CHAPTER 3

The Kernel: Basic Organization

The study of a modern operating system leads down many paths and requires that we consider a number of different challenges and their solutions. The HP-UX kernel is a multitasking, multiuser, multiprocessor, multithreaded, load-leveling, modular operating system with real-time scheduling extensions—to list just the highlights. To support such capabilities requires many levels of design abstraction, data tables, and lists as well as a host of subsystems, drivers, and dynamic modules.

In this chapter, we examine the basic organization of the kernel and its data structures, and we consider which are dependent on the underlying hardware platform (hardware dependent-layer, HDL) and which are independent (hardware-independent layer, HIL).

Before addressing HP-UX-specific topics, let's stop and think about what an operating system really is.

A Generic Overview

Approaching an operating system as a whole can be a bit overwhelming, so let's break it down a bit. Think of an operating system simply as a bootable piece of application code. True, it is a somewhat large piece of application code, and it employs many diverse and complex functionalities (even an abstracted form of self-modifying code), but in the final analysis it is still just an executable image!

A programmer must design and create data structures to store and manipulate data in a logical and efficient manner to support the operation and design goals of the program. An operating system designer faces the same challenge, only in spades. Understanding and identifying the resulting data structures and their interaction is a major focus of this book.

The UNIX operating system design is very simple.

This statement usually draws an assortment of looks or comments; indeed, some may question its basic premise or perhaps the qualifications of the one who spoke it! The essence that the statement is meant to draw out is that in the design of a UNIX operating system there are a few key elements (see Figure 3-1).

- First, and most important, a UNIX operating system is responsible for the scheduling and management of individual threads of execution. Later, we discuss processes and threads, but for now let's just consider a thread of execution as an operational piece of code, something that is accomplishing "work" on behalf of a "user." Who gets to run, when, and how long are the issues under the control of the operating system.
- UNIX is the ultimate control freak—it is prosecutor, judge, and jury for all code that attempts to circumvent its authority. The kernel manages access to all system resources. The only way executing code may make use of a system resource is to request access through well-defined programmatic kernel hooks (called *system calls*).
- All I/O is file I/O. That is to say that all devices are defined by "special" device file handles and treated as if they are simple files.
- All forms of synchronous and asynchronous interruptions are handled by kernel routines. Simply put, if it is unexpected, then let the kernel figure it out.
- System resources should be available to all requestors in a balanced manner when possible.

These are the prime directives of a UNIX kernel's statement of design. While this may seem an oversimplification, it is our hope that by setting these elements as our focal points and by relating other topics and features back to them, we will stay on track.

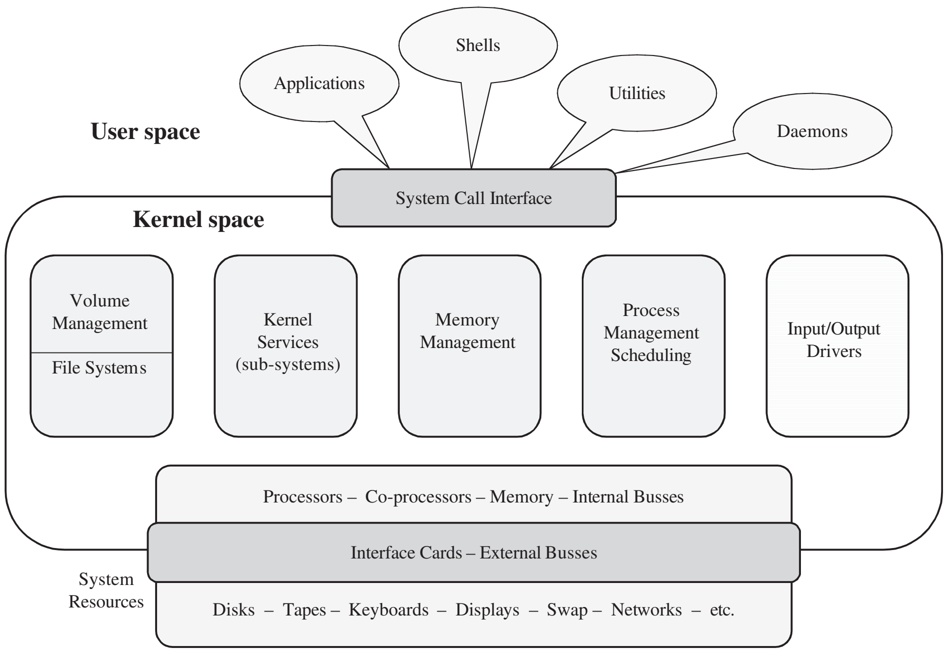
Shells
Applications
Utilities
Daemons
User space
System Call Interface
Kernel space
Volume Management
File Systems
Kernel Services (sub-systems)
Memory Management
Process Management Scheduling
Input/Output Drivers
Processors – Co-processors – Memory – Internal Busses
Interface Cards – External Busses
System Resources
Disks – Tapes – Keyboards – Displays – Swap – Networks – etc.

Figure 3-1 The Big Picture

The UNIX kernel is very modular in its design and evolution. The design was influenced in large part by the C programming language—itself a very modular language—in which the bulk of UNIX kernel code is written. A lot of the personality of the original "kernel hackers" (a term used with great respect) is still present in the heart and soul of today's UNIX incarnations.

When studying a UNIX operating system, always be vigilant for similarities between different sections of the kernel code; that is, look for variations on a theme. It will help your comprehension to compare and contrast the various programmatic tools and tricks of the trade as they come into focus during subsequent chapters.

All I/O Is File I/O

Continuing with the theme of simplification, let's think about the fundamentals of process I/O and the UNIX kernel. The UNIX kernel is charged with policing all forms of I/O as part of its resource management duties. To simplify this task, UNIX has reduced all types of I/O to the level of file I/O. By representing all external devices as files, the system needs only one type of access-control mechanism. This is the infamous "rwx rwx rwx," user group and other (or UGO) security model presented in all introductory-level UNIX training courses and books. UNIX basically follows the KISS (keep it simple, stupid) principle of design.

Abstraction: A Fundamental of Kernel Design

How is an external tape drive or the key on a keyboard reduced to a file path? Through the use of abstraction, a filename (known as a *device file* or *special file*) references a driver in the kernel and passes operational parameters to it (more on this in Chapter 10, "I/O and Device Management." For now, suffice it to say that things are often not what they appear to be at first glance. The kernel contains many layers of abstraction and indirection—smoke and mirrors, my friend, smoke and mirrors. Our challenge is to blow away the smoke and study the reflections in the mirrors.

Is It Real or Is It Virtual?

A major portion of the UNIX operating system is devoted to the management of and translation between "real" and "virtual" addressing modes. From the viewpoint of a process, all possible memory locations fall within a logical address range: 32-bit applications are called *narrow*, and 64-bit applications are called *wide*. The kernel also comes in both narrow and wide versions, usually dictated by the width of the processor architecture on which it is running.

When a program's source code is compiled, references to individual execution modules, library routines, and data elements are stored as symbolic names. This type of code module is called an *object module*. A symbol table is required to define the name and attributes of each item in the table.

When all the related modules of a program have been collected, a linking-loader is used to create the final executable image. The *loader* orders all the individual items within the executable image, and the *linker* replaces all the symbolic references with the logical addresses of the objects that have been loaded. The resulting "image" is fixed within the process's logical address range. As the kernel and all of its processes must share the available physical, or "real," memory, some type of translation must be performed between the process's logical address and the system's physical address. To facilitate this abstraction, an address translation scheme is used.

Most UNIX operating systems employ a concept known as *virtual addressing*. In a virtual memory system, the kernel maintains and manages an address space that is many times larger than the physical memory size addressable by the hardware. This address space exists only as an organizational definition and requires constant translation to true physical addresses during program execution. A virtual memory system requires hardware support as well as implementation in kernel code.

The major advantage of a virtual memory system implementation is that it allows many processes to coexist within the virtual address space (VAS). Each process is allowed its own logical view. Some regions of the virtual space are kept private for a single process; others may be maintained by the kernel as shared regions (see Figure 3-2). This is the basis for shared code and data objects, and is the focus of an entire chapter later in this book (Chapter 10, "Memory Management").

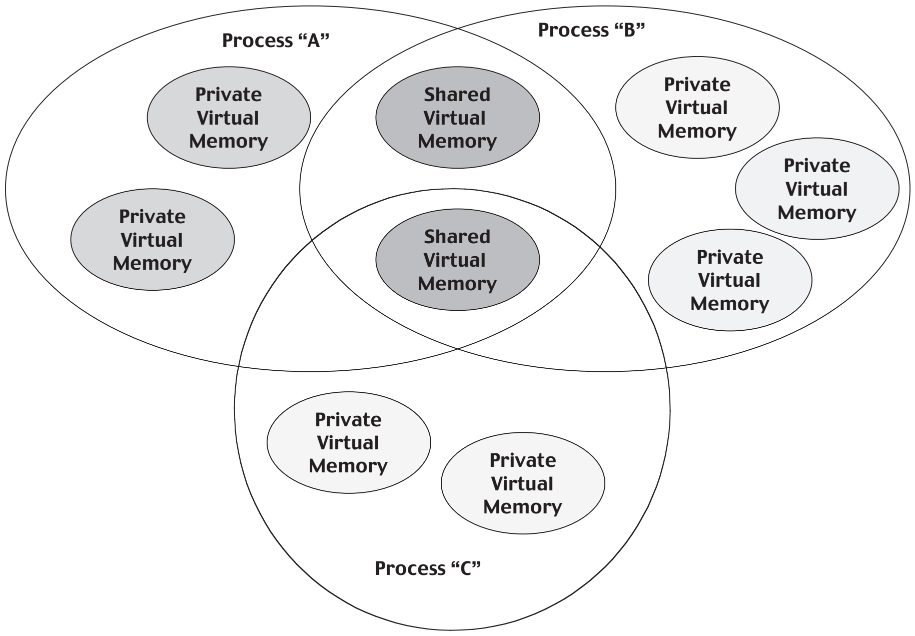

Figure 3-2 Virtual Memory Objects, Private and Shared

Abstraction Layers

We have used the term *abstraction* and should spend a moment talking about what it means with regard to kernel design. When we consider the task of resource management, we must determine the level of control we wish to implement in our management scheme. One of the tricks of the trade involves grouping individual components of a system resource into uniform-sized blocks, chunks, extents, pages, and so on. The availability of the resource is then tracked at the level of these granular units, thus reducing the complexity of kernel data structures.

A classic case is that of memory management. A computer's physical RAM consists of circuits representing single bits of data storage; these are combined into sets of eight and referred to as bytes. An operating system combines bytes into words (for HP-UX, a word is 32 bits, or 4bytes; this is true for both narrow and wide kernels). The word is still a very small amount of storage space, and if the kernel needed to manage each word (by manage, we mean keep track of which words are currently being used, which are free, and who is using what), the amount of memory needed to build such structures could easily require as much space as or more than the memory being managed!

To reduce this management overhead, we make the managed unit size larger than a word. In UNIX, this is accomplished by combining sequential physical words of memory into units called *page frames* (on HP-UX the page frame is 4096 bytes, or 1024 words). Now the task of keeping track of what is free and what is in use becomes much simpler. This is a very basic layer of abstraction; the kernel manages page frames, which you and I know are actually blocks of words made up of bytes that are 8-bits each.

A modern UNIX kernel may use multiple layers of abstraction. Let's continue with our discussion of fundamental memory management. UNIX kernels often employ a scheme whereby a process that requires a number of page frames to hold its code is assigned an appropriately sized *region* in a virtual page-frame map. Virtual page frames are mapped to specific physical page frames by means of processor hardware and supporting kernel tables (discussed later in this book). This additional layer of abstraction greatly simplifies issues such as allowing two or more processes to share the same view of executable code, shared libraries, shared memory, and other process-level shared objects.

There are structures in the kernel to keep track of each element at each abstraction layer. Entities at higher layers simply point to the tracking structures at the lower layers. Lower level resource attributes are inherited by the upper abstraction layers.

Care must be taken in deciding the size of each management unit—too large and you may waste a limited resource; too small and the overhead of the tracking structures may be excessive. The kernel designer constantly walks a fine line between convenience and efficiency. As you study resource management, note the granularity of control the kernel has over its charges.

Some Generic Kernel Techniques

The discussion of operating system internals presents many challenges and opportunities to an author. Our approach is to discuss each area of the kernel, consider the challenges faced by kernel designers, and then explore the path taken toward the final solution implemented in the HP-UX code.

Before we talk about HP-UX specifics, let's discuss some generic challenges faced by kernel designers. As with any programming assignment, there are frequently many different ways to approach and solve a problem. Sometimes the decision is based on the programmer's past experience, and sometimes it is dictated by the specific requirements of each kernel design feature. As an operating system matures, these individual point solutions are often modified or "tweaked" in order to tune a kernel's operating parameters and bring them into alignment with performance objectives or system benchmark metrics. The HP-UX operating system is the product of a continuous improvement process that has enabled the refinement of core features and introduced many enhancements and services over the years.

Kernel Data Structures

Programmers often use algorithms or borrow coding techniques from a "bag-of-tricks" that belongs to the software development community at large. This common knowledge base contains many elements of special interest to those who craft operating system code. Let's explore some of the challenges that kernel programmers face and try to develop a basic understanding of a few of the common programmatic solutions they employ.

Static Lists (Static Tables)

The kernel often needs to maintain an extensive list of parameters related to some data structure or managed resource. The simplest way to store this type of data is to create an ordered, static list of the attributes of each member of the list.

Each data structure is defined according to the individual pieces of data assigned to each element. Once each parameter is typed, the size (in bytes) of the structure is known. These structures are then stored in contiguous kernel space (as an array of structures) and may be easily indexed for fast access.

As a general observation, and by no means a hard and fast rule, the naming convention of these lists may resemble the following pattern (see Figure 3-3). If the name of the data structure for a single member of the list is defined as `data_t`, then the kernel pointer to the start of the list would be `data*`. The list could also be referenced by an array named `data[x]`, and the number of elements would be stored in `ndata`. Many examples in the kernel follow this convention, but by no means all of them.

Pros

The space needed for a static list must be allocated during system initialization and is often controlled by a *kernel-tunable parameter*, which is set prior to the building of the kernel image. The first entry in a static data table has the index value of 0, which facilitates easy calculation of the starting address of each element within the table (assuming a fixed size for each member element).

Example

Assume that each data structure contains exactly 64 bytes of data and that the beginning of the static list is defined by the kernel symbol `mylist`. If you wanted to access the data for the list member with an index number of 14, you could simply take the address stored in the kernel pointer `mylist*` and add 14×64 to it to arrive at the byte address corresponding to the beginning of the 15th element in the list (don't forget that the list index starts with 0). If the structure is defined as an array, you could simplify the access by referencing `mylist[14]` in your code.

Cons

The main drawback to this approach is that the kernel must provide enough list elements for all potential scenarios that it may encounter. Many system administrators are considered godlike, but very few are truly clairvoyant! In the case of a static list, the only way for it to grow is for the kernel to be rebuilt and the system rebooted.

Another consideration is that the resource being managed must have an index number associated with it wherever it needs to be referenced within the kernel. While this may seem simple at first, think about the scenarios of initial assignment, index number reuse, resource sharing and locking, and so on.

Summary

While this type of structure is historically one of the most common, its lack of dynamic sizing and requirement to plan for the worst case has put it on the hit list for many kernel improvement projects.

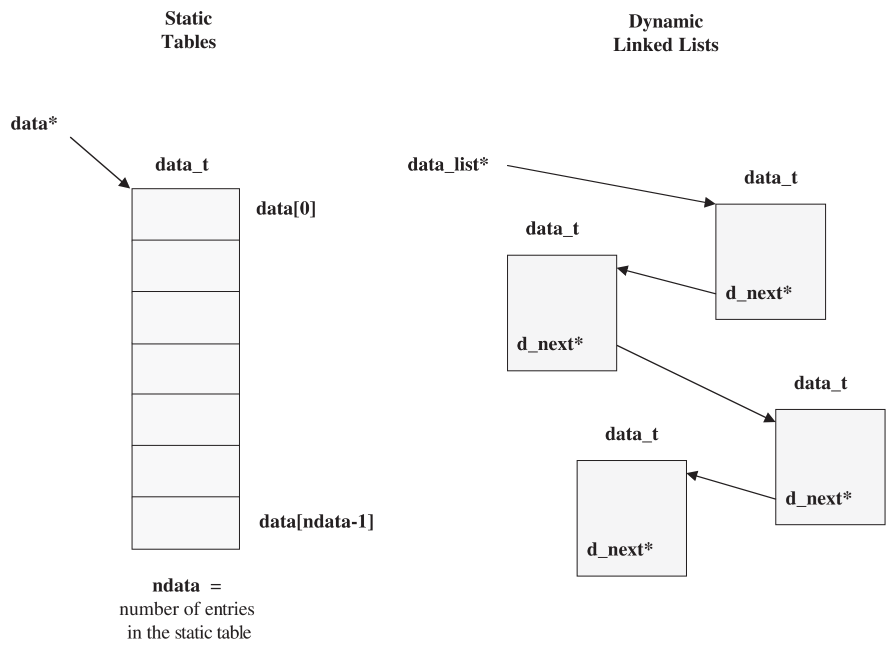

Figure 3-3 Tables and Lists

Dynamic Linked Lists (Dynamic Tables)

The individual elements of a list must be maintained in a manner that allows the kernel to monitor and manage them. Unlike the elements in a static list, all the elements of a dynamic list are not neatly grouped together in a contiguous memory space. Their individual locations and relative order are not known or predictable to the kernel (as the name “dynamic” indicates).

It is a relatively simple task to add elements to a list as they are requested (providing the kernel has an efficient kernel memory-management algorithm, which is discussed later). Once a data structure has been allocated, it must be linked with other structures of the same list. Linkage methods vary in complexity and convenience.

Once a structure has been allocated and the correct data stored, the challenge is in accessing the data in a timely manner. A simple index will not suffice due to the noncontiguous nature of the individual list elements. The choice is to "walk" the list by following forward pointers inserted into each list element as a means of building a continuous path through the list or to implement some other type of index data structure or hash function. While a hash greatly reduces the access/search time, it is a calculated overhead and must be used each time an item in the list is needed.

An additional challenge comes when it is time for the kernel to clean up a structure that is no longer needed. If the individual elements of the list have been simply linked by a single forward-linkage pointer, then the task of removing a single element from the list can be time consuming. The list element, which points to the element to be removed, must be identified in order to repair the break in the chain that the removal will cause. These requirements lead to the development of bidirectional linkage schemes, which allow for quicker deletion but require additional overhead during setup and maintenance.

Pros

The main attraction to the dynamic list is that the resources consumed by the list are only allocated as they are needed. If the need arises for additional list elements, they are simply allocated on the fly, and a kernel rebuild and reboot are not needed. In addition, when a list element is no longer needed, its space may be returned to the kernel pool of available memory. This could reduce the overall size of the kernel, which may positively affect performance on a system with tight memory size constraints.

Cons

Nothing is free when it comes to programming code! The convenience of dynamic lists comes with several associated costs. The kernel must have an efficient way to allocate and reclaim memory resources of varying sizes (different list structures have different element size requirements).

The challenge of how to link individual list elements together increases the complexity and size of each data structure in the list (more choices to be evaluated by the kernel designer!). The dynamic list creates additional challenges in the realm of searching algorithms and indexing.

Summary

The current movement is clearly toward a totally dynamic kernel, which necessitates incorporation of an ever-increasing number and variety of dynamic lists. The challenge for the modern kernel designer is to help perfect the use and maintenance of dynamic lists. There is ample opportunity here to think outside the box and create unique solutions to the indexing and linkage challenges.

Resource Allocation

An early challenge for a kernel designer is to track the usage of a system resource. The resource may be memory, disk space, or available kernel data structures themselves. Any item that may be used and reused throughout the operational life cycle of the system must to be tracked by the kernel.

Bit Allocation Maps

A bitmap is perhaps one of the simplest means of keeping track of resource usage. In a bitmap, each bit represents a fixed unit of the managed resource, and the state of the bit tracks its current availability.

A resource must be quantified as a fixed unit size, and the logic of the bitmap must be defined (does a 0 bit indicate an available resource or a used resource?). Once these ground rules have been determined, the map may be populated and maintained.

Example

In practices a resource bit map requires relatively low maintenance overhead. The actual size of the map will vary according to the number of resource units being mapped. As the unit size of the resource increases, the map becomes proportionally smaller, and vice versa. The size of the map comes into play when it is being searched: the larger the map, the longer the search may take. Let's assume that we have reserved a contiguous 32-KB block of kernel memory and we want to store data there in 32-byte structures. It becomes a fairly simple issue to allocate a 1024-bit bitmap structure (128 bytes) to track our resource's utilization. When you need to find an available storage location, you perform a sequential search of the bitmap until you find an available bit, set the bit to indicate that the space is now used, and translate its relative position to indicate the available 32-byte area in the memory block.

Pros

The relative simplicity of the of the bitmap approach makes it an attractive first-pass solution in many instances. A small map may be used to track a relatively large resource. Most processors feature assembly language–level bit-test, bit-set, and bit-clear functions that facilitate the manipulation of bitmaps.

Cons

As the size of the bitmap increases, the time spent locating an available resource also increases. If there is a need for sequential units from the mapped space, the allocation algorithms become much more complex. A resource map is a programmatic agreement and is not a resource lock by any means. A renegade section of kernel code, which ignores the bitmapping protocol, could easily compromise the integrity of the bitmap and the resource it manages.

Summary

If a system resource is of a static size and always utilized as a fixed-sized unit, then a bitmap may prove to be the most cost-effective management method.

Resource Maps

Another type of fixed resource mapping involves the utilization of a structure known as a *resource map* (see Figure 3-4). The following is a generic explanation of the approach as there are many differing applications of this technique. In the case of a resource map, you have a resource of a fixed size against which individual allocations of varying sizes need to be made.

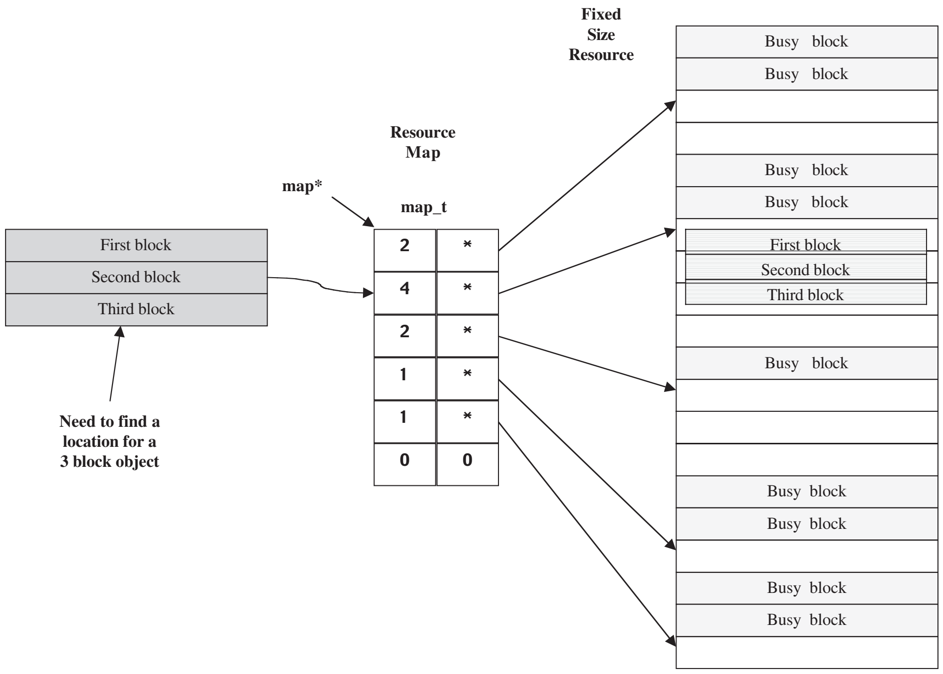

Figure 3-4 Resource Maps

Example

For our example, let's consider a simple message board. The message board has 20 available lines for message display; each line has room for 20 characters. The total resource has room for 400 characters, but individual messages must be displayed on sequential lines. Consider posting the two following messages:

House broken

Beagle puppy

Free to good home

12-year-old boy

Seeks lawns to mow

If the lines of each message were not assigned sequential space on the message board, you could end up with the following mess!

Beagle puppy

Seeks lawns to mow

House broken

12-year-old boy

Free to good home

To avoid such a situation, a resource map could be employed to allocate sequential lines. Each entry in the resource map would point to the next block of available line(s) on the board.

If the message board were blank, then there would be only one entry in our resource map pointing to the first line and stating that 20 lines were sequentially available. To list the first message, we would allocate the first three lines from the board, adjust our resource map entry to point to the fourth line, and adjust the count to 17. To add the second message to the board, we would allocate two more lines and adjust the first entry in the map to point to the sixth line, with the count adjusted to 15.

In effect a resource map points to the unused "holes" in the resource. The size of the resource block tracked by each map entry varies according to usage patterns.

Pros

A resource map requires relatively few actual entries to manage a large number of resources. If the allocation block size varies and needs to be contiguously assigned, then this may be your best bet.

Cons

Map entries are constantly being inserted and deleted from the maps. This requires constant shifting of the individual map entries (the saving grace here is that there are relatively few entries). Another concern is the size of the resource map itself: if you run out of entries, then freed resources may not be accounted for and in effect will be lost (a type of memory leak) to system usage until a reboot.

Summary

Resource maps have long been utilized by System V Interprocess communication kernel services, and if care is taken in their sizing, they are very efficient.

Searching Lists and Arrays

Where there are arrays and lists of data structures, there is always a need to search for specific elements. In many cases, one data structure may have a simple pointer to related structures, but there are times when all you know is an attribute or attributes of a desired structure and not an actual address.

Hashtables: An Alternative to Linear Searches

Searching a long list item by item (often called a sequential or linear search) can be very time consuming. To reduce the latency of such searches, *hash* lists are created. Hashes are a type of indexing and may be used with either static arrays or linked lists to speed up the location of a specific element in the list.

To use a hash, a known attribute of the item being searched for is used to calculate an offset into the hashtable (hash arrays are frequently sized to a power of two to assist in the calculation and truncation of the hashed value to match the array size). The hashtable will contain a pointer to a member of the list that matches the hashed attribute. This entry may or may not be the actual entry you are looking for. If it is the item you seek, then your search is over; if not, then there may be a forward *hash-chain* pointer to another member of the list that shares the same hash attribute (if one exists). In this manner, you will have to follow the hash-chain pointers until you find the correct entry. While you will still have to perform a search of one or more linked items, the length of your search will be abbreviated.

The efficiency depends on how evenly distributed the attribute used for the hash algorithm is within the members of the list. Another key factor is the overall size of the hashtable.

Example

Suppose you have a list of your friends' names and phone numbers. As you add names and numbers to the list, they are simply placed in an available storage slot and not stored in any particular order. Traditionally, you might sort the list alphabetically each time an entry is made, but this would require "reordering the deck." Consider instead using a hashtable.

As each entry is made, the number of letters in each name is counted; if there are more than nine, then only the last digit of the count is kept. A separate *hashtable* array with 10 entries is also maintained with a pointer to a name in the list with that *hash count*. As each name is added to the list, it is linked to all the other list members with the same hash count. See Figure 3-5.

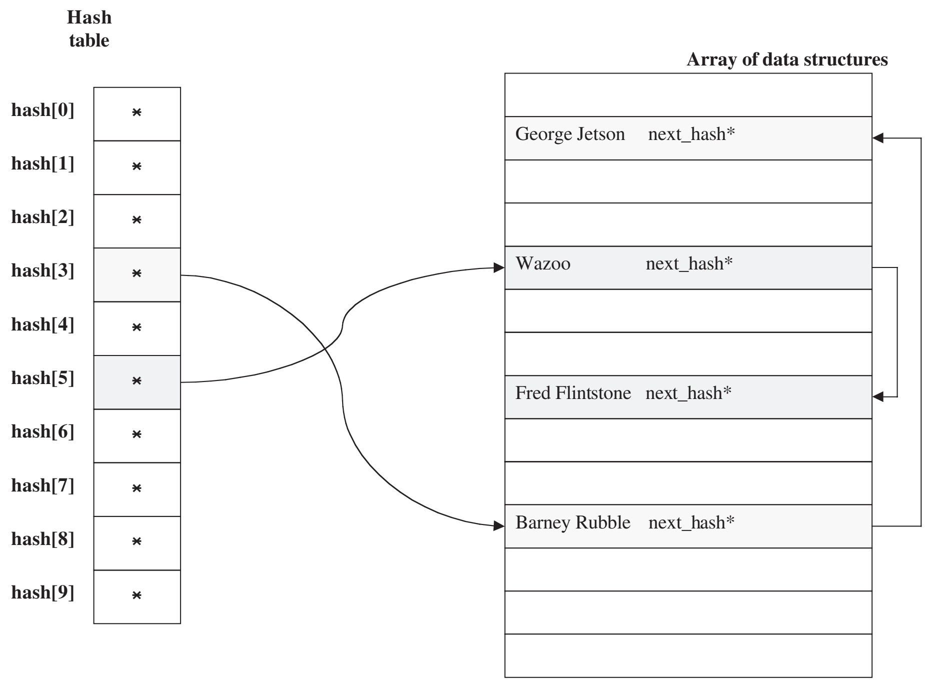

Figure 3-5 Hashtables and Chains

If your list had 100 names in it and your were looking for *Fred Flintstone*, the system would first total the character count (*Fred* has 4 and *Flintstone* has 10, a total of 15 counting the space between the names), which would send us to `hash[5]`. This entry would point to a name whose hash count is 5; if this were not the *Fred Flintstone* entry, you would simply follow the embedded hash-chain pointer until you found Fred's entry (or reached the end of the list and failed the search).

If there were 100 entries in the table and 10 entries in the hashtable, using a standard distribution, then each hash chain would have 10 entries. On average, you would have to follow an individual chain for half of its length to get the data you wanted. That would be a five-linkage pointer search in our example. If we had to perform a sequential search on the unordered data, the average search length would have been 50 elements! Even considering the time required to perform the hash calculation, this could result in considerable savings.

While this example is greatly simplified, it does demonstrate the basics of hash-headers and hash chains to speed up the location of the "right" data structure in a randomly allocated data table.

Pros

Hashing algorithms offer a versatile indexing method that is not tied to basics such as numerical sequence or alphabetic order. Relevant attributes are used to calculate an offset into the hash-chain header array. The header points to a linked list of all items sharing the same hash attribute, thus reducing the overall search time required to locate a specific item in the list.

Cons

The specific attributes used for the hash may be somewhat abstract in concept and must be carefully considered to assure that they are not artificially influenced and do not result in uneven distributions to the individual chains. If the size of the resource pool being hashed grows, the length of the individual chains may become excessively long and the payback may be diminished.

While the basic concept of hashing is very simple, each implementation is based on specific attributes, some numeric, some character-based, and so on. This requires the programmer to carefully study the data sets and identify which attribute to use as a key for the hash. Frequently, the most obvious one may not be the most efficient one.

Summary

Hashing is here to stay (at least for the foreseeable future). Make your peace with the concept, as you will see various implementations throughout all areas of kernel code.

Binary Searches

When it comes to searching a fixed list for a value, there are many approaches. The brute-force method is to simply start with the first element and proceed in a linear manner through the entire list. In theory, if there were 1024 entries in the list, the average search time would be 512 tests (sometimes the item you are looking for would be at the front of the list and sometimes toward the end, so the average would be 1024/2 or 512).

Another method involves the use of a *binary search algorithm*. The decision branch employed by the algorithm is based on a binary-conditional test: the item being tested is either too high or too low. In order for a binary search to work, the data in the list must be ordered in an increasing or decreasing order. If the data is randomly distributed, another type of search algorithm should be used.

Consider a 1024-element list. We would start the search by testing the element in the middle of the list (element 512). Depending on the outcome of this test, we would then check either element 256 or 768. We would keep halving the remaining list index offset until we found the desired element.

Pros

Following this method, the worst-case search length for our theoretical 1024-element list would be 10! Compare this to 1024 for the brute-force linear search method.

Cons

While the reduction in the number of individual comparisons is impressive, the basic list elements must be ordered. The impact of this on list maintenance (adding items to or removing them from the list) should not be underestimated. An unordered list may be easily managed through the use of a simple *free-list* pointer and an embedded linkage pointer between all the unused elements of a list. If the list is ordered, then many of its members may need to be moved each time an item is added or removed from the list.

Summary

We have considered only a very basic form of binary search. Kernels employ many variations on this theme, each tuned to match the needs of a specific structure.

Partitioned Tables

Modern architectures present kernel designers with many challenges; one is the mapping of resources (both contiguous and noncontiguous). Consider the task of tracking the page-frames of physical memory on a system. If physical memory is contiguous, then a simple usage map may be created, one entry per pageframe, the page number would be the same as the index into the array.

On modern cell-oriented systems, there may be multiple memory controllers on separate busses. Often, the hardware design dictates that each bus be assigned a portion of the memory address space. This type of address allocation may result in "holes" in the physical memory map. The use of *partitioned tables* offers a method to efficiently map around these holes.

Example

Consider the greatly simplified example in Figure 3-6. In order to manage a resource of 16, items we could use a simple 16-element array (as shown on the left side of the figure). In this example, there is a hole in the resource allotment; physically, the 5th through 14th elements are missing. If we use the simple array method, 16 elements will still be needed in the array if we want to preserve the relationship between the array index and the corresponding address of the resource.

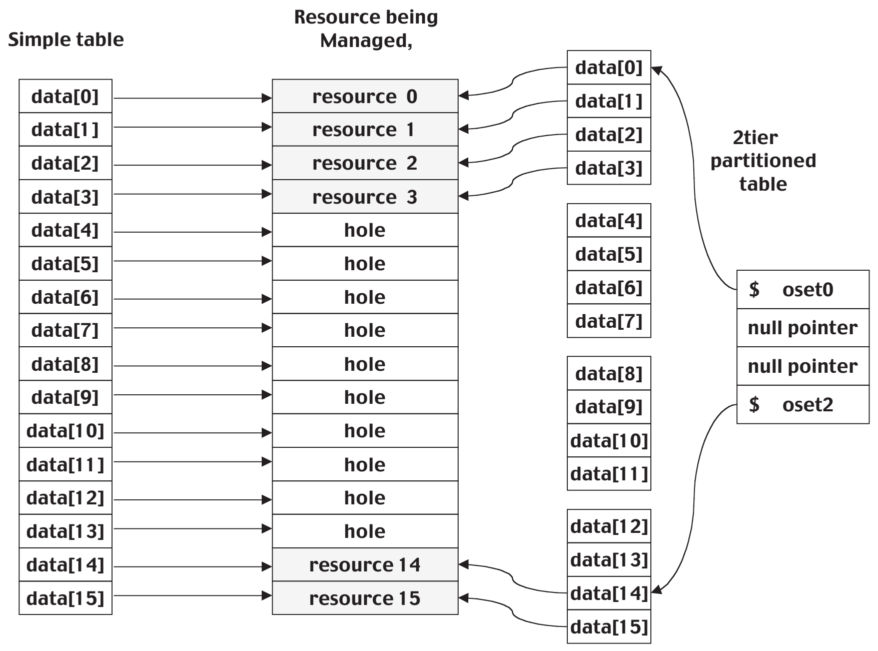

Figure 3-6 Partitioned Tables

By switching to a two-tier partitioned table, we can map the resources on both sides of the hole and reduce the amount of table space needed. The first tier is a simple array of four elements, each either a valid pointer to a block of data structures or a null pointer signifying that the associated block of resources does not exist.

In addition to the pointer, an offset value is stored in each element. This is used in the case where the hole extends partially into a block's range (as does the last pointer in our example). The offset allows us to skip to the element containing the first valid data structure.

Let's compare the effort required to locate a data structure. If you needed the information related to the 15th resource and were using the simple array approach, you would only have to index into the 15th element of the array (`data[14]`).

If the partitioned approach were being used, you would first divide the element address by the size of the second-tier structures. For our example that would be 14/4, which would yield 3 with a remainder of 2. You would then index into the first-tier array to the fourth element (index = 3), follow the pointer found there, and use the remainder to offset into the partitioned table to the third element (index = 2).

In our simplified example, the single array approach required room for 16 data structures even though there were only six resources being mapped. The partitioned approach required room for only eight data structures (in two partitioned tables of four elements each) plus the very simple four-element first-tier structure.

At first glance, it may not seem that the payback is worth the extra effort of navigating two tables, but this is a very simple example. As we mentioned earlier, the approach is used to manage tables large enough to map all the physical page-frames of a modern enterprise server! There can be millions of potential pages needing their own data structures (and large holes may exist). We will see partition tables in use when we discuss memory management.

Pros

The value of partitioned tables is in the reduction of kernel memory usage. The less memory used by the kernel, the more available for user programs!

Cons

The method actually has very few drawbacks; the referencing of a map element is now a two-step process. The map element number must be divided by the number of elements in each partitioned table structure (second-tier structure) to yield an index into the first-tier structure. The remainder from this operation is the offset into the second-tier structure for the desired element. In practice, the partitioned tables are often sized to a power of two, which reduces the calculation of the offsets to simple bit-shifting operations.

Summary

Partitioned tables are dictated by the architecture and are a necessary tool in the kernel designer's bag of tricks.

The B-Tree Algorithm

The *b-tree* is a somewhat advanced binary search mechanism that involves making a series of index structures arranged in a type of relational tree. Each structure is known as `bnode`; the number of `bnodes` depends on the number of elements being managed. The first `bnode` is pointed to by a `broot` structure, which defines the width and depth of the overall tree.

One of the interesting and powerful aspects of the b-tree is that it may be expanded on the fly to accommodate a change in the number of items being managed. B-trees may be used to map a small number of items or hundreds of thousands by simply adjusting the depth of the structure.

The simple `bnode` consists of an array of key-value pairs. The key data must be ordered in an ascending or descending manner. To locate a needed value, a linear search is performed on the keys. This may seem to be an old-fashioned approach, but let's consider what happens as the data set grows.

The first issue is the size of the `bnode`. A b-tree is said to be of a particular order. The order is the number of keys in the array structure (minus 1—we will explain this as we discuss a simple example). If we have a third-order b-tree, then at most we would have three keys to check for each search. Of course, we could only reference three values with this simple structure!

In order to grow the scope of our b-tree's usefulness, we have to grow the depth of the tree. Once a b-tree expands beyond its order, additional `bnodes` are created and the depth of the tree is increased.

Only `bnodes` at the lowest level of the tree contain key-value data. The `bnodes` at all other levels contain key-pointer data. This means that in order to find a specific value, we must conduct a search of a `bnode` at each level of the tree. Each search, on average, requires half as many compare operations as there are keys in the `bnode` (the order). This means that the average search length is defined as (order/2) × depth. Optimization is achieved by adjusting both the order and the depth of the b-tree.

Example: Growing the b-tree

From Figure 3-7, consider a very simple example of a third-order b-tree. The original `bnode` has keys: 1, 2, 3. Everything fits in a single `bnode`, so the depth is simply 1.

When we add a fourth key, 4, to the tree, it fills the `bnode` to capacity and causes the growth of the tree. If the number of keys in a `bnode` exceeds the order, then it is time to split the node and grow the tree.

To grow this simple tree, we create two new `bnode` structures and move half of the existing key–value pairs to each. Notice that the data is packed into the first two entries of each of the new `bnodes`, which allows for room to grow.

Next, we must adjust the depth value in the `broot` data structure that defines the tree. The original `bnode` must also be reconfigured. First, it is cleared, and a single key is created.

Let's take a closer look at the `bnode` structure. Notice that there are actually more value slots than there are key slots. This allows for two pointers to be created in relation to each key entry. The pointer down and to the left is used if you are seeking a key of a lower value. The pointer down and to the right is used if you are looking for one that is greater than or equal to the key.

Let's try locating a value given a key = 2:

Follow the `broot` pointer to the first-level `bnode`.

Search for a key = 2. Because there is no perfect match, we look for a key that is > 2 and follow the pointer down and to the left of that key to the next `bnode`.

Search for a key = 2. A match here will yield the appropriate value. We know that this is a value and not another pointer, since the `broot` structure told us we had a depth of two!

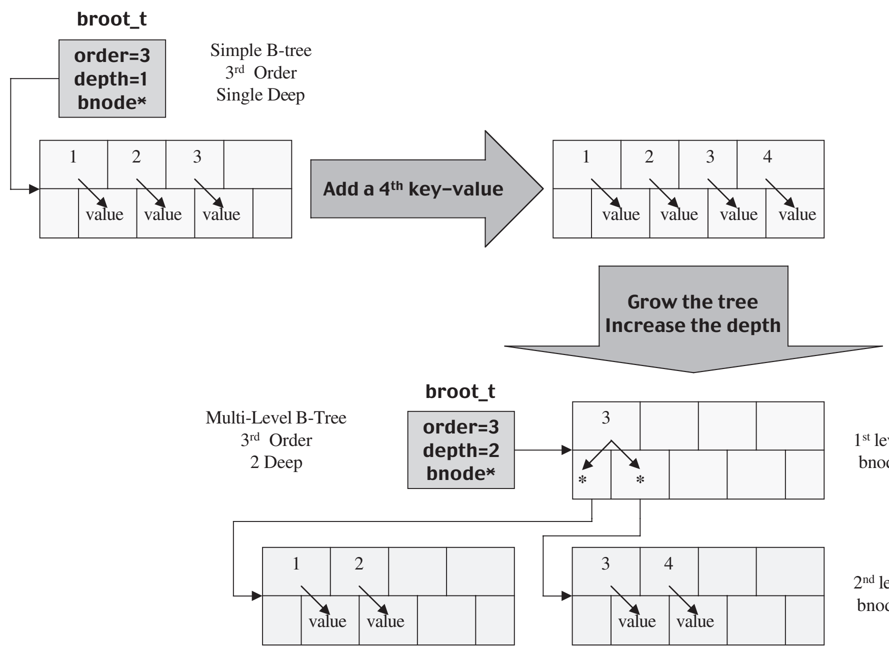

Figure 3-7 B-Trees

Note that the key values do not have to be sequential and may be sparse as long as they are ordered. Searches on the lowest level of the tree return either a value or a "no match" message.

The search for a key always attempts to find a perfect match yielding either a value or a pointer to the next lower level. If no match is found and we are not on the lowest level, the following logic is used. If your search key lays between two adjacent key entries, the pointer to the next level lies below and between them. A search key less than the first key entry uses the first pointer in the `bnode`. If your search key is larger than the last valid key in the `bnode`, then the last valid pointer is used.

Pros

B-trees may be grown dynamically, while their table maintenance is modest. This makes them ideal for managing kernel structures that vary in size. Key values may be packed or sparse and added to the table at any time, providing for a flexible scope.

Cons

Another benefit is that given a sufficiently sized order, a b-tree may grow to manage a large number of items while maintaining a fairly consistent search time. Consider a 15th-order b-tree: the first depth would map 16 items, the second depth would map 256 items, and the third depth would yield a scope of 4096 while only requiring the search of three `bnode`s! This type of exponential growth makes it very popular for management of small to large resources.

The b-tree, binary-tree, and balanced-tree belong to a family of related search structures. While the b-tree has a modest maintenance overhead due to its simple top-down pointer logic, its growth algorithm may result in sparsely populated `bnodes`. This increases the number of nodes required to map a given number of data values. As with most approaches, we trade table size for maintenance cost.

Another issue is that while the b-tree may grow its depth to keep up with demand, it may not change its order (without basically cutting it down to the ground and rebuilding it from scratch). This means that designers need to pay attention to its potential usage when sizing the order in their implementations.

Summary

The b-tree requires a bit of study prior to its implementation but offers an effective method for the mapping of ordered dynamic lists ranging in size from slight to huge. We will see a practical application of the b-tree when we examine kernel management of virtual memory region structures.

Sparse Tables

We discussed the use of a hash to speed access to members of a static, unordered array. What would happen if the hash size were aggressively increased (even to the size of the array it referenced or larger)? At first you might think this would be a great solution: simply create a hashing algorithm with enough scope, and lookups become a single-step process. The problem is that the kernel data array and its corresponding hash could become prohibitively large in order to guarantee a unique reference.

A compromise is to size the data structure large enough to hold your worst-case element count and hope that the hashing algorithm is fully distributive in its nature. In an ideal situation, no two active elements would share the same hash.

In the less-than-ideal real world, there will be cases where two data elements do share a common hash. We may solve the problem by dynamically allocating an additional data structure outside the fixed array and creating a forward hash-chain link to it.

Usually, this step is not necessary if the hash formula is sufficiently distributive. In practice, a forward pointer may only be needed in a very small percentage of the cases (less than 1% or 2%). In the very rare case where a third element must share the same hash, an additional structure would be chained to the second, one and so on (reference Figure 3-8).

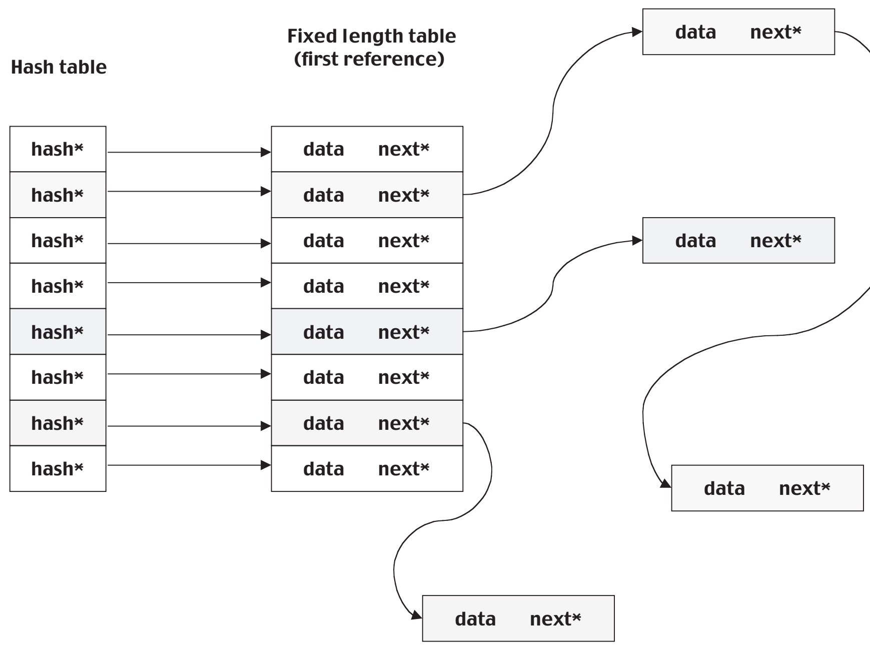

Figure 3-8 Sparse Tables

Pros

Sparse lists greatly reduce the average search time to locate items in unordered tables or lists.

Cons

Sparse lists require the kernel to manage the available sparse data-element locations as yet another kernel resource. As there is a possibility that the data element first pointed to may not be the actual one you are searching for, the target structure must contain enough data to validate that it is or is not the one you want. If it isn't, a routine needs to be developed to "walk the chain."

Summary

Spare lists work best when there is some known attribute(s) of the desired data set that may be used to generate a sufficiently large and distributive hash value. The odds of needing to create a forward chain pointer decrease greatly as the scope of the hash increases. We will see an example of this approach in the HP-UX kernel's virtual-to-physical page-frame mapping. In actual use, it is a one-in-a-million chance to find a hash-chain with more than three linked elements!

The Skip List

In the world of search algorithms, the skip list is a new kid on the block. Its use was first outlined in the 1990s in a paper prepared for the Communications of the Association for Computing Machinery (CACM) by William Pugh of the University of Maryland. For additional information, visit *ftp://ftp.cs.umd.edu/pub/skipLists/skiplists.ps.Z.*

The algorithm may be employed to reduce search times for dynamic linked lists. The individual list elements must be assigned to the list according to some ordered attribute. This approach works well for linked lists with only a dozen or so members and equally as well for lists of several hundred members.

At first glance, skip lists appear to be simply a series of forward- and reverse-linkage pointers. Upon closer examination, we see that some point directly to a neighbor, while others skip several in-between structures. The surprising part is that the number of elements skipped is the result of random assignment of the pointer structures to each list member as it is linked into the list.

List maintenance is fairly simple. To add a new member element, we simply skip through the list until we find its neighboring members. We then simply link the new structure between them. The removal of a member follows the same logic in reverse.

When a new member is added, we must decide at which levels it will be linked. The implementation used in the HP-UX **`pregion`** lists uses a skip pointer array with four elements. All members have a first-level forward pointer. The odds are one in four that it will have a second-level pointer, one in four of these will have a third-level pointer, and one in four of these will have a fourth-level pointer. As elements may be added or removed from the list at any time, the actual distribution of the pointers takes on a pseudorandom nature.

To facilitate the method, a symbolic first element is created, which always contains a forward pointer at each of the skip levels. It also stores a pointer to the highest active level for the list. See Figure 3-9.

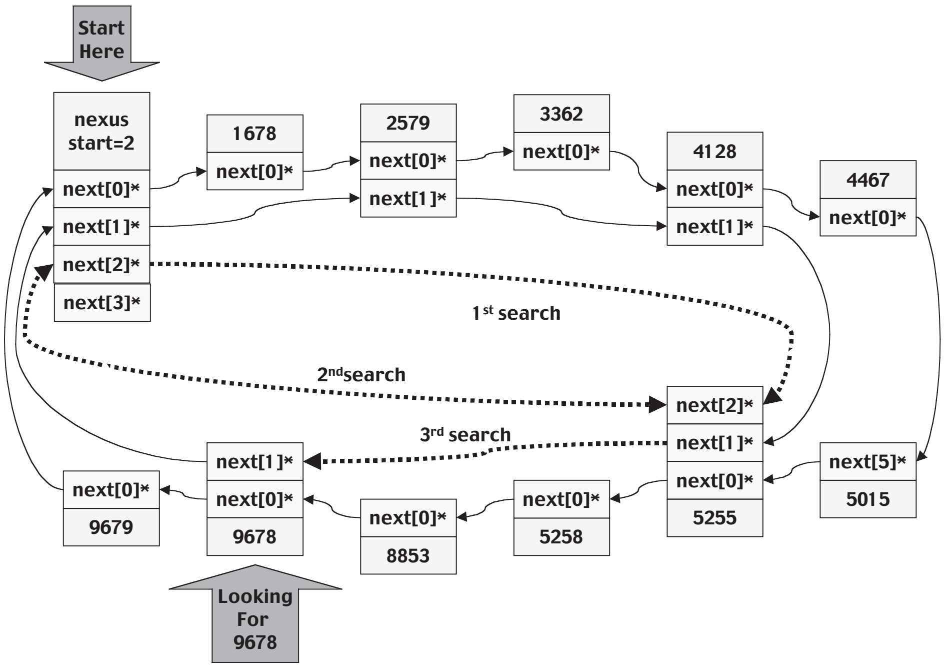

Figure 3-9 Skip List

Example

From Figure 3-9, let's assume that we need to locate the structure with the attribute 9678. In the list nexus structure, we see that the highest level active pointer is at `next[2]`, so we follow it. This structure has an attribute value of 5255, so we need to continue our search at this level.

We arrive back at the starting point structure, so we backtrack to the 5255 structure, drop down a level to `next[1]`, and continue.

We now arrive at the structure with the 9678 attribute—it's a match! Our search is over.

In the example, it took only three searches. A simple binary search would have taken four searches.

Pros

The skip list offers an interesting approach for searching that often results in a reduction of search times when compared to a simple binary method. Adding and removing members to and from the list is reasonably quick.

Cons

It requires the creation of a multielement array for the forward linkages. The random nature of the pointer assignment does not take into account the relative size or frequency of use of the various list elements. A frequently referenced structure might be inefficiently mapped by the luck-of-the-draw (in our example we beat the binary method, but other members of our list would not: try searching for the 5015 structure).

Summary

Despite the random nature of this beast, the overall effect may be a net-sum gain if the ratio between the number of items and the number of levels is carefully tuned.

Operations Arrays

Modern kernels are often required to adapt to a variety of different subsystems that may provide competing or alternate approaches to the same management task. A case in point is that of a kernel that needs to support multiple file system implementations at the same time.

To accomplish this goal, specific file systems may be represented in the kernel by a virtual object. Virtual representation masks all references to the actual object. This is all well and good, but what if kernel routines needed to interact with the real object required code and supporting data dependent upon type-specific attributes? An operations array, or vectored jump table, may be of service here.

Example

Consider Figure 3-10. Here we see a simple kernel table with four elements, each representing a member of a virtual list. Each list member has its actual `v_type` registered, a type-specific `v_data[]` array, and a pointer to a `v_ops[]` operational array.

For this model to work, the number of entries in the operational array and the functions they point to must be matched for each type the kernel is expected to handle. In our example, there are four operational target functions: `open()`, `close()`, `read()`, and `write()`. Currently, our system has only two variations labeled type: `X` and `Y`.

When a routine is called through a vectored jump referenced through `v_ops[x]`, it is passed the address of the virtual objects `v_data[]` array. This allows the final type-specific function to work with a data set type that it understands.

The end result is that all other kernel objects need only to request a call to `v_ops[0]` to instigate an `open()` of the virtual object without concern or knowledge of whether it is of type `X` or `Y`. The operations array will handle the redirection of the call. In practice, we will see many examples of this type of structure in the kernel.

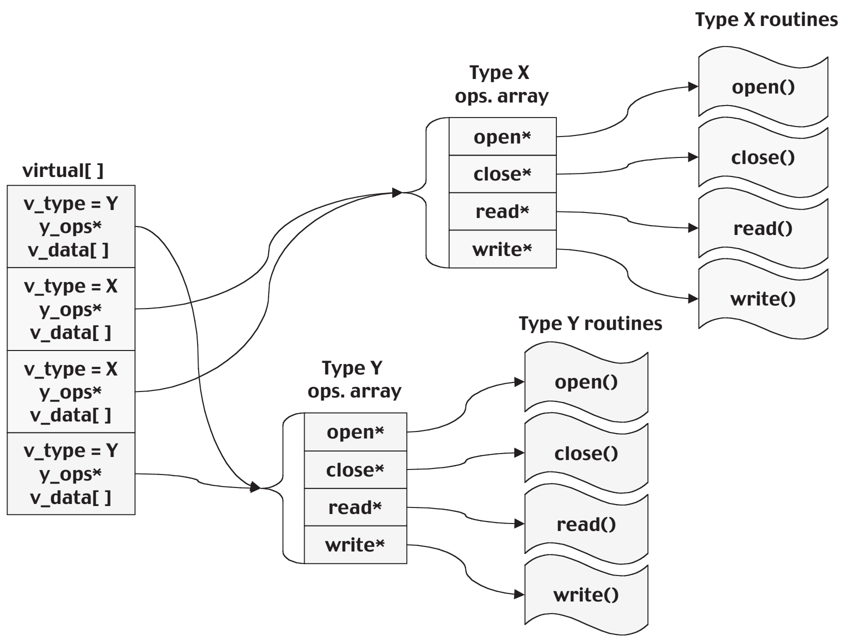

Figure 3-10 Operations Arrays: A Vectored Jump Table

Pros

The cost of redirecting a procedure call through a vector jump table is very low and for the most part transparent to all that use it.

Cons

In debugging, this is yet one more level of indirection to contend with.

Summary

The vectored jump table, or operational array, provides a very effective abstraction layer between type-specific, kernel-resident functions, and other kernel subsystems.

The HP-UX Kernel Overview

Now that we have spent some time considering a generic UNIX kernel, the tools of the trade, and some of the challenges faced by the kernel designers, let's turn our attention to the specifics of the HP-UX kernel.

The current release of the Hewlett-Packard HP-UX Operating System is HP-UX 11.i (the actual revision number is 11.11). We concentrate on the current release, but as many production systems are still running HP-UX 10.20 and HP-UX 11.0, where appropriate we try to cover material relevant to these releases as well.

The HP-UX kernel is a collection of subsystems, drivers, kernel data structures, and services that has been developed and modified for the past 20 years. This legacy has yielded the kernel we present in this book. Over the years, virtually no part of the kernel has gone undisturbed: the engineers and programmers at HP have shown an unwavering commitment to the continuous process-improvement cycle that defines the HP-UX kernel. The authors of this book tip our collective hat to their continuing efforts and vision.

In its current incarnation HP-UX runs primarily on systems built on the Hewlett-Packard Precision Architecture processor family. This was not always the case. Early versions ran on workstations designed on the Motorola 68xxx family of processors. As in the past when HP-UX was ported to the HP-PA RISC chip set, today we are on the threshold of another port of this operating system to an emerging new platform: the Intel IA-64 processor family. In this book, we concentrate on the HP PA-RISC implementation.

Fundamental Kernel Data Structures: A First Pass

HP-UX kernel data structures are a key focus of our discussion. We break them down into several key areas: kernel memory tables, process tables, disk space tables, file system tables, and input/output tables. As we explore these various areas, keep your eyes open for similarities in approach and design. Many teams of programmers work on the various modules and subsystems that make up the kernel. They frequently borrow methods and algorithms from one another, and there seems to be a never-ending attempt to tweak and tune them for improved performance. This type of crosspollination helps the kernel mature and improve.

Kernel Memory Tables

A prime concern of the kernel is the management of the system memory resources (see Figure 3-11). Memory comes in many flavors and types: physical, virtual, and logical.

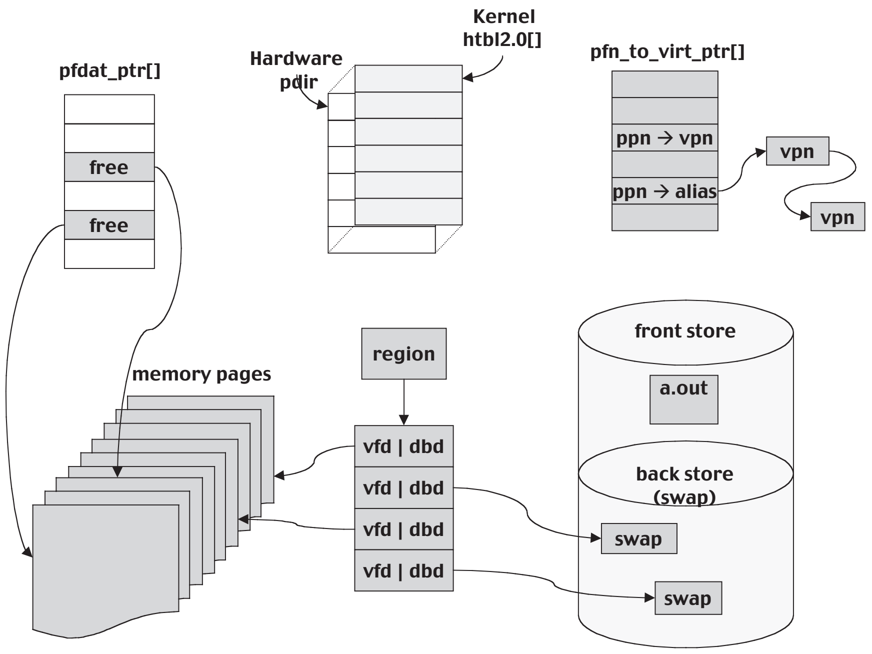

Figure 3-11 Kernel Memory Tables

Physical Memory

HP-UX runs on processors that have a 32-bit instruction word size. The primary memory allocation size is called a physical pageframe. On current HP-UX systems, a pageframe is 4096 bytes (or 1024 words); this has been constant for many years. While this reduces the number of on-demand page-in operations required for a process and its threads, it creates challenges for the memory management schemes. We explore this fully in Chapter 6.

Several primary data structures are required to track and map the system's physical memory. The `pfdat_ptr[x]` array is commonly called the *free-page table* and is used to keep track of which pages are currently in use by the kernel and which have been assigned to a process. This table is a partitioned table to allow for the mapping of physical memory around holes in the physical memory map. As a general rule, if a table name ends in `_ptr`, it is most likely a partitioned table.

With the release of HP-UX 11.i, a process may be assigned larger contiguous sets of physical page-frames under a newly introduced Variable Page Size (VPS) feature. This is also called Performance-Optimized Page size (POPs) in some sales and training literature. To accommodate these features, the `pfdat_ptr` table has been modified to allow the pooling of contiguous free pages into larger views, ponds, and pools of various colors.

Virtual Address Space

The VAS does not reference a physical system entity; instead it is the conceptual memory space onto which the underlying hardware platform (HP PA-RISC) and the kernel must map all potential regions of use. This phantom map is a key concept to master as we study the kernel's theory of operation.

The kernel memory management structures must allow the hardware to map virtual page-frames to the physical page-frames that contain current process code or data. The primary data structure for this task is the `htbl2.0[`*x*`]`. If the needed page-frame is not currently memory-resident, then it is up to the kernel to handle the resulting page fault and get it loaded as soon as possible.

The HP-PA RISC hardware as well as the HP-UX kernel requires this virtual-to-physical page-frame map. The hardware calls this table the page directory (or `pdir[`*x*`]`) and uses its entries, defined as page data entries (or `pdes`), to update the CPU translation lookaside buffer (TLB). The hardware and kernel names are different to illustrate that the hardware does not specify the use of all the various bits in this structure: the kernel designers use the undefined bits for their own purposes.

> **N O T E** On the older 32-bit HP PA-RISC–based systems (called narrow systems), this table is named **`htbl[`*x*`]`**.

The `htbl2.0[`*x*`]` only provides for the mapping of virtual pageframes to physical page-frames. While this is the direction of translation most frequently needed by kernel functions, occasionally there is a requirement to identify which virtual pageframe has been assigned to a particular physical pageframe. This requirement is fulfilled by the `pfn_to_virt_ptr[`*x*`]` table. In addition to this basic feature, it is also used to link `alias` data structures if they are required. An `alias` is used if more than one virtual pageframe has been mapped to a single physical page-frame, an important feature allowing copy-on-write semantics during the `fork()` system call.

Process Logical Memory Space

As a matter of concept, we need to consider a process's view of memory. Linking-loaders create executable image files (for C, the common name is *`a.out`*). These files and their headers contain information about which system resources will be needed for the program to run. For a program to run, its page images must be loaded into consecutive pages in the VAS. This is because when the image was created, all references to data and procedure calls were coded as absolute addresses within the process's logical address space.

To facilitate the sharing of process code, dynamic shared library code, shared memory-mapped files, and other shared objects and related consecutive pages in the program's image are said to occupy *regions* of address space. The mapping of these logical process address regions to kernel-managed virtual memory regions is the job of the kernel's many `region` data structures.

The `region` structure contains a database with a page-by-page description that indicates if a page is currently in physical memory, stored as an image on a front-store (an executable program file), stored as an image on a back-store (a swap page), or still awaiting initialization (used for uninitialized data pages, BSS).

Managing Memory for Internal Kernel Usage

So far, we have discussed only the structures used for managing memory for use by the system's many processes. This type of memory management is done at the granularity of the page-frame. Additional structures are used inside the kernel for the allocation of smaller sized blocks of memory to be used by the kernel's many dynamic tables and linked lists. Until the 11.i release, HP-UX utilized the rather classic "kernel bucket" memory allocation scheme. This has been replaced with an "arena" allocation approach. This change was made to improve flexibility, reduce waste, and facilitate page reclamation.

All Together Now

It may seem at first glance that many structures are playing in the same sandbox. To some degree, this is an accurate assessment, and for it to work, all of the tables must play nice together! Each table has been optimized to provide support for a particular piece of the puzzle and must be meticulously managed to avoid system corruption. There are many levels of checks and balances used to maintain the memory management system's integrity.

Kernel Process Tables

The process is truly the king of the HP-UX operating system, the primary entity managed by the kernel. In general, the prime directive for the HP-UX kernel is to level the playing field and make sure every process thread gets a chance to run.

The Process Table

Threads are the schedulable entity, but it is the venerable process that owns resources (at least as long as it is active). As such, the process table (see Figure 3-12) is the starting point for all process and thread management-related activities in the kernel. HP-UX 11.i features a dynamic process table. Entries are created as needed by allocating memory from a kernel memory arena and adding it to the linked list of active processes. The beginning of the process list is pointed to by the kernel pointer `proc_list*`.

Prior to the HP-UX 11.i release, the process table was a fixed-length table initialized at system boot, pointed to by the kernel pointer `proc*`, and defined in size by the tunable kernel parameter `nproc`. The original `nproc` parameter has been maintained through the current release for a couple of reasons. There are several static kernel tables that are sized directly in proportion to the process table. While the process table is now dynamic, not all related structures have made this change. It is reasonable to assume that in future releases this may not continue to be an issue, but for now `nproc` needs to be declared. This parameter also serves as a maximum limit for the growth of a system's process table.

Process management starts by using the `phash[x]` to locate a specific entry in the process table. Once we have located the appropriate `proc` structure, we follow its pointers to other key kernel tables.

The `kthread` Table

`kthread` structures are also dynamically allocated from a kernel memory arena and linked onto a list of active `kthreads`. In addition, all `kthread` structures belonging to a single process are linked together and to the parent process `proc` structure. Such threads are commonly called siblings. The process table has a pointer to the first and last of its sibling threads and maintains a count of how many it has spawned.

Prior to HP-UX 11.i, the `kthread` table was also a static array allocated at system boot and sized by the kernel parameter `kthreadNTHREAD`.

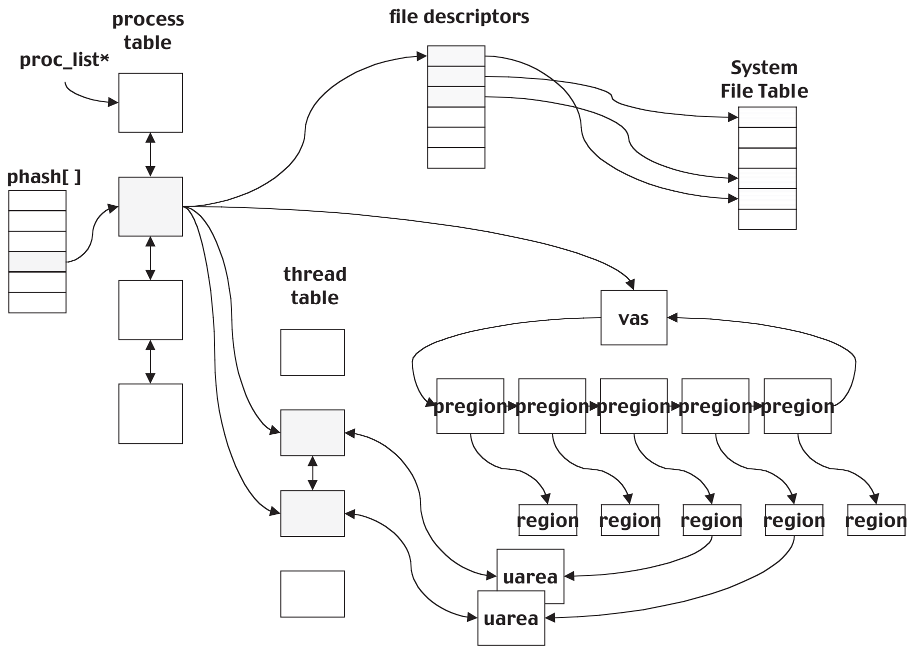

Figure 3-12 Kernel Process Tables

The `vas` and `pregion` Tables

In the grand scheme of things, a **`pregion`** contains the address offset and size of a process's logical region, access identifier information, and a type designator that specifies the usage mode. The `pregion` also maps a process's logical region into the system's VAS and creates a linked list of a process's virtual memory elements. When a process causes a page fault (requests access to a virtual page that is not currently memory-resident), the fault handler must search the process's linked list of `pregions` to determine which kernel region contains the necessary data for obtaining the faulting page. These structures and the underlying kernel region structures are the backbone of memory fault handling and have been optimized for function and efficiency.In addition to keeping track of its threads, a process table must also provide a map to each of the memory regions it will require for execution. To this end, a structure called a `vas` is created and linked to the `proc` of the process. The `vas` is the nexus of a linked list of `pregion` structures connected by a skip list. Each **`pregion`** links the process and its threads to a kernel-managed `region` structure. The **`pregion`**-to-**`region`** abstraction layer facilitates the mapping of shared objects by the kernel.

Process File Descriptors

All sibling threads share access to any file that has been opened in the name of its parent process. Each `open()` system call results in an entry in the process's file descriptor table. The size of this table is controlled by the kernel-tunable parameter `nfile` and prevents unlimited opens by a runaway thread.

All I/O is file I/O, and as such the file descriptor table represents the process's gateway to the "real world." File descriptors are required for each read and write operation. For example, consider the shell's default **`STDIN`**, **`STDOUT`**, and **`STDERR`** descriptors. As is the case with the shell, multiple file descriptors may point to the same file path, some for reading and some for writing.

The `uarea` Structure

An additional structure being introduced here is the `uarea`. The **`uarea`** is implemented as a kernel memory region, and unlike the other regions mapped in the parent process's `pregion` chain, access to the `uarea` is private to a single `kthread`. If there are multiple siblings, then there is an equal number of `uarea pregions` linked to the process's `vas`. An interesting note is that the `uarea` is mapped into a process's `vas` but never directly accessed by a user thread. The kernel has exclusive usage rights and stores the threads register image, known as a process control block (PCB) prior to a context switch-out. It then loads the next thread on the run queue's PCB from its `uarea` prior to switching it in.

The Kernel File System Tables

The kernel must maintain a complete list of all opened files on the system, active mount points, and what is mounted there. Performance requires that many file system structures be maintained in core-resident `inode` caches and buffer caches as well as in a variety of tables.

The System File Table

Each discrete, open system call results in an entry to the system file table (see Figure 3-13). If the same file is opened 20 times, there will be 20 separate entries in the table. Discrete entries are required because this table keeps track of the type of open (read or write), the current offset into the file, and the number of linkages to it. As the various processes are terminated, their file descriptors are closed, and the linkage count in the system file table is decremented. If the linkage count goes to zero, then the entry is placed on a free list and may be reused.

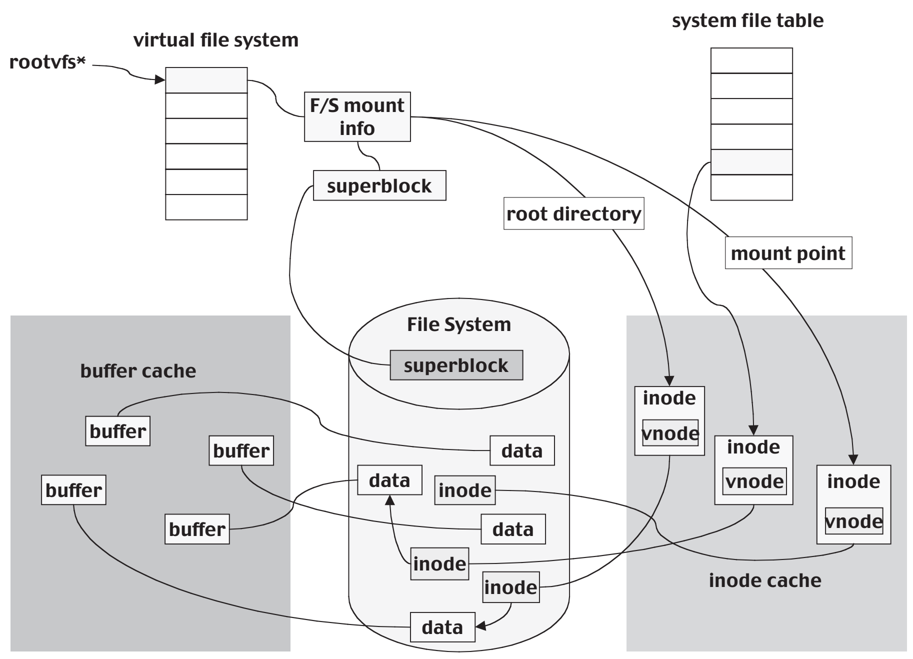

Figure 3-13 Kernel File System Tables

The Virtual File System

The HP-UX kernel supports access to several different types of file systems. On earlier versions of the operating system, the specifics of each supported file system type was crafted into the kernel's core image. Changing file system attributes was a major challenge and required patching the kernel. To move away from this dependency, a virtual file system interface was designed and implemented in the kernel.

The virtual file system treats all file system types the same. It is primarily concerned with their type, where they are mounted, a pointer to core-resident copies of any metadata that may be required to manage them, a cached copy of their respective root directories, and a pointer to an operational array of routines customized to handle their specific type.

The `Inode` Cache

The `inode` is the heart and soul of a UNIX file; it contains all attribute information with respect to a specific file. As the file is the basic object of all I/O operations, the `inode` attribute information is the key to access rights and data security within the system.

File attribute information is needed each time a thread requests access to file data or to modify the `inode` data itself. To speed this operation, an in-core copy is maintained of the `inode` data for each open file on the system. This is known as an `inode` cache.

As each file system type may define different `inode` data structures (different sizes, different block location schemes, different immediate data storage methods), it is the job of each configured file system type to define and build its own `inode` cache. To mask this difference from the higher levels of the kernel, an abstraction layer is added in which each file has assigned to it a systemwide unique virtual node (or `vnode`). Actions aimed at the `vnode` are translated through an operations array to file system type-specific routines in the kernel.

The Buffer Cache

Just as we keep copies of file attributes in an `inode` cache, the system maintains and manages a memory-resident buffer cache to hold recently requested copies of file data. When a process requests a read or write of file data, the buffer cache is checked first to see if a copy is present. If a cached copy is present, then the system merely needs to perform a memory-to-memory transfer of the requested buffer to or from the program's data space. Memory-to-memory transfers of this nature are called *logical reads* or *logical writes*. If a requested buffer is not present or the buffer is filled, a transfer must take place between the buffer cache and the physical disk. This constitutes a *physical read* or *physical write*.

When a read or write request results in an immediate physical action, it is said to be a *buffer cache miss*. The ratio between logical and physical reads and writes is called the *hit rate* and may be viewed using tools such as HP's `glance` or `gpm`.

The Kernel Input/Output Tables

The HP-UX kernel allows no direct access to physical devices. All I/O must be requested through the proper channel: the system call interface. A key component of this model is the kernel's extensive set of drivers, device maps, and the overall system's general input/output (GIO) system.

A large portion of the kernel memory space is occupied by these various components. The modern HP-UX kernel provides for the dynamic mapping of some of these should it be required.

The I/O Tree

A key table is appropriately named the I/O tree. This tree structure is actually a collection of linked objects used to identify and map system hardware devices to specific software drivers. The tree's roots stretch back to the system initialization, occurring even before the HP-UX kernel is loaded into memory. A subsection of this structure is the more elemental kernel I/O tree and is created in memory visible to the processor-dependent code (sometimes called the boot-rom) to be used in the event of a system crash.

The individual nodes of the I/O tree contain basic information about the hardware address: its type, which software has claimed it (driver association), and which context it belongs to. Additional properties, such as a kernel-assigned instance number and pointers to its parent, sibling, and children, are also present.

The I/O tree follows an object model; nodes lower down in the tree structure inherit attributes from their parents and pass them on to their children. We explore this fully in Chapter 10, "I/O and Device Management."

System Interruptions

In addition to the I/O tree, the I/O subsystem is responsible for the detection and response to interruptions. You may notice that we use the word *interruption*, not *interrupts*: HP-UX describes all categories of events affecting normal processor execution as being system interruptions. They are broken into four categories: faults, traps, interrupts, and checks.

To further complicate the topic, there are internal (CPU hardware, clocks, timers, etc.) and external (I/O devices, interface cards, physical device) interrupts, not to mention process signal handling!

As shown in Figure 3-14, internal interruptions are registered by a CPU and cause a vectored branch to memory-resident handling routines contained in an array called the interrupt vector table (IVT). Each entry in this table consists of a 32-byte block of code in which the initial portion of the interruption handler must be loaded. Also note that control register `CR14` contains the interrupt vector address (IVA), the starting address of the IVT. As each physical processor maintains its own register set, it would be possible to have interruption handlers specific to each processor. This is not generally the case, but it is an interesting option.

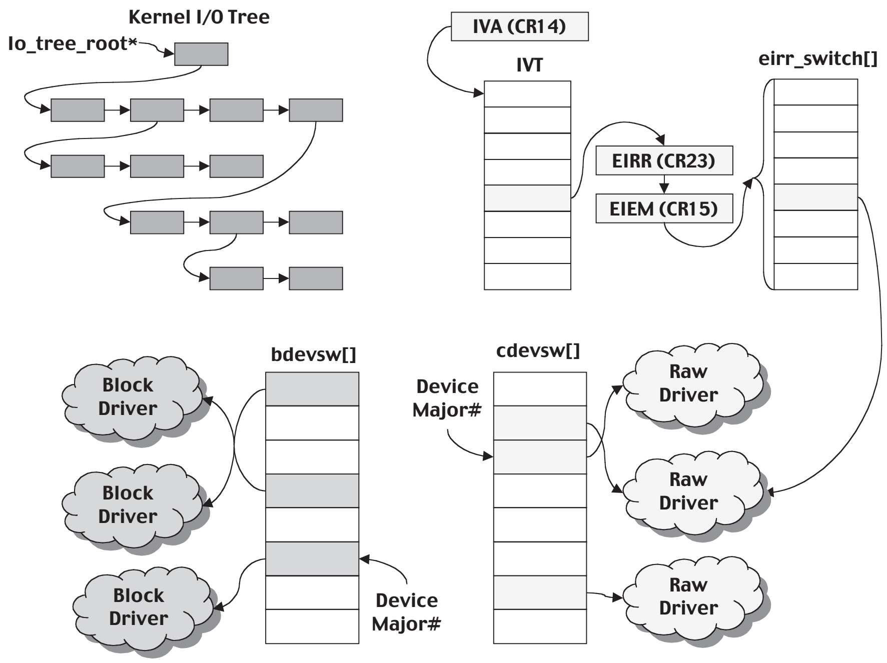

Figure 3-14 Kernel I/O Tables

An interesting interruption (registered as internal interrupt #4) is called the *external interrupt* and evokes routines vectored through an array named the *external interrupt receipt register switch table*, or simply the `eirr_switch[x]`. In most cases, this vectored switch table points toward the interrupt handling routine entry points associated with the kernel's various hardware drivers. Two additional control registers assist in registering and enabling the various external interrupts: `CR23` (the `eirr`) and `CR15` (the external interrupt enabling mask, or `eiem)`.

Drivers and Switch Tables

The bottom portion of Figure 3-14 shows two switch tables. One is used for accessing block device drivers and the other for raw device drivers. A raw device is one to which data is transferred in byte streams of varying sizes.

A raw device may receive a single byte or many megabytes in a single transfer (or even gigabytes in today's computing climate). Raw device access is also used to talk to and configure device controllers and interface cards when necessary.

Block devices hold mountable file systems. All I/O to a block device is directed through the system's buffer cache to reduce physical requests to logical requests whenever possible for speed and convenience

As drivers are built into the kernel image or registered for future dynamic loading, they are assigned a driver number, often called their *major* number. Many of the major numbers are reserved and always assigned to a specific driver; for example, the logical volume manager pseudodriver is always assigned major number 64, while others are assigned from a dynamic pool of available numbers. This major number is visible through the `ll` or `ls -l` commands. To list all the currently used major numbers for an HP-UX kernel, enter the `lsdev` command. In general, many more devices have raw drivers than have block drivers.

The device switch tables are arrays of operational pointers. Each entry in the table consists of a subarray of jump points. The routines pointed to are entered in a documented order and cover actions such as open, close, read, write, and several more. If it sounds like we are talking about file I/O, you are absolutely correct—but remember that in the world of HP-UX (and UNIX in general), all types of I/O are treated like file I/O. No matter what type of physical device a driver was created to work with, to system process threads, first you open it, then you read or write to it, and when you are through you close it!

Summary

This concludes our first pass through the kernel designer's tool kit. Knowledge of these structures and tables and how they work will help prepare us for an in-depth examination of the HP-UX kernel. HP-UX and UNIX in general have many nooks and crannies to be explored and understood. Please reference the generic algorithm explanations offered in this chapter as we encounter specific examples of their usage in the chapters to come.