Queues

A common programming pattern in any operating system kernel is *producer and consumer.* In this pattern, a producer creates data—say, error messages to be read or networking packets to be processed—while a consumer, in turn, reads, processes, or otherwise *consumes* the data. Often the easiest way to implement this pattern is with a *queue.* The producer pushes data onto the queue and the consumer pulls data off the queue. The consumer retrieves the data in the order it was enqueued. That is, the first data on the queue is the first data off the queue. For this reason, queues are also called *FIFOs,* short for *first-in, first-out.* See Figure 6.5 for an example of a standard queue.

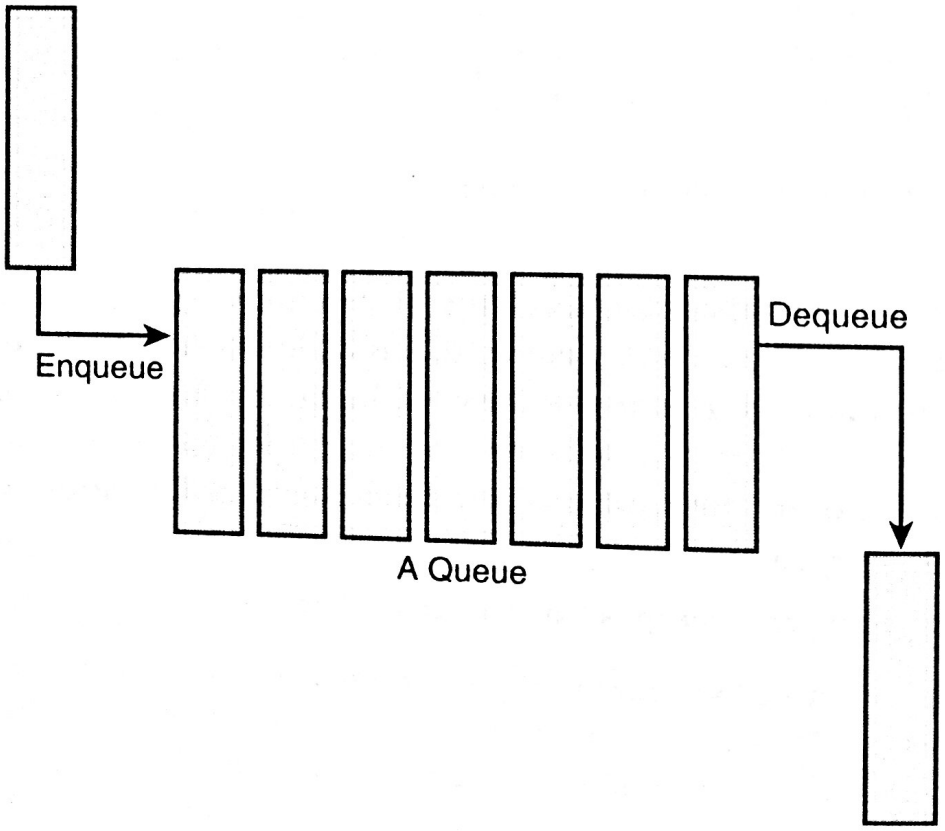

Figure 6.5 A queue (FIFO).

The Linux kernel's generic queue implementation is called *kfifo* and is implemented in `kernel/kfifo.c` and declared in `<linux/kfifo.h>`. This section discusses the API after an update in 2.6.33. Usage is slightly different in kernel versions prior to 2.6.33—double-check `<linux/kfifo.h>` before writing code.

kfifo

Linux's kfifo works like most other queue abstractions, providing two primary operations: enqueue (unfortunately named *in*) and dequeue (*out*). The kfifo object maintains two offsets into the queue: an *in offset* and an *out offset*. The in offset is the location in the queue to which the next enqueue will occur. The out offset is the location in the queue from which the next dequeue will occur. The out offset is always less than or equal to the in offset. It wouldn't make sense for it to be greater; otherwise, you could dequeue data that had not yet been enqueued.

The enqueue (in) operation copies data into the queue, starting at the in offset. When complete, the in offset is incremented by the amount of data enqueued. The dequeue (out) operation copies data out of the queue, starting from the out offset. When complete, the out offset is incremented by the amount of data enqueued. When the out offset is equal to the in offset, the queue is empty: No more data can be dequeued until more data is enqueued. When the in offset is equal to the length of the queue, no more data can be enqueued until the queue is reset.

Creating a Queue

To use a kfifo, you must first define and initialize it. As with most kernel objects, you can do this dynamically or statically. The most common method is dynamic:

```
int kfifo_alloc(struct kfifo *fifo, unsigned int size, gfp_t gfp_mask);
```

This function creates and initializes a kfifo with a queue of `size` bytes. The kernel uses the *gfp mask* `gfp_mask` to allocate the queue. (We discuss memory allocations in Chapter 12, "Memory Management"). Upon success, `kfifo_alloc()` returns zero; on error it returns a negative error code. Following is a simple example:

```
struct kfifo fifo;
int ret;

ret = kfifo_alloc(&kifo, PAGE_SIZE, GFP_KERNEL);
if (ret)
        return ret;

/* 'fifo' now represents a PAGE_SIZE-sized queue ... */
```

If you want to allocate the buffer yourself, you can:

```
void kfifo_init(struct kfifo *fifo, void *buffer, unsigned int size);
```

This function creates and initializes a kfifo that will use the `size` bytes of memory pointed at by `buffer` for its queue. With both `kfifo_alloc()` and `kfifo_init()`, `size` must be a power of two.

Statically declaring a kfifo is simpler, but less common:

```
DECLARE_KFIFO(name, size);
INIT_KFIFO(name);
```

This creates a static kfifo named `name` with a queue of `size` bytes. As before, `size` must be a power of 2.

Enqueuing Data

When your kfifo is created and initialized, enqueuing data into the queue is performed via the `kfifo_in()` function:

```
unsigned int kfifo_in(struct kfifo *fifo, const void *from, unsigned int len);
```

This function copies the `len` bytes starting at `from` into the queue represented by `fifo`. On success it returns the number of bytes enqueued. If less than `len` bytes are free in the queue, the function copies only up to the amount of available bytes. Thus the return value can be less than `len` or even zero, if nothing was copied.

Dequeuing Data

When you add data to a queue with `kfifo_in()`, you can remove it with `kfifo_out()`:

```
unsigned int kfifo_out(struct kfifo *fifo, void *to, unsigned int len);
```

This function copies at most `len` bytes from the queue pointed at by `fifo` to the buffer pointed at by `to`. On success the function returns the number of bytes copied. If less than `len` bytes are in the queue, the function copies less than requested.

When dequeued, data is no longer accessible from the queue. This is the normal usage of a queue, but if you want to "peek" at data within the queue without removing it, you can use `kfifo_out_peek()`:

```
unsigned int kfifo_out_peek(struct kfifo *fifo, void *to, unsigned int len,
                            unsigned offset);
```

This works the same as `kfifo_out()`, except that the out offset is not incremented, and thus the dequeued data is available to read on a subsequent call to `kfifo_out()`. The parameter `offset` specifies an index into the queue; specify zero to read from the head of the queue, as `kfifo_out()` does.

Obtaining the Size of a Queue

To obtain the total size in bytes of the buffer used to store a kfifo's queue, call `kfifo_size()`:

```
static inline unsigned int kfifo_size(struct kfifo *fifo);
```

In another example of horrible kernel naming, use `kfifo_len()` to obtain the number of bytes enqueued in a kfifo:

```
static inline unsigned int kfifo_len(struct kfifo *fifo);
```

To find out the number of bytes available to write into a kfifo, call `kfifo_avail()`:

```
static inline unsigned int kfifo_avail(struct kfifo *fifo);
```

Finally, `kfifo_is_empty()` and `kfifo_is_full()` return nonzero if the given kfifo is empty or full, respectively, and zero if not:

```
static inline int kfifo_is_empty(struct kfifo *fifo);
static inline int kfifo_is_full(struct kfifo *fifo);
```

Resetting and Destroying the Queue

To reset a kfifo, jettisoning all the contents of the queue, call `kfifo_reset()`:

```
static inline void kfifo_reset(struct kfifo *fifo);
```

To destroy a kfifo allocated with `kfifo_alloc()`, call `kfifo_free()`:

```
void kfifo_free(struct kfifo *fifo);
```

If you created your kfifo with `kfifo_init()`, it is your responsibility to free the associated buffer. How you do so depends on how you created it. See Chapter 12 for a discussion on allocating and freeing dynamic memory.

Example Queue Usage

With these interfaces under our belt, let's take a look at a simple example of using a kfifo. Assume we created a kfifo pointed at by `fifo` with a queue size of 8KB. We can now enqueue data onto the queue. In this example, we enqueue simple integers. In your own code, you will likely enqueue more complicated, task-specific structures. Using integers in this example, let's see exactly how the kfifo works:

```
unsigned int i;

/* enqueue [0, 32) to the kfifo named 'fifo' */
for (i = 0; i < 32; i++)
        kfifo_in(fifo, &i; sizeof(i));
```

The kfifo named `fifo` now contains 0 through 31, inclusive. We can take a peek at the first item in the queue and verify it is 0:

```
unsigned int val;
int ret;

ret = kfifo_out_peek(fifo, &val, sizeof(val), 0);
if (ret != sizeof(val))
        return -EINVAL;
```

```
printk(KERN_INFO "%u\n", val); /* should print 0 */
```

To dequeue and print all the items in the kfifo, we can use `kfifo_out()`:

```
/* while there is data in the queue ... */
while (kfifo_avail(fifo)) {
        unsigned int val;
        int ret;

        /* ... read it, one integer at a time */
        ret = kfifo_out(fifo, &val, sizeof(val));
        if (ret != sizeof(val))
                return -EINVAL;

        printk(KERN_INFO "%u\n", val);
}
```

This prints 0 through 31, inclusive, and in that order. (If this code snippet printed the numbers backward, from 31 to 0, we would have a stack, not a queue.)

Maps

A *map*, also known as an *associative array*, is a collection of unique keys, where each key is associated with a specific value. The relationship between a key and its value is called a *mapping*. Maps support at least three operations:

- Add (key, value)
- Remove (key)
- value = Lookup (key)

Although a hash table is a type of map, not all maps are implemented via hashes. Instead of a hash table, maps can also use a self-balancing binary search tree to store their data. Although a hash offers better average-case asymptotic complexity (see the section "Algorithmic Complexity" later in this chapter), a binary search tree has better worst-case behavior (logarithmic versus linear). A binary search tree also enables *order preservation*, enabling users to efficiently iterate over the entire collection in a sorted order. Finally, a binary search tree does not require a hash function; instead, any key type is suitable so long as it can define the <= operator.

Although the general term for all collections mapping a key to a value, the name *maps* often refers specifically to an associated array implemented using a binary search tree as opposed to a hash table. For example, the C++ STL container `std::map` is implemented using a self-balancing binary search tree (or similar data structure), because it provides the ability to in-order traverse the collection.

The Linux kernel provides a simple and efficient map data structure, but it is not a general-purpose map. Instead, it is designed for one specific use case: mapping a unique

identification number (UID) to a pointer. In addition to providing the three main map operations, Linux's implementation also piggybacks an *allocate* operation on top of the *add* operation. This allocate operation not only adds a UID/value pair to the map but also generates the UID.

The idr data structure is used for mapping user-space UIDs, such as inotify watch descriptors or POSIX timer IDs, to their associated kernel data structure, such as the `inotify_watch` or `k_itimer` structures, respectively. Following the Linux kernel's scheme of obfuscated, confusing names, this map is called *idr*.

Initializing an idr

Setting up an idr is easy. First you statically define or dynamically allocate an `idr` structure. Then you call `idr_init()`:

```
void idr_init(struct idr *idp);
```

For example:

```
struct idr id_huh;  /* statically define idr structure */
idr_init(&id_huh);  /* initialize provided idr structure */
```

Allocating a New UID

Once you have an idr set up, you can allocate a new UID, which is a two-step process. First you tell the idr that you want to allocate a new UID, allowing it to resize the backing tree as necessary. Then, with a second call, you actually request the new UID. This complication exists to allow you to perform the initial resizing, which may require a memory allocation, without a lock. We discuss memory allocations in Chapter 12 and locking in Chapters 9 and 10. For now, let's concentrate on using idr without concern to how we handle locking.

The first function, to resize the backing tree, is `idr_pre_get()`:

```
int idr_pre_get(struct idr *idp, gfp_t gfp_mask);
```

This function will, if needed to fulfill a new UID allocation, resize the idr pointed at by `idp`. If a resize is needed, the memory allocation will use the *gfp flags* `gfp_mask` (gfp flags are discussed in Chapter 12). You do not need to synchronize concurrent access to this call. Inverted from nearly every other function in the kernel, `idr_pre_get()` returns one on success and zero on error—be careful!

The second function, to actually obtain a new UID and add it to the idr, is `idr_get_new()`:

```
int idr_get_new(struct idr *idp, void *ptr, int *id);
```

This function uses the idr pointed at by idp to allocate a new UID and associate it with the pointer `ptr`. On success, the function returns zero and stores the new UID in `id`. On error, it returns a nonzero error code: `-EAGAIN` if you need to (again) call `idr_pre_get()` and `-ENOSPC` if the idr is full.

Let's look at a full example:

```
int id;

do {
        if (!idr_pre_get(&idr_huh, GFP_KERNEL))
                return -ENOSPC;
        ret = idr_get_new(&idr_huh, ptr, &id);
} while (ret == -EAGAIN);
```

If successful, this snippet obtains a new UID, which is stored in the integer `id` and maps that UID to `ptr` (which we don't define in the snippet).

The function `idr_get_new_above()` enables the caller to specify a minimum UID value to return:

```
int idr_get_new_above(struct idr *idp, void *ptr, int starting_id, int *id);
```

This works the same as `idr_get_new()`, except that the new UID is guaranteed to be equal to or greater than `starting_id`. Using this variant of the function allows idr users to ensure that a UID is never reused, allowing the value to be unique not only among currently allocated IDs but across the entirety of a system's uptime. This code snippet is the same as our previous example, except that we request strictly increasing UID values:

```
int id;

do {
        if (!idr_pre_get(&idr_huh, GFP_KERNEL))
                return -ENOSPC;
        ret = idr_get_new_above(&idr_huh, ptr, next_id, &id);
} while (ret == -EAGAIN);

if (!ret)
        next_id = id + 1;
```

Looking Up a UID

When we have allocated some number of UIDs in an idr, we can look them up: The caller provides the UID, and the idr returns the associated pointer. This is accomplished, in a much simpler manner than allocating a new UID, with the `idr_find()` function:

```
void *idr_find(struct idr *idp, int id);
```

A successful call to this function returns the pointer associated with the UID `id` in the idr pointed at by `idp`. On error, the function returns `NULL`. Note if you mapped `NULL` to a UID with `idr_get_new()` or `idr_get_new_above()`, this function successfully returns `NULL`, giving you no way to distinguish success from failure. Consequently, you should not map UIDs to `NULL`.

Usage is simple:

```
struct my_struct *ptr = idr_find(&idr_huh, id);
if (!ptr)
        return -EINVAL;  /* error */
```

Removing a UID

To remove a UID from an idr, use `idr_remove()`:

```
void idr_remove(struct idr *idp, int id);
```

A successful call to `idr_remove()` removes the UID `id` from the idr pointed at by `idp`. Unfortunately, `idr_remove()` has no way to signify error (for example if `id` is not in `idp`).

Destroying an idr

Destroying an idr is a simple affair, accomplished with the `idr_destroy()` function:

```
void idr_destroy(struct idr *idp);
```

A successful call to `idr_destroy()` deallocates only unused memory associated with the idr pointed at by `idp`. It does not free any memory currently in use by allocated UIDs. Generally, kernel code wouldn't destroy its idr facility until it was shutting down or unloading, and it wouldn't unload until it had no more users (and thus no more UIDs), but to force the removal of all UIDs, you can call `idr_remove_all()`:

```
void idr_remove_all(struct idr *idp);
```

You would call `idr_remove_all()` on the idr pointed at by `idp` *before* calling `idr_destroy()`, ensuring that all idr memory was freed.

Binary Trees

A *tree* is a data structure that provides a hierarchical tree-like structure of data. Mathematically, it is an *acyclic, connected, directed graph* in which each vertex (called a *node*) has zero or more outgoing edges and zero or one incoming edges. A *binary tree* is a tree in which nodes have at most two outgoing edges—that is, a tree in which nodes have zero, one, or two children. See Figure 6.6 for a sample binary tree.

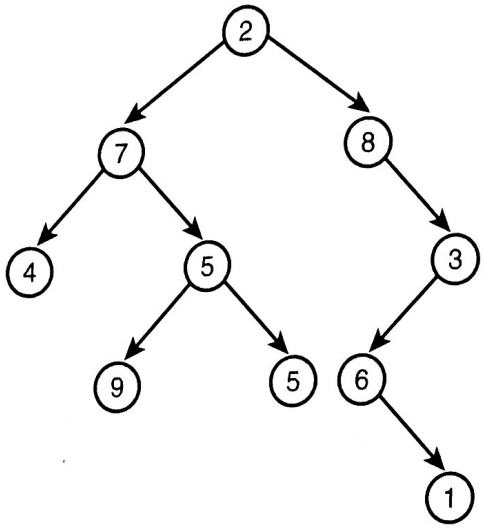

Figure 6.6 A binary tree.

Binary Search Trees

A *binary search tree* (often abbreviated *BST*) is a binary tree with a specific ordering imposed on its nodes. The ordering is often defined via the following induction:

- The left subtree of the root contains only nodes with values less than the root.
- The right subtree of the root contains only nodes with values greater than the root.
- All subtrees are also binary search trees.

A binary search tree is thus a binary tree in which all nodes are *ordered* such that left children are less than their parent in value and right children are greater than their parent. Consequently, both searching for a given node and in-order traversal are efficient (logarithmic and linear, respectively). See Figure 6.7 for a sample binary search tree.

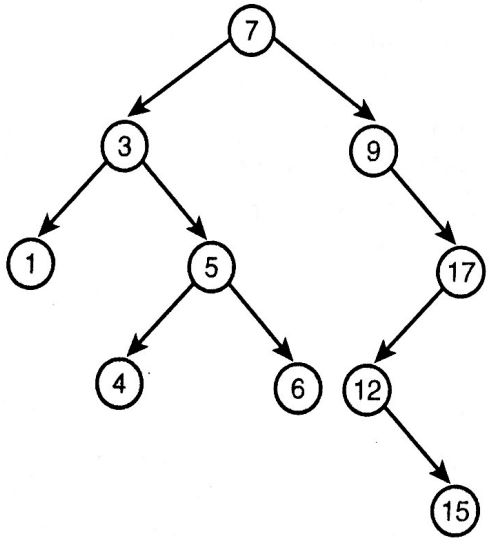

Figure 6.7 A binary search tree (BST).

Self-Balancing Binary Search Trees

The *depth* of a node is measured by how many parent nodes it is from the root. Nodes at the "bottom" of the tree—those with no children—are called *leaves*. The *height* of a tree is the depth of the deepest node in the tree. A *balanced binary search tree* is a binary search tree in which the depth of all leaves differs by at most one (see Figure 6.8). A *self-balancing binary search tree* is a binary search tree that attempts, as part of its normal operations, to remain (semi) balanced.

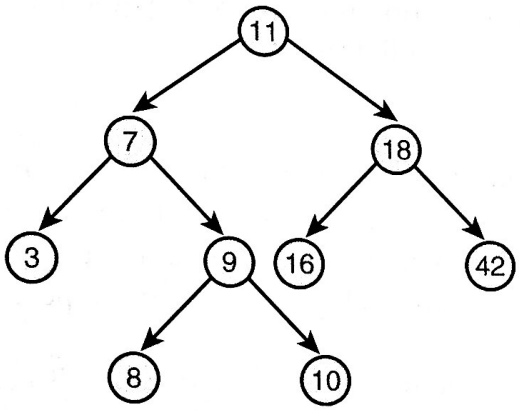

Figure 6.8 A balanced binary search tree.

Red-Black Trees

A *red-black tree* is a type of self-balancing binary search tree. Linux's primary binary tree data structure is the red-black tree. Red-black trees have a special color attribute, which is either *red* or *black*. Red-black trees remain semi-balanced by enforcing that the following six properties remain true:

1. All nodes are either red or black.
2. Leaf nodes are black.
3. Leaf nodes do not contain data.
4. All non-leaf nodes have two children.
5. If a node is red, both of its children are black.
6. The path from a node to one of its leaves contains the same number of black nodes as the shortest path to any of its other leaves.

Taken together, these properties ensure that the deepest leaf has a depth of no more than double that of the shallowest leaf. Consequently, the tree is always semi-balanced. Why this is true is surprisingly simple. First, by property five, a red node cannot be the child or parent of another red node. By property six, all paths through the tree to its leaves have the same number of black nodes. The longest path through the tree alternates red and black nodes. Thus the shortest path, which must have the same number of black nodes, contains only black nodes. Therefore, the longest path from the root to a leaf is no more than double the shortest path from the root to any other leaf.

If the insertion and removal operations enforce these six properties, the tree remains semi-balanced. Now, it might seem odd to require insert and remove to maintain *these* particular properties. Why not implement the operations such that they enforce other, simpler rules that result in a balanced tree? It turns out that these properties are relatively easy to enforce (although complex to implement), allowing insert and remove to guarantee a semi-balanced tree without burdensome extra overhead.

Describing *how* insert and remove enforce these rules is beyond the scope of this book. Although simple rules, the implementation is complex. Any good undergraduate-level data structures textbook ought to give a full treatment.

rbtrees

The Linux implementation of red-black trees is called *rbtrees*. They are defined in `lib/rbtree.c` and declared in `<linux/rbtree.h>`. Aside from optimizations, Linux's rbtrees resemble the "classic" red-black tree as described in the previous section. They remain balanced such that inserts are always logarithmic with respect to the number of nodes in the tree.

The root of an rbtree is represented by the `rb_root` structure. To create a new tree, we allocate a new `rb_root` and initialize it to the special value `RB_ROOT`:

```
struct rb_root root = RB_ROOT;
```

Individual nodes in an rbtree are represented by the `rb_node` structure. Given an `rb_node`, we can move to its left or right child by following pointers off the node of the same name.

The rbtree implementation does not provide search and insert routines. Users of rbtrees are expected to define their own. This is because C does not make generic programming easy, and the Linux kernel developers believed the most efficient way to implement search and insert was to require each user to do so manually, using provided rbtree helper functions but their own comparison operators.

The best way to demonstrate search and insert is to show a real-world example. First, let's look at search. The following function implements a search of Linux's page cache for a chunk of a file (represented by an inode and offset pair). Each inode has its own rbtree, keyed off of page offsets into file. This function thus searches the given inode's rbtree for a matching offset value:

```
struct page * rb_search_page_cache(struct inode *inode,
                                   unsigned long offset)
{
        struct rb_node *n = inode->i_rb_page_cache.rb_node;

        while (n) {
                struct page *page = rb_entry(n, struct page, rb_page_cache);
```

```
                if (offset < page->offset)
                        n = n->rb_left;
                else if (offset > page->offset)
                        n = n->rb_right;
                else
                        return page;
        }
        return NULL;
}
```

In this example, the while loop iterates over the rbtree, traversing as needed to the left or right child in the direction of the given offset. The `if` and `else` statements implement the rbtree's comparison function, thus enforcing the tree's ordering. If the loop finds a node with a matching offset, the search is complete, and the function returns the associated page structure. If the loop reaches the end of the rbtree without finding a match, one does not exist in the tree, and the function returns `NULL`.

Insert is even more complicated because it implements both search and insertion logic. The following isn't a trivial function, but if you need to implement your own insert routine, this is a good guide:

```
struct page * rb_insert_page_cache(struct inode *inode,
                                   unsigned long offset,
                                   struct rb_node *node)
{
        struct rb_node **p = &inode->i_rb_page_cache.rb_node;
        struct rb_node *parent = NULL;
        struct page *page;

        while (*p) {
                parent = *p;
                page = rb_entry(parent, struct page, rb_page_cache);

                if (offset < page->offset)
                        p = &(*p)->rb_left;
                else if (offset > page->offset)
                        p = &(*p)->rb_right;
                else
                        return page;
        }

        rb_link_node(node, parent, p);
        rb_insert_color(node, &inode->i_rb_page_cache);

        return NULL;
}
```

As with our search function, the while loop is iterating over the tree, moving in the direction of the provided offset. Unlike with search, however, the function is hoping *not* to find a matching offset but, instead, reach the leaf node that is the correct insertion point for the new offset. When the insertion point is found, `rb_link_node()` is called to insert the new node at the given spot. `rb_insert_color()` is then called to perform the complicated rebalancing dance. The function returns `NULL` if the page was added to the page cache and the address of an existing `page` structure if the page is already in the cache.

What Data Structure to Use, When

Thus far we've discussed four of Linux's most important data structures: linked lists, queues, maps, and red-black trees. In this section, we cover some tips to help you decide which data structure to use in your own code.

If your primary access method is iterating over all your data, use a linked list. Intuitively, no data structure can provide better than linear complexity when visiting every element, so you should favor the simplest data structure for that simple job. Also consider linked lists when performance is not important, when you need to store a relatively small number of items, or when you need to interface with other kernel code that uses linked lists.

If your code follows the producer/consumer pattern, use a queue, particularly if you want (or can cope with) a fixed-size buffer. Queues make adding and removing items simple and efficient, and they provide first-in, first-out (FIFO) semantics, which is what most producer/consumer use cases demand. On the other hand, if you need to store an unknown, potentially large number of items, a linked list may make more sense, because you can dynamically add any number of items to the list.

If you need to map a UID to an object, use a map. Maps make such mappings easy and efficient, and they also maintain and allocate the UID for you. Linux's map interface, being specific to UID-to-pointer mappings, isn't good for much else, however. If you are dealing with descriptors handed out to user-space, consider this option.

If you need to store a large amount of data and look it up efficiently, consider a red-black tree. Red-black trees enable the searching in logarithmic time, while still providing an efficient linear time in-order traversal. Although more complicated to implement than the other data structures, their in-memory footprint isn't significantly worse. If you are not performing many time-critical look-up operations, a red-black tree probably isn't your best bet. In that case, favor a linked list.

None of these data structures fit your needs? The kernel implements other seldom-used data structures that might meet your needs, such as radix trees (a type of *trie*) and bitmaps. Only after exhausting all kernel-provided solutions should you consider "rolling your own" data structure. One common data structure often implemented in individual source files is the hash table. Because a hash table is little more than some buckets and a hash function, and the hash function is so specific to each use case, there is little value in providing a kernelwide solution in a nongeneric programming language such as C.

Conclusion

In this chapter, we discussed many of the generic data structures that Linux kernel developers use to implement everything from the process scheduler to device drivers. You will find these data structures useful as we continue our study of the Linux kernel. When writing your own kernel code, always reuse existing kernel infrastructure and don't reinvent the wheel.

We also covered algorithmic complexity and tools for measuring and expressing it, the most notable being *big-o* notation. Throughout this book and the Linux kernel, big-o notation is an important notion of how well algorithms and kernel components scale in light of many users, processes, processors, network connections, and other ever-expanding inputs.