Kernel data structures

Kernel Data Structures Umair Hussain Malik p10-6016 edu. pk As with any large software project, the Linux kernel provides these generic data structures and primitives to encourage code reuse. Kernel developers should use these data structures whenever possible and not “ roll your own” solutions. In the following sections, we cover the most useful of these generic data structures, which are the following: * Linked lists * Queues * Maps * Binary trees Linked Lists

The linked list is the most common data structure in the Linux kernel which, allows the storage and manipulation of a variable number of elements, called the nodes of the list. The elements in a linked list are dynamically created and inserted into the list. This enables the management of a varying number of elements unknown at compile time and each element in the list contains a pointer to the next element. As elements are added to or removed from the list, the pointer to the next node is simply adjusted. Singly and Doubly Linked Lists

The simplest data structure representing such a linked list might look similar to the following: /* an element in a linked list */ struct list_element { void *data; struct list_element *next; }; In some linked lists, each element also contains a pointer to the previous element. These lists are called doubly linked lists because they are linked both forward and backward. Linked lists that do not have a pointer to the previous element are called singly linked lists. A data structure representing a doubly linked list would look similar to this: /* an element in a linked list */ struct list_element { void *data;/* truct list_element *next; struct list_element *prev; }; Circular Linked Lists Normally, the last element in a linked list has no next element, it is set to

point to a special value, such as NULL, to indicate it is the last element in the list. In some linked lists, the last element does not point to a special value. Instead, it points back to the first value. This linked list is called a circular linked list because the list is cyclic. Circular linked lists can come in both doubly and singly linked versions. Although the Linux kernel’s linked list implementation is unique, it is fundamentally a circular doubly linked list.

Using this type of linked list provides the greatest flexibility. Moving Through a Linked List Movement through a linked list occurs linearly. You visit one element, follow the next pointer, and visit the next element. Rinse and repeat. This is the easiest method of moving through a linked list, and the one for which linked lists are best suited. Linked lists are illsuited for use cases where random access is an important operation. Instead, you use linked lists when iterating over the whole list is important and the dynamic addition and removal of elements is required.

In linked list implementations, the first element is often represented by a special pointer—called the head—that enables easy access to the “ start” of the list. In a noncircular-linked list, the last element is delineated by its next pointer being NULL. In a circularlinked list, the last element is delineated because it points to the head element. Traversing the list, therefore, occurs linearly through each element from the first to the last. In a doubly linked list, movement can also occur backward, linearly from the last element to the first.

Of course, given a specific element in the list, you can iterate backward and forward any number of elements, too. You need not traverse the whole list.

The Linux Kernel’s Implementation In comparison to most linked list implementations—including the generic approach described in the previous sections—the Linux kernel’s implementation is unique. Recall from the earlier discussion that data (or a grouping of data, such as a struct) is maintained in a linked list by adding a next (and perhaps a previous) node pointer to the data. For example, assume we had a fox structure to describe that member of the Canidae family: truct fox { unsigned long tail_length; unsigned long weight; boolis_fantastic; }; The common pattern for storing this structure in a linked list is to embed the list pointer in the structure. For example: struct fox { unsigned long tail_length; unsigned long weight; boolis_fantastic struct fox*next; struct fox*prev; }; The Linux kernel approach is different. Instead of turning the structure into a linked list, the Linux approach is to embed a linked list node in the structure. The Linked List Structure In the old days, there were multiple implementations of linked lists in the kernel.

A single, powerful linked list implementation was needed to remove duplicate code. During the 2. 1 kernel development series, the official kernel linked-list implementation was introduced. All existing uses of linked lists now use the official implementation; do not reinvent the wheel! The linked-list code is declared in the header file and the data structure is simple: struct list_head { struct list_head *next struct list_head *prev; }; The next pointer points to the next list node, and the prev pointer points to the previous list node. Yet, seemingly, this is not particularly useful.

What value is a giant linked list of linked list nodes? The utility is in how the list_head structure is used: struct fox { unsigned longtail_length; unsigned

longweight; boolis_fantastic; struct list_head list; }; With this, list. next in fox points to the next element, and list. prev in fox points to the previous. Now this is becoming useful, but it gets better. The kernel provides a family of routines to manipulate linked lists. For example, the list_add() method adds a new node to an existing linked list. These methods, however, are generic: They accept only list_head structures.

Using the macro container_of(), we can easily find the parent structure containing any given member variable. This is because in C, the offset of a given variable into a structure is fixed by the ABI at compile time. Defining a Linked List As shown, a list_head by itself is worthless; it is normally embedded inside your own structure: struct fox { unsigned longtail_length; unsigned longweight; boolis_fantastic; struct list_head list; }; List Heads One nice aspect of the kernel’s linked list implementation is that our fox nodes are indistinguishable.

Each contains a list_head, and we can iterate from any one node to the next, until we have seen every node. This approach is elegant, but you will generally want a special pointer that refers to your linked list, without being a list node itself. Interestingly, this special node is in fact a normal list_head: static LIST_HEAD(fox_list); This defines and initializes a list_head named fox_list. The majority of the linked list routines accept one or two parameters: the head node or the head node plus an actual list node. Manipulating Linked Lists The kernel provides a family of functions to manipulate linked lists.

They all take pointers to one or more list_head structures. The functions are implemented as inline functions in generic C and can be found in . Interestingly, all these functions are O(1). 1 This means they execute in constant time, regardless of the size of the list or any other inputs. * To add a node immediately after the head of linked list: list_add(struct list_head *new, struct list_head *head) * To add a node to the end of a linked list: list_add_tail(struct list_head *new, struct list_head *head) * After adding a node to a linked list, deleting a node from a list is the next most important operation.

To delete a node from a linked list, use list_del(): list_del(struct list_head *entry) 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. The Linux kernel’s generic queue implementation is called kfifo and is implemented in kernel/kfifo. c and declared in . 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. 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); * 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); * 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); 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); * 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); * 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”, 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 … */ hile (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”, 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 a binary search tree has better worst-case behavior. 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 s 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. 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);

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. * The first function, to resize the backing tree, is idr_pre_get(): int idr_pre_get(struct idr *idp, gfp_t gfp_mask); * 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); 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); Removing a UID To remove a UID from an idr, use idr_remove(): void idr_remove(struct idr *idp, int id); Destroying an IDR Destroying an idr is a simple affair, accomplished with the idr_destroy()function: void idr_destroy(struct idr *idp); 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. 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. 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. A self-balancing binary search tree is a binary search

tree that attempts, as part of its normal operations, to remain (semi) balanced.

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: * All nodes are either red or black. * Leaf nodes are black. * Leaf nodes do not contain data. * All non-leaf nodes have two children. * If a node is red, both of its children are black. * 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.

The Linux implementation of red-black trees is called rbtrees. They are defined in lib/rbtree. c and declared in . Conclusion In this paper we discussed many of the generic data structures that Linux kernel developers use to implement everything from the process scheduler to device drivers. When writing your own kernel code, always reuse existing kernel infrastructure and don’t reinvent the wheel. Reference Linux Kernel Development (Third Edition) by Robert Love