Linux Kernel Training. Lecture 4

Basic data structures

Oleksandr Redchuk
<oleksandr.redchuk@gmail.com>

April 3, 2020. GlobalLogic

C99-style initializers

```
struct hello_data {
        int tcount;
        struct list_head tlist;
};

static struct hello_data data = {
        .tlist = LIST_HEAD_INIT(data.tlist),
};

enum { STATUS_OK, STATUS_SO_SO, STATUS_BAD, STATUS_CODE_MAX};
const char *status_names[STATUS_CODE_MAX] = {
    [STATUS_OK]  = "All OK",
    [STATUS_BAD] = "All bad",
};

static struct hlist_head htable[HTABLE_SIZE] = {
    [ 0 ... HTABLE_SIZE - 1 ] = HLIST_HEAD_INIT,
};
```

C magic in Linux Kernel world

- There are two zero-overhead if compile-time and low-overhead if run-time data manipulation primitives in Linux Kernel implemented using standard C facilities:
  - size_t offsetof(type, member)
  - type *container_of(ptr, type, member)
- First is used in places where offset of the @member field in structured data @type is calculated in a portable way because counting number of bytes from start of the structure to the @member manually isn’t portable.
- Second mostly used in call back functions, like workqueue work, to return pointer to structured @type where @ptr contains address of @member field in that @type data structure.

C magic in Linux Kernel world (cont.)

- Let’s look how `size_t offsetof(type, member)` defined in `<linux/stddef.h>` and actually works:

```
/* It takes casts zero (NULL pointer) to TYPE then takes
 * address (via &) of MEMBER in TYPE starting from 0.
 *
 * This effectively the same as offset of MEMBER field starting
 * from the beginning of the TYPE as compiler sees this at
 * compile-time where macro is expanded.
 */
#define offsetof(TYPE, MEMBER)    ((size_t)&((TYPE *)0)->MEMBER)
```

- In most cases offsetof() is used with compile time C type declaration and name of the field known at compile time. However is is perfectly legal to use it with MEMBER data that calculated at runtime.

C magic in Linux Kernel world (cont.)

- Here is example on how offsetof() can be used with runtime data

```
struct my_data {
    unsigned long n;
    unsigned long p[];
};

static unsigned int n = 2;
module_param(n, uint, S_IRUGO);

static int test1_init(void) /* there is no __init here */
{
    return offsetof(struct my_data, p[n]);
}

module_init(test1_init);
```

C magic in Linux Kernel world (cont.)

- To prove that run-time overhead is really negligible we can look at assembly

```
static unsigned int n = 2;
module_param(n, uint, S_IRUGO);

struct my_data {
    unsigned long k;
    unsigned long p[];
};

static int test1_init(void)
{
     return offsetof(struct my_data, p[n]);
}

00000000 <init_module>:
   0:   e3003000    movw    r3, #0     @ load lower 16 bits of n addr
   4:   e3403000    movt    r3, #0     @ load higher 16 bits of n addr
   8:   e5930000    ldr    r0, [r3]    @ load n into r0
   c:   e2800001    add    r0, r0, #1  @ add length of the first structure field k in words
  10:   e1a00100    lsl    r0, r0, #2  @ multiply by 4 (sizeof(unsigned long))
  14:   e12fff1e    bx    lr           @ return
```

C magic in Linux Kernel world (cont.)

- Now look at `type *container_of(ptr, type, member)` defined in `<linux/kernel.h>`

```
/**
 * 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) );})
```

- It is also can be used with runtime evaluable data

C magic in Linux Kernel world (cont.)

- Again, look at assembly

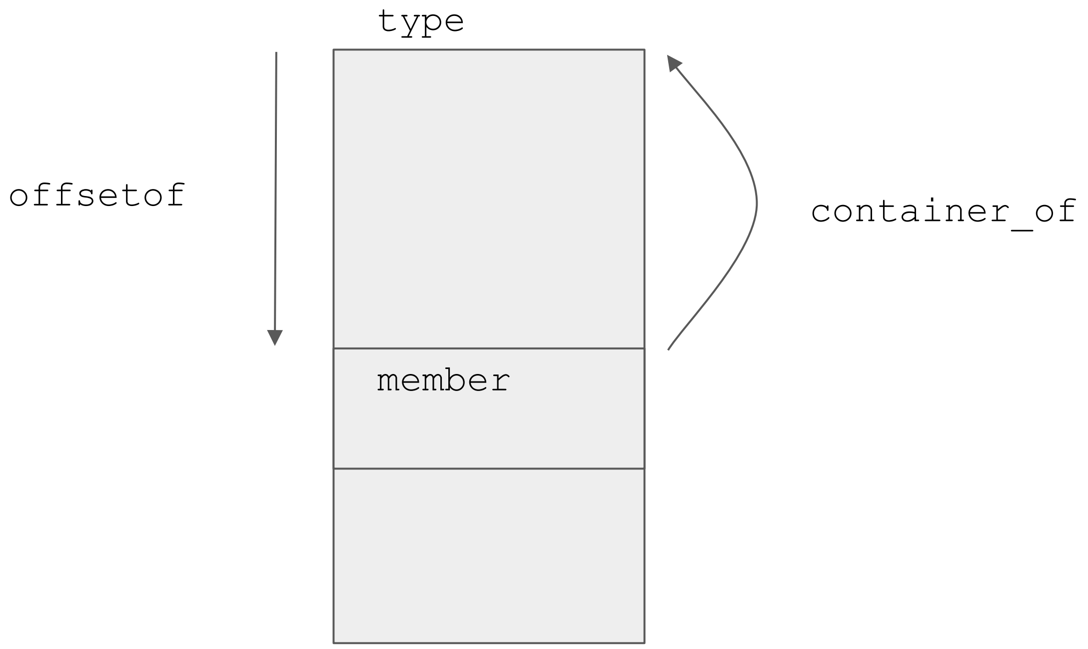
```
struct my_data2 {                    /* offsetof */
    unsigned int __unused1;          /*     0    */
    unsigned int __unused2;          /*     4    */
    unsigned long seed;              /*     8    */
    unsigned int __unused3;          /*    12    */
    struct timer_list timer;         /*    16    */
    struct my_data data;
};

static void test2_timer_func(struct timer_list *timer)
{
    struct my_data2 *md2 = container_of(timer, struct my_data2, timer);
    pr_info("%lx\n", md2->seed);
}

00000000 <test2_timer_func>:
   0:   e5101008    ldr    r1, [r0, #-8] @ "-8" is (offsetof(seed) - offsetof(timer))
   4:   e3000000    movw    r0, #0       @ load lower 16 bits of format string addr
   8:   e3400000    movt    r0, #0       @ load higher 16 bits of format string addr
   c:   eafffffe    b    0 <printk>      @ sibling call optimization, goto to printk
```

C magic in Linux Kernel world (cont.)

- Graphical representation



C magic in Linux Kernel world (cont.)

In Linux Kernel, there are a lot of macros and functions based on container_of

```
#define from_timer(var, callback_timer, timer_fieldname) \
    container_of(callback_timer, typeof(*var), timer_fieldname)
```

```
static inline struct ieee80211_local *hw_to_local(struct ieee80211_hw *hw)
{
    return container_of(hw, struct ieee80211_local, hw);
}
```

C99 flexible array vs gcc zero-length array

```
struct c99_style {
        int count;
        uint32_t data[];
};
```

- Can be placed only at end of structure.
- Incomplete type:
  - such a structure shall not be a member of an another structure or an element of an array;
  - one can't create an instance of such structure.

```
struct usb_ftdi {
        /* ... */
        struct resource resources[0];
        /* ... */
};
```

- Can be placed at any position.
- Complete type, can be used anywhere and an instance can be created.

```
int ftdi_elan_hcd_init(struct usb_ftdi *ftdi)
{
        /* ... */
        ftdi->platform_dev.resource =
                ftdi->resources;
        ftdi->platform_dev.num_resources =
                ARRAY_SIZE(ftdi->resources);
        /* ... */
}
```

C99 flexible array vs gcc zero-length array (cont.)

```
struct sk_buff {
        /* ... */
        __u32    headers_start[0];

        /* ... */

#define PKT_TYPE_OFFSET() offsetof(struct sk_buff, __pkt_type_offset)

        __u8    __pkt_type_offset[0];
        __u8    pkt_type:3;
        __u8    ignore_df:1;
        __u8    nf_trace:1;

        /* ... */

        __u16    network_header;
        __u16    mac_header;

        __u32    headers_end[0];

        /* ... */
};
```

C99 flexible array vs gcc zero-length array (cont.)

Cache-friendly data structure: array `drv_priv[0]` is the last member of `struct ieee80211_sta` which is is the last member of the (private) `struct sta_info`. All this three entities are allocated together as contiguous memory block and we can access an outer structure using `container_of`.

```
struct ieee80211_sta {
     u32 upp_rates[NUM_NL80211_BANDS];
     u16 aid;
    /* ... */
     /* must be last */
     u8 drv_priv[0] __aligned(sizeof(void *));
};

struct sta_info *sta_info_alloc(struct ieee80211_sub_if_data *sdata,
                                const u8 *addr, gfp_t gfp)
{
     struct ieee80211_local *local = sdata->local;
     struct ieee80211_hw *hw = &local->hw;
     struct sta_info *sta;
    /* ... */
     sta = kzalloc(sizeof(*sta) + hw->sta_data_size, gfp);
    /* ... */
}
```

Algorithm complexity

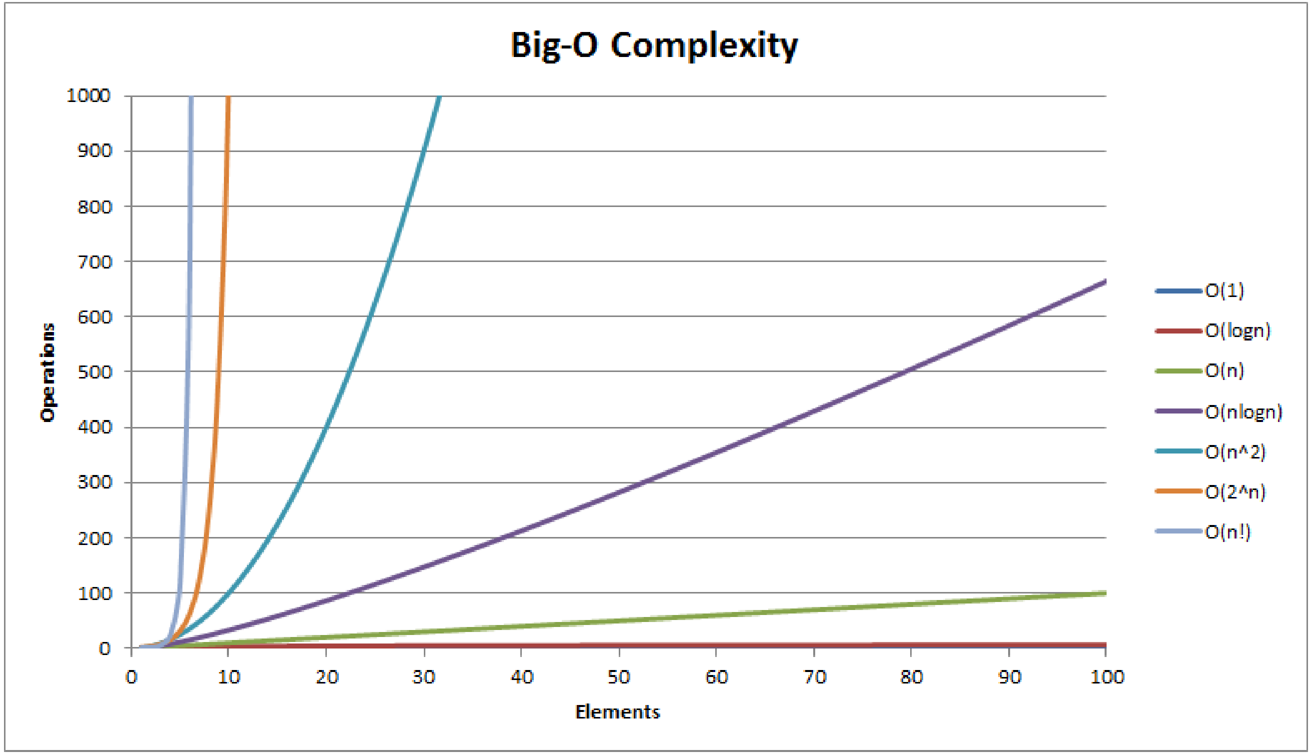
- It is mathematical model to describe how good/bad scales algorithm or it’s specific implementation.
- In general, model is applicable to any kind of resource used involved in algorithm implementation, but most commonly following two are considered:
  - Execution time complexity
  - Storage space complexity
- To find complexity algorithm treated as function $y = f(x)$ where x is algorithm input value (N later here), typically aspiring to infinity.
- There are number of simplifications and assumptions made in the $f(x)$ to make function as simple as possible like remove constants, leave only high affecting part of function, treat nested loops as power of instruction number.

Algorithm complexity (cont.)

- There are many different types of limits (Ω(N), Θ(N), ...). However often, for simplicity of analysis **big O notation** is used to describe **worst**, theoretically possible, TOP boundary complexity of evaluated algorithm that is **never** reached in practice (i.e. one saying O() mean that overall algorithm is better than described, but not worse than).
- Here is few assumptions and approaches to algorithm complexity evaluation:
  - In algorithm containing no loops assume complexity is O(1)
  - For single loop we have O(N) complexity
  - For nested loops we have $O(N^{loop\_nest})$ complexity (e.g. for two nested loops $O(N^2)$)
  - Loops coming one after each other are summed (e.g. $f(N) = N^2+N^4 + 5$)
  - All constant qualifiers are removed from the algorithm complexity function computation
  - Less significant parts of algorithm complexity function removed in favor for most significant (e.g. function is $f(N) = N^3+N^2+2$ can be simplified to $f(N) = N^3$)
- It might be confusing, but most of the algorithms, including custom ones, are simplified to the one of the well known in big O notation.

Algorithm complexity (cont.)

- Here are most valuable well known algorithm complexities O(n)



Algorithm complexity (cont.)

- Here are few examples of algorithms and their complexity
  - O(1)
    - Accessing index within array (i.e. val = a[index])
    - Push/Pop element to stack or Put/Get element to queue data structure
  - O(N)
    - Searching element in array (e.g. libc's memchr() family functions)
    - Two memory region comparison (e.g. libc's memcmp() family functions)
  - O(log N)
    - Binary Search (e.g. search in BST or sorted array)
  - O(N * log N)
    - Quick sort algorithm (e.g. libc's qsort() function)
  - $O(N^2)$
    - Bubble sort

Linked lists in Linux Kernel

- Most common, simple and convenient data structures
- In general developers are free in implementation choice. They can either use their own data structure and list manipulation primitives (iterators, insertion/deletion helpers, etc), but it is first better to look at Linux kernel “standard” linked list implementation if it suits task needs.
- Lists can also be double linked, where each node has pointer to previous one. Also list can terminate with NULL or point to the same stub head.
- In most simple case, singly directed linked list might look as following:

```
struct my_data {
        struct my_data *next;
        unsigned long canary;
};
```

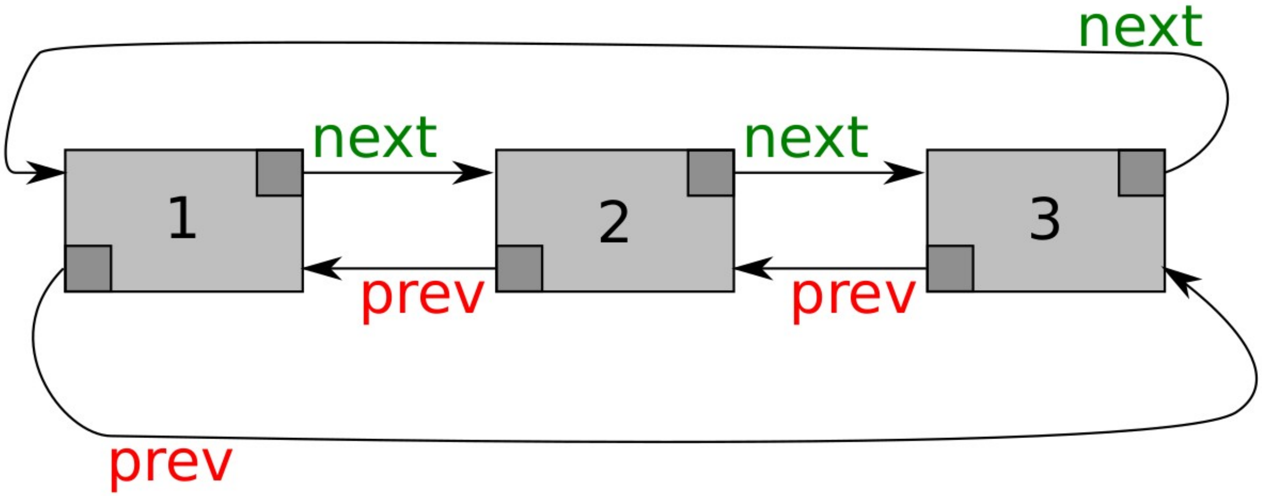
Linked lists in Linux Kernel (cont.)

- Standard Linux Kernel linked lists implemented as doubly linked circular lists.
- In common cases they use stub head to hold reference to the whole list, not just pointer to the first (last) element of the list. This property is used by most of list manipulation primitives as well as iterators.
- These lists are generic, embeddable into customers data structure turning these structures in a linked list.
- Linked list type is so common in kernel, so it is declared in `<linux/types.h>`.
- General linked list manipulation primitives are declared in `<linux/list.h>`.
- There is RCU variant list manipulation primitives in `<linux/rculist.h>`.
- As expected O(N) complexity is for list traversal and O(1) for list manipulation

Linked lists in Linux Kernel (cont.)

- struct list_head definition (see linux/types.h)

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



Linked lists in Linux Kernel (cont.)

- Here is brief overview on how to turn custom data structure into linked list that can be manipulated by standard Linux Kernel linked list primitives.

```
#include <linux/types.h>
#include <linux/list.h>

struct my_data {
    struct list_head list_node;
    struct list_head another_list_node;
    unsigned long canary;
};
```

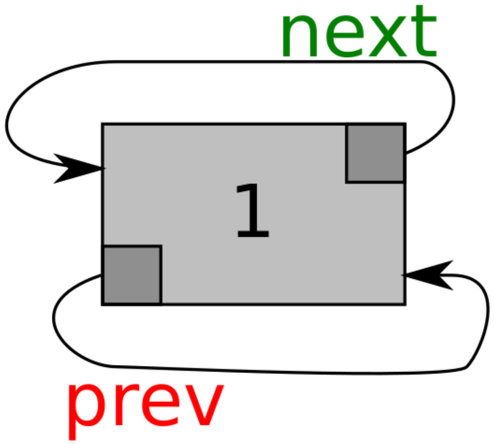
Linked lists in Linux Kernel (cont.)

- And here is how to define Linux Kernel linked list head

```
/* if list is global this can be used to define stub list head */
static LIST_HEAD(my_list_head);

/* which is in turn equivalent to */
static struct list_head my_list_head = LIST_HEAD_INIT(my_list_head);

static struct my_data *cool_init_function(void)
{
    /* here @os some other struct
     * containing stub list head is
     * allocated at runtime (e.g. in heap)*/
    INIT_LIST_HEAD(&os->my_list_head);
}
```



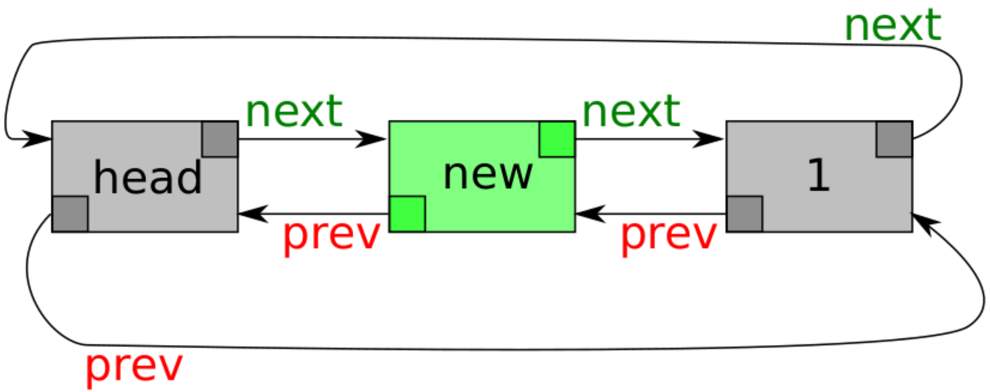
Linked lists in Linux Kernel (cont.)

- Let’s add some static data to the static, global stubby list head

```
struct LIST_HEAD(my_list_head);

/* static array of lists! */
static struct my_list ml[] = {
     { .canary = 0xfade0f01, },
     { .canary = 0xfade0f02, },
     { .canary = 0xfade0f03, },
};

/* somewhere in .text */
for (i = 0; i < ARRAY_SIZE(ml); i++) {
    list_add(&ml[i].list_node, &my_list_head);
}
```



Linked lists in Linux Kernel (cont.)

- Now to test if list is empty one can use list_empty()

```
static LIST_HEAD(my_list_head);

static int init_my_data_list(unsigned int nr_items)
{
    /* Calling this function with non-empty
     * list head isn't supported for now.
     */
    BUG_ON(!list_empty(&my_list_head));
    /* rest of the code goes here */
}
```

- Internally list_empty(head) is simple inline helper in `<linux/list.h>` that returns @true if head->next == head and @false overwise.

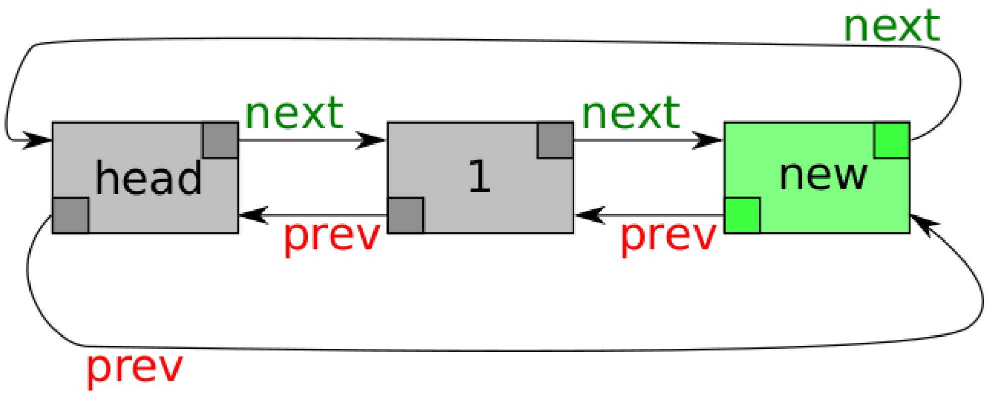
Linked lists in Linux Kernel (cont.)

- And to add in @canary increasing order using list_add_tail()

```
struct LIST_HEAD(my_list_head);

/* static array of lists! */
static struct my_data md[] = {
     { .canary = 0xfade0f01, },
     { .canary = 0xfade0f02, },
     { .canary = 0xfade0f03, },
};

/* somewhere in .text */
for (i = 0; i < ARRAY_SIZE(md); i++) {
    list_add_tail(&md[i].list_node, &my_list_head);
}
```



Linked lists in Linux Kernel (cont.)

- Okay, I want to replace item at known position within list with new one

```
struct my_data new_md = {
    .canary = 0xfade0ff,
};

list_replace(&md[1].list_node, &new_md.list_node);
```

- Note that there most of the routines can be called in empty list safely.
- To check if list is empty there is special helper list_empty() that compares head->next == head:

```
if (!list_empty(&my_list_head))
    /* do something that relies on non-empty list */
```

Linked lists in Linux Kernel (cont.)

- To splice together two lists one can use list_splice() or list_splice_tail()

```
static struct my_data *odd_canary(unsigned int nr_items);
static struct my_data *even_canary(unsigned int nr_items);

static LIST_HEAD(my_list_head);
static LIST_HEAD(my_list_even), LIST_HEAD(my_list_odd);

/* somewhere in the .text */
even_canary(&my_list_even, 10)
odd_canary(&my_list_odd, 10);

list_replace_init(&my_list_even, &my_list_head);
list_splice_tail_init(&my_list_odd, &my_list_head);
```

Linked lists in Linux Kernel (cont.)

- Now let’s delete something from the list using list_del()

```
list_del(&ml[1].list_node);

/* and then flush list completely */
struct my_data *md, *tmp;
list_for_each_entry_safe(md, tmp, my_list_head, list_node) {
        list_del(&md->list_entry);
}

/* check that list is really empty after flush via BUG_ON() */
BUG_ON(!list_empty(my_list_head));
```

- Note there is no need to reinitialize @my_list_head after delete since it becomes empty like after calling INIT_LIST_HEAD() on it.

Linked lists in Linux Kernel (cont.)

- Following routines are used to help to get parent (container) structure from the pointer to the list node and they are basis for rest of the primitives (mostly list iterators):

```
#define list_entry(ptr, type, member) \
    container_of(ptr, type, member)
#define list_first_entry(ptr, type, member) \
    list_entry((ptr)->next, type, member)
#define list_last_entry(ptr, type, member) \
    list_entry((ptr)->prev, type, member)
#define list_next_entry(pos, member) \
    list_entry((pos)->member.next, typeof(*(pos)), member)
#define list_prev_entry(pos, member) \
    list_entry((pos)->member.prev, typeof(*(pos)), member)
#define list_first_entry_or_null(ptr, type, member) \
    (!list_empty(ptr) ? list_first_entry(ptr, type, member) : NULL)
```

Linked lists in Linux Kernel (cont.)

- Now let’s define list manipulation routines explicitly

```
/* Note that all of these functions very simple and thus inline functions in
 * <linux/list.h>. We just omit “static” and “inline” qualifiers here.
 */
int list_empty(const struct list_head *head);
void list_add(struct list_head *new, struct list_head *head);
void list_add_tail(struct list_head *new, struct list_head *head);
void list_del(struct list_head *entry);
void list_del_init(struct list_head *entry);
void list_replace(struct list_head *old, struct list_head *new);
void list_replace_init(struct list_head *old, struct list_head *new);
void list_move(struct list_head *list, struct list_head *head);
void list_move_tail(struct list_head *list, struct list_head *head);
void list_splice(const struct list_head *list, struct list_head *head);
void list_splice_tail(struct list_head *list, struct list_head *head);
void list_splice_init(struct list_head *list, struct list_head *head);
void list_splice_tail_init(struct list_head *list, struct list_head *head);
```

Linked lists in Linux Kernel (cont.)

- When talking about list iterators to travel linked list we can assume following
  - There are two classes
    - First presents list node parent data structure item in the body.
    - Second, less common presents list node itself in the list. Use list_entry() primitive to get pointer to the parent data structure when required, or (better) use one of the iterators from the first class.
  - There are support to travel forward/backward in the list
  - There are support to start from given entry or from next entry in the list. Useful to resume iterations in case of break
  - There are variants “safe” against list element deletion during iteration
  - After iterator completes reaching end-of-list, cursor pointer is **never** NULL
  - They just plain C for() loop statement

Linked lists in Linux Kernel (cont.)

- Here is list of iterators

```
/* First class. Presents parent structure pointed by @pos in iterator body.
 *
 * @pos - pointer to parent struct containing struct list_head @member.
 * @n - same type as @pos, but used to store next entry in the list.
 * @head - stub list head where iterations will stop.
 * @member - field of struct list_head type in structure pointed by @pos
 */
#define list_for_each_entry(pos, head, member)
#define list_for_each_entry_reverse(pos, head, member)
#define list_for_each_entry_continue(pos, head, member)
#define list_for_each_entry_continue_reverse(pos, head, member)
#define list_for_each_entry_from(pos, head, member)
#define list_for_each_entry_safe(pos, n, head, member)
#define list_for_each_entry_safe_continue(pos, n, head, member)
#define list_for_each_entry_safe_from(pos, n, head, member)
#define list_for_each_entry_safe_reverse(pos, n, head, member)
```

Linked lists in Linux Kernel (cont.)

- Here is list of iterators (cont.)

```
/* Second class. Presents list_head structure pointed by @pos in iterator body.
 *
 * @pos - pointer to parent struct containing struct list_head @member.
 * @n - same type as @pos, but used to store next entry in the list.
 * @head - stub list head where iterations will stop.
 */
#define list_for_each(pos, head)
#define list_for_each_prev(pos, head)
#define list_for_each_safe(pos, n, head)
#define list_for_each_prev_safe(pos, n, head)
```

Linked lists in Linux Kernel (cont.)

- The most common pitfall when searching for something and return pointer

```
static struct my_data *find_by_canary(unsigned int canary)
{
    struct my_data *md = NULL;

    list_for_each_entry(md, &my_list_head, list_node) {
        if (md->canary == canary)
            break;
    }

    return md;
}
```

Linked lists in Linux Kernel (cont.)

- The most common pitfall when using routine to populate list

```
static struct list_head *prepare_list_on_stack(unsigned int nr_items)
{
    LIST_HEAD(head);
    unsigned int i;
    for (i = 0; i < nr_items; i++) {
        struct md_data *md;
        /* alloc @md in heap */
        list_add(&md->list_node, &head);
    }
    /* it is illegal to return pointer to stack data, however
     * if you list_del(&head) here you need to list_add() new head
     * in the caller. */
    return &head;
}
```

Hash tables in Linux Kernel

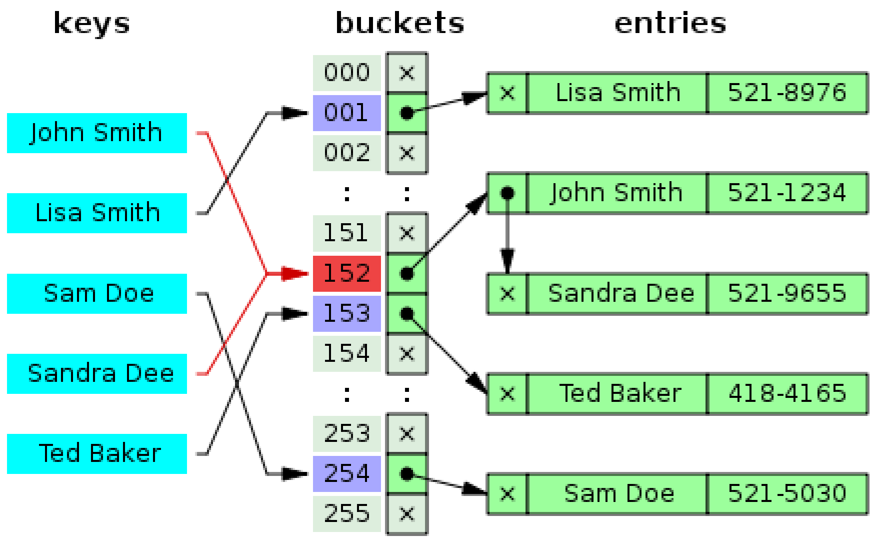
- Besides linked list in Linux Kernel there is another, more powerful and also generic that can be used to implement advanced storage for your data structures in-kernel.
- It is a hash tables that implemented in the similar, generic fashion as linked list in Linux Kernel.
- There are number of helper primitives available as well as data structures to embed into your custom data structures to let them benefit from in-kernel hash tables routines.
- As second, but not last, component required to properly implement hash tables Linux provides proven by time, well tested and fast hash function.

Hash tables in Linux Kernel (cont.)

- To implement hashtable to store data following components may be used:
  - Data type to embed to custom data structures. Declared in `<linux/types.h>`.
  - One dimensional array to serve as hash buckets table. For it's simplest case can be defined as static, but if scalability under large amount of data is required this typically implemented dynamically (re)allocatable array.
  - Hash function index=H(key) that returns for given key index within hash buckets table and meet at least following requirements:
    - It is deterministics (i.e. returns same index for same key)
    - Provides efficient index distribution
    - Fast to compute
    - Must be resistant to collision attacks (this is security sensitive and may not be a strict)
  - Mechanism to resolve collisions: in Linux this achieved by linking all elements into non-circular doubly linked list.

Hash tables in Linux Kernel (cont.)

- What is actually hash table? From the following picture it becomes more clear



- Taken from good article about hash tables on wikipedia.

Hash tables in Linux Kernel (cont.)

- Now let's look to pros/cons when choosing to use hash tables:
  - The main advantage from using hash table is a speed of most common operations (search, insert, remove, etc).
    - For insert/remove to the bucket head or before/after known element this is O(1)
    - For searching some element this is ideally O(1) too, but worst case is O(n) in case of linked list chain in given bucket is too long
    - For traversal this is as expected O(n)
  - For disadvantages we can move following facts:
    - There is no easy way to enumerate elements in hash in some predictable order. This actually requires previous sorting or putting data into linked list in sorted order.
    - Poor processor cache locality. Prefetch is not efficient since requested data is very unlikely in contiguous regions.
    - Subject for collision triggered Denial Of Service (DoS) attacks.

Hash tables in Linux Kernel (cont.)

- In general developers can implement their own hash function index=H(key) to distribute data against of bucket table.
- However before doing so answer following questions:
  - Does your function provide better data distribution and using standard ones?
  - It is faster than any known hash functions in-kernel?
  - Is it secure?
- In Linux there are following standard hashing functions available:
  - Bob Jenkins, lookup3 hash function (see http://burtleburtle.net/bob/hash/) implemented in kernel `<linux/jhash.h>` as inline functions.
  - Donald Knuth functions to hash 32/64 bit unsigned integer and pointers (e.g. hash_32(), hash_64(), hash_ptr()) implemented in kernel `<linux/hash.h>` as inline functions.
  - Function(s) used to hash file name's in `<linux/dcache.h> (e.g. full_name_hash())`

Hash tables in Linux Kernel (cont.)

- Okay, I feel my own specific implementation is better than ones provided in kernel currently. Why and when should I should take care about it’s security?
- Answer for last question isn’t trivial. It requires your data, it’s source as input of H(key) function analysis before you should worry about security. In general you may need to find answer following common questions:
  - Do you fully trust source of data used as key to your H(key)?
  - Is it possible to supply data in specific (specially crafted) way that H(key) will return same index(es) for slightly different data patterns?
  - What countermeasure you provide to protect your implementation?
- If neither of these questions are true: most probably your hashing implementation is safe to use?

Hash tables in Linux Kernel (cont.)

- Well, what about built-in kernel functions? Is all of them secure?
- Unfortunately, not all of them. And should not because each function has (should) it’s own usage scope.
- In general lookup3, jhash() family, should be considered safe as long as random (truly?) data is used as initial value (see initval parameter to jhash() functions family).
- Function(s) used in dcache (e.g. full_name_hash()) protected by the distribution of dentry name values in different directories, etc.
- hash_32() and hash_64() should be used with care.
- hash_ptr() when used properly for kernel address space pointers is safe.

Hash tables in Linux Kernel (cont.)

- When custom hashing is desirable?
  - When data pattern coming as key to H(key) is known or predictable.
  - There is performance benefit using custom hash function.
  - It does not compromise security.
- One of the good examples for custom hashing function is using simplified modular hashing:

```
/* There is H(key), where key is from [U32_MIN .. U32_MAX]  */
static inline u32 H(u32 key, u32 htable_size)
{
    BUILD_BUG_ON(!__builtin_constant(htable_size));
    BUILD_BUG_ON_NOT_POWER_OF_2(htable_size);
    return key & (htable_size - 1);
}
```

Hash tables in Linux Kernel (cont.)

- When to consider dynamic hash table allocation and it’s resize?
- Amounts of data are large enough and can’t be predicted at the compile time.
- It is quite complex to reply to the question on when to resize. Generally this might depend on single bucket chain length, but this requires to track this as well as rehash is quite complex and might require new bucket table allocation as well as remove/insert with hash value recompute from old bucket table to new. This of course would broke O(1) complexity at insert time.
- Furthermore hash table resize in general blocks all further operations on hash table and thus pose negative effects on overall application performance.
- Thus it is quite complex to implement dynamic hash table resize.

Hash tables in Linux Kernel (cont.)

- Let’s look on how to implement static hashtable to store prandom integers.
- At first define static hashtable itself:

```
#include <linux/types.h>
#include <linux/list.h>

/* Here is our hash table order and number of buckets. */
#define HTABLE_SIZE_SHIFT          4
#define HTABLE_SIZE            (1U << HTABLE_SIZE_SHIFT)

static struct hlist_head htable[HTABLE_SIZE] = {
    /* Alternative approach is to use runtime
     * initialized INIT_HLIST_HEAD() for example
     * at the module load time.
     */
    [ 0 ... HTABLE_SIZE - 1 ] = HLIST_HEAD_INIT,
};
```

Hash tables in Linux Kernel (cont.)

- Next define hash function

```
static inline unsigned int my_data_hash(const struct my_data4 *md)
{
#if   HASH_FN_OPT == 1
    return hash_32(md->val, HTABLE_SIZE_SHIFT);
#elif HASH_FN_OPT == 2
    /* let's use some prime number to divide first */
    return (md->val % 97) % HTABLE_SIZE;
#elif HASH_FN_OPT == 3
    /* Not power of two sizes gives very poor hash value distribution here. Why? */
    BUILD_BUG_ON_NOT_POWER_OF_2(HTABLE_SIZE);
    return md->val & (HTABLE_SIZE - 1);
#else
#error HASH_FN_OPT is not defined correctly
#endif
}
```

Hash tables in Linux Kernel (cont.)

- Then after we populate hash table with data let’s find max chain length

```
static unsigned int max_chain_htable(void)
{
    unsigned int i, max_len = 0;

    for (i = 0; i < ARRAY_SIZE(htable); i++) {
        struct hlist_head *head = &htable[i];
        struct hlist_node *n;
        unsigned int count = 0;

        hlist_for_each(n, head)
            count++;

        max_len = max(max_len, count);
    }
    return max_len;
}
```

Hash tables in Linux Kernel (cont.)

- It is time to describe API

```
/* These helpers used to define double linked list for hash buckets */
#define HLIST_HEAD_INIT { .first = NULL }
#define HLIST_HEAD(name) struct hlist_head name = {  .first = NULL }

/* Initialize head at runtime */
#define INIT_HLIST_HEAD(ptr) ((ptr)->first = NULL)

/* Used to initialize node entry at runtime */
static inline void INIT_HLIST_NODE(struct hlist_node *h)
{
    h->next = NULL;
    h->pprev = NULL;
}
```

Hash tables in Linux Kernel (cont.)

- Here are routines used to insert, remove and check

```
/* Note that all of these functions very simple and thus inline
 * functions in <linux/list.h>. We just omit “static” and “inline”
 * qualifiers here.
 */
int hlist_unhashed(const struct hlist_node *h);
int hlist_empty(const struct hlist_head *h);
void hlist_del(struct hlist_node *n);
void hlist_del(struct hlist_node *n);
void hlist_add_head(struct hlist_node *n, struct hlist_head *h);
void hlist_add_before(struct hlist_node *n, struct hlist_node *next);
void hlist_add_behind(struct hlist_node *n, struct hlist_node *prev);
void hlist_move_list(struct hlist_head *old, struct hlist_head *new);
```

Hash tables in Linux Kernel (cont.)

- Finally there is iterators

```
/* @pos, @n - is a pointer to struct hlist_node
 * @head - pointer to struct hlist_head
 */
#define hlist_for_each(pos, head)
#define hlist_for_eacah_safe(pos, n, head)

/* @pos - pointer to the parent struct containing @member
 * @head - pointer to struct hlist_head
 * @member - name of the hlist_head pointed by @pos in typeof(*pos)
 */
#define hlist_entry_safe(pos, type, member)
#define hlist_for_each_entry(pos, head, member)
#define hlist_for_each_entry_continue(pos, member)
#define hlist_for_each_entry_from(pos, member)
```

Hash tables in Linux Kernel (cont.)

- There is one question left after all: why hashtable related manipulation primitives are in `<linux/list.h>` where circular double linked lists live?
- Actually as hash table collision resolution mechanism in bucket linked lists are used on Linux.
- Therefore `struct hlist_head` and `struct hlist_node` data structures define head and node for another linked list kind.
- Thins linked list is double linked, but not circular with single pointer in struct hlist_head. Thus you can’t access list tail at O(1) complexity anymore, but this saves `sizeof(void *)` bytes in structure defining list head.
- It is used quite commonly in the Linux Kernel to for various purposes.

Binary Search Trees (BST) in Linux Kernel

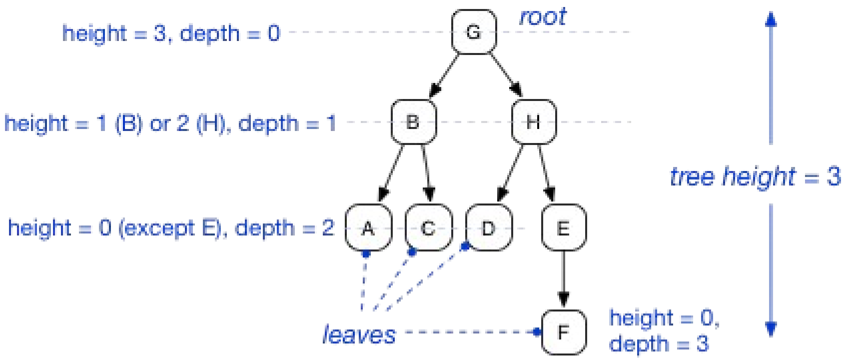
- BST is a binary tree with stored key and optionally value, corresponding to that key, and references to two sub-trees called accordingly “left subtree” and “right subtree” with following properties applied at each node:
  - Keys in **left** subtree always **less** than key of it’s parent node
  - Keys in **right** subtree always **greater** than key of it’s parent node
- BST’s time complexity on common operations like *insert*, *delete* and *lookup* is
  - O(log N) on average or best cases when tree is well balanced
  - O(N) for worst case when tree degenerates to linked list (and note that unlike list *insert* and *delete* operations are O(N) complexity too)
- It is a building block for more high level structures like sets, associative arrays and dictionaries.

Binary Search Trees (BST) in Linux Kernel (cont.)

- Why to use trees if hashes has better time complexity in average case?
  - Some information has tree structure naturally (e.g. dictionary)
  - Memory efficient: no need extra space for hash table array
  - No need to maintain resize of hash table array when number of data grows/shrinks
  - Trees, when nodes allocated properly are processor cache friendly
  - All keys can be obtained in certain order by means of inorder traversal (e.g. from left to right)
  - Good **constant** average time complexity when BST implemented as self-balancing trees
- And what disadvantages BST’s have?
  - Rebalancing to maintain average complexity is complex and inefficient
  - Insert and Delete operations are of O(log N) time complexity compared to O(1) in hash tables
  - Might be quite complex to implement self-balancing/balancing algorithms

Binary Search Trees (BST) in Linux Kernel (cont.)

- Let's define terminology commonly used to describe trees
  - Root node - a node with no parent node
  - Leaf node - a node without child nodes
  - Depth - length of the path from the root node
  - Height - length of the path from the node to the deepest leaf reachable from it
  - Tree height - is the path from root node to the deepest leaf




```
struct bst_node {
        struct bst_node *parent;
        struct bst_node *left;
        struct bst_node *right;
};
```

Binary Search Trees (BST) in Linux Kernel (cont.)

- Keeping tree balanced during common modification operations like insert or delete achieved via rotation operations.
- Algorithms that perform tree rotation called self-balancing
- There are two well known ones from this family
  - AVL Tree named in honor of it's inventors
  - Red Black Tree named according to the colors used for nodes
- Goal of both is to keep tree height proportional to $\log_2(n)$, where n is a number of nodes
- Here is topic on wikipedia about self-balancing trees
- In Linux self-balancing trees implemented using Red Black Trees
  - include/linux/rbtree.h
  - include/linux/rbtree_augmented.h

Binary Search Trees (BST) in Linux Kernel (cont.)

- Red Black Trees use node “color” to keep tree height at most 2*log(n)
- Implementation contains a small hack to make struct rb_node smaller

```
struct rb_node {
    struct bst_node *parent;
    struct bst_node *left;
    struct bst_node *right;
    bool black;
};
```

```
struct rb_node {
    unsigned long  __rb_parent_color;
    struct rb_node *rb_right;
    struct rb_node *rb_left;
}
```

```
#define rb_parent(r)   ((struct rb_node *)((r)->__rb_parent_color & ~3))

static inline void rb_set_black(struct rb_node *rb)
{
    rb->__rb_parent_color |= RB_BLACK;
}
```

Binary Search Trees (BST) in Linux Kernel (cont.)

- To use Red Black trees on Linux one need to implement common operations by itself, using primitives declared/defined `<linux/rbtree.h>` and rebalance implementation in `lib/rbtree.c`.
- There is good source of information on how to use these primitives is a Red Black Tree unit test implementation in `lib/rbtree_test.c`.
- Let’s implement common tree operations using Red Back Tree for collection of strings.
- We will use strcmp() to direct newly inserted values to either left subtree if function return is -1 and to the right subtree if return is 1.
- Implementation is available as part of this presentation in examples/ directory.

That’s all for now. Thank you for viewing.

Authors:

- Serhii Popovych, Software Engineer
  - Cisco-IOSXE Linux SDK, GlobalLogic Ukraine LLC
  - Email: serhii.popovych@globallogic.com
- Oleksandr Redchuk, Software Engineer
  - GlobalLogic Ukraine
  - Email: oleksandr.redchuk@gmail.com