dressed; therefore, address translation is not required when there is a cache hit. Virtually addressed caches require kernel involvement to address some consistency problems. These issues are described in Section 15.13.

13.3.4 The MIPS R3000

The MIPS R3000 is a RISC system and has been a platform for SVR4 UNIX as well as Digital Equipment Corporation's ULTRIX (a 4.2BSD-based system). It has an unusual MMU architecture [Kane 88] in that there is no hardware support for page tables. The only address translations performed by the hardware are those defined by the on-chip TLB.

This has far-reaching implications on the division of memory management tasks and the interface between the hardware and the kernel. In the Intel *x*86 architecture, for instance, the structure of the TLB entry is opaque to the kernel. The only operations allowed are invalidation of single entries keyed by virtual address or of the entire TLB. In contrast, the MIPS architecture makes the format and contents of the TLB entry public to the kernel and allows operations to read, modify, and load specific entries.

The virtual address space itself is divided into four segments, as shown in Figure 13-11. The *kuseg,* spanning the first two gigabytes, contains the user address space. The other three segments are accessible only in kernel mode. *kseg0* and *kseg1* each map directly to the first 512 megabytes of physical memory, thus requiring no TLB mapping. Of these, *kseg0* uses the data/instruction caches, but *kseg1* does not. The top gigabyte is devoted to *kseg2,* which is the mapped, cacheable kernel segment. Addresses in *kseg2* can be mapped to any physical memory location.

Figure 13-12 describes the MMU registers and the format of the TLB entry. The MIPS page size is fixed at 4 kilobytes; thus the virtual address is divided into a 20-bit virtual page number and a 12-bit offset. The TLB contains 64 entries, and each entry is 64 bits in size. The **entryhi** and **entrylo** registers have the same format as the high and low 32 bits of the TLB entry, respectively, and are used to read and write a TLB entry. The VPN (virtual page number) and PFN (physical frame number) fields allow translation of virtual to physical page numbers. The PID field acts as a tag, associating each TLB entry with a process. This PID, which is 6 bits in size, can take the values 0

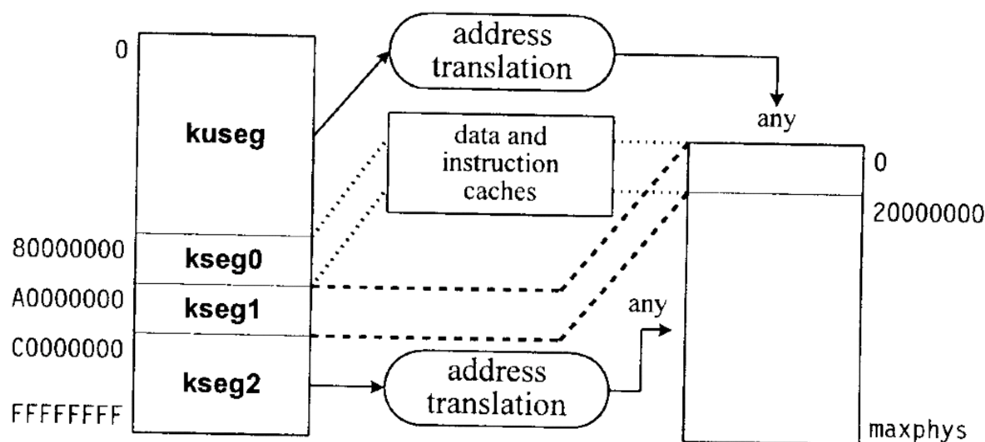

Figure 13-11. MIPS R3000 virtual address space.

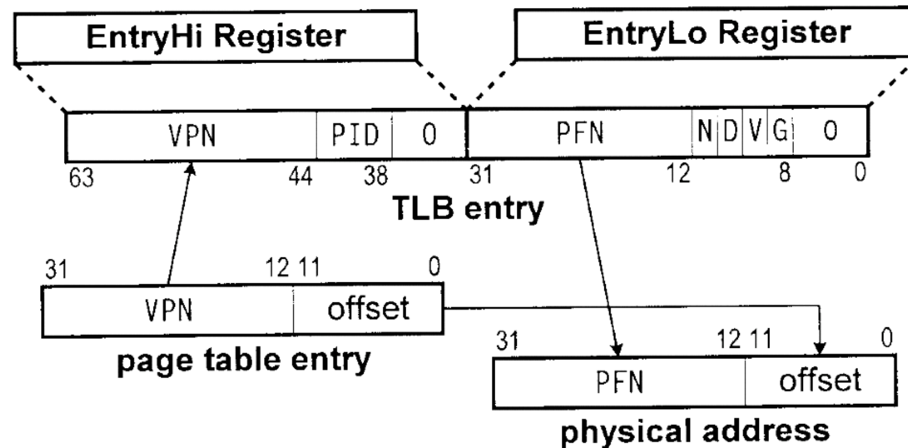

Figure 13-12. MIPS R3000 address translation.

through 63, and is not the same as the traditional process ID. Each process that may have active TLB entries will be assigned a *tlbpid* between 0 and 63. The kernel sets the PID field in the **entryhi** register to the *tlbpid* of the current process. The hardware compares it to the corresponding field in the TLB entries, and rejects translations that do not match. This allows the TLB to contain entries for the same virtual page number belonging to different processes without conflict.

The N *(no-cache)* bit, if set, says that the page should not go through the data or instruction caches. The G *(global)* bit specifies that the PID should be ignored for this page. If the V *(valid)* bit is clear, the entry is invalid, and if the D *(dirty)* bit is clear, the entry is write-protected. Note that there is neither a *referenced* bit nor a *modified* bit.

In translating *kuseg* or *kseg2* addresses, the virtual page number is compared with all TLB entries simultaneously. If a match is found and the G bit is clear, the PID of the entry is compared with the current *tlbpid*, stored in the **entryhi** register. If they are equal (or if the G bit is set) and the V bit is set, the PFN field yields the valid physical page number. If not, a **TLBmiss** exception is raised. For write (store) operations, the D bit must be set, or else a **TLBmod** exception will be raised.

Since the hardware provides no further facilities (such as page table support), these exceptions must be handled by the kernel. The kernel will look at its own mappings, and either locate a valid translation or send a signal to the process. In the former case, it must load a valid TLB entry and restart the faulting instruction. The hardware imposes no requirements on whether the kernel mappings should be page table-based and what the page table entries should look like. In practice, however, UNIX implementations on MIPS use page tables so as to retain the basic memory management design. The format of the **entrylo** register is the natural form of the PTEs, and the eight low-order bits, which are unused by hardware, may be used by the kernel in any way.

The lack of *referenced* and *modified* bits places further demands on the kernel. The kernel must know which pages are modified, since they must be saved before reuse. This is achieved by write-protecting all clean pages (clearing the D bit in their TLBs), so as to force a **TLBmod** exception on the first write to them. The exception handler can then set the D bit in the TLB and set ap-

propriate bits in the software PTE to mark the page as dirty. Reference information must also be collected indirectly, as shown in Section 13.5.3.

This architecture leads to a larger number of page faults, since every TLB miss must be handled by the software. The need to track page modifications and references causes even more page faults. This is offset by the speed gained by a simpler memory architecture, which allows very fast address translation when there is a TLB cache hit. Further, the faster CPU speed helps keep down the cost of the page fault handling. Finally, the unmapped region *kseg0* is used to store the static text and data of the kernel. This increases the speed of execution of kernel code, since address translations are not required. It also reduces contention on the TLB, which is needed only for user addresses and for some dynamically allocated kernel data structures.

13.4 4.3BSD — A Case Study

So far we have described the basic concepts of demand paging, and how hardware characteristics can influence the design. To understand the issues involved more clearly, we use 4.3BSD memory management as a case study. The first UNIX system to support virtual memory was 3BSD. Its memory architecture evolved incrementally over the subsequent releases. 4.3BSD was the last Berkeley release based on this memory model. 4.4BSD adopted a new memory architecture based on Mach; this is described in Section 15.8. [Leff 89] provides a more complete treatment of 4.3BSD memory management. In this chapter, we summarize its important features, evaluate its strengths and drawbacks, and develop the motivation for the more sophisticated approaches described in the following chapters.

Although the target platform for the BSD releases was the VAX-11, it has been successfully ported to several other platforms. The hardware characteristics impact many kernel algorithms, in particular the lower-level functions that manipulate page tables and the translation buffer. Porting BSD memory management has not been easy, since the hardware dependencies permeate through all parts of the system. *As a result, several BSD-based implementations emulate the VAX memory architecture in software, including its address space layout and its page table entry format.* We avoid a detailed description of the VAX memory architecture, since the machine is now obsolete. Instead, we describe some of its important features as part of the BSD description.

4.3BSD uses a small number of fundamental data structures—the *core map* describes physical memory, the *page tables* describe virtual memory, and the *disk maps* describe the swap areas. There are also resource maps to manage allocation of resources such as page tables and swap space. Finally, some important information is stored in the `proc` structure and `u` area of each process.

13.4.1 Physical Memory

Physical memory can be viewed as a linear array of bytes ranging from 0 to *n*, where *n* is the total amount of memory on the system. It is logically divided into pages, with the page size dependent on the machine architecture. This memory can be divided into three sections, as shown in Figure 13-13. At the low end is the *nonpaged pool*, which contains the kernel code and the portion of the kernel data that can be allocated either statically or at boot time. Since a kernel page fault can block a process in the kernel at an inconvenient point, most UNIX implementations require all kernel pages to