- Limit fields restrict the size of a process and the size of a file it can *write*.
- A permission modes field masks mode settings on files the process *creat*s.

This section has described the process state transitions on a logical level. Each state has physical characteristics managed by the kernel, particularly the virtual address space of the process. The next section describes a model for memory management; later sections describe the states and state transitions at a physical level, focusing on the states "user running," "kernel running," "preempted," and "sleep (in memory)." The next chapter describes the states "created" and "zombie," and Chapter 8 describes the state "ready to run in memory." Chapter 9 discusses the two "swap" states and demand paging.

6.2 LAYOUT OF SYSTEM MEMORY

Assume that the physical memory of a machine is addressable, starting at byte offset 0 and going up to a byte offset equal to the amount of memory on the machine. As outlined in Chapter 2, a process on the UNIX system consists of three logical sections: text, data, and stack. (*Shared memory*, discussed in Chapter 11, should be considered part of the data section for purposes of this discussion.) The text section contains the set of instructions the machine executes for the process; addresses in the text section include text addresses (for branch instructions or subroutine calls), data addresses (for access to global data variables), or stack addresses (for access to data structures local to a subroutine). If the machine were to treat the generated addresses as address locations in physical memory, it would be impossible for two processes to execute concurrently if their set of generated addresses overlapped. The compiler could generate addresses that did not overlap between programs, but such a procedure is impractical for general-purpose computers because the amount of memory on a machine is finite and the set of all programs that could be compiled is infinite. Even if the compiler used heuristics to try to avoid unnecessary overlap of generated addresses, the implementation would be inflexible and therefore undesirable.

The compiler therefore generates addresses for a *virtual address space* with a given address range, and the machine's memory management unit translates the virtual addresses generated by the compiler into address locations in physical memory. The compiler does not have to know where in memory the kernel will later load the program for execution. In fact, several copies of a program can coexist in memory: All execute using the same virtual addresses but reference different physical addresses. The subsystems of the kernel and the hardware that cooperate to translate virtual to physical addresses comprise the *memory management* subsystem.

6.2.1 Regions

The System V kernel divides the virtual address space of a process into logical *regions*. A region is a contiguous area of the virtual address space of a process that can be treated as a distinct object to be shared or protected. Thus text, data, and stack usually form separate regions of a process. Several processes can share a region. For instance, several processes may execute the same program, and it is natural that they share one copy of the text region. Similarly, several processes may cooperate to share a common shared-memory region.

The kernel contains a region table and allocates an entry from the table for each active region in the system. Section 6.5 will describe the fields of the region table and region operations in greater detail, but for now, assume the region table contains the information to determine where its contents are located in physical memory. Each process contains a private *per process region table*, called a *pregion* for short. Pregion entries may exist in the process table, the *u area*, or in a separately allocated area of memory, dependent on the implementation, but for simplicity, assume that they are part of the process table entry. Each pregion entry points to a region table entry and contains the starting virtual address of the region in the process. Shared regions may have different virtual addresses in each process. The pregion entry also contains a permission field that indicates the type of access allowed the process: read-only, read-write, or read-execute. The pregion and the region structure are analogous to the file table and the inode structure in the file system: Several processes can share parts of their address space via a region, much as they can share access to a file via an inode; each process accesses the region via a private pregion entry, much as it accesses the inode via private entries in its user file descriptor table and the kernel file table.

Figure 6.2 depicts two processes, A and B, showing their regions, pregions, and the virtual addresses where the regions are connected. The processes share text region 'a' at virtual addresses 8K and 4K, respectively. If process A reads memory location 8K and process B reads memory location 4K, they read the identical memory location in region 'a'. The data regions and stack regions of the two processes are private.

The concept of the region is independent of the memory management policies implemented by the operating system. Memory management policy refers to the actions the kernel takes to insure that processes share main memory fairly. For example, the two memory management policies considered in Chapter 9 are process swapping and demand paging. The concept of the region is also independent of the memory management implementation: whether memory is divided into pages or segments, for example. To lay the foundation for the description of demand paging algorithms in Chapter 9, the discussion here assumes a memory architecture based on pages, but it does not assume that the memory management policy is based on demand paging algorithms.

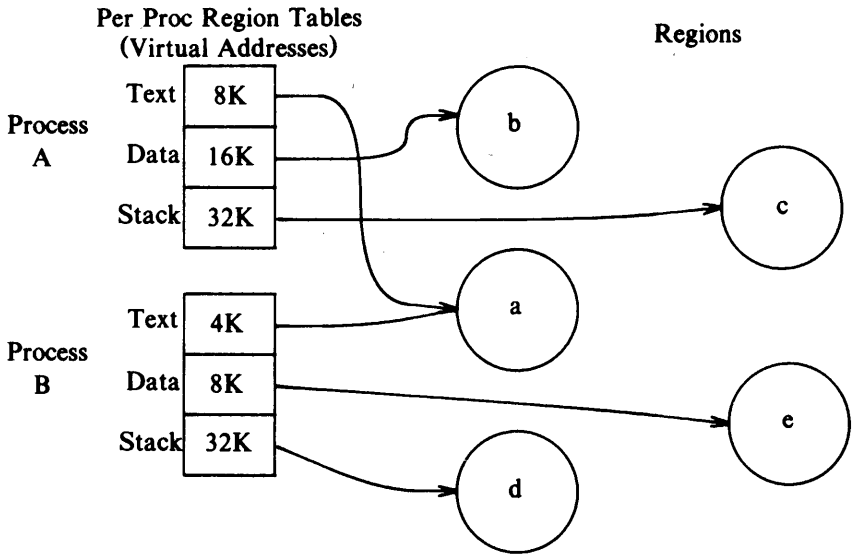

Figure 6.2. Processes and Regions

6.2.2 Pages and Page Tables

This section defines the memory model that will be used throughout this book, but it is not specific to the UNIX system. In a memory management architecture based on *pages*, the memory management hardware divides physical memory into a set of equal-sized blocks called pages. Typical page sizes range from 512 bytes to 4K bytes and are defined by the hardware. Every addressable location in memory is contained in a page and, consequently, every memory location can be addressed by a

(page number, byte offset in page)

pair. For example, if a machine has 2^{32} bytes of physical memory and a page size of 1K bytes, it has 2^{22} pages of physical memory; every 32-bit address can be treated as a pair consisting of a 22-bit page number and a 10-bit offset into the page (Figure 6.3).

When the kernel assigns physical pages of memory to a region, it need not assign the pages contiguously or in a particular order. The purpose of paged memory is to allow greater flexibility in assigning physical memory, analogous to the assignment of disk blocks to files in a file system. Just as the kernel assigns blocks to a file to increase flexibility and to reduce the amount of unused space caused by block fragmentation, so it assigns pages of memory to a region.

Hexadecimal Address	58432	
Binary	0101 1000 0100 0011 0010	
Page Number, Page Offset	01 0110 0001	00 0011 0010
In Hexadecimal	161	32

Figure 6.3. Addressing Physical Memory as Pages

Logical Page Number	Physical Page Number
0	177
1	54
2	209
3	17

Figure 6.4. Mapping of Logical to Physical Page Numbers

The kernel correlates the virtual addresses of a region to their physical machine addresses by mapping the logical page numbers in the region to physical page numbers on the machine, as shown in Figure 6.4. Since a region is a contiguous range of virtual addresses in a program, the logical page number is the index into an array of physical page numbers. The region table entry contains a pointer to a table of physical page numbers called a *page table*. Page table entries may also contain machine-dependent information such as permission bits to allow reading or writing of the page. The kernel stores page tables in memory and accesses them like all other kernel data structures.

Figure 6.5 shows a sample mapping of a process into physical memory. Assume that the size of a page is 1K bytes, and suppose the process wants to access virtual memory address 68,432. The pregion entries show that the virtual address is in the stack region starting at virtual address 64K (65,536 in decimal), assuming the direction of stack growth is towards higher addresses. Subtracting, address 68,432 is at byte offset 2896 in the region. Since each page consists of 1K bytes, the address is contained at byte offset 848 in page 2 (counting from 0) of the region, located at physical address 986K. Section 6.5.5 (loading a region) discusses the meaning of the page table entry marked "empty."

Modern machines use a variety of hardware registers and caches to speed up the address translation procedure just described, because the memory references and address calculations would otherwise be too slow. When resuming the execution of a process, the kernel therefore informs the memory management

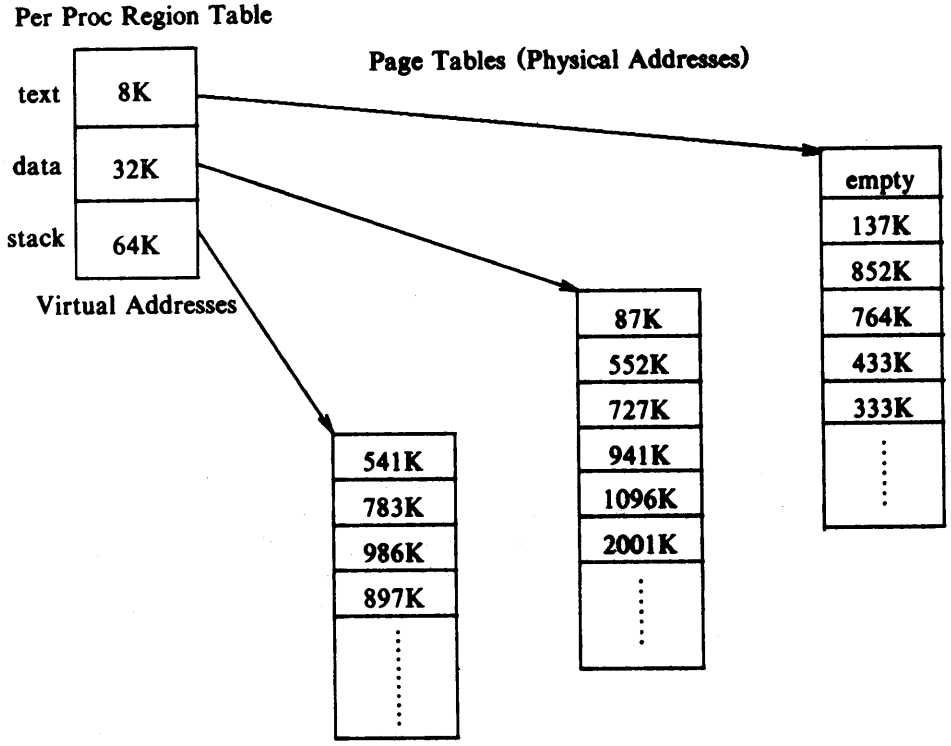

Figure 6.5. Mapping Virtual Addresses to Physical Addresses

hardware where the page tables and physical memory of the process reside by loading the appropriate registers. Since such operations are machine dependent and vary from one implementation to another, this text will not discuss them. The exercises at the end of the chapter cite specific machine architectures.

Let us use the following simple memory model in discussing memory management. Memory is organized in pages of 1K bytes, accessed via page tables as described earlier. The system contains a set of memory management register triples (assume a large supply), such that the first register in the triple contains the address of a page table in physical memory, the second register contains the first virtual address mapped via the triple, and the third register contains control information such as the number of pages in the page table and page access permissions (read-only, read-write). This model corresponds to the region model, just described. When the kernel prepares a process for execution, it loads the set of memory management register triples with the corresponding data stored in the pregion entries.

If a process addresses memory locations outside its virtual address space, the hardware causes an exception condition. For example, if the size of the text region in Figure 6.5 is 16K bytes and a process accesses virtual address 26K, the hardware will cause an exception that the operating system handles. Similarly, if a process tries to access memory without proper permissions, such as writing an address in its write-protected text region, the hardware will cause an exception. In both these examples, the process would normally *exit*; the next chapter provides more detail.

6.2.3 Layout of the Kernel

Although the kernel executes in the context of a process, the virtual memory mapping associated with the kernel is independent of all processes. The code and data for the kernel reside in the system permanently, and all processes share it. When the system is brought into service (booted), it loads the kernel code into memory and sets up the necessary tables and registers to map its virtual addresses into physical memory addresses. The kernel page tables are analogous to the page tables associated with a process, and the mechanisms used to map kernel virtual addresses are similar to those used for user addresses. In many machines, the virtual address space of a process is divided into several classes, including system and user, and each class has its own page tables. When executing in kernel mode, the system permits access to kernel addresses, but it prohibits such access when executing in user mode. Thus, when changing mode from user to kernel as a result of an interrupt or system call, the operating system collaborates with the hardware to permit kernel address references, and when changing mode back to user, the operating system and hardware prohibit such references. Other machines change the virtual address translation by loading special registers when executing in kernel mode.

Figure 6.6 gives an example of the virtual addresses of the kernel and a process, where kernel virtual addresses range from 0 to 4M−1 and user virtual addresses range from 4M up. There are two sets of memory management triples, one for kernel addresses and one for user addresses, and each triple points to a page table that contains the physical page numbers corresponding to the virtual page addresses. The system allows address references via the kernel register triples only when in kernel mode; hence, switching mode between kernel and user requires only that the system permit or deny address references via the kernel register triples.

Some system implementations load the kernel into memory such that most kernel virtual addresses are identical to their physical addresses and the virtual to physical memory map of those addresses is the identity function. However, the treatment of the *u area* requires virtual to physical address mapping in the kernel.

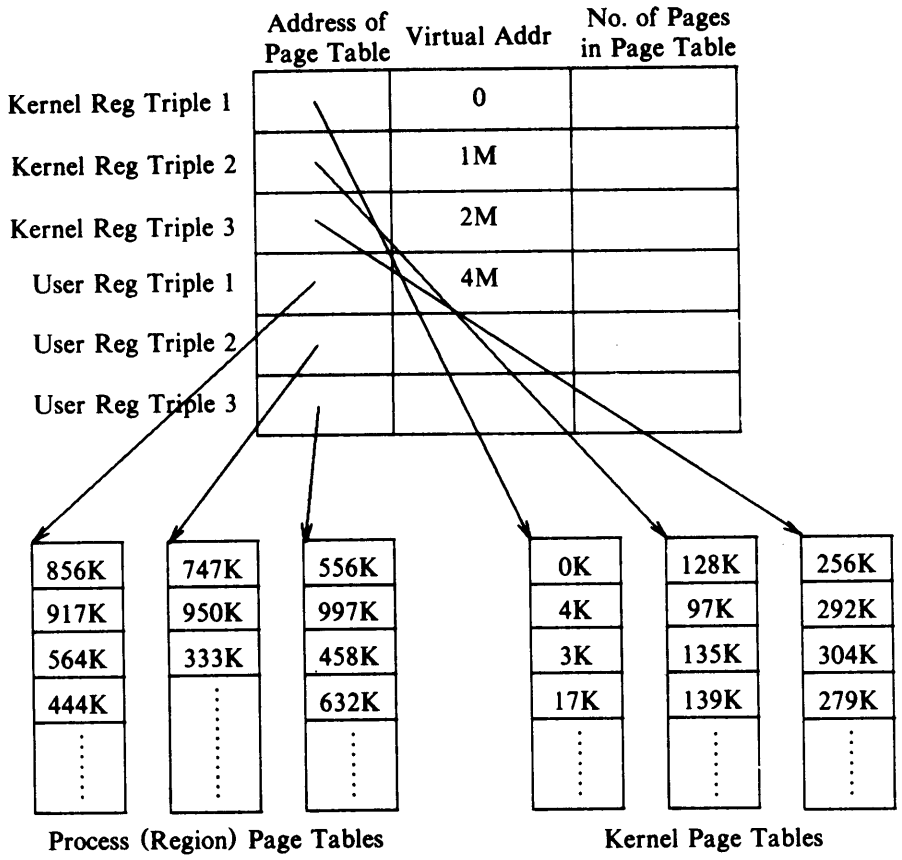

Figure 6.6. Changing Mode from User to Kernel

6.2.4 The U Area

Every process has a private *u area*, yet the kernel accesses it as if there were only one *u area* in the system, that of the running process. The kernel changes its virtual address translation map according to the executing process to access the correct *u area*. When compiling the operating system, the loader assigns the variable *u*, the name of the *u area*, a fixed virtual address. The value of the *u area* virtual address is known to other parts of the kernel, in particular, the module that does the context switch (Section 6.4.3). The kernel knows where in its memory management tables the virtual address translation for the *u area* is done, and it can dynamically change the address mapping of the *u area* to another physical address. The two physical addresses represent the *u areas* of two processes, but the kernel

accesses them via the same virtual address.

A process can access its *u area* when it executes in kernel mode but not when it executes in user mode. Because the kernel can access only one *u area* at a time by its virtual address, the *u area* partially defines the context of the process that is running on the system. When the kernel schedules a process for execution, it finds the corresponding *u area* in physical memory and makes it accessible by its virtual address.

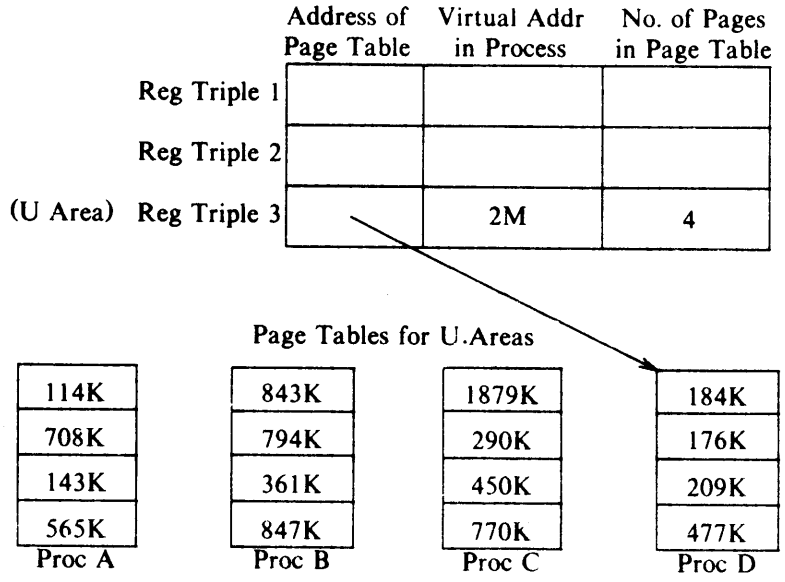

Figure 6.7. Memory Map of U Area in the Kernel

For example, suppose the *u area* is 4K bytes long and resides at kernel virtual address 2M. Figure 6.7 shows a sample memory layout, where the first two register triples refer to kernel text and data (the addresses and pointers are not shown), and the third triple refers to the *u area* for process D. If the kernel wants to access the *u area* of process A, it copies the appropriate page table information for the *u area* into the third register triple. At any instant, the third kernel register triple refers to the *u area* of the currently running process, but the kernel can refer to the *u area* of another process by overwriting the entries for the *u area* page table address with a new address. The entries for register triples 1 and 2 do not change for the kernel, because all processes share kernel text and data.

6.3 THE CONTEXT OF A PROCESS

The context of a process consists of the contents of its (user) address space and the contents of hardware registers and kernel data structures that relate to the process. Formally, the context of a process is the union of its *user-level context*, *register context*, and *system-level context*.[1] The user-level context consists of the process text, data, user stack, and shared memory that occupy the virtual address space of the process. Parts of the virtual address space of a process that periodically do not reside in main memory because of swapping or paging still constitute a part of the user-level context.

The register context consists of the following components.

- The program counter specifies the address of the next instruction the CPU will execute; the address is a virtual address in kernel or in user memory space.
- The processor status register (PS) specifies the hardware status of the machine as it relates to the process. For example, the PS usually contains subfields to indicate that the result of a recent computation resulted in a zero, positive or negative result, or that a register overflowed and a carry bit is set, and so on. The operations that caused the PS to be set were done for a particular process, hence the PS contains the hardware status of the machine as it relates to the process. Other important subfields typically found in the PS are those that indicate the current processor execution level (for interrupts) and the current and most recent modes of execution (such as kernel, user). The subfield that shows the current execution mode determines whether a process can execute privileged instructions and whether it can access kernel address space.
- The stack pointer contains the current address of the next entry in the kernel or user stack, determined by the mode of execution. Machine architectures dictate whether the stack pointer points to the next free entry on the stack or to the last used entry. Similarly, the machine dictates the direction of stack growth toward numerically higher or lower addresses, but such issues are immaterial for purposes of this discussion.
- The general-purpose registers contain data generated by the process during its execution. To simplify the following discussion, let us distinguish two general purpose registers, register 0 and register 1, for additional use in transmitting information between processes and the kernel.

The system-level context of a process has a "static part" (first three items of the following list) and a "dynamic part" (last two items). A process has one static part of the system-level context throughout its lifetime, but it can have a variable number of dynamic parts. The dynamic part of the system-level context should be

1. The terms *user-level context*, *register context*, *system-level context*, and *context layers* used in this section are the author's terminology.

viewed as a stack of *context layers* that the kernel pushes and pops on occurrence of various events. The system-level context consists of the following components.

- The process table entry of a process defines the state of a process, as described in Section 6.1, and contains control information that is always accessible to the kernel.
- The *u area* of a process contains process control information that need be accessed only in the context of the process. General control parameters such as the process priority are stored in the process table because they must be accessed outside the process context.
- Pregion entries, region tables and page tables, define the mapping from virtual to physical addresses and therefore define the text, data, stack, and other regions of a process. If several processes share common regions, the regions are considered part of the context of each process, because each process manipulates the regions independently. Part of the memory management task is to indicate which parts of the virtual address space of a process are not memory resident.
- The kernel stack contains the stack frames of kernel procedures as a process executes in kernel mode. Although all processes execute the identical kernel code, they have a private copy of the kernel stack that specifies their particular invocation of the kernel functions. For instance, one process may invoke the *creat* system call and go to sleep waiting for the kernel to assign a new inode, and another process may invoke the *read* system call and go to sleep awaiting the transfer of data from disk to memory. Both processes execute kernel functions, but they have separate stacks that contain their private function call sequence. The kernel must be able to recover the contents of the kernel stack and the position of the stack pointer to resume execution of a process in kernel mode. System implementations frequently place the kernel stack in the process *u area*, but it is logically independent and can exist in an independently allocated area of memory. The kernel stack is empty when the process executes in user mode.
- The dynamic part of the system-level context of a process consists of a set of layers, visualized as a last-in-first-out stack. Each *system-level context layer* contains the necessary information to recover the previous layer, including the register context of the previous level.

The kernel pushes a context layer when an interrupt occurs, when a process makes a system call, or when a process does a context switch. It pops a context layer when the kernel returns from handling an interrupt, when a process returns to user mode after the kernel completes execution of a system call, or when a process does a context switch. The context switch thus entails a push and a pop of a system-level context layer: The kernel pushes the context layer of the old process and pops the context layer of the new process. The process table entry stores the necessary information to recover the current context layer.

Figure 6.8 depicts the components that form the context of a process. The left side of the figure shows the static portion of the context. It consists of the user-

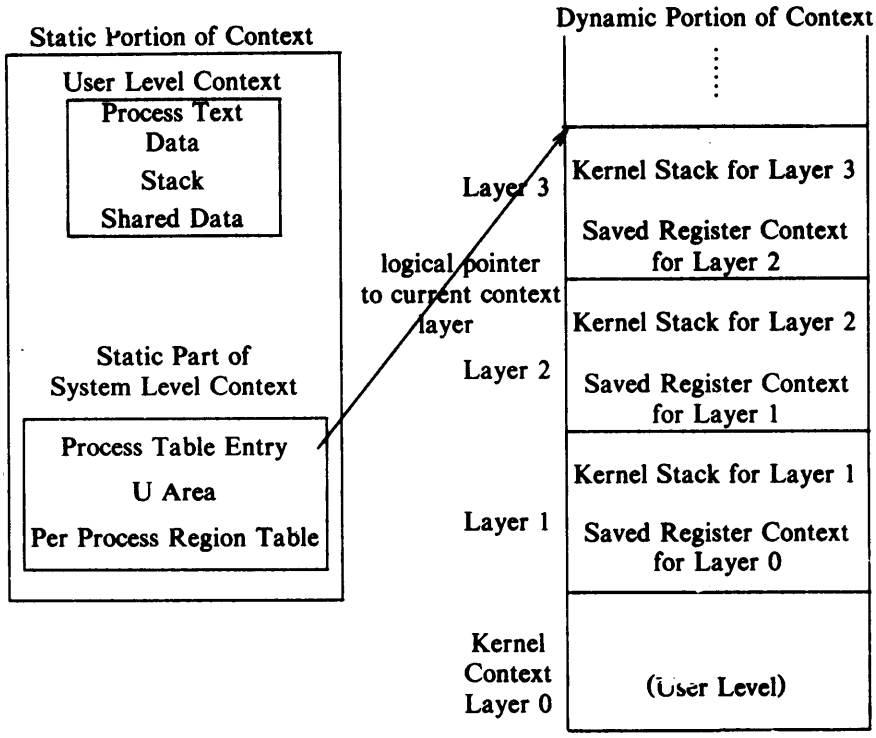

Figure 6.8. Components of the Context of a Process

level context, containing the process text (instructions), data, stack, and shared memory (if the process has any), and the static part of the system-level context, containing the process table entry, the *u area*, and the pregion entries (the virtual address mapping information for the user-level context). The right side of the figure shows the dynamic portion of the context. It consists of several stack frames, where each frame contains the saved register context of the previous layer, and the kernel stack as the kernel executes in that layer. System context layer 0 is a dummy layer that represents the user-level context; growth of the stack here is in the user address space, and the kernel stack is null. The arrow pointing from the static part of the system-level context to the top layer of the dynamic portion of the context represents the logical information stored in the process table entry to enable the kernel to recover the current context layer of the process.

A process runs within its context or, more precisely, within its current context layer. The number of context layers is bounded by the number of interrupt levels the machine supports. For instance, if a machine supports different interrupt levels for software interrupts, terminals, disks, all other peripherals, and the clock, it

supports 5 interrupt levels, and hence, a process can contain at most 7 context layers: 1 for each interrupt level, 1 for system calls, and 1 for user-level. The 7 layers are sufficient to hold all context layers even if interrupts occur in the "worst" possible sequence, because an interrupt of a given level is blocked (that is, the CPU defers it) while the kernel handles interrupts of that level or higher.

Although the kernel always executes in the context of some process, the logical function that it executes does not necessarily pertain to that process. For instance, if a disk drive interrupts the machine because it has returned data, it interrupts the running process and the kernel executes the interrupt handler in a new system-level context layer of the executing process, even though the data belongs to another process. Interrupt handlers do not generally access or modify the static parts of the process context, since those parts have nothing to do with the interrupt.

6.4 SAVING THE CONTEXT OF A PROCESS

As observed in previous sections, the kernel saves the context of a process whenever it pushes a new system context layer. In particular, this happens when the system receives an interrupt, when a process executes a system call, or when the kernel does a context switch. This section considers each case in detail.

6.4.1 Interrupts and Exceptions

The system is responsible for handling interrupts, whether they result from hardware (such as from the clock or from peripheral devices), from a programmed interrupt (execution of instructions designed to cause "software interrupts"), or from exceptions (such as page faults). If the CPU is executing at a lower processor execution level than the level of the interrupt, it accepts the interrupt before decoding the next instruction and raises the processor execution level, so that no other interrupts of that level (or lower) can happen while it handles the current interrupt, preserving the integrity of kernel data structures (see Section 2.2.2). The kernel handles the interrupt with the following sequence of operations:

1. It saves the current register context of the executing process and creates (pushes) a new context layer.
2. It determines the "source" or cause of the interrupt, identifying the type of interrupt (such as clock or disk) and the unit number of the interrupt, if applicable (such as which disk drive caused the interrupt). When the system receives an interrupt, it gets a number from the machine that it uses as an offset into a table, commonly called an *interrupt vector*. The contents of interrupt vectors vary from machine to machine, but they usually contain the address of the interrupt handler for the corresponding interrupt source and a way of finding a parameter for the interrupt handler. For example, consider the table of interrupt handlers in Figure 6.9. If a terminal interrupts the system, the kernel gets interrupt number 2 from the hardware and invokes the