often operate on larger sets of data or that must run many applications simultaneously may merit larger caches to offset less optimal locality properties. Computers meant to function as computation engines and network file servers can include several megabytes of L2 cache. Smaller embedded systems may suffice with only several kilobytes of L1 cache.

7.4 VIRTUAL MEMORY AND THE MMU

Multitasking operating systems execute multiple programs at the same time by assigning each program a certain percentage of the microprocessor's time and then periodically changing which instruction sequence is being executed. This is accomplished by a periodic timer interrupt that causes the OS kernel to save the state of the microprocessor's registers and then reload the registers with preserved state from a different program. Each program runs for a while and is paused, after which execution resumes without the program having any knowledge of having been paused. In this respect, the individual programs in a multitasking environment appear to have complete control over the computer, despite sharing the resources with others. Such a perspective makes programming for a multitasking OS easier, because the programmer does not have to worry about the infinite permutations of other applications that may be running at any given time. A program can be written as if it is the only application running, and the OS kernel sorts out the run-time responsibilities of making sure that each application gets fair time to run on the microprocessor.

Aside from fair access to microprocessor time, conflicts can arise between applications that accidentally modify portions of each other's memory—either program or data. How does an application know where to locate its data so that it will not disturb that of other applications and so that it will not be overwritten? There is also the concern about system-wide fault tolerance. Even if not malicious, programs may have bugs that cause them to crash and write data to random memory locations. In such an instance, one errant application could bring down others or even crash the OS if it overwrites program and data regions that belong to the OS kernel. The first problem can be addressed with the honor system by requiring each application to dynamically request memory allocations at run time from the kernel. The kernel can then make sure that each application is granted an exclusive region of memory. However, the second problem of errant writes requires a hardware solution that can physically prevent an application from accessing portions of memory that do not belong to it.

Virtual memory is a hardware enforced and software configured mechanism that provides each application with its own private memory space that it can use arbitrarily. This virtual memory space can be as large as the microprocessor's addressing capability—a full 4 GB in the case of a 32-bit microprocessor. Because each application has its own exclusive virtual memory space, it can use any portion of that space that is not otherwise restricted by the kernel. Virtual memory frees the programmer from having to worry about where other applications may locate their instructions or data, because applications cannot access the virtual memory spaces of others. In fact, operating systems that support virtual memory may simplify the physical structure of programs by specifying a fixed starting address for instructions, the local stack, and data. UNIX is an example of an OS that does this. Each application has its instructions, stack, and data at the same virtual addresses, because they have separate virtual memory spaces that are mutually exclusive and, therefore, not subject to conflict.

Clearly, multiple programs cannot place different data at the same address or each simultaneously occupy the microprocessor's entire address space. The OS kernel configures a hardware *memory management unit* (MMU) to map each program's *virtual addresses* into unique *physical addresses* that correspond to actual main memory. Each unique virtual memory space is broken into many small *pages* that are often in the range of 2 to 16 kB in size (4 kB is a common page size). The OS and MMU refer to each virtual memory space with a *process ID* (PID) field. Virtual memory is han-

dled on a process basis rather than an application basis, because it is possible for an application to consist of multiple semi-independent processes. The high-order address bits referenced by each instruction form the *virtual page number* (VPN). The PID and VPN are combined to uniquely map to a physical address set aside by the kernel as shown in Fig. 7.5. Low-order address bits represent offsets that directly index into mapped pages in physical memory. The mapping of virtual memory pages into physical memory is assigned arbitrarily by the OS kernel. The kernel runs in real memory rather than in virtual memory so that it can have direct access to the computer's physical resources to allocate memory as individual processes are executed and then terminated.

Despite each process having a 4-GB address space, virtual memory can work on computers with just megabytes of memory, because the huge virtual address spaces are sparsely populated. Most processes use only a few hundred kilobytes to a few megabytes of memory and, therefore, multiple processes that collectively have the potential to reference tens of gigabytes can be mapped into a much smaller quantity of real memory. If too many processes are running simultaneously, or if these processes start to consume too much memory, a computer can exhaust its physical memory resources, thereby requiring some intervention from the kernel to either suspend a process or handle the problem in some other way.

When a process is initiated, or *spawned*, it is assigned a PID and given its own virtual memory space. Some initial pages are allocated to hold its instructions and whatever data memory the process needs available when it begins. During execution, processes may request more memory from the kernel by calling predefined kernel memory management routines. The kernel will respond by allocating a page in physical memory and then returning a pointer to that page's virtual mapping. Likewise, a process can free a memory region when it no longer needs it. Under this circumstance, the kernel will remove the mapping for the particular pages, enabling them to be reallocated to another process, or the same process, at a later time. Therefore, the state of memory in a typical multitasking OS is quite dynamic, and the routines to manage memory must be implemented in software because of their complexity and variability according to the platform and the nature of processes running at any given time.

Not all mapped virtual memory pages have to be held in physical RAM at the same time. Instead, the total virtual memory allocation on a computer can spill over into a secondary storage medium such as a hard drive. The hard drive will be much slower than DRAM, but not every memory page in every process is used at the same time. When a process is first loaded, its entire instruction image is typically loaded into virtual memory. However, it will take some time for all of those instructions to reach their turn in the execution sequence. During this wait time, the majority of a process's program

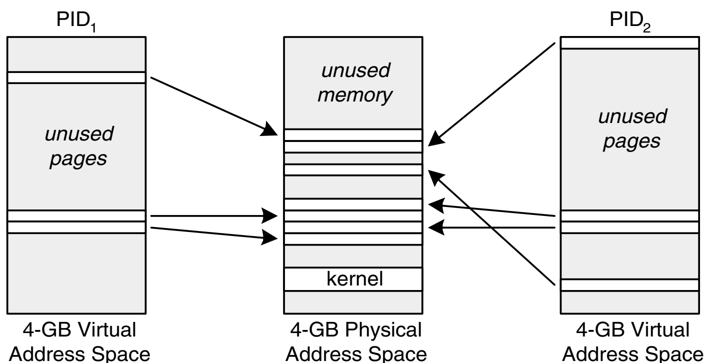

FIGURE 7.5 32-bit virtual memory mapping.