# Kernel Data Structures

This is an individual homework, not a group effort. Please be warned – this assignment **looks long**. There is a lot of background information I’m providing to you here. For some, linked lists are totally new. For others, you may have used lists but not built them yourself. For the majority of you, you’ve probably never implemented lists like this.

In the end, my total solution for this homework assignment (all three parts) was 332 lines (including comments, blank lines, etc). I’m giving you almost 275 lines of that code (more than 80%) throughout this document, so there isn’t that much that you’ve got to write yourself.

# Linked List

This chapter introduced operating systems structures. One of the most important data structures, whether in the kernel or any other application is the linked list:

Each node contains a data element that stores the desired information, and a “next” pointer that points to the next node in the list. Recall that we keep a “head” node that tracks the first node in the list. The end of the list is signified by a “next” pointer that is empty.

One of the neat tricks that Linux kernel uses to implement a linked list is to required that any data structure that can be stored on a list must have the linked list node as its first data element. Then the rest of the “struct” can include whatever data it wants.



Figure 1 – A simple linked list.

The structure of a Linked List is *recursive*. It might help if you take a sheet of paper, or just use your hand to cover up parts of the list in the Figure 1. If you cover up all of the nodes except the right-most box, you have an *empty list*. Cover up all of the nodes except #37. You have a linked list that has a “head” and “tail” of #37; a list of size one. If you cover up node #12, you now have a link list that has a “head” at 99, a tail at 37; a list of size two. But, that size 2 list still contains the size 1 list starting at 37; and that size 1 list still contains a size 0 list. This is what we mean when we describe the Linked List as inherently *recursive.* It is this very property that we’ll exploit in the Linux implementation of the Linked List.

### Linked List in Linux

For a variety of reasons, chief among them being that memory is allocated differently in the kernel than in an application the Linux kernel linked list implementation (see <linux/list.h>) is a doubly-linked list that takes a different approach than the one you may have learned in a previous class (e.g. ENGR120, CSC111). At the heart of this implementation is the following structure:



Figure

Linked Lists are seldom used to store integers like Figure 1; rather they are used to store more complex data structures. In the typical C language implementation of the Linked List, the “add a node” function is typically responsible for allocating the linked list memory; but the linked list code doesn’t “know” when to release that memory. Linux solves that by insisting that the anything that ***can***be stored on a list contains the list node’s memory, flipping the relationship on its head.

For example, this creates a “todo” list (examples from [1])

struct todo\_struct {

struct list\_head list;

int priority;

void \*private\_data;

};

Figure

To initialize the to-do list, we simply need to define a “stand alone” list-node, and then initialize it. There are two ways that the Linux kernel, both accomplish the same thing, and both depend on the C pre-processor.

LIST\_HEAD(todo\_list);

Figure – List declaration using LIST\_HEAD macro

### C Pre-Processor Magic

The C pre-processor is a powerful tool and is used extensively in the Linux kernel and other production code bases. In particular, <linux/list.h> file uses the #define pragma to define two *function like macros*. These are macros that look like functions. The arguments that are “passed” in are used in to build the *replacement text*:

#define LIST\_HEAD\_INIT(name) { &(name), &(name) }

#define LIST\_HEAD(name) \  
 struct list\_head name = LIST\_HEAD\_INIT(name)

Figure

To explain how this works, remember that the #define *macro* will be *replaced* with the *text* when the C pre-processor runs, and before the C compiler see the text. So, starting with Figure 4, when the pre-processor encounters:

LIST\_HEAD(todo\_list);

It assigns the macro’s name argument to the text todo\_list. In fact, whatever text we put there will be substituted, the pre-processor is just a string replacer, not a syntax checker. Next, it rewrites the macro with its replacement text (the blue text indicates the rewritten text:  
However, the pre-processor isn’t done, now it sees the macro LIST\_HEAD\_INIT, which has to be expanded too.

struct list\_head todo\_list = { &(todo\_list), &(todo\_list) };

struct list\_head todo\_list = LIST\_HEAD\_INIT(todo\_list);

So, if you’ve followed that, the statement “LIST\_HEAD(todo\_list);” gets replaced by the pre-processor with the declaration of a struct that initializes its previous and next nodes to itself. The C-compiler will *never* see the macros, only the replaced text.

This demonstrates the real power of the C pre-processor hide complex code and give it human readable names. There is an implied “contract” that whomever is responsible for the list code that they had better make it right because no one else is going to look at that code.

### Adding Nodes

We are going to build two “add” functions, one to add at the front (or head), and one to add at the end (or tail). The two functions we are going to study are:

list\_add(struct list\_head \*new, struct struct\_list \*head);

list\_add\_tail(struct list\_head \*new, struct list\_head \*head);

Figure

If we are adding a node at the front then the new node we are adding *becomes* the head of a new linked list. For example, in Figure 1, adding 37 to the empty list makes it the “head” of that list. Adding 99 to that list makes 99 the head of that list, with its next node pointing at the old head node.

So, this is exactly what the Linux list\_add() function is going to accomplish.

/\*\*

\* 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.

\*/  
inline void list\_add(struct list\_head \*new, struct list\_head \*head)  
{  
 struct list\_head \*next = head->next;  
 next->prev = new;  
 new->next = next;  
 new->prev = prev;  
 prev->next = new;

}

Figure Linux list\_add()

So we have a new node, with empty “next” and “prev” pointers. We have a head node with existing “next” and “prev” pointers (possibly pointing to itself). Remember that this is a *doubly linked list*, so if we were to update Figure 1, each node has arrows pointing to the previous node. The head node (#12) has a pointer that points to the tail (#37). The tail node has a next pointer that points to the head node (#12). In effect, this is a *circular* *doubly linked list*.

LIST\_HEAD(todo\_list);

struct todo\_struct first = { .priority = 37 };  
struct todo\_struct second = { .priority = 99 };

struct todo\_struct third = { .priority = 12 };  
  
list\_add(&first.list, &todo\_list);  
list\_add(&second.list, &todo\_list);  
list\_add(&third.list, &todo\_list);

Figure Using the list\_add function.

## List Traversal

Traversing a list means “walking” across the nodes of the list, in order from first to last. The Linux structure makes this relatively simple and provides consistent syntax across *all* lists of any type. We can simply start at *any* part of the list, and follow next or prev pointers to go forwards or backwards. When we arrive at the starting point, we’ve finished.

struct list\_head \*curr;

struct todo\_struct \*entry;

for (curr = todo\_list.next; curr != &todo\_list; curr = curr->next) {  
 entry = list\_entry(curr, struct todo\_struct, list);

printf(“priority %d\n”, entry->priority);  
}

Figure Traversing a Linux linked list

### The “list\_entry” macro

If you read that code carefully, it should have caused you to pause and think “*That’s not valid C syntax – we cannot use the name of a type as an argument to a function call[[1]](#footnote-1).*” What we have here is another function like macro:

/\*\* 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\_head within the struct

\*/  
#define list\_entry(ptr, type, member) \  
 container\_of(ptr, type, member)

Figure Linux list\_entry function like macro

So this function like macro simply gets “expanded” into another function like macro. This is common in Linux. What *probably* happened is that a developer created the “list\_entry” macro and used it throughout the kernel, and then someone else realized it did the same thing as this other extant macro, so they made its implementation invoke the more common version. Since its not really declaring a function there isn’t any penalty for this duplication. Remember, the C compiler will see neither “list\_entry” nor “container\_of,” but only what the final replacement text is.

The container\_of macro is defined in the <linux/kernel.h>:

Figure The Linux container\_of function like macro

/\*\*

\* 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) ({ \

void \*\_\_mptr = (void \*)(ptr); \

BUILD\_BUG\_ON\_MSG(!\_\_same\_type(\*(ptr), ((type \*)0)->member) && \

!\_\_same\_type(\*(ptr), void), \

"pointer type mismatch in container\_of()"); \

((type \*)(\_\_mptr - offsetof(type, member))); })

So, as before, lets decode this step by step. In our source code, we wrote:

entry = list\_entry(curr, struct todo\_struct, list);

The C pre-processor first applies the list\_entry replacement text (in blue):

entry = container\_of(curr, struct todo\_struct, list);

Then the C pre-processor rewrites this using the container\_of replacement text (in red):

{  
 void \*\_\_mptr = (void \*)(curr);

BUILD\_BUG\_ON\_MSG(!\_\_same\_type(\*(curr), ((struct todo\_struct \*)0)->list) &&

!\_\_same\_type(\*(curr), void),

"pointer type mismatch in container\_of()");

((struct todo\_struct \*)(\_\_mptr - offsetof(struct todo\_struct, list)));   
 }

Which, surprisingly, isn’t a syntax error. The BUILD\_BUG\_ON\_MSG macro will actually cause a syntax error during compilation if the types don’t match. The \_\_same\_type macro expands into a compiler built-in operation that asks if two things are of the same type. So, what we’re really checking here is whether the list head node and the list member of the struct todo\_struct are both struct list\_head types. So, lets distill this down a little further, and ignore the error checking:

{  
 void \*\_\_mptr = (void \*)(curr);

((struct todo\_struct \*)(\_\_mptr - offsetof(struct todo\_struct, list)));   
 }

This is a little easier to read. Notice that the macro includes a pair of braces, which creates a new *scope*. We can declare new local variables, and then at the end close of the brace, they go away. Then, we create a void \* \_\_mptr to point at the same memory address as the current node. A “void \*” can point to anything, and doing “pointer arithmetic” is done in bytes. The “offsetof” is a C language built-in that returns the number of bytes a member is away from the beginning of a struct. Since we have the address of the list pointer, we can subtract the offset of the list pointer to get the start of the containing struct. Using this trick we can actually put the “list” member *anywhere* in our struct. We could even have more than one struct list\_head members, and put the object on multiple lists!

So, the “simplified” version of list\_entry is:  
  
#define list\_entry(ptr, type, member) ({ \

void \*\_\_mptr = (void \*)(ptr); \

((type \*)(\_\_mptr - offsetof(type, member))); \

})

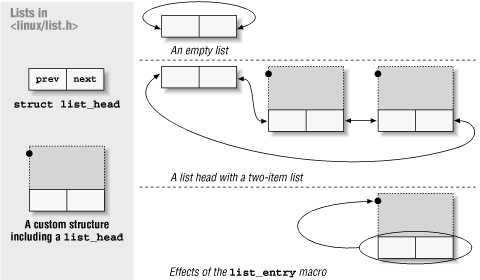


Figure Structure of the linux list

# Actual Homework

Create a “list.h” file that includes the following:

* An “include guard” for this header file (look it up if its an unfamiliar term)
* The function declarations for each of the “list.c” functions below
* The “struct list\_head” declaration
* The function like macros for LIST\_HEAD and list\_entry  
  You can skip the “BUG\_ON” error checking – if a user program dies, it won’t stop the whole machine, but you’ll need to keep the “offsetof” and the rest of the existing list\_entry junk. make sure to “#include <stddef.h>”

Create a “list.c” file and implement the following functions:

* void list\_add(struct list\_head \*new, struct list\_head \*head);

Adds the new item to the list, at the front.

* void list\_add\_tail(struct list\_head \*new, struct list\_head \*head);  
  Adds the new item at to the list, at the end.
* void list\_del(struct list\_head \*entry);

Removes the item from the list. Since the entry that is removed is no longer part of the original list, update its next and prev pointers to itself (like LIST\_INIT).

* int list\_empty(struct list\_head \*head);  
  Returns non-zero if the list is empty, or 0 if the list is not empty.
* void list\_splice(struct list\_head \*list, struct list\_head \*head)  
  Joins two lists by insert list immediately after the head. *Note: the original head node will point to the new list’s head node. The new lists tail node will be updated to point to the original lists next node, completing the chain*.

A note on list splice. I had to draw this on a sheet of paper. Draw *list* to have nodes A, B. Draw *head* to have nodes C and D. After splice, you should have a list: A, B, C, D. Draw what *that* loos like. Then, figure out what pointers need to be adjusted to make that happen. **If you ask me about this function, I will first ask to see your picture to see if you at least tried to figure it out yourself.**

This may initially seem like its a lot, but its not really too bad. I wrote 30 lines of header code, and 55 lines of list.c code. Doing lists this way means we don’t have any mallocs or frees to worry about. If you get stuck, check out the Linux source for these operations for some help – look at

<https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/linux/list.h?h=v5.11-rc5>

To help you get started, here is starting code for part 1. This code will copy all of the command line arguments to the program and add them to a linked list for those that are just words, and another list for when the argument starts with a dash. Note the use “strdup” which may be a new function to you. Look it up if you can’t figure out what it does.  
  
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "list.h"

enum argtype {

WORD, DASH

};  
  
struct argument {

enum argtype type;

char \*contents;

struct list\_head list;

};  
  
int main(int argc, char \*\*argv)

{

int i;

LIST\_HEAD(list\_words);

LIST\_HEAD(list\_dashes);

for(i = 1; i < argc; i++) {

struct argument \*arg = malloc(sizeof(struct argument));

arg->contents = strdup(argv[i]);

if (argv[i][0] == '-') {

arg->type = DASH;

list\_add(&arg->list, &list\_dashes);

}

else {

arg->type = WORD;

list\_add(&arg->list, &list\_words);

}

}

}

Finally, we want to actually do something with the data, so to complete part 1, create a new function that display the items in the list. You’ve already got code to “traverse” the list, so its just a matter of adapting it to use printf. Finally, modify main, so that if the list is empty, it doesn’t print anything, but if it is not empty, it prints a message, and then displays the items

### Part 1 - Sample Output

$ ./part1 one two  
Arguments with words  
two  
one

$ ./part1 -a -b -c  
Arguments with dashes  
-c  
-b  
-a

$ ./part1 a b c -a -b -c -e

Arguments with dashes

-e

-c

-b

-a

Arguments with words

c

b

a

$ ./part1

$

# Part 2 – Cleaning Up

One major advantage to this approach is that the list code doesn’t allocate any memory, and it doesn’t care where the list items come from. But, for many of our programs, we very much care about not leaking memory everywhere. So, **our** program has to clean up after itself.

Here is a function to clean up a list, first freeing the text that was copied using strdup(), and then clearing the argument structure itself.

void clear\_list(struct list\_head \*list)

{

    struct argument \*entry;

    struct list\_head \*curr;

curr = list->next;

    while (! list\_empty(list)) {

        entry = list\_entry(curr, struct argument, list);

  curr = list->next;

        list\_del(&entry->list);

        free(entry->contents);

        free(entry);

    }

}

The astute amongst us will notice that the list traversal used a for loop, whilst this does not. The reason is that we’re going to delete and then free the node that current is pointing at. If we read the next pointer from that memory it *might not be the same value* after the free. So, we have to make *curr* point to the next value before we delete it, but the for() loop cannot do that, so we have to use the while loop here instead. This will typically be the cases whenever we’re deleting from the list in a loop like this.

Copy your part1.c file into part2.c, and then add this clean-up code. Verify that you don’t have any leaks using valgrind.

### Part 2 – Sample Output

$ valgrind ./part1 a b c

Arguments with words

c

b

a

==9088==

==9088== HEAP SUMMARY:

==9088== in use at exit: 0 bytes in 0 blocks

==9088== total heap usage: 7 allocs, 7 frees, 1,126 bytes allocated

==9088==

==9088== All heap blocks were freed -- no leaks are possible

==9088==

==9088== For lists of detected and suppressed errors, rerun with: -s

==9088== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

# Part 3 – Making Your Own Lists

You are on your own for part 3. Make a program that prompts the user to enter the length and width of a rectangle. Create a struct to hold these two doubles. Add them to a list. Stop prompting when they enter a 0 for either length or width.

Then, write a function that compute the area of each rectangle in the list and prints it.

Then, write another function that computes the total area of all the rectangles, use your main logic to print the total.

Then, write a function that *deletes* all of the elements of the list if they have an area less than 10.

Then, use the same function above to print the areas of the remaining rectangles.

Then, use the same function above to compute the total and then print it.

Make sure program cleans up after itself.

### Part 3 – Sample Output

$ ./part3

Enter a length and width: 5 5

Enter a length and width: 3 3

Enter a length and width: 7 7

Enter a length and width: 0 0

area 7.000 \* 7.000 = 49.000

area 3.000 \* 3.000 = 9.000

area 5.000 \* 5.000 = 25.000

Total area: 83.000

Removing small rectangles

area 7.000 \* 7.000 = 49.000

area 5.000 \* 5.000 = 25.000

Total area: 74.000

# Deliverables

Zip up the “Makefile” and all of the C files, and submit them to the Autograder in Gradescope for this problem. Of course, you can also use GitHub and use that to post your solutions too!

Your executables should be named.

* part1
* part2
* part3

As before, I will be grading your source code as well:

* Style / Readability – yes, you may have document/comment MY code, and in your own words
* Proper use of system calls – including relevant error handling
* Proper functional decomposition
* Memory allocation / free – valgrind reports no leaks / misuse of memory

1. Ok, lets just agree that it gave you pause, it makes me feel better. [↑](#footnote-ref-1)