Loadable Kernel Modules

Advanced Operating Systems and Virtualization

Alessandro Pellegrini

A.Y. 2019/2020

Basics

- A Loadable Kernel Module (LKM) is a software component which can be added to the memory image of the Kernel while it is already running
- The kernel does not need to be recompiled to add new software facilities
- They are also used to develop new parts of the Kernel that can be then integrated in the final image once stable
- They are also used to tailor the start up of a kernel configuration, depending on specific needs

Module Initialization and Description

```
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Author <me@example.com>");
MODULE_DESCRIPTION("My Fancy Module");
module_init(my_module_init);
module_exit(my_module_cleanup);
```

Reference Counters

- The Kernel keeps a reference counter for each loaded LKM
- If the reference counter is greater than zero, then the module is locked
- This means that there are other services in the system which rely on facilities exposed by the module
- If not forced, unloading of the module fails
- `lsmod` gives also information on who is using the module

Reference Counters

- `try_module_get()`: try to increment the reference counter
- `module_put()`: decrement the reference counter
- `CONFIG_MODULE_UNLOAD` is a global macro which allows the kernel to unload modules
  - it can be used to check unloadability

Module Parameters

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

Module Parameters

- Defined in `include/linux/module.h` or `include/linux/moduleparm.h`
  - `MODULE_PARM(variable, type)` (deprecated)
  - `module_param(variable, type, perm)`
  - `module_param_array(name, type, num, perm);`
- These macros specify the name of the global variable which "receives" the input, its type, and its permission (when mapped to a pseudofile)
- Based on pseudo-files in /sys
- Initialization is done upon module load

Module Parameters

```
module_param(myshort, short, S_IRUSR | S_IWUSR |
                        S_IRGRP | S_IWGRP);
MODULE_PARM_DESC(myshort, "A short integer");

module_param(myint, int, S_IRUSR | S_IWUSR | S_IRGRP
                | S_IROTH);
MODULE_PARM_DESC(myint, "An integer");

module_param(mylong, long, S_IRUSR);
MODULE_PARM_DESC(mylong, "A long integer");

module_param(mystring, charp, 0000);
MODULE_PARM_DESC(mystring, "A character string");

module_param_array(myintArray, int, &arr_argc, 0000);
MODULE_PARM_DESC(myintArray, "An array of integers");
```

Module Parameters

```
/* You can use " around spaces, but can't escape ". */
/* Hyphens and underscores equivalent in parameter names. */

static char *next_arg(char *args, char **param, char **val)
{
    ...
    for (i = 0; args[i]; i++) {
        if (isspace(args[i]) && !in_quote)
            break;
    ...
}
```

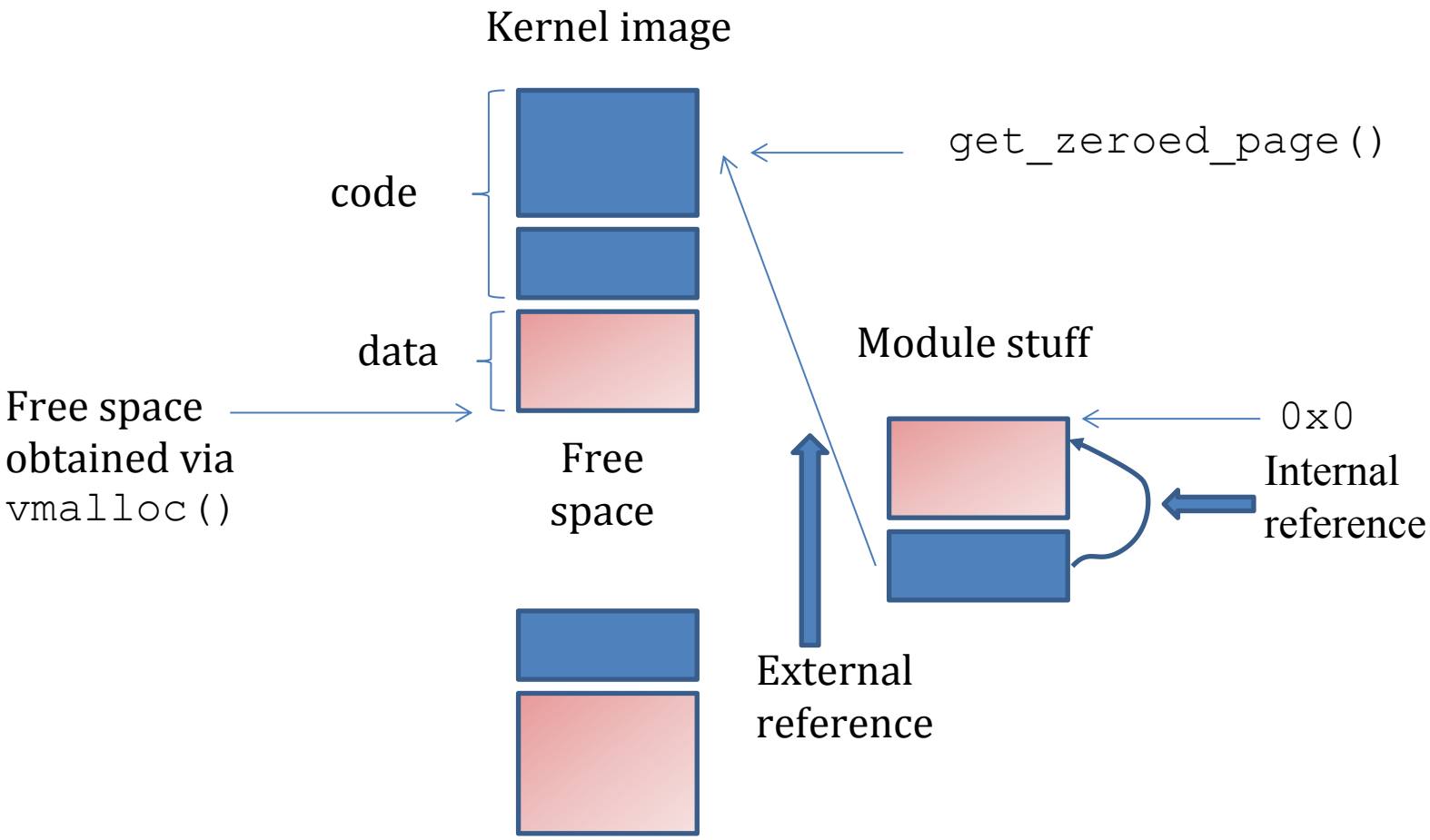
Loading/Unloading a Module

- A module is loaded by the administrator via the shell command `insmod`
- It can also be used to pass parameters (`variable=value`)
- It takes as a parameter the path to the object file generated when compiling the module
- A module is unloaded 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)`

Steps to Load a Module

- We need memory to load in RAM both code and data structures
- We need to know several memory locations to perform a dynamic resolution:
  - Base address of the module, for internal references
  - Locations in memory of static Kernel facilities (functions and data)

Loading Scheme



Who does the job?

- Up to kernel 2.4 most of the job is done is userspace
  - A module is a **.o ELF**
  - Applications 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, module loading, and address resolution

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.

System Calls up to 2.4

```
#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;
  };
```

System Calls up to 2.4

```
#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 autoclean 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.

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);
int delete_module(const char *name, 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.

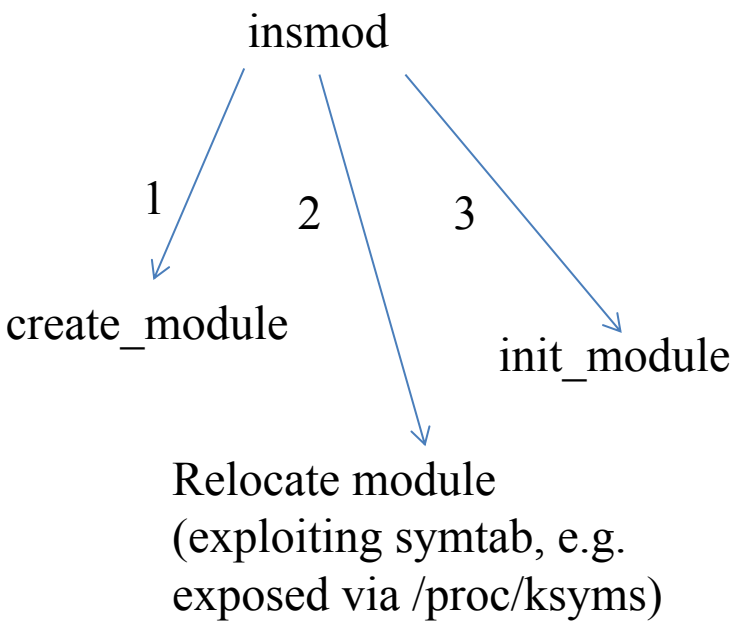
The module_image argument points to a buffer containing the binary image to be loaded; len specifies the size of that buffer. The module image should be a valid ELF image, built for the running kernel.

Dynamic Resolution on 2.6

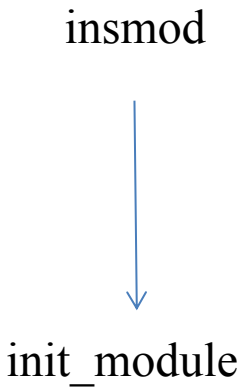
- To make a .ko file, we start with a regular .o file.
- The **modpost** program creates (from the .o file) a C source file that describes the additional sections that are required for the .ko file
- The C file is called .mod file
- The .mod file is compiled and linked with the original .o file to make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

`insmod` Operations

Up to Kernel 2.4



Since Kernel 2.6



Kernel Exported Symbols

- Symbols from the Kernel or from modules can be *exported*
- An exported symbol can be referenced by other modules
- If a module references a symbol which is not exported, then loading the module will fail
- `EXPORT_SYMBOL(symbol)` defined in `include/linux/module.h`
- This must be configured:

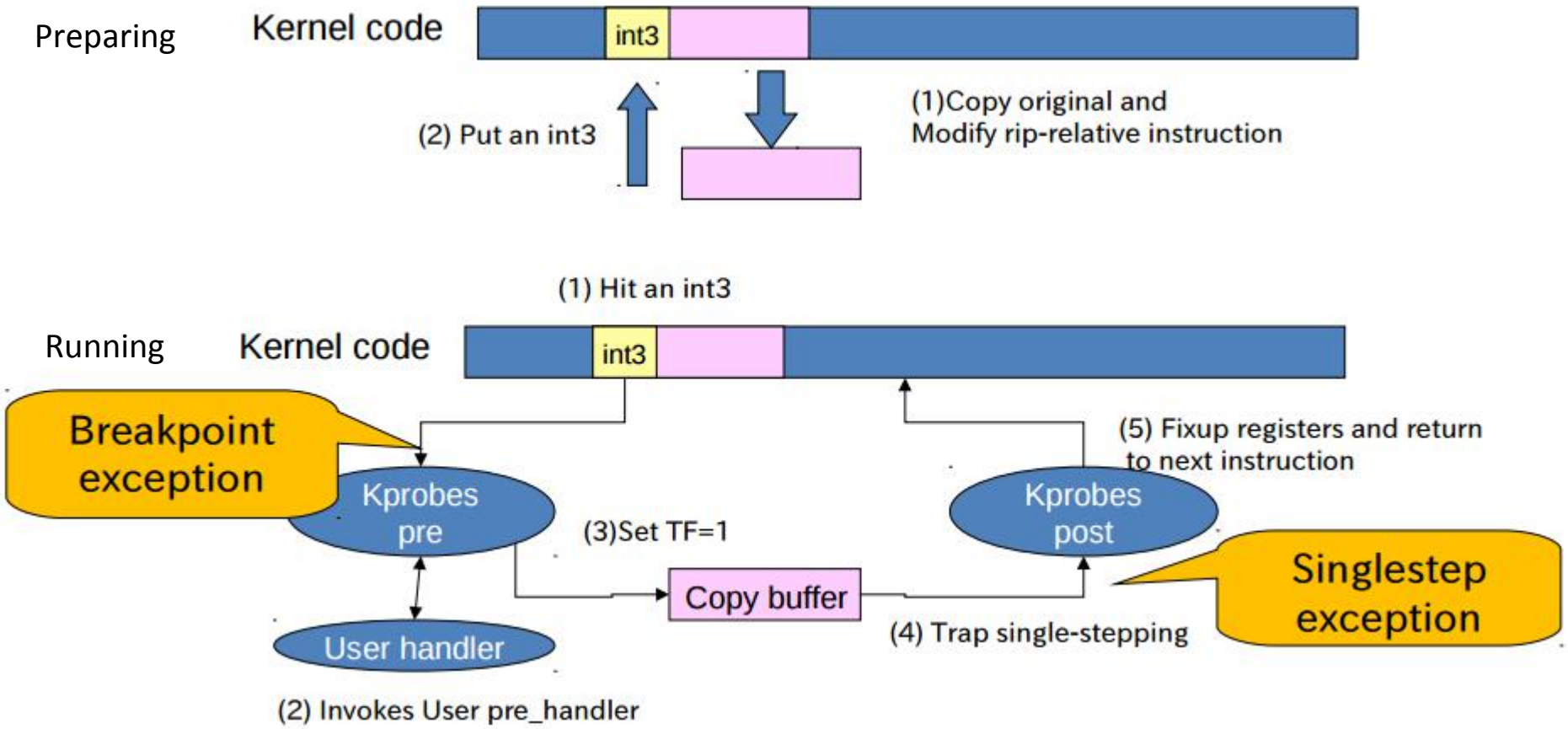
  `CONFIG_KALLSYMS = y`
  `CONFIG_KALLSYMS_ALL = y` (include all symbols)
- Exported symbols are placed in the `__ksymtab` section

Kprobes

- Kprobes are meant as a support for dynamic tracing in the Kernel
- `int register_kprobe(struct kprobe *p)` in `include/linux/kprobes.h` specifies where the probe is to be inserted and what `pre_` and `post_` handlers are to be called when the probe is hit.
- `unregister_kprobe(struct kprobe *p)`
- `typedef int (*kprobe_pre_handler_t) (struct kprobe *, struct pt_regs *)`
- To enable kprobes:
  - `CONFIG_KPROBES=y` and `CONFIG_KALLSYMS=y` or `CONFIG_KALLSYMS_ALL=y`

Kprobes

- Use breakpoints and single-step on copied code



Kprobes

- Kprobes can be installed anywhere in the kernel
  - Multiple probes at the same address
  - Multiple handlers (or multiple instances of the same handler) may run concurrently on different CPUs.
  - Registered kprobes are visible under the `/sys/kernel/debug/kprobes/` directory
  - when registered, probes are saved in a hash table hashed by the address of the probe
  - Hash table is protected by `kprobe_lock` (a spinlock)
- Kprobes cannot probe itself
  - use a blacklist to prevent recursive traps
- Probe handlers are run with preemption disabled.
  - Depending on the architecture and optimization state, handlers may also run with interrupts disabled (not on x86/x86-64).
  - In any case, should not yield the CPU (e.g., by attempting to acquire a semaphore).

Kprobes and Non-Exported Symbols

```
// 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);
}
```

Linux Kernel Versioning

- `include/linux/version.h` is automatically included via the inclusion of `include/linux/module.h` (except for cases where the `__NO_VERSION__` macro is used)
- `include/linux/version.h` provides macros that can be used to get information related to the "current" kernel version such as:
  - `UTS_RELEASE`: expanded to a string defining the kernel version (e.g. "4.1.7")
  - `LINUX_VERSION_CODE`: expanded to the binary representation of the kernel version (with one byte for each number specifying the version)
  - `KERNEL_VERSION(major,minor,release)`: expanded to the binary value representing the version number as defined by `major`, `minor` and `release`

/sys/module

- /sys/module/MODULENAME: The name of the module that is in the kernel. This module name will always show up if the module is loaded as a dynamic module.
- /sys/module/MODULENAME/parameters: This directory contains individual files that are each individual parameters of the module that are able to be changed at runtime.
- /sys/module/MODULENAME/refcnt: If the module is able to be unloaded from the kernel, this file will contain the current reference count of the module.
  - Note: If CONFIG_MODULE_UNLOAD kernel configuration value is not enabled, this file will not be present.