# Advanced Operating Systems (and System Security)
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
4. Kernel audit

# 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 baseline 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

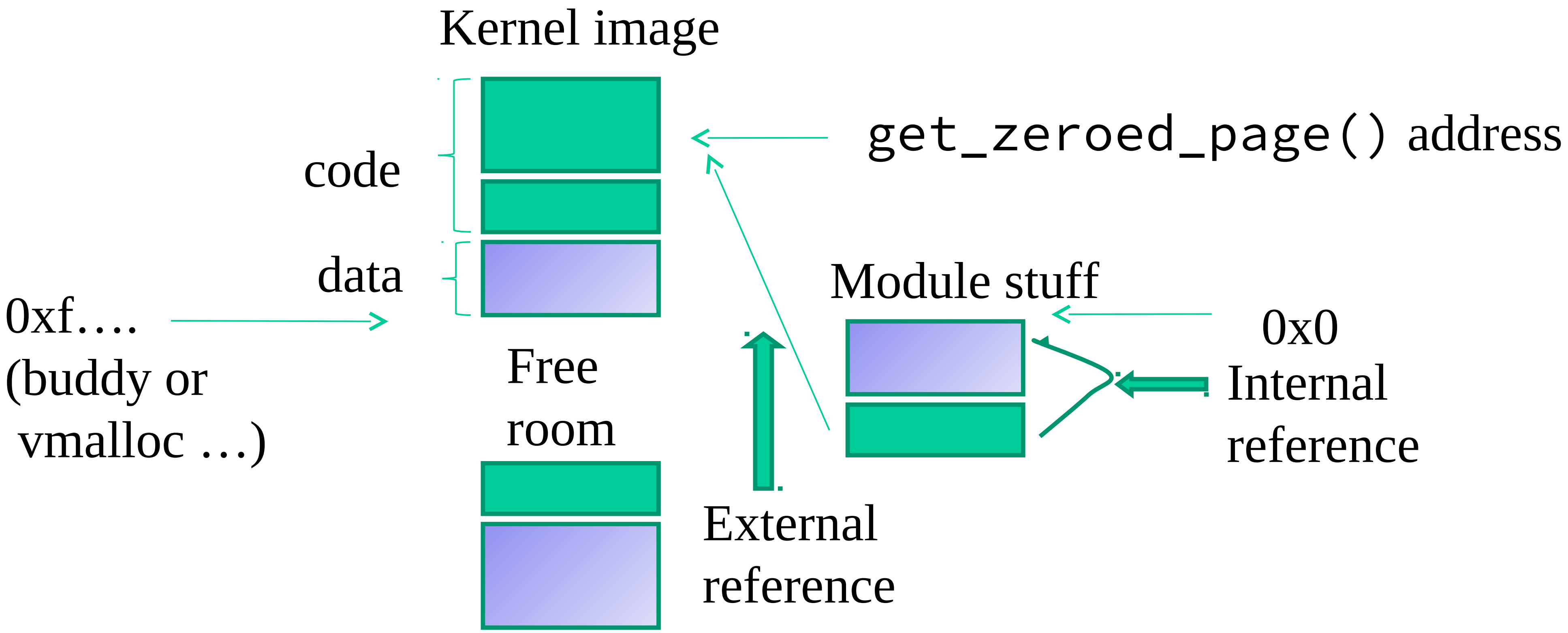
- We need memory for loading in RAM both code blocks and data structures included in the module

- 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

code

data

`get_zeroed_page()` address

0xf….

(buddy or

vmalloc …)

Free

room

Module stuff

0x0

Internal

reference

External

reference

# 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, <u>resolve the symbols' addresses</u> 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, <u>address resolving</u> 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 kernel buffer

# System call suite from kernel 2.6

**create_module**

✓ no longer 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 kernel 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
 initialition 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 used 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 <u>**pass parameters to modules**</u> 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/moduleparm.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

- 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

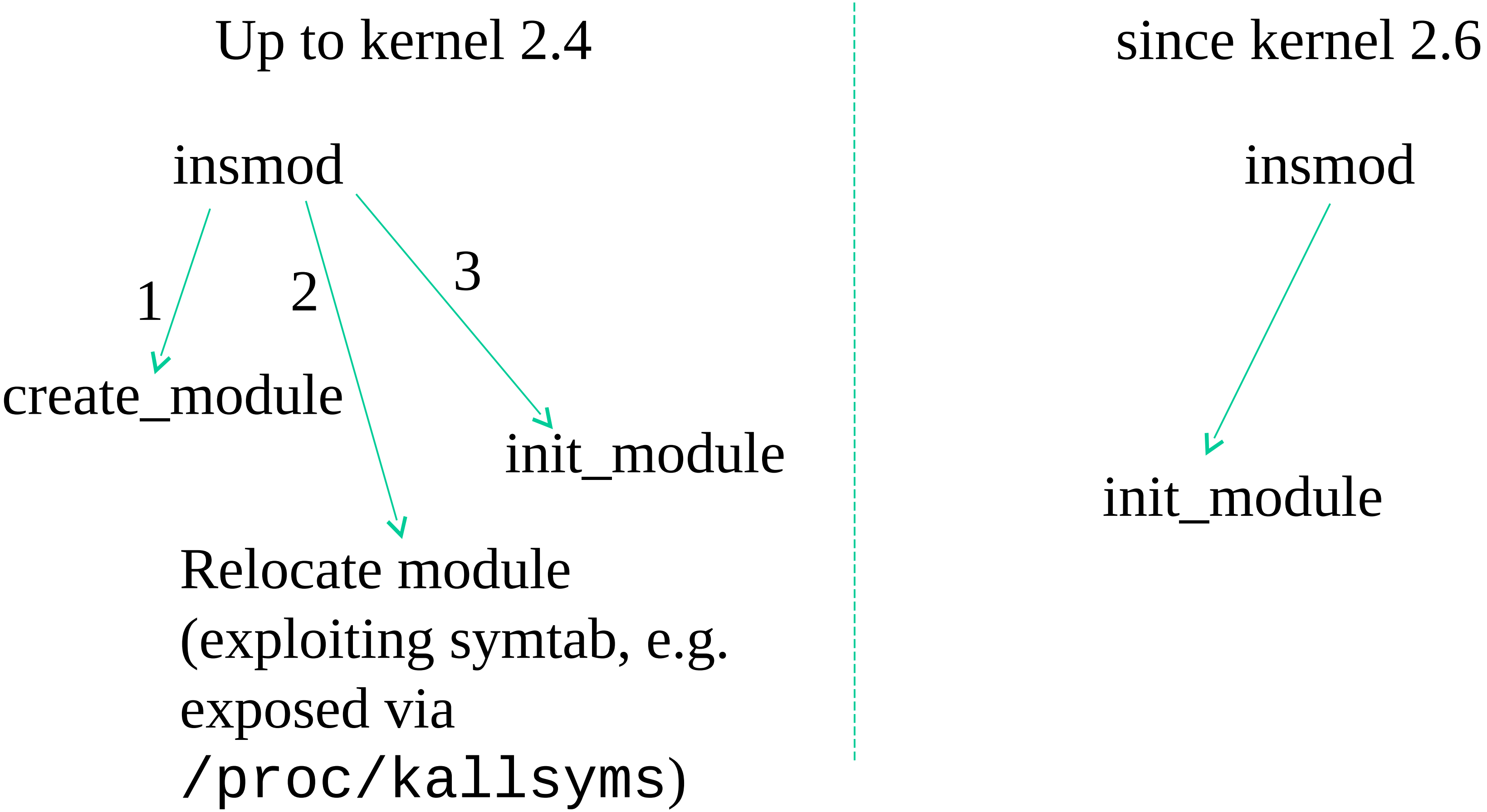
- `module_param_array()` can be used to declare the presence of parameters that are array of values
- this macro takes in input 4 parameters
  - ✓ The array-variable name
  - ✓ The base type of an array element
  - ✓ The address of a variable that will specify the array size
  - ✓ The permission for 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 it 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

insmod

1 create_module

2 Relocate module (exploiting symtab, e.g. exposed via `/proc/kallsyms`)

3 init_module

since kernel 2.6

insmod

init_module

# 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
  };
```

# Note on parameters

- 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);
```

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 file parts
with pre-processor
directive `ifdef __KERNEL__`

```
#define __KERNEL__
#define MODULE
#include <linux/module.h>
#include <linux/kernel.h>
......
#include <linux/smp.h>
```

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

SMP specific stuff

# Module in-use indications (classical style)

- The kernel associates 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
- If the counter is greater than zero, the unload of the module will fail (unless forcing with –f on a kernel with CONFIG_MODULE_FORCE_UNLOAD activated)
- 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 kernel 2.6 (or later)

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_get/module_put`

# Kernel exported symbols

- Either the Linux kernel or its modules can **<u>export symbols</u>**

- 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 modules) can export symbols by relying on the macro `EXPORT_SYMBOL (symbol)` which is defined in `include/linux/module.h`

# The kernel symbols table

- There exist a table including all the symbols that are available (e.g. exported)

- In general the actual symbol state can be
  - Static (not kept by the kernel table)
  - Available (kept by the kernel table)
  - Exported (kept by the kernel table and usable upon mounting modules)

- All the symbols that are currently in (or exported by) the kernel (and by its modules) are accessible via the **proc file system** through the file `/proc/kallsyms`

- This file keeps a line for each exported symbol, which has the following format

```
Kernel-memory-address symbol-type symbol-name
```

# A note on exporting symbols

- The kernel can be parameterized (compiled) to export differentiated types of symbols via standard facilities

- 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)

# Actually usable exported symbols in recent kernels

- They do not longer appear in `/proc/kallsyms`
- This is why, e.g. `sys_close,` is not actually usable while mounting modules
- The actually exported symbols are reported in

```
/lib/modules/<kernel version>/build/Module.symvers
```

- The `/proc/kallsyms` file is still useful to inspect the type of symbols within the kernel (e.g. ‘T’ vs ‘t’)

# Bypassing the symbol-architecture rules

- Up to kernel 5.7 Linux offers a module-usable interface for querying the kernel symbols table

- This is the void* kallsyms_lookup_name(const char*) exported function

- This function enables finding (and then using) any available kernel symbol

…. anyhow be carefull with the kprobe subsystem we will look  at shortly ….

# Dynamic symbols querying and kernel patching - kprobes

```
int kprobes register_kprobe(struct kprobe *)

void unregister_kprobe(struct kprobe *)

int register_kretprobe(struct kretprobe *)

int unregister_kretprobe(struct kretprobe *)
```

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

# Usage example – memory address discovery

```
// 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);
            }
```

What about discovering the position of kallsyms_lookup_name() ?!?!

# struct kprobe

```
<linux/kprobes.h>


struct kprobe {
    struct hlist_node hlist; /* Internal */
    ……
    kprobe_opcode_t addr; /* Address of probe */
    ……
    const char *symbol_name; /* probed function name */
    …
    Unsigned long missed;
    …
    kprobe_pre_handler_t pre_handler;
               /* Address of pre-handler */
    kprobe_post_handler_t post_handler;
               /* Address of post-handler */
    ………
};
```

These are pre/post the single stepped execution of the instrumented instruction/opcode

# kprobe disabling/enabling

- Each kprobe can be dynamically disabled/enabled
- The following kernel level API can be used for this task

```
int disable_kprobe(struct kprobe *kp)

int enable_kprobe(struct kprobe *kp)
```

# struct kretprobe

```
<linux/kprobes.h>


struct kretprobe {
    struct kprobe kp;
    kretprobe_handler_t handler;
    kretprobe_handler_t entry_handler;
    int maxactive;
    int nmissed;
    size_t data_size;
#ifdef CONFIG_KRETPROBE_ON_RETHOOK
    struct rethook *rh;
#else
    struct freelist_head freelist;
    struct kretprobe_holder *rph;
#endif
};
```

Return 0 on the entry handler
tells kprobe to lead
the actual handler to execute
otherwise not (e.g. a return
error is provided by the entry
handler)

This allows to control the
number of instances to be
kept concurrently alive
(-1 is the default value)

# kretprobe disabling/enabling

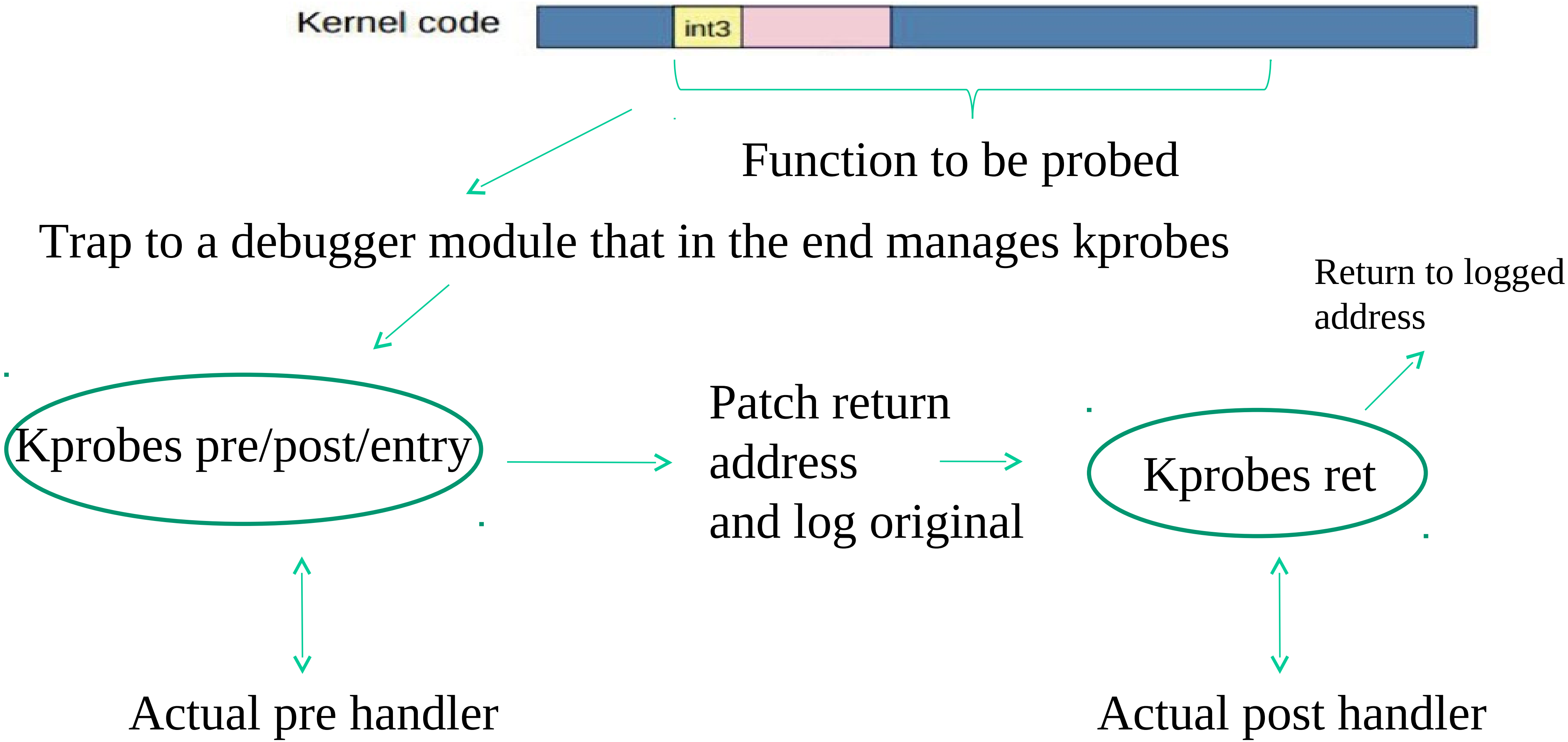
- Each kretprobe can be dynamically disabled/enabled
- The following kernel level API can be used for this task

```
static inline int disable_kretprobe(struct kretprobe *rp) {
        return disable_kprobe(&rp->kp);
}
static inline int enable_kretprobe(struct kretprobe *rp) {
        return enable_kprobe(&rp->kp);
}
```

# kprobe vs kretprobe

- kprobe  is usefull for instrumenting a single instruction in the kernel code
- anyhow it can be used to instrutment a generic function via the interception of the execution of ists very first instruction
- kretprobe is much more useful when the instrumentation process is oriented to check the start and the end of a given code block corresponding to a entire function
- what about looking at the actual parameter values of a function upon its startup? ……….. we will check with jprobes later on

# Kprobe mechanism



Function to be probed

Trap to a debugger module that in the end manages kprobes

Kprobes pre/post/entry

Patch return
address
and log original

Kprobes ret

Return to logged
address

Actual pre handler

Actual post handler

# On the maximum number of concurrent kretprobes

- The log of the return address of the original function to which the kretprobe is applied is kept into a kretprobe_instance structure

- We need one of these structures for each execution of a kreprobed function

- If this function is preempted, then the log of return address values can increase

- For kernels wth CONFIG_PREEMPT enabled the default value for the maximum number of cocnurrent kreproes is max(10, 2 * NR_CPUS)

- For non-premptable kernel confiurations the masimum number is NR_CPUS

# An interesting macro

```
struct pt_regs *regs

regs_return_value(regs)
```

←

This enables taking the return value of a function in  kretprobe handler with no need to work in machine dependent manner

# Better performing support

- The `INT3` instruction requires the management of traps (similar to what happens with `INT 0x80` for accessing the kernel code)
- `INT3` has been substituted via a jump whenever possible, for kernel scompiled with `CONFIG_OPTPROBES=y`
- This enables activating the probing system with significantly less clock cycles
- Functions that can be probed are compiled initially having a (multi-byte) `NOP` instruction
- The `NOP` instruction is (atomically) rewritten with the jump to the kernel probe entry point

# 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);

static int (*handler)
   (struct kretprobe_instance *ri, struct pt_regs *regs)

static int (*entry_handler)
   (struct kretprobe_instance *ri,
      struct pt_regs *regs)
```

Modifiable registers status

# Probing deny

- Not all kernel functions can be probed (by their name)
- A few of them are blacklisted (depending on compilation choices)
- Those that are blacklisted can be fount 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 (indirectly) triggered by probe executions

# kprobes vs preemptability

- Currently Linux supports non-preemptable kprobes (this is what happens for x86 implementations)

- It already sets up the current thread as non-preemptable before the actual kprobe code starts

- Hence the kprobe logic needs to not release the CPU via blocking services

- The motivation is that the current kprobe (its address) possibly processed by the CPU is registered into the per-CPU variable current_kprobe

- Enabling preemption would lead to a scenario where the value of current_kprobe can be overwritten with no support for kinds of stack based restoration of the original value

# Coming to jprobe

- It relies internally on kprobe
- It enables placing a wrapper to a give function in the kernel software
- The wrapper has the same signature as the wrapped function and receives exactly the same identical parameters
- It can be used for, e.g., code debug in an immediate manner
- Let’s look at an example …...

# jprobe example

```
int my_handler (…..)

………….

static struct jprobe my_probe

my_probe.kp.addr = (kprobe_opcode_t *)target_address;
my_probe.entry = (kprobe_opcode_t *)my_handler;
register_jprobe(&my_probe);
```

# 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. “4.12.14”)
  - ➢ `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`

# Kernel versioning exploitation

Compiler defined outcome

```
#if LINUX_VERSION_CODE > KERNEL_VERSION(x,y,z)
   <whatever you want to specify or include>
#else
   <whatever else you want to specify or include>
#endif
```

Programmer
specified outcome

# 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

# The Linux kernel messaging system

- Kernel level software can provide output messages in relation to events occurring during the execution
- The messages can be produced both during initialization and steady state operations, hence
    - ➢ Sofware modules forming the messaging system cannot rely on I/O standard services (such as `sys_write() or kernel_write()`)
    - ➢ No standard library function can be used for output production
- Management of kernel level messages occurs via specific modules that take care of the following tasks
    - ➢ Message print onto the “console” device
    - ➢ Message logging into a circular buffer kept within kernel level virtual addresses

# The printk() function

- The kernel level module for producing output messages is called `printk()` and is defined within the file `kernel/printk.c`

- This function accepts an input parameter representing a format string, which is similar to the one used for the `printf()` standard library function

- The major difference is that with `printk()` we cannot specify floating point values (these are unallowed in kernel toolchains)

- The format string optionally entails an indication in relation to the priority (or criticality) level for the output message

- The message priority level can be specified via macros (expanded as strings) which can be pre-fixed to the arguments passed in input to `printk()`

# Message priority levels

- The macros specifying the priority levels are defined in the `include/linux/kernel.h` header file

```
#define KERN_EMERG "<0>"     /* system is unusable */
#define KERN_ALERT "<1>"    /* action must be taken  immediately */
#define KERN_CRIT "<2>"      /* critical conditions */
#define KERN_ERR "<3>"       /* error conditions */
#define KERN_WARNING "<4>" /* warning conditions */
#define KERN_NOTICE "<5>" /* normal but significant condition */
#define KERN_INFO "<6>"    /* informational */
#define KERN_DEBUG "<7>"  /* debug-level messages */
```

- One usage example

```
printk(KERN_WARNING  “message to print”)
```

# A few details on data format for pointers (addresses)

- kernel level printing of addresses is an aspect to be carefully considered, especially for security

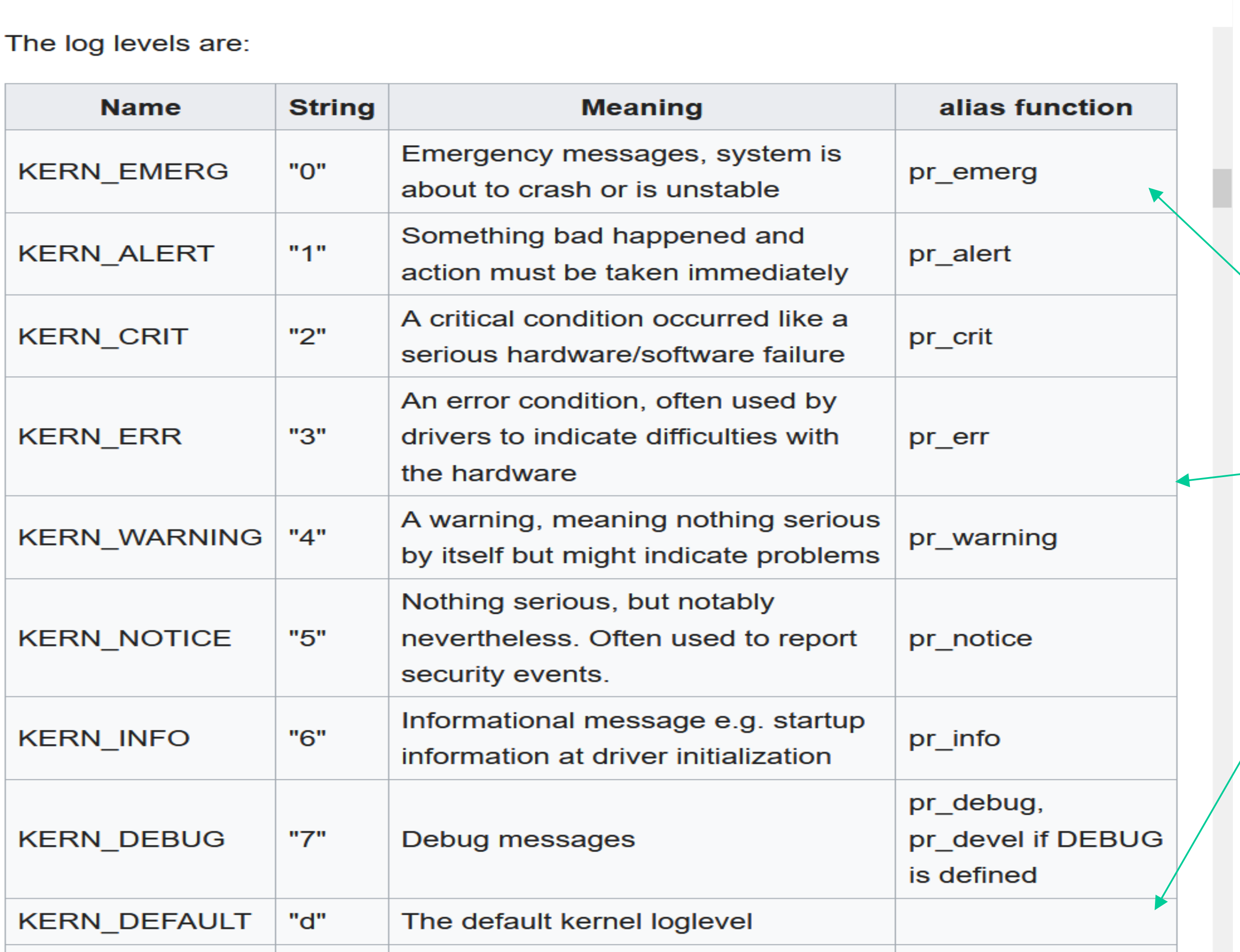
- current configurations of the `printk` service make any pointer to be encrypted when printing using `%p`

- otherwise we might diffuse memory positioning of kernel-level information (e.g. the address of a function) too easily

- using `%p` the encryption is typically deterministic with respect to the actual pointer value to be printed

- to avoid encryption `%px` can be used

# Function aliases (via macros)



The log levels are:

| Name | String | Meaning | alias function |
|---|---|---|---|
| KERN_EMERG | "0" | Emergency messages, system is about to crash or is unstable | pr_emerg |
| KERN_ALERT | "1" | Something bad happened and action must be taken immediately | pr_alert |
| KERN_CRIT | "2" | A critical condition occurred like a serious hardware/software failure | pr_crit |
| KERN_ERR | "3" | An error condition, often used by drivers to indicate difficulties with the hardware | pr_err |
| KERN_WARNING | "4" | A warning, meaning nothing serious by itself but might indicate problems | pr_warning |
| KERN_NOTICE | "5" | Nothing serious, but notably nevertheless. Often used to report security events. | pr_notice |
| KERN_INFO | "6" | Informational message e.g. startup information at driver initialization | pr_info |
| KERN_DEBUG | "7" | Debug messages | pr_debug, pr_devel if DEBUG is defined |
| KERN_DEFAULT | "d" | The default kernel loglevel | |

The aliases automatically generate the priority string

# Message priority treatment

- There exist 4 configurable parameters which determine actual output-message treatment

- They are associated with the following variables
  - `console_loglevel` (this is the level under which the messages are actually logged on the console device)
  - `default_message_loglevel` (this is the priority level that gets associated by default with any message not specifying any priority value explicitly)
  - `minimum_console_loglevel` (this is the minimum level for admitting the log of messages onto the console device)
  - `default_console_loglevel` (this is the default level for messages destined to the console device)

# Inspecting the current log level settings

- Look at the special file `/proc/sys/kernel/printk`

- Write into this file for modifications of these parameters (if supported by the specific kernel version/configuration)

- This is not a real stable storage file (updates need to be reissued or need to be implemented at kernel startup)

# console_loglevel

- Typically `console_loglevel` is associated with the value 7 (this settings is anyhow non-mandatory)
- Hence all messages, except debug messages, need to be shown onto the console device
- Setting this parameter to the value 8 enables printing debug messages onto the console device
- Setting this parameter to the value 1 any message is disabled to be logged onto the console, except emergency messages

# Circular buffer management

```
int syslog(int type, char *bufp, int len);
```

- This is the system call for performing management operation onto the kernel level circular buffer hosting output messages
- the `bufp` parameter points to the memory area where the bytes read from the circular buffer needs to be logged
- `len` specifies how many bytes we are interested in or a flag (depending on the value of `type`)
- for `type` we have the following options → ….

**SYSLOG_ACTION_CLOSE** (0) Close the log. Currently a NOP.

**SYSLOG_ACTION_OPEN** (1) Open the log. Currently a NOP.

**SYSLOG_ACTION_READ** (2) Read from the log.
The call waits until the kernel log buffer is nonempty, and then reads at most *len* bytes into the buffer pointed to by *bufp*. The call returns the number of bytes read. Bytes read from the log disappear from the log buffer: the information can be read only once. This is the function executed by the kernel when a user program reads */proc/kmsg*.

**SYSLOG_ACTION_READ_ALL** (3) Read all messages remaining in the ring buffer, placing them in the buffer pointed to by *bufp*. The call reads the last *len* bytes from the log buffer (nondestructively), but will not read more than was written into the buffer since the last "clear ring buffer" command (see command 5 below)). The call returns the number of bytes read.

**SYSLOG_ACTION_READ_CLEAR** (4) Read and clear all messages remaining in the ring buffer. The call does precisely the same as for a *type* of 3, but also executes the "clear ring buffer" command.

**SYSLOG_ACTION_CLEAR** (5) The call executes just the "clear ring buffer" command. The *bufp* and *len* arguments are ignored. This command does not really clear the ring buffer. Rather, it sets a kernel bookkeeping variable that determines the results returned by commands 3 (**SYSLOG_ACTION_READ_ALL**) and 4 (**SYSLOG_ACTION_READ_CLEAR**). This command has no effect on commands 2 (**SYSLOG_ACTION_READ**) and 9 (**SYSLOG_ACTION_SIZE_UNREAD**).

**SYSLOG_ACTION_CONSOLE_OFF** (6) The command saves the current value of *console_loglevel* and then sets *console_loglevel* to *minimum_console_loglevel*, so that no messages are printed to the console. Before Linux 2.6.32, the command simply sets *console_loglevel* to *minimum_console_loglevel*. See the discussion of */proc/sys/kernel/printk*, below. The *bufp* and *len* arguments are ignored.

**SYSLOG_ACTION_CONSOLE_ON** (7) If a previous **SYSLOG_ACTION_CONSOLE_OFF** command has been performed, this command restores *console_loglevel* to the value that was saved by that command. Before Linux 2.6.32, this command simply sets *console_loglevel* to *default_console_loglevel*. See the discussion of */proc/sys/kernel/printk*, below. The *bufp* and *len* arguments are ignored.

**SYSLOG_ACTION_CONSOLE_LEVEL** (8) The call sets *console_loglevel* to the value given in *len*, which must be an integer between 1 and 8 (inclusive). The kernel silently enforces a minimum value of *minimum_console_loglevel* for *len*. See the *log level* section for details. The *bufp* argument is ignored.

**SYSLOG_ACTION_SIZE_UNREAD** (9) (since Linux 2.4.10) The call returns the number of bytes currently available to be read from the kernel log buffer via command 2 (**SYSLOG_ACTION_READ**). The *bufp* and *len* arguments are ignored.

**SYSLOG_ACTION_SIZE_BUFFER** (10) (since Linux 2.6.6) This command returns the total size of the kernel log buffer. The *bufp* and *len* arguments are ignored.

# Updates of console_loglevel

`console_loglevel` can be set (to a value in the range 1-8) by the call **syslog**() (8,*dummy*,*value*)

The calls **syslog**() (*type*,*dummy*,*dummy*) with *type* equal to 6 or 7, set it to 1 (kernel panics only) or 7 (all except debugging messages), respectively

# Messaging management demon

**klogd - Kernel Log Daemon**

SYNOPSIS

klogd [ -c n ] [ -d ] [ -f fname ] [ -iI ] [ -n ] [ -o ] [ -p ] [ -s ] [ -k fname ] [ -v ] [ -x ] [ -2 ]

DESCRIPTION

klogd is a system daemon which intercepts and logs Linux kernel messages

# Circular buffer features

- The circular buffer keeping the kernel output messages has size that varies over time
  - originally 4096 bytes,
  - Since kernel version 1.3.54, we had up to 8192 bytes,
  - Since kernel version 2.1.113, we had  up to 16384 bytes … much more in more recent versions
- A unique buffer is used for any message, independently of the message priority level
- The buffer content can be accessed by also relying on the shell command “dmesg”

# Actual management of messages

- In order to enable the delivery of messages with exactly-once semantic, message printing onto the console is executed synchronously (recall that standard library functions only enable at-most-once semantic, just due to asynchronous management)

- Hence the `printk()` function does not return control until the message is delivered to any active console-device driver

- The driver, in its turn does not return control until the message is actually sent to the (physical) console device

- NOTE: this may impact performance

  ➢ As an example, the delivery of a message on a serial console device working at 9600 bit per second, slows down system speed by 1 millisecond per char

# The long story of printk()

- While it may appear a simple subsystem, printk() is extremely complex
- As an example, it should be usable in any context (including interrupt contexts)
- Hence its service is non-blocking
- At the same time, the reliance in spinlocks for protecting, e.g. the ring buffer appears to be non-scalable
- The architectural design of the printk() subsystem is always ongoing
- Currently there is attention to lockless management + other optimizations

# The panic() function

- The `panic()` function is defined in `kernel/panic.c`
- This function prints the specified message onto the console device (by relying on `printk()`)
- The string "*Kernel panic:*" is prefixed to the message
- Further, this function halts the machine, hence leading to stopping the execution of the kernel