Introduction to Linux Kernel Programming

Luca Abeni, Claudio Scordino

The Kernel Source Tree

- About 16,000 source files
- Main directories in the kernel source:

`arch/`	Architecture-specific code
`Documentation/`	Kernel source documentation
`drivers/`	Device drivers
`fs/`	File systems
`include/`	Kernel headers
`kernel/`	**Core**
`net/`	Networking

Differences wrt normal user-space applications

- Not a single entry point: a different entry point for any type of interrupt recognized by the kernel
- No memory protection
  - No control over illegal memory access
- Synchronization and concurrency are major concerns
  - Susceptible to race conditions on shared resources!
  - Use spinlocks and semaphores.
- No libraries to link to
  - Never include the usual header files, like `<stdio.h>`
- A fault can crash the whole system
- No debuggers
- Small stack: 4 or 8 KB
  - Do not use large variables
  - Allocate large structures at runtime (`kmalloc`)
- No floating point arithmetic

Programming language

- Like all Unix-like OSs, Linux is coded mostly in C
- No access to the C library
  - No `printf`: use `printk`:

```
printk(KERN_ERR "This is an error!");
```

- Not coded in ANSI C
  - Both ISO C99 and GNU C extensions used
  - 64-bit `long long` data type
  - Inline functions to reduce overhead:

```
static inline void foo (...);
```

  - Branch annotation:

```
if (likely(pippo)) {
/*...*/
}
```

Programming language

- Like all Unix-like OSs, Linux is coded mostly in C
- No access to the C library
  - No `printf`: use `printk`:

```
printk(KERN_ERR "This is an error!");
```

- Not coded in ANSI C
  - Both ISO C99 and GNU C extensions used
  - 64-bit `long long` data type
  - Inline functions to reduce overhead:

```
static inline void foo (...);
```

  - Branch annotation:

```
if (likely(pippo)) {
/*...*/
}
```

Programming language (2)

- Few small critical functions coded in Assembly (around 10% of the code)
  - Architecture-dependent code placed in `linux/arch`
  - The symbolic link `linux/include/asm` identifies all architecture-dependent header files
  - Inline assembly (`asm` primitive)

Loadable Kernel Modules

- Linux provides the ability of inserting (and removing) services provided by the kernel at runtime
- Every piece of code that can be dynamically loaded (and unloaded) is called **Kernel Module**

Loadable Kernel Modules (2)

- A kernel module provides a new service (or services) available to users
- Event-driven programming:
  - Once inserted, a module just registers itself in order to serve future requests
  - The initialization function terminates immediately
- Once a module is loaded and the new service registered
  - The service can be used by all the processes, as long as the module is in memory
  - The module can access all the kernel's public symbols
- After unloading a module, the service is no longer available
- In the 2.6 series, modules have extensions `.ko`

Loadable Kernel Modules (3)

- The kernel core must be self-contained. Everything else can be written as a kernel module
- A kernel module is desirable for:
  - Device drivers
  - Filesystems
  - Network protocols
- Modules can only use **exported** functions (a collection of functions available to kernel developers). The function must already be part of the kernel at the time it is invoked.
- A module can export symbols through the following macros:
  - `EXPORT_SYMBOL(name);`
  - `EXPORT_SYMBOL_GPL(name);`
    makes the symbol available only to GPL-licensed modules

Loadable Kernel Modules (3)

- The kernel core must be self-contained. Everything else can be written as a kernel module
- A kernel module is desirable for:
  - Device drivers
  - Filesystems
  - Network protocols
- Modules can only use **exported** functions (a collection of functions available to kernel developers). The function must already be part of the kernel at the time it is invoked.
- A module can export symbols through the following macros:
  - `EXPORT_SYMBOL(name);`
  - `EXPORT_SYMBOL_GPL(name);`
    makes the symbol available only to GPL-licensed modules

Why using kernel modules

- Not all kernel services of features are required every time into the kernel: a module can be loaded only when it is necessary, saving memory
- Easier development: kernel modules can be loaded and unloaded several times, allowing to test and debug the code without rebooting the machine.

How to write a kernel module

Ways to write a kernel module:

1. Insert the code into the Linux kernel main source tree
   - Modify the `Kconfig` and the main `Makefile`
   - Create a patch for each new kernel version
2. Write the code in a separate directory, without modifying any file in the main source tree
   - More flexible
   - In the 2.6 series, the modules are linked against object files in the main source tree:
     $\Rightarrow$ The kernel must be already configured and compiled

How to write a kernel module

Ways to write a kernel module:

1. Insert the code into the Linux kernel main source tree
   - Modify the `Kconfig` and the main `Makefile`
   - Create a patch for each new kernel version

2. Write the code in a separate directory, without modifying any file in the main source tree
   - More flexible
   - In the 2.6 series, the modules are linked against object files in the main source tree:
     $\Rightarrow$ The kernel must be already configured and compiled

Loading/unloading a module

- Only the superuser can load and unload modules
- `insmod` inserts a module and its data into the kernel.
- The kernel function `sys_init_module`:
  1. Allocates (through `vmalloc`) memory to hold the module
  2. Copies the module into that memory region
  3. Resolves kernel references in the module via the kernel symbol table (works like the linker `ld`)
  4. Calls the module's initialization function
- `modprobe` works as `insmod`, but it also checks module dependencies. It can only load a module contained in the `/lib/modules/` directory
- `rmmod` removes a loaded module and all its services
- `lsmod` lists modules currently loaded in the kernel
  - Works through `/proc/modules`

The Makefile

- The `Makefile` uses the extended GNU *make* syntax
- Structure of the `Makefile`:

```
## Name of the module:
obj-m = mymodule.o

## Source files:
example-objs = file1.o file2.o
```

- Command line:

```
make -C kernel_dir M=‘pwd‘ modules
```

Example 1: the include part

- We now see how to write a simple module that writes "Hello World" at module insertion/removal
- For a simple module we need to include at least the following

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
```

that define some essential macros and function prototypes.

Example 1: the init function

- Function called when the module is inserted:

```
static int __init hello_init(void)
{
   printk(KERN_ALERT "Hello world!\n");
   return 0;
}

module_init(hello_init);
```

- The function is defined `static` because it shouldn't be visible outside of the file
- The `__init` token tells the kernel that the function can be dropped after the module is loaded
  - Similar tag for data: `__initdata`
- The `module_init` macro specifies which function must be called when the module is inserted

Example 1: the cleanup function

- The unregister function must remove all the resources allocated by the init function so that the module can be safely unloaded

```
static void __exit hello_exit(void)
{
    printk(KERN_ALERT "Goodbye, cruel world!\n");
}

module_exit(hello_exit);
```

- The `__exit` token tells the compiler that the function will be called only during the unloading stage (the compiler puts this function in a special section of the ELF file)
- The `module_exit` macro specifies which function must be called when the module is removed
- It **must** release any resource and undo everything the *init* function built up
- If it is not defined, the kernel does not allow module unloading

Other information

- Some other information should be specified:
  - `MODULE_AUTHOR("Claudio Scordino");`
  - `MODULE_DESCRIPTION("Kernel Development Example");`
  - `MODULE_VERSION("1.0");`
- License:
  - `MODULE_LICENSE("GPL");`
  - The kernel accepts also `"GPL v2"`, `"GPL and additional rights"`, `"Dual BSD/GPL"`, `"Dual MPL/GPL"` and `"Proprietary"`
- Convention: put all information at the end of the file

Module parameters

- Both `insmod` and `modprobe` accept parameters given at loading time
- Require `#include <linux/moduleparam.h>`
- A module parameter is defined through a macro:

```
static int myvar = 13;
module_param(myvar, int, SIRUGO);
```

  - All parameters should be given a default value
  - The last argument is a permission bit-mask (see `linux/stat.h`)
  - The macro should be placed outside of any function

Module parameters (2)

- Supported types: `bool`, `charp`, `int`, `long`, `short`, `uint`, `ulong`, `ushort`
- A module "mod" can be loaded assigning a value to the parameter myvar by doing:
  `insmod mod myvar=27`
- Another macro allows to accept array parameters:
  `module_param_array(name, type, num, permission);`
  - The module loader refuses to accept more values than will fit in the array

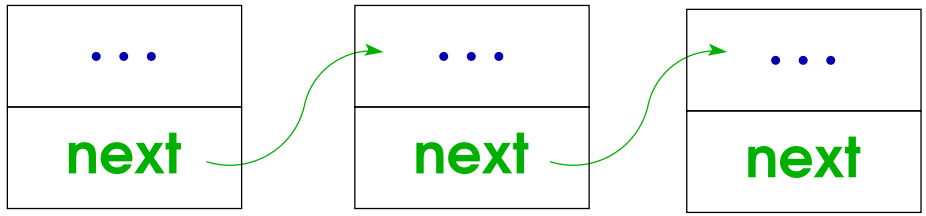
Example: Kernel Linked Lists

- Data structure that stores a certain amount of **nodes**
- The nodes can be dynamically created, added and removed at runtime
  - Number of nodes unknown at compile time
  - Different from array
- For this reason, the nodes are linked together
  - Each node contains at least one pointer to another element

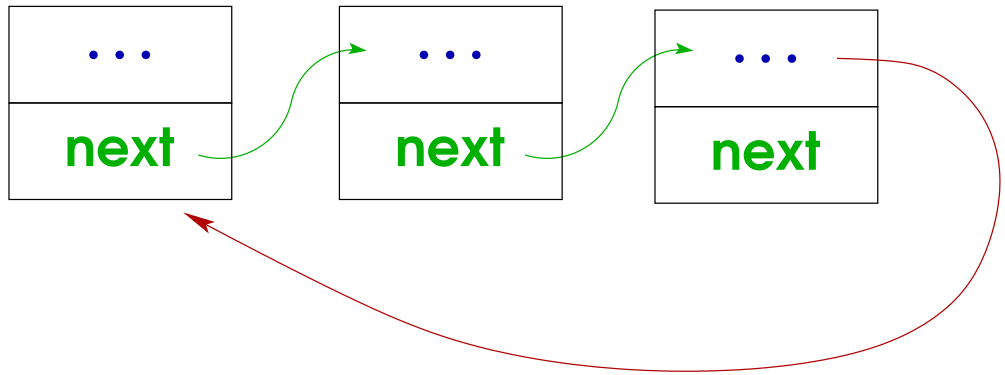
Singly linked lists

```
struct list_element {
    int data;
    struct list_element *next;
};
```

- Singly linked list:



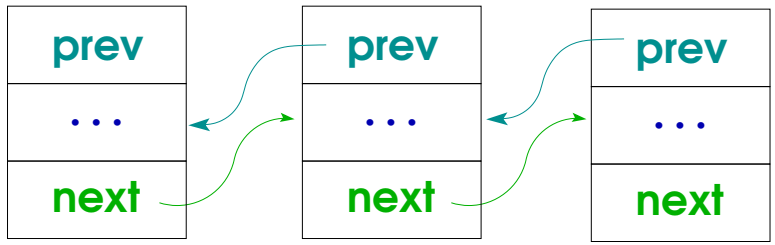
- Circular singly linked list:



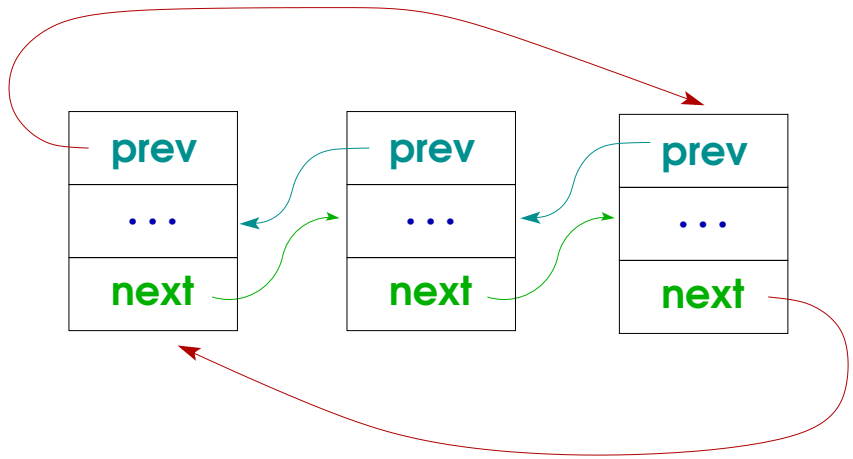
Doubly linked lists

```
struct list_element {
    int data;
    struct list_element *next;
    struct list_element *prev;
};
```

- Doubly linked list:



- Circular doubly linked list:



Kernel's linked list implementation

- Circular doubly linked list
- No head pointer: does not matter where you start...
  - All individual nodes are called *list heads*
- Declared in `linux/list.h`
- Data structure:

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

- No locking: your responsibility to implement a locking scheme

Defining linked lists

1. Include the `list.h` file:

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

2. Embed a `list_head` inside your structure:

```
struct my_node {
    struct list_head klist;
    /* Data */
};
```

3. Define a variable to access the list:

```
struct list_head my_list;
```

4. Initialize the list:

```
INIT_LIST_HEAD(&my_list);
```

Defining linked lists

1. Include the `list.h` file:

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

2. Embed a `list_head` inside your structure:

```
struct my_node {
    struct list_head klist;
    /* Data */
};
```

3. Define a variable to access the list:

```
struct list_head my_list;
```

4. Initialize the list:

```
INIT_LIST_HEAD(&my_list);
```

Defining linked lists

1. Include the `list.h` file:

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

2. Embed a `list_head` inside your structure:

```
struct my_node {
    struct list_head klist;
    /* Data */
};
```

3. Define a variable to access the list:

```
struct list_head my_list;
```

4. Initialize the list:

```
INIT_LIST_HEAD(&my_list);
```

Defining linked lists

1. Include the `list.h` file:

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

2. Embed a `list_head` inside your structure:

```
struct my_node {
    struct list_head klist;
    /* Data */
};
```

3. Define a variable to access the list:

```
struct list_head my_list;
```

4. Initialize the list:

```
INIT_LIST_HEAD(&my_list);
```

Using linked lists

- Add a new node after the given list head:

```
struct my_node *q = kmalloc(sizeof(my_node));
list_add (&(q->klist), &my_list);
```

- Remove a node:

```
list_head *to_remove = q->klist;
list_del (&to_remove);
```

- Traversing the list:

```
list_head *g;
list_for_each (g, &my_list) {
    /* g points to a klist field inside
     * the next my_node structure */
}
```

- Knowing the structure containing a `klist* h`:

```
struct my_node *f = list_entry(h, struct my_node, klist);
```

Using linked lists

- Add a new node after the given list head:

```
struct my_node *q = kmalloc(sizeof(my_node));
list_add (&(q->klist), &my_list);
```

- Remove a node:

```
list_head *to_remove = q->klist;
list_del (&to_remove);
```

- Traversing the list:

```
list_head *g;
list_for_each (g, &my_list) {
    /* g points to a klist field inside
     * the next my_node structure */
}
```

- Knowing the structure containing a `klist* h`:

```
struct my_node *f = list_entry(h, struct my_node, klist);
```

Using linked lists

- Add a new node after the given list head:

```
struct my_node *q = kmalloc(sizeof(my_node));
list_add (&(q->klist), &my_list);
```

- Remove a node:

```
list_head *to_remove = q->klist;
list_del (&to_remove);
```

- Traversing the list:

```
list_head *g;
list_for_each (g, &my_list) {
    /* g points to a klist field inside
     * the next my_node structure */
}
```

- Knowing the structure containing a `klist* h`:

```
struct my_node *f = list_entry(h, struct my_node, klist);
```

Using linked lists

- Add a new node after the given list head:

```
struct my_node *q = kmalloc(sizeof(my_node));
list_add (&(q->klist), &my_list);
```

- Remove a node:

```
list_head *to_remove = q->klist;
list_del (&to_remove);
```

- Traversing the list:

```
list_head *g;
list_for_each (g, &my_list) {
    /* g points to a klist field inside
     * the next my_node structure */
}
```

- Knowing the structure containing a `klist* h`:

```
struct my_node *f = list_entry(h, struct my_node, klist);
```

Using linked lists: Example

How to remove from the linked list the node having value 7:

```
struct my_node {
    struct list_head klist;
    int value;
};

struct list_head my_list;

struct list_head *h;
list_for_each_safe(h, &my_list)
    if ((list_entry(h, struct my_node, klist))->value == 7)
        list_del(h);
```

Using linked lists (3)

- Add a new node after the given list head: `list_add_tail();`
- Delete a node and reinitialize it: `list_del_init();`
- Move one node from one list to another: `list_move();`, `list_move_tail();`
- Check if a list is empty: `list_empty();`
- Join two lists: `list_splice();`
- Iterate without prefetching: `__list_for_each();`
- Iterate backward: `list_for_each_prev();`
- If your loop may delete nodes in the list: `list_for_each_safe();`