the kernel. It performs this task by writing values into the LOAD structure which are then read by the kernel initialization program.

4 Kernel Initialization

The kernel entry point is specified in the LOAD structure and is the point to which the boot program transfers control. The kernel initialization process is responsible for the following tasks:

- Initializing the supervisor (SV) section.
- Initializing the memory management unit (MMU) section.
- Initializing the low level device drivers
- Calling **KnInit()** (Initializing the portable section.)
- Branching to **KnMain()**.

Of these tasks, the most work went into the supervisor and memory management initialization.

4.1 SV Initialization

The supervisor is responsible for initializing the machine dependent data structures, e.g., processor registers, interrupt vectors, etc. The supervisor must also call the C++ constructors for static objects and set up the dummy thread structure. These tasks are described in the supervisor document[8].

4.2 MMU Initialization

The Chorus MMU initialization deals with the construction of the machine dependent virtual memory data structures. After this initialization is performed, the MMU level should be able to provide the MMU interface described in [7]. Chorus MMU initialization has three major tasks:

- Create the kernel context.

- Initialize the pool of MMU pages.
- Fill in certain fields of the LOAD structure.

The kernel context, referred to by the global pointer **KernelContext**, is the Chorus object representing the address space used by the kernel and all system actors.

The pool of MMU pages contains **mmuPage** objects. Each object represents a physical page of memory. When mmuPage objects are created and destroyed they are allocated and returned to this pool, called the **pagePool**. This pool has functionality similar to that of the Unix BSD *core map*. Once the pool has been created, the **mmuPage** objects describing the kernel and system actors should be initialized and removed from the pool since those physical pages are already in use.

The LOAD structure is initially filled in by **btChorus** and later updated by the boot program. During the kernel initialization phase certain fields are modified by the machine dependent section which are later used by the portable section of the kernel initialization. These fields are described in section 6.4.

5 Memory Map

This section describes the memory map of the PA-Chorus kernel and how it evolved from HP-UX and Chorus. We tried to keep close to the HP-UX memory map to reuse as much Tut code as possible.

Figure 5 diagrams the kernel memory layout used by HP-UX 2.0.[4] A section is *equivalently mapped* if its virtual address and its physical address are identical i.e., the lower 32 bits are identical and the space identifier is equal to zero. Any section that is addressed using both physical and virtual addresses should be equivalently mapped to avoid losing cache consistency.

Chorus places no restrictions on the ordering or the placement of the boot, kernel, and system actors other than the fact that they must be paged aligned. The **ROOT structure** is page aligned and located at a fixed virtual address (referred to by the symbolic name ChorusContext) so that all actors can access it.

The PA-Chorus boot image is shown in figure 6 and the kernel memory map is displayed in figure 7. Since we reused the HP-UX boot code, the lower portion of the kernel memory map is identical to that of HP-UX. The boot text and data are a single page apiece since the boot actor's only purpose is to branch into the kernel.

While HP-UX uses two procedures to set up the kernel memory map PA-Chorus extends the functionality of **realmain()** (see section 6.3.1) to perform the memory mapping task. The UNIX related data structures in HP-UX have been encapsulated in the data sections of the Chorus/MiX system actors and are not seen by the kernel.

[4]The format of these tables was based on those used in the Tut documentation.

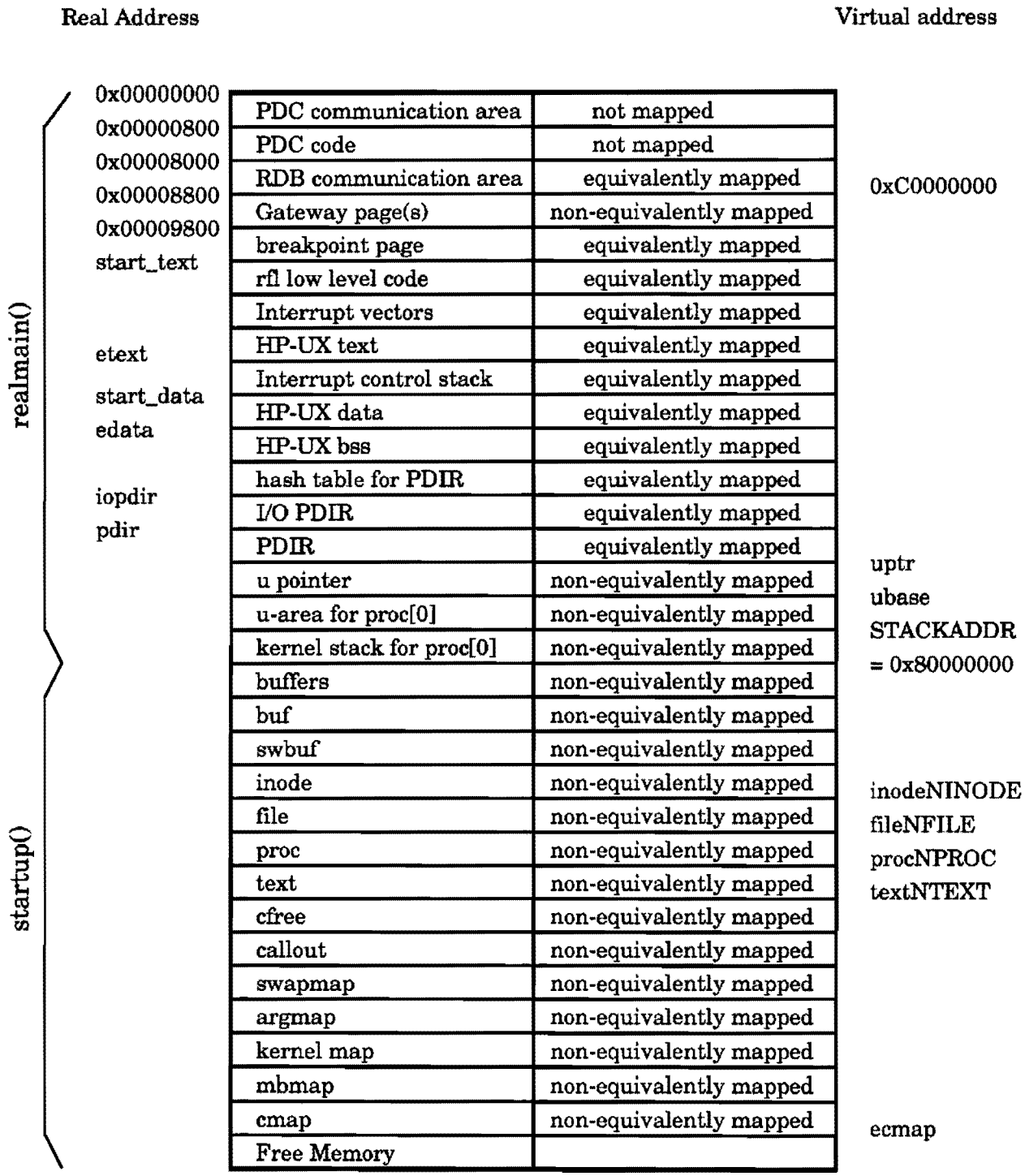

Figure 5: HP-UX Kernel Memory Layout

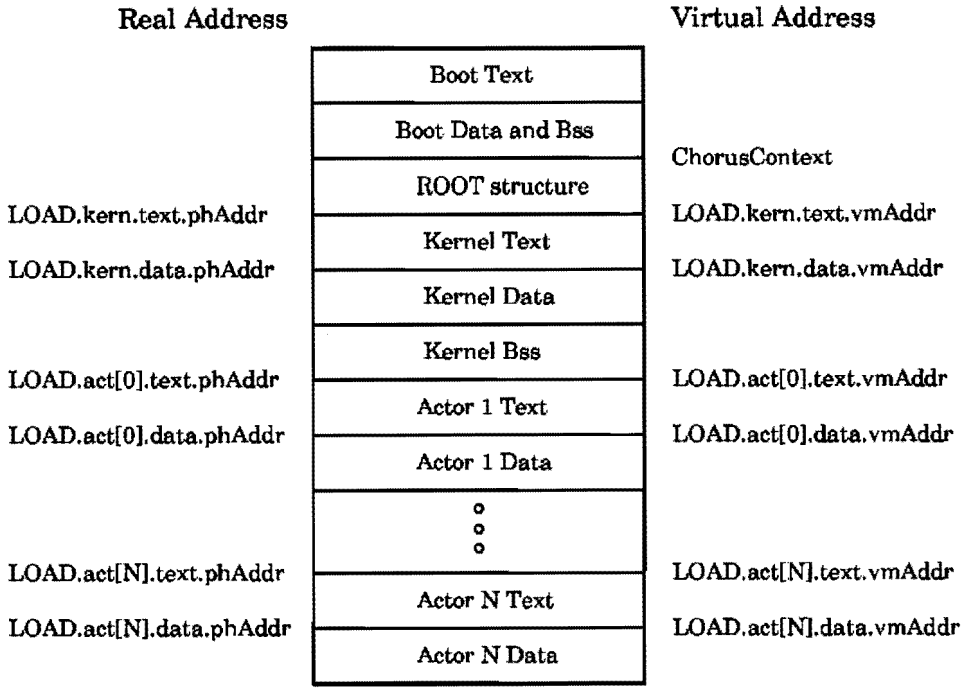

Figure 6: PA-Chorus Boot Image

6 Implementation

This section discusses the files and routines that represent the tasks performed in the Chorus startup sequence.

6.1 btChorus.c

This file implements the **btChorus** utility. We modified this file to comply with the HP SOM format.

In order to reuse as much low-level Tut code as possible we made a major change in the division of work between the *boot* and *kernel* relative to other ports of Chorus. In other Chorus ports, the *boot* code establishes the virtual memory mapping and turns on virtual memory before entering the kernel code. In our case, *boot* is a stub that transfers control to the kernel which does all the virtual memory initialization.

The syntax of command line arguments for **btChorus** have been changed. The earlier syntax was:

btChorus -l *load* **-b** *boot* **-k** *kernel* [**-s Actor+**]

On the PA-RISC, this syntax does not allow supervisor and user actors to be distinguishable. The

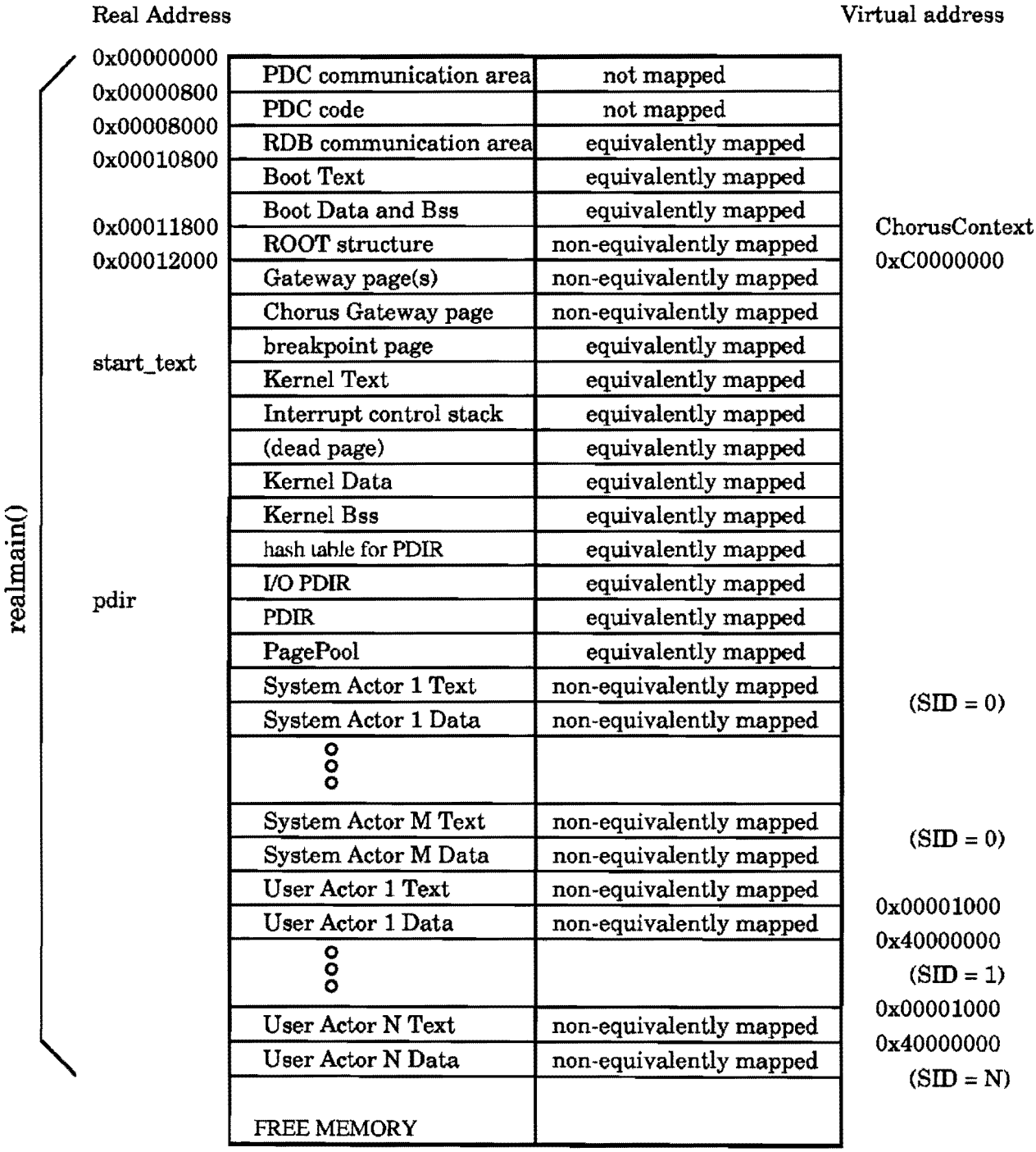

Figure 7: PA-Chorus Kernel Memory Layout

reason for the change is that it is not possible to distinguish a supervisor actor and a user actor based on their starting address since PA-RISC uses a segmented architecture.

Another change is the allocation of BSS for the kernel. Chorus does not allocate BSS for kernel and lets the *boot* code do all the allocation by moving the code to proper addresses in memory. Since our *boot* program was actually in the kernel, if we didn't allocate kernel BSS in **btChorus** then a system actor would be residing in the kernel's BSS space. This meant we couldn't use any uninitialized variables until we moved the system actors to a new location. Rather than worry about this, we decided it would be simpler to allocate the kernel's BSS area in the boot archive. This resulted in our boot archive being larger than necessary.

6.2 locore.s

This file contains the majority of the low level assembly language routines that make up the boot program (but are placed in the kernel actor). These routines set up the initial processor context, initial system stack, interrupt vectors, and turn on virtual memory. Modifications made to this file are described in the supervisor document [8].

6.3 vm_machdep.c

The routines in this file are responsible for setting up the machine dependent data structures so that address translation can be enabled. We started out with the original Tut file vm_machdep.c [1] which was used during the port of Mach 2.0 to the HP 9000 Series 800 workstation. We only made modifications to the procedure **realmain()**.

6.3.1 realmain()

The routine that is responsible for setting up the initial memory structures and mappings is called **realmain()**. The comprehensive list of the tasks performed by this routine is given in [6]. The following is a brief list of the tasks performed by **realmain()**.

- Initialize memory globals
- Scan the I/O system and set up I/O space.
- Patch the TLB miss handling routines
- Initialize hash table (HT) and physical page directory (PDIR).
- Map initial kernel address space and system actors into HT and PDIR.

There are several global variables that are set by **realmain()**. Several of these are obtained from locations in **Page Zero** [5], e.g., the size of physical memory. The PDC_CACHE call is performed

to obtain the various characteristics of the cache such as its size, block size, associativity, etc. These parameters are used to optimize the performance of cache flush/purge operations.

To determine the size and location of the data and BSS sections, **realmain()** makes use of loader flags such as **edata** and **end**. This creates a problem when the kernel is not loaded as the boot image. Because PA-Chorus uses a boot archive, the kernel is simply part of the BSS section of the boot actor and the loader flags point to the boot actor's locations instead of the kernel's locations. All reference to these flags were replaced by references to the global LOAD structure.

We made several minor code alterations to both **btChorus** and **realmain()** to adjust to the Chorus memory map (shown in figure 7). These alterations dealt with the additional task of mapping the actors within the boot image into the virtual address space. We allocated the kernel BSS (uninitialized data) within the boot image so we did not need to allocate it in **realmain()**. Once the amount of memory required for the system tables (hash table, physical page directory, and page pool) is calculated, the system actors are moved to the first page following the system tables. The BSS sections for the actors are allocated after kernel initialization is completed.

realmain() also initializes the hash table and PDIR entries for the initial memory map.

6.3.2 module_configure()

realmain() calls the routine **module_configure()** which is responsible for scanning the I/O system and setting up the I/O memory space. It needs to find out what busses are available, and what modules are available on those busses [5]. It allocates a portion of the I/O address space for each bus, and sets up the hard physical address space (HPA) for each module on a bus. We made no modifications to this procedure.

6.4 mmu.cxx

This file contains the routine **mmuInit()** which performs the initialization of the kernel virtual memory structures. It must perform the following tasks:

- Create the KernelContext.
- Create the pool of free mmuPage descriptors.
- Initialize all mmuPage descriptors mapping the boot actors and remove all descriptors mapping the kernel and boot actors from the pool.
- Fill in the fields of the LOAD structure.

The KernelContext is created by calling the method **new** for class **context**. Normally, this involves the creation of a page table for the address space. Since we chose to adopt the HP-UX PDIR structure, all contexts share a single inverted page table which simplifies context creation.

The class **PagePool** (described in detail in section 6.4.1) consists of a pool of **mmuPage** descriptors, where each mmuPage descriptor represents a physical page. The mmuPage descriptors representing the physical pages being used by the system actors must be initialized. Those descriptors representing the pages being used by the kernel do not need to be initialized since they will never be swapped out.

The **firstPage** fields (two for each boot actor[5]) in the LOAD structure should be updated. During boot, this field contains the physical page number of the first page used by each boot actor's code section.[6] After MMU initialization, the **firstPage** field should reference the mmuPage descriptor representing that physical page instead of the page number. The MMU initialization should also set the context start and end addresses for both user and supervisor actors (**ctxUsrStartAddr, ctxUsrEndAddr, ctxSvStartAddr, and ctxSvEndAddr**). These addresses specify the range in which regions may be created.

6.4.1 PagePool

The class **PagePool** is responsible for the allocation and deallocation of mmuPage descriptors. Each mmuPage descriptor represents a physical page of memory. There is one instance of class PagePool (called **pagePool**) that is created in the static constructor phase of the kernel initialization. The pagePool contains a reference to a contiguous block of memory which can be conceptually thought of as a pool of mmuPage descriptors. There is a unique mmuPage descriptor associated with each physical page in memory. The block of memory for the pagePool is allocated in **realmain()** and its location and size (in elements) is stored in the LOAD structure. The PagePool constructor then accesses the information in the LOAD structure to initialize itself. The pagePool is used by the mmuPage C++ methods **new** and **delete** to allocate and deallocate mmuPage descriptors.

7 Evaluation

Chorus' separation of the boot and kernel initialization components is an interesting new concept. However, we didn't believe it had enough advantages to modify the Tut code to separate the boot and kernel initialization tasks.

The most interesting thing about the boot sequence is Chorus' method of creating a bootable image from multiple executable files. We like this modular aspect of Chorus which allows you to boot up different system actors (servers) without having to recompile the kernel.

While most Chorus system actors are able to run outside kernel space, they are usually placed in kernel space for performance reasons. The majority of these performance reasons are related to the cost of entering and exiting the kernel. On other architectures, entering the kernel may require a trap and a change an address space, i.e., switching page tables, flushing the cache. Using

[5]In this instance boot actor refers to one of the actors in the boot image other than then the boot and kernel actors.

[6]There is also a field for the data section.

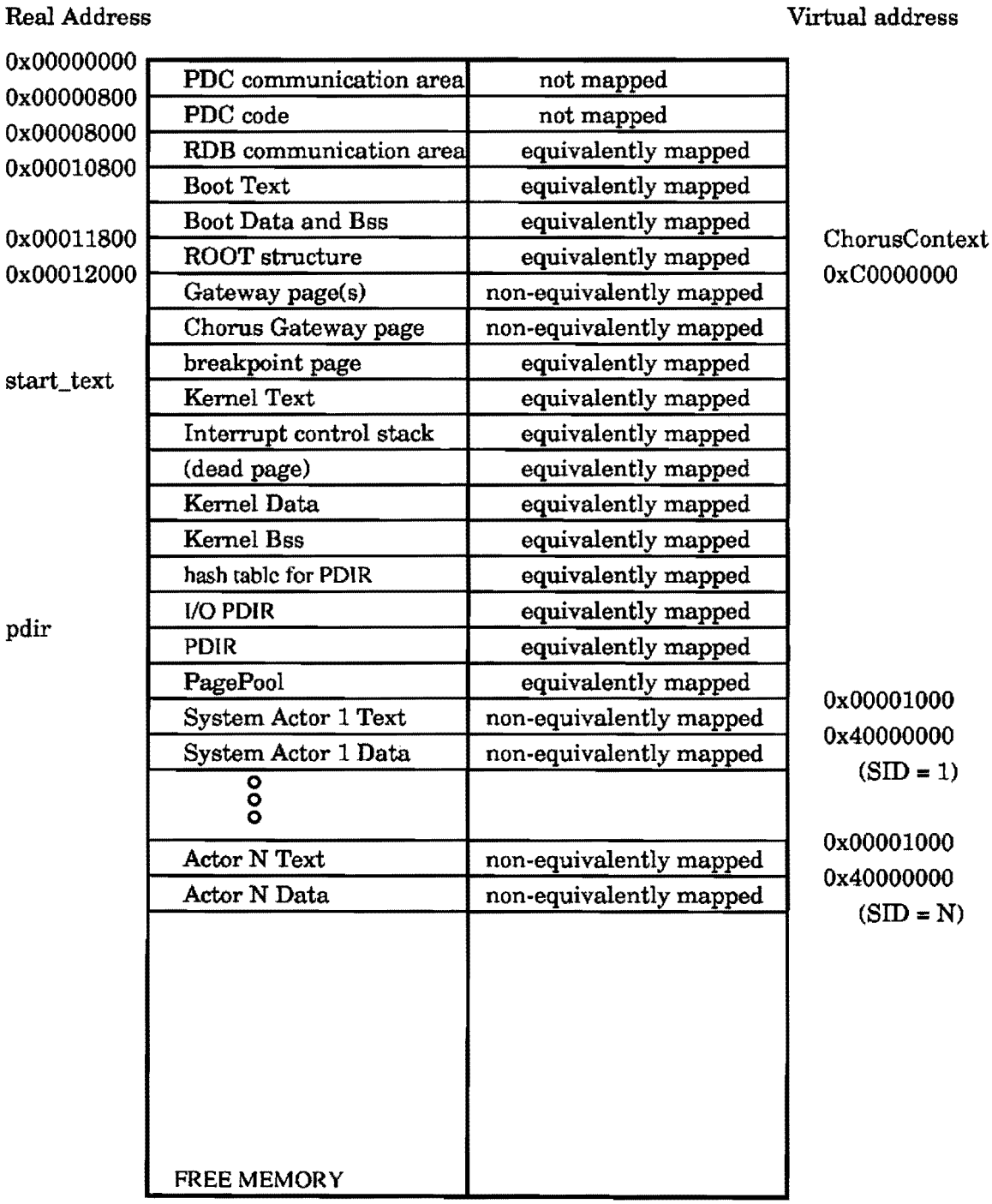

Figure 8: Future PA-Chorus Kernel Memory Layout

the gateway and global address space, the PA-RISC allows kernel entry with a cost slightly more expensive than a procedure call.

A disadvantage of having system actors reside in the kernel address is that they require an address range to be reserved for them. Since no two actors may overlap in the system address space it can be bothersome to manage the address space as the number of possible system actors grow. There is also the problem of managing different data pointers within the same 32-bit address space. This problem is described in the Supervisor document [8].

It would be nice to have the ability to place each actor (even system actors) into a separate 2^{32} byte segment by using a different space identifier for each actor. While user actors would run at the lowest privilege level, system actors could run at the same privilege level as the kernel but in a different protection domain. This would allow the kernel's segment to be protected from all actors and remove the tedious partitioning of the kernel's 32-bit address space. Figure 8 shows a possible memory layout where each actor would reside in its own 32-bit address space. It would still be possible for system actors to share the kernel's 32-bit address space but it would no longer be the *default* solution.

Another aspect of Chorus that we were not able to evaluate was the placement of device initialization code in the device manager (DM), a system actor. The Tut code initializes devices in the kernel. Since memory modules need to be initialized before use and the I/O address space is dependent on the number of I/O modules, certain device initialization needs to be performed in the kernel. Should we ever attempt to port the DM, it will be interesting to see how appropriate the code is to the PA-RISC.

8 Future Work

There is a considerable amount of work to do to separate boot code from kernel initialization code. We didn't feel there were enough advantages in it to perform this task.

A more interesting task would be placing system actors into separate (32-bit) address spaces. This may help create a more modular implementation without sacrificing performance.

9 Acknowledgements

Many thanks to Jean-Jacques Germond who gave us detailed information on the Chorus boot image and to Vadim Abrossimov who helped clear up some misconceptions about the responsibilities of the MMU initialization routine. Bart Sears suggested the partitioning of system actors into separate address spaces and outlined the advantages.

References

[1] Chia Chao. Initializing the Tut Kernel. Technical Report HPL-DSD-88-39, Hewlett-Packard Laboratories, 1988.

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

[3] Hewlett-Packard. *Spectrum Object File Format*, October 22, 1985.

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

[5] Hewlett-Packard. *Precision I/O Architecture Reference Specification*, 0.93 edition, January 1990.

[6] Hewlett-Packard. *Snakes HP-UX/Spectrum Kernel Initialization and Machine Interface Internal Maintenance Specifications*, August 1990.

[7] Jon Inouye, Marion Hakanson, Ravindranath Konuru, and Jonathan Walpole. Porting Chorus to the PA-RISC: Virtual Memory Manager. Technical Report CSE-92-5, Oregon Graduate Institute, January 1992.

[8] Ravindranath Konuru, Marion Hakanson, Jon Inouye, and Jonathan Walpole. Porting the Chorus Supervisor and Related Low-Level Functions to the PA-RISC. Technical Report CSE-92-6, Oregon Graduate Institute, January 1992.