Loadable Kernel Modules (LKM)

Real-Time Operating Systems Programming (RTOSP)
Master in Critical Computing Systems Engineering (MCCSE)

2022/23

Paulo Baltarejo Sousa
pbs@isep.ipp.pt

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):
    - 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 **LKM**s.
    - 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 classes):
    - 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

```
#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 "LKM: I am in the Linux kernel.\n");
 return 0;
}
static void __exit hello_exit(void){
    printk(KERN_INFO "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 ");
```

- 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 (I)

- `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.

Macros (II)

- `__init` & `__exit`
  - These **do not have any relevance** in case we are using them for a **dynamically loadable modules**
  - These **do have relevance** only when the **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
    - This macro causes the init function to be discarded and its memory freed once the init function finishes for built-in drivers, but not loadable modules.
  - 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 critical levels** (descending by critical level):
  - `KERN_EMERG`: system is unusable
  - `KERN_ALERT`: action must be taken immediately
  - `KERN_CRIT` critical conditions
  - `KERN_ERR`: error conditions
  - `KERN_WARNING`: warning conditions
  - `KERN_NOTICE`: normal but significant condition
  - `KERN_INFO`: informational
  - `KERN_DEBUG`: debug-level messages
- All `printk` calls put their output into a (log) ring buffer;
- 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 `/include/uapi/asm-generic/errno-base.h`

```
#define EPERM  1 /* Operation not permitted */
#define ENOENT  2 /* No such file or directory */
#define ESRCH  3 /* No such process */
#define EINTR  4 /* Interrupted system call */
...
```

  - **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)

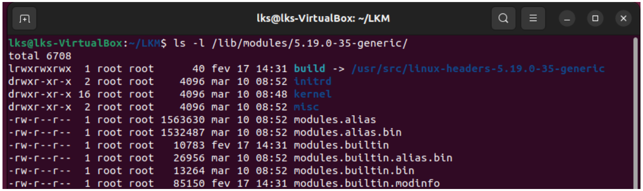
- To build a LKM, you need to have the **kernel source (or, at least, the kernel headers) installed on your system**.
  - The kernel source is assumed to be installed at `/usr/src/`.
- The command `uname -r` prints out the currently running kernel
- To compile the `hello.c` LKM
  - Create a `Makefile`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



lks@lks-VirtualBox: ~/LKM

```
lks@lks-VirtualBox:~/LKM$ ls -l /lib/modules/5.19.0-35-generic/
total 6708
lrwxrwxrwx  1 root root      40 fev 17 14:31 build -> /usr/src/linux-headers-5.19.0-35-generic
drwxr-xr-x  2 root root    4096 mar 10 08:52 initrd
drwxr-xr-x 16 root root    4096 mar 10 08:48 kernel
drwxr-xr-x  2 root root    4096 mar 10 08:52 misc
-rw-r--r--  1 root root 1563630 mar 10 08:52 modules.alias
-rw-r--r--  1 root root 1532487 mar 10 08:52 modules.alias.bin
-rw-r--r--  1 root root   10783 fev 17 14:31 modules.builtin
-rw-r--r--  1 root root   26956 mar 10 08:52 modules.builtin.alias.bin
-rw-r--r--  1 root root   13264 mar 10 08:52 modules.builtin.bin
-rw-r--r--  1 root root   85150 fev 17 14:31 modules.builtin.modinfo
```

  - `-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.

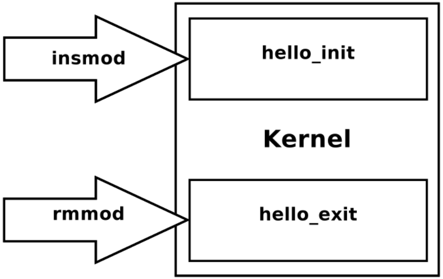
Loading and Unloading Modules

- **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

Module Features

- **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**.

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

- 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 **init** 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.
    - Typically, **it undoes what init function** has performed.
        - It should tell the kernel, "I'm not there anymore; don't ask me to do anything else".

Purpose of a Module's entry and exit functions

- **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**.

```
#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.
  - `lsmod` is the same as `cat /proc/modules`
  - `lspci` is a synonym for `cat /proc/pci`.

Create a /proc entry (I)

```
int proc_open(struct inode *inode, struct file *filp){
 printk(KERN_INFO "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 "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 "LKM: %s:[%d] write\n",ENTRY_NAME, current->pid);
 return count;
}
int proc_close(struct inode *inode, struct file *filp){
 printk(KERN_INFO "LKM: %s:[%d] close\n",ENTRY_NAME, current->pid);
 return 0;
}
```

Create a /proc entry (II)

```c
#define ENTRY_NAME "hello2"
struct proc_dir_entry *proc_entry = NULL;
...
static const struct proc_ops proc_ops = {
  .proc_open    = proc_open,
  .proc_read    = proc_read,
  .proc_write   = proc_write,
  .proc_release  = proc_close,
};
int hello_proc_init(void){
   proc_entry = proc_create(ENTRY_NAME,0, NULL, &proc_fops);
  if(proc_entry == NULL)
     return -ENOMEM;
   return 0;
}
```

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

```c
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 proc_ops *proc_ops /*The structure in which the file operations for
        the proc entry will be created*/
)
```

Remove a /proc entry

```
#define ENTRY_NAME "hello2"
...
void hello_proc_exit(void){
   remove_proc_entry(ENTRY_NAME, NULL);
}
```

- 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
*/
)
```

proc_ops structure

```c
struct proc_ops {
 unsigned int proc_flags;
 int (*proc_open)(struct inode *, struct file *);
 ssize_t (*proc_read)(struct file *, char __user *, size_t, loff_t *);
 ssize_t (*proc_read_iter)(struct kiocb *, struct iov_iter *);
 ssize_t (*proc_write)(struct file *, const char __user *, size_t, loff_t *);
 /* mandatory unless nonseekable_open() or equivalent is used */
 loff_t (*proc_lseek)(struct file *, loff_t, int);
 int (*proc_release)(struct inode *, struct file *);
 __poll_t (*proc_poll)(struct file *, struct poll_table_struct *);
 long (*proc_ioctl)(struct file *, unsigned int, unsigned long);
 #ifdef CONFIG_COMPAT
 long (*proc_compat_ioctl)(struct file *, unsigned int, unsigned long);
 #endif
 int (*proc_mmap)(struct file *, struct vm_area_struct *);
 unsigned long (*proc_get_unmapped_area)(struct file *, unsigned long, unsigned long,
        unsigned long, unsigned long);
} __randomize_layout;
```

- It is a **collection of function pointers**.
- Each open file is associated with its own set of functions.

proc_ops structure fields

- `int (*proc_open) (struct inode *, struct file *)`
    - This is always the first operation performed on the file structure.
- `int (*proc_release) (struct inode *, struct file *)`
    - This operation is invoked when the file structure is being released.
- `ssize_t (*proc_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 (*proc_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.

proc_ops structure parameters (I)

- `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.
- `struct 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.

proc_ops structure parameters (II)

- 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
  - Theloff_t parameter is a "long offset" and is at least 64 bits wide even on 32-bit platforms;
  - The current reading or writing position.

Everything is a File: Working with Files

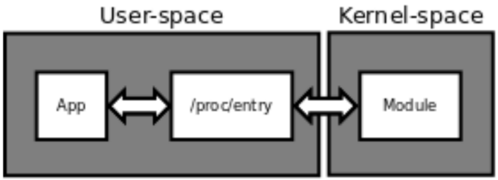
- Reading the content of a file

```
#include<stdio.h>
#include<fcntl.h>
#include<unistd.h>
#define BUF_LEN 50
int main(){
  char buf[BUF_LENGTH];
  int ret;
  int fd = open("foo.txt", O_RDONLY, 0);
  while(ret = read(fd, buf, BUF_LEN-1)){
   buf[ret]='\0';
   printf("%s", buf);
  }
  close(fd);
  return 0;
}
```

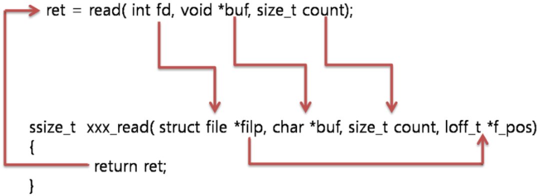
- Writing to a file

```
#include<stdio.h>
#include<string.h>
#include<unistd.h>
#include<fcntl.h>

#define BUF_LENGTH 50
int main(){

  char buf[BUF_LENGTH] = "hello world";
  int ret;
  int fd = open("foo.txt", O_WRONLY |
       O_CREAT | O_TRUNC, 0644);
  ret = write(fd, buf, strlen(buf));
  close(fd);
  return 0;
}
```

Interacting with /proc entry (I)



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



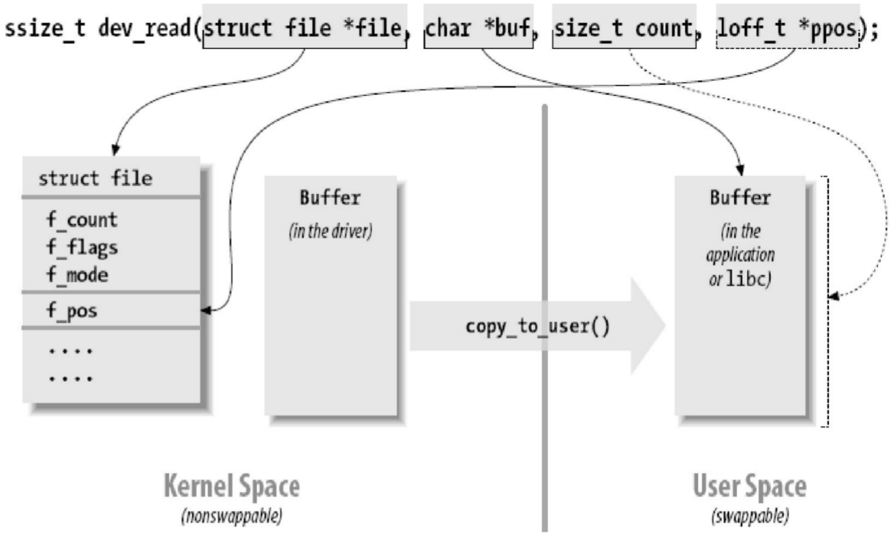
```
ret = read( int fd, void *buf, size_t count);

ssize_t  xxx_read( struct file *filp, char *buf, size_t count, loff_t *f_pos)
{
        return ret;
}
```

- `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 >.

Copy data from/to kernel (I)

- Functions **to copy data to and from user-space**:
  - `unsigned long copy_to_user(void __user *to, const void *from, unsigned long count);`
    - It copies the `count` bytes pointed at by `from`, which must exist in kernel-space, to `to`, which must exist in user-space.
    - It returns the number of bytes not copied, which means on success it returns zero.
  - `unsigned long copy_from_user(void *to, const void __user *from, unsigned long count);`
- 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**.
- In order to use these functions, you have to prefix with `raw_`.
  - `raw_copy_to_user(...)`
  - `raw_copy_from_user(...)`
    - Assembly somewhat optimized copy routines

Copy data from/to kernel (II)



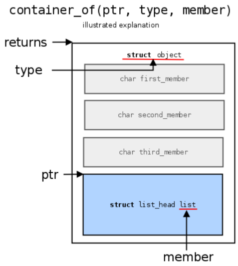
Advanced concepts

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.

Magical Macro: `container_of`

- **Cast a member of a structure out to the containing structure**



  - 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**.

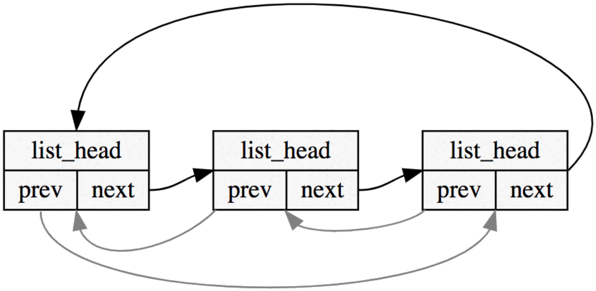
```
struct type1 {
  char member1;
  int member2;
  long member3;
  char member4[20];
  int member5;
};

struct type1 obj1={
'a',50,300,"LKS",100
};
struct type1 *ptr=NULL;

ptr = container_of(
  &obj1.member3,
  struct type1,
  member3);

printf("%c:%d:%lu:%s:%d",
  ptr->member1,
  ptr->member2,
  ptr->member3,
  ptr->member4,
  ptr->member5);
```

Linked List (I)

- The Linux kernel has a standard implementation of **circular, doubly linked lists**;

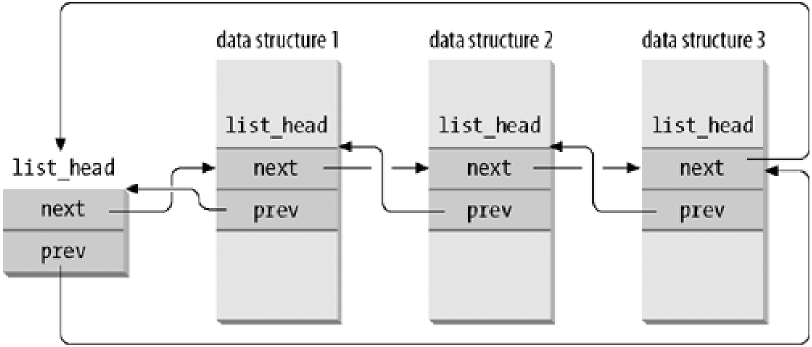
```
struct list_head {
  struct list_head *next, *prev;
};
```

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



Linked List (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 [1] (I)

```
struct queue_item{
 char buffer[BUFFER_LEN];
 struct list_head node;
};
struct list_head head;
INIT_LIST_HEAD(&head);

if(list_empty(&head)){
    ...
}
struct queue_item *node_queue = kmalloc(sizeof(struct queue_item),GFP_KERNEL);
...
list_add_tail(&node_queue->node,&head);

struct queue_item *node_queue = list_first_entry(&head,struct queue_item, node);
...
list_del(&node_queue->node);

struct list_head *ptr;
struct queue_item *node_queue;
list_for_each(ptr,&head){//for (ptr=head.next;ptr!=&head; ptr=ptr->next)
    node_queue = list_entry(ptr, struct queue_item, node);
}
```

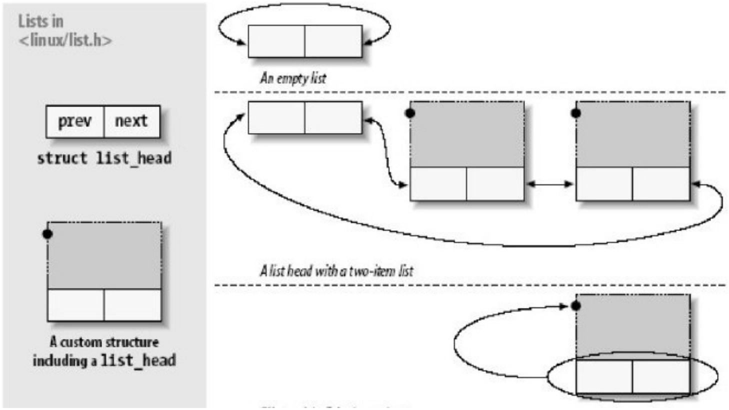
[1] /include/linux/list.h

Linked list API (II)

- 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 (III)

- `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.

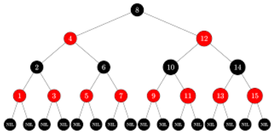

Lists in <linux/list.h>
prev
next
struct list_head
A custom structure including a list_head
An empty list
A list head with a two-item list

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.

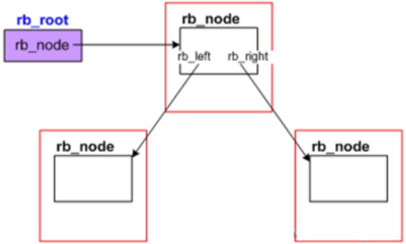
Tree concepts (II)

- 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` [2]

- The Linux implementation of **red-black tree is called `rbtree`**.
- The root of an `rbtree` is represented by the `rb_root` structure.
- Each node of an `rbtree` is represented by the `rb_node` structure.



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

[2] /include/linux/rbtree.h

rbtree API (I)

- Defining data structure:

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

- Creating the root of a rbtree:

```
struct rb_root root = RB_ROOT;
```

- Checking if there is any element into the tree:

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

rbtree API (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;
   //Insert this new node as a red leaf.
   rb_link_node(source, parent, n);
    //Rebalance the tree, finish inserting
   rb_insert_color(source, root);
   return source;
}
```

rbtree API (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;
}
```

`rbtree` API (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
}
```

- Iterating over tree

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

`rbtree` API (V)

- `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.
- `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` API (VI)

- `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.

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, exceptions (software interrupts) and others;
    - The process exits.
    - The process uses up its time slice.
    - The process requires another resource that is not currently available or needs to wait for I/O to complete.
    - The process relinquishes the CPU using a semaphore or similar system call.
  - **External events**: interrupts;
- **Race conditions can be avoided by preventing context switch**:
  - Eliminate internal events: disable preemption;
  - Eliminate external event: disable interrupts.

Preemption Enabling and 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`.
- kernel preemption can be enabled via `preempt_enable`.
- Example:

```
void kick_process(struct task_struct *p){
 int cpu;
 preempt_disable();
 cpu = task_cpu(p);
 if ((cpu != smp_processor_id()) && task_curr(p))
   smp_send_reschedule(cpu);
 preempt_enable();
}
EXPORT_SYMBOL_GPL(kick_process);
```

Interrupts Enabling and Disabling (I)

- **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.

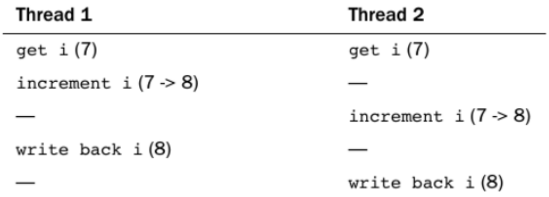
Interrupts Enabling and Disabling (II)

- Kernel interrupt can be disabled via `local_irq_disable`.
- Kernel interrupt can be enabled via `local_irq_enable`.
- Example:

```c
static int migration_cpu_stop(void *data)
{
 struct migration_arg *arg = data;
 struct task_struct *p = arg->task;
 struct rq *rq = this_rq();
 struct rq_flags rf;
 /*
  * The original target CPU might have gone down and we might
  * be on another CPU but it doesn't matter.
  */
local_irq_disable();
 ...

local_irq_enable();
 return 0;
}
```

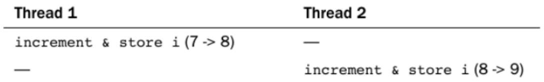
Deal with shared resources

- 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)

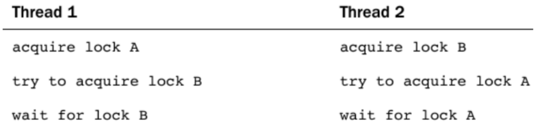
- Atomic operation



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.
- 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

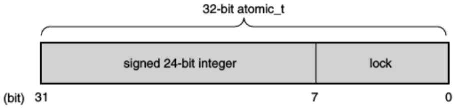
- **Lock contention, occurs whenever one thread attempts to acquire a lock held by another thread**.
  - High contention can occur because a lock is frequently obtained, held for a long time after it is obtained, or both.
  - A highly contended lock can become a bottleneck in the system, quickly limiting 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

- Atomic operations **provide instructions that execute atomically without interruption**.
- Sometimes, a shared resource is a simple integer value.
- Example: i++;
- The kernel provides an atomic integer type called `atomic_t`
- 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 [3]

- `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.

[3] for x86 architectures `tools/arch/x86/include/asm/atomic.h`

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.
- 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.

Notice

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

Spinlocks API [4](I)

```
spinlock_t lock;

spin_lock_init(&lock);

spin_lock(&lock);
  ...
spin_unlock(&lock);
```

- `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).
-

[4] /include/linux/spinlock.h

Spinlocks API(II)

- `spin_lock_irqsave(spinloc_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`.
- `int spin_trylock(spinlock_t *lock)`
  - Nonblocking spinlock.
  - Set lock to 0 if unlocked and return 1; return 0 if locked.