/**home**/ /**research**/ /**stuff**/

Linux Kernel Linked List Explained

(*Note: This is a working copy, last updated on April 5th, 2005. Feel free to email your comments.*)

Introduction:

Linux kernel is mostly written in the C language. Unlike many other languages C does not have a good collection of data structures built into it or supported by a collection of standard libraries. Therefore, you're probably excited to hear that you can borrow a good implementation of a circularly-linked list in C from the Linux kernel source tree.

The file `include/linux/list.h` in the source tree implements a type oblivious, easy-to-use, circularly-linked list in the C language. The implementation is efficient and portable-- otherwise it would not have made it into the kernel. Whenever someone needs a list in the Linux kernel they rely on this implementation to strung up any data structure they have. With very little modifications (removing hardware prefetching of list items) we can also use this list in our applications. A usable version of this file is available here for download.

Some of the advantages of using this list are:

1. **Type Oblivious:**
   This list can be used to strung up any data structure you have in mind.
2. **Portable:**
   Though I haven't tried in every platform it is safe to assume the list implementation is very portable. Otherwise it would not have made it into the kernel source tree.
3. **Easy to Use:**
   Since the list is type oblivious same functions are used to initialize, access, and traverse any list of items strung together using this list implementation.
4. **Readable:**
   The macros and inlined functions of the list implementation makes the resulting code very elegant and readable.
5. **Saves Time:**
   Stops you from reinventing the wheel. Using the list really saves a lot of debugging time and repetitively creating lists for every data structure you need to link.

Linux implementation of the linked list is different from the many linked list implementations you might have seen. Usually a linked list *contains* the items that are to be linked. For example:

```
struct my_list{
        void *myitem;
        struct my_list *next;
        struct my_list *prev;
        };
```

The kernel's implementation of linked list gives the illusion that the list is contained in the items it links! For example, if you were to create a linked list of `struct my_cool_list` you would do the following:

```
struct my_cool_list{
        struct list_head list; /* kernel's list structure */
        int my_cool_data;
        void* my_cool_void;
        };
```

Couple of things to note here:

1. *List is inside the data item you want to link together.*
2. *You **can put** `struct list_head` **anywhere** in your structure.*
3. *You **can name** `struct list_head` **variable anything** you wish.*
4. *You **can have*** multiple lists!

So for example, the declaration below is also a valid one:

```
struct todo_tasks{
        char *task_name;
        unsigned int name_len;
        short int status;

        int sub_tasks;

        int subtasks_completed;
        struct list_head completed_subtasks;/* list structure */

        int subtasks_waiting;
        struct list_head waiting_subtasks;      /* another list of same or different items! */

        struct list_head todo_list;                     /* list of todo_tasks */
        };
```

Here are some examples of such lists from the kernel:

- include/linux/fs.h:362
- include/linux/fs.h:429

While we are at this, kernel's list structure is declared as follows in `include/linux/list.h`:

```
struct list_head{
        struct list_head *next;
        struct list_head *prev;
        }
```

Having said that this is probably a good time to delve into the details. First let us see how we can use this data structure in our programs. Then, we will see how the data structure actually works.

Using the List:

I think the best way to get familiar with the list functions is to simply scan the file for them. The file is well commented so there should not be any trouble understanding what is available to a user.

Here is an example of creating, adding, deleting, and traversing the list. You can download the source code here.

```
#include <stdio.h>
#include <stdlib.h>

#include "list.h"


struct kool_list{
        int to;
        struct list_head list;
        int from;
        };

int main(int argc, char **argv){

        struct kool_list *tmp;
        struct list_head *pos, *q;
        unsigned int i;

        struct kool_list mylist;
        INIT_LIST_HEAD(&mylist.list);
        /* or you could have declared this with the following macro
         * LIST_HEAD(mylist); which declares and initializes the list
         */

        /* adding elements to mylist */
        for(i=5; i!=0; --i){
                tmp= (struct kool_list *)malloc(sizeof(struct kool_list));

                /* INIT_LIST_HEAD(&tmp->list);
                 *
                 * this initializes a dynamically allocated list_head. we
                 * you can omit this if subsequent call is add_list() or
                 * anything along that line because the next, prev
                 * fields get initialized in those functions.
                 */
                printf("enter to and from:");
                scanf("%d %d", &tmp->to, &tmp->from);

                /* add the new item 'tmp' to the list of items in mylist */
                list_add(&(tmp->list), &(mylist.list));
                /* you can also use list_add_tail() which adds new items to
                 * the tail end of the list
                 */
        }
        printf("\n");


        /* now you have a circularly linked list of items of type struct kool_list.
         * now let us go through the items and print them out
         */


        /* list_for_each() is a macro for a for loop.
         * first parameter is used as the counter in for loop. in other words, inside the
         * loop it points to the current item's list_head.
         * second parameter is the pointer to the list. it is not manipulated by the macro.
         */
        printf("traversing the list using list_for_each()\n");
        list_for_each(pos, &mylist.list){

                /* at this point: pos->next points to the next item's 'list' variable and
                 * pos->prev points to the previous item's 'list' variable. Here item is
                 * of type struct kool_list. But we need to access the item itself not the
                 * variable 'list' in the item! macro list_entry() does just that. See "How
                 * does this work?" below for an explanation of how this is done.
                 */
                 tmp= list_entry(pos, struct kool_list, list);

                 /* given a pointer to struct list_head, type of data structure it is part of,
                  * and it's name (struct list_head's name in the data structure) it returns a
                  * pointer to the data structure in which the pointer is part of.
                  * For example, in the above line list_entry() will return a pointer to the
                  * struct kool_list item it is embedded in!
                  */
```

```
                printf("to= %d from= %d\n", tmp->to, tmp->from);

        }
        printf("\n");
        /* since this is a circularly linked list. you can traverse the list in reverse order
         * as well. all you need to do is replace 'list_for_each' with 'list_for_each_prev'
         * everything else remain the same!
         *
         * Also you can traverse the list using list_for_each_entry() to iterate over a given
         * type of entries. For example:
         */
        printf("traversing the list using list_for_each_entry()\n");
        list_for_each_entry(tmp, &mylist.list, list)
                printf("to= %d from= %d\n", tmp->to, tmp->from);
        printf("\n");


        /* now let's be good and free the kool_list items. since we will be removing items
         * off the list using list_del() we need to use a safer version of the list_for_each()
         * macro aptly named list_for_each_safe(). Note that you MUST use this macro if the loop
         * involves deletions of items (or moving items from one list to another).
         */
        printf("deleting the list using list_for_each_safe()\n");
        list_for_each_safe(pos, q, &mylist.list){
                tmp= list_entry(pos, struct kool_list, list);
                printf("freeing item to= %d from= %d\n", tmp->to, tmp->from);
                list_del(pos);
                free(tmp);
        }

        return 0;
}
```

How Does This Work?

Well most of the implementation is quite trivial but finesse. The finesse relies on the fact that somehow we can obtain the address of an item that contains the list (struct list_head list) given the pointer to the list. This trick is done by the `list_entry()` macro as we saw above. Let us now understand what it does.

```
#define list_entry(ptr, type, member) \
        ((type *)((char *)(ptr)-(unsigned long)(&((type *)0)->member)))
```

Macro expansion for the above example is as follows:

```
        ((struct kool_list *)((char *)(pos) - (unsigned long)(&((struct kool_list *)0)->list)))
```

This is what confuses most people but it is quite simple and a well-known technique (See Question 2.14). Given a pointer to `struct list_head` *in* a data structure, macro `list_entry()` simply computes the pointer *of* the data structure. To achieve this we need to figure out where in the data structure the list_head is (offset of list_head). Then, simply deduct the list_head's offset from the actual pointer passed to the macro.

Now the question is how can we compute the offset of an element in a structure? Suppose you have a data structure `struct foo_bar` and you want to find the offset of element `boo` in it, this is how you do it:

```
(unsigned long)(&((struct foo_bar *)0)->boo)
```

Take memory address 0, and cast it to whatever the type of structure you have-- in this case `struct foo_bar`. Then, take the address of the member you're interested in. This gives the offset of the member within the structure. Since we already know the absolute memory address of this element for a particular instance of the structure (for example, by way of `pos`) deducting this offset points us to the address of the structure itself. That's all there is to it. To get a better handle on this I suggest you play around with this piece of code.

```
#include <stdio.h>
#include <stdlib.h>

struct foobar{
        unsigned int foo;
        char bar;
        char boo;
};

int main(int argc, char** argv){

        struct foobar tmp;

        printf("address of &tmp is= %p\n\n", &tmp);
        printf("address of tmp->foo= %p \t offset of tmp->foo= %lu\n", &tmp.foo, (unsigned long) &((struct foobar *)0)->foo);
        printf("address of tmp->bar= %p \t offset of tmp->bar= %lu\n", &tmp.bar, (unsigned long) &((struct foobar *)0)->bar);
        printf("address of tmp->boo= %p \t offset of tmp->boo= %lu\n\n", &tmp.boo, (unsigned long) &((struct foobar *)0)->boo);

        printf("computed address of &tmp using:\n");
        printf("\taddress and offset of tmp->foo= %p\n",
              (struct foobar *) (((char *) &tmp.foo) - ((unsigned long) &((struct foobar *)0)->foo)));
        printf("\taddress and offset of tmp->bar= %p\n",
              (struct foobar *) (((char *) &tmp.bar) - ((unsigned long) &((struct foobar *)0)->bar)));
        printf("\taddress and offset of tmp->boo= %p\n",
              (struct foobar *) (((char *) &tmp.boo) - ((unsigned long) &((struct foobar *)0)->boo)));

        return 0;
}
```

Output from this code is:

```
address of &tmp is= 0xbfffed00

address of tmp->foo= 0xbfffed00   offset of tmp->foo= 0
address of tmp->bar= 0xbfffed04   offset of tmp->bar= 4
address of tmp->boo= 0xbfffed05   offset of tmp->boo= 5

computed address of &tmp using:
address and offset of tmp->foo= 0xbfffed00
address and offset of tmp->bar= 0xbfffed00
address and offset of tmp->boo= 0xbfffed00
```

See Also

- Please also have a look at the hash table that uses the linked list.
- /sutff/src/ for more source code

TODO

- Figure to explain list_entry() better
- Post the C Data Structures Library (CDSL) that contains hashtables, maps etc. for peer review. Think of it as the Java.Util for C. Clean syntax, prepackaged data structures to make your C life easy!
