PREV < Day Day Up > NEXT

11.5. Linked Lists

Operating system kernels, like many other programs, often need to maintain lists of data structures. The Linux kernel has, at times, been host to several linked list implementations at the same time. To reduce the amount of duplicated code, the kernel developers have created a standard implementation of circular, doubly linked lists; others needing to manipulate lists are encouraged to use this facility.

When working with the linked list interface, you should always bear in mind that the list functions perform no locking. If there is a possibility that your driver could attempt to perform concurrent operations on the same list, it is your responsibility to implement a locking scheme. The alternatives (corrupted list structures, data loss, kernel panics) tend to be difficult to diagnose.

To use the list mechanism, your driver must include the file *<linux/list.h>*. This file defines a simple structure of type `list_head`:

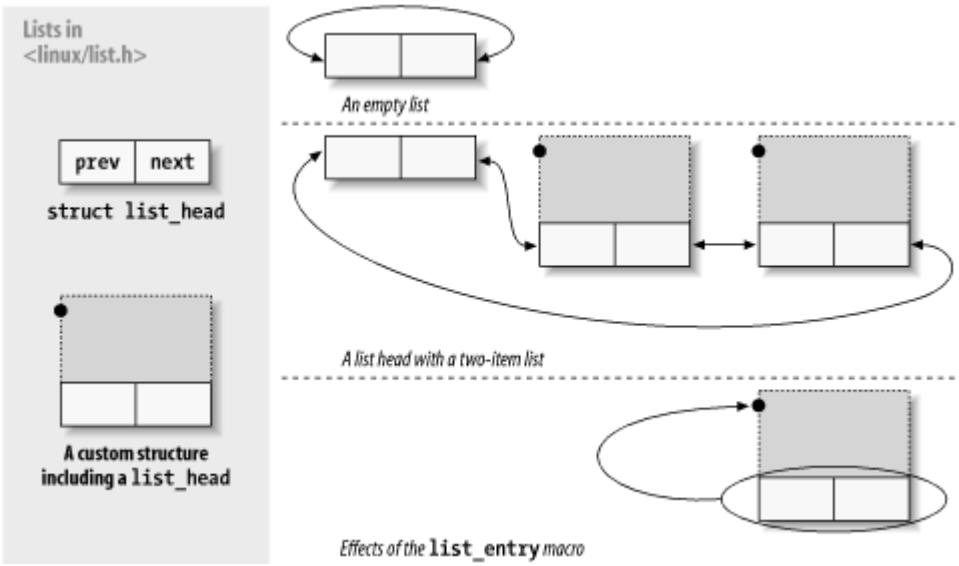
```
struct list_head {

    struct list_head *next, *prev;

};
```

Linked lists used in real code are almost invariably made up of some type of structure, each one describing one entry in the list. To use the Linux list facility in your code, you need only embed a `list_head` inside the structures that make up the list. If your driver maintains a list of things to do, say, its declaration would look something like this:

```
struct todo_struct {

    struct list_head list;

    int priority; /* driver specific */

    /* ... add other driver-specific fields */

};
```

The head of the list is usually a standalone `list_head` structure. Figure 11-1 shows how the simple `struct list_head` is used to maintain a list of data structures.

Figure 11-1. The list_head data structure



List heads must be initialized prior to use with the `INIT_LIST_HEAD` macro. A "things to do" list head could be declared and initialized with:

```
struct list_head todo_list;


INIT_LIST_HEAD(&todo_list);
```

Alternatively, lists can be initialized at compile time:

```
LIST_HEAD(todo_list);
```

Several functions are defined in *<linux/list.h>* that work with lists:

```
list_add(struct list_head *new, struct list_head *head);
```

> Adds the `new` entry immediately after the list head—normally at the beginning of the list. Therefore, it can be used to build stacks. Note, however, that the `head` need not be the nominal head of the list; if you pass a `list_head` structure that happens to be in the middle of the list somewhere, the new entry goes immediately after it. Since Linux lists are circular, the head of the list is not generally different from any other entry.

```
list_add_tail(struct list_head *new, struct list_head *head);
```

> Adds a new entry just before the given list head—at the end of the list, in other words. *list_add_tail* can, thus, be used to build first-in first-out queues.

```
list_del(struct list_head *entry);


list_del_init(struct list_head *entry);
```

The given entry is removed from the list. If the entry might ever be reinserted into another list, you should use *list_del_init*, which reinitializes the linked list pointers.

```
list_move(struct list_head *entry, struct list_head *head);
```

```
list_move_tail(struct list_head *entry, struct list_head *head);
```

The given `entry` is removed from its current list and added to the beginning of `head`. To put the entry at the end of the new list, use *list_move_tail* instead.

```
list_empty(struct list_head *head);
```

Returns a nonzero value if the given list is empty.

```
list_splice(struct list_head *list, struct list_head *head);
```

Joins two lists by inserting `list` immediately after `head`.

The `list_head` structures are good for implementing a list of like structures, but the invoking program is usually more interested in the larger structures that make up the list as a whole. A macro, *list_entry*, is provided that maps a `list_head` structure pointer back into a pointer to the structure that contains it. It is invoked as follows:

```
list_entry(struct list_head *ptr, type_of_struct, field_name);
```

where `ptr` is a pointer to the `struct list_head` being used, `type_of_struct` is the type of the structure containing the `ptr`, and `field_name` is the name of the list field within the structure. In our `todo_struct` structure from before, the list field is called simply `list`. Thus, we would turn a list entry into its containing structure with a line such as:

```
struct todo_struct *todo_ptr =
    list_entry(listptr, struct todo_struct, list);
```

The *list_entry* macro takes a little getting used to but is not that hard to use.

The traversal of linked lists is easy: one need only follow the `prev` and `next` pointers. As an example, suppose we want to keep the list of `todo_struct` items sorted in descending priority order. A function to add a new entry would look something like this:

```
void todo_add_entry(struct todo_struct *new)
{
    struct list_head *ptr;
    struct todo_struct *entry;

    for (ptr = todo_list.next; ptr != &todo_list; ptr = ptr->next) {
```

```
        entry = list_entry(ptr, struct todo_struct, list);
        if (entry->priority < new->priority) {
            list_add_tail(&new->list, ptr);
            return;
        }
    }
    list_add_tail(&new->list, &todo_struct)
}
```

However, as a general rule, it is better to use one of a set of predefined macros for creating loops that iterate through lists. The previous loop, for example, could be coded as:

```
void todo_add_entry(struct todo_struct *new)
{
    struct list_head *ptr;
    struct todo_struct *entry;

    list_for_each(ptr, &todo_list) {
        entry = list_entry(ptr, struct todo_struct, list);
        if (entry->priority < new->priority) {
            list_add_tail(&new->list, ptr);
            return;
        }
    }
    list_add_tail(&new->list, &todo_struct)
}
```

Using the provided macros helps avoid simple programming errors; the developers of these macros have also put some effort into ensuring that they perform well. A few variants exist:

```
list_for_each(struct list_head *cursor, struct list_head *list)
```

> This macro creates a `for` loop that executes once with `cursor` pointing at each successive entry in the list. Be careful about changing the list while iterating through it.

```
list_for_each_prev(struct list_head *cursor, struct list_head *list)
```

This version iterates backward through the list.

`list_for_each_safe(struct list_head *cursor, struct list_head *next, struct list_head *list)`

If your loop may delete entries in the list, use this version. It simply stores the next entry in the list in `next` at the beginning of the loop, so it does not get confused if the entry pointed to by `cursor` is deleted.

`list_for_each_entry(type *cursor, struct list_head *list, member)`

`list_for_each_entry_safe(type *cursor, type *next, struct list_head *list, member)`

These macros ease the process of dealing with a list containing a given `type` of structure. Here, `cursor` is a pointer to the containing structure type, and `member` is the name of the `list_head` structure within the containing structure. With these macros, there is no need to put `list_entry` calls inside the loop.

If you look inside *<linux/list.h>*, you see some additional declarations. The `hlist` type is a doubly linked list with a separate, single-pointer list head type; it is often used for creation of hash tables and similar structures. There are also macros for iterating through both types of lists that are intended to work with the read-copy-update mechanism (described in Section 5.7.5 in Chapter 5). These primitives are unlikely to be useful in device drivers; see the header file if you would like more information on how they work.


< Day Day Up >