Rights to copy

3. Developing Kernel Modules

Advantages of modules

- Modules make it easy to develop drivers without rebooting: load, test, unload, rebuild, load...
- Useful to keep the kernel image size to the minimum (essential in GNU/Linux distributions for PCs).
- Also useful to reduce boot time: you don't spend time initializing devices and kernel features that you only need later.
- Caution: once loaded, have full control and privileges in the system. No particular protection. That's why only the root user can load and unload modules.

Module dependencies

- Some kernel modules can depend on other modules, which need to be loaded first.
- Example: the ubifs module depends on the ubi and mtd modules.
- Dependencies are described both in
  - /lib/modules/<kernel-version>/modules.dep
  - /lib/modules/<kernel-version>/modules.dep.bin
  - These files are generated when you run make modules_install.

Kernel log

- When a new module is loaded, related information is available in the kernel log.
  - The kernel keeps its messages in a circular buffer (so that it doesn't consume more memory with many messages)
  - Kernel log messages are available through the dmesg command (diagnostic message)
  - Kernel log messages are also displayed in the system console

```
$ sudo insmod ./intr_monitor.ko
insmod: error inserting ./intr_monitor.ko: -1 Device or resource busy
$ dmesg
[17549774.552000] Failed to register handler for irq channel 2
```

  - console messages can be filtered by level using the loglevel kernel command line parameter, or completely disabled with the quiet parameter.
    - Example: console=ttyS0 root=/dev/mmcblk0p2 loglevel=5
  - Note that you can write to the kernel log from user space too:
    - echo "<n>Debug info" > /dev/kmsg

Module Utilities

- `modinfo <module_name> (for modules in /lib/modules),`
- `modinfo <module_path>.ko`
  - Gets information about a module without loading it: parameters, license, description and dependencies.
  - useful before deciding to load a module or not.

- `sudo insmod <module_path>.ko`
  - Tries to load the given module. The full path to the module object file must be given.

- `sudo modprobe <module_name>`
  - Most common usage of modprobe: tries to load all the modules the given module depends on, and then this module.
  - Lots of other options are available. modprobe automatically looks in /lib/modules/<version>/ for the object file corresponding to the given module name.

Module Utilities

- `lsmod`
  - Displays the list of loaded modules
  - Compare its output with the contents of /proc/modules!

- `sudo rmmod <module_name>`
  - Tries to remove the given module.
  - Will only be allowed if the module is no longer in use (for example, no more processes opening a device file)

- `sudo modprobe -r <module_name>`
  - Tries to remove the given module and all dependent modules (which are no longer needed after removing the module)

Passing parameters to modules

- Find available parameters:
  - modinfo usb-storage
- Through insmod:
  - sudo insmod ./usb-storage.ko delay_use=0
- Through modprobe:
  - Set parameters in /etc/modprobe.conf or in any file in /etc/modprobe.d/
    - options usb-storage delay_use=0
- Through the kernel command line, when the driver is built statically into the kernel:
  - usb-storage.delay_use=0
    - usb-storage: *driver name*
    - delay_use: *driver parameter name*. It specifies a delay before accessing a USB storage device (useful for rotating devices).
    - 0: *driver parameter value*

Check module parameter values

- How to find the current values for the parameters of a loaded module?
  - Check /sys/module/<name>/parameters.
  - There is one file per parameter, containing the parameter value.

Hello Module

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

static int __init hello_init(void)
{
  pr_alert("Good morrow to this fair assembly.\n");
  return 0;
}

static void __exit hello_exit(void)
{
  pr_alert("Alas, poor world, what treasure hast thou lost!\n");
}

module_init(hello_init);
module_exit(hello_exit);
MODULE_LICENSE("GPL");
MODULE_DESCRIPTION("Greeting module");
MODULE_AUTHOR("William Shakespeare");
```

Headers specfic to the Linux kernel: linux/xxx.h
- No access to the usual C library, we're doing kernel programming

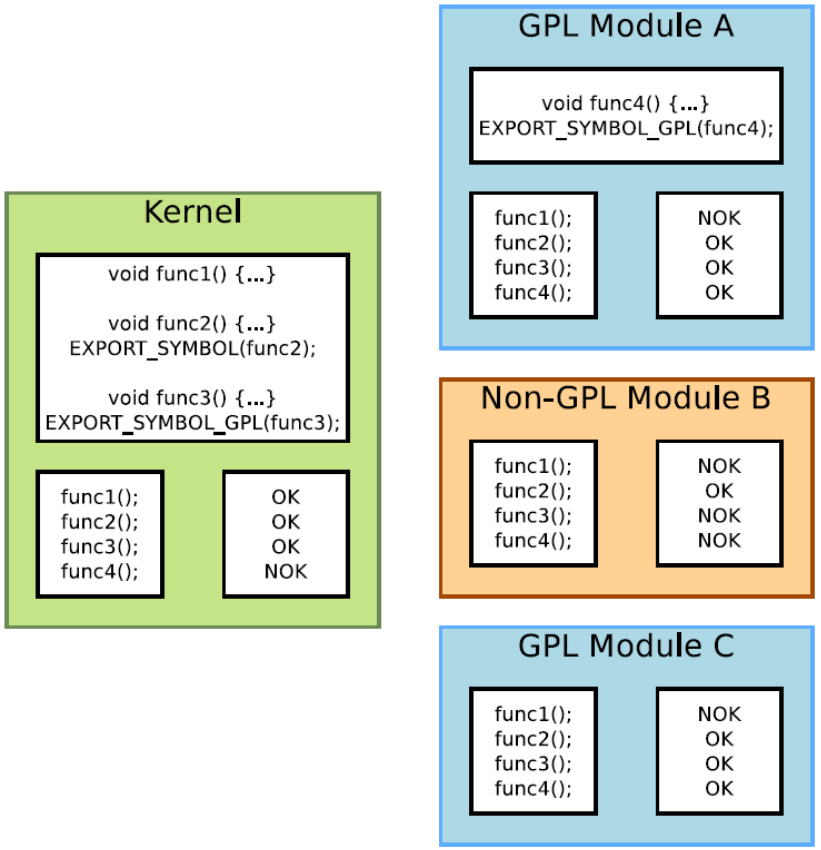
Called when the module is loaded
Removed after initialization (static kernel or module.)
Declared by the module_init() macro

Called when the module is unloaded
discarded when module compiled statically into the kernel

Metadata information declared using
MODULE_LICENSE(),
MODULE_DESCRIPTION()
MODULE_AUTHOR()

Symbols Exported to Modules

- From a kernel module, only a limited number of kernel functions can be called
- Functions and variables have to be explicitly exported by the kernel to be visible from a kernel module
- Two macros are used in the kernel 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.



Module License

- Several usages
  - Used to restrict the kernel functions that the module can use if it isn't a GPL licensed module
    - Difference between EXPORT_SYMBOL() and EXPORT_SYMBOL_GPL()
  - Used by kernel developers to identify issues coming from proprietary drivers, which they can't do anything about ("Tainted" kernel notice in kernel crashes and oopses).
  - Useful for users to check that their system is 100% free (check /proc/sys/kernel/tainted)
- Values
  - GPL compatible (see include/linux/license.h: GPL, GPL v2, GPL and additional rights, Dual MIT/GPL, Dual BSD/GPL, Dual MPL/GPL)
  - Proprietary

Compiling a Module

- Two solutions
  - Out of tree
    - When the code is outside of the kernel source tree, in a different directory
    - Advantage: Might be easier to handle than modifications to the kernel itself
    - Drawbacks: Not integrated to the kernel configuration/compilation process, needs to be built separately, the driver cannot be built statically
  - Inside the kernel tree
    - Well integrated into the kernel configuration/compilation process
    - Driver can be built statically if needed

Compiling an out-of-tree Module

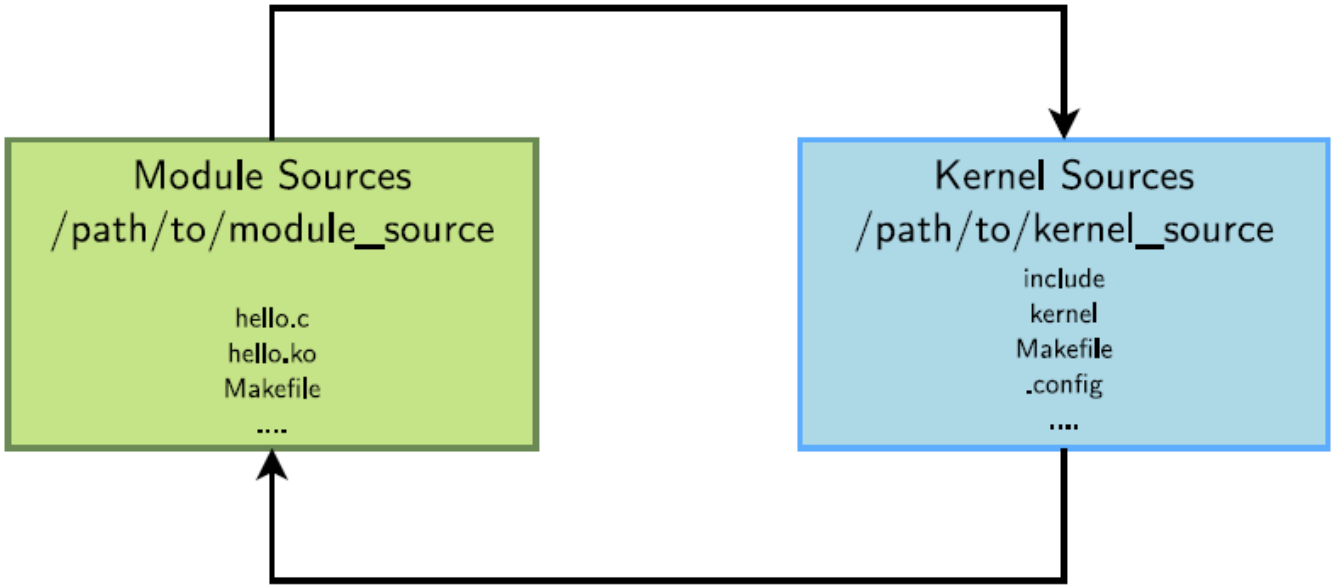
- The below Makefile should be reusable for any single-file out-of-tree Linux module
  - The source file is hello.c
  - Just run make to build the hello.ko file
  - KDIR: kernel source or headers directory

```
ifneq ($(KERNELRELEASE),)
obj-m := hello.o
else
KDIR := /path/to/kernel/sources

all:
<tab>$(MAKE) -C $(KDIR) M=$$PWD
endif
```

Compiling an out-of-tree Module

- The module Makefile is interpreted with KERNELRELEASE undefined, so it calls the kernel Makefile, passing the module directory in the M variable
- The kernel Makefile knows how to compile a module, and thanks to the M variable, knows where the Makefile for our module is.
- The module Makefile is interpreted with KERNELRELEASE defined, so the kernel sees the obj-m definition.



Modules and Kernel Version

- To be compiled, a kernel module needs access to the kernel headers, containing the definitions of functions, types and constants.
- Two solutions
  - Full kernel sources (configured + make modules_prepare)
  - Only kernel headers (linux-headers-* packages in Debian/Ubuntu distributions, or directory created by make headers_install)
- The sources or headers must be configured
  - Many macros or functions depend on the configuration
- A kernel module compiled against version X of kernel headers will not load in kernel version Y
  - modprobe / insmod will say Invalid module format

New Driver in Kernel Sources

- To add a new driver to the kernel sources:
  - Add your new source file to the appropriate source directory.
  - Example: drivers/usb/serial/navman.c
  - Single file drivers in the common case, even if the file is several thousand lines of code big.
  - Describe the configuration interface for your new driver by adding the following lines to the Kconfig file in this directory:

```
config USB_SERIAL_NAVMAN
        tristate "USB Navman GPS device"
        depends on USB_SERIAL
        help
          To compile this driver as a module, choose M
          here: the module will be called navman.
```

New Driver in Kernel Sources

- Add a line in the Makefile file based on the Kconfig setting:
  - obj-$(CONFIG_USB_SERIAL_NAVMAN) += navman.o
    - tells the kernel build system to build navman.c when the USB_SERIAL_NAVMAN option is enabled.
  - Works both if compiled statically or as a module.
  - Run make xconfig and see your new options!
  - Run make and your new files are compiled!
  - See Documentation/kbuild/ for details and more elaborate examples like drivers with several source files, or drivers in their own subdirectory, etc.

Hello Module with Parameters

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/moduleparam.h>
MODULE_LICENSE("GPL");

/* A couple of parameters that can be passed in: how
many times we say hello, and to whom */
static char *whom = "world";
module_param(whom, charp, 0);

static int howmany = 1;
module_param(howmany, int, 0);

static int __init hello_init(void)
{
  int i;
  for (i = 0; i < howmany; i++)
  pr_alert("(%d) Hello, %s\n", i, whom);
  return 0;
}

static void __exit hello_exit(void)
{
  pr_alert("Goodbye, cruel %s\n", whom);
}

module_init(hello_init);
module_exit(hello_exit);
```

```
#include <linux/moduleparam.h>
module_param(
      name, /* name of an already defined variable */
      type, /* either byte, short, ushort, int, uint, long, ulong,
               charp, or bool. (checked at run time!) */
      perm /* for /sys/module/<module_name>/parameters/<param>,
               0: no such module parameter value file */
);
```

```
% insmod hello howmany=10 whom="Mom"
```

Modules parameter arrays are also possible

```
int myintarray[2];
module_param_array(myintarray,int,NULL,0)
int myshortarray[4];
int count;
module_param_array(myshortarray, int, &count, 0);
```

Useful general-purpose kernel APIs

- **Memory/string utilities**
- In include/linux/string.h
  - Memory-related: memset(), memcpy(), memmove(), memscan(), memcmp(), memchr()
  - String-related: strcpy(), strcat(), strcmp(), strchr(), strrchr(), strlen() and variants
  - Allocate and copy a string: kstrdup(), kstrndup()
  - Allocate and copy a memory area: kmemdup()
- In include/linux/kernel.h
  - String to int conversion: simple_strtoul(), simple_strtol(), simple_strtoull(), simple_strtoll()
  - Other string functions: sprintf(), sscanf()

Useful general-purpose kernel APIs

- **Linked lists**
- Convenient linked-list facility in include/linux/list.h
  - Used in thousands of places in the kernel
- Add a struct list_head member to the structure whose instances will be part of the linked list. It is usually named node when each instance needs to only be part of a single list.

From include/linux/atmel_tc.h

```
/*
 * Definition of a list element, with a
 * struct list_head member
 */
struct atmel_tc
{
    /* some members */
    struct list_head node;
};
```

Useful general-purpose kernel APIs

- Define the list with the LIST_HEAD() macro for a global list, or define a struct list_head element and initialize it with INIT_LIST_HEAD() for lists embedded in a structure.

From drivers/misc/atmel_tclib.c

```
/* Define the global list */
static LIST_HEAD(tc_list);
```

Useful general-purpose kernel APIs

- Then use the list_*() API to manipulate the list
  - Add elements: list_add(), list_add_tail()
  - Remove, move or replace elements: list_del(), list_move(), list_move_tail(), list_replace()
  - Test the list: list_empty()
  - Iterate over the list: list_for_each_*() family of macros

```
static int __init tc_probe(struct platform_device *pdev) {
    struct atmel_tc *tc;
    tc = kzalloc(sizeof(struct atmel_tc), GFP_KERNEL);
    /* Add an element to the list */
    list_add_tail(&tc->node, &tc_list);
}

struct atmel_tc *atmel_tc_alloc(unsigned block, const char *name)
{
    struct atmel_tc *tc;
    /* Iterate over the list elements */
    list_for_each_entry(tc, &tc_list, node) {
        /* Do something with tc */
    }
    [...]
}
```