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

Contenuti:

1. Audit/debugging supports within the LINUX kernel: printk() and panic() functions

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()`)
  - 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 different is that with `printk()` we cannot specify floating point values
- 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”)
```

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 specific priority value)
  - `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: `syslog()`

```
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:

```
/*
* Commands to sys_syslog:
*
*      0 -- Close the log.  Currently a NOP.
*      1 -- Open the log. Currently a NOP.
*      2 -- Read from the log.
*      3 -- Read up to the last 4k
                  of messages in the ring buffer.
*      4 -- Read and clear last 4k
                  of messages in the ring buffer
*      5 -- Clear ring buffer.
*      6 -- Disable printk's to console
*      7 -- Enable printk's to console
       8 -- Set level of messages printed
                  to console
*/
```

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 `LOG_BUF_LEN`, which is defined in `kernel/printk.c`
  - 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
- 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 `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