Chapter 4

Virtual Memory

Linux processes execute in a virtual environment that makes it appear as if each process had the entire address space of the CPU available to itself. This virtual address space extends from address 0 all the way to the maximum address. On a 32-bit platform, such as IA-32, the maximum address is $2^{32} - 1$ or `0xffffffff`. On a 64-bit platform, such as IA-64, this is $2^{64} - 1$ or `0xffffffffffffffff`.

While it is obviously convenient for a process to be able to access such a huge address space, there are really three distinct, but equally important, reasons for using virtual memory.

1. ***Resource virtualization.*** On a system with virtual memory, a process does not have to concern itself with the details of how much physical memory is available or which physical memory locations are already in use by some other process. In other words, virtual memory takes a limited physical resource (physical memory) and turns it into an infinite, or at least an abundant, resource (virtual memory).

2. ***Information isolation.*** Because each process runs in its own address space, it is not possible for one process to read data that belongs to another process. This improves security because it reduces the risk of one process being able to spy on another process and, e.g., steal a password.

3. ***Fault isolation.*** Processes with their own virtual address spaces cannot overwrite each other's memory. This greatly reduces the risk of a failure in one process triggering a failure in another process. That is, when a process crashes, the problem is generally limited to that process alone and does not cause the entire machine to go down.

In this chapter, we explore how the Linux kernel implements its virtual memory system and how it maps to the underlying hardware. This mapping is illustrated specifically for IA-64. The chapter is structured as follows. The first section provides an introduction to the virtual memory system of Linux and establishes the terminology used throughout the remainder of the chapter. The introduction is followed by a description of the software and hardware structures that form the virtual memory system. Specifically, the second section describes the Linux virtual address space and its representation in the kernel. The third section describes the Linux page tables, and the fourth section describes how Linux manages

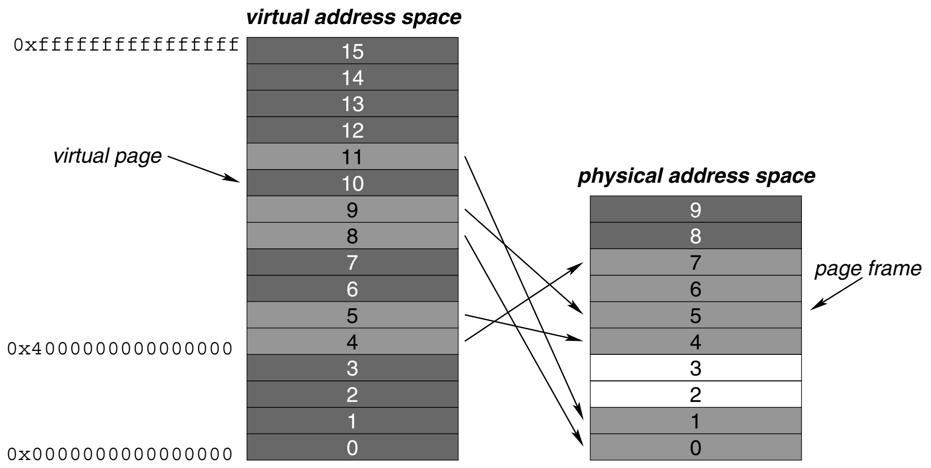


Figure 4.1. Virtual and physical address spaces.

the *translation lookaside buffer (TLB)*, which is a hardware structure used to accelerate virtual memory accesses. Once these fundamental structures are introduced, the chapter describes the operation of the virtual memory system. Section five explores the Linux page fault handler, which can be thought of as the engine driving the virtual memory system. Section six describes how memory coherency is maintained, that is, how Linux ensures that a process sees the correct values in the virtual memory locations it can access. Section seven discusses how Linux switches execution from one address space to another, which is a necessary step during a process context switch. The chapter concludes with section eight, which provides the rationale for some of the virtual memory choices that were made for the virtual memory system implemented on IA-64.

4.1 INTRODUCTION TO THE VIRTUAL MEMORY SYSTEM

The left half of Figure 4.1 illustrates the virtual address space as it might exist for a particular process on a 64-bit platform. As the figure shows, the virtual address space is divided into equal-sized pieces called *virtual pages*. Virtual pages have a fixed size that is an integer power of 2. For example, IA-32 uses a page size of 4 Kbytes. To maximize performance, IA-64 supports multiple page sizes and Linux can be configured to use a size of 4, 8, 16, or 64 Kbytes. In the figure, the 64-bit address space is divided into 16 pages, meaning that each virtual page would have a size of $2^{64}/16 = 2^{60}$ bytes or 1024 Pbytes (1 Pbyte = 2^{50} bytes). Such large pages are not realistic, but the alternative of drawing a figure with several billion pages of a more realistic size is, of course, not practical either. Thus, for this section, we continue to illustrate virtual memory with this huge page size. The figure also shows that virtual pages are numbered sequentially. We can calculate the *virtual page number (VPN)* from a virtual address by dividing it by the page size and taking the integer portion of the result. The remainder is called the *page offset*. For example, dividing virtual address `0x40000000000003f8` by the page size yields 4 and a remainder of `0x3f8`. This address therefore maps to virtual page number 4 and page offset `0x3f8`.

Let us now turn attention to the right half of Figure 4.1, which shows the physical address space. Just like the virtual space, it is divided into equal-sized pieces, but in physical memory, those pieces are called *page frames*. As with virtual pages, page frames also are numbered. We can calculate the *page frame number (PFN)* from a physical address by dividing it by the page frame size and taking the integer portion of the result. The remainder is the *page frame offset*. Normally, page frames have the same size as virtual pages. However, there are cases where it is beneficial to deviate from this rule. Sometimes it is useful to have virtual pages that are larger than a page frame. Such pages are known as *superpages*. Conversely, it is sometimes useful to divide a page frame into multiple, smaller virtual pages. Such pages are known as *subpages*. IA-64 is capable of supporting both, but Linux does not use them.

While it is easiest to think of physical memory as occupying a single contiguous region in the physical address space, in reality it is not uncommon to encounter *memory holes*. Holes usually are caused by one of three entities: firmware, memory-mapped I/O devices, or unpopulated memory. All three cause portions of the physical address space to be unavailable for storing the content of virtual pages. As far as the kernel is concerned, these portions are holes in the physical memory. In the example in Figure 4.1, page frames 2 and 3 represent a hole. Note that even if just a single byte in a page frame is unusable, the entire frame must be marked as a hole.

4.1.1 Virtual-to-physical address translation

Processes are under the illusion of being able to store data to virtual memory and retrieve it later on as if it were stored in real memory. In reality, only physical memory can store data. Thus, each virtual page that is in use must be mapped to some page frame in physical memory. For example, in Figure 4.1, virtual pages 4, 5, 8, 9, and 11 are in use. The arrows indicate which page frame in physical memory they map to. The mapping between virtual pages and page frames is stored in a data structure called the *page table*. The page table for our example is shown on the left-hand side of Figure 4.2.

The Linux kernel is responsible for creating and maintaining page tables but employs the CPU's *memory management unit (MMU)* to translate the virtual memory accesses of a process into corresponding physical memory accesses. Specifically, when a process accesses a memory location at a particular virtual address, the MMU translates this address into the corresponding physical address, which it then uses to access the physical memory. This is illustrated in Figure 4.2 for the case in which the virtual address is `0x40000000000003f8`. As the figure shows, the MMU extracts the VPN (4) from the virtual address and then searches the page table to find the matching PFN. In our case, the search stops at the first entry in the page table since it contains the desired VPN. The PFN associated with this entry is 7. The MMU then constructs the physical address by concatenating the PFN with the frame offset from the virtual address, which results in a physical address of `0x70000000000003f8`.

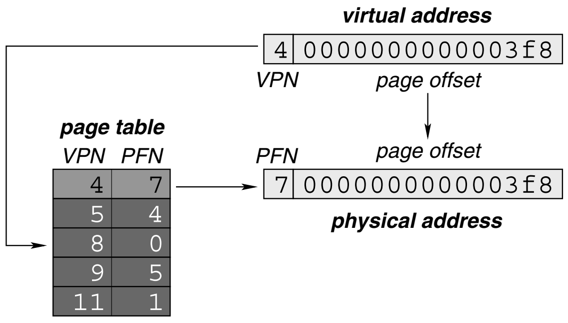


Figure 4.2. Virtual-to-physical address translation.

4.1.2 Demand paging

The next question we need to address is how the page tables get created. Linux could create appropriate page-table entries whenever a range of virtual memory is allocated. However, this would be wasteful because most programs allocate much more virtual memory than they ever use at any given time. For example, the text segment of a program often includes large amounts of error handling code that is seldom executed. To avoid wasting memory on virtual pages that are never accessed, Linux uses a method called *demand paging*. With this method, the virtual address space starts out empty. This means that, logically, all virtual pages are marked in the page table as **not present**. When accessing a virtual page that is not present, the CPU generates a *page fault*. This fault is intercepted by the Linux kernel and causes the page fault handler to be activated. There, the kernel can allocate a new page frame, determine what the content of the accessed page should be (e.g., a new, zeroed page, a page loaded from the data section of a program, or a page loaded from the text segment of a program), load the page, and then update the page table to mark the page as **present**. Execution then resumes in the process with the instruction that caused the fault. Since the required page is now present, the instruction can now execute without causing a page fault.

4.1.3 Paging and swapping

So far, we assumed that physical memory is plentiful: Whenever we needed a page frame to back a virtual page, we assumed a free page frame was available. When a system has many processes or when some processes grow very large, the physical memory can easily fill up. So what is Linux supposed to do when a page frame is needed but physical memory is already full? The answer is that in this case, Linux picks a page frame that backs a virtual page that has not been accessed recently, writes it out to a special area on the disk called the *swap space*, and then reuses the page frame to back the new virtual page. The exact place to which the old page is written on the disk depends on what kind of swap space is in use. Linux can support multiple swap space areas, each of which can be either an entire disk partition or a specially formatted file on an existing filesystem (the former is generally more efficient and therefore preferable). Of course, the page table of the process from which Linux “stole” the page frame must be updated accordingly. Linux does this update by marking the page-table entry as **not present**. To keep track of where the old page has been saved, it also uses the entry to record the disk location of the page. In other words, a page-table entry that is **present** contains the page frame number of the physical page frame that backs the virtual page, whereas a page-table entry that is **not present** contains the disk location at which the content of the page can be found.

Because a page marked as **not present** cannot be accessed without first triggering a page fault, Linux can detect when the page is needed again. When this happens, Linux needs to again find an available page frame (which may cause another page to be paged out), read the page content back from swap space, and then update the page-table entry so that it is marked as **present** and maps to the newly allocated page frame. At this point, the process that attempted to access the paged-out page can be resumed, and, apart from a small delay, it will execute as if the page had been in memory at all along.

The technique of stealing a page from a process and writing it out to disk is called *paging*. A related technique is *swapping*. It is a more aggressive form of paging in the sense that it does not steal an individual page but steals *all* the pages of a process when memory is in short supply. Linux uses paging but not swapping. However, because both paging and swapping write pages to swap space, Linux kernel programmers often use the terms “swapping” and “paging” interchangeably. This is something to keep in mind when perusing the kernel source code.

From a correctness point of view, it does not matter which page is selected for page out, but from a performance perspective, the choice is critical. With a poor choice, Linux may end up paging out the page that is needed in the very next memory access. Given the large difference between disk access latency (on the order of several milliseconds) and memory access latency (on the order of tens of nanoseconds), making the right replacement choices can mean the difference between completing a task in a second or in almost three hours!

The algorithm that determines which page to evict from main memory is called the *replacement policy*. The provably *optimal replacement policy (OPT)* is to choose the page that will be accessed farthest in the future. Of course, in general it is impossible to know the future behavior of the processes, so OPT is of theoretical interest only. A replacement policy that often performs almost as well as OPT yet is realizable is the *least recently used (LRU)* policy. LRU looks into the past instead of the future and selects the page that has not been accessed for the longest period of time. Unfortunately, even though LRU could be implemented, it is still not practical because it would require updating a data structure (such as an LRU list) on *every* access to main memory. In practice, operating systems use approximations of the LRU policy instead, such as the *clock replacement* or *not frequently used (NFU)* policies [11, 69].

In Linux, the page replacement is complicated by the fact that the kernel can take up a variable amount of (nonpageable) memory. For example, file data is stored in the page cache, which can grow and shrink dynamically. When the kernel needs a new page frame, it often has two choices: It could take away a page from the kernel or it could steal a page from a process. In other words, the kernel needs not just a replacement policy but also a *memory balancing policy* that determines how much memory is used for kernel buffers and how much is used to back virtual pages. The combination of page replacement and memory balancing poses a difficult problem for which there is no perfect solution. Consequently, the Linux kernel uses a variety of heuristics that tend to work well in practice.

To implement these heuristics, the Linux kernel expects the platform-specific part of the kernel to maintain two extra bits in each page-table entry: the **accessed** bit and the **dirty** bit. The **accessed** bit is an indicator that tells the kernel whether the page was accessed (read, written, or executed) since the bit was last cleared. Similarly, the **dirty** bit is an indicator that tells whether the page has been modified since it was last paged in. Linux uses a kernel thread, called the kernel swap daemon *kswapd*, to periodically inspect these bits. After inspection, it clears the **accessed** bit. If kswapd detects that the kernel is starting to run low on memory, its starts to proactively page out memory that has not been used recently. If the **dirty** bit of a page is set, it needs to write the page to disk before the page frame can be freed. Because this is relatively costly, kswapd preferentially frees pages whose **accessed** and **dirty** bits are cleared to 0. By definition such pages were not accessed recently and do not have to be written back to disk before the page frame can be freed, so they can be reclaimed at very little cost.

4.1.4 Protection

In a multiuser and multitasking system such as Linux, multiple processes often execute the same program. For example, each user who is logged into the system at a minimum is running a command shell (e.g., the Bourne-Again shell, `bash`). Similarly, server processes such as the Apache web server often use multiple processes running the same program to better handle heavy loads. If we looked at the virtual space of each of those processes, we would find that they share many identical pages. Moreover, many of those pages are never modified during the lifetime of a process because they contain read-only data or the text segment of the program, which also does not change during the course of execution. Clearly, it would make a lot of sense to exploit this commonality and use only one page frame for each virtual page with identical content.

With N processes running the same program, sharing identical pages can reduce physical memory consumption by up to a factor of N. In reality, the savings are usually not quite so dramatic, because each process tends to require a few private pages. A more realistic example is illustrated in Figure 4.3: Page frames 0, 1, and 5 are used to back virtual pages 1, 2, and 3 in the two processes called `bash 1` and `bash 2`. Note that a total of nine virtual pages are in use, but thanks to page sharing, only six page frames are needed.

Of course, page sharing cannot be done safely unless we can guarantee that none of the shared pages are modified. Otherwise, the changes of one process would be visible in all the other processes and that could lead to unpredictable program behavior.

This is where the page *permission bits* come into play. The Linux kernel expects the platform-specific part of the kernel to maintain three such bits per page-table entry. They are called the R, W, and X permission bits and respectively control whether read, write, or execute accesses to the page are permitted. If an access that is not permitted is attempted, a *page protection violation* fault is raised. When this happens, the kernel responds by sending a segmentation violation signal (SIGSEGV) to the process.

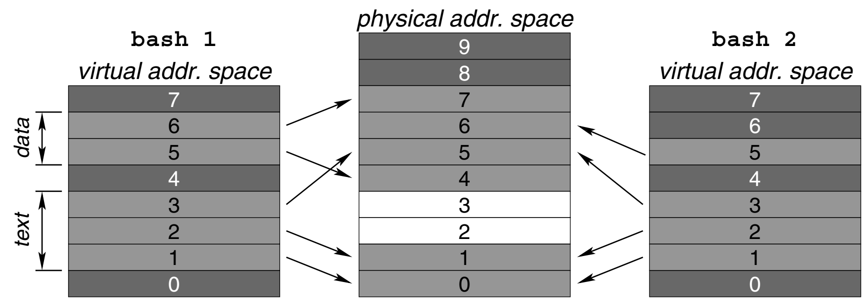


Figure 4.3. Two processes sharing the text segment (virtual pages 1 to 3).

The page permission bits enable the safe sharing of page frames. All the Linux kernel has to do is ensure that all virtual pages that refer to a shared page frame have the W permission bit turned off. That way, if a process attempted to modify a shared page, it would receive a segmentation violation signal before it could do any harm.

The most obvious place where page frame sharing can be used effectively is in the text segment of a program: By definition, this segment can be executed and read, but it is never written to. In other words, the text segment pages of all processes running the same program can be shared. The same applies to read-only data pages.

Linux takes page sharing one step further. When a process forks a copy of itself, the kernel disables write access to *all* virtual pages and sets up the page tables such that the parent and the child process share all page frames. In addition, it marks the pages that were writable before as *copy-on-write (COW)*. If the parent or the child process attempts to write to a copy-on-write page, a protection violation fault occurs. When this happens, instead of sending a segmentation violation signal, the kernel first makes a private copy of the virtual page and then turns the write permission bit for that page back on. At this point, execution can return to the faulting process. Because the page is now writable, the faulting instruction can finish execution without causing a fault again. The copy-on-write scheme is particularly effective when a program does a *fork()* that is quickly followed by an *execve()*. In such a case, the scheme is able to avoid almost all page copying, save for a few pages in the stack- or data-segment [9, 64].

Note that the page sharing described here happens automatically and without the explicit knowledge of the process. There are times when two or more processes need to cooperate and want to explicitly share some virtual memory pages. Linux supports this through the *mmap()* system call and through System V shared memory segments [9]. Because the processes are cooperating, it is their responsibility to map the shared memory segment with suitable permission bits to ensure that the processes can access the memory only in the intended fashion.

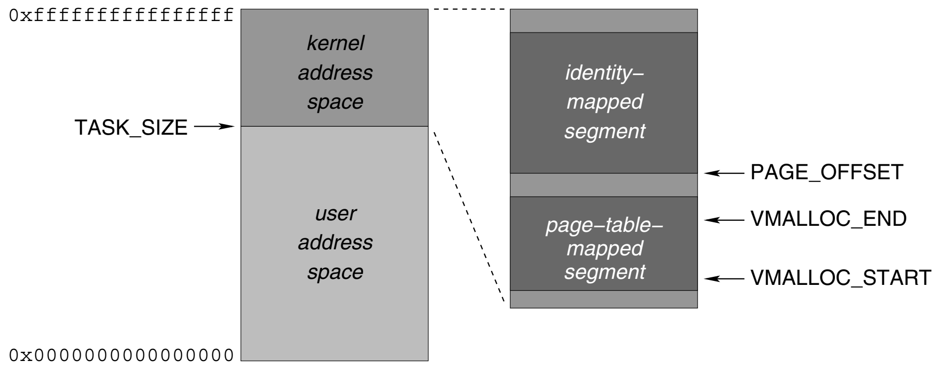


Figure 4.4. Structure of Linux address space.

4.2 ADDRESS SPACE OF A LINUX PROCESS

The virtual address space of any Linux process is divided into two subspaces: kernel space and user space. As illustrated on the left-hand side of Figure 4.4, user space occupies the lower portion of the address space, starting from address 0 and extending up to the platform-specific *task size limit* (TASK_SIZE in file include/asm/processor.h). The remainder is occupied by kernel space. Most platforms use a task size limit that is large enough so that at least half of the available address space is occupied by the user address space.

User space is private to the process, meaning that it is mapped by the process's own page table. In contrast, kernel space is shared across all processes. There are two ways to think about kernel space: We can either think of it as being mapped into the top part of each process, or we can think of it as a single space that occupies the top part of the CPU's virtual address space. Interestingly, depending on the specifics of CPU on which Linux is running, kernel space can be implemented in one or the other way.

During execution at the user level, only user space is accessible. Attempting to read, write, or execute kernel space would cause a protection violation fault. This prevents a faulty or malicious user process from corrupting the kernel. In contrast, during execution in the kernel, both user and kernel spaces are accessible.

Before continuing our discussion, we need to say a few words about the page size used by the Linux kernel. Because different platforms have different constraints on what page sizes they can support, Linux never assumes a particular page size and instead uses the platform-specific *page size constant* (PAGE_SIZE in file include/asm/page.h) where necessary. Although Linux can accommodate arbitrary page sizes, throughout the rest of this chapter we assume a page size of 8 Kbytes, unless stated otherwise. This assumption helps to make the discussion more concrete and avoids excessive complexity in the following examples and figures.

4.2.1 User address space

Let us now take a closer look at how Linux implements the user address spaces. Each address space is represented in the kernel by an object called the *mm structure* (struct mm_struct in file include/linux/sched.h). As we have seen in Chapter 3, *Processes, Tasks, and Threads*, multiple tasks can share the same address space, so the mm structure is a reference-counted object that exists as long as at least one task is using the address space represented by the mm structure. Each task structure has a pointer to the mm structure that defines the address space of the task. This pointer is known as the *mm pointer*. As a special case, tasks that are known to access kernel space only (such as kswapd) are said to have an *anonymous address space*, and the mm pointer of such tasks is set to NULL. When switching execution to such a task, Linux does not switch the address space (because there is none to switch to) and instead leaves the old one in place. A separate pointer in the task structure tracks which address space has been borrowed in this fashion. This pointer is known as the *active mm pointer* of the task. For a task that is currently running, this pointer is guaranteed not to be NULL. If the task has its own address space, the active mm pointer has the same value as the mm pointer; otherwise, the active mm pointer refers to the mm structure of the borrowed address space.

Perhaps somewhat surprisingly, the mm structure itself is not a terribly interesting object. However, it is a central hub in the sense that it contains the pointers to the two data structures that are at the core of the virtual memory system: the page table and the list of virtual memory areas, which we describe next. Apart from these two pointers, the mm structure contains miscellaneous information, such as the mm context, which we describe in more detail in Section 4.4.3, a count of the number of virtual pages currently in use (the *resident set size*, or *RSS*), the start and end address of the text, data, and stack segments as well as housekeeping information that kswapd uses when looking for virtual memory to page out.

Virtual memory areas

In theory, a page table is all the kernel needs to implement virtual memory. However, page tables are not effective in representing huge address spaces, especially when they are sparse. To see this, let us assume that a process uses 1 Gbytes of its address space for a hash table and then enters 128 Kbytes of data in it. If we assume that the page size is 8 Kbytes and that each entry in the page table takes up 8 bytes, then the page table itself would take up $1 \text{ Gbyte}/8 \text{ Kbytes} \cdot 8 \text{ byte} = 1 \text{ Mbyte}$ of space—an order of magnitude more than the actual data stored in the hash table!

To avoid this kind of inefficiency, Linux does not represent address spaces with page tables. Instead, it uses lists of *vm-area structures* (struct vm_area_struct in file include/-linux/mm.h). The idea is to divide an address space into contiguous ranges of pages that can be handled in the same fashion. Each range can then be represented by a single vm-area structure. If a process accesses a page for which there is no translation in the page table, the vm-area covering that page has all the information needed to install the missing page. For our hash table example, this means that a single vm-area would suffice to map the entire hash table and that page-table memory would be needed only for recently accessed pages.

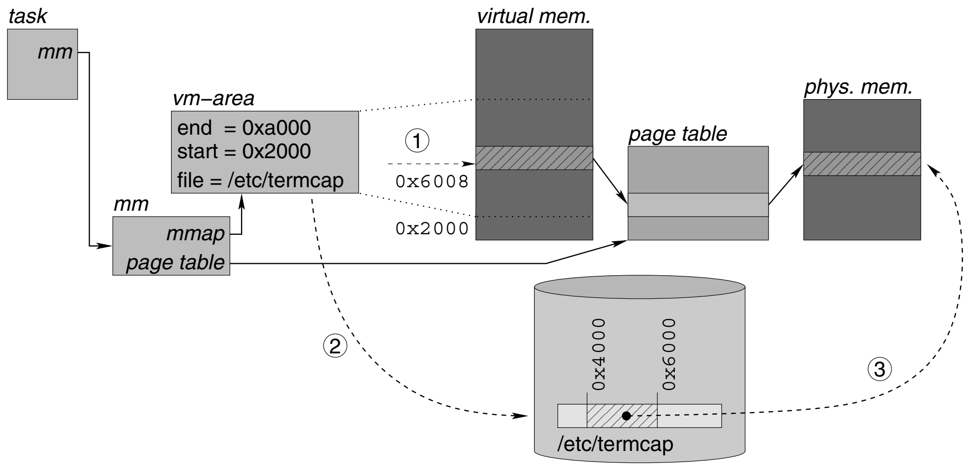


Figure 4.5. Example: vm-area mapping a file.

To get a better sense of how the kernel uses vm-areas, let us consider the example in Figure 4.5. It shows a process that maps the first 32 Kbytes (four pages) of the file /etc/termcap at virtual address `0x2000`. At the top-left of the figure, we find the task structure of the process and the mm pointer that leads to the mm structure representing the address space of the process. From there, the mmap pointer leads to the first element in the vm-area list. For simplicity, we assume that the vm-area for the mapped file is the only one in this process, so this list contains just one entry. The mm structure also has a pointer to the page table, which is initially empty. Apart from these kernel data structures, the process's virtual memory is shown in the middle of the figure, the filesystem containing /etc/termcap is represented by the disk-shaped form, and the physical memory is shown on the right-hand side of the figure.

Now, suppose the process attempts to read the word at address `0x6008`, as shown by the arrow labeled (1). Because the page table is empty, this attempt results in a page fault. In response to this fault, Linux searches the vm-area list of the current process for a vm-area that covers the faulting address. In our case, it finds that the one and only vm-area on the list maps the address range from `0x2000` to `0xa000` and hence covers the faulting address. By calculating the distance from the start of the mapped area, Linux finds that the process attempted to access page 2 ($\lfloor \texttt{0x6008} - \texttt{0x2000}/8192 \rfloor = 2$). Because the vm-area maps a file, Linux initiates the disk read illustrated by the arrow labeled (2). We assumed that the vm-area maps the first 32KB of the file, so the data for page 2 can be found at file offsets `0x4000` through `0x5fff`. When this data arrives, Linux copies it to an available page frame as illustrated by the arrow labeled (3). In the last step, Linux updates the page table with an entry that maps the virtual page at `0x6000` to the physical page frame that now contains the file data. At this point, the process can resume execution. The read access will be restarted and will now complete successfully, returning the desired file data.

As this example illustrates, the vm-area list provides Linux with the ability to (re-)create the page-table entry for any address that is mapped in the address space of a process. This implies that the page table can be treated almost like a cache: If the translation for a particular page is present, the kernel can go ahead and use it, and if it is missing, it can be created from the matching vm-area. Treating the page table in this fashion provides a tremendous amount of flexibility because translations for clean pages can be removed at will. Translations for dirty pages can be removed only if they are backed by a file (not by swap space). Before removal, they have to be cleaned by writing the page content back to the file. As we see later, the cache-like behavior of page tables provides the foundation for the copy-on-write algorithm that Linux uses.

AVL trees

As we have seen so far, the vm-area list helps Linux avoid many of the inefficiencies of a system that is based entirely on page tables. However, there is still a problem. If a process maps many different files into its address space, it may end up with a vm-area list that is hundreds or perhaps even thousands of entries long. As this list grows longer, the kernel executes more and more slowly as each page fault requires the kernel to traverse this list. To ameliorate this problem, the kernel tracks the number of vm-areas on the list, and if there are too many, it creates a secondary data structure that organizes the vm-areas as an AVL tree [42, 62]. An AVL tree is a normal binary search tree, except that it has the special property that for each node in the tree, the height of the two subtrees differs by at most 1. Using the standard tree-search algorithm, this property ensures that, given a virtual address, the matching vm-area structure can be found in a number of steps that grows only with the logarithm of the number of vm-areas in the address space.[1]

Let us consider a concrete example. Figure 4.6 show the AVL tree for an Emacs process as it existed right after it was started up on a Linux/ia64 machine. For space reasons, the figure represents each node with a rectangle that contains just the starting and ending address of the address range covered by the vm-area. As customary for a search tree, the vm-area nodes appear in the order of increasing starting address. Given a node with a starting address of *x*, the vm-areas with a lower starting address can be found in the lower ("left") subtree and the vm-areas with a higher starting address can be found in the higher ("right") subtree. The root of the tree is at the left end of the figure, and, as indicated by the arrows, the tree grows toward the right side. While it is somewhat unusual for a tree to grow from left to right, this representation has the advantage that the higher a node in the figure, the higher its starting address.

First, observe that this tree is not perfectly balanced: It has a height of six, yet there is a missing node at the fifth level as illustrated by the dashed rectangle. Despite this imperfection, the tree does have the AVL property that the height of the subtrees at any node never differs by more than one. Second, note that the tree contains 47 vm-areas. If we were to use a linear search to find the vm-area for a given address, we would have to visit 23.5 vm-area structures on average and, in the worst case, we might have to visit all 47 of them. In con-

[1]In Linux v2.4.10 Andrea Arcangeli replaced AVL trees with Red-Black trees [62]. Red-Black trees are also balanced, but can be implemented more efficiently.

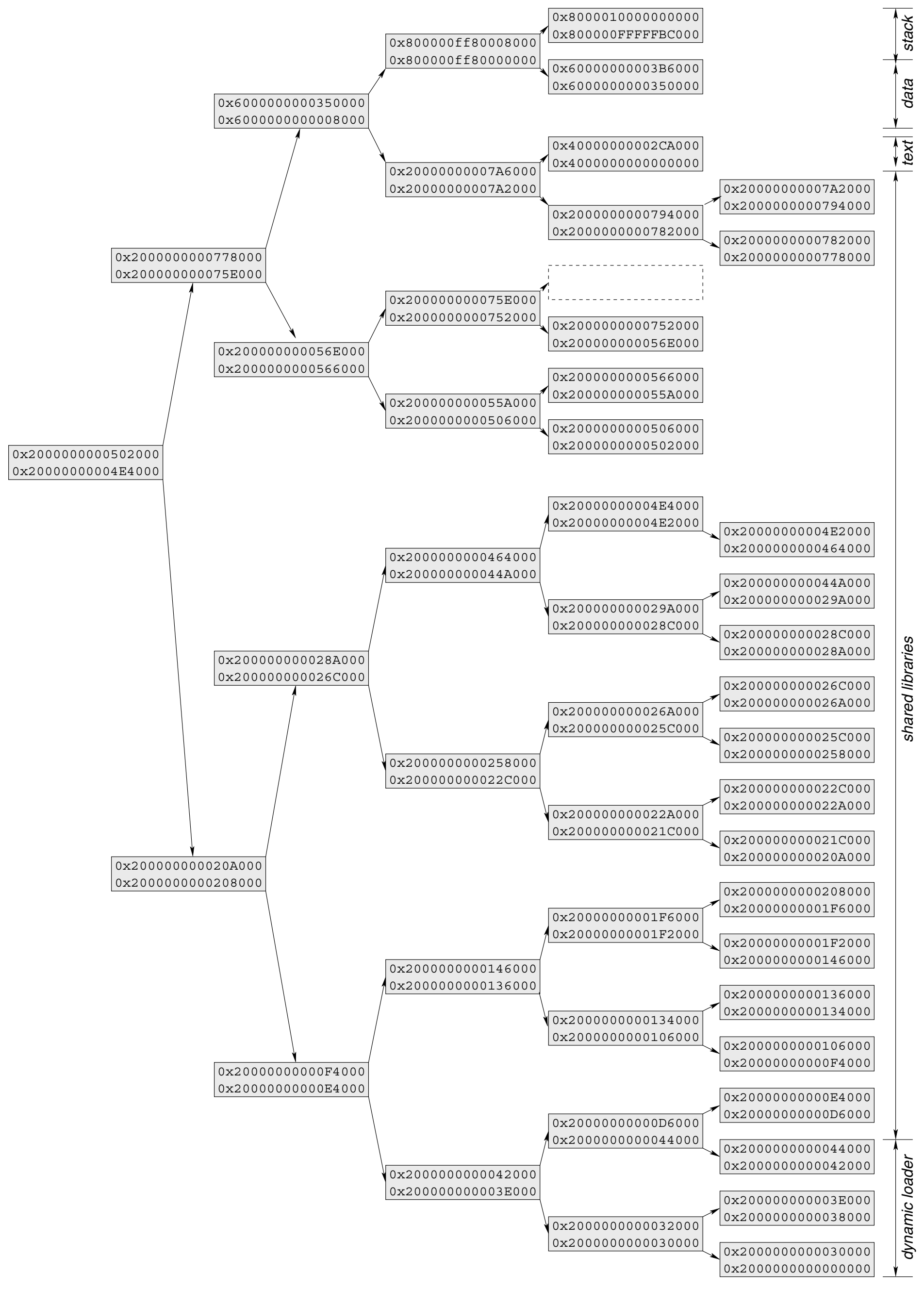


Figure 4.6. AVL tree of vm-area structures for a process running Emacs.

trast, when the AVL tree is searched, at most six vm-areas have to be visited, as given by the height of the tree. Clearly, using an AVL tree is a big win for complex address spaces. However, for simple address spaces, the overhead of creating the AVL tree and keeping it balanced is too much compared to the cost of searching a short linear list. For this reason, Linux does not create the AVL tree until the address space contains at least 32 vm-areas. Let us emphasize that even when the AVL tree is being maintained, the linear list continues to be maintained as well; this provides an efficient means to visit *all* vm-area structures.

Anatomy of the vm-area structure

So far, we discussed the purpose of the vm-area structure and how the Linux kernel uses it, but not what it looks like. The list below rectifies this situation by describing the major components of the vm-area:

- **Address range:** Describes the address range covered by the vm-area in the form of a start and end address. It is noteworthy that the end address is the address of the first byte that is *not* covered by the vm-area.
- **VM flags:** Consist of a single word that contains various flag bits. The most important among them are the access right flags VM_READ, VM_WRITE, and VM_EXEC, which control whether the process can, respectively, read, write, or execute the virtual memory mapped by the vm-area. Two other important flags are VM_GROWSDOWN and VM_GROWSUP, which control whether the address range covered by the vm-area can be extended toward lower or higher addresses, respectively. As we see later, this provides the means to grow user stacks dynamically.
- **Linkage info:** Contain various linkage information, including the pointer needed for the mm structure's vm-area list, pointers to the left and right subtrees of the AVL tree, and a pointer that leads back to the mm structure to which the vm-area belongs.
- **VM operations and private data:** Contain the *VM operations pointer*, which is a pointer to a set of callback functions that define how various virtual-memory-related events, such as page faults, are to be handled. The component also contains a private data pointer that can be used by the callback functions as a hook to maintain information that is vm-area–specific.
- **Mapped file info:** If a vm-area maps a portion of a file, this component stores the file pointer and the file offset needed to locate the file data.

Note that the vm-area structure is not reference-counted. There is no need to do that because each structure belongs to one and only one mm structure, which is already reference-counted. In other words, when the reference-count of an mm structure reaches 0, it is clear that the vm-area structures owned by it are also no longer needed.

A second point worth making is that the VM operations pointer gives the vm-area characteristics that are object-like because different types of vm-areas can have different handlers for responding to virtual-memory-related events. Indeed, Linux allows each filesystem, character device, and, more generally, any object that can be mapped into user space by *mmap()* to provide its own set VM operations. The operations that can be provided in this fashion are *open()*, *close()*, and *nopage()*. The *open()* and *close()* callbacks are invoked whenever a vm-area is created or destroyed, respectively, and is used primarily to keep track of the number of vm-areas that are currently using the underlying object. The *nopage()* callback is invoked when a page fault occurs for an address for which there is no page-table entry. The Linux kernel provides default implementations for each of these callbacks. These default versions are used if either the VM operations pointer or a particular callback pointer is NULL. For example, if the *nopage()* callback is NULL, Linux handles the page fault by creating an *anonymous page*, which is a process-private page whose content is initially cleared to 0.

4.2.2 Page-table-mapped kernel segment

Let us now return to Figure 4.4 and take a closer look at the kernel address space. The right-hand side of this figure is an enlargement of the kernel space and shows that it contains two segments: the identity-mapped segment and the page-table-mapped segment. The latter is mapped by a kernel-private page table and is used primarily to implement the kernel *vmalloc arena* (file include/linux/vmalloc.h). The kernel uses this arena to allocate large blocks of memory that must be contiguous in virtual space. For example, the memory required to load a kernel module is allocated from this arena. The address range occupied by the vmalloc arena is defined by the platform-specific constants VMALLOC_START and VMALLOC_END. As indicated in the figure, the vmalloc arena does not necessarily occupy the entire page-table-mapped segment. This makes it possible to use part of the segment for platform-specific purposes.

4.2.3 Identity-mapped kernel segment

The identity-mapped segment starts at the address defined by the platform-specific constant PAGE_OFFSET. This segment contains the Linux kernel image, including its text, data, and stack segments. In other words, this is the segment that the kernel is executing in when in kernel mode (unless when executing in a module).

The identity-mapped segment is special because there is a direct mapping between a virtual address in this segment and the physical address that it translates to. The exact formula for this mapping is platform specific, but it is often as simple as *vaddr* − PAGE_OFFSET. This one-to-one (identity) relationship between virtual and physical addresses is what gives the segment its name.

The segment could be implemented with a normal page table. However, because there is a direct relationship between virtual and physical addresses, many platforms can optimize this case and avoid the overhead of a page table. How this is done on IA-64 is described in Section 4.5.3.

Because the actual formula to translate between a physical address and the equivalent virtual address is platform specific, the kernel uses the interface in Figure 4.7 to perform such translations. The interface provides two routines: *__pa()* expects a single argument, *vaddr*, and returns the physical address that corresponds to *vaddr*. The return value is un-

unsigned long **__pa**(*vaddr*);	*/* translate virtual address to physical address */*
void ***__va**(*paddr*);	*/* translate physical address to virtual address */*

Figure 4.7. Kernel interface to convert between physical and virtual addresses.

defined if *vaddr* does not point inside the kernel's identity-mapped segment. Routine *__va()* provides the reverse mapping: it takes a physical address *paddr* and returns the corresponding virtual address. Usually the Linux kernel expects virtual addresses to have a pointer-type (such as **void ***) and physical addresses to have a type of **unsigned long**. However, the *__pa()* and *__va()* macros are polymorphic and accept arguments of either type.

A platform is free to employ an arbitrary mapping between physical and virtual addresses provided that the following relationships are true:

$$\begin{aligned} \_\_va(\_\_pa(vaddr)) &= vaddr \quad \text{for all } vaddr \text{ inside the identity-mapped segment} \\ paddr_1 < paddr_2 &\Rightarrow \_\_va(paddr_1) < \_\_va(paddr_2) \end{aligned}$$

That is, mapping any virtual address inside the identity-mapped segment to a physical address and back must return the original virtual address. The second condition is that the mapping must be monotonic, i.e., the relative order of a pair of physical addresses is preserved when they are mapped to virtual addresses.

We might wonder why the constant that marks the beginning of the identity-mapped segment is called PAGE_OFFSET. The reason is that the page frame number *pfn* for an address *addr* in this segment can be calculated as:

$$pfn = (addr - \mathsf{PAGE\_OFFSET})/\mathsf{PAGE\_SIZE}$$

As we will see next, even though the page frame number is easy to calculate, the Linux kernel does not use it very often.

Page frame map

Linux uses a table called the *page frame map* to keep track of the status of the physical page frames in a machine. For each page frame, this table contains exactly one *page frame descriptor* (struct page in file include/linux/mm.h). This descriptor contains various housekeeping information, such as a count of the number of address spaces that are using the page frame, various flags that indicate whether the frame can be paged out to disk, whether it has been accessed recently, or whether it is dirty (has been written to), and so on.

While the exact content of the page frame descriptor is of no concern for this chapter, we do need to understand that Linux often uses page frame descriptor pointers in lieu of page frame numbers. The Linux kernel leaves it to platform-specific code how virtual addresses in the identity-mapped segment are translated to page frame descriptor pointers, and vice versa. It uses the interface shown in Figure 4.8 for this purpose.

Because we are not concerned with the internals of the page frame descriptor, Figure 4.8 lists its type (struct page) simply as an opaque structure. The *virt_to_page()* routine can be

struct **page**;	*/* page frame descriptor */*
struct page ***virt_to_page**(*vaddr*);	*/* return page frame descriptor for vaddr */*
void ***page_address**(*page*);	*/* return virtual address for page */*

Figure 4.8. Kernel interface to convert between pages and virtual addresses.

used to obtain the page frame descriptor pointer for a given virtual address. It expects one argument, *vaddr*, which must be an address inside the identity-mapped segment, and returns a pointer to the corresponding page frame descriptor. The *page_address()* routine provides the reverse mapping: It expects the *page* argument to be a pointer to a page frame descriptor and returns the virtual address inside the identity-mapped segment that maps the corresponding page frame.

Historically, the page frame map was implemented with a single array of page frame descriptors. This array was called *mem_map* and was indexed by the page frame number. In other words, the value returned by *virt_to_page()* could be calculated as:

$$\&mem\_map[(addr - \mathsf{PAGE\_OFFSET})/\mathsf{PAGE\_SIZE}]$$

However, on machines with a physical address space that is either fragmented or has huge holes, using a single array can be problematic. In such cases, it is better to implement the page frame map by using multiple partial maps (e.g., one map for each set of physically contiguous page frames). The interface in Figure 4.8 provides the flexibility necessary for platform-specific code to implement such solutions, and for this reason the Linux kernel no longer uses the above formula directly.

High memory support

The size of the physical address space has no direct relationship to the size of the virtual address space. It could be smaller than, the same size as, or even larger than the virtual space. On a new architecture, the virtual address space is usually designed to be much larger than the largest anticipated physical address space. Not surprisingly, this is the case for which Linux is designed and optimized.

However, the size of the physical address space tends to increase roughly in line with Moore's Law, which predicts a doubling of chip capacity every 18 months [57]. Because the virtual address space is part of an architecture, its size cannot be changed easily (e.g., changing it would at the very least require recompilation of all applications). Thus, over the course of many years, the size of the physical address space tends to encroach on the size of the virtual address space until, eventually, it becomes as large as or larger than the virtual space.

This is a problem for Linux because once the physical memory has a size similar to that of the virtual space, the identity-mapped segment may no longer be large enough to map the entire physical space. For example, the IA-32 architecture defines an extension that supports a 36-bit physical address space even though the virtual address space has only 32 bits. Clearly, the physical address space cannot fit inside the virtual address space.

unsigned long **kmap**(*page*);	*/* map page frame into virtual space */*
kunmap(*page*);	*/* unmap page frame from virtual space */*

Figure 4.9. Primary routines for the highmem interface.

The Linux kernel alleviates this problem through the *highmem interface* (file include/-linux/highmem.h). *High memory* is physical memory that cannot be addressed through the identity-mapped segment. The highmem interface provides indirect access to this memory by dynamically mapping high memory pages into a small portion of the kernel address space that is reserved for this purpose. This part of the kernel address space is known as the *kmap segment*.

Figure 4.9 shows the two primary routines provided by the highmem interface: *kmap()* maps the page frame specified by argument *page* into the kmap segment. The argument must be a pointer to the page frame descriptor of the page to be mapped. The routine returns the virtual address at which the page was mapped. If the kmap segment is full at the time this routine is called, it will block until space becomes available. This implies that high memory cannot be used in interrupt handlers or any other code that cannot block execution for an indefinite amount of time. Both high and normal memory pages can be mapped with this routine, though in the latter case *kmap()* simply returns the appropriate address in the identity-mapped segment.

When the kernel has finished using a high memory page, it unmaps the page by a call to *kunmap()*. The *page* argument passed to this routine is a pointer to the page frame descriptor of the page that is to be unmapped. Unmapping a page frees up the virtual address space that the page occupied in the kmap segment. This space then becomes available for use by other mappings. To reduce the amount of blocking resulting from a full kmap segment, Linux attempts to minimize the amount of time that high memory pages are mapped.

Clearly, supporting high memory incurs extra overhead and limitations in the kernel and should be avoided where possible. For this reason, high memory support is an optional component of the Linux kernel. Because IA-64 affords a vastly larger virtual address space than that provided by 32-bit architectures, high memory support is not needed and therefore disabled in Linux/ia64. However, it should be noted that the highmem *interface* is available even on platforms that do not provide high memory support. On those platforms, *kmap()* is equivalent to *page_address()* and *kunmap()* performs no operation. These dummy implementations greatly simplify writing platform-independent kernel code. Indeed, it is good kernel programming practice to use the *kmap()* and *kunmap()* routines whenever possible. Doing so results in more efficient memory use on platforms that need high memory support (such as IA-32) without impacting the platforms that do not need it (such as IA-64).

Summary

Figure 4.10 summarizes the relationship between physical memory and kernel virtual space for a hypothetical machine that has high memory support enabled. In this machine, the identity-mapped segment can map only the first seven page frames of the physical address

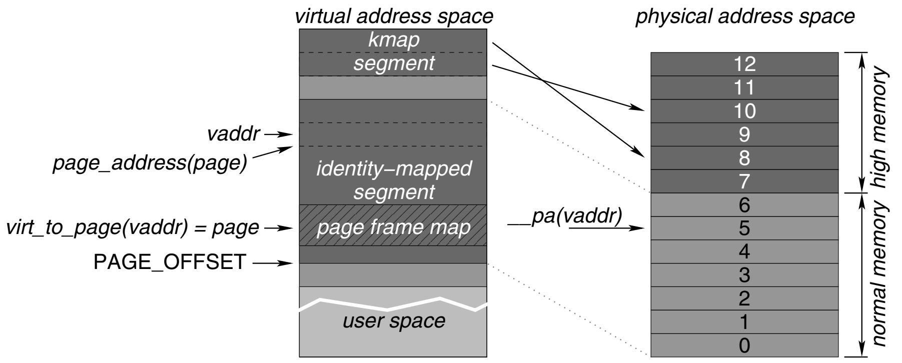


Figure 4.10. Summary of identity-mapped segment and high memory support.

space—the remaining memory consisting of page frames 7 through 12 is high memory and can be accessed through the kmap segment only. The figure illustrates the case in which page frames 8 and 10 have been mapped into this segment. Because our hypothetical machine has a kmap segment that consists of only two pages, the two mappings use up all available space. Trying to map an additional high memory page frame by calling *kmap()* would block the caller until page frame 8 or 10 is unmapped by a call to *kunmap()*.

Let us now turn attention to the arrow labeled *vaddr*. It points to the middle of the second-last page mapped by the identity-mapped segment. We can find the physical address of *vaddr* with the *__pa()* routine. As the arrow labeled *__pa(vaddr)* illustrates, this physical address not surprisingly points to the middle of page frame 5 (the second-to-last page frame in normal memory).

The figure illustrates the page frame map as the diagonally shaded area inside the identity-mapped segment (we assume that our hypothetical machine uses a single contiguous table for this purpose). Note that this table contains page frame descriptors for *all* page frames in the machine, including the high memory page frames. To get more information on the status of page frame 5, we can use *virt_to_page(vaddr)* to get the *page* pointer for the page frame descriptor of that page. This is illustrated in the figure by the arrow labeled *page*. Conversely, we can use the *page* pointer to calculate *page_address(page)* to obtain the starting address of the virtual page that contains *vaddr*.

4.2.4 Structure of IA-64 address space

The IA-64 architecture provides a full 64-bit virtual address space. As illustrated in Figure 4.11, the address space is divided into eight *regions* of equal size. Each region covers 2^{61} bytes or 2048 Pbytes. Regions are numbered from 0 to 7 according to the top three bits of the address range they cover. The IA-64 architecture has no a priori restrictions on how these regions can be used. However, Linux/ia64 uses regions 0 through 4 as the user address space and regions 5 through 7 as the kernel address space.

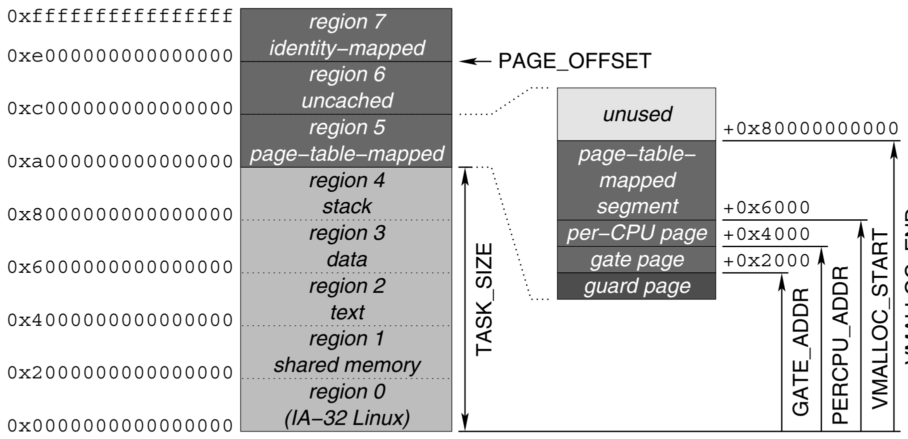


Figure 4.11. Structure of Linux/ia64 address space.

There are also no restrictions on how a process can use the five regions that map the user space, but the usage illustrated in the figure is typical: Region 1 is used for shared memory segments and shared libraries, region 2 maps the text segment, region 3 the data segment, and region 4 the memory and register stacks of a process. Region 0 normally remains unused by 64-bit applications but is available for emulating a 32-bit operating system such as IA-32 Linux.

In the kernel space, the figure shows that the identity-mapped segment is implemented in region 7 and that region 5 is used for the page-table mapped segment. Region 6 is identity-mapped like region 7, but the difference is that accesses through region 6 are not cached. As we discuss in Chapter 7, *Device I/O*, this provides a simple and efficient means for memory-mapped I/O.

The right half of Figure 4.11 provides additional detail on the anatomy of region 5. As illustrated there, the first page is the *guard page*. It is guaranteed *not* to be mapped so that any access is guaranteed to result in a page fault. As we see in Chapter 5, *Kernel Entry and Exit*, this page is used to accelerate the permission checks required when data is copied across the user/kernel boundary. The second page in this region serves as the *gate page*. It assists in transitioning from the user to the kernel level, and vice versa. For instance, as we also see in Chapter 5, this page is used when a signal is delivered and could also be used for certain system calls. The third page is called the *per-CPU page*. It provides one page of CPU-local data, which is useful on MP machines. We discuss this page in more detail in Chapter 8, *Symmetric Multiprocessing*. The remainder of region 5 is used as the vmalloc arena and spans the address range from VMALLOC_START to VMALLOC_END. The exact values of these platform-specific constants depend on the page size. As customary in this chapter, the figure illustrates the case in which a page size of 8 Kbytes is in effect.

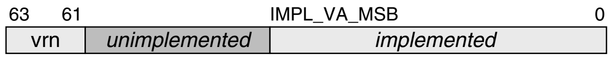


Figure 4.12. Format of IA-64 virtual address.

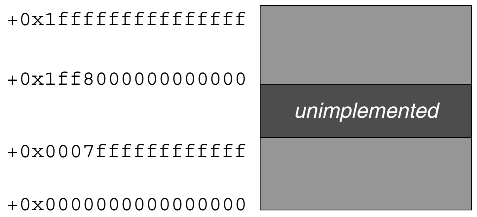


Figure 4.13. Address-space hole within a region with IMPL_VA_MSB $= 50$.

Virtual address format

Even though IA-64 defines a 64-bit address space, implementations are not required to fully support each address bit. Specifically, the virtual address format mandated by the architecture is illustrated in Figure 4.12. As shown in the figure, bits 61 through 63 must be implemented because they are used to select the virtual region number (vrn).

The lower portion of the virtual address consists of a CPU-model-specific number of bits. The most significant bit is identified by constant IMPL_VA_MSB. This value must be in the range of 50 to 60. For example, on Itanium this constant has a value of 50, meaning that the lower portion of the virtual address consists of 51 bits.

The unimplemented portion of the virtual address consists of bits IMPL_VA_MSB $+ 1$ through 60. Even though they are marked as **unimplemented**, the architecture requires that the value in these bits match the value in bit IMPL_VA_MSB. In other words, the unimplemented bits must correspond to the sign-extended value of the lower portion of the virtual address. This restriction has been put in place to ensure that software does not abuse unimplemented bits for purposes such as type tag bits. Otherwise, such software might break when running on a machine that implements a larger number of virtual address bits.

On implementations where IMPL_VA_MSB is less than 60, this sign extension has the effect of dividing the virtual address space within a region into two disjoint areas. Figure 4.13 illustrates this for the case in which IMPL_VA_MSB $= 50$: The sign extension creates the unimplemented area in the middle of the region. Any access to that area will cause the CPU to take a fault. For a user-level access, such a fault is normally translated into an illegal instruction signal (SIGILL). At the kernel level, such an access would cause a kernel panic.

Although an address-space hole in the middle of a region may seem problematic, it really poses no particular problem and in fact provides an elegant way to leave room for future growth without impacting existing application-level software. To see this, consider an application that requires a huge data heap. If the heap is placed in the lower portion of the

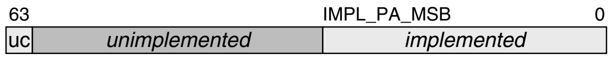


Figure 4.14. Format of IA-64 physical address.

region, it can grow toward higher addresses. On a CPU with IMPL_VA_MSB $= 50$, the heap could grow to at most 1024 Tbytes. However, when the same application is run on a CPU with IMPL_VA_MSB $= 51$, the heap could now grow up to 2048 Tbytes—without changing its starting address. Similarly, data structures that grow toward lower addresses (such as the memory stack) can be placed in the upper portion of the region and can then grow toward the CPU-model-specific lower bound of the implemented address space. Again, the application can run on different implementations and take advantage of the available address space without moving the starting point of the data structure.

Of course, an address-space hole in the middle of a region does imply that an application must not, e.g., attempt to sequentially access all possible virtual addresses in a region. Given how large a region is, this operation would not be a good idea at any rate and so is not a problem in practice.

Physical address space

The physical address format used by IA-64 is illustrated in Figure 4.14. Like virtual addresses, physical addresses are 64 bits wide. However, bit 63 is the uc bit and serves a special purpose: If 0, it indicates a cacheable memory access; if 1, it indicates an uncacheable access. The remaining bits in a physical address are split into two portions: implemented and unimplemented bits. As the figure shows, the lower portion must be implemented and covers bits 0 up to a CPU-model-specific bit number called IMPL_PA_MSB. The architecture requires this constant to be in the range of 32 to 62. For example, Itanium implements 44 address bits and therefore IMPL_PA_MSB is 43. The unimplemented portion of a physical address extends from bit IMPL_PA_MSB $+ 1$ to 62. Unlike a virtual address, a valid physical address must have all unimplemented bits cleared to 0 (i.e., the unimplemented portion is the zero-extended instead of the sign-extended value of the implemented portion).

The physical address format gives rise to the physical address space illustrated in Figure 4.15. As determined by the uc bit, it is divided into two halves: The lower half is the cached physical address space and the upper half is the uncached space. Note that physical addresses x and $2^{63} + x$ correspond to the same memory location—the only difference is that an access to the latter address will bypass all caches. In other words, the two halves alias each other.

If IMPL_PA_MSB is smaller than 62, the upper portion of each half is unimplemented. Any attempt to access memory in this portion of the physical address space causes the CPU to take an UNIMPLEMENTED DATA ADDRESS FAULT.

Recall from Figure 4.11 on page 149 that Linux/ia64 employs a single region for the identity-mapped segment. Because a region spans 61 address bits, Linux can handle IMPL-_PA_MSB values of up to 60 before the region fills up and high memory support needs to

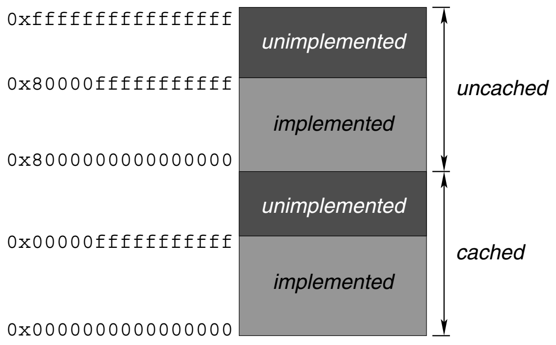


Figure 4.15. Physical address space with IMPL_PA_MSB = 43.

be enabled. To get a back-of-the-envelope estimate of how quickly this could happen, let us assume that at the inception of IA-64 the maximum practical physical memory size was 1 Tbytes (2^{40} bytes). Furthermore, let us assume that memory capacity doubles roughly every 18 months. Both assumptions are somewhat on the aggressive side. Even so, more than three decades would have to pass before high memory support would have to be enabled. In other words, it is likely that high memory support will not be necessary during most of the life span, or even the entire life span, of the IA-64 architecture.

4.3 PAGE TABLES

Linux maintains the page table of each process in physical memory and accesses a page table through the identity-mapped kernel segment. Because they are stored in physical memory, page tables themselves cannot be swapped out to disk. This means that a process with a huge virtual address space could run out of memory simply because the page table alone uses up all available memory. Similarly, if thousands of processes are running, the page tables could take up most or even all of the available memory, making it impossible for the processes to run efficiently. However, on modern machines, main memory is usually large enough to make these issues either theoretical or at most second-order. On the positive side, keeping the page tables in physical memory greatly simplifies kernel design because there is no possibility that handling a page fault would cause another (nested) page fault.

Each page table is represented as a multiway tree. Logically, the tree has three levels, as illustrated in Figure 4.16. As customary in this chapter, this figure shows the tree growing from left to right: at the first level (leftmost part of the figure), we find the *global directory* (*pgd*); at the second level (to the right of the global directory), we find the *middle directories* (*pmd*); and at the third level we find the *PTE directories*. Typically, each directory (node in the tree) occupies one page frame and contains a fixed number of entries. Entries in the global and middle directories are either **not present** or they point to a directory in the next level of the tree. The PTE directories form the leaves of the tree, and its entries consist of *page-table entries (PTEs)*.

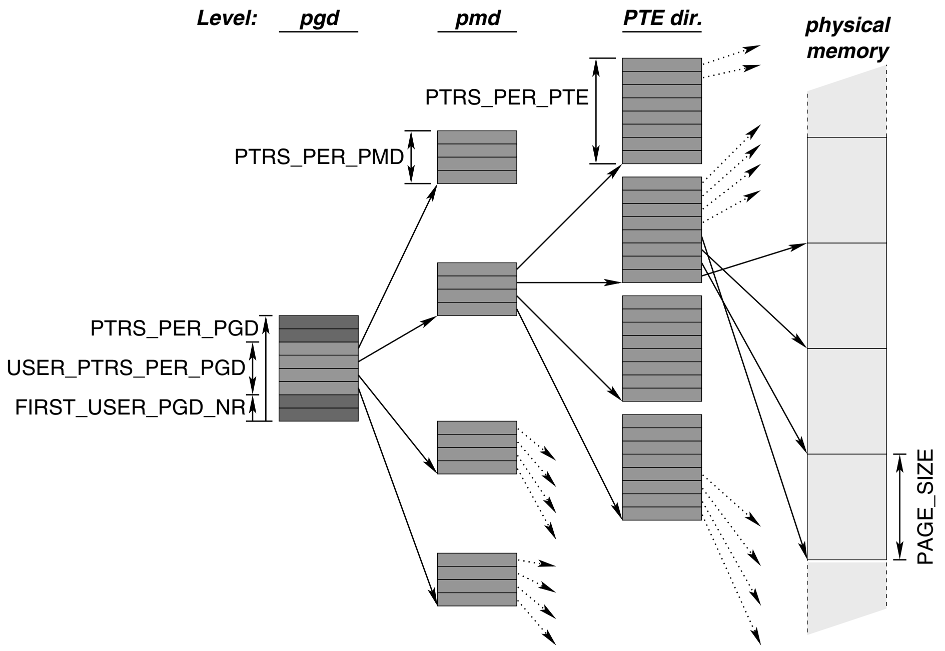


Figure 4.16. Linux page-table tree.

The primary benefit of implementing a page table with a multiway tree instead of a linear array is that the former takes up space that is proportional only to the virtual address space actually *in use*, instead of being proportional to the *maximum* size of the virtual address space. To see this, consider that with a 1-Gbyte virtual address space and 8-Kbyte pages, a linear page table would need storage for more than 131,000 PTEs, even if not a single virtual page was actually in use. In contrast, with a multiway tree, an empty address space requires only a global directory whose entries are all marked **not present**.

Another benefit of multiway trees is that each node (directory) in the tree has a fixed-size (usually a page). This makes it unnecessary to reserve large, physically contiguous regions of memory as would be required for a linear page table. Also, because physical memory is managed as a set of page frames anyhow, the fixed node size makes it easy to build a multiway tree incrementally, as is normally done with demand paging.

Going back to Figure 4.16, we see that the size and structure of the directories is controlled by platform-specific constants. The number of entries in the global, middle, and PTE directories are given by constants PTRS_PER_PGD, PTRS_PER_PMD, and PTRS-_PER_PTE, respectively. The global directory is special because often only part of it can be used to map user space (the rest is either reserved or used for kernel purposes). Two additional parameters define the portion of the global directory that is available for user space. Specifically, FIRST_USER_PGD_NR is the index of the first entry, and USER_PTRS-

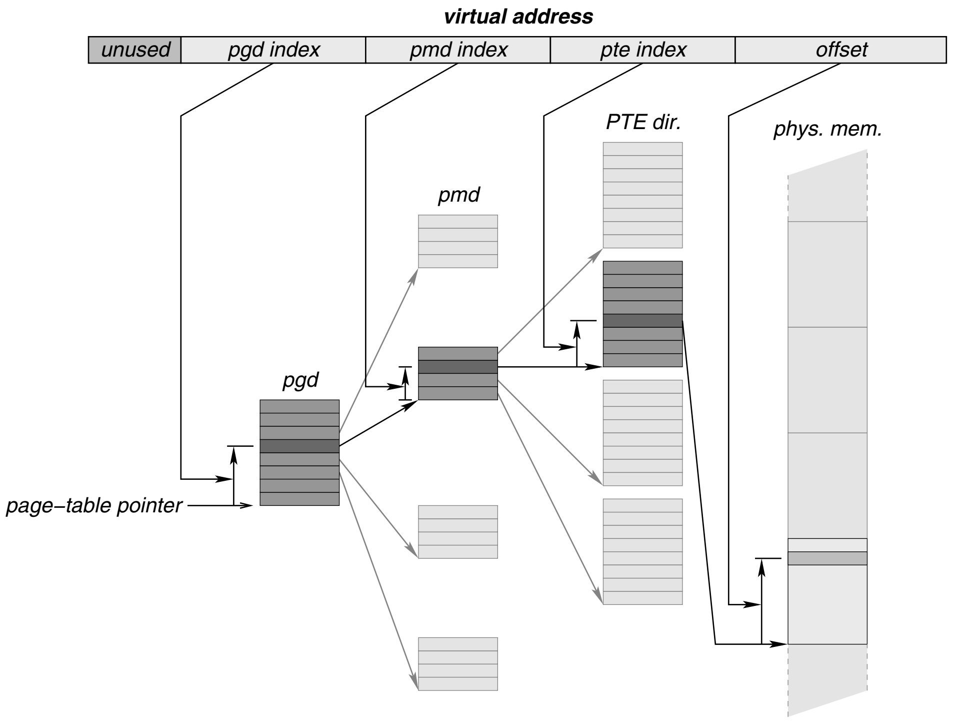


Figure 4.17. Virtual-to-physical address translation using the page table.

_PER_PGD is the total number of global-directory entries available to map user space. For the middle and PTE directories, all entries are assumed to map user space.

So how can we use the page table to translate a virtual address to the corresponding physical address? Figure 4.17 illustrates this. At the top, we see that a virtual address, for the purpose of a page-table lookup, is broken up into multiple fields. The fields used to look up the page table are pgd index, pmd index, and pte index. As their names suggest, these fields are used to index the global, middle, and PTE directories, respectively. A page-table lookup starts with the page-table pointer stored in the mm structure of a process. In the figure, this is illustrated by the arrow labeled *page-table pointer*. It points to the global directory, i.e., the root of the page-table tree. Given the global directory, the pgd index tells us which entry contains the address of the middle directory. With the address of the middle directory, we can use the pmd index to tell us which entry contains the address of the PTE directory. With the address of the PTE directory, we can use the pte index to tell us which entry contains the PTE of the page that the virtual address maps to. With the PTE, we can calculate the address of the physical page frame that backs the virtual page. To finish up the physical address calculation, we just need to add the value in the offset field of the virtual address to the page frame address.

Note that the width of the pgd index, pmd index, and pte index fields is dictated by the number of pointers that can be stored in a global, middle, and PTE directory, respectively. Similarly, the width of the offset field is dictated by the page size. Because these fields by definition consist of an integral number of bits, the page size and the number of entries stored in each directory all must be integer powers of 2. If the sum of the widths of these fields is less than the width of a virtual address, some of the address bits remain unused. The figure illustrates this with the field labeled **unused**. On 32-bit platforms, there are usually no unused bits. However, on 64-bit platforms, the theoretically available virtual address space is so big that it is not unusual for some bits to remain unused. We discuss this in more detail when discussing the IA-64 implementation of the virtual memory system.

4.3.1 Collapsing page-table levels

At the beginning of this section, we said that the Linux page tables *logically* contain three levels. The reason is that each platform is free to implement fewer than three levels. This is possible because the interfaces that Linux uses to access and manipulate page tables have been carefully structured to allow collapsing one or even two levels of the tree into the global directory. This is an elegant solution because it allows platform-independent code to be written as if each platform implemented three levels, yet on platforms that implement fewer levels, the code accessing the unimplemented levels is optimized away completely by the C compiler. That is, no extra overhead results from the extraneous logical page-table levels.

The basic idea behind collapsing a page-table level is to treat a single directory entry as if it were the entire directory for the next level. For example, we can collapse the middle directory into the global directory by treating each global-directory entry as if it were a middle directory with just a single entry (i.e., PTRS_PER_PMD = 1). Later in this chapter, we see some concrete examples of how this works when a page table is accessed.

4.3.2 Virtually-mapped linear page tables

As discussed in the previous section, linear page tables are not very practical when implemented in physical memory. However, under certain circumstances it is possible to map a multiway tree into virtual space and make it appear as if it were a linear page table. The trick that makes this possible is to place a *self-mapping* entry in the global directory. The self-mapping entry, instead of pointing to a middle directory, points to the page frame that contains the global directory. This is illustrated in Figure 4.18 for the case where the global-directory entry with index 7 is used as the self-mapped entry (labeled SELF). Note that the remainder of the global directory continues to be used in the normal fashion, with entries that are either not mapped or that point to a middle directory.

To make it easier to understand how this self-mapping works, let us consider a specific example: Assume that page-table entries are 8 bytes in size, pages are 8 Kbytes in size, and that the directories in the page tree all contain 1024 entries. These assumptions imply that the page offset is 13 bits wide and that the pgd, pmd, and pte indices are all 10 bits wide. The virtual address is thus 43 bits wide. A final assumption we need to make is that the

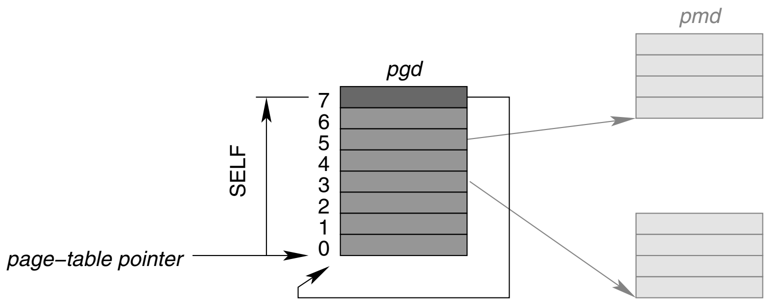


Figure 4.18. Self-mapping entry in the global directory.

format of the entries in the pgd and pmd directories is identical to the format used in the PTE directories.

Assuming the self-mapping entry has been installed in the global-directory entry with index SELF, we claim that the equations below for $L3(va)$, $L2(va)$, and $L1(va)$ are the virtual addresses at which we can find, respectively, the PTE-directory entry, middle-directory entry, and global-directory entry that correspond to virtual address va:

$$\begin{aligned} L3(va) &= \mathsf{SELF}\cdot 2^{33} + 8\cdot\lfloor va/2^{13}\rfloor \\ L2(va) &= \mathsf{SELF}\cdot 2^{33} + \mathsf{SELF}\cdot 2^{22} + 8\cdot\lfloor va/2^{23}\rfloor \\ L1(va) &= \mathsf{SELF}\cdot 2^{33} + \mathsf{SELF}\cdot 2^{22} + \mathsf{SELF}\cdot 2^{13} + 8\cdot\lfloor va/2^{33}\rfloor \end{aligned}$$

For example, if we assume that SELF $=$ `1023` $=$ `0x3ff`, we could access the page-table entry for virtual address $va =$ `0x80af3` at virtual address `0x7fe00000200`.

The effect of the self-mapping entry can be observed most readily when considering how the above equations affect the virtual address. Figure 4.19 illustrates this effect. The first line shows the virtual address va broken up into the three directory indices *va_pgd_idx*, *va_pmd_idx*, *va_pte_idx* and the page offset *va_off*. Now, if we consider the effect of equation $L3(va)$, we see that the page offset was replaced by $8\cdot$ *va_pte_idx* (the factor of 8 comes from the fact that each page-table entry is assumed to be 8 bytes in size). Similarly, the pte index has been replaced with *va_pmd_idx*, and the pmd index has been replaced with *va_pgd_idx*. Finally, the pgd index has been replaced with SELF, the index of the self-mapping entry. When we look at the effects of $L2(va)$ and $L1(va)$, we see a pattern emerge: At each level, the previous address is shifted down by 10 bits (the width of an index) and the top 10 bits are filled in with the value of SELF.

Now, let us take a look at what the operational effect of $L1(va)$ is when used in a normal three-level page table lookup. We start at the global directory and access the entry at index SELF. This lookup returns a pointer to a middle directory. However, because this is the self-mapping entry, we really get a pointer to the global directory. In other words, the global directory now serves as a fake middle directory. So we use the global directory

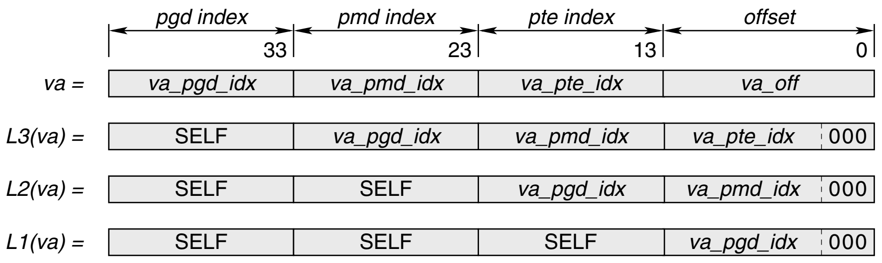


Figure 4.19. Effect of self-mapping entry on virtual address.

again to look up the pmd index, which happens to be SELF again. As before, this lookup returns a pointer back to the global directory but this time the directory serves as a fake PTE directory. To complete the three-level lookup, we again use the global directory to look up the pte index, which again contains SELF. The entry at index SELF is now interpreted as a PTE. Because we are dealing with the self-mapping entry, it once again points to the global directory, which now serves as a fake page frame. The physical address that $L1(va)$ corresponds to is thus equal to the address of global directory plus the page offset of the virtual address, which is $8 \cdot va\_pgd\_idx$. Of course, this is exactly the physical address of the global-directory entry that corresponds to address *va*—just as we claimed!

For equations $L2(va)$ and $L3(va)$ the same logic applies: Every time SELF appears as a directory index, the final memory access goes one directory level higher than it normally would. That is, with SELF appearing once, we "remove" one directory level and thus access the PTE instead of the word addressed by *va*, and so on.

Another way to visualize the effect of the self-mapping entry is to look at the resulting virtual address space. Unfortunately, it would be impossible to draw the space to scale with the 1024-way tree we assumed so far. Thus, Figure 4.20 illustrates the address space that would result for a machine where each of the three indices is only 2 bits wide and the page offset is 4 bits wide. In other words, the figure illustrates the address space for a three-level 4-way tree with a 16-byte page size and page-table entries that are 4 bytes in size. The value of SELF is assumed to be 2; i.e., the second-last entry in the global directory is used for the self-mapping entry.

The figure illustrates how the virtually-mapped linear page table occupies the address space from `0x200` to `0x2ff`. Not surprisingly, this is the quarter of the address space that corresponds to the global-directory entry occupied by SELF. Most of the linear page table is occupied by page-table entries. However, its third quadrant (`0x280` to `0x2df`) is occupied primarily by the middle-directory entries. The global-directory entries can be found in the tiny section covering the address range from `0x2a0` to `0x2af`. Note how the pmd (and pgd) entries create a hole in the linear page table that corresponds exactly to the hole that the page table creates in the virtual address space. This is necessarily so because otherwise there would be virtual memory addresses for which there would be no PTE entries in the linear page table!

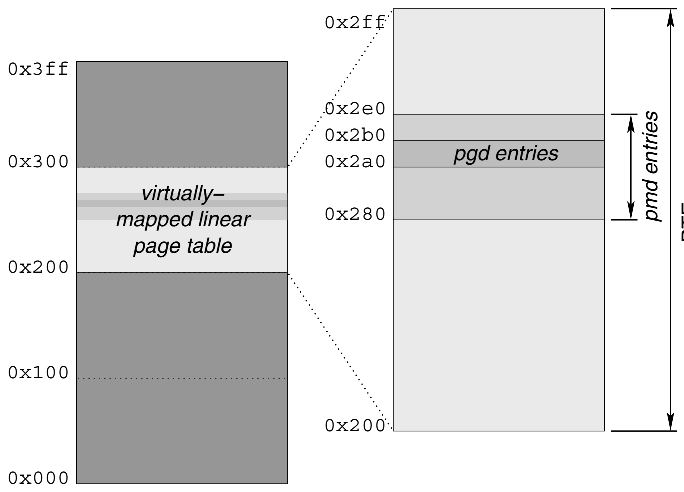


Figure 4.20. Effect of self-mapping entry on virtual address space.

Note that with a more realistic 1024-way tree, the virtual address space occupied by the virtually-mapped linear page table is less than 0.1 percent of the entire address space. In other words, the amount of space occupied by the linear page table is minuscule.

Applications of virtually-mapped linear page tables

A virtually-mapped linear page table could be used as the primary means to manipulate page tables. Once the self-mapping entry has been created in the global directory, all other directory entries can be accessed through the virtual table. For example, to install the page frame of the middle directory for virtual address *va*, we could simply write the new global-directory entry to address *L1*(*va*). While this would work correctly, it is usually more efficient to walk the page table in physical space. The reason is that a virtual access involves a full page-table walk (three memory accesses), whereas a page-directory access in physical space would take just one memory access.

The true value of a linear virtual page table derives from the fact that because it is a linear table, special hardware that accelerates page-table lookups can easily be built for it. We see an example of this in Section 4.4.1.

4.3.3 Structure of Linux/ia64 page tables

On IA-64, the Linux kernel uses a three-level page-table tree. Each directory occupies one page frame and entries are 8 bytes in size. With a page size of 8 Kbytes, this implies that the page table forms a 1024-way tree. The global- and middle-directory entries contain the physical address of the next directory or 0 if the entry is not mapped. In contrast, the PTE-directory entries follow the PTE format that we describe in Section 4.3.4.

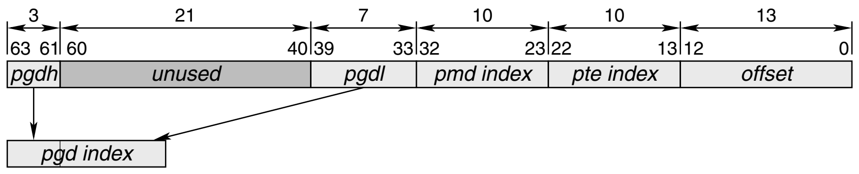


Figure 4.21. Format of user-space virtual address (with 8-Kbyte pages).

Figure 4.21 illustrates the user-space virtual address format. Corresponding to a page size of 8 Kbytes, the page offset occupies the lowest 13 bits. Similarly, because each directory level has 1024 ways, each index is 10 bits wide. As the figure shows, the PTE-directory index (pte index) occupies bits 13 to 22, and the middle-directory index (pmd index) occupies bits 23 to 32. What is unusual is that the global-directory index is split into a 7-bit low part (pgdl) and a 3-bit high part (pgdh) that covers bits 33 to 39 and 61 to 63, respectively. Bits 40 to 60 are unused and must be 0.

The global-directory index is split into two pieces to make it possible to map a portion of the virtual space of each IA-64 region. This provides room for future growth. For example, an application could place its data segment in region 3 and the stacks in region 4. With this arrangement, the starting addresses of the data segment and the stacks would be separated by 2^{61} bytes and the application would be virtually guaranteed that, no matter how much data the application will have to process in the future, its data segment will never collide with the stacks. Of course, the size of the data segment would still be limited by the amount of space that can be mapped within a region, but by using separate regions the application isolates itself from having to know this implementation detail. In other words, as future machines can support more memory, future Linux kernels can implement a larger virtual address space per region and the application will be able to take advantage of this larger space without requiring any changes.

It should be noted that the Linux kernel normally expects directory indices to take up consecutive bits in a virtual address. The exception to this rule is that discontiguities are permitted in the global-directory index *provided* that no vm-area structure maps an address range that would overlap with any of the unimplemented user-space areas. On IA-64, this means that all vm-areas must be fully contained within the first 2^{40} bytes of a region (assuming 8-Kbyte pages, as usual). Linux/ia64 checks for this constraint and rejects any attempts to violate it.

Recall from Figure 4.11 on page 149 that regions 5 through 7 are reserved for kernel use. A process therefore is not permitted to map any addresses into user space for which pgdh would be greater than or equal to 5. The Linux/ia64 kernel enforces this by setting parameters FIRST_USER_PGD_NR to 0 and USER_PTRS_PER_PGD to $5 \cdot 2^7 = 640$. This implies that the top three octants of each global directory remain unused. Apart from wasting a small amount of memory, this does not cause any problems.

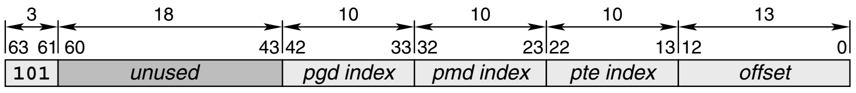


Figure 4.22. Format of kernel virtual address for region 5 (with 8-Kbyte pages).

Table 4.1. Summary of Linux/ia64 virtual memory and page-table parameters.

	Page size			
Parameter	4 Kbytes	8 Kbytes	16 Kbytes	64 Kbytes
total user address space	320 Gbytes	5 Tbytes	80 Tbytes	20 Pbytes
address space per region	64 Gbytes	1 Tbyte	16 Tbytes	4 Pbyte
width of directory index	9 bits	10 bits	11 bits	13 bits
PTRS_PER_PGD	512	1024	2048	8192
PTRS_PER_PMD	512	1024	2048	8192
PTRS_PER_PTE	512	1024	2048	8192
FIRST_USER_PGD_NR	0	0	0	0
USER_PTRS_PER_PGD	320	640	1280	5120

Kernel page table

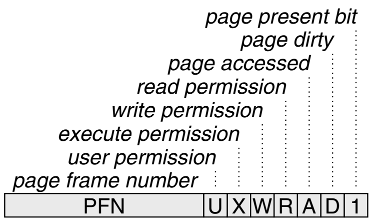
The kernel uses a separate page table to manage the page-table-mapped kernel segment. In contrast to user space, where there is one page table per process, this page table belongs to the kernel and is in effect independent of which process is running.

On IA-64, this segment is implemented in region 5; the structure of virtual addresses in this region is illustrated in Figure 4.22. We can see that it is identical to Figure 4.21, except that the global-directory index is no longer split into two pieces. Instead, it occupies bits 33 to 42. As required for region 5 addresses, the region number bits 61 to 63 have a fixed value of 5 or, in binary notation, `101`.

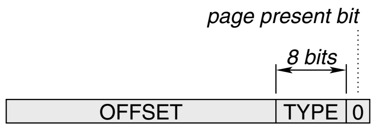
Summary

Table 4.1 summarizes key virtual-memory and page-table parameters as a function of the Linux/ia64 page size. With a page size of 8 Kbytes, the total mappable user address space is 5 Tbytes (five regions of 1 Tbyte each). Larger page sizes yield correspondingly larger spaces. With the smallest page size of 4 Kbytes, “only” 320 Gbytes of user address space are available. The other extreme is a page size of 64 Kbytes, which yields a mappable user address space of 20 Pbytes ($20 \cdot 2^{50}$ bytes!). By almost any standard, this can only be described as *huge*.

Because Linux/ia64 uses the same index width in each of the three levels of the page table, the three parameters PTRS_PER_PGD, PTRS_PER_PMD, and PTRS_PER_PTE have the same value. With a page size of 8 Kbytes, they are all equal to 1024, and with a page size of 64 Kbytes, they increase to 8192. The value for USER_PTRS_PER_PGD simply reflects the fact that the last three-eighths of the address space is occupied by kernel space.



(a) page present



(b) page not present

Figure 4.23. Logical format of a page-table entry (PTE).

4.3.4 Page-table entries (PTEs)

Linux assumes that page-table entries (PTEs) have one of two possible logical formats, depending on whether or not a virtual page is present in physical memory. If it is present, the PTE has the logical format shown in Figure 4.23 (a). If a virtual page is not present, the PTE has the format shown in Figure 4.23 (b). Comparing the two, we see that the **page present** bit is common, and, as the name suggests, it determines which format is in effect. A value of 1 indicates that the PTE has the **present** format, and a value of 0 indicates the PTE has the **not present** format.

Let us first take a closer look at Figure 4.23 (a). As it shows, the bit next to the **present** bit is the **dirty** (D) bit. It indicates whether there has been a write access to the virtual page since the bit was cleared last. The next bit is called the **accessed** (A) bit. It indicates whether the virtual page has been accessed at all (read, written, or executed) since the bit was cleared last. As explained earlier, the A and D bits are related in that they both play a role in implementing the page replacement and writeback policies. The next three bits are the permission bits; they control whether read (R), write (W), or execute (X) accesses to the page are permitted. A value of 1 indicates that accesses of that type are permitted, and a value of 0 prohibits such accesses. The permission bits are followed by a related bit, the **user** (U) bit. This bit controls whether the page is accessible from the user level. If it is 0, only the kernel can access the page. If it is 1, both the kernel and the user level can access the page in accordance with the permission bits. The last field in the Linux PTE contains the page frame number (PFN). The width of this field is platform specific, since it depends on the maximum amount of physical memory that the platform can support.

If the **present** bit is off, the PTE has the much simpler format shown in Figure 4.23 (b). Apart from the **present** bit, there are just two fields: TYPE and OFFSET. These fields record the disk location where a virtual page has been stored. Specifically, TYPE specifies the swap device (or swap file) and OFFSET specifies the swap block number. The TYPE field must be 8 bits wide, which implies that up to 256 distinct swap devices or files can be supported. There is no minimum width for the OFFSET field—Linux uses whatever number of bits are available on a particular platform. However, since its width determines the maximum block number, OFFSET determines the maximum size of a swap device or file, so it is generally a good idea to make this field as wide as possible.

The PTE manipulation interface

It is important to realize that the two formats depicted in Figure 4.23 are only *logical* formats. Linux never directly manipulates fields in a PTE. Instead, it manipulates them indirectly through the *PTE manipulation interface* (file include/asm/pgtable.h) shown in Figure 4.24. This interface gives each platform the flexibility to map the logical PTE format to a real format that best suits the requirements of the platform.

The interface defines three types, called **pte_t**, **swp_entry_t**, and **pgprot_t**. The first one is an opaque type that represents a PTE in the platform-specific format. The second type, **swp_entry_t** represents PTE values for which the **present** bit is off. The reason Linux requires a separate type for this case is that limitations in the page out code require that such PTEs must fit into a variable of type **unsigned long**. No such limitation exists for PTEs that have the **present** bit on, so **pte_t** could be a much larger type. However, the two types are related in the sense that it must be possible to convert a value of type **swp_entry_t** to a value of type **pte_t** without loss of information. The reverse also must be true for PTE values that have the **present** bit off.

The third type, **pgprot_t**, is also a platform-specific opaque type and represents the PTE bits other than the PFN field. Its name derives from the fact that values of this type contain the page permission bits. However, a platform is free to represent any other PTE bits in this type, so it generally contains more than just the permission bits. Associated with **pgprot_t** are several manifest constants that define the fundamental permission bit settings that the Linux kernel uses. Table 4.2 describes the available constants. The first column gives the name of the constant, the second column provides the logical permission bit settings (R, W, and X) as well as the value of the U bit that the constant encodes (see Figure 4.23), and the third column describes how the constant is used by the kernel. Constants whose name begins with PAGE represent protection values that are commonly used throughout the kernel. In contrast, the __P*xwr* and __S*xwr* constants translate the protection and sharing attributes of the *mmap()* system call to suitable protection values. Specifically, if the MAP_PRIVATE flag is specified in the system call, modifications to the mapped area should remain private to the process and Linux uses one of the __P*xwr* constants. If MAP_PRIVATE is not specified, the kernel uses one of the __S*xwr* constants instead. The specific constant used depends on the protection attributes specified in the system call. In the name of the constant, *r* corresponds to PROT_READ, *w* to PROT_WRITE, and *x* to PROT_EXEC. For example, for an *mmap()* system call with MAP_PRIVATE, PROT_READ, and PROT_WRITE specified, the kernel would use __P011. As the table illustrates, the major difference between the __P*xwr* and the __S*xwr* constants is that the former always have the W bit turned off. With write permission turned off, Linux can use the copy-on-write scheme to copy only those privately-mapped pages that really are modified by the process.

Comparing PTEs

The first routine in the interface is *pte_same()*, which compares two PTEs for equality. The routine expects two arguments *pte1* and *pte2* of type **pte_t** and returns a non-zero value if the two PTEs are identical.

```
/* PTE-related types: */
typedef pte_t;                                /* page-table entry (PTE) */
typedef unsigned long swp_entry_t;            /* swap space entry */
typedef pgprot_t;                             /* page protections */

/* comparing PTEs */
int pte_same(pte1, pte2);                     /* check for PTE equality */

/* converting between page frame descriptors and PTEs: */
pte_t mk_pte(page, prot);                     /* create PTE from page & prot. bits */
pte_t mk_pte_phys(addr, prot);                /* create PTE from addr. & prot. bits */
struct page *pte_page(pte);                   /* get page frame descriptor for PTE */

/* manipulating the accessed/dirty bits: */
pte_t pte_mkold(pte);                         /* mark PTE as not recently accessed */
pte_t pte_mkyoung(pte);                       /* mark PTE as recently accessed */
pte_t pte_mkclean(pte);                       /* mark PTE as not written */
pte_t pte_mkdirty(pte);                       /* mark PTE as written */
int pte_young(pte);                           /* PTE marked as accessed? */
int pte_dirty(pte);                           /* PTE marked as written? */

/* manipulating protection bits: */
pte_t pte_modify(pte, prot);                  /* set protection bits in PTE */
pte_t pte_wrprotect(pte);                     /* mark PTE as not writable */
pte_t pte_mkwrite(pte);                       /* mark PTE as writable */
pte_t pte_mkexec(pte);                        /* mark PTE as executable */
int pte_read(pte);                            /* PTE marked as readable? */
int pte_write(pte);                           /* PTE marked as writable? */
int pte_exec(pte);                            /* PTE marked as executable? */

/* modifying PTEs atomically: */
int ptep_test_and_clear_dirty(pte_ptr);       /* test and clear dirty bit */
int ptep_test_and_clear_young(pte_ptr);       /* test and clear accessed bit */
ptep_mkdirty(pte_ptr);                        /* mark PTE as written */
ptep_set_wrprotect(pte_ptr);                  /* mark PTE as not writable */
pte_t ptep_get_and_clear(pte_ptr);            /* get and clear PTE */

/* manipulating swap entries: */
swp_entry_t pte_to_swp_entry(pte);            /* extract swap entry from PTE */
pte_t swp_entry_to_pte(swp_entry);            /* convert swap entry to PTE */
unsigned long SWP_TYPE(swp_entry);            /* extract TYPE from swap entry */
unsigned long SWP_OFFSET(swp_entry);          /* extract OFFSET from swap entry */
swp_entry_t SWP_ENTRY(type, offset);          /* convert type & offset to swap entry */
```

Figure 4.24. Kernel interface to manipulate page-table entries (PTEs).

Table 4.2. Page protection constants.

Constant	UXWR	*Description*
PAGE_NONE	1 0 0 0	Protection value to use for a page with no access rights.
PAGE_READONLY	1 0 0 1	Protection value to use for a read-only page.
PAGE_SHARED	1 0 1 1	Protection value to use for a page where modifications are to be shared by all processes that map this page.
PAGE_COPY	1 1 0 1	Protection value to use for a copy-on-write page.
PAGE_KERNEL	0 1 1 1	Protection value to use for a page that is accessible by the kernel only.
__P*xwr*	1 *x* 0 *r*	Protection value to use on a page with access rights *xwr* when modifications to the page should remain private to the process. There are eight such constants, one for each possible combination of the execute (*x*), write (*w*), and read (*r*) bits. For example, constant __P001 defines the value to use for a page that is only readable.
__S*xwr*	1 *x w r*	Protection value to use on a page with access rights *xwr* when modifications to the page should be shared with other processes. There are eight such constants, one for each possible combination of the *xwr* access bits.

Converting between page frame descriptors and PTEs

The second set of routines in the interface provides the ability to convert page frame pointers to PTEs, and vice versa. Routine *mk_pte()* expects two arguments: a pointer *page* to a page frame descriptor and a protection value *prot* that specifies the page protection as a platform-specific **pgprot_t** value. The routine calculates the page frame number of *page* and combines it with *prot* to form a PTE that is returned in the form of a value of type **pte_t**. Routine *mk_pte_phys()* provides the same functionality, except that the first argument is a physical address instead of a page frame descriptor. This routine is used to map a page frame for which there is no page frame descriptor—a case that usually arises for memory-mapped I/O pages. The *pte_page()* routine provides the reverse operation. It expects the single argument *pte*, which must be a PTE, and returns a pointer to the page frame descriptor that corresponds to *pte*. The operation of this routine is undefined if *pte* maps a physical address for which no page frame descriptor exists.

Manipulating the accessed/dirty bits

The third set of routines provides the ability to manipulate and test the **accessed** (A) and **dirty** (D) bits in the PTE. They all expect the single argument *pte*, which must be a PTE of type **pte_t**. The routines that modify the PTE also return a value of that type. The routines that test a bit in the PTE simply return the current value of the bit. The *pte_mkold()* routine clears the A bit in the PTE, and *pte_mkyoung()* sets it. Similarly, *pte_mkclean()* clears the D bit in the PTE, and *pte_mkdirty()* sets it. The *pte_young()* and *pte_dirty()* routines can be used to query the current settings of the A and D bits, respectively.

Manipulating protection bits

The fourth set of routines provides the ability to manipulate the protection bits in a PTE. The first argument to these routines is always a PTE of type **pte_t**. Those routines that modify the PTE also return a value of that type, whereas the routines that test a particular permission bit return a non-zero value if the bit is set and 0 if the bit is cleared. The *pte_modify()* routine can be used to replace the protection bits with the value passed in the second argument, *prot*. This argument must be of type **pgprot_t** and contain a protection value in the platform-specific format. The *pte_wrprotect()* routine can be used to clear the W bit in a PTE, and *pte_mkwrite()* can be used to set it. Similarly, *pte_mkexec()* can be used to set the X bit. No routines directly manipulate the R bit. The current setting of the R, W, and X permission bits can be queried with *pte_read()*, *pte_write()*, and *pte_exec()*, respectively.

Modifying PTEs atomically

The fifth set of routines provides routines for atomically modifying existing PTEs. These routines need to be used to avoid potential race conditions when PTEs that are marked **present** in a page table are modified. For example, consider a machine with two CPUs. If one CPU performs a write access to a page just as the other CPU attempts to clear the **dirty** bit, the bit could end up with the wrong value if it is not cleared atomically. To avoid such problems, the interface provides *ptep_test_and_clear_dirty()* to atomically read and then clear the D bit. The routine expects one argument, *pte_ptr*, which must be a pointer to the PTE to be modified. The return value is non-zero if the D bit was set and 0 otherwise. Similarly, *ptep_test_and_clear_young()* can be used to atomically test and clear the A bit. The *ptep_mkdirty()* and *ptep_set_wrprotect()* routines are atomic variants of *pte_mkdirty()* and *pte_wrprotect()*, respectively. The last atomic routine, *ptep_get_and_clear()* is a catch-all routine in the sense that it can be used to atomically modify any bits in the PTE. It works by atomically reading and then clearing the PTE pointed to by *pte_ptr*. It is a catch-all because once the PTE is cleared, its **present** bit is off and all future accesses by other CPUs will cause a page fault. In the page fault handler, the other CPUs will have to follow the normal MP-locking protocol to avoid any potential race conditions. In other words, the other routines to atomically modify PTEs are not strictly needed—they could all be emulated with *ptep_get_and_clear()*. However, providing specialized routines for the most common operations makes sense because such routines are generally more efficient than implementations based on *ptep_get_and_clear()*.

Manipulating swap entries

The final set of PTE-related routines provides the means to support moving virtual pages to swap space and back. Specifically, *pte_to_swp_entry()* converts a PTE to the corresponding swap entry. The PTE is specified by argument *pte*, and the return value is the corresponding swap entry. This operation is meaningful only if *pte* has the **present** bit turned off, i.e., the PTE must already be in the format illustrated in Figure 4.23 (b). The *swp_entry_to_pte()* routine provides the reverse operation and translates the swap entry passed in argument *swp_entry* to an equivalent PTE. The TYPE and OFFSET fields can be extracted from a

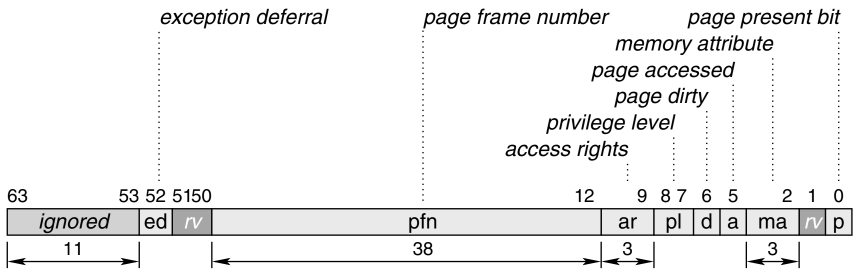


Figure 4.25. Format of IA-64 PTE when **present** bit is set (p = 1).

swap entry with *SWP_TYPE()* and *SWP_OFFSET()*, respectively. Conversely, *SWP_ENTRY()* can be used to construct a swap entry in accordance with the values of the *type* and *offset* arguments.

Note that no routine directly translates a PTE in the **present** format to a PTE in the **not present** format. Instead, Linux uses the following canonical sequence to move a page out to swap space:

1. Call *ptep_get_and_clear()* to read the old PTE and then clear it.
2. Reserve swap space for the page; the swap space location reserved determines the value of the TYPE and OFFSET fields.
3. Write the page frame to swap space.
4. Build the swap entry by calling *SWP_ENTRY()*.
5. Call *swp_entry_to_pte()* to translate the swap entry into the corresponding PTE.
6. Install the new PTE in the page table.

Similarly, when a page is moved from disk to memory, the corresponding steps happen in reverse. In other words, when a virtual page is moved from memory to disk, or vice versa, the old PTE is essentially "thrown away" and a new one is constructed from ground up, which is why there is no need for routines that explicitly manipulate the **present** bit.

IA-64 implementation

Figure 4.25 illustrates the PTE format used by Linux/ia64. To help distinguish the logical PTE format in Figure 4.23 from the IA-64–specific format shown here, the figure uses lowercase names for the field names. The figure illustrates that there is a **present** bit p, an **accessed** bit a, a **dirty** bit d, and a page frame number field pfn just as in the logical format. These bits serve exactly the same purpose as the corresponding bits in the logical format.

The IA-64 PTE format does not have separate permission bits. Instead, it uses the 2-bit privilege-level field pl and the 3-bit access-rights field ar to determine the protection

Table 4.3. IA-64 page access rights.

		user	
ar	***kernel***	**pl = 0**	**pl = 3**
0	R	–	R
1	RX	–	RX
2	RW	–	RW
3	RWX	–	RWX
4	RW	–	R
5	RWX	–	RX
6	RW	–	RWX
7	RX	XP0	X

status of a page. IA-64 supports four privilege levels, but Linux uses only two of them: 0 is the most-privileged level and is used for accesses by the kernel; 3 is the least-privileged level and is used for user accesses. Table 4.3 shows how the different values for ar and pl affect the rights for kernel and user-level accesses. Note that for the kernel, the access rights are determined solely by ar; pl has no influence. For user-level accesses, pl = 0 generally means that the page is owned by the kernel and therefore inaccessible. However, something interesting happens when ar = 7. In the table, this entry is marked XP0, which should be read as *execute, promote to privilege-level 0*. This means that the page is executable even at the user level. Furthermore, if an epc (enter privileged code) instruction is executed in such a page, the privilege level is raised to privilege-level 0 (kernel). This mechanism can be used to implement system calls and, as we see in Chapter 5, *Kernel Entry and Exit*, is also useful for signal handling. The gate page described in Section 4.2.4 is mapped in this fashion, and the IA-64–specific constant PAGE_GATE can be used to map other pages in this way.

Another interesting aspect of the IA-64 protection scheme is that there is no way to grant write permission without also granting read permission. Fortunately, for the R and X permission bits, Linux allows platform-specific code to grant the access right even if it was not set in the logical protection value (this is *not* true for the W bit, of course). Thus, Linux/ia64 simply maps logical protection values that have only the W bit set to privilege-level and access-right values that grant both read and write permission.

The IA-64 PTE defines two fields, ma and ed, that have no equivalent in the logical format. The ma field specifies the *memory attribute* of the page. For all normal memory pages, this value is 0, indicating that accesses to the page are cacheable. However, to support memory-mapped I/O, this field can be used to select different types of uncached accesses. For example, ma = 4 marks the page as **uncacheable**. A value of ma = 6 marks the page as **write coalescing**, which is similar to **uncacheable**, except that consecutive writes can be coalesced into a single write transaction. As a special case, ma = 7 marks a page as a **NaT Page**. Speculative loads from such a page will deposit a NaT token in the target register. Other memory accesses, such as nonspeculative loads or stores, will trigger a NAT PAGE CONSUMPTION FAULT. The ed bit is also related to speculation. It is meaningful only for pages that are executable. For such pages, the bit indicates the speculation model that is in effect. A value of 1 indicates the **recovery** model, and a value of 0 indicates the **no-recovery** model. See the discussion on page 199 for more details on the speculation model. Linux supports only the **recovery** model, so this bit should always be 1.

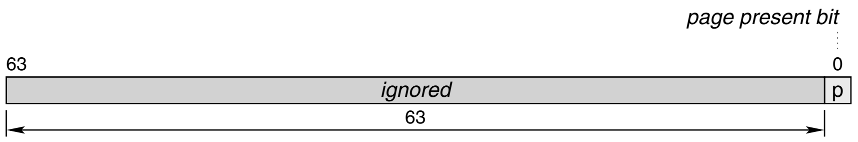


Figure 4.26. Format of IA-64 PTE when **present** bit is cleared (p = 0).

Finally, note that some bits in the IA-64 PTE are marked **rv**. These bits are reserved for future architectural uses. Attempting to set them will result in a RESERVED REGISTER/FIELD FAULT when the PTE is loaded into the CPU. In contrast, the most significant 11 bits of the PTE are defined as **ignored**. These bits are completely ignored by the CPU and are available for use by the operating system. Linux/ia64 uses bit 63 to mark a page that has no access rights. Because such a page is inaccessible, there is no need to back it with a physical page frame. Thus, such a page can be considered to be present even if the p bit in the PTE is turned off.

Figure 4.26 shows the IA-64 PTE format when the present bit p is cleared. In this case, the bits other than the present bit are all ignored by the CPU and are available for use by the operating system. Linux uses bits 1 through 8 to encode the swap entry TYPE field and bits 9 through 62 to encode the swap entry OFFSET field. Bit 63 is again used to mark a page with no access rights. Note that if this bit is set, Linux considers the page present even though bit p is cleared.

Given the IA-64 PTE format, it is clear that types **pte_t** and **pgprot_t** must be wide enough to hold a 64-bit word. They could be defined as being equivalent to **unsigned long**. However, to take advantage of C type-checking rules, both types are declared as separate structures that contain just one 64-bit word each. This is a compiler trick that makes it easier to detect programming errors early on because passing a PTE where a protection value is expected (or vice versa) results in a type violation that can be reported by the C compiler.

The IA-64 implementations of the many PTE manipulation routines defined in Figure 4.24 are straightforward; most of them involve just simple bit manipulation in PTE or protection values. The PTE manipulation routines that need to execute atomically are all implemented with the cmpxchg8 instruction that we already encountered in Chapter 3, *Processes, Tasks, and Threads*.

4.3.5 Page-table accesses

Just like PTEs, Linux never directly manipulates directory entries. Instead, it manipulates them through the interface shown in Figure 4.27. In this interface, all virtual address arguments are called *addr* and must be values of type **unsigned long**. Global- and middle-directory entries are represented by types **pgd_t** and **pmd_t**, respectively. As we have seen already, PTE-directory entries are represented by type **pte_t**. All types usually occupy one machine word, but for some platforms, they are more complex structures occupying multiple words.

```
typedef pgd_t;                          /* global-directory entry */
typedef pmd_t;                          /* middle-directory entry */
unsigned long pgd_index(addr);          /* extract pgd index from addr */
pgd_t *pgd_offset(mm, addr);            /* get pgd entry pointer for addr */
pgd_t *pgd_offset_k(addr);              /* get kernel pgd entry pointer for addr */
pmd_t *pmd_offset(pgd_entry_ptr, addr); /* get pmd entry pointer for addr */
pte_t *pte_offset(pmd_entry_ptr, addr); /* get PTE pointer for addr */
int pgd_none(pgd_entry);                /* check if pgd_entry is mapped */
int pgd_present(pgd_entry);             /* check if pgd_entry is present */
int pgd_bad(pgd_entry);                 /* check if pgd_entry is invalid */
int pmd_none(pmd_entry);                /* check if pmd_entry is mapped */
int pmd_present(pmd_entry);             /* check if pmd_entry is present */
int pmd_bad(pmd_entry);                 /* check if pmd_entry is invalid */
int pte_none(pte);                      /* check if pte is mapped */
int pte_present(pte);                   /* check if pte is present */
```

Figure 4.27. Kernel interface to access page tables.

The first routine in this interface, *pgd_index()*, simply extracts the global-directory index from virtual address *addr* and returns it.

The *pgd_offset()* routine looks up the global-directory entry for a user-space address. Argument *mm* specifies the mm structure (address space), and *addr* specifies the virtual address to look up. Similarly, *pgd_offset_k()* can be used to look up the global-directory entry for a kernel-space address. Because there is only one kernel page table, this routine needs just one argument: the virtual address *addr*, which must point inside the page-table-mapped kernel segment.

Once the global-directory entry has been found, *pmd_offset()* can be used to find the middle-directory entry of a virtual address. This routine expects two arguments, namely, *pgd_entry_ptr*, which must be a pointer to the global-directory entry found previously, and *addr*, the virtual address being looked up. This routine can be used both for user-space and kernel-space addresses.

Once the middle-directory entry has been found, *pte_offset()* can be used to find the PTE-directory entry of a virtual address. This routine again expects two arguments, namely, *pmd_entry_ptr*, which must be a pointer to the middle-directory entry found previously, and *addr*, the virtual address being looked up. This routine also can be used both for user-space and kernel-space addresses.

The final eight routines in the interface check the validity of a directory entry. The 3-character prefix of the routine name indicates the page-table level at which the check is used (**pgd** for global-, **pmd** for middle-, and **pte** for PTE-directory entries). Routines whose names end in **none** check whether the entry corresponds to a mapped portion of the address space. Routines whose names end in **present** check whether the entry corresponds to a portion of the address space that is present in physical memory. Because Linux page tables are not paged themselves, they are never stored in the swap space and the *pgd_present()* and *pmd_present()* routines always return the complement of *pgd_none()* and *pmd_none()* (an entry is present if it is mapped, and vice versa). Routines whose name end in **bad** check whether the entry is invalid. These routines are intended to make it easier to detect page table corruptions as early as possible but are not strictly required for the correct operation of Linux. Thus, on platforms that cannot (easily) distinguish valid and invalid entries, these routines can always return **false**. Also, note that *pte_bad()* does not exist, so there is no means to check whether a PTE is invalid.

The page-table access interface has been designed to support two primary access methods: address-driven and range-driven accesses. The former starts with a virtual address and, level by level, determines the global-, middle-, and PTE-directory entries that correspond to this address. In contrast, range-driven accesses visit all directory entries that fall into a certain address range. It is best to illustrate each method with an example. We do so next.

Example: Printing physical address of a virtual address

A good way to illustrate address-driven accesses is to look at how a user-space virtual address can be translated into the corresponding physical address. The code below implements a function called *print_pa()* that accomplishes this translation. It expects two arguments: a pointer, *mm*, to an mm structure and a virtual address, *va*. The function attempts to translate the virtual address by using the page table associated with *mm*. If the translation exists, the function prints the virtual address and the physical address it maps to. If no translation exists, nothing is printed.

```
print_pa (struct mm_struct *mm, unsigned long va) {
    pgd_t *pgd = pgd_offset(mm, va);
    if (pgd_present(*pgd)) {
        pmd_t *pmd = pmd_offset(pgd, va);
        if (pmd_present(*pmd)) {
            pte_t *pte = pte_offset(pmd, va);
            if (pte_present(*pte))
                printk("va 0x%lx -> pa 0x%lx\n",
                       va, page_address(pte_page(*pte));
        }
    }
}
```

Taking a closer look at the function, we see that it first uses *pgd_offset()* to obtain a pointer *pgd* to the global-directory entry for the virtual address. Then it uses *pgd_present()* to check whether the entry is present, i.e., to check whether the middle directory exists. If it exists, the function descends to the next level of the page-table tree where it uses *pmd_offset()* and *pmd_present()* to accomplish the equivalent steps for the middle directory. The page-table traversal is completed by a call to *pte_offset()*. This call accesses the middle directory and returns a pointer to the PTE. The function uses *pte_present()* to check whether the PTE maps a page that is present in physical memory. If so, it uses *pte_page()* to translate the PTE into the corresponding page frame descriptor. In the final step, the function uses *page_address()* to obtain the physical address of the page frame and prints both the virtual and physical addresses with a call to *printk()*.

Let us emphasize that the above three-level lookup works correctly regardless of how many levels are actually implemented on a platform. For example, on a platform that has the middle level collapsed into the global level, *pmd_offset()* would return the value of *pgd* after casting it to a pointer to a middle-directory entry (**pmd_t**) and routine *pmd_present()* would always return **true**. This behavior not only ensures proper operation but also lets the compiler optimize away the middle-directory access completely. Thus, even though logically a three-level lookup is being used, it would perform exactly like a two-level lookup.

Example: Counting the number of present pages

The kernel uses the range-driven access method when it needs to perform a certain operation on the directory entries that correspond to a given address range. For example, in response to a *munmap()* system call, the kernel needs to remove the page-table directory entries that fall in the range specified by the system call.

To see how this access method works, let us consider the simple example of counting the number of pages that are present in a given address range. This is what the *count_pages()* function below implements. It expects three arguments: a pointer, *mm*, to an mm structure and two user-space addresses, *start* and *end*, that specify the starting and ending point of the address range in which to count present pages (*end* points to the first byte *beyond* the desired range; both *start* and *end* are assumed to be global-directory aligned).

```
count_pages (struct mm_struct *mm, long start, long end) {
    int i, j, k, num_present = 0;
    pgd_t *pgd = pgd_offset(mm, 0);
    for (i = pgd_index(start); i < pgd_index(end); ++i)
        if (pgd_present(pgd[i]) {
            pmd_t *pmd = pmd_offset(pgd + i, 0);
            for (j = 0; j < PTRS_PER_PMD; ++j)
                if (pmd_present(pmd[j])) {
                    pte_t *pte = pte_offset(pmd + j, 0);
                    for (k = 0; k < PTRS_PER_PTE; ++k)
                        if (pte_present(pte[k]))
                            ++num_present;
                }
        }
    printk("%d pages present\n", num_present);
}
```

The function calls *pgd_offset()* to obtain a pointer *pgd* to the global-directory *page*. The first argument to this function is the *mm* pointer as usual. However, the second argument is 0. This causes *pgd_offset()* to return a pointer to the very first entry in the global directory. Since we know that the directory contains PTRS_PER_PGD entries, we can equally well treat this value as a pointer to the global directory itself and use array indexing to access particular entries. This is what the first **for**-loop does: It iterates variable *i* over the global-directory indices that correspond to the starting and ending addresses. Note how *pgd_index()* is used to extract these indices from the corresponding addresses. In the loop body, *pgd_present()* checks whether the *i*th entry in the directory is present. If not, the entry is skipped. If it is present, the function descends to the next level of the tree (the middle directory). Here the same basic steps are repeated: *pmd_offset()* obtains a pointer, *pmd*, to the middle directory, and a **for**-loop iterates over each index *j* in the middle directory (0 to PTRS_PER_PMD − 1). In the third level, the analogous steps are repeated for the PTE directory. In the body of the third **for**-loop, the function iterates *k* over the indices in the PTE directory (0 to PTRS_PER_PTE − 1); if the indexed entry refers to a page frame that is present in physical memory, the function increments the counter *num_present*. After iterating over the entire address range, the function prints the resulting page count by calling *printk()*.

There are two points worth emphasizing: First, just as in the previous example, the above code works properly (and efficiently) when one or more levels of the page-table tree are collapsed into a higher level of the tree; second, note that given the global-, middle-, and PTE-directory indices *i*, *j*, *k*, there is no platform-independent way to construct a virtual address that would yield these indices. Thus, in the above example it would not be possible, e.g., to print the virtual address of each page that has been found to be present. There is nothing fundamentally difficult about providing such an operation, but because Linux does not need it, there is little point in defining it.

IA-64 implementation

Given the IA-64 page-table structure described in Section 4.3.3, it is straightforward to see how the page-table access interface is implemented. In fact, all operations are so simple that they are implemented either as macros or as inlined functions. The directory entry types (**pgd_t**, **pmd_t**, and **pte_t**) can all be thought of as being equivalent to a 64-bit word. However, to take advantage of C type-checking rules, each type has its own structure declaration. This ensures that if, say, a pmd entry is accidentally used when a pte entry is expected, the C compiler can detect the problem and issue a suitable error message.

The IA-64 implementation of *pgd_index()* simply extracts the *pgdh* and *pgdl* components (see Figure 4.21 on page 159) from the virtual address, concatenates them, and returns the result. Similarly, the implementations of *pgd_offset()*, *pgd_offset_k()*, *pmd_offset()*, and *pte_offset()* extract the relevant address bits from the virtual address to form an index and then use this value to index the directory identified by the first argument passed to these routines.

In all three levels of the directory, Linux/ia64 uses a value of 0 to indicate an entry that is not mapped. Thus, routines *pgd_none()*, *pmd_none()*, and *pte_none()* simply test whether the directory entry passed to the routine is equal to 0. Similarly, *pgd_present()* and *pmd_present()* test whether the directory entry is not 0. Because virtual pages can be paged out to disk, *pte_present()* is different and checks the **present** bit in the PTE. As a special case, a virtual page with no access rights is always considered present. Finally, on IA-64 the *pgd_bad()* and *pmd_bad()* routines verify the validity of a directory entry by checking whether they contain a valid physical address. Specifically, they check whether any of the unimplemented physical address bits are non-zero and return **true** if so.

```
pgd_t *pgd_alloc();                              /* allocate global directory */
pmd_t *pmd_alloc_one();                          /* allocate middle directory */
pte_t *pte_alloc_one();                          /* allocate PTE directory */

pgd_free(pgd_ptr);                               /* free a global directory */
pmd_free(pmd_ptr);                               /* free a middle directory */
pte_free(pte_ptr);                               /* free a PTE directory */

pmd_t *pmd_alloc_one_fast();                     /* nonblocking allocate middle dir. */
pte_t *pte_alloc_one_fast();                     /* nonblocking allocate PTE dir. */
int do_check_pgt_cache(low, high);               /* free cached directory pages */

pgd_populate(pgd_entry_ptr, pmd_entry_ptr);      /* set global-directory entry */
pmd_populate(pmd_entry_ptr, pte_ptr);            /* set middle-directory entry */
set_pte(pte_ptr, pte);                           /* set PTE */

pgd_clear(pgd_entry_ptr);                        /* clear global-directory entry */
pmd_clear(pmd_entry_ptr);                        /* clear middle-directory entry */
pte_clear(pte_ptr);                              /* clear PTE */
```

Figure 4.28. Kernel interface to create page tables.

4.3.6 Page-table directory creation

Now that we understand how page tables can be accessed, we are ready to explore how they can be created in the first place. Figure 4.28 shows the interface Linux uses to abstract platform differences in how this creation is accomplished. The first three routines can be used to create directories at each of the three levels of the page-table tree. Specifically, *pgd_alloc()* creates a global directory, *pmd_alloc_one()* creates a middle directory, and *pte_alloc_one()* creates a PTE directory. The routines return a pointer to the newly created directory or NULL if they fail for any reason. The newly created directory is initialized such that its entries are marked as **not mapped**. The memory needed for the directory is normally obtained from the page allocator. This implies that if the page allocator is short on memory, the routines can block execution temporarily.

Once a directory is no longer needed, it can be freed by a call to one of *pgd_free()*, *pmd_free()*, or *pte_free()*, depending on whether it is a global, middle, or PTE directory, respectively. To reduce allocation and deallocation overheads, most platforms maintain a cache of free directories. This is usually done by implementing these routines such that they add the newly freed directory to a list of unused directories, instead of returning their memory to the page allocator. Access to this cache is provided through two separate allocation routines: *pmd_alloc_one_fast()* and *pte_alloc_one_fast()*. These work exactly like the normal middle-, and PTE-directory allocation routines, except that they guarantee not to block execution. The cache is particularly effective because it avoids not just the normal page allocator overheads but also the overhead of initializing the directory. To see this, consider that before a directory is freed, Linux removes all existing mappings from it, meaning that by the time *pgd_free()*, *pmd_free()*, or *pte_free()* is called, the directory is guaranteed to have all entries marked as **not mapped**. Thus, when a directory is taken from the cache, it is already initialized and can be returned directly, without the need to clear its content again. While the cache generally improves performance, it could create problems if it grew too large. To prevent this, the interface defines the *do_check_pgt_cache()* routine to make it possible to reduce or limit the size of the cache. The routine takes two arguments, *low* and *high*, which are both page counts that are interpreted as follows: if the cache occupies more than *high* pages, then the routine should free up directories until the cache occupies no more than *low* pages.

The final six routines set or clear directory entries. The *pgd_populate()* routine sets the pmd identified by argument *pmd_entry_ptr* as the new value for the pgd entry pointed to by argument *pgd_entry_ptr*. In contrast, *pgd_clear()* marks the entry pointed to by *pgd_entry-_ptr* as **not mapped**. After this operation, *pgd_none()* would return **true** for this entry. The *pmd_populate()*, *pmd_clear()*, *set_pte()*, and *pte_clear()* routines implement the equivalent operations for middle- and PTE-directory entries. Backward compatibility is the only reason why *set_pte()* is not called *pte_populate()*.

IA-64 implementation

The IA-64 implementation of the page-table creation interface is straightforward, largely because each directory occupies one page frame. Memory management of directory nodes is therefore particularly easy because the normal page allocator can be used to allocate and free directories. Like most other platforms, Linux/ia64 implements the nonblocking versions of the directory allocation routines by maintaining a cache of unused directories.

4.4 TRANSLATION LOOKASIDE BUFFER (TLB)

Every time the CPU accesses virtual memory, a virtual address must be translated to the corresponding physical address. Conceptually, this translation requires a page-table walk, and with a three-level page table, three memory accesses would be required. In other words, every virtual access would result in four physical memory accesses. Clearly, if a virtual memory access were four times slower than a physical access, virtual memory would not be very popular! Fortunately, a clever trick removes most of this performance penalty: modern CPUs use a small associative memory to cache the PTEs of recently accessed virtual pages. This memory is called the *translation lookaside buffer (TLB)*.

The TLB works as follows. On a virtual memory access, the CPU searches the TLB for the virtual page number of the page that is being accessed, an operation known as *TLB lookup*. If a TLB entry is found with a matching virtual page number, a *TLB hit* occurred and the CPU can go ahead and use the PTE stored in the TLB entry to calculate the target physical address. Now, the reason the TLB makes virtual memory practical is that because it is small—typically on the order of a few dozen entries—it can be built directly into the CPU and it runs at full CPU speed. This means that as long as a translation can be found in the TLB, a virtual access executes just as fast as a physical access. Indeed, modern CPUs often execute *faster* in virtual memory because the TLB entries indicate whether it is safe to access memory speculatively (e.g., to prefetch instructions).

But what happens if there is no TLB entry with a matching virtual page number? This event is termed a *TLB miss* and, depending on the CPU architecture, is handled in one of two ways:

- **Hardware TLB miss handling:** In this case, the CPU goes ahead and walks the page table to find the right PTE. If the PTE can be found and is marked **present**, then the CPU installs the new translation in the TLB. Otherwise, the CPU raises a page fault and hands over control to the operating system.

- **Software TLB miss handling:** In this case, the CPU simply raises a *TLB miss fault*. The fault is intercepted by the operating system, which invokes the TLB miss handler in response. The miss handler then walks the page table in software and, if a matching PTE that is marked **present** is found, the new translation is inserted in the TLB. If the PTE is not found, control is handed over to the page fault handler.

Whether a TLB miss is handled in hardware or in software, the bottom line is that miss handling results in a page-table walk and if a PTE that is marked **present** can be found, the TLB is updated with the new translation. Most CISC architectures (such as IA-32) perform TLB miss handling in hardware, and most RISC architectures (such as Alpha) use a software approach. A hardware solution is often faster, but is less flexible. Indeed, the performance advantage may be lost if the hardware poorly matches the needs of the operating system. As we see later, IA-64 provides a hybrid solution that retains much of the flexibility of the software approach without sacrificing the speed of the hardware approach.

TLB replacement policy

Let us now consider what should happen when the TLB is full (all entries are in use) and the CPU or the TLB miss handler needs to insert a new translation. The question now is, Which existing entry should be evicted (overwritten) to make space for the new entry? This choice is governed by the *TLB replacement policy*. Usually, some form or approximation of LRU is used for this purpose. With LRU, the TLB entry that has not been used the longest is the one that is evicted from the TLB. The exact choice of replacement policy often depends on whether the policy is implemented in hardware or in software. Hardware solutions tend to use simpler policies, such as *not recently used (NRU)*, whereas software solutions can implement full LRU or even more sophisticated schemes without much difficulty.

Note that if TLB miss handling is implemented in hardware, the replacement policy obviously also must be implemented in hardware. However, with a software TLB miss handler, the replacement policy can be implemented either in hardware or in software. Some architectures (e.g., MIPS) employ software replacement, but many newer architectures, including IA-64, offer a hardware replacement policy.

Removing old entries from the TLB

A final challenge in using a TLB is how to keep it synchronized (or *coherent*) with the underlying page table. Just as with any other cache, care must be taken to avoid cases where the TLB contains stale entries that are no longer valid. Stale entries can result from a number of scenarios. For example, when a virtual page is paged out to disk, the PTE in the page table is marked **not present**. If that page still has a TLB entry, it is now stale (because we assumed that the TLB contains only **present** PTEs). Similarly, a process might map a file into memory, access a few pages in the mapped area, and then unmap the file. At this point, the TLB may still contain entries that were inserted when the mapped area was accessed, but because the mapping no longer exists, those entries are now stale. The event that by far creates the most stale entries occurs when execution switches from one process to another. Because each process has its own address space, the *entire* TLB becomes stale on a context switch!

Given the number and complexity of the scenarios that can lead to stale entries, it is up to the operating system to ensure that they are flushed from the TLB before they can cause any harm. Depending on the CPU architecture, different kinds of TLB flush instructions are provided. Typical instructions flush the entire TLB, the entry for a specific virtual page, or all TLB entries that fall in a specific address range.

Note that a context switch normally requires that the entire TLB be flushed. However, because this is such a common operation and because TLB fault handling is relatively slow, CPU architects over the years have come up with various schemes to avoid this problem. These schemes go by various names, such as *address-space numbers*, *context numbers*, or *region IDs*, but they all share the basic idea: The tag used for matching a TLB entry is expanded to contain not just the virtual page number but also an address-space number that uniquely identifies the process (address space) to which the translation belongs. The CPU is also extended to contain a new register, asn, that identifies the address-space number of the currently executing process. Now, when the TLB is searched, the CPU ignores entries whose unique number does not match the value in the asn register. With this setup, a context switch simply requires updating the asn register—no flushing is needed anymore. Effectively, this scheme makes it possible to share the TLB across multiple processes.

4.4.1 The IA-64 TLB architecture

The IA-64 architecture uses an interesting approach to speed up virtual-to-physical address translation. Apart from the basic TLB, there are three other hardware structures, two of which, the *region registers* and the *protection key registers*, are designed to increase the effectiveness of the TLB. The third, the *virtual hash page table walker (VHPT walker)* is designed to reduce the penalty of a TLB miss.

Figure 4.29 illustrates how an IA-64 CPU translates a virtual address to a physical address. Let us start in the upper-right corner of the figure. There, we find a virtual address that has been divided into the three fields: the virtual region number vrn, the virtual page number vpn, and the page offset field offset. As usual, the page offset does not participate in the translation and is copied straight to the offset field of the physical address at the lower-right corner of the figure. In contrast, the 3-bit region number vrn is first sent to the region registers in the upper-left corner. Here, the region register indexed by vrn is read, and the resulting *region ID* value is sent to the TLB. At the TLB, the region ID is combined with the virtual page number vpn and the resulting region ID/vpn key is used to search the TLB. If an entry matches the search key, the remaining fields of the entry provide the information

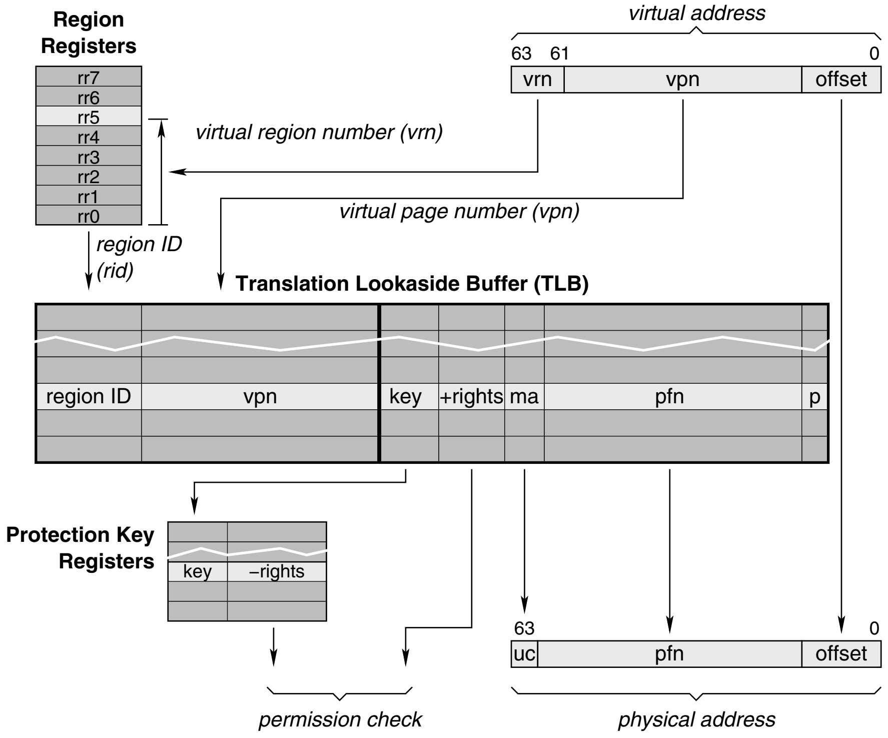


Figure 4.29. IA-64 virtual address translation hardware.

necessary to complete the address translation. Specifically, the pfn field provides the page frame number associated with the virtual page number. This field, too, can be copied down to the corresponding field in the physical address. The memory attribute field ma determines whether or not the memory access can be cached. If it can, the uc field (bit 63) of the physical address is cleared; otherwise, it is set. The final two fields, +rights and key, are used to check whether the memory access is permitted. The +rights field provides a set of positive rights that control what kind of accesses (read, write, or execute) are permitted at what privilege level (user and/or kernel). The key field is sent to the protection key registers. There, the register with a matching key value is read, and its -rights field supplies the negative rights needed to complete the permission check. Specifically, any kind of access specified in the -rights field is prohibited, even if it would otherwise be permitted by the +rights field. If there is no register matching the key value, a KEY MISS FAULT is raised. The operating system can intercept this fault and decide whether to install the missing key or take some other action (such as terminate the offending process). At this point the CPU has both the physical address and the information necessary to check whether the memory access is permitted, so the translation is complete.

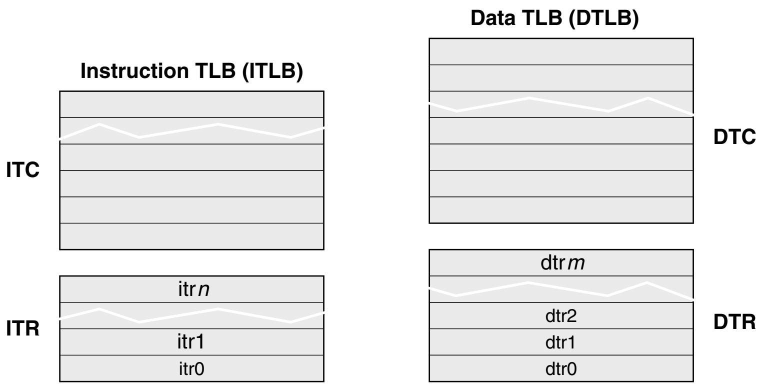


Figure 4.30. IA-64 TLB structure.

A somewhat unusual aspect of IA-64 is that the **present** bit is also part of the TLB entry. The Linux kernel never inserts a translation for a page that is not present, but the VHPT walker may do so.

TLB structure and management policies

As Figure 4.30 illustrates, the IA-64 TLB is divided into four logically separate units. On the left side is the *instruction TLB (ITLB)*, which translates instruction addresses; on the right side is the *data TLB (DTLB)*, which translates data addresses. Both the ITLB and the DTLB are further subdivided into *translation registers* (*ITR* and *DTR*) and *translation caches* (*ITC* and *DTC*). The difference between the two lies in where the replacement policy is implemented: for the translation caches, the hardware (CPU) implements the replacement policy, whereas for translation registers, the replacement policy is implemented in software. In other words, when a TLB entry is inserted into a translation register, both the TLB entry and the translation register name (e.g., itr1) have to be specified. In contrast, insertion into a translation cache requires specification of only the TLB entry—the hardware then picks an existing entry and replaces it with the new one.

The architecture guarantees that the ITC and DTC have a size of at least one entry. Of course, a realistic CPU typically supports dozens of entries in each cache. For example, Itanium implements 96 ITC entries and 128 DTC entries. Even so, to guarantee forward progress the operating system must never assume that more than one entry can be held in the cache at any given time. Otherwise, when inserting two entries back to back, the hardware replacement policy may end up replacing the first entry when inserting the second one. Thus, the operating system must be written in a way that ensures forward progress even if only the second entry survives.

Both ITR and DTR are guaranteed to support at least eight translation registers. However, the IA-64 architecture leaves hardware designers the option to implement translation registers in the form of translation cache entries that are marked so that they are never con-

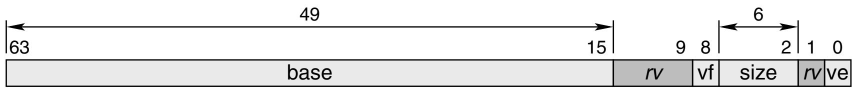


Figure 4.31. Format of IA-64 page table address register pta.

sidered for replacement by the hardware. With this option, the more translation registers used, the fewer entries available in the translation cache. For this reason, it is generally preferable to allocate only as many translation registers as are really needed and to allocate them in order of increasing register index.

Linux/ia64 uses translation registers to pin certain critical code sections and data structures. For example, one ITR entry is used to pin the TLB fault handlers of the kernel, and another is used to map firmware code that cannot risk taking a TLB miss fault. Similarly, the kernel uses a DTR entry to pin the kernel stack of the currently running process.

The VHPT walker and the virtually-mapped linear page table

One question we have glossed over so far is what happens when there is no matching TLB entry for a given region ID/vpn pair, i.e., when a TLB miss occurs. On IA-64, this event can be handled in one of two ways: if enabled, the VHPT walker becomes active and attempts to fill in the missing TLB entry. If the VHPT walker is disabled, the CPU signals a TLB miss fault, which is intercepted by the Linux kernel. The details of how a TLB miss is handled in software is described in Section 4.5. For now, let us focus on the case where the TLB miss is handled by the VHPT walker.

First, let us note that the use of the VHPT walker is completely optional. If an operating system decides not to use it, IA-64 leaves the operating system complete control over the page-table structure and the PTE format. In order to use the VHPT walker, the operating system may need to limit its choices somewhat. Specifically, the VHPT walker can support one of two modes: hashed mode or linear-page-table mode. In hashed mode, the operating system should use a hash table as its page table and the PTEs have a format that is known as the *long format*. With the long format, each PTE is 32 bytes in size. In linear-page-table mode, the operating system needs to be able to support a virtually-mapped linear page table and the PTEs have a format known as the *short format*. The short format is the one Linux uses. As shown in Figure 4.25, this type of PTE is 8 bytes in size.

The VHPT walker configuration is determined by the *page table address* control register pta. This register is illustrated in Figure 4.31. Bit ve controls whether or not the VHPT walker is enabled. The operation of the VHPT walker is further qualified by a control bit in each region register. Only when a TLB miss occurs in a region for which this control bit *and* pta.ve are both 1 does the VHPT walker become active. The second control bit in pta is vf; it determines whether the walker operates in hashed (long-format) or virtually-mapped linear page table (short-format) mode. A value of 1 indicates hashed mode. Let us assume that ve is 1 (enabled) and that vf is 0 (virtually-mapped linear page table mode). With this configuration, the base and size fields define the address range that the linear page table

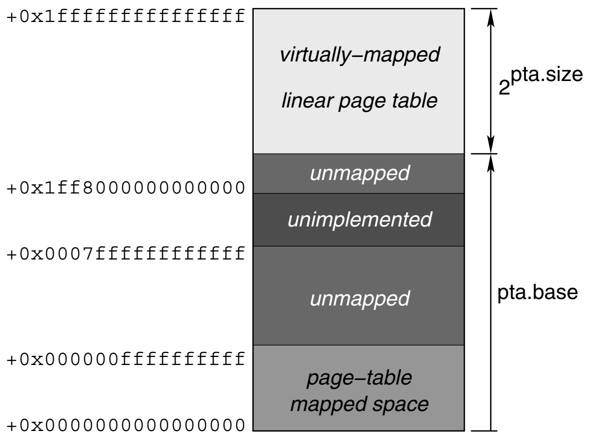


Figure 4.32. Virtually-mapped linear page table inside a Linux/ia64 region.

occupies in each region. The base field contains the 49 most significant bits of the region-relative offset at which the table starts, and the size field contains the number of address bits that the table spans (i.e., the table is $2^{\text{pta.size}}$ bytes long).

Note that while the VHPT walker can be disabled separately in each region (by a bit in the region registers), for those regions in which it is enabled, the linear page table is mapped at the same relative address range in each region. Given this constraint, the location at which the linear page table is placed needs to be chosen carefully.

Figure 4.32 illustrates the solution that Linux/ia64 is using. Two factors influence the placement of the linear page table. First, so that virtual address space is not wasted, it is preferable for the page table to not overlap with the normal space mapped by the page table. The latter space is illustrated in the figure by the rectangle at the bottom of the region. As usual, the addresses listed next to it are valid for the case in which a three-level page table is used with a page size of 8 Kbytes. Second, the page table also may not overlap with the address-space hole in the middle of the region that exists if the CPU does not implement all address bits (as determined by the IMPL_VA_MSB parameter). This hole is illustrated in the figure by the dark-shaded rectangle in the middle of the region. The addresses listed next to it are valid for the case where IMPL_VA_MSB = 50. With these two factors in mind, Linux/ia64 sets up the pta register such that the linear page table is mapped at the top end of the region, as illustrated by the lightly shaded rectangle. Note that for certain combinations of page sizes and IMPL_VA_MSB, the page-table-mapped space might either cross over into the unimplemented space or might overlap with the virtually-mapped linear page table (when IMPL_VA_MSB = 60). The Linux kernel checks for this at boot time, and if it detects an overlap condition, it prints an error message and halts execution.

Now, let us take a look at how the VHPT walker operates. When a TLB miss occurs for virtual address *va*, the walker calculates the virtual address va' of the PTE that maps *va*.

Using the virtually-mapped linear page table, this address is given by:

$$va' = \lfloor va/2^{61} \rfloor \cdot 2^{61} + \mathsf{pta.base} \cdot 2^{15} + 8 \cdot \left(\lfloor va/\mathsf{PAGE\_SIZE} \rfloor \bmod 2^{\mathsf{pta.size}} \right)$$

That is, va' is the sum of the region's base address, the region offset of the linear page table, and the offset of the PTE within the linear page table. In the last summand, the factor of 8 is used because each PTE has a size of 8 bytes, and the modulo operation truncates away the most significant address bits that are not mapped by the page table. The VHPT walker then attempts to read the PTE stored at this address. Because this is again a virtual address, the CPU goes through the normal virtual-to-physical address translation. If a TLB entry exists for va', the translation succeeds and the walker can read the PTE from physical memory and install the PTE for va. However, if the TLB entry for va' is also missing, the walker gives up and requests assistance by raising a VHPT TRANSLATION FAULT.

Let us emphasize that the VHPT walker *never* walks the Linux page table. It could not possibly do so because it has no knowledge of the page-table structure used by Linux. For example, it does not know how many levels the page-table tree has or how big each directory is. But why use the VHPT walker at all given that it can handle a TLB miss only if the TLB entry for the linear page table is already present? The reason is spatial locality of reference. Consider that the TLB entry that maps a particular PTE actually maps an entire page of PTEs. Thus, after a TLB entry for a page-table page is installed, all TLB misses that access PTEs in the same page can be handled entirely by the VHPT walker, avoiding costly TLB miss faults. For example, with a page size of 8 Kbytes, each page-table TLB entry maps 8 Kbytes$/8 = 1024$ PTEs and hence $1024 \cdot 8$ Kbytes $=$ 8 Mbytes of memory. In other words, when accessing memory sequentially, the VHPT walker reduces TLB miss faults from one per page to only one per 1024 pages! Given the high cost of fielding a fault on modern CPUs, the VHPT walker clearly has the potential to dramatically increase performance.

On the other hand, if memory is accessed in an extremely sparse pattern, the linear page table can be disadvantageous because the TLB entries for the page table take up space without being of much benefit. For example, again assuming a page size of 8 Kbytes, the most extreme case would occur when one byte is accessed every 8 Mbytes. In this case, each memory access would require two TLB entries: one for the page being accessed and one for the corresponding page-table page. Thus, the effective size of the TLB would be reduced by a factor of two! Fortunately, few applications exhibit such extreme access patterns for prolonged periods of time, so this is usually not an issue.

On a final note, it is worth pointing out that the virtually-mapped linear page table used by Linux/ia64 is *not* a self-mapped virtual page table (see Section 4.3.2). The two are very similar in nature, but the IA-64 page table does not have a self-mapped entry in the global directory. The reason is that none is needed: The self-mapped entry really is needed only if the virtual page table is used to access global- and middle-directory entries. Since Linux/ia64 does not do that, it needs no self-mapping entry. Another way to look at this situation is to think of the virtual page table as existing in the TLB only: If a page in the virtual page table happens to be mapped in the TLB, it can be used to access the PTE directory, but if it is not mapped, an ordinary page-table walk is required.

Linux/ia64 and the region and protection key registers

Let us now return to Figure 4.29 on page 177 and take a closer look at the workings of the region and protection key registers and how Linux uses them. Both register files are under complete control of the operating system; the IA-64 architecture does not dictate a particular way of using them. However, they clearly were designed with a particular use in mind. Specifically, the region registers provide a means to share the TLB across multiple processes (address spaces). For example, if a unique region ID is assigned to each address space, the TLB entries for the same virtual page number vpn of different address spaces can reside in the TLB at the same time because they remain distinguishable, thanks to the region ID. With this use of region IDs, a context switch no longer requires flushing of the entire TLB. Instead, it is simply necessary to load the region ID of the new process into the appropriate region registers. This reduction in TLB flushing can dramatically improve performance for certain applications. Also, because each region has its own region register, it is even possible to have portions of different address spaces active at the same time. The Linux kernel takes advantage of this by permanently installing the region ID of the kernel in rr5–rr7 and installing the region ID of the currently running process in rr0–rr4. With this setup, kernel TLB entries and the TLB entries of various user-level processes can coexist without any difficulty and without wasting any TLB entries.

Whereas region registers make it possible to share the entire TLB across processes, protection key registers enable the sharing of *individual* TLB entries across processes, even if the processes must have distinct access rights for the page mapped by the TLB entry. To see how this works, suppose that a particular TLB entry maps a page of a shared object. The TLB entry for this page would be installed with the access rights (+rights) set according to the needs of the owner of the object. The key field would be set to a value that uniquely identifies the shared object. The operating system could then grant a process restricted access to this object by using one of the protection key registers to map the object's unique ID to an appropriate set of negative rights (-rights). This kind of fine-grained sharing of TLB entries has the potential to greatly improve TLB utilization, e.g., for shared libraries. However, the short-format mode of the VHPT walker cannot take advantage of the protection key registers and, for this reason, Linux disables them by clearing the pk bit in the processor status register (see Chapter 2, *IA-64 Architecture*).

4.4.2 Maintenance of TLB coherency

For proper operation of the Linux kernel to be guaranteed, the TLB must be kept coherent with the page tables. For this purpose, Linux defines an interface that abstracts the platform differences of how entries are flushed from the TLB. The interface used for this purpose, shown in Figure 4.33, is called the *TLB flush interface* (file include/asm/pgalloc.h).

The first routine in this interface is *flush_tlb_page()*. It flushes the TLB entry of a particular page. The routine takes two arguments, a vm-area pointer *vma* and a virtual address *addr*. The latter is the address of the page whose TLB entry is being flushed, and the former points to the vm-area structure that covers this page. Because the vm-area structure contains a link back to the mm structure to which it belongs, the *vma* argument indirectly also identifies the address space for which the TLB entry is being flushed.

flush_tlb_page(*vma*, *addr*);	*/* flush TLB entry for virtual address addr */*
flush_tlb_range(*mm*, *start*, *end*);	*/* flush TLB entries in address range */*
flush_tlb_pgtables(*mm*, *start*, *end*);	*/* flush virtual-page-table TLB entries */*
flush_tlb_mm(*mm*);	*/* flush TLB entries of address space mm */*
flush_tlb_all();	*/* flush the entire TLB */*
update_mmu_cache(*vma*, *addr*, *pte*);	*/* proactively install translation in TLB */*

Figure 4.33. Kernel interface to maintain TLB coherency.

The second routine, *flush_tlb_range()*, flushes the TLB entries that map virtual pages inside an arbitrary address range. It takes three arguments: an mm structure pointer *mm*, a start address *start*, and an end address *end*. The *mm* argument identifies the address space for which the TLB entries should be flushed, and *start* and *end* identify the first and the last virtual page whose TLB entries should be flushed.

The third routine, *flush_tlb_pgtables*, flushes TLB entries that map the virtually-mapped linear page table. Platforms that do not use a virtual page table do not have to do anything here. For the other platforms, argument *mm* identifies the address space for which the TLB entries should be flushed, and arguments *start* and *end* specify the virtual address range for which the virtual-page-table TLB entries should be flushed.

The fourth routine, *flush_tlb_mm()*, flushes all TLB entries for a particular address space. The address space is identified by argument *mm*, which is a pointer to an mm structure. Depending on the capabilities of the platform, this routine can either truly flush the relevant TLB entries or simply assign a new address-space number to *mm*.

Note that the four routines discussed so far may flush more than just the requested TLB entries. For example, if a particular platform does not have an instruction to flush a specific TLB entry, it is safe to implement *flush_tlb_page()* such that it flushes the entire TLB.

The next routine, *flush_tlb_all()*, flushes the entire TLB. This is a fallback routine in the sense that it can be used when none of the previous, more fine-grained routines are suitable. By definition, this routine flushes even TLB entries that map kernel pages. Because this is the only routine that does this, any address-translation-related changes to the page-table-mapped kernel segment or the kmap segment must be followed by a call to this routine. For this reason, calls to *vmalloc()* and *vfree()* are relatively expensive.

The last routine in this interface is *update_mmu_cache()*. Instead of flushing a TLB entry, it can be used to proactively install a new translation. The routine takes three arguments: a vm-area pointer *vma*, a virtual address *addr*, and a page-table entry *pte*. The Linux kernel calls this routine to notify platform-specific code that the virtual page identified by *addr* now maps to the page frame identified by *pte*. The *vma* argument identifies the vm-area structure that covers the virtual page. This routine gives platform-specific code a hint when the page table changes. Because it gives only a hint, a platform is not required to do anything. The platform could use this routine either for platform-specific purposes or to aggressively update the TLB even before the new translation is used for the first time. However, it is important to keep in mind that installation of a new translation generally displaces an existing entry from the TLB, so whether or not this is a good idea depends both on the applications in use and on the performance characteristics of the platform.

IA-64 implementation

On Linux/ia64, *flush_tlb_mm()* is implemented such that it forces the allocation of a new address-space number (region ID) for the address space identified by argument *mm*. This is logically equivalent to purging all TLB entries for that address space, but it has the advantage of not requiring execution of any TLB purge instructions.

The *flush_tlb_all()* implementation is based on the ptc.e (purge translation cache entry) instruction, which purges a large section of the TLB. Exactly how large a section of the TLB is purged depends on the CPU model. The architected sequence to flush the entire TLB is as follows:

```
long flags, i, j, addr = BASE_ADDR;
local_irq_save(flags);              /* disable interrupts */
for (i = 0; i < COUNT0; ++i, addr += STRIDE0) {
   for (j = 0; j < COUNT1; ++j, addr += STRIDE1)
      ptc_e(addr);
}
local_irq_restore(flags);           /* reenable interrupts */
```

Here, BASE_ADDR, COUNT0, STRIDE0, COUNT1, and STRIDE1 are CPU-model-specific values that can be obtained from PAL firmware (see Chapter 10, *Booting*). The advantage of using such an architected loop instead of an instruction that is guaranteed to flush the entire TLB is that ptc.e is easier to implement on CPUs with multiple levels and different types of TLBs. For example, a particular CPU model might have two levels of separate instruction and data TLBs, making difficult to clear all TLBs atomically with a single instruction. However, in the particular case of Itanium, COUNT0 and COUNT1 both have a value of 1, meaning that a single ptc.e instruction flushes the entire TLB.

All other flush routines are implemented with either the ptc.l (purge local translation cache) or the ptc.ga (purge global translation cache and ALAT) instruction. The former is used on UP machines, and the latter is used on MP machines. Both instructions take two operands—a start address and a size—which define the virtual address range from which TLB entries should be purged. The ptc.l instruction affects only the local TLB, so it is generally faster to execute than ptc.ga, which affects the entire machine (see the architecture manual for exact definition [26]). Only one CPU can execute ptc.ga at any given time. To enforce this, the Linux/ia64 kernel uses a spinlock to serialize execution of this instruction.

Linux/ia64 uses *update_mmu_cache()* to implement a cache flushing that we discuss in more detail in Section 4.6. The routine could also be used to proactively install a new translation in the TLB. However, Linux gives no indication whether the translation is needed for instruction execution or for a data access, so it would be unclear whether the translation should be installed in the instruction or data TLB (or both). Because of this uncertainty and because installing a translation always entails the risk of evicting another, perhaps more useful, TLB entry, it is better to avoid proactive installation of TLB entries.

```
int init_new_context(task, mm);   /* force new address-space number for mm */
get_mmu_context(mm);              /* allocate address-space number if necessary */
reload_context(mm);               /* activate address-space number of mm */
destroy_context(mm);              /* free address-space number */
```

Figure 4.34. Kernel interface to manage address-space numbers.

4.4.3 Lazy TLB flushing

To avoid the need to flush the TLB on every context switch, Linux defines an interface that abstracts differences in how address-space numbers (ASNs) work on a particular platform. This interface is called the *ASN interface* (file include/asm/mmu_context.h), shown in Figure 4.34. Support for this interface is optional in the sense that platforms with no ASN support simply define empty functions for the routines in this interface and instead define *flush_tlb_mm()* in such a way that it flushes the entire TLB.

Each mm structure contains a component called the *mm context*, which has a platform-specific type called the *mm context type* (mm_context_t in file include/asm/mmu.h). Often, this type is a single word that stores the ASN of the address space. However, some platforms allocate ASNs in a CPU-local fashion. For those, this type is typically an array of words such that the ith entry stores the ASN that the address space has on CPU i.

When we take a look at Figure 4.34, we see that it defines four routines. The first routine, *init_new_context()*, initializes the mm context of a newly created address space. It takes two arguments: a task pointer *task* and an mm structure pointer *mm*. The latter is a pointer to the new address space, and the former points to the task that created it. Normally, this routine simply clears the mm context to a special value (such as 0) that indicates that no ASN has been allocated yet. On success, the routine should return a value of 0.

The remaining routines all take one argument, an mm structure pointer *mm* that identifies the address space and, therefore, the ASN that is to be manipulated. The *get_mmu-_context()* routine ensures that the mm context contains a valid ASN. If the mm context is already valid, nothing needs to be done. Otherwise, a free (unused) ASN needs to be allocated and stored in the mm context. It would be tempting to use the process ID (pid) as the ASN. However, this does not work because the *execve()* system call creates a new address space without changing the process ID.

Routine *reload_context()* is responsible for activating on the current CPU the ASN represented by the mm context. Logically, this activation entails writing the ASN to the CPU's asn register, but the exact details of how this is done are, of course, platform specific. When this routine is called, the mm context is guaranteed to contain a valid ASN.

Finally, when an ASN is no longer needed, Linux frees it by calling *destroy_context()*. This call marks the ASN represented by the mm context as available for reuse by another address space and should free any memory that may have been allocated in *get_mmu_context()*. Even though the ASN is available for reuse after this call, the TLB may still contain old translations with this ASN. For correct operation, it is essential that platform-specific code purges these old translations before activating a reused ASN. This is usually achieved by allocating ASNs in a round-robin fashion and flushing the entire TLB before wrapping around to the first available ASN.

IA-64 implementation

Linux/ia64 uses region IDs to implement the ASN interface. The mm context in the mm structure consists of a single word that holds the region ID of the address space. A value of 0 means that no ASN has been allocated yet. The *init_new_context()* routine can therefore simply clear the mm context to 0.

The IA-64 architecture defines region IDs to be 24 bits wide, but, depending on CPU model, as few as 18 bits can be supported. For example, Itanium supports just the architectural minimum of 18 bits. In Linux/ia64, region ID 0 is reserved for the kernel, and the remaining IDs are handed out by *get_mmu_context()* in a round-robin fashion. After the last available region ID has been handed out, the entire TLB is flushed and a new range of available region IDs is calculated such that all region IDs currently in use are outside this range. Once this range has been found, the *get_mmu_context()* continues to hand out region IDs in a round-robin fashion until the space is exhausted again, at which point the steps of flushing the TLB and finding an available range of region IDs are repeated.

Region IDs are 18 to 24 bits wide but only 15 to 21 bits are effectively available for Linux. The reason is that IA-64 requires the TLB to match the region ID and the virtual page number (see Figure 4.29 on page 177) but not necessarily the virtual region number (vrn). Thus, a TLB may not be able to distinguish, e.g., address `0x2000000000000000` from address `0x4000000000000000` unless the region IDs in rr1 and rr2 are different. To ensure this, Linux/ia64 encodes vrn in the three least significant bits of the region ID.

Note that the region IDs returned by *get_mmu_context()* are shared across all CPUs. Such a global region ID allocation policy is ideal for UP and small to moderately large MP machines. A global scheme is advantageous for MP machines because it makes possible the use of the ptc.ga instruction to purge translations from all TLBs in the machine. On the downside, global region ID allocation is a potential point of contention and, perhaps worse, causes the region ID space to be exhausted faster the more CPUs there are in the machine. To see this, assume there are eight distinct region IDs and that a CPU on average creates a new address space once a second. With a single CPU, the TLB would have to be flushed once every eight seconds because the region ID space has been exhausted. In a machine with eight CPUs, each creating an address space once a second, a global allocation scheme would require that the TLB be flushed once every second. In contrast, a local scheme could get by with as little TLB flushing as the UP case, i.e., one flush every eight seconds. But it would have to use an interprocessor interrupt (IPI, see Chapter 8, *Symmetric Multiprocessing*) instead of ptc.ga to perform a global TLB flush (because each CPU uses its own set of region IDs). In other words, deciding between the local and the global scheme involves a classic tradeoff between smaller fixed overheads (ptc.ga versus IPI) and better scalability (region ID space is exhausted with a rate proportional to total versus per-CPU rate at which new address spaces are created).

On IA-64, *reload_context()* has the effect of loading region registers rr0 to rr4 according to the value in the mm context. As explained in the previous paragraph, the value actually stored in the region registers is formed by shifting the mm context value left by three bits and encoding the region's vrn value in the least significant bits.

The IA-64 version of *destroy_context()* does not need to do anything: *get_mmu_context()* does not allocate any memory, so no memory needs to be freed here. Similarly, the range of available region IDs is recalculated only after the existing range has been exhausted, so there is no need to flush old translations from the TLB here.

4.5 PAGE FAULT HANDLING

A page fault occurs when a process accesses a virtual page for which there is no PTE in the page table or whose PTE in some way prohibits the access, e.g., because the page is not present or because the access is in conflict with the access rights of the page. Page faults are triggered by the CPU and handled in the *page fault handler*.

Because Linux uses demand paging and page-fault-based optimizations such as copy-on-write, page faults occur during the normal course of operation and do not necessarily indicate an error. Thus, when the page fault handler is invoked, it first needs to determine whether the page fault is caused by an access to a valid page. If not, the page fault handler simply sends a segmentation violation signal to the faulting process and returns. Otherwise, it takes one of several possible actions:

- If the page is being accessed for the first time, the handler allocates a new page frame and initializes it, e.g., by reading its content from disk. Page faults caused by first-time accesses are called *demand page faults*.
- If the page has been paged out to swap space, the handler reads it back from disk into a newly allocated page frame.
- If the page fault occurred because of a page-fault-based optimization (such as copy-on-write), the handler takes the appropriate recovery action (such as performing the delayed page copy).

Each of these actions results either in a new or updated page, and the handler must accordingly either create or update the PTE in the page table. Because the page-table tree is also created on demand, installing a new PTE may require allocating and initializing a middle and a PTE directory (the global directory is guaranteed to exist already). The page fault handler can use the *pmd_alloc()* and *pte_alloc()* routines from Section 4.3.6 for this purpose. Before the handler installs the PTE, it updates the **accessed** and **dirty** bits as well. Since the page fault itself is an indication that the page is being accessed, the handler uses *pte_mkyoung()* to unconditionally turn on the **accessed** bit. On the other hand, the **dirty** bit is turned on by *pte_mkdirty()* only if the page fault is the result of a write access. After updating the page table, the page fault handler returns and execution resumes in the faulting process. There, the instruction that previously faulted is restarted and, thanks to the updated page table, it can now complete execution.

The above description glanced over two important questions: how does the kernel determine whether an access is valid and, if it is, how does it determine what action to take?

To verify the validity of an access, the page fault handler must search the vm-area list (or the AVL tree if it exists) for the vm-area that covers the page being accessed. If the vm-area exists and its access right flags (VM_READ, VM_WRITE, and VM_EXEC) permit the access, then the access is valid. The answer to the second question is determined primarily by the state of the page table as it exists on entry to the handler:

1. The handler can determine whether it is dealing with a demand page fault by checking whether the PTE exists in the page table. If either *pmd_none()* or *pte_none()* returns **true**, then the PTE does not yet exist and the fault is a demand page fault. The exact action to be taken for such a fault depends on the vm-area that covers the page. If this vm-area defines its own *nopage()* callback, the handler deals with the fault by invoking the callback; otherwise, it handles the fault by creating an anonymous page. The *nopage()* handler often ends up reading the content of the page from a file, but more complex actions, such as reading the page from a remote machine, are also possible.

2. The **present** bit of the PTE tells the handler whether a page has been paged out to swap space. If the bit is cleared (*pte_present()* returns **false**), the handler knows that it is dealing with a page that has been paged out. The appropriate action is for the handler to read the page back in from swap space.

3. Finally, page-fault-based optimizations manifest themselves as differences between the PTE and the vm-area for the page. For example, a copy-on-write page can be identified by the fact that the PTE disallows write accesses (*pte_write()* returns **false**), yet the vm-area permits them (the VM_WRITE access right is enabled). The appropriate action to handle such faults depends on the exact optimization that is being performed. For a copy-on-write page, the handler would have to perform the delayed page copy.

The discussion so far illustrates that page fault handling is a complex operation: it may involve reading a page from swap space, from an arbitrary filesystem, or even from a remote machine. Fortunately, as we see in Section 4.5.2, the platform-specific part of the kernel only has to handle those aspects that are unique to the platform, because all other work is taken care of by the Linux page fault handler.

4.5.1 Example: How copy-on-write really works

To get a better idea of how the page fault handler coordinates its work with the rest of the kernel, let us look at how Linux handles copy-on-write pages. Let us assume a process *A* with a single region of virtual memory that is writable and that occupies the address range from `0x8000` to `0xe000`. Let us further assume that only the page at address `0xa000` is resident in the virtual memory. Figure 4.35 (a) illustrates this case. At the top left, we find the task structure labeled *task A*. The task structure contains a pointer to the mm structure that represents the address space of this process. In the box labeled *mm*, we find a pointer to the vm-area list, labeled *mmap*, and a pointer to the page table, labeled *pgd*. In reality, *pgd* points to the global directory of the page table. However, for simplicity the figure represents

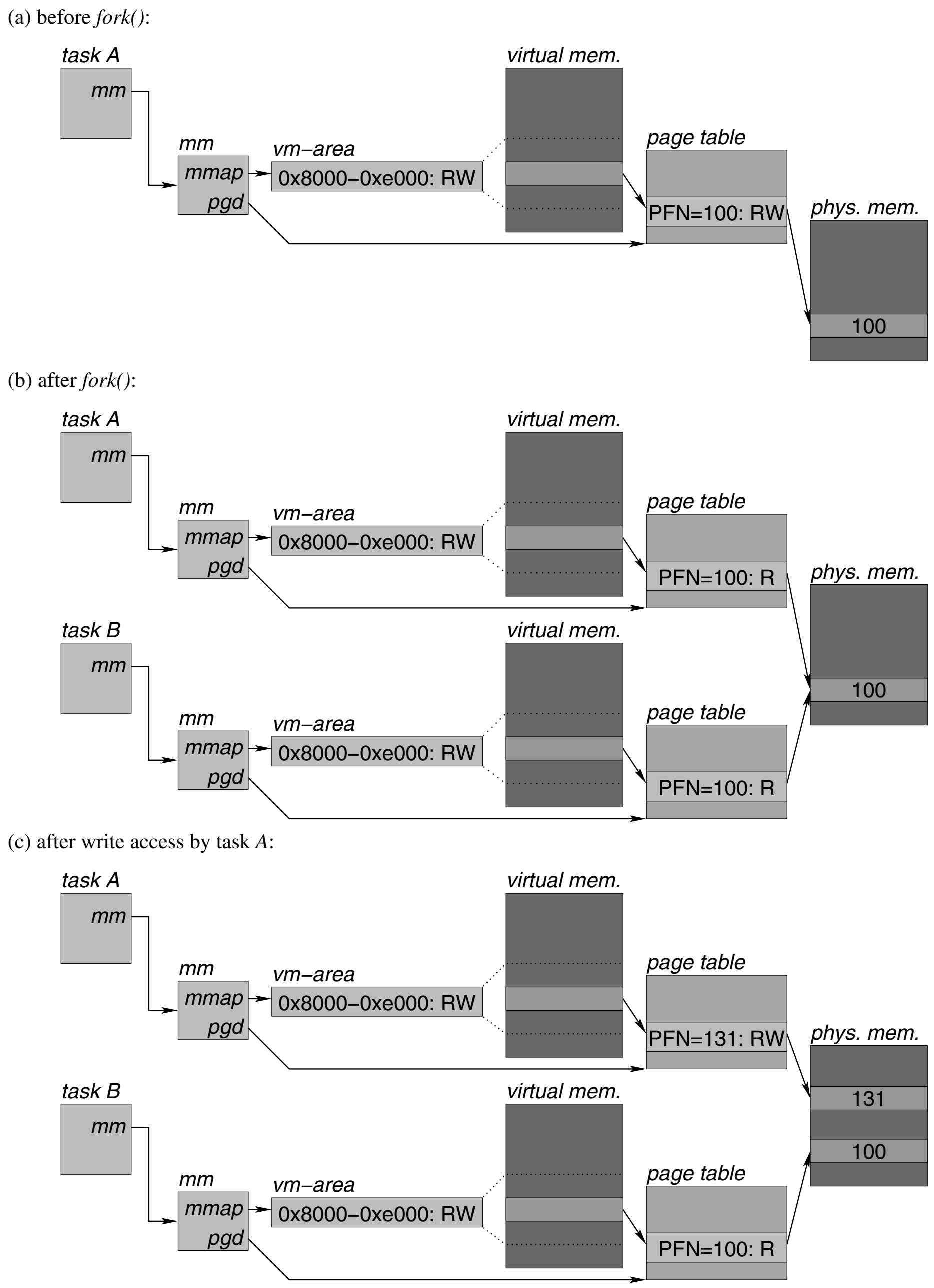


Figure 4.35. Example of a copy-on-write operation.

the page table as the linear table labeled *page table*. Because the process has only a single region mapped in the virtual memory, the vm-area list has just a single entry, represented by the box labeled *vm-area*. To keep the complexity of the figure within reason, the vm-area shows only the starting and ending address of the region and the access rights (RW, for read and write access). Because we assumed that the page at `0xa000` is resident, it has an entry in the page table and maps to some page frame. In the figure, we assumed that the virtual page is backed by physical page frame 100. Since the page is both readable and writable, the PTE for this page has the permission bits set to RW.

Now let us assume that process *A* invokes the *clone2()* system call without specifying the CLONE_VM flag. In a traditional UNIX system, this would be equivalent to calling *fork()* and has the effect of creating a new process that is a copy of the caller. The state as it would exist after *clone2()* returns is illustrated in Figure 4.35 (b). As the figure shows, the new process has its own task structure, labeled *task B* and its own copies of *mm*, *vm-area*, and *page table*. The two processes are identical, but note how *clone2()* turned off write permission in the PTEs of both the parent and the child. It does this for every writable PTE in an attempt to delay the copying of writable pages as long as possible. Turning off write permission in the PTEs of writable pages is the first step of a copy-on-write; it ensures that neither process can write to the page without first causing a page fault. Note that the access permissions in the vm-area structures remain unchanged at RW.

So what happens when one of the processes attempts to write to the virtual page at `0x8000`? Suppose process *A* writes to the page first. Because the PTE allows only read accesses, this write triggers a page fault. The page fault handler goes through the steps described in the previous section and first locates the matching vm-area. It then checks whether the vm-area permits write accesses. Because the vm-area in process *A* still has the access rights set to RW, the write access is permitted. The page fault handler then checks whether the PTE exists in the page table and whether the PTE has the **present** bit on. Because the page is resident, both of these checks pass. In the last step, the page fault handler checks whether it is dealing with a write access to a page whose PTE does not permit write accesses. Because this is the case, the handler detects that it is time to copy a copy-on-write page. It proceeds by checking the page frame descriptor of page frame 100 to see how many processes are currently using this page. Because process *B* is also using this page frame, the count is 2 and the page fault handler decides that it must copy the page frame. It does this by first allocating a free page frame, say, page frame 131, copying the original frame to this new frame, and then updating the PTE in process *A* to point to page frame 131. Because process *A* now has a private copy of the page, the access permission in the PTE can be set to RW again. The page fault handler then returns, and at this point the write access can complete without any further errors. Figure 4.35 (c) illustrates the state as it exists at this point.

Note that the PTE in process *B* still has write permission turned off, even though it is now the sole user of page frame 100. This remains so until the process attempts a write access. When that happens, the page fault handler is invoked again and the same steps are repeated as for process *A*. However, when checking the page frame descriptor of page frame 100, it finds that there are no other users and goes ahead to turn on write permission in the PTE without first making a copy of the page.

4.5.2 The Linux page fault handler

The Linux kernel provides a platform-independent *page fault handler* (*handle_mm_fault()* in file mm/memory.c) that takes care of handling most aspects of page fault handling. Platform-specific code is responsible for intercepting any virtual-memory-related faults that a CPU may raise and invoking the handler as necessary. The interface of the Linux page fault handler is shown below:

```
int handle_mm_fault(mm, vma, addr, access_type);
```

The routine takes four arguments: *mm* is a pointer to the mm structure of the address space in which the fault occurred, *vma* is a pointer to the vm-area that covers the page that is being accessed, *addr* is the virtual address that caused the fault, and *access_type* is an integer indicating the type of access (read, write, or execute). The return value is an indication of how the page fault was handled. A return value of 1 signifies that the fault was handled successfully and that it should be counted as a *minor fault*. This means that the page installed was already in memory, e.g., because the page is in use by another process as well. A return value of 2 signifies that the fault was also handled successfully, but that it should be counted as a *major fault* because the page had to be read from disk, e.g., from a file or swap space. A value of 0 signifies that the page fault could not be handled properly and that the kernel should send a bus error signal (SIGBUS) to the process. Finally, a negative return value signifies that the kernel is completely out of memory and that the faulting process should be terminated immediately.

Before invoking *handle_mm_fault()*, platform-specific code must perform these steps:

1. Determine the virtual address *addr* that triggered the fault.
2. Determine the access type (read, write, or execute).
3. Verify that the fault occurred in user mode and, if so, get the mm structure pointer *mm* of the currently running process (task).
4. Find the vm-area structure that covers the page being accessed.
5. Verify that the access type is permitted by the vm-area structure.

If all of these steps are completed successfully, the Linux page fault handler can be invoked. After the handler returns, the platform-specific code is responsible to account the fault either as a minor or a major fault, depending on whether the return value was 1 or 2, respectively (this accounting info is stored in the task structure described in Chapter 3, *Processes, Tasks, and Threads*). If the return value is 0, a bus error signal must be sent to the process and if it is negative, the process must be terminated immediately (as if it had called *exit()*).

What should the platform-specific code do if one of the above steps fails? The answer depends on exactly *which* step failed. The first two steps (determining the fault address and access type) usually cannot fail.

If the third step fails, the page fault occurred in kernel mode. This is normally an indication of a kernel bug and results in a panic (kernel stops execution). However, the Linux kernel may legitimately cause page faults while copying data across the user/kernel boundary. Thus, when a page fault happens in kernel mode, platform-specific code must check whether it was the result of such a copy and, if so, initiate an appropriate recovery action. We describe in Chapter 5, *Kernel Entry and Exit*, exactly what this entails.

If the fourth step fails, there is no vm-area that covers the page being accessed; this is normally an indication of an attempt to access nonexistent virtual memory. If this failure occurs, the platform-specific code sends a segmentation violation signal to the process. There are two special cases, however: If the page accessed is just above a vm-area with the VM_GROWSUP flag set or just below a vm-area with the VM_GROWSDOWN flag set, the platform-specific code must expand the corresponding vm-area to include the page being accessed and then use this expanded vm-area to finish processing the page fault in the normal fashion. This mechanism is intended for automatic stack expansion and is permitted only if the resulting stack size does not exceed the stack size limit (RLIMIT_STACK) or the virtual-address-space limit (RLIMIT_AS) established by the *setrlimit()* system call.

If the fifth and last step fails, the process attempted to access the address space in an illegal fashion (e.g., it tried to execute a read-only page) and the platform-specific code again sends a segmentation violation signal to the process.

4.5.3 IA-64 implementation

When a virtual-memory-related fault occurs on IA-64, the interruption handling described in Chapter 2, *IA-64 Architecture*, is initiated. Recall that, among other things, this handling switches the CPU to the most-privileged execution level (level 0), turns off the interruption collection (ic is cleared in psr), activates bank 0 of registers r16 to r31, and then hands over control to the appropriate handler in the interruption vector table (IVT).

The architecture defines a total of 13 virtual-memory-related faults, each with its own handler. A set of six faults handles TLB misses and another set of seven faults handles PTE-related faults. Three primary control registers pass information to these handlers:

- The *interruption status register* (isr) contains various flag bits that indicate the type of access that caused the fault. For example, three bits indicate whether the fault occurred as a result of a read, write, or execute access.

- The *interruption faulting address* (ifa) register contains the virtual address that caused the fault.

- The *interruption hash address* (iha) register is used when a TLB miss occurs as a result of the VHPT walker attempting to access the virtually-mapped linear page table. When this happens, the iha register contains the virtual address that the VHPT walker was attempting to access.

The fault handlers first read these control registers to determine how to handle the fault. The bank 0 registers are active at that time, so the handlers can use registers r16 to r31 for this purpose. Doing so avoids the need to save CPU state to memory. Indeed, whenever possible, the handlers attempt to complete the entire fault handling with only the bank 0 registers. However, because the bank 0 registers are available only while ic is off, more complicated

Table 4.4. IA-64 TLB miss faults.

Fault	*Description*
ITLB FAULT	Instruction access missed in the instruction TLB.
DTLB FAULT	Data access missed in the data TLB.
VHPT TRANSLATION FAULT	VHPT access missed in the data TLB.
ALTERNATE ITLB FAULT	Instruction access missed in the instruction TLB and the VHPT walker is disabled.
ALTERNATE DTLB FAULT	Data access missed in the data TLB and the VHPT walker is disabled.
DATA NESTED TLB FAULT	Data access missed in the TLB during execution with ic off.

fault handling, in particular any fault handling that requires invoking the Linux page fault handler, forces the handlers off the fast path and also forces them to save the essential CPU state in a pt-regs structure in memory (see Chapter 3, *Processes, Tasks, and Threads*). Once this state has been saved, control can be handed over to the Linux page fault handler.

TLB miss handlers

Table 4.4 lists the six types of TLB-related misses. As the table shows, IA-64 breaks down TLB faults according to the cause of the miss. There are separate handlers for misses triggered by instruction fetches (execute accesses), data accesses (read or write accesses), and VHPT walker accesses. The alternate ITLB and DTLB miss handlers are triggered when the VHPT walker is disabled for the region being accessed (as controlled by a bit in the region register) or when the walker is disabled completely (as controlled by the page table address register bit pta.ve). The DATA NESTED TLB FAULT is triggered when a TLB miss occurs while interruption collection is disabled (ic bit is off), i.e., while another fault is being handled. Ordinarily, a nested fault would indicate a kernel bug. However, as we will see below, Linux/ia64 uses nested TLB misses to support nonspeculative accesses to the virtually-mapped linear page table.

Nested DTLB miss handler

Let us start by taking a look at how the handler for the DATA NESTED TLB FAULT works. It is effectively a helper routine that translates a virtual address *addr* to the physical address *pte_paddr* at which the corresponding PTE can be found. Because this fault is triggered only while ic is off, the CPU does not update the control registers and they instead continue to contain the information for the *original* fault. Unfortunately, the information in the control registers is not sufficient for the nested DLTB miss handler to complete its job. Thus, Linux/ia64 uses three bank 0 registers to pass additional information between the handler of the original fault and the nested DTLB miss handler. The first register is used to pass the virtual address *addr*, and the second is used to pass the address *rlabel* to which execution should return once the nested DTLB handler is done. The third register serves as the result register and is used to return *pte_paddr*.

The nested DTLB miss handler operates by turning off the data translation bit psr.dt and then walking the three-level page table. Because physical data accesses are used during the page-table walk, there is no danger of triggering further TLB misses. The starting point of the page-table walk is determined by the region that *addr* is accessing. If it accesses region 5, the kernel page table is used; otherwise (access to region 0 to 4), the page table of the currently running process is used. If the walk is completed successfully, the physical address of the found PTE is placed in the result register and control is returned to the original handler by a jump to address *rlabel*. If any error is encountered during the page-table walk (e.g., because the virtual address is not mapped), control is transferred to the Linux page fault handler instead.

ITLB/DTLB miss handler

With the operation of the nested DTLB miss handler explained, it is straightforward to describe the ITLB and DTLB miss handlers. In a first step the TLB miss handlers read the ifa control register to determine the faulting address *addr*. In a second step, the bank 0 registers are set up so that the nested DTLB handler can find the necessary information in case execution later triggers a DATA NESTED TLB FAULT. Specifically, the faulting address *addr* and the return address *rlabel* are placed in the bank 0 registers used for this purpose. In the third step, the actual work can begin: The handlers use the thash instruction to translate the faulting address *addr* to *pte_vaddr*, the address in the virtually-mapped linear page table at which the PTE for *addr* can be found. The handlers then attempt to read the PTE by loading the word at this address. If the TLB entry for *pte_vaddr* exists, the load succeeds and the PTE can be installed either in the ITLB (for an ITLB miss) or in the DTLB (for a DTLB miss). Conversely, if the TLB entry for *pte_vaddr* is missing, a DATA NESTED TLB FAULT is triggered and the CPU hands over control to the nested DTLB miss handler. As described previously, this handler walks the page table in physical mode and, if a mapping exists for *addr*, places the physical address *pte_paddr* of the PTE in the result register. The nested miss handler then returns control to the original handler with the data translation bit dt still turned off. The original handler now has the physical address *pte_paddr* of the PTE, and because data accesses are still done in physical mode, the handler can directly load the word at this address, without the risk of causing any further faults. This procedure again yields the desired PTE, which can then be installed in the appropriate TLB (ITLB or DTLB).

The elegance of handling TLB misses in this fashion derives from the fact that both the virtual and physical access cases use exactly the same instruction sequence once the PTE address has been determined. The only difference is whether loading the PTE occurs in physical or in virtual mode. This means that the address *rlabel* can be the address of the same load instruction that attempts to read the PTE from the virtually-mapped linear page table. If the virtual access fails, the load instruction is reexecuted after the nested DTLB miss handler returns, but now in physical mode. Note that, for this technique to work, the address register of the load instruction must be the same as the result register that the nested DTLB handler uses to return the physical address of the PTE.

VHPT miss handler

When a TLB miss occurs in a region for which the VHPT walker has been enabled, the walker first attempts to handle the miss on its own. Conceptually, it does this by using the thash instruction to translate the faulting address *addr* to *pte_vaddr*, the address in the virtually-mapped linear page table at which the PTE for *addr* can be found. If the TLB entry for *pte_vaddr* is present in the TLB, the VHPT walker (usually) can handle the miss on its own. On the other hand, if this TLB entry is also missing, the CPU raises a VHPT TRANSLATION FAULT and passes *addr* in control register ifa and *pte_vaddr* in iha.

Once the VHPT miss handler starts executing, it extracts the original faulting address *addr* from ifa and then traverses the page table in physical mode, just like the nested DTLB miss handler. Once the physical address of the PTE has been found, the PTE is loaded and installed in the appropriate TLB (DTLB if the original access was a data access, ITLB otherwise). In addition, the VHPT constructs and inserts into the DTLB a translation that maps the address contained in iha. This ensures that future TLB misses to nearby addresses can be handled through the virtually-mapped linear page table.

Because this handler inserts two translations in the TLB, care must be taken to ensure that the CPU can make forward progress, even if only one of the two translations survives. Fortunately, in this particular case, the order of insertion does not matter because the CPU can make forward progress either way: If the translation for the original faulting address survives, the access that caused the fault can obviously complete without further TLB misses. Conversely, if translation for the virtually-mapped linear page table survives, the VHPT walker is either able to handle the reoccurrence of the TLB miss for the original faulting address on its own or the CPU raises a regular ITLB or DTLB FAULT, which would also resolve the fault.

Note that with a perfect VHPT walker, the CPU would only raise VHPT TRANSLATION FAULTs—regular ITLB or DTLB FAULTs would never be raised. However, the IA-64 architecture leaves CPU designers the option of implementing an imperfect VHPT walker or of omitting it completely. This flexibility is achieved by the requirement that if a VHPT walker cannot handle a particular TLB miss, the CPU must raise an ITLB or DTLB FAULT instead. In the most extreme case of a nonexistent VHPT walker, this means that instead of VHPT TRANSLATION FAULTs, only ITLB or DTLB FAULTs would occur. In a more realistic scenario, the VHPT walker would be able to handle most TLB misses except that in certain corner cases, it would have to resort to raising an ITLB or DTLB FAULT. For example, the Itanium VHPT walker can handle virtual page-table accesses as long as the PTE can be found in the second-level cache. If the PTE is not cached, the walker gives up and raises an ITLB or DTLB FAULT instead.

Alternate ITLB/DTLB miss handler

The CPU dispatches to the alternate ITLB and DTLB handlers when a TLB miss occurs and the VHPT walker is disabled. Because regions 6 and 7 of the virtual address space of Linux/ia64 are identity-mapped, they have no associated page table and the VHPT walker has to be disabled for those regions. In other words, TLB misses caused by accesses to regions 6 and 7 always result in the one of the alternate TLB miss handlers being invoked.

For the other regions, the VHPT walker is normally enabled. However, for performance measurements and debugging purposes, it is sometimes useful to turn off the VHPT in these regions as well. The bottom line is that accesses to region 6 and 7 are always handled by the alternate miss handlers, and accesses to region 0 to 5 are only sometimes handled here. Thus, before doing anything else, the alternate TLB miss handlers first check whether the faulting access is to region 6 or 7. If not, the miss is redirected to the normal ITLB/DTLB miss handler described earlier. If it is, the handler can calculate the necessary PTE directly from the address being accessed. That is, there is no need to walk a page table. The PTE calculated in this fashion permits read, write, and execute accesses by the kernel only and maps to the physical address that is equal to the least significant 61 bits of the virtual address. The **dirty** and **accessed** bits are set to 1, and the memory attribute is derived from the region being accessed: For region 6 the **uncacheable** attribute is used, and for region 7 the **cacheable** attribute is used.

Calculating the PTE is straightforward, but a few corner cases need to be taken care of. First, user-level accesses to regions 6 and 7 have to be intercepted and redirected to cause a segmentation fault. This may seem strange because the permission bits in the translations that would be inserted in response to such accesses would prevent user-level accesses at any rate. However, the reason the redirection is necessary is that the IA-64 architecture does not permit the same physical address to be mapped with conflicting memory attributes. Suppose an application accessed a particular physical address first through region 6 and then through region 7. Both accesses would trigger a segmentation violation signal, but the application can intercept those and skip over the faulting instruction. Now, if user-level accesses were not intercepted, the two accesses would result in the same physical address being mapped both cached and uncached, which violates the architecture and could cause a failure on some CPUs. Because this is an unacceptable risk, the alternate TLB miss handlers must prevent such translations from being inserted in the first place. This is most easily achieved by rejecting all user-level accesses to regions 6 and 7.

A related problem arises from speculative loads in the kernel. If TLB misses are not deferred (dcr.dm is 0), a speculative load inside the kernel may cause a TLB miss to an arbitrary address. If that address happens to fall inside region 6 or 7, the speculative load would trigger an alternate TLB fault. This again poses the risk of inserting a translation with conflicting memory attributes. To prevent this, the alternate DTLB miss handler also checks whether the faulting access was caused by a speculative load and, if so, turns on the exception deferral bit (ed in psr) instead of installing a translation. The net effect of this method is that all speculative loads to region 6 and 7 produce a NaT value, unless the translation for the page being accessed happens to be in the TLB already. This solution may sometimes produce a NaT unnecessarily, but apart from a small performance impact, does not affect the correct operation of the kernel. This solution also has the advantage that speculative loads cannot pollute the TLB with unnecessary translations.

PTE fault handlers

Let us now turn our attention to the PTE-related faults. As Table 4.5 shows, there are seven such faults. The INSTRUCTION ACCESS-BIT FAULT and DATA ACCESS-BIT FAULTs are

Table 4.5. IA-64 PTE faults.

Fault	***Description***
INSTR. ACCESS-BIT FAULT	Instruction fetch to a page with the **accessed** bit cleared.
DATA ACCESS-BIT FAULT	Data (read or write) access to a page with the **accessed** bit cleared.
DIRTY-BIT FAULT	Write access to a page with the **dirty** bit cleared.
PAGE NOT PRESENT FAULT	Access to a page with the **present** bit cleared.
INSTR. ACCESS RIGHT FAULT	Instruction fetch to a page with no execute access right.
DATA ACCESS RIGHTS FAULT	Data (read or write) access to page violates access rights.
KEY PERMISSION FAULT	Access to a page violates protection key permissions.

raised if an instruction fetch or a data access is performed to a page with the **accessed** (A) bit turned off in the PTE. Linux uses this bit to drive its page replacement algorithm, and the handlers for these faults turn on this bit in the PTE, update the TLB entry, and then return. Just like the DTLB/ITLB miss handlers, they use the virtually-mapped linear page table to access the PTE and fall back on the nested DTLB miss handler if necessary.

The DIRTY-BIT FAULT is raised on a write access to a page with the **dirty** (D) bit turned off in the PTE. This fault is handled exactly like the DATA ACCESS-BIT FAULT except that it turns on both the **dirty** and the **accessed** bits. Turning on just the **dirty** bit would also work correctly but would be suboptimal because as soon as the handler returned, the lower-priority DATA ACCESS-BIT FAULT would be raised. In other words, by turning on both bits in this handler, the work of two faults can be accomplished with a single fault, which results in better performance.

The PAGE NOT PRESENT FAULT, INSTRUCTION ACCESS RIGHT FAULT, and DATA ACCESS RIGHTS FAULT are raised when the **present** bit is cleared in a PTE or when a page is accessed in a way that violates the permission bits in the PTE. None of these faults can be handled in the IVT itself. Consequently, the handlers for these faults unconditionally transfer control to the Linux page fault handler, where the appropriate action is taken. This action often has the effect of changing the PTE. If so, Linux normally takes care of flushing the old TLB entry. However, Linux assumes that PTEs with the **present** bit cleared are never installed in the TLB, so it does not flush the TLB after turning on the **present** bit on a page that was paged in. Because IA-64 violates this assumption, the PAGE NOT PRESENT FAULT handler must flush the TLB entry before calling the Linux page fault handler.

The final PTE-related fault is the KEY PERMISSION FAULT. This fault can only be raised if protection key checking is enabled (psr.pk is 1). Because Linux/ia64 does not use protection key registers, this checking is disabled and hence the fault cannot occur.

Multiprocessor considerations

The combination of a global TLB purge instruction and software TLB miss handling creates a subtle race condition. Suppose there are two CPUs, P_0 and P_1, with P_0 accessing virtual address a, and P_1 updating the page-table entry for this address. We could then get the sequence of events illustrated in Figure 4.36. P_0 accesses virtual memory address a, and

P_0	P_1
1. Access virtual address a	
2. TLB miss on address a	
3. Read PTE for address a	
	4. Update PTE for address a
	5. Flush TLB for address a (ptc.ga)
6. Insert PTE into TLB (itc)	

Figure 4.36. Example race condition between TLB miss handling on CPU P_0 and TLB flush on CPU P_1.

this might trigger a TLB miss. The TLB miss handler would then start executing and read the corresponding PTE from the page table. Right after that, P_1 might update this PTE and, to ensure that all CPUs are aware of this modification, it would use ptc.ga to flush the TLB entries for the page at address a. On P_0, this instruction would have no effect, because the TLB entry for address a is not yet present. However, right after the TLB flush has completed, P_0 might finish the TLB miss handling and execute an itc instruction to insert the PTE it read in step 3. This means that after step 6, the TLB of P_0 would contain a stale translation: address a would be mapped according to the old PTE read in step 3, not according to the updated PTE written in step 4!

To avoid this problem, the TLB miss handler can reread the PTE from the page table after step 6 and check to see if it changed. If it did change, there are two options: the handler can either restart miss handling from the beginning or it can return after flushing the entry inserted in step 6. In the latter case, the memory access will be reexecuted, and because the translation for address a is still missing, the TLB miss handler will be invoked again. Eventually, the miss handler will be able to execute without encountering a race condition, and at that point the correct PTE for address a will be inserted into the TLB.

Note that even though we used a TLB miss to illustrate the race condition, it can arise with any fault handler that inserts a TLB translation based on the content of a page table. Also note that the race condition arises from the fact that the fault handling is not atomic with respect to the ptc.ga instruction. If global TLB flushes were implemented with interprocessor interrupts instead, the fault handling would be atomic and the PTE rechecking would not be necessary.

Dispatching to the Linux page fault handler

The previous discussion of the fault handlers makes it amply clear that there are many cases in which the assistance of the Linux page fault handler is required. However, before the platform-independent *handle_mm_fault()* can be invoked, the IA-64–specific code must locate the appropriate vm-area structure and verify that the access does not violate the access rights in the vm-area structure. Because these actions must be performed whenever control is transferred to the Linux page fault handler, they are implemented in a common routine called *ia64_do_page_fault()*. Most actions performed by this routine are straightforward and follow the steps outlined at the beginning of this section. However, two interesting IA-64–specific aspects warrant further discussion.

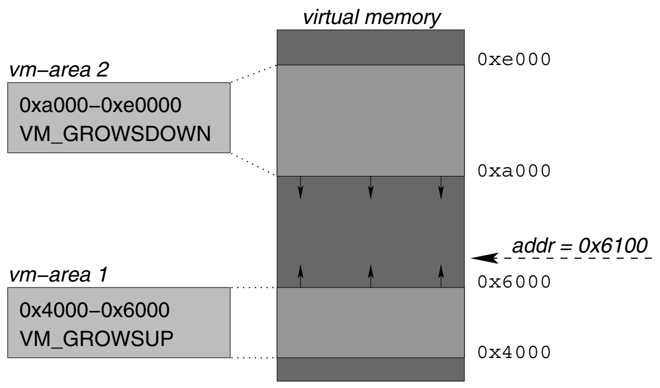


Figure 4.37. Ambiguity caused by vm-areas with different growth directions.

First, note that the platform-specific code is responsible for supporting automatic expansion of vm-areas with the flag VM_GROWSUP or VM_GROWSDOWN set. For most platforms, stacks grow either toward higher or lower addresses, and consequently these platforms support just the flag that corresponds to the stack growth direction. IA-64 is special in that it needs to support both growth directions as the register stack grows toward higher addresses and the memory stack grows toward lower addresses. As Figure 4.37 illustrates, this requirement introduces a potential ambiguity. Suppose *vm-area 1* covers the address range `0x4000` to `0x6000` and grows toward higher addresses and *vm-area 2* covers the range `0xa000` to `0xe000` and grows toward lower addresses. Now, if a process accesses address `0x6100`, should *vm-area 1* or *vm-area 2* be expanded? Without knowing the intent of the process there is no way to resolve this ambiguity in a way that is guaranteed to be correct. However, because the register stack engine (RSE) accesses the register backing store in a strictly sequential fashion, Linux/ia64 adopts the policy that a vm-area with the VM_GROWSUP flag set is expanded only if the page fault was caused by an access to the word immediately above the end of the vm-area. This implies that the case illustrated in the figure would be resolved in favor of expanding *vm-area 2*. Only if address `0x6000` were to be accessed would *vm-area 1* be expanded. This policy is guaranteed to make the right choice provided VM_GROWSUP is used only to map register backing store memory.

The second question is how faults caused by speculative loads should be treated (see Chapter 2, *IA-64 Architecture*, for more information on speculation). The IA-64 architecture defines two speculation models: **recovery** and **no-recovery** [76]. The **recovery** model requires that speculative loads are always accompanied by corresponding recovery code. As the name suggest, the **no-recovery** model does not require recovery code. For this model to work, speculative faults must not produce a NaT unless it is *guaranteed* that a nonspeculative load to the same address would also fail.

Linux/ia64 does not support the **no-recovery** model at the user level because it could cause unexpected application failures. To see why, consider an application that implements a distributed shared memory system (DSM) by using segmentation violation signals to detect which pages need to be fetched from a remote machine. The TreadMarks DSM is an example of such a system [3]. If this application performed a speculative load to unmapped memory and the offending code uses the **no-recovery** model, the Linux page fault handler would be faced with the difficult choice of returning a NaT or sending a segmentation violation signal. If it were to return a NaT, it would have made an error if the speculative load was to a DSM page that the signal handler would have fetched from the remote machine. On the other hand, if it were to deliver the signal, then it would have made an error if the speculative load was accessing an address that was indeed illegal. This problem could be solved if the application knew about speculative loads and could decide on its own whether a NaT should be produced. However, because this would result in an application that is not portable to other architectures, Linux/ia64 does not support the **no-recovery** model.

The **recovery** model does not suffer from this problem because the presence of the recovery code ensures that it is always safe to return a NaT. This behavior also answers how page faults caused by speculative loads should be handled: Because Linux/ia64 does not support the **no-recovery** model, it handles such accesses by setting the ed bit in the processor status register (psr) of the interrupted process. This action instructs the CPU to place a NaT value in the target register when the speculative load is restarted after returning from the page fault handler. In our DSM example, this might cause the application's recovery code to be invoked unnecessarily at times, but other than a slight performance degradation, there are no ill effects.

4.6 MEMORY COHERENCY

A given memory location is said to be *coherent* if all CPUs and I/O devices in a machine observe one and the same value in that location. Modern machines make aggressive use of memory caches and thereby introduce many potential sources for incoherence. For example, to maximize performance, caches usually delay stores by the CPU as long as possible and write them back to main memory (or the next level in the cache hierarchy) only when absolutely necessary. As a result, the same memory location may have different values in the various caches or in main memory. Modern machines also often use separate caches for instructions and data, thereby introducing the risk of the same location being cached both in the instruction and data cache, but with potentially different values.

A particularly insidious source of incoherence arises from *virtually-tagged* caches that tag the cache contents not with the memory location (physical address) but with the virtual address with which the location was accessed. Consider the scenario where the same location is accessed multiple times with different virtual addresses. This is called *virtual aliasing* because different virtual addresses refer to the same memory location. Linux frequently does this and with virtually-tagged caches, the memory location may end up being cached multiple times! Moreover, updating the memory location through one virtual address would not update the cache copies created by the aliases, and we would again have a situation where a memory location is incoherent.

flush_cache_all();	*/* make all kernel data coherent */*
flush_icache_range(*start*, *end*);	*/* make range in i- and d-caches coherent */*
flush_cache_mm(*mm*);	*/* make data mapped by mm coherent */*
flush_cache_range(*mm*, *start*, *end*);	*/* make data in address range coherent */*
flush_cache_page(*vma*, *addr*);	*/* make data page in vma coherent */*
flush_dcache_page(*pg*);	*/* make page cache page coherent */*
clear_user_page(*to*, *uaddr*, *pg*);	*/* clear page and maintain coherency */*
copy_user_page(*from*, *to*, *uaddr*, *pg*);	*/* copy page and maintain coherency */*

Figure 4.38. Kernel interface to maintain memory coherency.

I/O devices are yet another source of incoherence: a device may write a memory location through DMA (see Chapter 7, *Device I/O*) and this new value may or may not be observed by the memory caches.

We would like to emphasize that, by itself, an incoherent memory location is not an issue. A problem arises only if an incoherent value is *observed*, e.g., by a CPU or an I/O device. When this happens, the result is usually catastrophic. For example, suppose we are dealing with a platform that uses separate instruction and data caches. If the operating system reads a page of code from the text section of an executable file and copies it to the user space of a process, the data cache will be updated but the instruction cache will remain unchanged. Thus, when the process attempts to execute the newly loaded code, it may end up fetching stale instructions and the process may crash. To avoid such problems, memory locations must be made coherent before a stale value can be observed. Depending on the platform architecture, maintaining coherency may be the responsibility of hardware or software. In practice, the two often share the responsibility, with the hardware taking care of certain sources of incoherence and the software taking care of the rest.

4.6.1 Maintenance of coherency in the Linux kernel

To accommodate the wide variety of possible memory coherence schemes, Linux defines the interface shown in Figure 4.38. Every platform must provide a suitable implementation of this interface. The interface is designed to handle all coherence issues except DMA coherence. DMA is handled separately, as we see in Chapter 7, *Device I/O*.

The first routine in this interface is *flush_cache_all()*. It must ensure that for data accessed through the kernel address space, all memory locations are coherent. Linux calls this routine just before changing or removing a mapping in the page-table-mapped kernel segment or the kmap segment. On platforms with virtually-tagged caches, the routine is usually implemented by flushing all data caches. On other platforms, this routine normally performs no operation.

The second routine, *flush_icache_range()*, ensures that a specific range of memory locations is coherent with respect to instruction fetches and data accesses. The routine takes two arguments, *start* and *end*. The address range that must be made coherent extends from *start* up to and including $end - 1$. All addresses in this range must be valid kernel or mapped user-space virtual addresses. Linux calls this routine after writing instructions to memory. For example, when loading a kernel module, Linux first allocates memory in the page-table-mapped kernel segment, copies the executable image to this memory, and then calls *flush_icache_range()* on the text segment to ensure the d-caches and i-caches are coherent before attempting to execute any code in the kernel module. On platforms that do not use separate instruction or data caches or that maintain coherency in hardware, this routine normally performs no operation. On other platforms, it is usually implemented by flushing the instruction cache for the given address range.

The next three routines are all used by Linux to inform platform-specific code that the virtual-to-physical translation of a section of a (user) address space is about to be changed. On platforms with physically indexed caches, these routines normally perform no operation. However, in the presence of virtually indexed caches, these routines must ensure that coherency is maintained. In practice, this usually means that all cache lines associated with any of the affected virtual addresses must be flushed from the caches. The three routines differ in the size of the section they affect: *flush_cache_mm()* affects the entire address space identified by mm structure pointer *mm*; *flush_cache_range()* affects a range of addresses. The range extends from *start* up to and including $end - 1$, where *start* and *end* are both user-level addresses and are given as the second and third arguments to this routine. The third routine, *flush_cache_page()*, affects a single page. The vm-area that covers this page is identified by the *vma* argument and the user-space address of the affected page is given by argument *addr*. These routines are closely related to the TLB coherency routines *flush_tlb_mm()*, *flush_tlb_range()*, and *flush_tlb_page()* in the sense that they are used in pairs. For example, Linux changes the page table of an entire address space *mm* by using code that follows the pattern shown below:

```
flush_cache_mm(mm);
...change page table of mm...
flush_tlb_mm(mm);
```

That is, before changing a virtual-to-physical translation, Linux calls one of the *flush_cache* routines to ensure that the affected memory locations are coherent. In the second step, it changes the page table (virtual-to-physical translations), and in the third step, it calls one of the *flush_tlb* routines to ensure that the TLB is coherent. The order in which these steps occur is critical: The memory locations need to be made coherent *before* the translations are changed because otherwise the flush routine might fault. Conversely, the TLB must be made coherent *after* the translations are changed because otherwise another CPU might pick up a stale translation after the TLB has been flushed but before the page table has been fully updated.

The three routines just discussed establish coherence for memory locations that may have been written by a user process. The next three routines complement this by providing the means to establish coherence for memory locations written by the kernel.

The first routine is *flush_dcache_page()*. The Linux kernel calls it to notify platform-specific code that it just dirtied (wrote) a page that is present in the page cache. The page is identified by the routine's only argument, *pg*, which is a pointer to the page frame descriptor of the dirtied page. The routine must ensure that the content of the page is coherent as far as any user-space accesses are concerned. Because the routine affects coherency only in regards to user-space accesses, the kernel does not call it for page cache pages that cannot possibly be mapped into user space. For example, the content of a symbolic link is never mapped into user space. Thus, there is no need to call *flush_dcache_page()* after writing the content of a symbolic link. Also, note that this routine is used only for pages that are present in the page cache. This includes all non-anonymous pages and old anonymous pages that have been moved to the swap cache.

Newly created anonymous pages are not entered into the page cache and must be handled separately. This is the purpose of *clear_user_page()* and *copy_user_page()*. The former creates an anonymous page, which is cleared to 0. The latter copies a page that is mapped into user space (e.g., as a result of a copy-on-write operation). Both routines take an argument called *pg* as their last argument. This is a pointer to the page frame descriptor of the page that is being written. Apart from this, *clear_user_page()* takes two other arguments: *to* and *uaddr*. The former is the kernel-space address at which the page resides, and *uaddr* is the user-space address at which the page will be mapped (because anonymous pages are process-private, there can be only one such address). Similarly, *copy_user_page()* takes three other arguments: *from*, *to*, and *uaddr*. The first two are the kernel-space addresses of the source and the destination page, and *uaddr* is again the user-space address at which the new page will be mapped. Why do these two routines have such a complicated interface? The reason is that on platforms with virtually indexed caches, it is possible to write the new page and make it coherent with the page at *uaddr* without requiring any explicit cache flushing. The basic idea is for the platform-specific code to write the destination page not through the page at address *to*, but through a kernel-space address that maps to the same cache lines as *uaddr*. Because anonymous pages are created frequently, this clever trick can achieve a significant performance boost on platforms with virtually indexed caches. On the other hand, on platforms with physically indexed caches, these operations normally perform no operation other than clearing or copying the page, so despite having rather complicated interfaces, the two routines can be implemented optimally on all platforms.

4.6.2 IA-64 implementation

The IA-64 architecture guarantees that virtual aliases are supported in hardware but leaves open the possibility that on certain CPUs there may be a performance penalty if two virtual addresses map to the same memory location and the addresses do not differ by a value that is an integer multiple of 1 Mbyte. This implies that Linux can treat IA-64 as if all caches were physically indexed, and hence *flush_cache_all()*, *flush_cache_mm()*, *flush_cache_range()*, and *flush_cache_page()* do not have to perform any operation at all. To avoid the potential performance penalty, Linux/ia64 maps shared memory segments at 1-Mbyte–aligned addresses whenever possible.

While IA-64 generally requires that coherence is maintained in hardware, there is one important exception: When a CPU writes to a memory location, it does not have to maintain coherence with the instruction caches. For this reason, the IA-64 version of *flush_icache_range()* must establish coherence by using the flush cache (fc) instruction. This instruction takes one address operand, which identifies the cache line that is to be written back (if it is dirty) and then is evicted from all levels of the cache hierarchy. This instruction is broadcast to all CPUs in an MP machine, so it is sufficient to execute it on one CPU to establish coherence across the entire machine. The architecture guarantees that cache lines are at least 32 bytes in size, so the routine can be implemented by executing fc once for every 32-byte block in the address range from *start* to $end - 1$. Care must be taken not to execute fc on an address that is outside this range; doing so could trigger a fault and crash the kernel.

Is implementing *flush_icache_range()* sufficient to guarantee coherence between the data and instruction caches? Unfortunately, the answer is no. To see this, consider that Linux calls this routine only when it positively knows that the data it wrote to memory will be executed eventually. It does not know this when writing a page that later on gets mapped into user space. For this reason, *flush_dcache_page()*, *clear_user_page()*, and *copy-_user_page()* logically also must call *flush_icache_range()* on the target page. Given that there are usually many more data pages than code pages, this naive implementation would be prohibitively slow. Instead, Linux/ia64 attempts to delay flushing the cache with the following trick: The Linux kernel reserves a 1-bit field called PG_arch_1 in every page frame descriptor for platform-specific purposes. On IA-64, this bit indicates whether or not the page is coherent in the instruction and data caches. A value of 0 signifies that the page *may not* be coherent and a value of 1 signifies that the page is *definitely coherent*. With this setup, *flush_dcache_page()*, *clear_user_page()*, and *copy_user_page()* can be implemented so that they simply clear the PG_arch_1 bit of the dirtied page. Of course, before mapping an executable page into user space, the kernel still needs to flush the cache if the PG_arch_1 bit is off. Fortunately, we can use the platform-specific *update_mmu_cache()* for this purpose. Recall from Section 4.4.2 that this routine is called whenever a translation is inserted or updated in the page table. On IA-64, we can use this as an opportunity to check whether the page being mapped has the execute permission (X) bit enabled. If not, there is no need to establish coherency with the instruction cache and the PG_arch_1 bit is ignored. On the other hand, if the X bit is enabled and the PG_arch_1 bit is 0, coherency must be established by a call to *flush_icache_range()*. After this call returns, the PG_arch_1 bit can be turned on, because it is now known that the page is coherent. This approach ensures that if the same page is mapped into other processes, the cache flush does not have to be repeated again (assuming it has not been dirtied in the meantime).

There is just one small problem: Linux traditionally maps the memory stack and memory allocated by the *brk()* system call with execute permission turned on. This has a devastating effect on the delayed i-cache flush scheme because it causes all anonymous pages to be flushed from the cache even though usually none of them are ever executed. To fix this problem, Linux/ia64 is lazy about turning on the X bit in PTEs. This works as follows: When installing the PTE for a vm-area that is both writable and executable, Linux/ia64 does *not* turn on the X bit. If a process attempts to execute such a page, a protection violation fault is raised by the CPU and the Linux page fault handler is invoked. Because the vm-area permits execute accesses, the page fault handler simply turns on the X bit in the PTE and then updates the page table. The process can then resume execution as if the X bit had been turned on all along. In other words, this scheme ensures that for vm-areas that are

```
activate_mm(prev_mm, next_mm);                   /* activate new address space */
switch_mm(prev_mm, next_mm, next_task, cpu);     /* switch address space */
```

Figure 4.39. Kernel interface to switch address spaces.

mapped both executable and writable, the X bit in the PTEs will be enabled only if the page experiences an execute access. This technique has the desired effect of avoiding practically all cache flushing on anonymous pages.

The beauty of combining the delayed i-cache flush scheme with the lazy execute bit is that together they are able to not just delay but completely avoid flushing the cache for pages that are never executed. As we see in Chapter 7, *Device I/O*, the effectiveness of this pair is enhanced even further by the Linux DMA interface because it can be used to avoid the need to flush even the executable pages. The net effect is that on IA-64 it is virtually never necessary to explicitly flush the cache, even though the hardware does not maintain i-cache coherence for writes by the CPU.

4.7 SWITCHING ADDRESS SPACES

In Chapter 3, *Processes, Tasks, and Threads*, we discussed how Linux switches the execution context from one thread to another. We did not discuss how the address space is switched. The reason is that Linux treats context switching and address-space switching as separate operations. This makes sense because a context switch triggers an address-space switch only if the old and the new thread do not share the same address space. Conversely, there are occasions, such as an *execve()* system call, where the address space is switched but the currently executing thread remains the same. In other words, not every context switch causes an address-space switch, and vice versa—a good indication that these are fundamentally separate operations.

4.7.1 Address-space switch interface

Linux uses the *address-space switch interface* (file include/asm/mmu_context.h) in Figure 4.39 to abstract platform differences in how an address-space switch can be effected. The *activate_mm()* routine switches the address space of the currently executing thread. It takes two arguments, *prev_mm* and *next_mm*, which are pointers to the mm structure of the old and the new address space, respectively. Similarly, *switch_mm()* is called when the address space is switched as part of a context switch. The first two arguments have the same meaning as for *activate_mm()*, but this routine receives two additional arguments: *next_task* and *cpu*. The *next_task* argument is a pointer to the task structure of the thread that will be executed next. As usual, the currently running thread is implicitly identified by global variable *current*. The *cpu* argument is the unique ID of the CPU that is performing the address-space switch (see Chapter 8, *Symmetric Multiprocessing*).

In an ideal world, *switch_mm()* would not be necessary and *activate_mm()* could be used instead. However, Linux provides separate routines to accommodate platforms that inextricably (and incorrectly) tie together context switching and address-space switching. On those platforms, an address-space switch can be effected only through a context switch, and *activate_mm()* must implicitly perform a dummy context switch. This means that if *activate_mm()* were used in place of *switch_mm()*, the dummy context switch would almost immediately be followed by a real context switch, which would be inefficient. In other words, *switch_mm()* can be thought of as a version of *activate_mm()* that is optimized for the case where it is known that the address-space switch will quickly be followed by a context switch. Of course, on all other platforms, *switch_mm()* can be implemented directly in terms of *activate_mm()*.

4.7.2 IA-64 implementation

On Linux/ia64, switching the address space involves three simple steps:

1. Set current page-table pointer to the global directory of the new address space.

2. Ensure that the new address space has a valid ASN (region ID) by calling *get_mmu-_context()*.

3. Load the region ID of new address space into region registers by calling *reload_con-text()*.

The first step is necessary only because Linux/ia64 maintains the physical address of the global directory of the current address space in kernel register k6. This simplifies the TLB fault handler slightly and also lets it run more efficiently.

The IA-64 architecture leaves context and address-space switching entirely to software, so there is no problem in keeping the two operations separate. Thus, *switch_mm()* can be implemented simply as a call to *activate_mm()*, passing the first two arguments, *prev_mm* and *next_mm*, and dropping the remaining three arguments.

4.8 DISCUSSION AND SUMMARY

Let us close this chapter by discussing the rationale behind some of the virtual memory choices of Linux/ia64.

First, there is the question of page size. The IA-64 architecture supports a large number of different page sizes including at least 4 Kbytes, 8 Kbytes, 16 Kbytes, 256 Kbytes, 1 Mbyte, 4 Mbytes, 16 Mbytes, 64 Mbytes, and 256 Mbytes. Linux uses a three-level page table, so the page size it uses directly affects the amount of virtual memory that a page table can map. For example, with a page size of 4 Kbytes, only 39 virtual address bits can be mapped. Thus, even though IA-64 can support pages as small as 4 Kbytes, from the perspective of Linux it is much better to pick a larger page size. Indeed, each time the page size is doubled, the amount of virtual memory that can be mapped increases 16 times! With a page size of 64 Kbytes, for example, 55 virtual address bits can be mapped.

Another consideration in choosing a page size is that programs should perform no worse on a 64-bit platform than on a 32-bit platform. Because pointers are twice as big on IA-64 as on a 32-bit platform, the data structures of a program may also be up to twice as big (on average, the data size expansion factor is much smaller: on the order of 20–30 percent [47]). Given that IA-32 uses a page size of 4 Kbytes, this would suggest a page size of 8 Kbytes for IA-64. This size would guarantee that data structures that fit on a single page on IA-32 would also fit on a single page on IA-64. But we should also consider code size. Comparing the size of the text section of equivalent IA-32 and IA-64 programs, we find that IA-64 code is typically between two to four times larger. This would suggest a page size of 16 Kbytes.

The optimal page size depends heavily on both the machine and the applications in use. Taking all these factors into account, it is clear that any single page size cannot be optimal for all cases. Consequently, Linux/ia64 adopts the pragmatic solution of providing a kernel compile-time choice for the page size. In most cases, a choice of 8 Kbytes or 16 Kbytes would be reasonable, but under certain circumstances a page size as small as 4 Kbytes or as large as 64 Kbytes could be preferable. Of course, this solution implies that applications must not rely on Linux/ia64 implementing a particular page size. Fortunately, this is not a problem because the few applications that really do need to know the page size can obtain it by calling the *getpagesize()* library routine. Another approach to dealing with the page size issue is to put intelligence into the operating system to detect when a series of pages can be mapped by a single page of larger size, i.e., by a superpage. Superpage support can often mitigate some of the performance problems that occur when the page size is too small for a particular application. However, it does introduce additional complexity that could slow down all programs. More importantly, because superpages do not change the mapping granularity, they do not increase the amount of virtual memory that a page table can map.

Second, the choice with perhaps the most dramatic consequences is the structure of the address space that Linux/ia64 supports. The IA-64 architecture leaves the operating system designer almost complete freedom in this respect. For example, instead of the structure described in this chapter, Linux/ia64 could have implemented a linear address space whose size is determined by the amount of memory that can be mapped by a page table. This approach would have the disadvantage of placing everything in region zero. This would imply that the maximum distance by which different program segments can be separated is limited by the amount of virtual memory that can be mapped by the smallest supported page size (4 Kbytes). This is a surprisingly serious limitation, considering that a page size of 4 Kbytes limits virtual memory to just 2^{39} bytes. In contrast, designers can exploit the region support in IA-64 and separate segments by 2^{61} bytes, no matter what the page size.

A third choice that is closely related to the address-space structure is the format of the page-table entries. The short format is the most natural choice for Linux/ia64 because it makes possible the use of the VHPT walker by mapping the Linux page table into virtual space. On the downside, the short-format PTEs cannot take advantage of all the capabilities provided by IA-64. For example, with the long format, it would be possible to specify a separate protection key for each PTE, which in turn could enable more effective use of the TLB. Although Linux/ia64 directly maps its page table into the virtually-mapped linear page table, the IA-64 architecture does not require this to be the case. An alternative would be to operate the VHPT walker in the long-format mode and map it to a separate cache of recently used translations. This cache can be thought of as a CPU-external (in-memory)

TLB. With this, Linux could continue to use a compact 8-byte long page-table entry and still take full advantage of operating the VHPT walker in long-format mode. Of course, the cost of doing this would be that the cache would take up additional memory and extra work would be required to keep the cache coherent with the page tables.

In summary, the IA-64 architecture provides tremendous flexibility in implementing the virtual memory system. The virtual memory design described in this chapter closely matches the needs of Linux, but at the same time it represents just one possible solution in a large design space. Because the machines that Linux/ia64 is running on and the applications that use it change over time, it is likely that the virtual memory system of Linux/ia64 will also change from time to time. While this will affect some of the implementation details, the fundamental principles described in this chapter should remain the same.