How does the kernel implements Linked Lists?

kernelnewbies.org/FAQ/LinkedLists

Navigation

It wasn't uncommon, when working with older versions of the kernel, to encounter redundant code managing classical data structures such as linked lists, hashtables, etc. Recent versions of the kernel now features a "unified" and very smart generic API to manipulate such data structures. Understanding this API can help you make sense of tidbits of kernel code here and there but is also a great opportunity to improve your C programming knowledge by reading some very interestingly put together code.

Let's take a look at the kernel's linked list API from the perspective of "how would I use it in my own code?" (e.g. a Loadable Kernel Module). Let's start by defining a data structure that we will then embed in a kernel linked list:

```
struct mystruct {
     int data ;
} ;
```

To be able to link each element of type*struct mystruct* to others, we need to add a*struct list_head* field:

```
struct mystruct {
     int data ;
     struct list_head mylist ;
} ;
```

When first encountering this, most people are confused because they have been taught to implement linked lists by adding a pointer in a structure which points to the next similar structure in the linked list. The drawback of this approach, and the reason for which the kernel implements linked lists differently, is that you need to write code to handle adding / removing / etc elements specifically for that data structure. Here, we can add a *struct list_head* field to any other data structure and, as we'll see shortly, make it a part of a linked list. Moreover, if you want your data structure to be part of several data structures, adding a few of these fields will work out just fine.

Back to our example, let's create our first variable representing an element of our linked-list-soon-to-be:

```
struct mystruct first = {
     .data = 10,
     .mylist = LIST_HEAD_INIT(first.mylist)
} ;
```

The last line is calling a macro *LIST_HEAD_INIT* which is defined in */include/linux/list.h*:

```

#define LIST_HEAD_INIT(name) { &(name), &(name) }

```

This macro is simply used to assign each pointer inside the mylist field to point to that very field thus representing a list of a single element.

Let's create a second variable and initialize it:

```
struct mystruct second ;
second.data = 20 ;
INIT_LIST_HEAD( & second.mylist ) ;
```

This time, we used a function to initialize the list:

```
static inline void INIT_LIST_HEAD(struct list_head *list)
{
        list->next = list;
        list->prev = list;
}

```

We now need a variable to represent the start (head) of our list, initialize it as an empty linked list to start off with and then add the two elements above.

```
LIST_HEAD(mylinkedlist) ;
```

This macro declares a variable of type *struct list_head* and initializes it for us as defined in:

```
#define LIST_HEAD(name) \
        struct list_head name = LIST_HEAD_INIT(name)

```

Once we have this variable, we add elements to our list:

```
list_add ( &first.mylist , &mylinkedlist ) ;
list_add ( &second.mylist , &mylinkedlist ) ;
```

list_add is a function defined as follows:

```
/**
 * list_add - add a new entry
 * @new: new entry to be added
 * @head: list head to add it after
 *
 * Insert a new entry after the specified head.
 * This is good for implementing stacks.
 */
static inline void list_add(struct list_head *new, struct list_head *head)
{
        __list_add(new, head, head->next);
}

```

It relies on the internal function *list_add*:

```
/*
 * Insert a new entry between two known consecutive entries.
 *
 * This is only for internal list manipulation where we know
 * the prev/next entries already!
 */

static inline void __list_add(struct list_head *new,
                              struct list_head *prev,
                              struct list_head *next)
{
        next->prev = new;
        new->next = next;
        new->prev = prev;
        prev->next = new;
}

```

At this point, we have a handle on a doubly linked list (*mylinkedlist*) which contains two elements. We can iterate over the elements of such a linked list easily but, once again, the kernel linked list API provides us with some macro to make this task even simpler.

```
/**
 * list_for_each        -       iterate over a list
 * @pos:        the &struct list_head to use as a loop counter.
 * @head:       the head for your list.
 */
#define list_for_each(pos, head) \
        for (pos = (head)->next; pos != (head); pos = pos->next)

```

This macro expands into a for loop and requires you to provide a pointer to the list head (*head*) and a pointer to be updated by the loop to point to each consecutive element of the linked list (*pos*).

In our example, we could log to the console the values of the elements of our linked list by using:

```
struct list_head* position ; list_for_each ( position , & mylinkedlist )
    {
         printk ("surfing the linked list next = %p and prev = %p\n" ,
             position->next,
             position->prev );
    }
```

Notice how we use the list_for_each macro so that it expands into the for loop definition and then simply add a body to it. Something else should bother you... We displayed the contents of the struct list_head because this is what the item pointer points to. What if we want to display the contents of the data field of the *struct mystruct*. Afterall, we'll probably want to access the elements we are linking one day or another. Let's start now.

When we have a pointer on a struct *list_head* field which is part of a struct mystruct element, we need to be able to retrieve the address of the latter from the former. The *list_entry* macro does this for us:

```
/**
 * list_entry - get the struct for this entry
 * @ptr:        the &struct list_head pointer.
 * @type:       the type of the struct this is embedded in.
 * @member:     the name of the list_struct within the struct.
 */
#define list_entry(ptr, type, member) \
        container_of(ptr, type, member)
```

With *container_of* being defined in /include/linux/kernel.h as:

```
/**
 * container_of - cast a member of a structure out to the containing structure
 * @ptr:        the pointer to the member.
 * @type:       the type of the container struct this is embedded in.
 * @member:     the name of the member within the struct.
 *
 */
#define container_of(ptr, type, member) ({                      \
        const typeof( ((type *)0)->member ) *__mptr = (ptr);    \
        (type *)( (char *)__mptr - offsetof(type,member) );})

```

The code above deserves some explanation (Also read FAQ/ContainerOf). We have the address of the *struct list_head* field inside of another data structure (let's say *struct task_struct* for sake of example). The first line of the macro casts the value 0 into a pointer of the encapsulating data structure type (*struct task_struct* in our example). We use this pointer to access the field in that data structure which corresponds to our *list_head* and get its type with the macro *typeof* to declare a pointer *mptr* initialised to the value contained in ptr.

The next step is to subtract from the address of ptr (stored in

mptr right now) the offset separating the struct *list_head* field from the beginning of the data structure it is embedded in. This done by using the offsetof macro defined in */include/linux/stddef.h* as;

```
#define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)
```

```
#define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)
```

As you can see, this macro simply expands to the address of the *MEMBER* field in the data structure *TYPE*. To obtain an offset, we take the address of that *MEMBER* from a NULL pointer cast into *TYPE*.

Once this computation is done the container_of macro simply expands to its results as it is composed of 2 lines of code between parenthesis.

We can now write a loop which will display to the console the contents of the data fields of our linked list elements:

```
struct list_head *position = NULL ;
struct mystruct  *datastructureptr  = NULL ;
list_for_each ( position , & mylinkedlist )
    {
         datastructureptr = list_entry ( position, struct mystruct , mylist );
         printk ("data  =  %d\n" , datastructureptr->data );
    }
```

Once again, this has been thought through by the Kernel Developpers who provide us with another macro to simplify this work:

```
/**
 * list_for_each_entry  -       iterate over list of given type
 * @pos:        the type * to use as a loop counter.
 * @head:       the head for your list.
 * @member:     the name of the list_struct within the struct.
 */
#define list_for_each_entry(pos, head, member)                          \
        for (pos = list_entry((head)->next, typeof(*pos), member);      \
             &pos->member != (head);        \
             pos = list_entry(pos->member.next, typeof(*pos), member))

```

Our little example now reads:

```
struct mystruct  *datastructureptr = NULL ;
list_for_each_entry ( datastructureptr , & mylinkedlist, mylist )
    {
         printk ("data  =  %d\n" , datastructureptr->data );
    }
```

That's it for now folks, if you want to explore further, other classic data structures are defined in *include/linux/list.h* for you to learn and toy with.

CategoryFAQ

KernelNewbies: FAQ/LinkedLists (last edited 2017-12-30 01:30:12 by localhost)

Tell others about this page:

SHARE