Linux Kernel Development (LKD)

Session 1

Loadable Kernel Modules (LKM)

Paulo Baltarejo Sousa
pbs@isep.ipp.pt

2017

CISTER - Research Centre in Real-Time & Embedded Computing Systems
isep Instituto Superior de Engenharia do Porto
INESCTEC

Disclaimer

Material and Slides

Some of the material/slides are adapted from various:

- Presentations found on the internet;
- Books;
- Web sites;
- ...

Outline

Basics

Extensibility

- Two mechanisms for **extensibility**:
  - **Loadable kernel modules** (LKMs):
    - But you can also add code to the Linux kernel while it is running. A chunk of code that you add in this way is called a LKMs.
    - These modules can do lots of things.
    - Also **allows us to study how kernel works**;
    - **No need to recompile the kernel** and then reboot;
    - But **inherently unsafe**: any “bug” can cause a system malfunction or complete crash.
- **Kernel development** (Next session):
  - If you want to add code to a Linux kernel, the most basic way to do that is to add some source files to the kernel source tree and recompile the kernel;

The simplest kernel module (I)

```
#include <linux/module.h> /* Needed by all modules */
#include <linux/kernel.h> /* Needed for KERN_INFO */

static int __init hello_init(void){
printk(KERN_INFO "S01-LKM: I am in the Linux kernel.\n");
return 0;
}
static void __exit hello_exit(void){
printk(KERN_INFO "S01-LKM: I am no longer in the Linux kernel.\n");
}
module_init(hello_init);
module_exit(hello_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("PBS");
MODULE_DESCRIPTION("The simplest kernel module ");
```

The simplest kernel module (II)

- This module defines two functions:
  - `hello_init`: **it is invoked when the module is loaded** into kernel;
  - `hello_exit`: **it is invoked when the module is removed** from kernel;
- Macros:
  - `module_init` specify which function is executed during at module insertion time;
  - `module_exit` specify which function is executed at module removal time;
  - `MODULE_LICENSE` macro is used to tell the kernel that this module bears a free license – without such a declaration, the kernel complains when the module is loaded
  - `MODULE_DESCRIPTION` macro is used to describe what the module does;
  - `MODULE_AUTHOR` declares the module's author.

`__init` & `__exit`

- These do not have any relevance in case we are using them for a **dynamically loadable modules**, but only when the same code gets **built into the kernel**.
- All functions marked with `__init` get placed inside the `init` section of the kernel image automatically during kernel compilation; and all functions marked with `__exit` are placed in the `exit` section of the kernel image.
- What is the benefit of this?
  - All functions in the `init` section are supposed **to be executed only once during bootup** (and not executed again till the next bootup);
  - All functions in the `exit` section are supposed **to be called during system shutdown**.

Kernel message logging

- The `printk` function behaves similarly to the standard C library function `printf`.
- There are eight macros defined in `linux/kernel.h`:
  - Each macro represents an integer in angle brackets. Integers range from 0 to 7, with smaller values representing higher priorities.

```
#define KERN_EMERG "<0>" /* system is unusable */
#define KERN_ALERT "<1>" /* action must be taken immediately */
#define KERN_CRIT "<2>" /* critical conditions */
#define KERN_ERR "<3>" /* error conditions */
#define KERN_WARNING "<4>" /* warning conditions */
#define KERN_NOTICE "<5>" /* normal but significant condition */
#define KERN_INFO "<6>" /* informational */
#define KERN_DEBUG "<7>" /* debug-level messages */
```

- All `printk` calls put their output into a (`log`) ring buffer;
  - The `syslog` daemon running in user-space picks them up and redirect them to `/var/log/syslog` (from Ubuntu 11.04).
- The `dmesg` command parses the ring buffer and dump it to standard output.

Function's return guidelines

- Typically, returns an integer:
  - **For an error**, it returns a **negative number**: a minus sign appended with a macro that is available through the kernel header `linux/errno.h`

```
...
#define EPERM 1 /* Operation not permitted */
#define ENOENT 2 /* No such file or directory */
#define ESRCH 3 /* No such process */
...
#define EAGAIN 11 /* Try again */
#define ENOMEM 12 /* Out of memory */
#define EACCES 13 /* Permission denied */
#define EFAULT 14 /* Bad address */
...
```

  - **For success**, **zero** is the most common return value, unless there is some additional information to be provided. In that case, a **positive value** is returned, the value indicating the information, such as the number of bytes transferred by the function.

Compiling Kernel Modules (I)

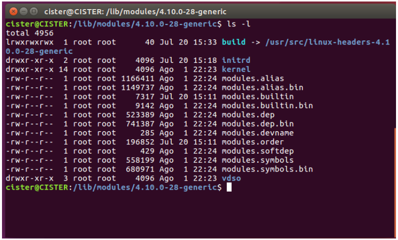
- To build a LKM, you need **to have the kernel source (or, at least, the kernel headers) installed on your system**.
  - The kernel source is assumed to be installed at `/usr/src/`.
- The command `uname -r` prints out the currently running kernel
- To compile the `hello.c` LKM
  - Create a `Makefile` in the same directory and type `make` in a terminal.

```
#Makefile
obj-m:=hello.o

all:
   make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

clean:
   make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

  - After the above successful compilation you will find a `.ko` file in the same directory where the compilation took place.
    - In this case, `hello.ko` file is the LKM.

Compiling Kernel Modules (II)

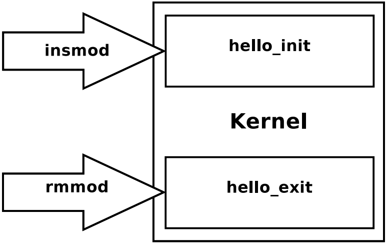
- In the `Makefile` **there is no reference to kernel source code directory, but it needs it**:
  - The reason for that is: the `/lib/modules/$(shell uname -r)/build` is a symbolic link that is linked:
    - To the kernel source code directory
    - To the kernel headers directory.



  - `-C <dir>` option instructs the `make` command to change to directory `<dir>` before reading the makefile.
  - `M=$(PWD)` tells to the compiler where is the module source code.

Loading and Unloading Modules

- To insert `hello` module into the kernel type the following command:
  - `> sudo insmod ./hello.ko`
- To remove `hello` module from the kernel type the following command:
  - `> sudo rmmod hello`



Modules info

- All modules loaded into the kernel are listed in `/proc/modules`. A list of all modules loaded in the kernel can be listed using the command:
  - `> lsmod`
  - Alternatively, you can cat the `/proc/modules` file to see all modules:
    - `> cat /proc/modules`
- Information on currently loaded modules can also be found in the sysfs virtual filesystem under `/sys/module`:
  - `> ls /sys/module`
- All messages printed by `printk` function can be listed using:
  - `> dmesg`

Developing LKM

Modules

- A module runs in kernel space, whereas applications run in user space.
    - This concept is at the base of operating systems theory.
- The role of a module is to extend kernel functionality

Notice

Most applications, with the notable exception of multithreading applications, typically run sequentially, from the beginning to the end, without any need to worry about what else might be happening to change their environment. Kernel code does not run in such a simple world, and even the simplest kernel modules must be written with the idea that many things can be happening at once.

Coding Modules

- Not all kernel source code is available for coding modules;
- Functions and variables have to be explicitly exported by the kernel to be visible to a module.
- Two macros are used to export functions and variables:
  - `EXPORT_SYMBOL(symbolname)`, which exports a function or variable to all modules;
  - `EXPORT_SYMBOL_GPL(symbolname)`, which exports a function or variable only to GPL modules.
- A normal driver should not need any non-exported function.

Example

A module can refer to the current process by accessing the `current`. The `current` points to the process that is currently executing.

```
printk(KERN_INFO "The process is [%s] [%i]
\n",current->comm, current->pid);
```

Modules' `init` and `exit` functions (I)

- At module's initialization function, every kernel module just registers itself in order to serve future requests, and its initialization function terminates immediately.
  - The task of the module's initialization function (`hello_init` in the example) is to prepare for later invocation of the module's functions; it's as though the module were saying, "Here I am, and this is what I can do."
- The module's exit function (`hello_exit` in the example) gets invoked just before the module is unloaded.
  - It should tell the kernel, "I'm not there anymore; don't ask me to do anything else".

Modules' `init` and `exit` functions (II)

- **The purpose of a module's entry and exit functions** is:
  - `init`:
    - Allocating memory, registering devices, etc.
  - `exit`:
    - Freeing memory, unregistering devices, etc.
- `hello_init(void)` and `hello_exit(void)` functions **have no argument**.
  - Shared data must be declared as global.

```c
#include <linux/module.h>
#include <linux/kernel.h>
#define BUF_SIZE 50 /*Number of bytes*/
char *buf; /*Global Variable*/

static int hello_init(void){
   printk(KERN_INFO "Hello world.\n");
   /*Memory allocation*/
   buf = kmalloc(BUF_SIZE, GFP_KERNEL);
   if (!buf)
      return -ENOMEM;
   return 0;
}
static void hello_exit(void){
   printk(KERN_INFO "Goodbye world.\n");
   if (buf){
      /*freeing memory*/
      kfree(buf);
   }
}
module_init(hello_init);
module_exit(hello_exit);
...
```

Working with /proc directory

`/proc` directory

- **It is a virtual filesystem**.
  - It is sometimes referred to as a process information pseudo-file system.
  - It does not contain 'real' files but runtime system information (e.g. system memory, devices mounted, hardware configuration, etc).
  - It can be regarded as a control and information centre for the kernel.
  - In fact, a lot of system utilities are simply calls to files in this directory.
    - For example,`lsmod` is the same as `cat /proc/modules` while `lspci` is a synonym for `cat /proc/pci`.

Create a /proc entry (I)

```
#include <linux/proc_fs.h>
#include <linux/sched.h>
#define ENTRY_NAME "hello"
struct proc_dir_entry *proc_entry = NULL;
...
static const struct file_operations proc_fops = {
   .owner = THIS_MODULE,
   .open = proc_open,
   .read = proc_read,
   .write = proc_write,
   .release = proc_close,
};
int hello_proc_init(void){
   proc_entry = proc_create(ENTRY_NAME,0, NULL, &proc_fops);
   if(proc_entry == NULL)
      return -ENOMEM;
   printk("S01-LKM:/proc/%s created\n", ENTRY_NAME);
   return 0;
}
void hello_proc_exit(void){
   remove_proc_entry(ENTRY_NAME, NULL);
   printk("S01-LKM:/proc/%s removed\n", ENTRY_NAME);
}
module_init(hello_proc_init);
module_exit(hello_proc_exit);
```

Create a /proc entry (II)

```
int proc_open(struct inode *inode, struct file *filp){
   printk(KERN_INFO "S01-LKM:%s:[%d] open\n",ENTRY_NAME, current->pid);
   return 0;
}
ssize_t proc_read(struct file *filp, char __user *buf, size_t count, loff_t *f_pos){
   printk(KERN_INFO "S01-LKM:%s:[%d] read\n",ENTRY_NAME, current->pid);
   return 0;
}
ssize_t proc_write(struct file *filp, const char *buf, size_t count, loff_t *f_pos){
   printk(KERN_INFO "S01-LKM:%s:[%d] write\n",ENTRY_NAME, current->pid);
   return count;
}
int proc_close(struct inode *inode, struct file *filp){
   printk(KERN_INFO "S01-LKM:%s:[%d] release\n",ENTRY_NAME, current->pid);
   return 0;
}
```

Manage /proc directory

- `proc_create`: creates a file in the /proc directory;

```
struct proc_dir_entry *proc_create(
   const char *name, /*The name of the proc entry*/
   umode_t mode, /*The access mode for proc entry*/
   struct proc_dir_entry *parent, /*The name of the parent directory under /
        proc*/
   const struct file_operations *proc_fops /*The structure in which the file
        operations for the proc entry will be created*/
)
```

- `remove_proc_entry`: removes a file from /proc directory;

```
void remove_proc_entry(
   const char *name, /*The name of the proc entry*/
   struct proc_dir_entry *parent /*The name of the parent directory under /proc
        */
)
```

file_operations structure (I)

- struct file_operations
  - It is a collection of function pointers.
  - Each open file is associated with its own set of functions;
- Fields:
  - struct module *owner
    - It is a pointer to the module that "owns" the structure.
    - This field is used to prevent the module from being unloaded while its operations are in use.
    - Almost all the time, it is simply initialized to THIS_MODULE.
  - ssize_t (*read) (struct file *, char __user *, size_t, loff_t *)
    - It is used to retrieve data from the kernel.
    - A non negative return value represents the number of bytes successfully read.
  - ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *)
    - It writes(or sends) data to the kernel.
    - The return value, if non-negative, represents the number of bytes successfully written.

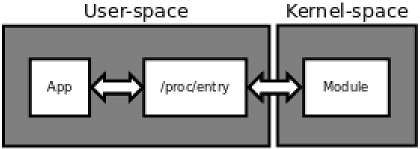
file_operations structure (II)

- Fields (continue):
  - `int (*open) (struct inode *, struct file *)`
    - This is always the first operation performed on the file structure.
  - `int (*release) (struct inode *, struct file *)`
    - This operation is invoked when the file structure is being released.
- Parameters:
  - `struct file`
    - It represents an open file.
    - It is created by the kernel on open and is passed to any function that operates on the file, until the last close.
    - After all instances of the file are closed, the kernel releases the data structure.
  - `inode`
    - It is used internally by the kernel to represent files.
  - `__user`
    - This is a form of documentation, noting that a pointer is a user-space address that cannot be directly dereferenced.
    - For normal compilation, `__user` has no effect, but it can be used by external checking software to find misuse of user-space addresses.

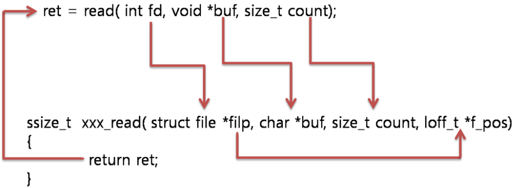
file_operations structure (III)

- Parameters (continue):
  - ssize_t and size_t
    - ssize_t data type is used to represent the sizes of blocks that can be read or written in a single operation. It is similar to size_t, but must be a signed type.
  - loff_t
    - The loff_t parameter is a "long offset" and is at least 64 bits wide even on 32-bit platforms;
    - The current reading or writing position.

Interacting with /proc/entry (I)



- cat /proc/entry.
  - The most common use of cat is to read the contents of files.

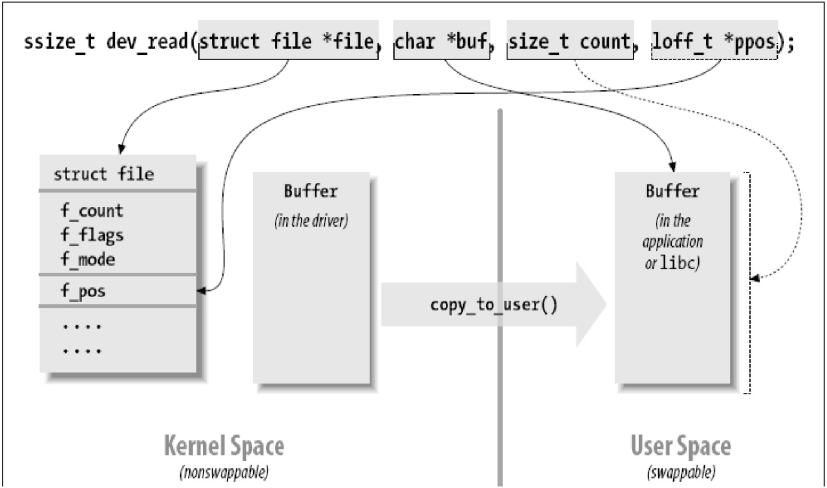


- echo "1" > /proc/entry
  - echo is a command that writes its arguments to standard output, however, the output can be redirect to a file by using ">".

Interacting with `/proc/entry` (II)

- Functions **to copy data to and from** user-space (defined in `asm/uaccess.h`)):
  - `unsigned long copy_to_user(void __user *to, const void *from, unsigned long count);`
  - `unsigned long copy_from_user(void *to, const void __user *from, unsigned long count);`
- The role of the two functions is not limited to copying data to and from user-space:
  - They also check whether the user space pointer is valid.
  - If the pointer is invalid, no copy is performed;
  - Return value is the amount of memory still to be copied or error codes.

Interacting with /proc/entry (III)



Advanced concepts

Memory Allocation

- The most important are the `kmalloc` (for allocation memory) and `kfree` (for freeing memory) functions.
- These functions, defined in `linux/slab.h`:
  - `void *kmalloc(size_t size, int flags);`
    - `size_t size`: is the size of the block to be allocated.
    - `int flags`: it controls the behavior of `kmalloc`. For instance, `GFP_KERNEL` means that the allocation is performed on behalf of a process running in kernel space. In other words, this means that the calling function is executing a system call on behalf of a process. Using `GFP_KERNEL` means that `kmalloc` can put the current process to sleep waiting for a page when called in low-memory situations.
  - `void kfree(void *ptr).`
    - Allocated memory should be freed with `kfree`.

Memory example

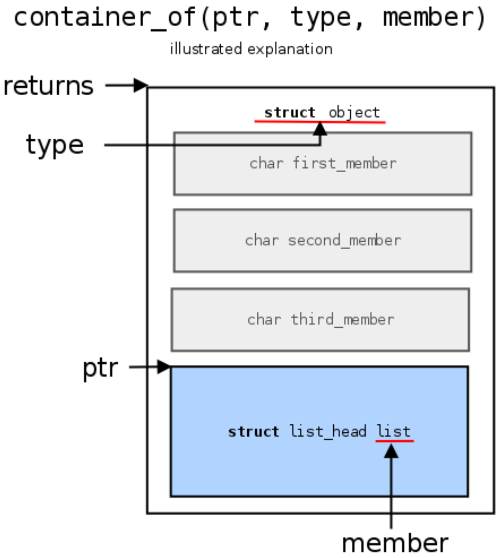
```
#include <linux/module.h>
#include <linux/kernel.h>
#define BUF_SIZE 50 /*Number of bytes*/
char *buf; /*Global Variable*/

static int hello_init(void){
   printk(KERN_INFO "Hello world.\n");
   /*Memory allocation*/
   buf = kmalloc(BUF_SIZE, GFP_KERNEL);
   if (!buf)
      return -ENOMEM;
   return 0;
}
static void hello_exit(void){
   printk(KERN_INFO "Goodbye world.\n");
   if (buf){
      /*freeing memory*/
      kfree(buf);
   }
}
module_init(hello_init);
module_exit(hello_exit);
...
```

container_of (I)

```
#define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)
/**
* 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 takes three arguments – a **pointer**, **type of the container**, and the **name of the member** the pointer refers to.
- The macro retrieves the address of the container which accommodates the respective member.

container_of (II)



Linked lists (I)

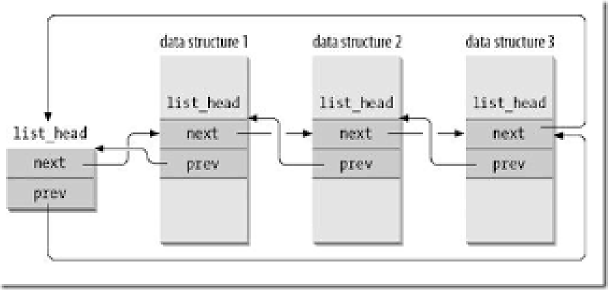
- The Linux kernel has a standard implementation of circular, doubly linked lists;
- To use the list mechanism, your module must include the file `<linux/list.h>`. This file defines a simple structure of type `list_head`:

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

- When working with the linked list interface, you should always bear in mind that the list functions **perform no locking**.

Linked lists (II)

- `struct list_head` field is embedded into a structure;
- Given the address of a list, you can iterate through the list elements, add and delete elements, and so on.



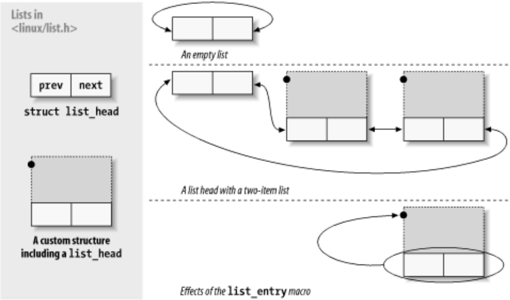
- `container_of` macro is used to get the address of the data structure element.

Linked list API (I)

- List heads must be initialized prior to use with the `INIT_LIST_HEAD` macro;
- `list_add(struct list_head *new, struct list_head *head)`
  - Adds the new entry immediately at the beginning of the list.
- `list_add_tail(struct list_head *new, struct list_head *head)`
  - Adds a new `entry` just before the given list `head`;
- `list_del(struct list_head *entry)`
  - Removes the `entry` from the list;
- `list_empty(struct list_head *head)`
  - Returns a nonzero value if the given list is empty.;
- `list_for_each(struct list_head *cursor, struct list_head *list)`
  - This macro creates a for loop that executes once with `cursor` pointing at each successive entry in the list.

Linked list API (II)

- `list_entry(struct list_head *ptr, type_of_struct, field_name)`
  - Returns a pointer to `type_of_struct` variable that embeds `field_name`, where `ptr` is a pointer to the `struct list_head` being used.



Linked list example

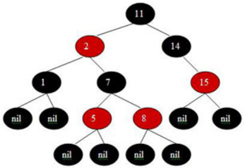
```
struct todo_struct {
struct list_head list;
int priority;
};

struct list_head todo_list;

INIT_LIST_HEAD(&todo_list);

void todo_add_entry(struct todo_struct *new)
{
struct list_head *ptr;
struct todo_struct *entry;
//for (ptr = todo_list.next; ptr != &todo_list; ptr = ptr->next) {
list_for_each(ptr, &todo_list) {
entry = list_entry(ptr, struct todo_struct, list);
if (entry->priority < new->priority) {
list_add_tail(&new->list, ptr);
return;
}
}
list_add_tail(&new->list, &todo_struct)
}
```

Tree concepts (I)

- A **tree** is a data structure that provides a hierarchical tree-like structure of data. Mathematically, it is an acyclic, connected, directed graph in which each vertex (called a **node**) has zero or more outgoing edges and zero or one incoming edges.
- A **binary tree** is a tree in which nodes have at most two outgoing edges – that is, a tree in which nodes have zero, one, or two children.
- A **binary search tree** is a binary tree with a specific ordering imposed on its nodes. The ordering is often defined via the following induction:
    - The left subtree of the root contains only nodes with values less than the root.
    - The right subtree of the root contains only nodes with values greater than the root.
    - All subtrees are also binary search trees.

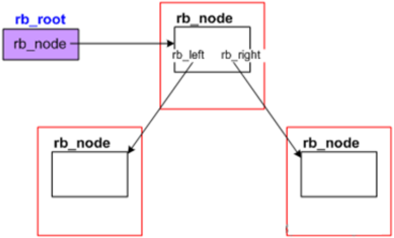
Tree concepts (II)

- The **depth** of a node is measured by how many parent nodes it is from the root. Nodes at the "bottom" of the tree – those with no children – are called **leaves**.
- The **height** of a tree is the depth of the deepest node in the tree.
- **A balanced binary search tree** is a binary search tree in which the depth of all leaves differs by at most one.
- **A self-balancing binary search tree** is a binary search tree that attempts, as part of its normal operations,to remain (semi) balanced.
- **A red-black tree** is a type of self-balancing binary search tree.



rbtree

- The Linux implementation of red-black tree is called `rbtree`.
- It is declared in `<linux/rbtree.h>`
- The root of an `rbtree` is represented by the `rb_root` structure.
- Each node of an `rbtree` is represented by the `rb_node` structure.



`rbtree` API (I)

- `struct rb_node *rb_first(struct rb_root *tree)`
  - Returns a pointer to the first node, if it exists, or `NULL`, otherwise;
- `struct rb_node *rb_last(struct rb_root *tree)`
  - Returns a pointer to the last node, if it exists, or `NULL`, otherwise;
- `struct rb_node *rb_next(struct rb_node *node)` and `struct rb_node *rb_prev(struct rb_node *node)`
  - Moving forward and backward through the tree is a simple matter of calling `rb_next` and`rb_prev`.
  - In both cases, a return value of `NULL` indicates that the requested node does not exist.

rbtree API (II)

- `void rb_link_node(struct rb_node *new_node, struct rb_node *parent, struct rb_node **link)`
  - Links the new node into the tree as a red node;
- `void rb_insert_color(struct rb_node *new_node, struct rb_root *tree)`
  - Rebalance the tree;
- `void rb_erase(struct rb_node *victim, struct rb_root *tree)`
  - Remove a node from a tree and if it is required rebalance it.
- `rb_entry(ptr, type_of_struct, field_name)`
  - Returns a pointer to `type_of_struct` variable that embeds `field_name`, where `ptr` is a pointer to the `struct rb_node` being used.

rbtree Example (I)

- Defining data structure:

```
struct node_item{
int id;
struct rb_node node;
};
```

- Creating the root of a rbtree:

```
\lstinputlisting{code/rbtree1.tex}
```

- Checking if there is any element into the tree:

```
if(RB_EMPTY_ROOT(root)){
// empty tree
}else{
//There is/are some nodes
}
```

rbtree Example (II)

- Inserting an item:

```
struct node_item * rb_insert_node_item(struct rb_root * root, int target){
    struct rb_node **n = &root->rb_node;
    struct rb_node *parent = NULL;
    struct rb_node * source = NULL;
    struct node_item * ans;
    while(*n){
        parent = *n;
        ans = rb_entry(parent, struct node_item, node);
        if(target < ans->id)
            n = &parent->rb_left;
        else if(target > ans->id)
            n = &parent->rb_right;
        else
            return ans;
    }
    source = ( struct node_item *)kmalloc(sizeof(struct struct node_item),
         GFP_KERNEL);
    source->id = target;
    rb_link_node(source, parent, n); //Insert this new node as a red leaf.
    rb_insert_color(source, root); //Rebalance the tree, finish inserting
    return NULL;
}
```

rbtree Example (III)

- Searching an item:

```
struct node_item * rb_search_node_item(struct rb_root * root, int target){
struct rb_node *n = root->rb_node;
struct node_item * ans;
while(n){
//Get the parent struct to obtain the data for comparison
ans = rb_entry(n, struct node_item, node);
if(target < ans->id)
    n = n->rb_left;
else if(target > ans->id)
    n = n->rb_right;
else
    return ans;
}
return NULL;
}
```

```
struct rb_node *n;
for (n = rb_first(&root); n;n = rb_next(n)){
ans = rb_entry(n, struct node_item, node);
...
}
```

rbtree Example (IV)

- Removing an item:

```
void rb_erase_node_item(struct rb_node * source, struct rb_root * root){

     struct node_item * target;
     target = rb_entry(source, struct node_item, node);
     rb_erase(source, root); //Erase the node
     kfree(target); //Free the memory
}
```

Concurrency

Race conditions and critical sections

- A **race condition** could occurs when a shared resource is accessed at the same time by two or more threads.
- Code paths that access and manipulate shared resource are called **critical regions** or **critical sections**.
- In the Linux system, there are numerous sources of concurrency and, therefore, possible race conditions;
  - Multiprocessing support implies that kernel code can simultaneously run on two or more processors;
  - Kernel code is preemptible, which means, the scheduler can preempt kernel code at virtually any point and reschedule another task;
  - Interrupts are asynchronous events

Notice

So the first rule of thumb to keep in mind is to avoid shared resources whenever possible. If there is no concurrent access, there is no race conditions.

Context switch

- A **context switch** (also sometimes referred to as a process switch or a task switch) is the switching of the CPU from one process or thread to another.
- Context switch occurs because of:
  - Internal events: system calls and exceptions (software interrupts);
  - External events: interrupts;
- Race conditions can be avoided by preventing context switch:
  - Eliminate internal events: disable preemption;
  - Eliminate external event: disable interrupts.

Preemption Disabling

- Because the **kernel is preemptive**, a process in the kernel can stop running at any instant to enable a process of higher priority to run.
  - This means a task can begin running in the same critical region as a task that was preempted.
- **It can be useful in per-processor variables**.
- kernel preemption can be disabled via `preempt_disable`.
  - The call is nestable; you may call it any number of times.
  - For each call, a corresponding call to `preempt_disable` is required.
  - The final corresponding call to `preempt_enable` re-enables preemption.
- Example:

```
preempt_disable();
/* preemption is disabled ... */
preempt_enable();
```

Interrupts Disabling

- Interrupts are signal that are sent across IRQ (Interrupt Request Line) by a hardware or software.
- Interrupts are used to let CPU knows that something needs its attention.
  - Once the CPU receives an interrupt Request, CPU will temporarily stop execution of running program and invoke a special program called Interrupt Handler;
  - After the interrupt is handled CPU resumes the interrupted program.
- **Disabling an interrupt forces the waiting for the completion of currently executing interrupt handler** (if any).
  - By disabling interrupts, it is guarantee that an interrupt handler will not preempt the executing thread.
- Example:

```
local_irq_disable();
/* interrupts are disabled .. */
local_irq_enable();
```

Deal with shared resources

- To prevent concurrent access during critical regions, the programmer must ensure that code executes `atomically`
  - Operations must complete without interruption as if the entire critical region were one indivisible instruction.
- Example: `i=7; i++ ;`
  - Race condition

Thread 1	Thread 2
`get i` (7)	`get i` (7)
`increment i` (7 -> 8)	—
—	`increment i` (7 -> 8)
`write back i` (8)	—
—	`write back i` (8)

  - Atomic operation

Thread 1	Thread 2
`increment & store i` (7 -> 8)	—
—	`increment & store i` (8 -> 9)

Deadlocks

- A **deadlock** is a condition involving one or more threads of execution and one or more shared resources, such that each thread waits for one of the resources, but all the resources are already held.
- The threads all wait for each other, but they never make any progress toward releasing the resources that they already hold.
- Therefore, none of the threads can continue, which results in a deadlock.

Thread 1	Thread 2
`acquire lock A`	`acquire lock B`
`try to acquire lock B`	`try to acquire lock A`
`wait for lock B`	`wait for lock A`

Prevention of deadlock

- Implement **lock ordering**.
  - Nested locks must always be obtained in the same order.
  - This prevents the deadly embrace deadlock. Document the lock ordering so others will follow it.
- **Prevent starvation**. Ask yourself, does this code always finish? *If* `foo` *does not occur, will* `bar` *wait forever?*
  - In computer science, **starvation** is a problem encountered in multitasking where a process is perpetually denied necessary resources. Without those resources, the program can never finish its task.
- Do not double acquire the same lock.
- Design for simplicity. Complexity in your locking scheme invites deadlocks.

Contention and Scalability

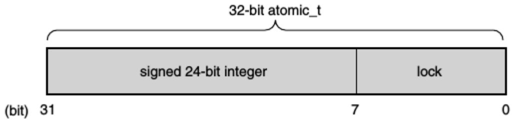
- **Lock contention**, occurs whenever one thread attempts to acquire a lock held by another thread.
  - High contention can occur because a lock is frequently obtained, held for a long time after it is obtained, or both.
  - A highly contended lock can become a bottleneck in the system, quickly limiting its performance.
- **Scalability** is a measurement of how well a system can be expanded.
  - Could be related to a large number of processes, a large number of processors, or large amounts of memory.

Rule

The more fine-grained the available locks, the less likely one process/thread will request a lock held by the other. (For example, locking a row rather than the entire table, or locking a cell rather than the entire row.)

atomic_t variables

- Sometimes, a shared resource is a simple integer value. Example: `i++`;
- The kernel provides an atomic integer type called `atomic_t`, defined in `<asm/atomic.h>`;
- An `atomic_t` holds an `int` value on all supported architectures.
  - Because of the way this type works on some processor architectures, however, the full integer range may not be available; thus, you should not count on an `atomic_t` holding more than 24 bits.



- `atomic_t` guarantees atomic operations

Atomic API

- `atomic_set(atomic_t *v, int i)`
  - Set the atomic variable `v` to the integer value `i`;
- `atomic_read(atomic_t *v)`
  - Return the current value of `v`.
- `atomic_add(int i, atomic_t *v)`
  - Add `i` to the atomic variable pointed to by `v`.
- `atomic_inc_and_test(atomic_t *v)`
  - Perform an increment and test the result; if, after the operation, the atomic value is `0`, then the return value is true; otherwise, it is false.
- `atomic_add_return(int i, atomic_t *v)`
  - Behave just like `atomic_add` with the exception that they return the new value of the atomic variable to the caller.
- `atomic_add_unless(atomic_t *v,int a, int u)`
  - Atomically adds `a` to `v`, so long as it was not `u`. Returns non-zero if `v` was not `u`, and zero otherwise.

Spinlocks

- A **spinlock** is a mutual exclusion component that can have only two values: “locked” and “unlocked”.
- Whenever a thread gets a spinlock:
  - If the lock is available, the “lock” value is set and the code continues into the critical section.
  - Otherwise, the code goes into a tight loop where it repeatedly checks the lock until it becomes available.
- Example:

```
spin_lock(&my_lock);
/* critical section */
spin_unlock(&my_lock);
```

Notice

The critical section protected by a spinlock is not allowed to sleep. So, be very careful not to call functions which can sleep!

Spinlock API (I)

- The “test and set" operation must be done in an atomic manner
  - Only one thread can obtain the lock, even if several are spinning at any given time.
  - kernel preemption is disabled when the kernel is in a critical region protected by a spinlock.
- It is defined in `<linux/spinlock.h>`;
- `spin_lock_init(spinlock_t *lock)`
  - Initializes the `lock` variable;
  - Initialize lock to 1 (unlocked).
- `spin_lock(spinlock_t *lock)`
  - Getting a lock;
  - Spin until lock becomes 1, then set to 0 (locked).
- `spin_unlock(spinlock_t *lock)`
  - Releasing a lock.
  - Set spin lock to 1 (unlocked).

Spinlock API (II)

- There are a few other spinlock operations:
  - `spin_lock_irqsave(spinlock_t *lock, unsigned long flags)`
    - Like `spin_lock`.
    - Also disables the interrupts on the local CPU, the previous interrupt state is stored in `flags`.
  - `spin_lock_irqrestore(spinlock_t *lock, unsigned long flags)`
    - Undoes `spin_lock_irqsave`. The `flags` argument passed to it must be the same variable passed to `spin_lock_irqsave`.
- There is also a set of nonblocking spinlock operations:
  - `int spin_trylock(spinlock_t *lock)`
    - Set lock to 0 if unlocked and return 1; return 0 if locked.
  - ...

Books and Useful links

Books

- *Linux Kernel Development: A thorough guide to the design and implementation of the Linux kernel, 3rd Edition*, Robert Love. Addison-Wesley Professional, 2010.
- *Professional Linux Kernel Architecture*, Wolfgang Mauerer. Wrox , 2008.
- *Linux Device Drivers, 3rd Edition*, Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman. O'Reilly, 2005.
- *Understanding the Linux Kernel, 3rd Edition*, Daniel P.Bovet, Marco Cesati, O'Reilly Media, 2005.

Links

- elixir.free-electrons.com/linux/v4.10/source
- www.kernel.org/doc/htmldocs/kernel-api/
- kernelnewbies.org/Documents
- lwn.net/Kernel/LDD3/