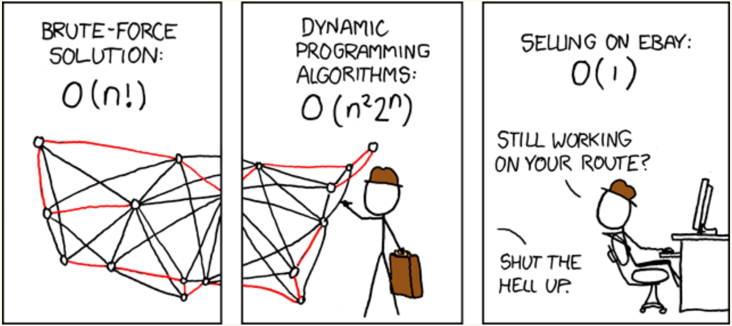
Linux Kernel Programming
Kernel Data Structures

Pierre Olivier

Systems Software Research Group @ Virginia Tech

February 7, 2017



- Source: `https://xkcd.com/399/`

Outline

1. Linked lists
2. Queues
3. Maps
4. Binary trees
5. The right data structure for the right problem

Introduction

- The kernel has efficient implementations of:
  1. **Lists** (singly/doubly linked): `include/linux/list.h`
  2. **Queues**: `include/linux/kfifo.h`
  3. **Maps**: `include/linux/idr.h`
  4. **Binary trees** (*red-black trees*): `include/linux/rbtree.h`
- Do not reinvent the wheel!

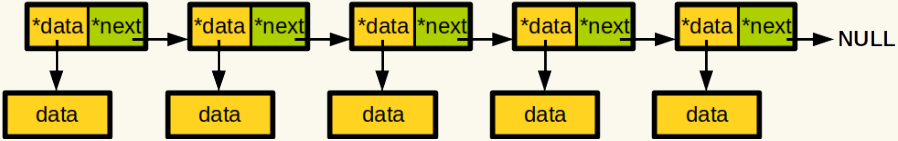
Outline

Linked lists

Singly linked list

```c
struct my_list_element
{
    void                    *data;
    struct my_list_element  *next;
};
```

- `void` pointer to point on generic data
  - Can also contain data directly for a non-generic version
- Pointer to the `next` element

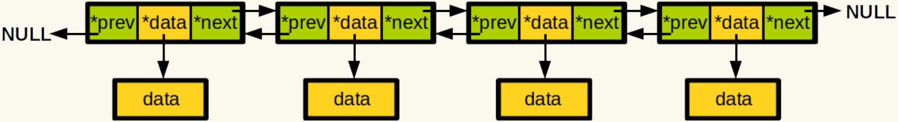


Linked lists

Doubly linked list

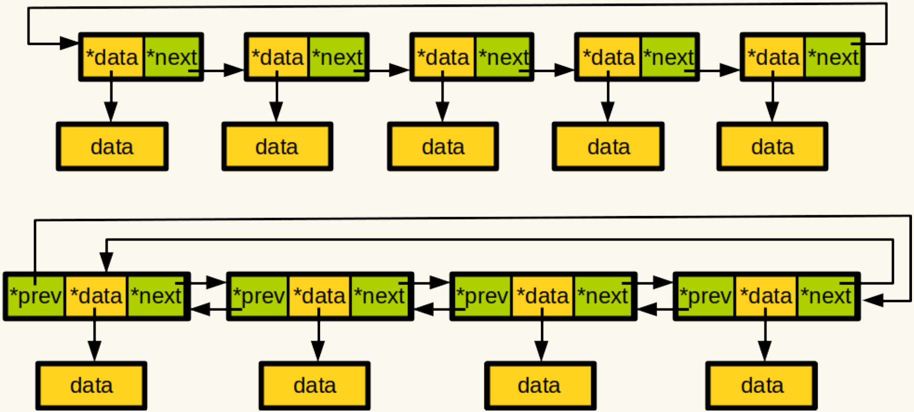
```c
struct my_list_element
{
    void                    *data;
    struct my_list_element  *prev;
    struct my_list_element  *next;
};
```

- Additional `prev` pointer
- Allows backward traversal



Linked lists

Circular lists



- Linked lists are iterated sequentially
  - Inappropriate when random (direct) access to a specific element is needed

Linked lists

Linux implementation: standard approach vs Linux

- Linux implements linked list a bit differently than the standard approach
- Standard approach → add `next`/`prev` pointers to a data structure:

```c
struct car
{
    unsigned int    max_speed;
    unsigned int    drive_wheel_num;
    double          price_in_dollars;
    struct car      *prev;              /* we add this ... */
    struct car      *next;              /* ... and this    */
};
```

Linked lists

Linux implementation: `struct list_head`

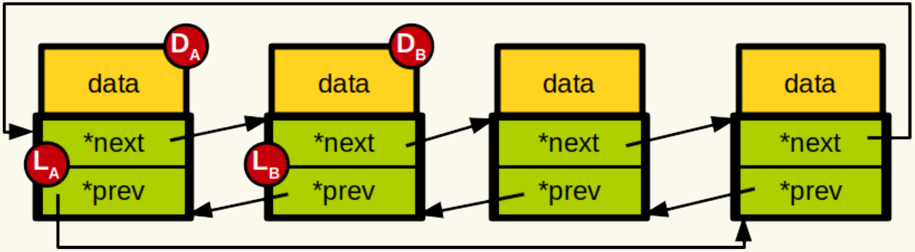
```
struct list_head
{
    struct list_head *next;
    struct list_head *prev;
};
```

- Current implementation was introduced in Linux 2.1
- **`struct list_head`** as the central data structure
- `list_head` is **embedded in the structure we want to link:**

```
struct car
{
    unsigned int        max_speed;
    unsigned int        drive_wheel_num;
    double              price_in_dollars;
    struct list_head    list;               /* we add this */
};
```

Linked lists

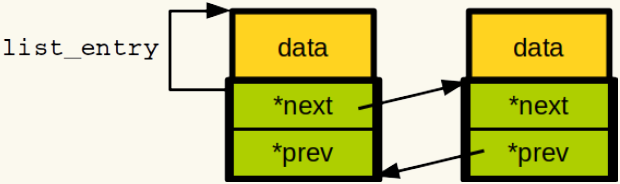
Linux implementation: `struct list_head` (2)



Linked lists

Linux implementation: list_entry

- The kernel provides a *generic* API to manipulates such lists
  - ex: `list_add(struct list_head *new, struct list_head *head)`
  - Manipulates `struct list_head` objects
- **How to get access to the containing data structure given a `struct list_head`?**
  - Use `list_entry`



Linked lists

Linux implementation: list_entry (2)

```
/* let's assume we have a pointer car_list_ptr to a struct list_head embedded into a struct
 * car data object */
struct car *amazing_car = list_entry(car_list_ptr, struct car, list)
```

```
#define list_entry(ptr, type, member) \
        container_of(ptr, type, member)
```

```
#define container_of(ptr, type, member) ({ \
        const typeof( ((type *)0)->member ) *__mptr = (ptr);
        (type *)( (char *)__mptr - offsetof(type,member) );})
```

Linked lists

The container_of macro

```
#define container_of(ptr, type, member) ({ \
  const typeof( ((type *)0)->member ) *__mptr = (ptr);
  (type *)( (char *)__mptr - offsetof(type,member) );})
```

```
/* call to list_entry expands into: */
struct car *amazing_car = container_of(car_list_ptr, struct car, list);
```

```
/* next expansion: */
amazing_car = ({
  const typeof( ((struct car *)0)->list) *__mptr = car_list_ptr;
  (struct car *)( (char *)__mptr - offsetof(struct car, list));
});
```

```
/* last expansion: */
amazing_car = ({
  const struct list_head *__mptr = car_list_ptr;
  (struct car *)( (char *)__mptr - 0x10);
});
```

Linked lists

Defining a linked list

- Previous example:

```
struct car
{
    unsigned int        max_speed;
    unsigned int        drive_wheel_num;
    double              price_in_dollars;
    struct list_head    list;
};
```

- Static (compile-time) definition:

```
struct car my_car =
{
    .max_speed = 150,
    .drive_wheel_num = 2,
    price_in_dollars = 10000.0,
    .list = LIST_HEAD_INIT()
}
```

- Dynamic (runtime) definition, most commonly used:

```
struct car *my_car =
    kmalloc(sizeof(*my_car), GFP_KERNEL);
my_car->max_speed = 150;
my_car->drive_wheel_num = 2;
my_car->price_in_dollars = 10000.0;
INIT_LIST_HEAD(&my_car->list);
```

- Canonical pointer representing the list as a whole:

```
LIST_HEAD(my_car_list); /* my_car_list is a struct list_head */
```

Linked lists

Adding/deleting a node to/from a list

- **`list_add (struct list_head *new, struct list_head *head)`**
  - Add the node right after the `head` node
- **`list_add_tail (struct list_head *new, struct list_head *head)`**
  - Add the node at the end of the list, i.e. before the `head` node
- **`list_del(struct list_head *entry)`**
  - Remove the element from the list
  - You still have to take care of the memory deallocation if needed

```
list_add(&my_car->list, &my_car_list);
list_add_tail(&my_car->list, &my_car_list);
list_del(&my_car->list);
```

Linked lists

Moving/Splicing nodes

- `list_move (struct list_head *list, struct list_head *head)`
- `list_move_tail (struct list_head *list, struct list_head *head)`
  - Move a node from one list to another one
- `list_empty(struct list_head *head)`
  - Returns nonzero if the list is empty
- `list_splice (struct list_head *list, struct list_head *head)`
  - Insert the list pointed by `list` after the element `head`

Linked lists

Iterating over a list

- **list_for_each(), list_for_each_entry()**

```
/* Temporary variable needed to iterate: */
struct list_head p;
/* This will point on the actual data structures (struct car) during the iteration: */
struct car *current_car;

list_for_each(p, &my_car_list)
{
    current_car = list_entry(p, struct car, list);
    printk(KERN_INFO "Price: %lf\n", current_car->price_in_dollars);
}

/* Simpler: use list_for_each_entry */
list_for_each_entry(current_car, &my_car_list, list)
{
    printk(KERN_INFO "Price: %lf\n", current_car->price_in_dollars);
}
```

- list_for_each_entry_reverse()
  - Iterate backwards

Linked lists

Removing while iterating

```
/* This will point on the actual data structures (struct car) during the iteration: */
struct car *current_car, *next;

list_for_each_entry_safe(current_car, next, my_car_list, list)
{
  printk(KERN_INFO "Price: %lf\n", current_car->price_in_dollars);
  list_del(current_car->list);
  kfree(current_car); /* if this was dynamically allocated through kmalloc */
}
```

- For each iteration, `next` points to the next node
  - Can safely remove the current node
  - Otherwise: → *use-after-free* bug

Linked lists

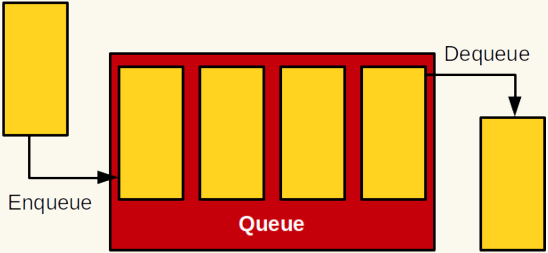
Linked lists: where are they used in the kernel?

- **Kernel code makes extensive use of linked lists:**
  - Linking threads that share a common PID;
  - Linking the superblocks of all partitions sharing a common file system type;
  - Linking processes in a CPU run queue
  - etc.

Outline

Queues

Definition



- Producer/consumer programming model
- **FIFO**: First-In-First-Out
- Implemented in Linux through `struct kfifo`

Queues

Queues: creation

```
struct kfifo my_queue;
char *buffer;
int ret;

/* Version 1: dynamic allocation + initialization of a 1024 bytes sized queue */
ret = kfifo_alloc(&my_queue, 1024, GFP_KERNEL);
if(ret)
    return ret; /* can fail! return error */

/* Version 2: initialization of a (dynamically) pre-allocated buffer to be used as a queue */
buffer = kmalloc(1024, GFP_KERNEL);
ret = kfifo_init(&my_queue, buffer, 1024);
if(ret)
    return ret;

/* Version 3: static declaration */
DECLARE_KFIFO(another_queue, 1024);   /* type of another queue is struct kfifo */
INIT_KFIFO(another_queue);
```

- Size should be a power of 2

Queues

Enqueueing/dequeueing

- Prototypes:

```
/* (these are actually macros in recent kernel versions ) */
unsigned int kfifo_in(struct kfifo *fifo, const void *from, unsigned int len);
unsigned int kfifo_out(struct kfifo *fifo, void *to, unsigned int len);
unsigned int kfifo_out_peek(struct kfifo *fifo, void *to, unsigned int len);
```

```
struct car
{
    unsigned int max_speed;
    unsigned int drive_wheel_num;
    double       price_in_dollars;
};
```

- Enqueueing:

```
unsigned int ret;
struct car car_to_add = {100, 2, 10000.0};

ret = kfifo_in(&fifo, &car_to_add,
    sizeof(struct car));
if(ret != sizeof(struct car))
    /* Not enough space left in the queue */
```

- Dequeueing:

```
struct car amazing_car;
unsigned int ret = kfifo_out(&fifo, &amazing_car, sizeof(struct car));
```

  - Use `kfifo_out_peek` to access the head of the queue without removal

Virginia Tech

Queues

Queue size/reset/destroy

- Information on queue size - prototypes:

```
/* Let's assume we have struct kfifo my_kfifo */
unsigned int buffer_total_size_in_bytes = kfifo_size(&my_kfifo);
unsigned int bytes_used = kfifo_len(&my_kfifo);
unsigned int bytes_free = kfifo_avail(&my_kfifo);
int empty = kfifo_is_empty(&my_kfifo);
int full = kfifo_is_full(&my_kfifo);
```

- Reset a queue (removes all content):

```
kfifo_reset(&my_kfifo); /* returns void */
```

- Free a queue previously allocated through `kfifo_alloc()`

```
kfifo_free(&my_kfifo); /* returns void */
```

- Sample of kernel queues usage:
  - In linux sources, `samples/kfifo`

Queues

Queues usage in the kernel

- List of free blocks for the SmartMedia flash driver
- Used in the message queue driver for TI OMAP processors to buffer messages
- etc.

Outline

Maps
Definition & initialization

- A **map** maps keys to values, supporting 3 main operations:
  - add(key, value)
  - remove(key)
  - value = lookup(key)
- Linux implementation indexes content using a **binary search tree**
  - Keys must support the operation $<=$
- Linux does not implement a general purpose map
  - The implementation named **idr**, **maps *integers* (keys) to *pointers* (values)**
    - These integers are named Unique Identification Numbers (UIDs)
- Initialization:

```
/* Statically */
struct idr my_map;
idr_init(&my_map);
```

```
/* Dynamically */
struct idr *my_map_ptr = kmalloc(sizeof(
      struct idr), GFP_KERNEL);
idr_init(my_map_ptr);
```

Maps

New UID allocation

- 3-steps process
- Prototypes:

```
/* 1. Pre-allocate the memory for the UID allocation request */
void idr_preload(gfp_t gfp_mask);
/* 2. Actual allocation request */
int idr_alloc(struct idr *idp, void *ptr, int start, int end, gfp_t gfp_mask);
/* 3. idr_preload disables preemption, needs to re-enable it: */
void idr_preload_end(void);
```

- Note that the interface to add a new UID has changed since the textbook publication
  - Simplified, removed the need for looping
  - `https://lwn.net/Articles/536293/`

Maps

Insertion: full example

```c
int id;

idr_preload(GFP_KERNEL);
id = idr_alloc(&the_idr, &some_data, 10, 550, GFP_KERNEL);
idr_preload_end();
if(id == -ENOSPC)
    /* error, no id available in the requested range */
else if(id == -ENOMEM)
    /* error, could not allocate memory */
```

- UID range constraints provide more control on allocated UIDs
  - Ex: loop device driver indexes a loop device partitions based on their *minor number*

Maps

UID lookup/removal, map destruction

- Prototypes:

```
void *idr_find(struct idr *idp, int id);
void idr_remove(struct idr *ipd, int id);
void idr_destroy(struct idr *idp);
```

- UID lookup:

```
struct car *my_car = idr_find(&my_map, id); /* returns NULL on error */
if(!my_car)
    return -EINVAL; /* not found */
```

- UID removal:

```
idr_remove(&my_map, id);
```

- Map destruction:

```
idr_destroy(&my_map);
```

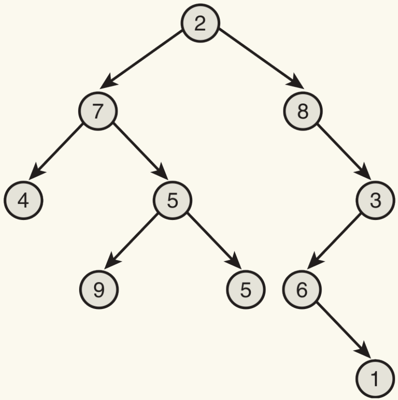
Maps

Maps usage in the kernel

- Indexing loop devices
  - (File acting like a virtual block device (ex: disk images, ISOs, etc.))
- Index permission data structures for IPCs in a namespace
  - (OS level virtualization)
- Index Performance Monitoring Unit events
- etc.

Outline

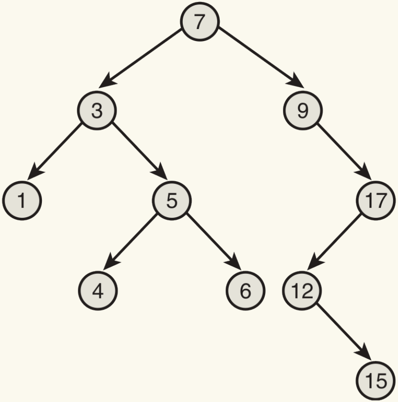
Binary trees
Definition



- **Binary Tree**
- Nodes have zero, one or two children
- Root has 0 parent, other nodes have one

Binary trees

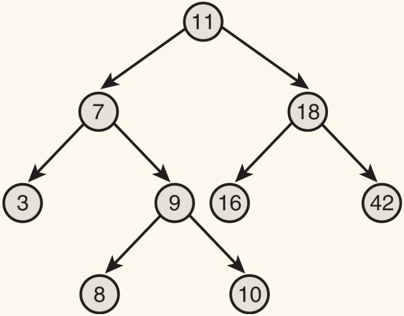
Definition (2)



- **Binary Search Tree**
- Ordered:
  - Left children $<$ parent
  - Right children $>$ parent
  - Search and in-order traversal are efficient

Binary trees

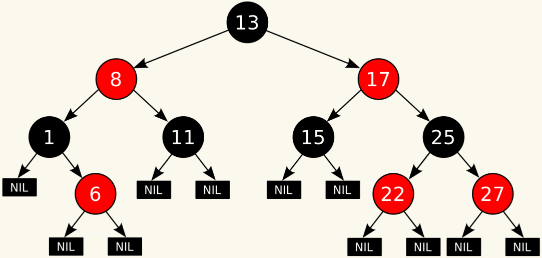
Definition (3)



- ***Balanced* Binary Search Tree**
- Depth of all leaves differs by at most 1
  - Puts a boundary on the worst case search operation

Binary trees

Definition (4)



- Linux implements **Red-Black Trees**
- Nodes: red or black
- Leaves: black, no data
- Non-leaves: two children
- Red nodes have two black children

- Path from one node to one of its leaves has same amount of black nodes as the shortest path to any of its other leaves
- Properties are maintained during tree modifications:
  - Red-black trees are **self-balanced**
    - Try to stay (semi-)balanced with modifications
    - Efficient insert operations

Binary trees

Tree creation and search

- Creation:

```
struct rb_root my_tree_root = RB_ROOT;
```

- Search routing must be implemented by the programmer:

```
static struct zswap_entry *zswap_rb_search(struct
      rb_root *root, pgoff_t offset)
{
  struct rb_node *node = root->rb_node;
  struct zswap_entry *entry;

  while (node) {
    entry = rb_entry(node, struct zswap_entry,
      rbnode);
    if (entry->offset > offset)
      node = node->rb_left;
    else if (entry->offset < offset)
      node = node->rb_right;
    else
      return entry;
  }
  return NULL;
}
```

```
struct zswap_entry {
  struct rb_node rbnode;
  pgoff_t offset;
  int refcount;
  unsigned int length;
  struct zswap_pool *pool;
  unsigned long handle;
};
```

- Use `rb_entry` to get the data structure from the corresponding indexing node

Binary trees

Insertion/deletion

```
static int zswap_rb_insert(struct rb_root *root, struct zswap_entry *entry,
      struct zswap_entry **dupentry)
{
  struct rb_node **link = &root->rb_node, *parent = NULL;
  struct zswap_entry *myentry;

  while (*link) {
    parent = *link;
    myentry = rb_entry(parent, struct zswap_entry, rbnode);
    if (myentry->offset > entry->offset)
      link = &(*link)->rb_left;
    else if (myentry->offset < entry->offset)
      link = &(*link)->rb_right;
    else {
      *dupentry = myentry;
      return -EEXIST;
    }
  }
  rb_link_node(&entry->rbnode, parent, link);
  rb_insert_color(&entry->rbnode, root);
  return 0;
}
```

- Deletion:
  - `rb_erase(struct rb_node *node, struct rb_root *root)`

Binary trees

Rbtrees: where are they used in the kernel?

- Rbtrees usage in the kernel:
  - Processes runqueues for the CFS (default) Linux scheduler
  - Indexing file (inode) fragments for the CEPH filesystem
  - Indexing memory areas in a process address space
  - etc.

Outline

The right data structure for the right problem

Linked lists:
- Sequential iteration over all data is needed
- There is an unknown number of elements

Queues:
- Useful with producer/consumer pattern
- When it's OK to work with a fixed size buffer

Maps:
- Need to map a unique integer to a pointer

Red-black trees:
- Large amount of data, efficient search

- Other data structures in the kernel:
  - Radix trees [2]
  - Bitmaps [1]
  - etc.

Bibliography I

[1] Bit arrays and bit operations in the linux kernel.
https://0xax.gitbooks.io/linux-insides/content/DataStructures/bitmap.html.
Accessed: 2017-02-07.

[2] Lwn - trees i: Radix trees.
https://lwn.net/Articles/175432/.
Accessed: 2017-02-07.