


Advanced Operating Systems
MS degree in Computer Engineering
University of Rome Tor Vergata 
Lecturer: Francesco Quaglia

Linux modules

1. Support system calls and services
2. Programming facilities
3. Kernel probing

Modules basics

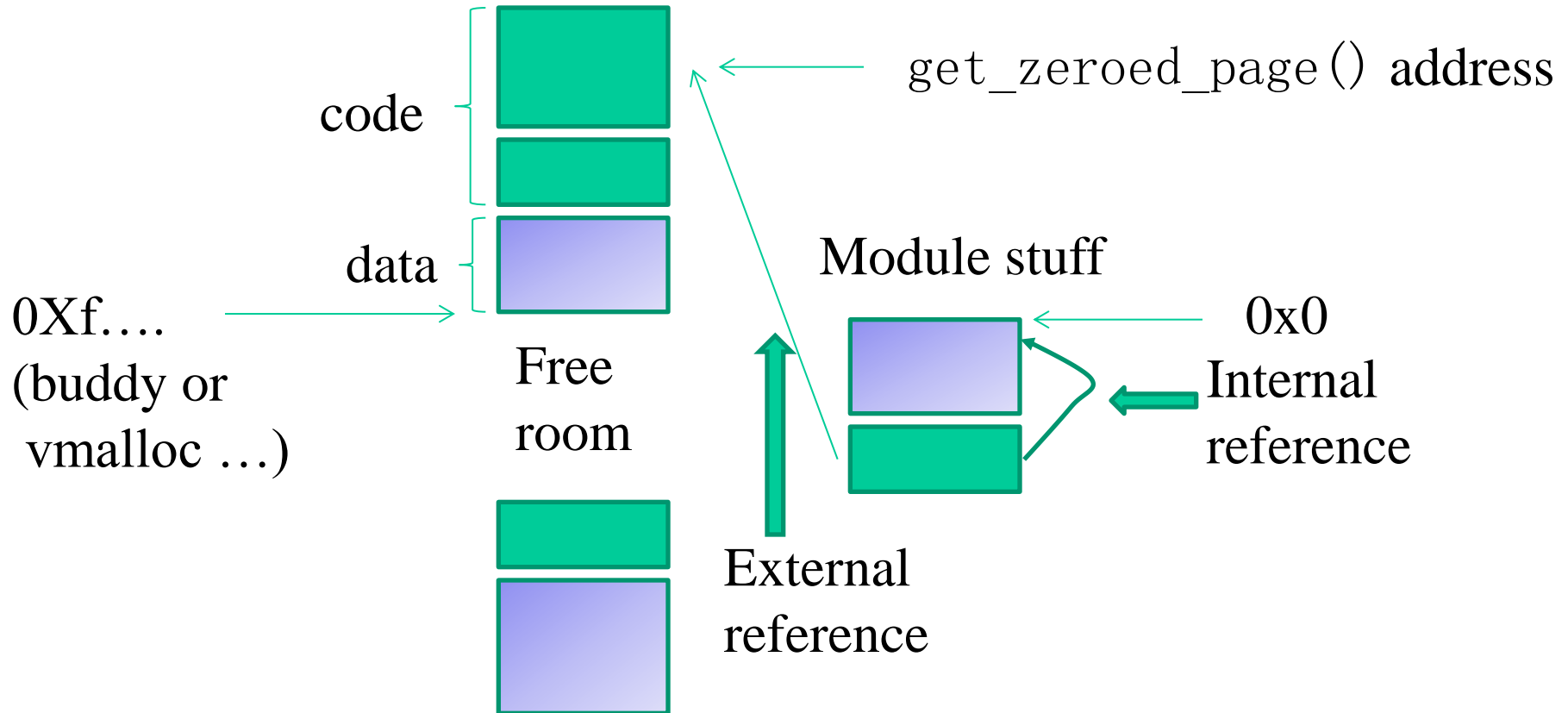
- A LINUX module is a software component which can be added as part of the kernel (hence being included into the kernel memory image) when the latter is already running
- One advantage of using modules is that the kernel does not need to be recompiled in order to add the corresponding software facility
- Modules are also used as a base technology for developing new parts of the kernel that are then integrated (once stable) in the original compiled image
- They are also used to tailor the start-up of a kernel configuration, depending on specific needs

Steps for module insertion

- We need memory for loading in RAM both code blocks and data structures included in the module
- We need to know where the corresponding logical buffer is located in order to resolve internal references by the module (to either data or code)
- We need to know where in logical memory are located the kernel facilities the module relies on
- While loading the module, actual manipulation for symbols resolution (to addresses) needs to be carried out

A scheme

Kernel image



Who does the job??

- It depends on the kernel release
- Up to kernel 2.4 most of the job (but not all) is done at application level
 - ✓ A module is a **.o ELF**
 - ✓ Shell commands are used to reserve memory, resolve the symbols' addresses and load the module in RAM
- From kernel 2.6 most of the job is kernel-internal
 - ✓ A module is a **.ko ELF**
 - ✓ Shell commands are used to trigger the kernel actions for memory allocation, address resolving and module loading

System call suite up to kernel 2.4

create_module

- ✓ reserves the logical kernel buffer
- ✓ associates a name to the buffer

init_module

- ✓ loads the finalized module image into the kernel buffer
- ✓ calls the module setup function

delete_module

- ✓ calls the module shutdown function
- ✓ releases the logical memory buffer

System call suite from kernel 2.6

create_module

- ✓ no more supported

init_module

- ✓ reserves the logical kernel buffer
- ✓ associates a name to the buffer
- ✓ loads the non-finalized module image into the kernel buffer
- ✓ calls the module setup function

delete_module

- ✓ calls the module shutdown function
- ✓ releases the logical memory buffer

Common parts (i)

- A module is featured by two main functions which indicate the actions to be executed upon **loading or unloading** the module
- These two functions have the following prototypes

```
int init_module(void) /*used for all  
                        initialization stuff*/  
                { ... }  
  
void cleanup_module(void) /*used for a  
                            clean shutdown*/  
                  { ... }
```


Common parts (ii)

- Within the metadata that are used to handle a module we have a so called usage-count (or reference-count)
- If the usage-count is not set to zero, then the module is so called “locked”
- This means that we can expect that some thread will eventually need to use the module stuff (either in process context or in interrupt context), e.g. for task finalization purposes
- Unload in this case fails, except if explicitly forced
- If the usage-count is set to zero, the module is unlocked, and can be unloaded with no particular care (or force command)

Common parts (iii)

- We can pass parameters two modules in both technologies
- These are not passed as actual function parameters
- Rather, they are passed as initial values of global variables appearing in the module source code
- These variables, after being declared, need to be marked as “module parameters” explicitly

Declaration of module parameters

- For any parameter to be provided in input we need to rely on the below macros defined in `include/linux/module.h` or `include/linux/moduleparam.h`
 - ✓ `MODULE_PARM(variable, type)` (**old style**)
 - ✓ `module_param(variable, type, perm)`
- These macros specify the name of the global variable to be treated as input parameter and the corresponding data type (“i” int – “l” long – “s” string – “b” byte etc.)
- The three-parameter version is used in order to expose the variable value as a pseudo-file content (hence we need to specify permissions)

Module parameters dynamic audit

- It can be done via the `/sys` pseudo-file system
- It is an aside one with respect to `/proc`
- In `/sys` for each module we find pseudo-files for inspecting the state of the module
- These include files for all the module parameters that are declared as accessible (on the basis of the permission mask) in the pseudo file system
- We can even modify the parameters at run-time, if permissions allow it

A variant for array arguments

- `module_param_array()` can be used to declare the presence of parameters that are array of values
- this macro takes in input 4 parameters
 - ✓ The array-variable name
 - ✓ The base type of an array element
 - ✓ The address of a variable that will specify the array size
 - ✓ The permission for the access to the module parameter on the pseudo file system
- An example

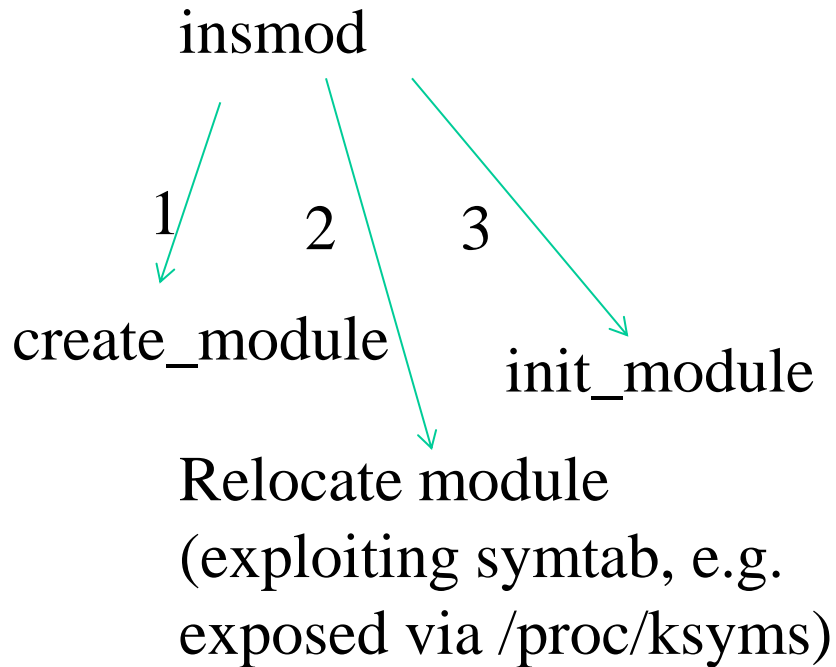
```
module_param_array(myintarray,int,&size,0)
```

Loading/unloading a module

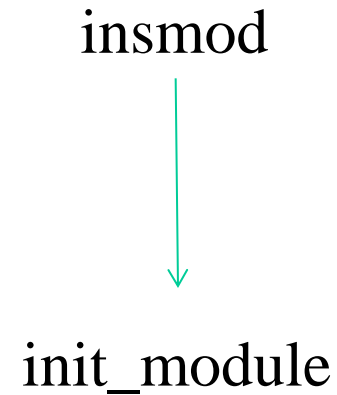
- A module can be loaded by the administrator via the shell command `insmod`
- You can use this also for passing parameters (in the form `variable=value`)
- This command takes the name of the object file generated by compiling the module source code as the parameter
- The unloading of a module can be executed via the shell command `rmmod`
- We can also use `modprobe`, which by default looks for the actual module in the directory `/lib/modules/'uname -r'`

Actual execution path of insmod

Up to kernel 2.4



since kernel 2.6



Module suited system calls – up to 2.4

```
#include <linux/module.h>
caddr_t create_module(const char *name,
size_t size);
```

DESCRIPTION

`create_module` attempts to create a loadable module entry and reserve the kernel memory that will be needed to hold the module. This system call is only open to the superuser.

RETURN VALUE

On success, returns the kernel address at which the module will reside. On error -1 is returned and `errno` is set appropriately.


```
#include <linux/module.h>
int init_module(const char *name, struct module
*image);
```

DESCRIPTION

`init_module` **loads the relocated module** image into kernel space and runs the module's init function. The module image begins with a module structure and is followed by code and data as appropriate. The module structure is defined as follows:

```
struct module {
    unsigned long size_of_struct;
    struct module *next;    const char *name;
    unsigned long size;      long usecount;
    unsigned long flags;     unsigned int nsyms;
    unsigned int ndeps;      struct module_symbol *syms;
    struct module_ref *deps; struct module_ref *refs;
    int (*init)(void); void (*cleanup)(void);
    const struct exception_table_entry *ex_table_start;
    const struct exception_table_entry *ex_table_end;
#ifdef __alpha__
    unsigned long gp;
#endif
};
```

In the 2.4 tool chain parameters are setup by the insmod user program

In fact their existence is not reflected into any module-suited system call signature

They cannot be changed at run-time from external module stuff (except if we hack their memory locations)

```
#include <linux/module.h>
int delete_module(const char *name);
```

DESCRIPTION

`delete_module` attempts to remove an unused loadable module entry. If `name` is `NULL`, all unused modules marked auto-clean will be removed. This system call is only open to the superuser.

RETURN VALUE

On success, zero is returned. On error, -1 is returned and `errno` is set appropriately.

Module suited system calls – since 2.6

SYNOPSIS

```
int init_module(void *module_image, unsigned long len,  
               const char *param_values);  
  
int finit_module(int fd, const char *param_values,  
                int flags);
```

Note: glibc provides no header file declaration of `init_module()` and no wrapper function for `finit_module()`; see NOTES.

DESCRIPTION

`init_module()` loads an ELF image into kernel space, performs any necessary symbol relocations, initializes module parameters to values provided by the caller, and then runs the module's init function. This system call requires privilege.

The `module_image` argument points to a buffer containing the binary image to be loaded; `len` specifies the size of that buffer. The module image should be a valid ELF image, built for the running kernel.

What about the missing address resolution job by insmod in the 2.6 tool-chain?

- To make a .ko file, we start with a regular .o file.
- The **modpost** program creates (from the .o file) a C source file that describes the additional sections that are required for the .ko file
- The C file is called .mod file
- The .mod file is compiled and linked with the original .o file to make a .ko file

Module headings

For inclusion of header files with pre-processor directive `ifdef __KERNEL__`

`#define __KERNEL__`

For inclusion of header files with Pre-processor directive `ifdef MODULE`

`#define MODULE`

`#include <linux/module.h>`

`#include <linux/kernel.h>`

`#define __SMP__ /* if compiled for SMP */`

Module in-use indications (classical style)

- The LINUX kernel associated with any loaded module a counter
- Typically, this counter is used to indicate how many **processes/threads/top-bottom-halves** still need to rely on the module software for finalizing some job
- In case the counter is currently greater than zero, the unload of the module will fail
- There are macros defined in `include/linux/module.h`, which are suited for accessing/manipulating the counter
 - `MOD_INC_USE_COUNT`
 - `MOD_DEC_USE_COUNT`
 - `MOD_IN_USE`

- **NOTE:**

- While debugging the module it would be convenient to redefine the macros `MOD_INC_USE_COUNT` and `MOD_DEC_USE_COUNT` as **no-ops**, so to avoid blocking scenarios when attempting to unload the module

- **NOTE:**

- the `proc` file system exposes a proper file `/proc/modules` which provides information on any loaded module, including the usage counter and the amount of memory reserved for the module

Reference counter interface in 2.6

We have the following functions:

- ✓ try_module_get(struct module *module) for incrementing the reference counter
- ✓ module_put(struct module *module) for decrementing the reference counter
- ✓ CONFIG_MODULE_UNLOAD can be used to check unloadability

Finding a module to lock/unlock

```
struct module *find_module(const char *name)
```



This provides us with capabilities of targeting an “external” module

The macro `THIS_MODULE` passed in input can be used to identify the module that is calling the API, it clearly works also with `try_module_lock/module_put`

Kernel exported symbols

- Either the LINUX kernel or its modules can **export symbols**
- An exported symbol (e.g., the name of a variable or the name of a function) is made available and can be referenced by any module to be loaded
- If a module references a symbol which is not exported, then the loading of the module will fail
- The Kernel (including the modules) can export symbols by relying on the macro `EXPORT_SYMBOL (symbol)` which is defined in `include/linux/module.h`
- **NOTE** (for old style management)
 - If a module exports symbols it must rely on the pre-processor directive specified by the macro `EXPORT_SYMTAB,` to be inserted right before including `include/linux/module.h`
 - In case no symbol is exported by the module, then it is common practice to indicate this to the pre-processor by exploiting the macro `EXPORT_NO_SYMBOLS`

Exported symbols table

- There exist a table including all the symbols that are exported by the compiled kernel
- Further, each module is associated with a per module table of exported symbols (if any)
- All the symbols that are currently exported by the kernel (and by its modules) are accessible via the **proc file system** through the file `/proc/ksyms`
- This file keeps a line for each exported symbol, which has the following format

`Kernel-memory-address symbol-name`

- More recently the `/proc/kallsyms` file has been used, also exposing the symbol type

A note on exporting symbols

- kernel can be parameterized (compiled) to export differentiated types of symbols via standard facilities (e.g. `/proc/kallsyms`)

- A few examples

`CONFIG_KALLSYMS = y`

`CONFIG_KALLSYMS_ALL = y` symbol table includes all the variables (including `EXPORT_SYMBOL` derived variables)

All the previous are required for exporting variables (not located in the stack)

Dynamic symbols querying and kernel patching

```
int __kprobes register_kprobe(struct kprobe *p)
```

```
static int __kprobes __unregister_kprobe_top(struct kprobe *p)
```

```
int __kprobes register_kretprobe(struct kprobe *p)
```

To enable kprobes: CONFIG_KPROBES=y and
CONFIG_KALLSYMS=y or CONFIG_KALLSYMS_ALL=y

Example usage

```
// Get a kernel probe to access flush_tlb_all
```

```
    memset(&kp, 0, sizeof(kp));
```

```
    kp.symbol_name = "flush_tlb_all";
```

```
    if (!register_kprobe(&kp)) {
```

```
        flush_tlb_all_lookup = (void *) kp.addr;
```

```
        unregister_kprobe(&kp);
```

```
    }
```

struct kprobe

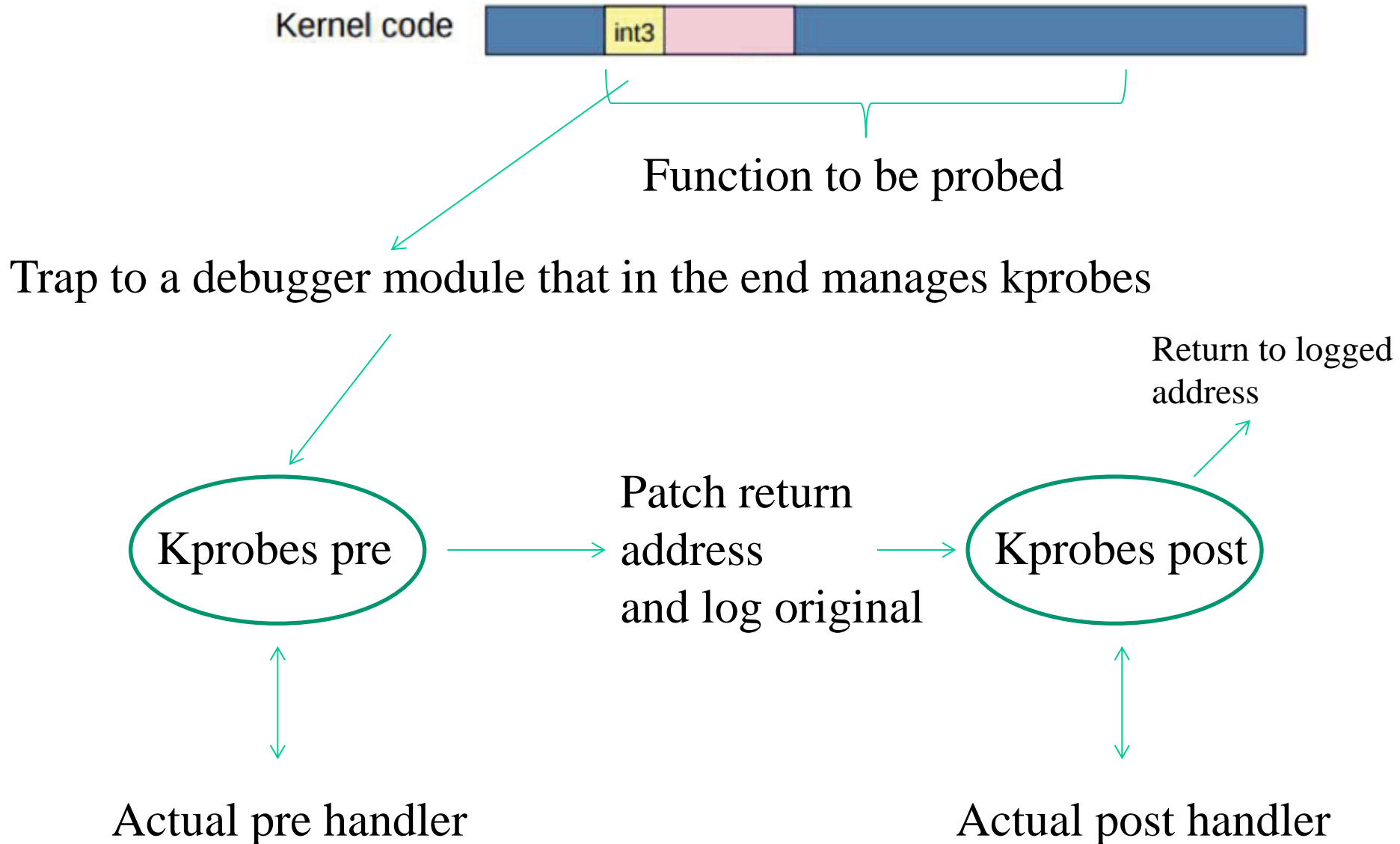
<linux/kprobes.h>

```
struct kprobe {
    struct hlist_node hlist; /* Internal */
    kprobe_opcode_t addr; /* Address of probe */
    kprobe_pre_handler_t pre_handler;
                        /* Address of pre-handler */
    kprobe_post_handler_t post_handler;
                        /* Address of post-handler */
    kprobe_fault_handler_t fault_handler;
                        /* Address of fault handler */
    kprobe_break_handler_t break_handler;
                        /* Internal */
    kprobe_opcode_t opcode; /* Internal */
    kprobe_opcode_t insn[MAX_INSN_SIZE]; /* Internal */
};
```

struct kprobe (kernel 3 or 4)

```
struct kprobe {  
    struct hlist node hlist;  
  
    /* list of kprobes for multi-handler support */  
    struct list head list;  
  
    /*count the number of times this probe  
       was temporarily disarmed */  
    unsigned long nmisses;  
    .....  
    .....  
}
```


Kprobe mechanism



Kprobe handlers

```
typedef int (*kprobe_pre_handler_t)
            (struct kprobe*, struct pt_regs*);

typedef void (*kprobe_post_handler_t)
            (struct kprobe*, struct pt_regs*,
             unsigned long flags);

typedef int (*kprobe_fault_handler_t)
            (struct kprobe*, struct pt_regs*, int trapnr);
```

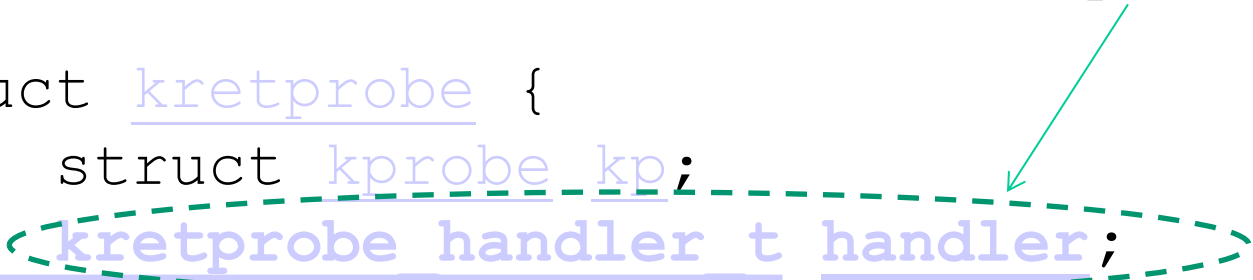
Modifiable registers status



kretprobe

Same interface as other
probe handlers

```
struct kretprobe {  
    struct kprobe kp;  
    kretprobe handler t handler;  
    kretprobe handler t entry handler;  
    int maxactive;  
    int nmissed;  
    size t data size;  
    struct hlist head free_instances;  
    raw spinlock t lock;  
};
```



Probing deny

- Not all kernel functions can be probed
- A few of them are blacklisted (depending on compilation choices)
- Those that are blacklisted can be found in the pseudofile

`/sys/kernel/debug/kprobes/blacklist`

- Motivations can be compiler optimizations (such as in-lining) or the fact that these functions can be triggered by probe executions

Symbol versioning

- If the kernel is compiled with the macro `CONFIG_MODVERSIONS` activated (as usually happens), then each symbol is associated with a specific code representing the symbol version
- The code is computed by an algorithm relying on the Cyclic Redundancy Code (CRC) applied to the symbol prototype
- In such a case the `/proc/ksyms` file will contain symbol names expressed as

`symbol-name1_Rversion`

- **NOTE:**
 - If `CONFIG_MODVERSIONS` is active a check on the symbols used within the module is performed in order to verify their version compliance with the ones exported by the kernel (except for cases where the `-f` option is passed to `insmod`)
 - Hence we can recognize whether a module is adequate for loading onto a specific kernel version

Still on versioning

- the macro expanding the symbol name into one having the versioning suffix is defined within the file `include/linux/modversions.h`
- `EXPORT_SYMBOL()` will rely onto the above macro
- For enabling versioning within a module we need `CONFIG_MODVERSIONS==1`, `MODVERSIONS` needs to be defined and `include/linux/modversions.h` must be included
- **NOTE:**
 - Symbols can anyhow be exported without versioning
 - This can be done via the macro `EXPORT_SYMBOL_NOVERS (symbol)` which is defined in `/include/linux/module.h`

LINUX kernel versioning

- The `include/linux/version.h` file is automatically included via the inclusion of `include/linux/module.h` (except for cases where the `__NO_VERSION__` macro is used)
- The `include/linux/version.h` file entails macros that can be used for catching information related to the actual kernel version such as:
 - `UTS_RELEASE`, which is expanded as a string defining the version of the kernel which is the target for the compilation of the module (e.g. “2.3.48”)
 - `LINUX_VERSION_CODE` which is expanded to the binary representation of the kernel version (with one byte for each number specifying the version)
 - `KERNEL_VERSION(major, minor, release)` which is expanded to the binary value representing the version number as defined via `major`, `minor` and `release`

Renaming of module startup/shutdown functions

- Starting from version 2.3.13 we have facilities for renaming the startup and shutdown functions of a module
- These are defined in the file `include/linux/init.h` as:
 - `module_init(my_init)` which generates a startup routine associated with the symbol `my_init`
 - `module_exit(my_exit)` which generates a shutdown routine associated with the symbol `my_exit`
- These should be used at the bottom of the main source file for the module
- They can help on the side of debugging since we can avoid using functions with the same name for the modules
- Further, we can develop code that can natively be integrated within the initial kernel image or can still represent some module for specific compilation targets