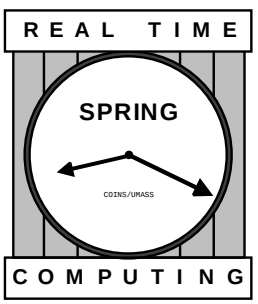
Address Space and Resource Management in the Spring System

Douglas Niehaus, Chung-Huei Kuan, Decao Mao

Spring Project
Dept. of Computer and Information Science
University of Massachusetts Amherst, Ma 01002

June 1992

Draft



Spring Project Documentation

Project Directors

Prof. K. Ramamritham

Prof. J. A. Stankovic

Contents

1 Introduction **1**

2 Hardware Overview **2**

3 Predictable Address Space Mapping **4**
3.1 Hardware Details 4
3.1.1 Translation Control Register 5
3.1.2 Root Pointer 8
3.1.3 Address Translation Cache 10
3.2 Address Mapping 12
3.2.1 Memory Map Structure 14
3.3 Memory Map Overhead 16
3.4 Memory Map Initialization 18
3.5 Memory Map Use 19
3.6 ATC Management 21

4 Resource Management **23**
4.1 Physical Page Pool 23
4.2 Logical Page Pool 24
4.3 Homogeneous Object Pools 26
4.4 Heterogeneous Object Pool 27

5 Address Space Creation and Manipulation **28**
5.1 Spring Computation Model 28
5.2 Shared Segments 29
5.2.1 Attaching Shared Segments 31
5.2.2 Shared Segment Description 33
5.2.3 Predefining and Compiling Segments 34
5.2.4 Current Status and Future Development 36

6 Address Space Manipulations at Boot Time **37**

7 Shared Segment Examples **38**

1 Introduction

The Spring kernel is intended as a platform for real-time system research which supports *predictable* program execution. This does not mean that *every* aspect of a program's execution must be predictable, but it does require a great deal of information that is not normally of concern to conventional systems. This information, which includes worst case execution time (WCET) and the program's resource use, is used by the system when scheduling program execution. Note also that since program behavior depends in many ways on the behavior of the system, that many aspects of the system's behavior must also be predictable.

As with any operating system, managing the many types of system resources is an important issue in Spring, but the requirement that system behavior be predictable has led to several design decisions that are significantly different than those for conventional systems. The system design has also been significantly constrained by the properties of the target hardware, which was clearly designed to provide good average case performance and not predictability of program behavior.

The need to predict a wide range of program and system behaviors affects every aspect of the system design, but does not eliminate the need for a reasonable, and reasonably powerful, programming environment. A computation under Spring is described as a single process, or as a group of interacting processes. Each process is a single thread of control executing within an address space. Processes can interact, among other ways, by sharing segments of their address spaces. These shared segments can be created and managed as independent objects during program compilation and execution. A process can *attach* a segment to its address space in one of several ways. All system processes, and application processes running in system mode while executing system calls, share a common *system* address space.

The inclusion of logical address spaces in the programming model is not *required* to support predictable execution of real-time programs. Indeed, supporting address translation in a predictable way presents some non-trivial challenges. However, it is a familiar and powerful aspect of many programming environments, and is thus desirable as long as it does not compromise predictability. A Spring node contains several process or boards, each with an MMU which supports address mapping fopr the processes running on it. The structure of the memory maps and how the MMU uses them have been carefully coordinated to make the behavior of the application programs in particular, and system behavior in general, as predictable as possible. Significant restrictions on other aspects of the system, particularly on the maximum size of both the system and application spaces result from measures required to support predictable behavior, but we consider them worth the advantages in flexibility and protection provided by logical address space use.

In addition to the support of logical address spaces, the system must also create and manage a number of resource pools. These include the physical pages of each processor in the system, the pages of the logical space for each process in the system, the pool of process control blocks, and that of memory maps. While basic resource management is not a particularly challenging topic, the four types of resource management used in the Spring kernel deserve some discussion for two reasons. First, the particular resources mentioned are fundamental to the support of processes executing on the system. Second, the four types of management methods should be sufficient to manage *all* resources in the kernel, and thus serve as examples for the management of types of resource types not required.

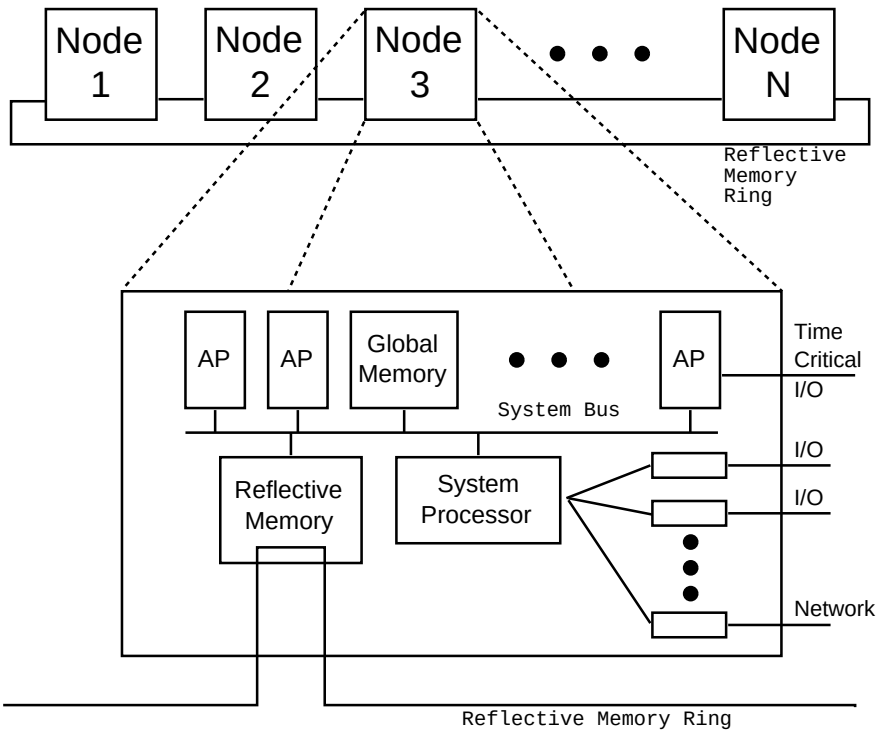


Figure 1: Spring Network and Node Architecture

The rest of this document discusses several aspects of the support and use of memory mapping and resource management in the Spring system. Section 2 briefly discusses the several properties of the target hardware that have a significant effect on the issues discussed in this paper. Section 3 discusses how predictable address space mapping has been implemented in the Spring system. Section 4 discusses the support for resource management in the system.

Section 5 discusses the Spring programming model, and how shared segments are used to manipulate the address space of a process. Section 6 discusses how shared segments are used at boot time to establish the environment required by the system code on each processor, and how a set of persistent segments is created for use by application processes. Finally, Section 7 esplains some examples of how shared segments can be used by applciation processes.

2 Hardware Overview

The Spring kernel is targeted to a multiprocessor node containing several single board microcomputers connected by a bus giving global access to the memory on each processor board. Each node also contains a special *reflective memory* board, whose contents are constantly updated to reflect the changes made by all the nodes on the ring. The reflective memory is thus effectively a single area of memory shared among all the nodes on the ring, and can be used as a medium for communication across node boundaries. Figure 1 illustrates both the network and node architecture for the current version of Spring. In the figure, the

rectangles marked "AP" denote processor boards devoted to running application programs, while the "system" processor supervises the activities of the node.

In the current configuration, each board within a node has its own M68020 CPU, M68851 memory management unit, and a bank of dual ported memory which can be accessed by the CPU on the board using a local bus, or globally using the node's VME backplane. It is also important to note that a local memory access takes significantly less time than a global access. This NUMA property has important implications for both predictability and performance, and implies that we must be very careful how the VME bus is used.

Memory access using the bus is problematic for two reasons affecting the predictability of worst case execution time (WCET) of a process. First, we must assume that all other users of the bus are also contending for it, thus increasing the WCET of the process using the bus. The use of round-robin mode for the VME bus ensures fair treatment of all global requests, and thus imposes an upper bound on the time of a global access.

The second problem with global access is that delays can arise from contention over access to a given memory location. Consider what can happen when a segment of memory on a processor board is shared by a computation on that board and one on another processor board. The computation on the remote board can access the shared segment across the VME bus while an unrelated process is executing on the board containing the shared segment. A local access which is initiated after a global access from the bus begins can be delayed, *victimizing* the unrelated computation which happens to be executing, thus making it more difficult to predict WCET.

We could assume that *all* local memory accesses are delayed in this way, and thus significantly increase the predicted WCET, or we could assume that this kind of victimization would be comparatively rare, and ignore it while accepting a slight decrease in the accuracy of the WCET prediction. A third possibility is to have a separate memory board to which we assign all shared segments. We choose the second approach in some situations, and the third in others. For example, we might choose the second approach in situations where we can assign all processes using the shared segment to the same board, thus eliminating the possibility of victimization, or when the delay caused by victimization is very small. We would choose to use the separate memory board when the victimization delay is significant, thus imposing a global delay on *all* accesses to the shared segment. One advantage of the second approach is that the WCET of the processes using the shared segment is lower for the processes executing on the board where the shared segment resides, since accesses to its memory are local.

This discussion illustrates that many aspects of the system can interact with one another. The placement of processes on processor boards can affect the predicted WCET since it affects memory access time. Conversely, constraints on the WCET of a process might well create constraints on its placement, the placement of any shared segments it uses, and that of other processes with which it may interact directly or indirectly. The use of logical address spaces for the system and processes executing under it provides a way to support program execution, while preserving flexibility in process and shared segment placement, and also avoiding any undo complication or constraint on program compilation. Logical address spaces and shared segments also provide a powerful and reasonably convenient way to structure the Spring operating system, which must coordinate activities occurring on the several processor boards within a node. The rest of this document will first discuss how

address spaces are managed predictably, and then how address space manipulations play an important role in the implementation of Spring.

3 Predictable Address Space Mapping

While the combination of the M68020 processor (CPU) and the M68851 page memory management unit (MMU) provides a powerful and flexible set of capabilities, finding a way to use them which will support predictable behavior requires careful consideration and coordinated selection of a large number of parameters affecting the hardware's behavior. This task is made particularly difficult since the hardware was designed for conventional systems where excellent average case performance was the goal. The need for Spring to provide predcitable logical address space management leads, in many cases, to decisions that are different than those normally used for conventional systems. In this section we first discuss some details of the target hardware, particularly those of the MMU, and then show how address spaces are managed predictably under Spring.

3.1 Hardware Details

In this section we give a brief explanation of the memory management features provided by the CPU and MMU, so that the design and implementation decisions described in later sections can be more easily understood. The CPU has a 32 bit address bus, enabling it to access a 2^{32} byte address space. Furthermore, the processor produces a 3 bit "function code" during each read/write cycle, classifying the memory access into one of eight categories, and thus effectively creating eight separate address spaces. The function code can thus be thought of as extending the M68020 addressing capability to 2^{35} bytes, with the restriction that the value of the 3 bit function code is determined automatically by the processor execution mode (user or supervisor) and the current micro-operation (e.g., fetch instruction or read/write data), rather than being specified by programmer. Among the eight possible spaces, only four spaces are of concern to system programmers. These are the data and instruction spaces in both the user and supervisor execution modes. For more detailed explanation of the function codes, please consult the CPU [1] and MMU [2] manuals.

As the name "Page Memory Management Unit" suggests, the M68851 manages the memory using a paged model, rather than in a segment model. This is not entirely desirable, given the needs of the Spring system, but is a restriction we must deal with until we adopt another target architecture. Inside the M68851 MMU are several general control/status registers, and an Address Translation Cache (ATC) which gives immediate response for a limited number of logical to physical mappings.

Given a logical address reference issued by the processor, together with a function code, the MMU can translate the logical address into a unique physical address which is used to access the physical memory. If the MMU is *disabled*, the logical address is passed unchanged to the memory address bus. If the MMU is enabled, it translates the address using a data structure which specifies the mapping from the logical address space of the process to the physical address space of the board. The details of the translation, and thus the structure of the map, are determined by the combination of translation parameters specified in several

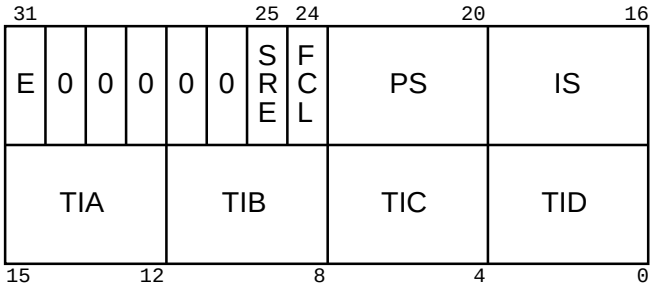


Figure 2: Translation Control Register

control registers of the MMU. The most important of these are the translation control (TC) and root pointer (RP) registers. The rest of this section discusses these registers and the ATC.

3.1.1 Translation Control Register

The TC register plays a key role in specifying MMU behavior. It specifies the format of the data structure specifying the mapping between logical and physical space, and precisely how the mapping is performed. The register is described on page 6-4 in the MMU manual[2], but a short explanation of its elements is appropriate here. Figure 2 illustrates its fields as they are given in the manual. Note that the contents of the TC register are established by the **config_mmu** routine during system initialization, and remain unchanged thereafter. The behavior its contents specify thus applies to all processes running on the system, in both system and user mode.

Enable Bit (E) Enable or disable address translation by the MMU. As mentioned, when disabled, the address is passed straight through to physical memory.

Supervisor Root Pointer Enable Bit (SRE) This bit determines whether the system and user spaces are defined by separate maps, or by a single map. If the SRE bit is set, then the spaces are separately defined, and the supervisor root pointer register (SRP) contains the *physical* address of the system map, while the CPU root pointer register (CRP) contains the *physical* address of the user space map. The MMU decides which of the root pointers to use based on the function code associated with the memory reference.

If the SRE bit is a 0 in TC register, then SRP is disabled, and only the map pointed to by the CRP is used, whether the function code indicates a system or user mode reference. However, inside the map it is still possible to separate the system spaces and user spaces into distinct sections. It is easy to see that the choice of how to set the SRE bit will have a significant impact on the map structure required to describe the system, and thus on system properties such as context switching time.

In Spring we have chosen to set the SRE bit, and to manage the system and user spaces with separate maps. There are several reasons, but the most obvious is that the system

space is much larger than that of an application program. The reason for this is that the 2Mb of reflective memory is included as part of the system address space. Other reasons for using separate maps are related to context switching overhead, and how we use the ATC.

Function Code Lookup (FCL) The FCL bit also has a significant impact on map structure. A map is actually a translation tree, with the root pointer specifying its root. If the FCL bit is set, then the first level of the map is indexed by the function code, dividing the whole translation tree into 8 subtrees which correspond to the categories specified in Table 6-1 on page 6-4 of the MMU manual, and are illustrated in Figure 7-2 on page 7-2 of the MMU manual. Only five spaces are actually defined: program and data space for user and system mode operation, and CPU space. Within the subtrees defined for each of the function codes, the logical address is decoded as explained in the Table Index section that follows.

If the FCL bit is a 0, then the function code is not used to determine which portions of the map are used, although the classification of the access according to its being made in system or user mode, and whether it is a program or data reference still has an effect on the MMU operation. However, as far as the map structure is concerned, when the FCL bit is 0, then the top level of the map is indexed as described in the Table Index section. Spring currently sets the FCL bit to 0.

Page Size (PS) The PS field holds the $\log_2$ of the page size selected. The smallest possible page size is 256 bytes, while the largest possible page size is 32k bytes. The page size must obviously be some power of 2. The current choice for Spring is 8K. This corresponds to a value of 13 ($\log_2(8196) = 13$) for the PS field.

Initial Shift (IS) The IS field specifies how many of the most significant bits in a logical address will be ignored by the MMU during the address translation. Note that although these bits will not be decoded by the address translation facility, the upper 3 logical address bits may be used by the access level control if access levels are enabled. For more detail see page 6-7 and section 7.2 in the MMU Manual.

Table Index (TIA, TIB, TIC, and TID) After the IS bits have been ignored, what remains of the logical address is divided into up to five fields. These are called: TIA, TIB, TIC, TIB, and *offset*. The offset field is the section of the address which specifies the address's offset within the logical page. Its width is obviously determined by the page size selected in the PS field. It should also be obvious that this field occupies the least significant portion of the address. The remaining portion of the address, between the IS and offset sections, is divided into as many sections as there are levels in the address translation map. The TIA field specifies the portion of the address to use as an index into the first level of the table. The TIB field specifies the portion of the address used as an index into the second level, and so on for TIC and TID. Accordingly, in the TC there are TIA, TIB, TIC, and TID fields which specify the corresponding field widths in the logical address.

The tree thus has at most 5 levels, where level 0 is a root pointer, level 4 nodes are page descriptors, and the intermediate levels are "table descriptors". In this case the function

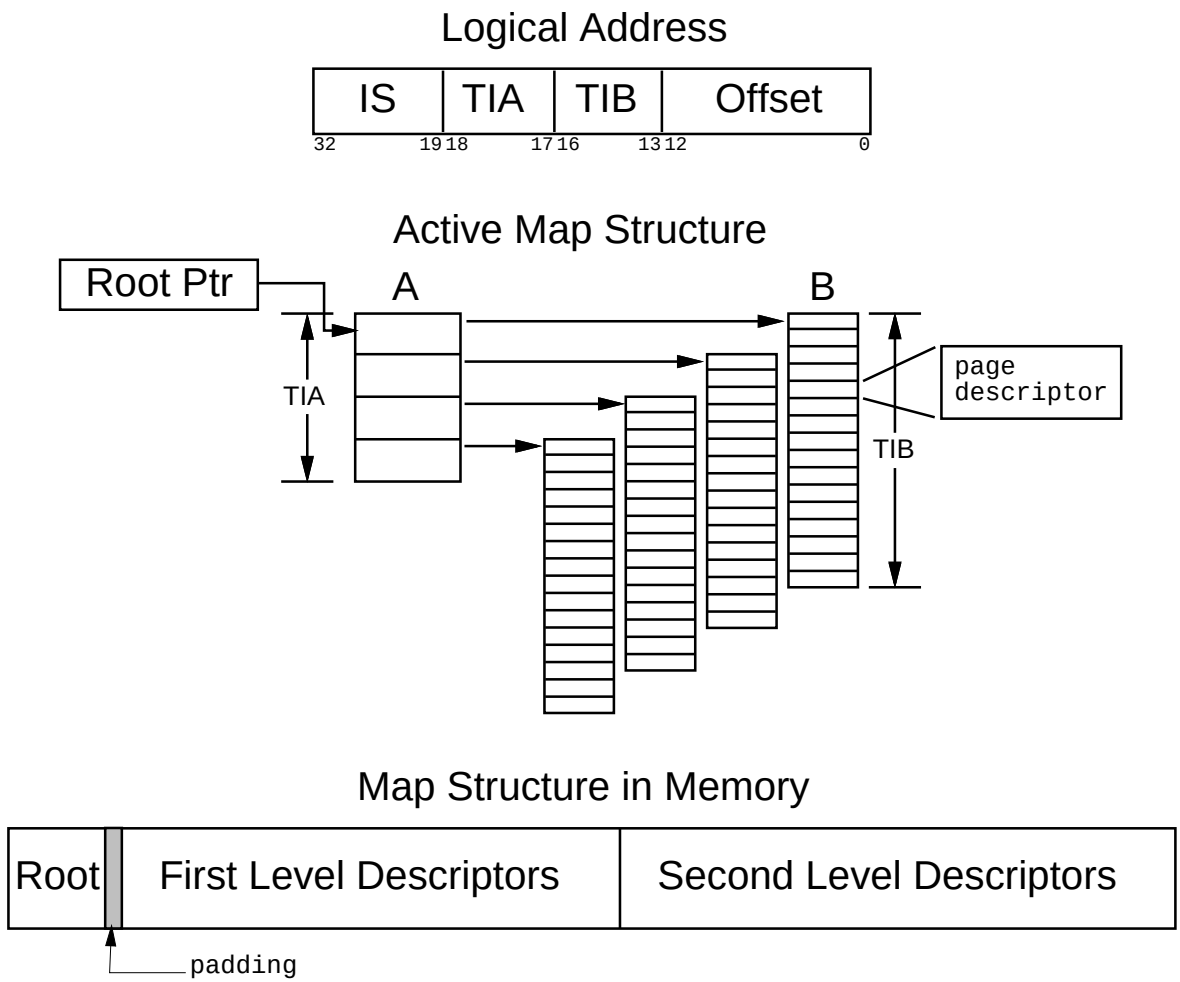


Figure 3: Logical Address Fields and Map Structure

code is only used to determine which root pointer, and therefore which map, should be used, but is not used to lookup a branch inside a map. A three level example (both TIC and TID are set to zero, so there is no C level and D level) can be found in Fig 5-3 on page 5-6 in MMU manual. In the example each B level node (each item in the B level table) is a page descriptor, while each A level node is a table descriptor. This is also the structure used for maps in Spring.

Figure 3 illustrates a logical address divided into the fields used in Spring, shows the corresponding map structure as interpreted by the MMU, and the layout of a map in memory. The SRP or CRP register points to the base of the A level table for the system or user map, respectively. The layout of the map in memory shows that the root structure is at the beginning of the memory map, followed by the first level table, and the second level tables. A table is *active* when its root structure is loaded from memory into a root pointer register in the MMU. Of particular interest is the shaded region within the root pointer structure. This represents padding added after the structure describing the root pointer, which play a vital role when we wish to create the kernel's pool of memory maps, as discussed in Section

4.3.

For a user program, Spring currently supports a 512K address space, although there are restrictions on how much of it can be in use during execution, as explained in Section 3.6. The map structure illustrated has a TIA of 4, and a TIB of 16 giving table sized of 4 at the first level, and of 16 at the second level. This map thus specifies the mapping for 64 pages, which at 8K per page provides a 512K address space. The initial shift (IS) is thus 13. When an address translation is in progress, the MMU uses the values of the IS, TIA, and TIB fields to gain access to relevant portions of the logical address. These values are then used as indices into the corresponding tables of the map.

These values are appropriate for Spring application processes, but the system map in Spring must be larger, requiring 32 entries at the A level. This provides a logical address pace of 4Mb, and requires a TIA value of 5. Since the TC register fields are constant for all maps in the system, this would imply that the first level table would contain 32 entries for all maps. However, we can avoid this by using the limit field in the root pointer, described in the next section.

3.1.2 Root Pointer

The root pointer register is a register in MMU pointing to the top level table of the map currently in use, the *active* map. The MMU always consults the address translation cache (ATC) before it goes to memory to consult the map. When the cache lookup fails, the MMU uses the appropriate root pointer register to find the map, and then uses the logical address processing described in the table index section to find the relevant descriptor. If the right page descriptor is found, the MMU will load the descriptor into the ATC.

Note that a root pointer register being active does not necessarily mean the register is actually used during address translation. A root pointer register is actually used only if a page descriptor is to be loaded into ATC by explicitly using the **pload** MMU instruction, or as an implicit result of an ATC miss. Once a page descriptor is loaded in the ATC, then references to addresses on the page it describes will be translated without consulting the relevant root pointer. The root pointer is discussed in detail in section 6.1.1 of the MMU manual.

There are 3 root pointer registers in the M68851 MMU:

- CPU Root Pointer (CRP): If the SRE bit in the TC is set, then the CRP register will only be used for user space references, and therefore should always point to the current application process's memory map. Recall that in this case when the user process makes a system call, thus shifting to system mode execution, it will use the system map pointed to by the SRP. Otherwise, if the SRE bit is not set, then the CRP should always point to the current process's map, which must contain a mapping for both the application and system spaces.

- Supervisor Root Pointer (SRP): The SRP register will be active only if the SRE bit is set, as is true for Spring. The SRP points to the system map associated with the current process. However, usually a single system map will be shared by all processes and the system kernel, including interrupt service routines.

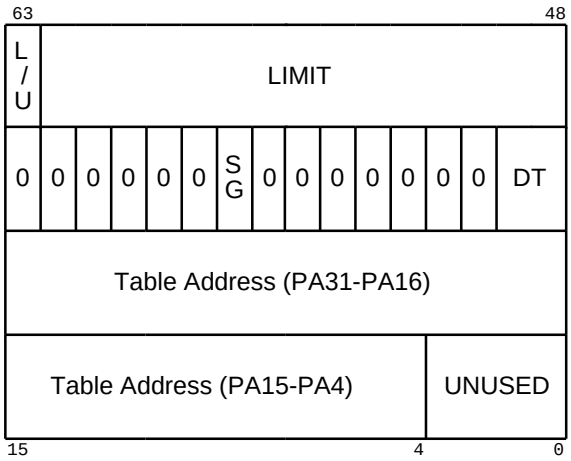


Figure 4: Root Pointer Register Format

- DMA Root Pointer (DRP): The DRP is only active during DMA cycles. We are not currently doing any DMA in Spring, and this register is therefor not used.

Spring currently uses the MMU with settings which cause it to use the SRP for references made while the processor is running in supervisor mode, the CRP for use mode access, and which leave the DRP unused. Figure 4 illustrates the fields and layout of the root pointer register.

Lower/Upper (L/U) This bit specifies whether the value contained in the limit field is an upper or lower limit. The limit is applied to the value of indices into the first level translation table which are extracted from the logical address as described. If the bit is unset, the value is used as an upper limit, if set it is used as the lower limit. Spring uses an upper limit on the indices.

Limit The limit itself applies to the value of the indices into the translation table derived from the logical address as specified by the values of IS, TIA, TIB, and PS. The limit value is set appropriately for a system and user map under Spring. A system map has access to all the first level table entries, while a user map is only permitted access to the first four, in keeping with the smaller size of the application process address space. Note that the limit register's value is *inclusive*. So, for instance, if we use it to limit the size of the first level table to 4, we would set it to 3, which then permits access to table entries 0 through 3.

Share Globally (SG) The shared globally bit indicates that the logical space mapped by the root pointer is shared by all processes within the system. This is used in Spring for the system space map, indicating that all processes share a single system address space. This is extremely important in coaxing predictable behavior from the MMU, and is discussed in greater detail in section 3.6.

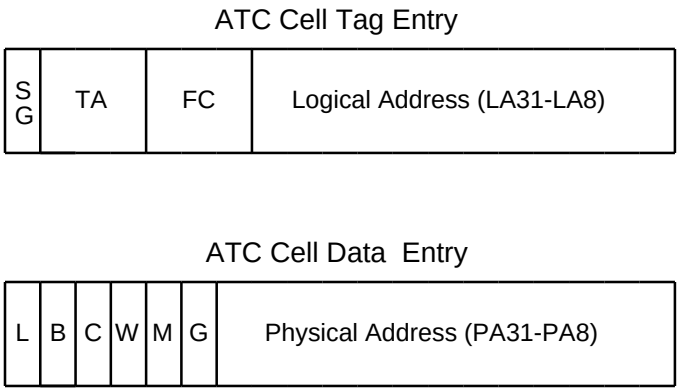


Figure 5: Address Translation Cache Cell

Descriptor Type (DT) The descriptor type of the descriptor contained in the root pointer. Spring uses an 8 byte descriptor in the first level table. Those used in the second level tables are 4 bytes long.

Table Address The physical address of the first level translation table. Note that the restricted width of this field means that the descriptors which compose the first and second level tables must all exist on 16-byte boundaries. This is extremely important when creating a pool of memory maps, and is why the padding after the root pointer was added to the map as it is stored in memory, as illustrated in Figure 3. All the descriptors used are either 4 or eight bytes in length. Spring ensures proper alignment as discussed in Section 4.3.

Unused The unused bits are available for the use of the operating system, but Spring has, as yet, not used them.

3.1.3 Address Translation Cache

The Address Translation Cache (ATC), is a set of very fast of mapping registers which normally contain the descriptors for the most recently accessed pages. Since the MMU was actually designed for conventional systems where average case performance is the relevant metric, the hardware is designed to automatically replace an existing entry if the descriptor for the current reference is not found. The MMU is designed to find the relevant entry in the memory map independent of the CPU. The ATC is described in section 5.2 of the MMU manual.

Figure 5 illustrates the contents of an ATC cell, which consists of a "tag" section, and a "data" section. The tag section uniquely identifies each descriptor stored in the corresponding data section using a combination of the logical page address, the function code (FC), and a "task alias" (TA). The task alias is mapped to a specific process through a table in another section of the MMU which contains the process's map address in the task alias position. How much of the logical address field is actually used in calculating a match depends on the page

size specified in the PS field of the TC register.

The data entry of the ATC cell contains the physical address of the page, and some control bits. These are generally duplicates of the control bits in the map's page descriptor. Most of the bits are not of importance with respect to how Spring approaches memory mangement, but they are all at least mentioned.

Locked (L) The L bit signifies whether the entry is locked into the cache or not. Spring could, but does not need to, lock entries in the cache. The entries are not currently locked.

Bus Error (B) This bit indicates that references to the address should produce a bus error. This feature is not used in Spring.

Cache Inhibit (C) This bit indicates that references on the page mapped by this entry should not be cached. Specifically, this causes the cache inhibit line to be asserted when the corresponding physical reference is made. This feature is not relevant to Spring since the current target hardware has no caches of any consequence, and the cache that does exist is disabled.

Write Protect (W) This bit states that the page mapped by this entry is read-only. Spring uses this feature to detect write operations to code pages, and its use could be expanded to read-only data pages as the SGS matures. This bit is copied out of the page descriptor in the memory map.

Modified (M) This bit indicates whether the page has been modified. Spring sets this bit for data pages, but the system largely ignores it. The reason is that Spring is not implementing virtual memory and thus does not have to manage the consistency of a working set of pages. However, it is important that Spring sets it from the beginning, because this avoids having the MMU try to set it automatically, when a data page is modified for the first time.

Gate (G) The gate bit indicates whether the corresponding page is permitted to contain module descriptors. This is not used in Spring.

When a logical address is considered for translation, each cell in the ATC makes a comparison in parallel, and the cell with the tag section which matches *all* aspects of the logical reference then places the physical address it stores on the output lines of the cache. The cell considers a reference to match only if every portion of its tag is identical to the corresponding aspects of the reference. The only exception to this is when the shared global (SG) bit is set in a tag, then only the logical address and the function code are considered. This is why all pages in the system map are marked as shared globally, so that all processes making system calls can use the same ATC entries for the system space. This is discussed in greater detail in Section 3.6.

The ATC in the MMU contains 64 cells. The strategy used for Spring is to load all the descriptors for the system space in the ATC, and leave them there. The descriptors

for processes are then explicitly loaded and flushed as the currently active process changes. Spring is thus using this feature of the MMU quite differently than its designers intended, but this strategy is required to obtain performance that is both predictable and satisfactory. This is discussed in Section 3.6.

3.2 Address Mapping

The Spring system's goal is to support the execution of real-time programs in a way that permits their behavior to be predicted. It must also do so at a performance level that is not grossly degraded by the measures used to ensure predictable behavior. The management of the system and user logical address spaces is particularly important in this respect. The predictability and performance issues for logical address space management arise in two contexts: address translation and context switching.

Address translation for a given address space is specified by the memory map. As described, Spring uses a two level map, which supports a 512K address space for applications, and a 4M address space for the system, assuming 8K pages. Spring sets the SRE bit in the TC, thus specifying that both the CRP and SRP registers are used. The function code is used to choose between the CRP and the SRP, thus helping to support the simultaneous use of separate user and system mode address spaces. A process uses the application address space when it is executing in user mode, but uses the system address space when executing a system call, since it is in system mode.

In a conventional system supporting virtual memory, the ATC is used to decrease the *average* address translation time. If a mapping for the page referenced is not present in the cache, then the MMU consults the memory map specified by the relevant root pointer register. The MMU does this *independently*, while the CPU is stalled waiting for its address reference to be satisfied. If the map entry indicates that the logical page is currently mapped onto physical memory, then the descriptor is loaded into the ATC, flushing an existing entry if the ATC is full. If the logical page is *not* currently mapped onto physical memory, then a page fault is generated, and the system code responsible for mapping pages takes over.

All of this is perfectly reasonable for managing virtual memory in conventional systems, but is hopelessly *unpredictable*, and thus unsuitable for real-time systems. More precisely, the only correct prediction for the time required to satisfy an address reference, in a completely unrestricted program, would be that required to service a page fault. This would result in ludicrously large worst case execution times.

The Spring system imposes a number of restrictions on the programming language for real-time programs, and on the behavior of many other aspects of the system to achieve reasonable performance which is also predictable. In the context of address translation, we accept restrictions on the size of both the user and system address spaces. In particular, we accept hard upper limits on the number of pages actually used by the system and the current process.

To avoid the page fault delay, we ensure that all the logical pages of the system and the real-time processes are always mapped to physical memory. We can thus rely on the assertion that a memory reference generated by a process required to satisfy real-time preformance and predictability requirements will never cause a page fault. However, we would, in the

worst case, still have to assume that every memory reference would cause an ATC miss. This at least quadruples the effective memory access time, since every memory reference would require finding the relevant page descriptor in a two level map. For example, it is easy to construct a process which uses 65 pages, and accesses them in a "round-robin" fashion, thus guaranteeing an ATC miss if a LRU replacement policy is used.

To avoid the memory map search time, Spring accepts a strong restriction, but one which is required to implement a predictable system with adequate performance on the current target hardware. We limit the total number of pages *used* by the system and the current process's logical address spaces to the size of the ATC, which has 64 entries. Since the system space is shared by all processes, we keep its entries in the ATC at all times. Those left over are available for use by the *current* process.

Given that the ATC is loaded with mappings for all the pages that are required to support the current process execution in both user and system mode, memory access time becomes easily predictable, since all ATC searches will always result in hits. Our strategy for predictably managing logical address spaces is thus based on explicitly managing the contents of the ATC. When a context switch is made between one process and another, the ATC entries for the old process are flushed from the ATC, the CRP register is set to the root of the map for the next process, and the the new ATC entries are loaded.

This obviously adds to the overhead of context switching, but it is comparable to what would have occurred as the pages which the process actually used were faulted into the ATC on demand. We ensure that the MMU does not resolve an ATC miss without our knowledge by marking the first level entries of all maps *invalid*, except when the ATC contents are being explicitly manipulated. This ensures that the MMU will generate a bus error trap if an ATC miss happens at any time we do not anticipate one.

This approach to managing the logical address spaces in the Spring system obviously imposes significant limitations on the size of the system and application code, but while they are somewhat annoying, they are not as onerous as it might seem. If we assume, for example, a page size of 8K and that 40 entries are used by the system, leaving 24 for the current user process, then the system code can occupy 320K, while the user process could use 192K.

We can increase the upper size limits in several ways. First, we can use a larger page size; up to 32K. This would, of course, increase internal fragmentation. However, possibly more important, is the tradeoff that exists between the page size and the maximum number of processes that a processor board can support. Each board has 4Mb of memory. The system requires some number of pages, leaving the rest for processes. However, as currently implemented each process requires a *minimum* of four pages; one each for the code, data, user stack, and system stack. A larger page size obviously means a smaller upper limit on the number of active processes that can be supported by a processor board. We believe that the 8K page size gives a reasonable compromise between the competing issues of maximum process size and the maximum number of processes.

The number of pages used by the system can also be somewhat reduced by judiciously manipulating the system entries in the ATC. It is probably necessary to have all the system code represented in the ATC at all times, and also the page containing the system stack, and the generic system global variables. However, there are other system pages that can

be selectively mapped. The IPC facilities are a good example of this. They manipulate a large number of buffer areas, which cannot all be mapped at the same time. The current design has the IPC routines manipulating the mappings of various logical pages as they copy messages from place to place. We are thus able to limit them to using only three or four ATC entries beyond those required for their code and non-buffer data.

Our approach is sufficient to our needs, and is capable of obtaining predictable behavior and adequate performance from our target hardware. The limitations of our approach are not trivial, but Spring can still be implemented successfully and provide an excellent research platform. The system tends to have greater internal memory fragmentation as a result, but that is a comparatively small price. However, this does indicate that a MMU better suited for real-time systems may be among the most pressing issues for future development. An architecture which managed segments in memory, rather than pages, would have several advantages. We discuss this and many other real-time hardware issues elsewhere [5, 4].

The rest of this section first considers the memory map structure and the memory required to support it, and then discusses the various routines providing support for the various aspects of memory management, and some aspects of their use. These are more detailed presentations of the major points made in the general approach just described.

3.2.1 Memory Map Structure

As mentioned in Section 3.1.1, there are two map structures that are of potential use for Spring. The first one describes separate system and user spaces in a single map, and uses the function code as an index into the first level of the map. The second way is to describe system and user space with separate maps, and to use the function code to make the choice between the SRP and the CRP.

Spring uses the second method, and describe and address space using a two level translation tree, as illustrated in Figure 3. The MMU TC register fields which control the size of the TIA and TIB fields in the address apply to all maps. Once the TC is set up, and the MMU is turned on, we cannot alter its contents until the MMU is turned off or reset. As a result, some clash between the system and user space requirements is inevitable since we wish to use different sizes for the system and user spaces.

As shown in Figure 3, the map in memory begins with the root pointer structure, followed by the first level table, and then the second level tables. Each map begins with a root descriptor, which has the following structure, consuming eight bytes:

```
typedef struct rpr {
    unsigned    l_u:       1; /* upper or low limit */
    unsigned    limit:    15; /* level access limit */
    unsigned    :          6;
    unsigned    sg:        1; /* shared global */
    unsigned    :          7;
    unsigned    dt:        2; /* descriptor type on the next level */
    unsigned    tbl_addr: 28; /* pointer to the first level table */
    unsigned unused:    4; /* available for use */
} rp_reg;
```

The root structure *rp_reg*, as described in Figure 5-9 on page 5-14 of the MMU manual, sets the limit for the first level table, notes whether the space described by this map will be shared globally or not, and specifies the address of the first level table. However, note that the table address field contains only the most significant 28 bits of the table address. The MMU assumes that the table will be aligned on a 16 byte boundary, so that it can assume that the last four bits of the address are all zero.

```
typedef struct lftd {
    unsigned    l_u:       1; /* upper or lower limit */
    unsigned    limit:    15;
    unsigned    ral:       3; /* read access level    */
    unsigned    wal:       3; /* write access level   */
    unsigned    sg:        1; /* shared globally      */
    unsigned    s:         1; /* supervisor/user mode */
    unsigned    :          4;
    unsigned    u:         1; /* used/unused/accessed */
    unsigned    wp:        1; /* write protection     */
    unsigned    dt:        2; /* next level descr type */
    unsigned    tbl_addr: 28; /* address of next level's table */
    unsigned    unused:    4; /* available for use */
} lftbl_dscp;
```

The first level table descriptor *lftbl_dscp*, as described in Figure 5-13 on page 5-18 of the MMU manual, is 8 bytes in length, and describes the limit for the second level tables using the *l_u* and *limit* fields. The *sg* field indicates whether the pages in the descriptor's domain are shared or not, while the *s* field, if set, specifies that this section of the address space is only accessible in supervisor mode. The *u* and *wp* fields indicate that the area has been accessed, and that it is write protected, respectively. The *dt* field specifies the type of descriptors in the second level table, and *tbl_addr* gives the 28 most significant bits of the table's address.

```
typedef struct sfpd {
    unsigned    pg_addr: 24;  /* physical page address */
    unsigned    g:        1;  /* module descriptors for CALLM instr */
    unsigned    ci:       1;  /* cache inhibit */
    unsigned    l:        1;  /* Lock */
    unsigned    m:        1;  /* modified */
    unsigned    u:        1;  /* used */
    unsigned    wp:       1;  /* write protect */
    unsigned    dt:       2;  /* descriptor type */
} sfpg_dscp;
```

The second level table descriptor *sfpg_dscp*, as described in Figure 5-14 on page 5-18 of the MMU manual, is 4 bytes in length, and describes the mapping of a logical page to a physical page. The *pg_addr* field holds up to 24 of the most significant bits of the physical address, which shows why the minimum page size if 256 bytes. The *g* field is unused in

Spring, since we are not using the CALLM instruction. The *ci* field inhibits instruction caching when set. We do not set this field, but turn caching off at the processor level.

The *l* field exempts the entry created in the ATC from replacement by other entries. We do not set this bit, but ensure that ATC entries are not flushed through explicit management of the ATC as described in Section 3.6. The *m* and *u* fields indicate that the page has been modified and that it has been accessed, respectively. Spring sets these as part of its ATC management strategy. The *wp* field protects the page against being written, and is used for code pages. The *dt* field indicates whether this descriptor is valid or not. That is, whether it represents the mapping of a logical page in use, or represents an unused logical page.

3.3 Memory Map Overhead

As with everything else in the system, the memory maps themselves occupy memory, and memory is a tight resource in Spring. As a result, we have taken some care to reduce the overhead of the memory map pool, though it may not yet be fully minimized. The previous section described the structures from which the memory maps are constructed. In this section we discuss how various approaches to a map's definition and use affects its size.

Spring's approach to predictable process management requires that all of a process's page descriptors must be loaded in the ATC while it is executing. This means that the maximum number of pages used by a process cannot exceed 64, the size of the ATC. Since the process entries must share the ATC with those for the system space, the upper limit will obviously be smaller, but there are other reasons to give the process some flexibility in deciding *which* pages within its space it will use, so we give it 64.

```
typedef struct{
    rp_reg     rtp;                        /* root pointer struct        */
    char       padding[MM_INTERNAL_PAD];   /* pad to a 16 bytes boundary */
    lftbl_dscp lvl1nodes[U_TIA];           /* level 1 nodes              */
    sfpg_dscp  lowpages[U_TIA][TIB];       /* level 2 tables: page descr */
} usermap_t;
```

We need to be able to describe the root pointer, first, and second level tables of a memory map as the structure *user_map_t*, which is contiguous in memory, so that we can create a memory map resource pool as described in Section 4.3. However, as discussed in the previous section, the address fields within the root pointer and table descriptor structures omit the least significant 4 bits, and thus assume all tables are alinged on a 16 byte boundary in memory. Yet, the root pointer structure is only 8 bytes long. As a result, we add an area 8 bytes in length after the root structure, as illustrated by the grey rectangle in Figure 3. The tables themselves are all multiples of 16 bytes in length, and the map as a whole is thus a multiple of 16. We can thus ensure 16 byte alignment of every descriptor in every table simply by allocating the pool of maps on a 16 byte boundary.

We originally assumed that both the system and user spaces could be limited to 64 pages. Given a page size (PS) of 8K, the logical space of each process was thus $8K * 64 = 512K$. So, originally we used a TIA of 4 and TIB of 16, which specify a first level table with 4 entries and second level tables with 16 entries.

The size of these maps was then:

$$16 + TIA * 8 + (TIA * TIB) * 4 = 16 + 4 * 8 + (4 * 16) * 4 = 304$$

Given the memory size of each board, and the fact that each process requires a minimum of 4 pages, an upper limit of 100 processes on each board is reasonable for discussion. Given our original view of the memory map, the memory maps for 100 processes would have required 30400 bytes, or a bit less than four ATC entries worth of memory. using an 8K page size. For practical implementation reasons it is convenient to have the memory map pool always mapped into system space. While ATC entries are a scarce resource, four seems fairly reasonable.

However, for reasons related the IPC subsystem's use of Scramnet, the *system* space must be 4Mb. Providing maps capable of mapping such a space for *all* processes would consume a ludicrous amount of memory, so we were faced with the problem of using different map structures for the system and user spaces. To support the 4Mb address size, we use TIA and TIB values defining a map with 32 entries at the first level, which each point to a table with 16 page descriptors:

$$TIA * TIB * PS = 32 * 16 * 8K = 4M$$

The simplest approach to creating this map is to make it capable of mapping the entire 4M area. This would require the system map to have $4M/8K = 512$ page descriptors, even though at any given time substantially fewer than 64 of them can be in use because of the ATC limitation. However, assuming we wish to be able to map the entire system logical space, this would give a map size of:

$$16 + TIA * 8 + (TIA * TIB) * 4 = 16 + 32 * 8 + 512 * 4 = 2320$$

Since we need only a single system map, its memory overhead is not particularly important. The top 2M of this space are reserved for mapping the image of Scramnet's reflective memory upon which the IPC facilities are based []. The IPC code carefully manipulates the contents of the ATC to ensure that it is only using its allotted entries at any given time, but this manipulation is substantially simpler if the entire reflective memory is described by the map.

The system code can then occupy the lower 2Mb of the system space, but it is clear that given the ATC limitation the code will have to be much smaller than that. Given this reality, we have limited the system code and data to the first 512K of the address space. We cannot reduce the number of first level entries, but we *can* discard the second level tables for the 1.5Mb portion of the space never used. This approach requires only 4 of the first 16 possible second level tables, while the last 16 are used to map the reflective memory. This gives a system table size of:

$$16 + 32 * 8 + (4 * 16) * 4 + (16 * 16) * 4 = 1552$$

For a user process, a 512K space is appropriate, but the TIA value implies a first level table of size 32, even though just as with the first half of the system space, only the first 4

entries will be valid. If we marked all but the first four entries invalid, we could supply only four second level tables, giving a user map size of:

$$16 + 32 * 8 + (4 * 16) * 4 = 528$$

This is still fairly big, compared to the original 304 bytes. If there are 100 maps for user processes, the total amount of memory used by memory maps on each board would be:

$$1552 + 528 * 100 = 54352$$

This would require a bit less than 7 pages in the system map for memory maps, which is still substantially larger than the original requirement of 4 pages with a user map size of 304. However, the limit field of the root pointer can be used to ensure that any reference to an address beyond 512K will cause an exception. We can thus, for an application process's map, only define a first level table with four entries, even though TIA is 5, implying 32. This is why the *user_map_t* structure uses *U_TIA* as the size of the first level table. *U_TIA* is 4, and is used to set the *limit* field of the root pointer. The size of a user memory map is then:

$$16 + 4 * 8 + (4 * 16) * 4 = 304$$

and a total memory consumption by memory maps of:

$$1552 + 304 * 100 = 31952$$

This would require only the 4 pages used by the original scheme where the system and user spaces were the same size.

In summary, it should now be evident that the TC register is set up for handling two level maps, and that the system and user spaces are different sizes. We save some space in the system map by not allocating second level maps for 1.5M that we do not anticipate needing. In the user maps we support only a 512K address space, and thus only need 4 second level tables, using less space than the system map. We save even more space by using the limit field in the root pointer.

It is not possible to decrease the size of the memory map pool using the current map structure. However, if space becomes enough of an issue to justify the effort, it might be possible to use short format descriptors in the first level table. However, since this would only save 4 bytes per descriptor, and the first level table only contains four, this is only save 16 bytes per map. At this time the effort required to change the table structure and verify that it works correctly seems to be far greater than the space saving would justify.

3.4 Memory Map Initialization

The **init_map** function is defined in the file "mm/mm_map.c", and is responsible for initializing a map for use. It is important to note that this initialization consists of defining the map's structure and many of its properties, but does *not* include allocation or mapping of any physical pages. Physical pages are allocated and mapped as necessary after the map is initialized.

The routine first sets a few working variables to values that depend on whether the map is a user or system map, and the root pointer structure is then initialized. The *l_u* field is set to indicate that the limit field will be an upper value, rather than lower. The limit field is then set according to whether this is a user (U_TIA) or a system map (TIA). Remember that the limit field in the root pointer imposes the limit on the index into the first level table. This setting thus imposes the limitation that a user process can only use 4 entries in the first level map. The *sg* field is then set if this is a system map, since all system processes share the same address space. The descriptor type field *dt* is set to indicate that the first level table uses the 8-byte descriptor type.

The address for the first level table is set, as appropriate for a user or system map. It is important to note that this is guaranteed to be on the required 16-byte boundary as long as the whole map is defined on a 16-byte boundary, since we padded the root structure, and each of the first and second level tables are multiples of 16 in length. We guarantee that all maps are defined on 16-byte boundaries when we create them. In the case of the system map this required a small trick in the **init_system_process** routine. In the case of the user process maps, we simply arrange that the pool containing them is allocated on a 16-byte boundary.

The routine then sets the contents of the templates for the level one and level two descriptors. This is an optimization that avoids all the address arithmetic required to set each field from a pointer to the structure within the loops that follow. In any case, when the templates are set, the routine simply steps through the level 1 table, setting the descriptor, and initializing the corresponding level 2 table. The only tricky part is skipping the gap that exists in system maps.

During the initialization every page and table descriptor is marked invalid, and in each page descriptor the physical address field is set to 0. Pages will be mapped individually as a physical page is allocated, and the mapping specified. The *l_u* field in each first level descriptor is set to 0, and the *limit* field is TIB-1. This means all TIB page descriptors in the corresponding second level table can be accessed by the MMU in that section of the map, if the descriptor type field (dt) indicates there are valid descriptors on the next level. Both *ral* and *wal* fields are set to 0, since the fields are meant for access level control which is not used in the Spring kernel.

We initialized the access control (AC) register in **config_mmu** during system initialization. For more discussion of the fields in a long format table descriptor see section 5.1.5.3 on page 5-20 in the MMU manual and the comments in the **init_map** routine.

3.5 Memory Map Use

A map is allocated from the memory map resource pool, and it is initialized as an empty map using **init_map**. As physical pages are allocated to support pages of the process's logical address space during initialization, the mapping from the logical to physical page is added to the map using **map_page**. Note that the execution process control block (EPCB) contains, among the several items required to manage a running process, the address of the memory map and the structures required to manage its logical address space. The pointer to the EPCB of a process is the hook used by the system to obtain access to the information

require to manage process execution.

The **map_page** routine takes the map address, the logical and physical page addresses, and a page type as arguments. Currently, three types are defined: CODE_PAGE, RO_DATA_PAGE, and RW_DATA_PAGE. The types differ from one another in how the write protect (WP) and modified (M) bits are set in the memory map's page descriptor. A code page has its WP bit set and the M bit cleared. We use the same settings for a read-only data page, while for a read-write data page we clear the WP but *set* the M bit. We set the M bit for data pages for two reasons that are both subtle and instructive.

The first reason for setting the M bit is to avoid access of the memory map by the MMU during program execution. The MMU is designed to maintain the M bit as an indication that the page has been modified. When the page descriptor in memory has the bit set, there is no problem. However, if the modify bit is not set, and the program writes to the page, then the MMU will attempt to update the M bit in the descriptor in memory. This is an unnecessary overhead for Spring, and we keep the map marked invalid most of the time in any case. We mark the read-write data pages as modified so that this update will not happen. We also set the used (U) bit in the descriptor for a similar reason. The MMU will try to update it the first time a page is used, but this is a waste of time, and will fail since the map is marked invalid. We do not mark the code pages as modified because the MMU objects to the idea of fetching code from a modified page.

The reason we set the M bit for a read-only page is that we need to know whether a page contains code or data in order to load it correctly. In conventional systems, the ATC entries are loaded on demand and the proper function code is available, since it is associated with the memory reference that is causing the ATC miss. However, since Spring explicitly manages the ATC contents, we must use the **pload** instruction, which requires a function code as an argument. We must know if the page is for code or data to know the proper function code, but the page descriptor does not contain this information. As a result, we have used the slightly underhanded trick of semanticly overloading the M bit. If it is set we treat the page as a data page, and otherwise it is a code page. These are all examples of how the MMU is biased toward conventional systems. An MMU designed for Spring would have quite different properties from the one we are using [4].

An established page mapping can be canceled using **unmap_page**, which simply marks the page descriptor invalid, and sets the physical address field to 0.

Note that both **map_page** and **unmap_page** only deal with the map itself, and have no effect on the ATC. This is important because the mapping of a particular logical page may be changed by unmapping its previous physical target page, and then mapping it to a new target page. However, this only changes the map entry in memory. The entry in the ATC, which will remain active until it is explicitly flushed, still indicates the previous target page. This is a fairly simple problem in most cases, and is even an essential feature which is used by the procedure which switches context from one process to another. It is vital, however, to remain aware of the distinction between the map that is in memory and the entries that are in the ATC.

3.6 ATC Management

For a page mapping to be used, an entry for it must be loaded into the ATC. In conventional systems this is a natural consequence of program execution. When a page is referenced for the first time, it causes an ATC miss which then causes the MMU to consult the memory map. However, since Spring is being designed as a predictable system, we prefer to explicitly load the ATC with the page mappings for the active process, and thus ensure that all memory references are satisfied by ATC hits. However, since the chip is designed to resolve ATC misses without processor interference, it is not a trivial matter to enforce this. The strategy we have adopted is to mark the memory map *invalid* except when the ATC contents are being explicitly manipulated. This ensures that if the MMU tries to consult the memory map at any time except those explicitly permitted, it will encounter an invalid map and generate a bus error.

As a result, we have created a set of interface routines which enable us to explicitly manage the contents of the ATC, and to change the validity of memory maps. The **make_valid** and **make_invalid** routines mark the map specified as valid or invalid, respectively. This operation is fairly efficient, because we need only mark the level 1 descriptors as invalid, without having to touch the more numerous level 2 descriptors. When the system changes context marking all the level 1 descriptors is required, but in some other cases, we wish to manipulate only the mapping of a single page. The routines **make_valid_page** and **make_invalid_page** are provided for this purpose, and are slightly faster, since they change the validity of only the first level descriptor within whose domain the specified page falls.

The routine **mmu_load** provides the ability to load a page's mapping into the ATC. It can only be used when the RP register in the MMU has already been loaded with the root pointer of the map from which we wish to load the page's entry. The logical address of the page which the descriptor represents is used as an argument, as is the mode of the access. The mode is important because it determines the value of the function code field in the tag portion of the ATC entry created for the descriptor, which determines in turn the references for which the ATC entry will be valid. Further, different MMU instructions are used to load page descriptors with different attributes, for a read-only page **ploadr** is used, while **ploadw** is used for writable pages.

The **mmu_load** primitive is used by the **load_app_map** routine, which takes care of loading the entries associated with an entire application address space into the ATC. This routine has been fairly heavily optimized, and is thus regrettably less clear than it might be. The routine first loads the CRP with the root structure of the map, and then extracts the address of the first level table from the root pointer structure. It then loops through each entry in the first level table, but note that the limit on this loop is U_TIA, which is a reflection of the fact that this routine handles only application process's maps. The body of the loop first marks the table descriptor valid, and then extracts the address of the second level table mapping the pages within the domain of the first level entry. The routine then loops through the second level table, loading each valid descriptor it finds according to its mode, as indicated by the write protect and modify bits.

After the valid entries in the second level table are loaded into the ATC, the first level entry is again marked invalid. This ensures that we can enforce our assertion that all memory accesses are satisfied through ATC hits. If a miss occurs, the MMU will access the map and

find an invalid descriptor. This will cause it to generate an exception, and the system will thus detect the deviation from predicted behavior.

There are several interesting points exhibited by **load_app_map**. First, the map itself is searched to find all the pages currently used by a process. It might seem that another data structure would be more efficient, but a careful consideration of the overheads involved shows that, in the worst case, this approach is fairly reasonable. It is clearly less efficient for processes using a small number of pages, but any alternative would certainly have to list the pages used, and in the worst case this would closely approximate the linear scan of the memory map. However, The context switching time is clearly a function of the number of pages used, and explicitly listing them would greatly decrease the overhead of context switching. The practical problems of modifying the code to track the set of active pages whenever the application's map is modified would not be trivial, but should not be onerous either.

Another interesting point is that an application process could use too many pages. However, application process maps must be guaranteed to have no more valid entries than can be supported. Recall that the application process is limited to a 512K address space, and that the number of active pages within that space is limited by the number of ATC entries left when the active system entries have been loaded. The number of pages a process can use is thus a function of the size of the system code being used. However, the number of system pages required is known when system booting is complete. The system can thus monitor the number of active pages in an application address space when the map is being modified, making it unnecessary to do so while loading it. This is an obviously more efficient approach than checking while loading the map, which occurs with every context switch.

Several things are special about the system map. First, recall that it is only loaded into the ATC once, and then individual entries are managed thereafter. The reason for this is that the system map is shared among all processes. Another special thing about the system map is the gap that exists between the lower 512K where all the code and data of the system resides, and the upper 2M where the IPC section is mapped. For these and other reasons, we use the special routine **load_sys_map** to load the system map for the first and only time. Creating a separate routine enabled us to optimize **load_app_map** for application processes and thus to improve context switching time.

Entries in the ATC can be flushed in three ways. The **mmu_flusha** routine flushes all the ATC entries. This routine is not used in the system, and is not likely to be used since it would flush the system entries, but we have used it for testing purposes. The **mmu_flush_proc** routine flushes all the entries associated with a particular process from the ATC. The argument to the function is the address of the root pointer structure in the process's map. This function takes advantage of the fact that the MMU maintains an alias list tagged with the root pointer address. Recall that the tag portion of an ATC cell contained a task alias. The MMU consults the task alias list to see if any has a matching root pointer value. If such an entry exists, then it flushes all entries in the ATC with a matching alias. This routine is used in **change_context** to flush the entries for the old process from the ATC, to make room for those of the new process. Note that under our regime for explicitly managing the ATC contents, flushing the ATC is just as important as loading it.

A separate routine, **mmu_flush_shared_entry**, is required to flush a system entry from the ATC. It takes a function code and logical address as arguments. The function code

classifies the entry into system or user code and data. This and the logical address is sufficient to uniquely identify a shared entry.

4 Resource Management

An operating system manages the ownership and use of many different types of resources, including but not limited to the logical and physical memory spaces within which processes execute. This section of the document discusses the four categories into which Spring resources are divided, and the routines which support management of each type. The code within the system address space on each processor in a Spring node manages the resources resident on that board. Resources shared among all processors in a node are controlled using data structures within a shared segment which is attached to all the system address spaces in the node, but manipulated by code within the system address space on each board.

The first resource category is the set of physical memory pages on each processor board. Given the number of them, care was taken to minimize the size of the data structure used to describe each page so the space overhead would not be too great. Managing the second resource category, pages within the logical space of the system and of each process, presented a similar problem of efficient representation, but different requirements for what information must be represented.

These resource categories contain only a single instance. The third category contains more than one type of resource, but each shares the property of being a fixed sized object. For example, memory maps, execution process control blocks (EPCB) and scheduling process control blocks (SPCB) are each managed as separate resource pools, but since each are of fixed size, they can be managed using the same routines. The final resource category contains pools of variable sized objects, which are managed using the approach made familiar by the **malloc** routine on UNIX. While only a single such pool exists in the kernel space at this time, it is possible to imagine reasons for having more than one.

The rest of this section describes each resource category in greater detail, and then presents the routines available in the system address space for managing them.

4.1 Physical Page Pool

The most basic resource in the system is the physical memory of each board. Since we are using a page oriented MMU in the current version of Spring, physical memory is managed as a set of pages. The methods used to manage this resource do not have to be particularly sophisticated, but they must be able to support a set of basic operations. We clearly wish to allocated and deallocate pages, but it is also prudent to anticipate that physical pages will be shared under some circumstances. This will be useful, for example, when we wish to give the system the ability to have processes running the same program share the physical pages containing the program's code.

```
typedef struct {
    short size;                 /* body size */
    short free;                 /* free page counter */
```

```
    short head;                  /* index to the head of the free list */
    short tail;                  /* index to the tail of the free list */
    pg_item body[BOARD_SIZE/PS]; /* one item for each page  */
} phys_page_pool_t;

typedef struct {
    char ref;     /* reference count */
    char attrib;  /* page attributes */
    short next;   /* index to the next free page */
} pg_item;
```

The pool of physical pages is described by the *phys_page_pool_t* structure, which records the size of physical memory in pages, the number of free pages, and a head and tail pointer into a list of free pages. The data structure used to describe an individual page is kept simple to keep it small. The reference count in the *pg_item* structure keeps track of the number of address spaces using this physical page. The attribute field notes whether the page is available for use, and if used whether is can be shared or is held exclusively. The structures given are specified in "inc/sys/pools.h", and the routines discussed next are in "boot/pools.c".

When the kernel on each board boots, it runs the **init_phys_pool** routine, which initializes the structure *Physic_zone*, which is of type *phys_page_pool_t*, and the array of *pg_item* structures within it, adding all physical memory on the board to the free list. The booting sytem then "allocates" the physical pages the system code already occupies to reflect the situation, and is then ready to allocate new physical pages as needed.

The routine **alloc_phys_page** allocates a page with either the *EXCLUSIVE* attribute, or with *NO_EXCLUSIVE* if it can be shared. This routine simply goes to the free list and allocates whatever physical page is at the top, or fails if no free physical pages remain. The routine **alloc_specific_phys_page** is the same, except that it attempts to allocate the specified physical page. This routine is used when the system is booting to allocate the pages the system code already occupies, but is not really required at other times, since a program executing in a logical address space can use any physical page as easily as any other.

The routine **share_phys_page** records the fact that the system is granting use of a page already in use to an additional consumer. This function could have been built into **alloc_specific_phys_page** but it was decided that page sharing is a sensitive enough issue that the sharing should be explicitly requested, rather than having it happen as an implicit side effect of a request to allocate a page already in use. The routine **free_phys_page** performs the obvious function of decrementing the page's reference count, and adding the page to the free list when the count becomes zero.

4.2 Logical Page Pool

The Spring system manages many logical address spaces. The system code and data reside in one space, and each process running on the system has a separate address space. Each of these address spaces represents a resource pool of logical pages. However, the methods used to manage the logical page pools differ from those for the physical page pool because

sharing is not a consideration, and the *wp* and *m* fields of the page descriptor in the memory map are enough to tell **load_app_map** how to load the page's descriptor into the ATC as code or data. The data strucutres discussed are in "inc/sys/spr_proc.h" and the routines in "boot/pools.c".

```
typedef struct {
    int base;       /* base address of space, usually 0 */
    int size;       /* size of log_addr_space */
    int end;        /* address 1 unit beyond end of space */
    int first_free; /* address of first free page from the BOTTOM */
    int last_free;  /* address of first free page from the TOP */
} log_addr_mngmnt_t;
```

Since each process running on the system is associated with its own address space, the structure used to manage a logical space should be as small as possible. The structure *log_addr_mgmnt_t* provides a reasonable degree of control, at a moderate cost. The *base* field gives the logical base address of the area under management, while the *size* field gives the size. The *end* field is provided to speed range checking. When applied to process spaces, the base will be zero and the size 512K. However, the routines were written in a more general way to provide support if Spring begins managing spaces of different sizes at some time in the future. The *first_free* and *last_free* fields record the address of the free pages that are lowest and highest in the logical space, respectively. These are used by the allocation routines, and are updated by the deallocation routine.

The most notable feature of the structure is that there is no support for tracking the status of each page in the space. The reason for this is that we wished to used the same management routines for managing the system logical space and process spaces, even though they are different sizes. Thus, we declared the data structure tracking the status of each page separately. Since we only need to know if a page is allocated or not, we use a *bitmap* to track page status. For example, the declarations within the execution process control block (EPCB) for the structures tracking the process's logical address space are:

```
log_addr_mngmnt_t log_space_params;
char              log_space_bitmap[HALF_MEG_BITBYTES];
```

A bitmap is initialized using the **init_bitmap** routine, which takes a pointer to the management structure, a pointer to the bitmap, the base address of the space, and the space's size as parameters. The basic routines for manipulating bitmaps are **alloc_page_in_bitmap** and **alloc_specific_page_in_bitmap** for allocating the first available page found, or a specified page, respectively. The first routine takes a pointer to a *log_addr_mgmnt_t* structure, a pointer to the space's bitmap, the number of pages required, and a direction parameter indicating whether the search for a free page should start at the bottom or top of the address space. If an area larger than one page is requested, the routines allocate a contiguous area. The **free_page_in_bitmap** routine frees the pages described, and takes the management structure, bitmap, base address, and size as parameters. The basic routines for managing the bitmaps are then called from within the routines used within the kernel which operate on a process logical space, or on the system logical space.

The routines **init_sys_log_page_pool**, **alloc_sys_log_page**, and **alloc_specific_sys_log_page** perform the obvious functions on the system logical space. An interesting detail is that since there is only one system logical space per board, the management structure and bitmap pointers are not needed as parameters of these routines. The routines **init_proc_log_page_pool**, **alloc_proc_log_page**, **alloc_specific_proc_log_page**, and **free_proc_log_page** perform the obvious functions, but take a pointer to the process's EPCB as a parameter. The routines use the management structure and bitmap for the process contained within its EPCB.

4.3 Homogeneous Object Pools

Many resources in the operating system can be described as pools of objects which are of uniform size. Such objects include the memory maps used to describe a logical address space, the EPCBs containing the information required to manage process execution, the SPCBs containing the information required to schedule process execution, and buffer pools used to support IPC. Use of a data structure describing a pool of such fixed items makes it possible to manage most such pools using a single set of routines.

```
typedef struct pool_hdr{
        addr_t       base;      /* base addr of pool in system log space */
        int          blksize;   /* size of each item */
        int          blknum;    /* number of items */
        int          freeblks;  /* number of free items */
        pool_item_t *free_list; /* linked list of free blocks */
} pool_hdr_t;
```

The *base* field of the *pool_hdr_t* structure specifies the base of the pool in the system logical space, while the *blksize* and *blknum* give the size and number of the items comprising the pool. The *freeblks* field maintains a count of the number of items in the pool remaining unallocated, and the *free_list* field anchors a linked list of free items.

```
typedef struct pool_item_str {
    struct pool_item_str    *next;
} pool_item_t;
```

The free list is implemented using a simple C programming technique. The *pool_item_t* structure contains nothing but the pointer required to implement a simple linked list. The technique employed is to cast a pointer to an object in the pool as a pointer to a *pool_item_t* structure. The *next* pointer required to maintain the free list thus occupies the first few bytes of the pool object.

The **init_pool** routine takes a pointer to a *pool_item_t* structure, the base address of the pool, the number of items and their size as parameters. It then creates a free list containing all the items in the pool. The only limitations of this method are that the items must be at least large enough to contain a pointer, 4 bytes on the current target architecture, and that the first pointer sized section of a free item is available for use in this way.

This is not always as obvious as it seems. For example, the EPCB pool is maintained using this approach, but the pool is also searched as an array by other routines which use the

name and *pid* fields of the EPCB. Without arguing the merits of searching the pool linearly, it is important to note that for the two approaches to coexist without problems neither the *name* or *pid* fields could occupy the first four bytes of the EPCB.

The **alloc_pool_item** and **free_pool_item** routines are quite simple, since all they need do is take a pool item off the free list, or add one to it. The advantage of this approach in a real-time system is that the allocation and deallocation takes constant time, which makes this important aspect of the system predictable.

4.4 Heterogeneous Object Pool

Pools of objects of fixed size can be managed efficiently by simple routines. However, not all objects are easily described by structures of fixed size, and even those of fixed size may be required in quantities subject to such wide variation that creating separate pools for each is unacceptably wasteful. In such situations it is better to create a single pool from which objects of various types and sizes can be allocated. The drawback to this is that the time required for allocation and deallocation for such a pool is much less predictable.

```
typedef struct {
    int      size;
    addr_t   base;
    kchunk_t *free_list;
} kmalloc_pool_t;
```

In Spring we use the method made familiar to all by the *UNIX* **malloc** routine. The *kmalloc_pool_t* structure records the base and size of the area, as well as anchoring the free list. The **kmalloc_init** routine takes the base address and size of the area for the pool as parameters, records them in the *kmalloc_pool_t* structure, and establishes the area as a single block of memory on the free list. The routines currently do not take a pointer to the pool descriptor as an argument, thus assuming that there is but a single kernel pool of heterogeneous objects. However, they can be easily modified if that assumption becomes invalid.

```
typedef struct kchunk_str {
    int               size;
    struct kchunk_str *next;
} kchunk_t;
```

The **kmalloc** routine takes the size of the object required as an argument, and searches the free list for the first free area that is large enough. The free list is composed of *kchunk_t* structures linked together, with each indicating the size of its area. However, it is important to note that these structures occupy the first eight bytes of the *actual* free areas within the pool, and that the address of the *kchunk_t* structure is also the address of the free memory area. The items on the free list are sorted by address.

When an area large enough is found, it is split into a section that will contain the new object, and the area left over. There are several important points to note. First, the area

used to contain an object is actually four bytes larger than the object. The extra space, immediately preceding the object, contains its size. This is vital since the **kfree** routine must be able to know the size of the areas being freed to do its job. Another point is that an item on the free list cannot be split unless it leaves a piece at least as large as the *kchunk_t* structure. When this is not true, the routine simply allocates the whole chunk, providing an area a little larger than requested.

The **kfree** routine takes the pointer that **kmalloc** returned as a parameter. It then subtracts four, and casts that address as a pointer to a *kchunk_t* structure. This gives it easy access to the size of the object. The routine then scans the free list, looking for the proper place to put the freed area according to its address. When the freed area is disjoint from the items on either side of it in the list, it is a simple matter of linked list insertion. However, to avoid the fragmentation of the pool into ever smaller pieces, the **kfree** routine *merges* items on the free list when possible. The problem is obviously addressed when an item is inserted into the free list. When the proper place on the list is found, the routine checks to see if the new area is contiguous with either of its neighbors. If so, it merges the areas involved, thus avoiding unnecessary fragmentation.

5 Address Space Creation and Manipulation

Previous sections of the document have described the data structures and routines supporting the creation and management of address spaces and resource pools, but these low level operations merely provide the foundation for the issue of primary interest; how the system and processes' address spaces are created and used. However, more information about two subjects is required before the details of how the system address space is constructed during system booting will make sense. The first area of interest is the model of a computation in Spring as a single thread of control within an address space. While this is simple and familiar, explicitly discussing its details will make later points clearer. The other preparatory subject is how segments of memory shared among two or more address spaces are described, created, and used.

When the details of these subjects are known then the steps taken during system booting, discussed in Section 6, will make a great deal more sense. Section 7 will then discuss how processes sharing one or more segments are created and run.

5.1 Spring Computation Model

A computation in the Spring system, including those performing system supervisory roles, is a thread of control within an address space. There is a single system address space on each processor board within a Spring node, and each application process running on a board has a separate address space. Segments of memory can be shared among address spaces, and are attached to the spaces using them, as specified in one of several ways. The way in which shared segments are created and used in Spring presents some moderately novel features. These are discussed in the next section.

When the system boots it establishes the address space within which threads of control execute when they are in system mode. This includes all the processes' threads of control

while they are executing system calls. When a process is activated, an address space is established within which its thread of control will execute when it is in user mode. The system first allocates an SPCB, EPCB, and memory map for the process, and then allocates logical and physical pages as required to hold the code and data of the process as specified by the executable file. This includes reading the code and initialized data into the appropriate allocated pages. Initialization can also includes attaching some or all of the shared segments used by the process to its address space.

However, the initialization operations that the system can do for a process necessarily remain at a fairly generic level. Initialization that is specific to the semantics of the process must be left to the application code. In a conventional system this is easily handled in the code immediately following the entry point, and is of no particular interest. However, in a real-time system it is necessary to distinguish between the aspects of the program that are required to initialized its data space in *preparation* for execution under real-time constraints, and those aspects which are actually part of the real-time computation.

With this distinction in mind, Spring processes, by convention, have two entry points. The first, **proc_init** is used when the process is being activated, to give the application code an opportunity to perform application specific initialization of its data space in preparation for execution under real-time constraints. This initialization can take many forms, including the creation and attaching of shared segments. The **proc_exec** entry point is where the system transfers control when the execution of an instance of the process is initiated under real-time constraints.

A process's address space is thus defined by the code and data segments produced during compilation, and those shared segments attached to it during process activation. The system performs a number of operations that prepare the process for execution, but the **proc_init** entry point provides support for initialization that is specific to the semantics of the application. The **proc_exec** entry point is then available to precisely define the computation that is performed under real-time constraints, unsullied by any special cases for initialization.

5.2 Shared Segments

Shared segments in the Spring system play a number of different roles, but are always used to provide an area within which information can be exchanged among producers and consumers. For example, the system might define a shared segment within which it posts information about the current load on the system and other status information. The system would be the only writer of the segment, but any processes wishing to see this information could attach it to their address spaces. Other shared segments may be both read and written by all processes using them, thus supporting a multidirectional information flow. Shared segments can also be used to give application processes access to memory mapped control registers of peripheral devices or sensors. For example, in a robotics application, the sensors' control and status registers could be attached to a process's address space as a shared segment mapped to the physical memory areas containing them.

Shared segments generally define an area of overlap between two or more address spaces. Within the area of overlap, the processes view the data they share according to the definitions they have for the structure of the shared segment. In conventional systems shared segments

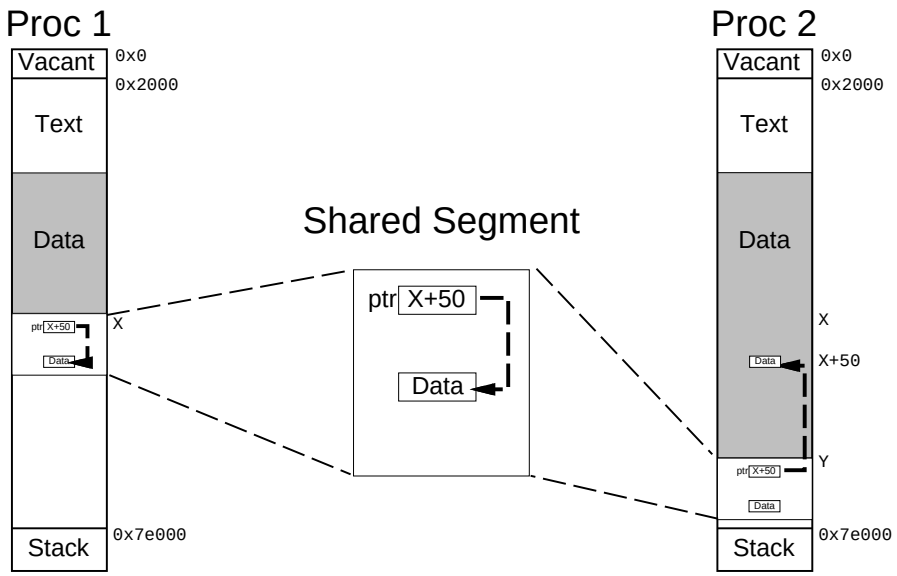


Figure 6: Shared Segment Pointer Conflict

are generally manipulated through system calls attaching the segment to the address space of the calling process, and returning the base address established for the segment. The process using the segment then uses the base address and a set of data structure definitions to create pointers to the various elements within the segment. This works well in many cases, but is subject to several limitations.

The most obvious is that the segment's structure is described by the code used to establish a process's pointers into it. If the code in one process using the segment has a different view from that in another process, the error can be difficult to detect since it can only be investigated at run-time, and there are no automatic ways to compare two descriptions of the segment as embodied in the code establishing each process's set of pointers. The second limitation is that the compiled code may be slightly slower, due to the address calculations associated with pointer use, than the equivalent code using data whose addresses can be determined at compile time. Using pointers can also make the code using a large segment containing a variety of information pretty complex. Neither of these limitations is particularly severe, and good programming practices can mitigate them, but an approach that made shared segments easier to describe and use would be welcome.

A more serious limitation may exist, however, depending on the semantics of the system call attaching a segment to the process's address space. The problem arises when the shared segment contains pointers to other data within the segment. In many systems, the system call attaching a segment to a process's address space simply picks the first free area in the logical space it finds, which is of sufficient size. However, if the area chosen by the same method in a second process's address space is attached at a *different* base address, then the pointers within the shared data are guaranteed to be wrong for one of the processes. Figure 6 illustrates the problem.

Assume that the two processes *Proc1* and *Proc2* contain code which will correctly find the pointer *ptr* relative to the base of the shared segment, and that *ptr* is *supposed* to point to the *Data* which is at an offset of 50 from the base of the segment. Let us also assume that the code is written so that the first process to attach to the segment initializes *ptr*. Finally, let us assume that the system call attaching a segment to an address space places it at the first available location following the current end of the data space.

If *Proc1* attaches the segment first, the segment will be attached to the address space at location X, and *ptr* initialized to the value $X + 50$. However, the data space of *Proc2* is larger than that of *Proc1*, and when it attaches the segment, it is placed at location Y. The code *Proc2* contains which gives it access to the variables within the segment works correctly, but the *value* of *ptr* is incorrect in the address space of *Proc2*, as illustrate by the arrow pointing to $X + 50$, which is in *Proc2's* existing data space.

One solution to this dilemma is to disallow the use of pointers within shared segments, but this is insufficient to support Spring's implementation. However, there is a way to provide a definition for the structure of a shared segment which takes advantage of the compilation process to maintain a consistent view of the segment contents, and which can be used to guarantee that shared segments containing pointers will appear at the *same* logical address in *all* spaces to which they are attached. The method does have costs, as well as providing benefits, but provides a novel way to manage shared segments as an integral part of the software during development and compilation. This is accomplished by treating shared segments as separately compiled objects in the system, by providing a language within which the properties of a shared segment can be precisely defined, and by providing far more sophisticated system support for them during system boot and at run-time.

5.2.1 Attaching Shared Segments

Shared segments can be attached to an address space in several ways. The most familiar is attachment at run-time using a system call. If a process attempts to attach a segment that does not already exist, the segment is created, and the calling process becomes the owner. Supporting the creation and attachment of a shared segment predictably under real-time constraints is theoretically possible, but the system currently limits segment creation to boot time and process initialization time, which are not subject to real-time constraints. If a process or group of processes are written using the system call approach to attaching shared segments, then each process would call **shm_attach** in its **proc_init** routine.

The call takes the name of the segment, its size, and a pointer to an integer as parameters. The name uniquely identifies the segment within the node. The system support for shared segment management is currently limited to each separate Spring node. Extension to segments shared across node boundaries, supported by the Scramnet reflective memory, would be fairly straightforward. If the segment is successfully attached to the calling process's address space, then the base address is returned. If not, then *NULL* is returned. If the call succeeds, then the integer pointed to by the third parameter is set to one if the calling process is the first process to attach the segment, and zero otherwise. This supports applications that need to initialize the shared segment contents. If the call fails, then the integer pointed to by the third parameter is set to an error code specifying the cause of failure.

```
typedef struct {
    char             name[SHM_SEG_NAME_SIZE];
    addr_t           log_addr;          /* logical address */
    seg_phys_page_t *phys_pages;        /* set of phys pages supporting seg */
    short            ref_count;         /* reference count of users */
    short            status;            /* status of the segment */
    int              size;              /* in pages for now   */
    int              owner;             /* board number where physical */
                                        /* memory resides is owner */
} shm_seg_descr_t;
```

When the **shm_attach** call is made, it calls the kernel's internal shared segment attach routine **k_shm_attach**. This routine searches the node's shared segment table, which is contained within a segment shared among the system address spaces of each board in the node. How this shared segment is defined and created is discussed in Section 6. The table is an array of *shm_seg_descr_t* structures. If an entry for the specified name is not found, a free entry is selected, and the segment created as specified.

If an entry already exists, then what happens depends on the attributes of the segment as indicated by the *status* field. If the segment has the *Matching* attribute, then it *must* appear at the same logical address in all spaces to which it is attached. The *log_addr* field in the segment descriptor stores the logical address of segments with the *Matching* attribute. For *Matching* segments, **k_shm_attach** uses **alloc_specific_page_in_bitmap** to allocate the necessary pages in the logical space of the calling process, using the logical space management structure and bitmap contained in the process's EPCB.

If the segment is does not have the *Matching* attribute, then **alloc_page_in_bitmap** is used to find an area of the segment's size, starting either from the bottom or the top of the process's address space. Since a segment that is not *Matching* can appear at different places in different processes, the EPCB contains a table of structures recording the logical address of each segment and pointing to its segment descriptor. Segments that are not *Matching* do not use the *log_addr* field of the segment descriptor.

```
typedef struct seg_phys_page {
    struct seg_phys_page   *next;
    addr_t                  phys_addr;
} seg_phys_page_t;
```

When the logical pages are allocated in the calling process's space, they are mapped to the segment's physical pages, as specified by the list of *seg_phys_page_t* structures anchored at the *phys_pages* field of the segment descriptor. The *ref_count* field is incremented for the segment, and control is returned to the calling process.

If the segment did not already exist, the logical space is found as described, but physical pages are also allocated. The *size* field of the segment descriptor records the segment's size, and the *owner* field records the board on which the physical memory was allocated; the board on which the calling process is executing. This is important because of the difference between local and global physical addresses, and their use of the node's VME bus. When a

```
shd_seg_desc  ::=   Shared_seg(name) { [seg_attr]* };

seg_attr      ::=   Code name;
                  | Logical_base HEXNUM;
                  | Memory_mapped;
                  | Mode mode_type;
                  | Physical_base HEXNUM;
                  | Predefined;
                  | Resources name_list;
                  | Size HEXNUM;
```

Figure 7: Shared Segment Description

process resides on the same board as the physical memory of a segment being attached to its space, local physical addresses are used in the process's memory map. However, when the segment's physical pages are on another board, then the global address of the memory is used in the map.

The **shm_attach_phys** routine also attaches a segment to the space of the calling process, but takes a physical address as a parameter. The shared segment is mapped to the specified physical address, rather than pages allocated from the physical page pool. The use of this routine is thus obviously restricted to attaching segments to physical memory *not* under control of the physical page pool of any processor in the system. It was intended for use by processes wishing to create a segment, shared or not, which would give it access to memory mapped I/O registers for some peripheral device.

The **shm_free** routine performs the obvious function of unattaching a segment from the calling process's address space. It finds the segment descriptor, decrements its reference count, and frees the physical memory and descriptor when the count goes to zero. It always frees the logical pages of the segment within the calling process's address space, and then returns.

5.2.2 Shared Segment Description

The system calls and routines inside the kernel which support attaching and removing shared segments from an address space provide the minimum foundation required to use shared segments. If they are the only support available, then the programmer is forced to do everything else by hand, writing application code that explicitly attaches to every segment. However, other forms of support are possible, providing the programmer with a more powerful interface for describing shared segments and manipulating address spaces.

Detailed description of the shared segments at the source level is supported by the System Description Language (SDL), which is discussed in detail elsewhere[3], but Figure 7 gives the grammar for the part of the SDL describing a shared segment. A shared segment description following this grammar will specify the name of the segment, and then list some or all of its attributes. The *Logical_base* statement permits the developer to specify a base address

for the segment. If this is done, then the segment will have the *Matching* attribute, and it will have the same address in all spaces attaching it. The *Memory_mapped* attribute states that this segment is used to gain access to a memory mapped area, and is used with the *Physical_base* statement which specifies the physical address of the memory mapped area.

The *Mode* statement indicates whether the segment is used by application processes (*Appl*), system processes (*Sys*), or both (*Both*). This provides a simple check on the sensibility of segment use without creating a burdensome overhead. The *Predefined* attribute states that the segment will be created during system initialization. The *Size* attribute gives the size in pages, but is not required when a *Code* file name is specified. When a code file exists, it specifies the data structures contained by the shared segment, and the size is thus readily available. The *Resources* statement lists the Spring resources supported by the shared segment, and is part of the Spring programming paradigm that will not be discussed in this document.

Note that a shared segment does not *have* to be defined using the SDL, since the system supports creating and attaching shared segments during initialization using system calls. Further, even if a description is given using the SDL, not all of the attributes of the segment need be specified. The SDL provides a way to specify properties of the segment, and makes that information available during compilation so that various parts of the software generation system (SGS) can use it to help properly prepare the segment.

5.2.3 Predefining and Compiling Segments

The novel aspect of the Spring approach to supporting shared segments is that, without eliminating the traditional system call interface, shared segments can also be treated as first class objects by the system, just as programs are. This has a number of interesting implications. First, shared segments can be described by "executable" files, just as executables describe programs. This implies, in turn, that source files describing shared segments are written and compiled in essentially the same way that programs are.

Shared segments can also exist as persistent objects, from the operating system's point of view, in that they can exist even though they are not attached to any process's address space. Such a persistent segment is described by a shared segment descriptor in the system's shared segment table, and can have physical memory already assigned to it. This is one reason why the system, when booting, has a phase where it loads the "predefined" shared segments. Entries for these segments are made in the system shared segment table, and are thus already defined when the initial set of processes is activated. Part of process activation is, of course, attaching the shared segments required to the process's address space.

Another interesting aspect of handling shared segments under Spring is that since they are compiled like programs, the data structures within a segment can be accessed by *name*, rather than only by *pointer* as is normally true. The data structures within a segment are described using normal C data structure declarations. For example, the data structures in the segment shared among the system address spaces on each processor board in a Spring node are declared in "init/kern_shared_seg.c" as follows:

```
#include "sys/spr_types.h"
#include "sys/mmu_prim.h"
```

```
#include "sys/mm.h"
#include "sys/shared_mem.h"
#include "sys/spr_proc.h"
#include "sys/pools.h"

/*
 * Global shared memory segment descriptor table
 */
shm_seg_descr_t    Shared_seg_table[SHM_SEG_TABLE_SIZE];

/*
 * segment physical page descriptor pool
 * these hang off of each segment descriptor
 * to list its physical pages
 */
pool_hdr_t  Shared_seg_physpg_desc_pool_hdr;
seg_phys_page_t    Shared_seg_physpg_desc_pool[SHM_SEG_PHYSPG_DESC_POOL_SIZE];
```

The segment is called the "basic" shared segment, since it contains the shared segment table, and the pool of physical page descriptors which list the physical pages supporting the segment. It is a good example of how a segment is compiled, and how the executable making references to data structures within it is compiled. However, it is not attached to the system address space on each board in the normal way, since it contains the table used to managed shared segments. It is attached to each space early in the booting sequence of the system code on each board, and is then avaialable to store information about all other shared segments in the node.

Giving a program, in this case the system code, access to a segment's contents by name obviously means resolving references to data structures within the segment when the program is compiled, which implies in turn that the location of the segment within the program's address space is determined at compile time. The shared segment's "executable" file is prepared by linking it, and specifying the logical address at which it should appear. This is currently done using an option to the linker, but will eventually be controlled by the description specified in the segment's source file using the SDL statement discussed. The kernel shared segment "executable" is produced using the following section of "init/Makefile":

```
kern_shared_seg: kern_shared_seg.o
    rm -f kern_shared_seg
    spr_ld -o kern_shared_seg -Tdata 0x7e000 kern_shared_seg.o
```

The "-Tdata" option specifies the base address used when the addresses of the data structures described in the object file are added to the executable produced. In this case the object file contains only data structures, and so the executable produced contains no code, only a data segment description and a symbol table. The symbol table is the most important part, since it gives the addresses of the data structures in the segment, assuming the specified base address. The executable "kern_shared_seg" is then copied up into the root of the kernel source tree, and used when the SP and AP kernel executable files are linked.

```
spr_sp: $(SUBDIR_LIB_TARGETS_SP) spr_entry.o
    rm -f spr_sp
    spr_ld $(ADDR_FLAGS_SP) -o spr_sp \
    spr_entry.o $(LIB_BOOT_SP) $(LIB_MM_SP) $(DEBUG_LIB) \
    $(SUBDIR_LIB2_SP) $(COMM_LIB)  $(MP_COMM_LIB) $(LIB_SP) \
    $(SUBDIR_LIB3) $(LIB_COMMON) -Shared kern_shared_seg   <---*****
    rm -f s
    ln spr_sp s
```

The important part of the link command is "-Shared kern_shared_seg", which specifies that the shared segment should be used to resolve variable references, but that the variables involved are *not* included in the data segment being accumulated for the executable. This is an extremely important point. The linker normally resolves references against collections of object files. When a new object file is encountered, its code and data are added to the code and data images being accumulated for the executable, the addresses of the routines and data structures added to the image are added to the symbol table, and any outstanding references to the newly added symbols are resolved to the newly determined addresses.

The new linker directive "-Shared" tells the loader that the file given as an argument to the option is an executable file, whose symbol addresses have already been established, and that the linker should use the addresses in the executable's symbol table to resolve any outstanding references. Such references are resolved to addresses *which are not part of the executable's address space.* The executable thus depends on the shared segment being attached to its address space before any references to its contents are made.

The system can satisfy this requirement, subject to a few restrictions. First, the program's executable file must list the shared segments which it assumes will be attached to its address space. This is part of the process description supported by the SDL, and the Spring executable file format includes a section for SDL information. Second, the shared segments used by programs in this way must be predefined, that is they must appear on the list of shared segments which are created prior to program activation. The layout portion of the SDL provides a way to specify what shared segments are loaded onto which boards, and the booting phase of the kernel loads the specified segments before activating any processes.

5.2.4 Current Status and Future Development

As the Spring kernel and SGS are developed, the definition, use, and run-time management of shared segments will be more strongly integrated into the various parts of the system affected. Figure 8 illustrates the major thrust of future development; the use of the compiled form of the SDL as a medium for recording information about the elements making up a Spring application, and exchanging this information among the SGS tools as compilation proceeds. Eventually, the SDL will be part of the Spring-C language, the SDL information will be part of the object files produced, will thus help to control how linking is done, and will be used by the system when activating a process.

At the current time, support exists at a fairly primitive level, requiring developers to explicitly specify some operations that will be automated as development proceeds. The SDL is defined and tools exist to compile it. A tool also exists to insert the compiled SDL

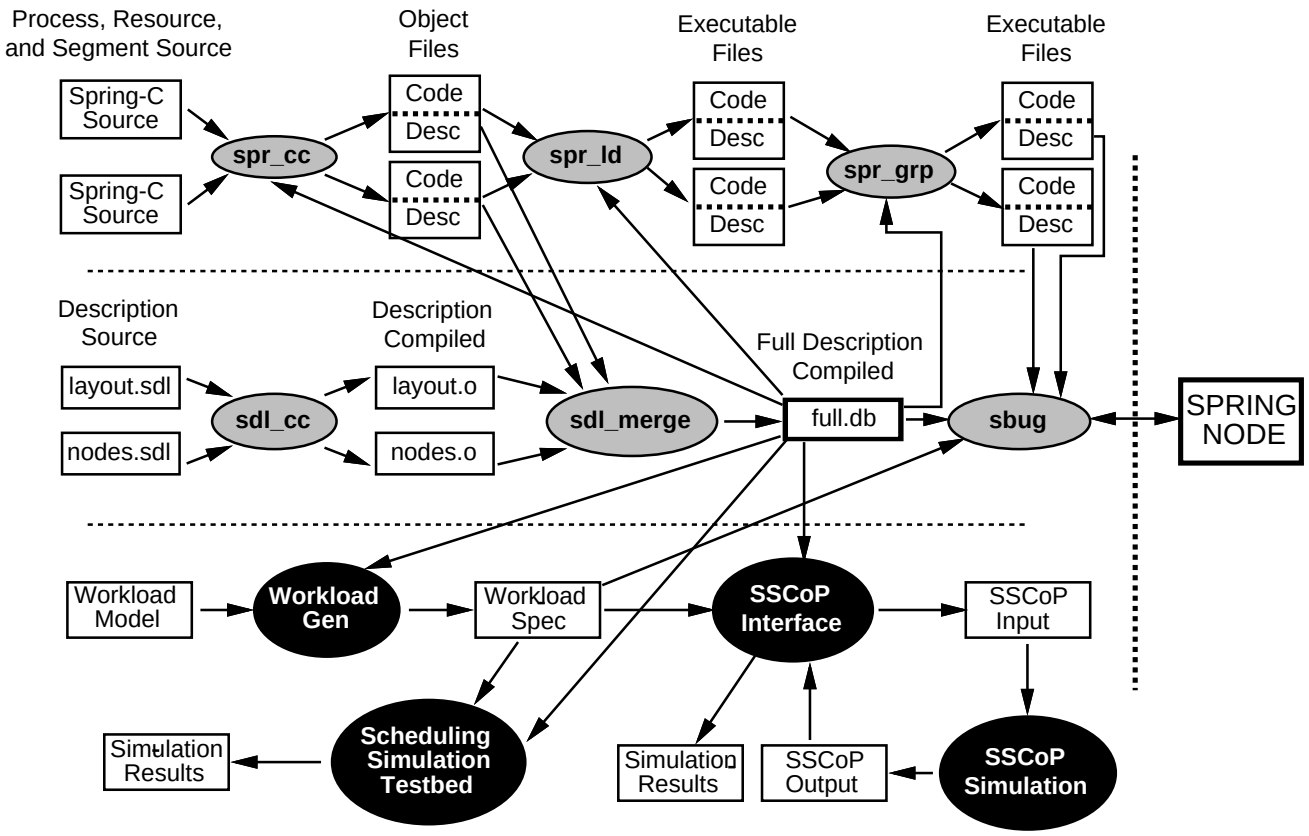


Figure 8: SGS Information Flow

information into a Spring executable. However, the C compiler is not yet able to process SDL statements included in source files, and neither the assembler nor loader use or include it in the executable files they produce. The loader option "-Shared" exists, and performs as described, but the need to specify it can be eliminated as the loader begins to use SDL information.

On the system side, the kernel supports predefinition of shared segments, and activation of processes during booting, but the debugger uses the "proc.dat" file to describe the shared segments and processes. It must begin to use the SDL layout description to drive the downloading. The system is being modified to use the SDL information contained in the executable during process activation. The current effort is to use the process and task group descriptions to set up the data structures used by the scheduler. However, the process description lists the shared segments required, which would then be attached to the process's address space.

6 Address Space Manipulations at Boot Time

- The system must establish its own address space and that of the processes in the initial

set established during booting.

- Figure 9 illustrates the structure of the application and system address spaces under Spring. This is just the layout, but is a good basic picture to keep in mind.
- The system image on each processor board first establishes the basic structure of the system space on its board, and then establishes the segments shared with other boards. Figure 10 illustrates the address spaces on each board in a Spring node, and how they share key sections with one another.
- Process activation creates address spaces for each new process. However, system processes which execute only in system mode, share the system address space. Figure 11 illustrates some interesting points. First, each process is supported by separate ohysical memory. Next, that each process has a physical page associated with it that supports the physical stack, but that only the system stack of the currently executing process is ever mapped. The system stack is *always* mapped to 0x1e000. finally, that a system process only requires a separate system stack page, since it shares the system address space with all other processes, and never executes in user mode.

7 Shared Segment Examples

- A simple shared segment mapepd into two address spaces
- A system "status" page available to all processes
- The scheduling segment shared among the scheduler and the dispatchers, while the PA segment is shared between the PA process, and the scheduler.

References

[1] Motorola, Inc. *MC68020 32-bit Microprocessor User's Manual*, 1986.

[2] Motorola, Inc. *MC68851 Paged Memory Management Unit User's Manual*, 1986.

[3] D. Niehaus. Spring System Description Language. Technical Report 93-01, Computer Science Department, University of Massachusetts, 1993.

[4] D. Niehaus, E. Nahum, J. Stankovic, and K. Ramamritham. Architecture and OS Support for Predictable Real-Time Systems. submitted to ACC Conference, 1992.

[5] J. A. Stankovic, D. Niehaus, and K. Ramamritham. SpringNet: A Scalable Architecture for High Performance, Predictable, and Distributed Real-Time Computing. Technical Report 91-74, Umass-Amherst, 1991.

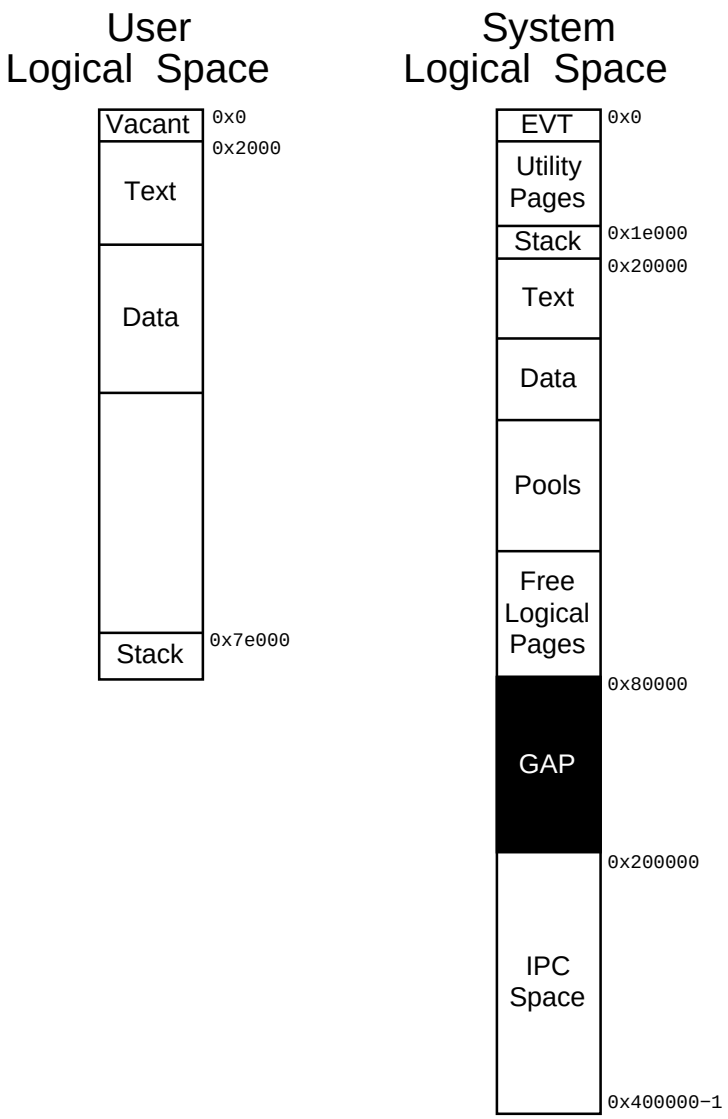


Figure 9: Application and System Address Space Structure

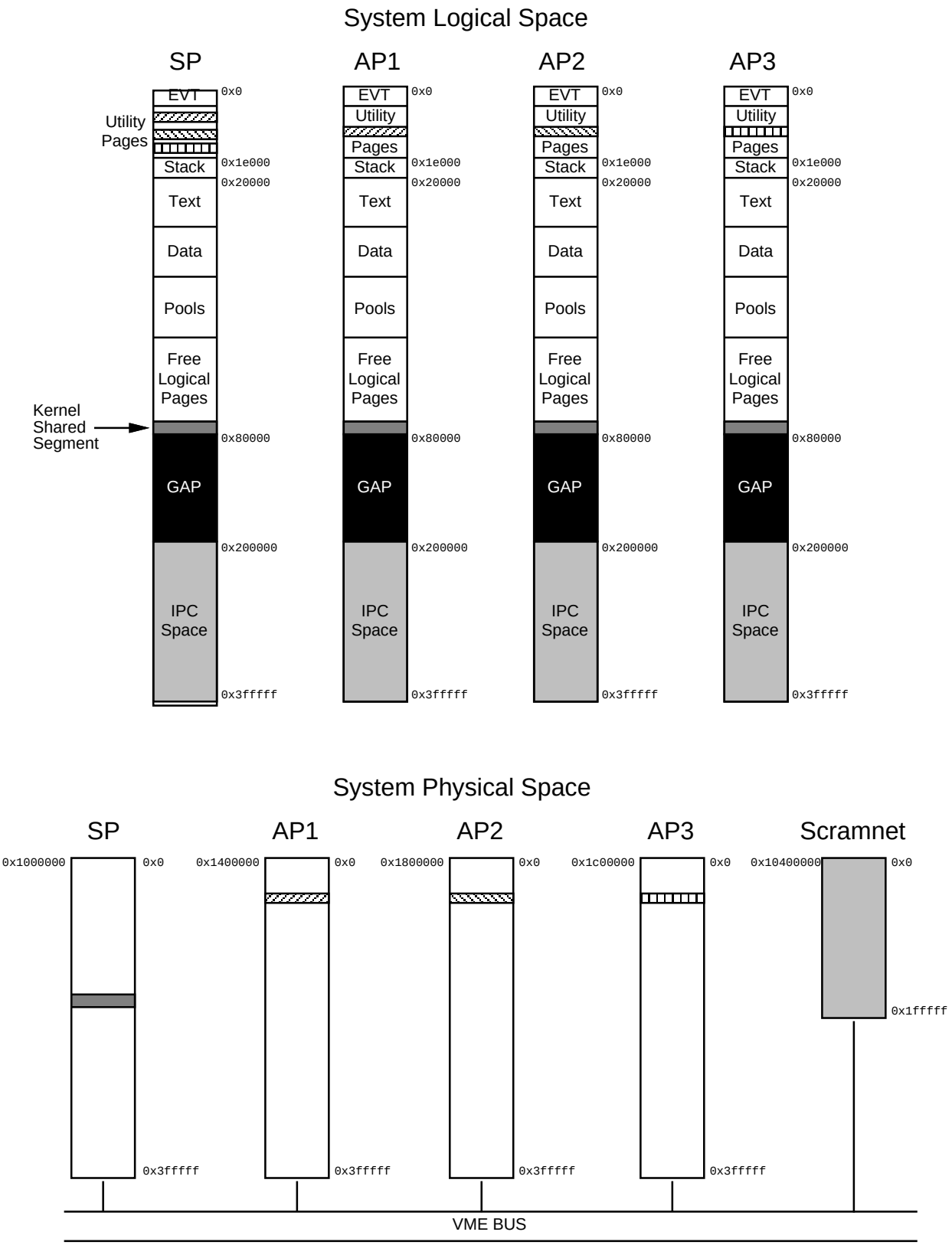


Figure 10: Spring Node System Address Space Layout

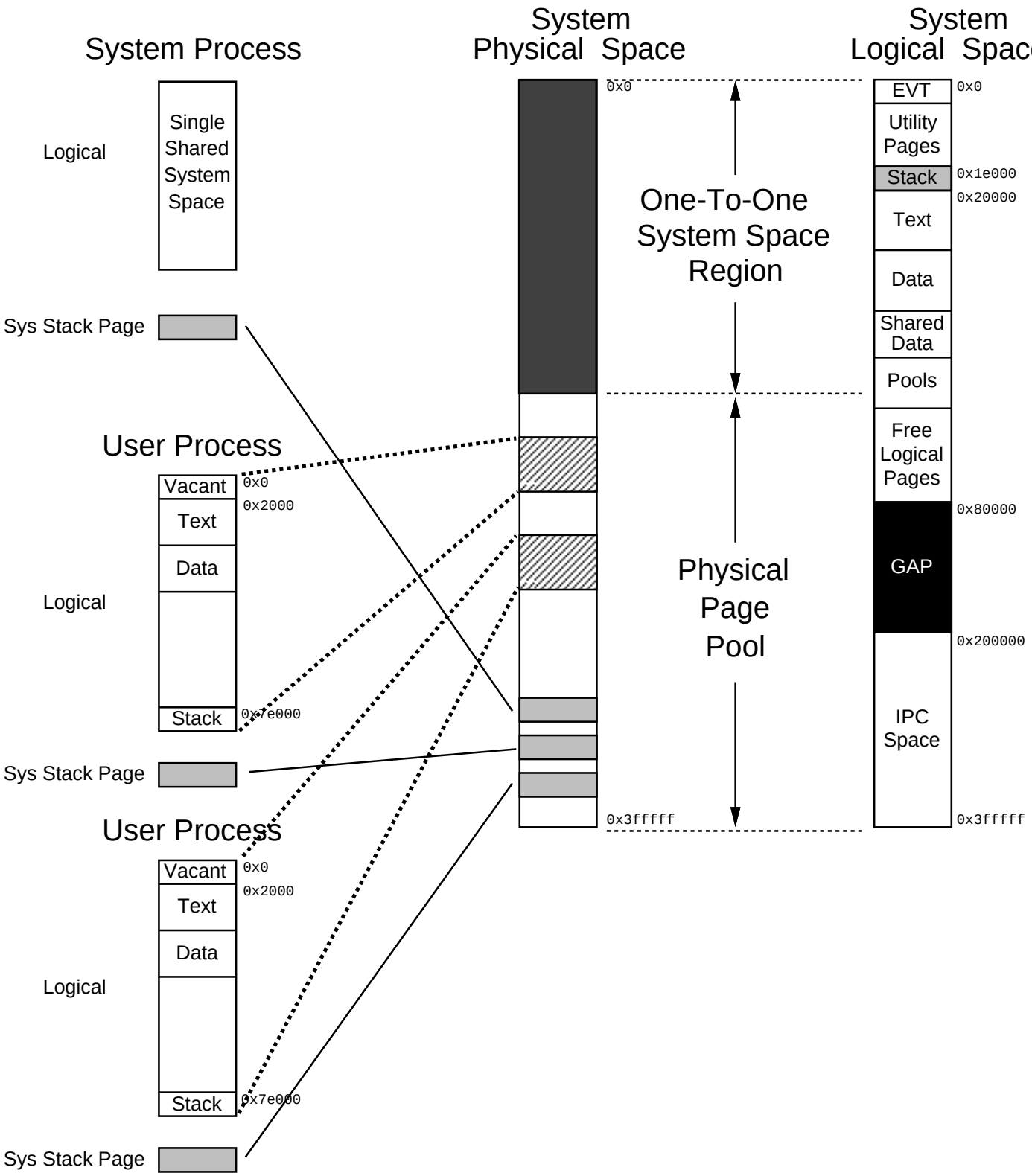


Figure 11: Application Process Address Spaces