Kernel debugging config options

- There are number of features in Linux related to debugging
- Most of them are consolidated under “Kernel Hacking” Kconfig menu
  - CONFIG_DEBUG_KERNEL — master switch
  - CONFIG_DEBUG_SLAB Each byte of allocated memory is set to special values, guard are added before and after allocated memory object.
  - CONFIG_INIT_DEBUG Enables checks for code that attempts to access initialization-time memory after initialization is complete
  - CONFIG_DEBUG_LIST Adds sanity check into list functions

And many others…

Kernel will be slower but be able to catch and report errors.

Kernel debugging config options (cont.)

```
#ifdef CONFIG_DEBUG_LIST
extern bool __list_del_entry_valid(struct list_head *entry);
#else
static inline bool __list_del_entry_valid(struct list_head *entry)
{
    return true;
}
#endif

static inline void __list_del_entry(struct list_head *entry)
{
    if (!__list_del_entry_valid(entry))
      return;

    __list_del(entry->prev, entry->next);
}
```

Controlling printk() behaviour

- Here is how message loglevel is defined in the Linux Kernel

0 (KERN_EMERG)	system is unusable
1 (KERN_ALERT)	action must be taken immediately
2 (KERN_CRIT)	critical conditions
3 (KERN_ERR)	error conditions
4 (KERN_WARNING)	warning conditions
5 (KERN_NOTICE)	normal but significant condition
6 (KERN_INFO)	informational
7 (KERN_DEBUG)	debug-level messages

- All Kernel Messages with a loglevel smaller than the console loglevel will be printed to the console. One can change the limit using:

```
# echo 4 > /proc/sys/kernel/printk
```

Controlling printk() behaviour (cont.)

- One interesting thing we should note explicitly `pr_fmt()` preprocessor macro in `<linux/printk.h>`

```
#ifndef pr_fmt
#define pr_fmt(fmt) fmt
#endif
```

- Your code should define custom `pr_fmt()` preprocessor macro that provides string if custom prefix for `pr_debug()` is desired

```
#define pr_fmt(fmt) KBUILD_MODNAME ": " fmt
```

- That would produce following messages with `pr_debug()` in kernel ring buffer

```
module_name_without_ko: fmt_message_goes_here
```

- One can even use

```
#define pr_fmt(fmt) KBUILD_MODNAME ":%s:%d: " fmt, __func__, __LINE__
```

Controlling printk() behaviour (cont.)

- There are number of preprocessor macro in pr_*() family that can be used instead of printk() in most cases. Main benefit from using of such functions is much simpler and cleaner code. They defined in `<linux/printk.h>` too, for example:

```
#define pr_crit(fmt, ...) \
    printk(KERN_CRIT pr_fmt(fmt), ##__VA_ARGS__)
#define pr_err(fmt, ...) \
    printk(KERN_ERR pr_fmt(fmt), ##__VA_ARGS__)
#define pr_warning(fmt, ...) \
    printk(KERN_WARNING pr_fmt(fmt), ##__VA_ARGS__)
#define pr_warn pr_warning
#define pr_info(fmt, ...) \
    printk(KERN_INFO pr_fmt(fmt), ##__VA_ARGS__)
```

Controlling printk() behaviour (cont.)

- There are also macros that print message only once. They use printk_once() macro in their implementation

```
#define printk_once(fmt, ...)                        \
({                                                   \
     static bool __print_once __read_mostly;         \
                                                     \
     if (!__print_once) {                            \
          __print_once = true;                       \
          printk(fmt, ##__VA_ARGS__);                \
                                                     \
     }                                               \
})
```

- and there are helpers, for example

```
#define pr_err_once(fmt, ...)                                      \
    printk_once(KERN_ERR pr_fmt(fmt), ##__VA_ARGS__)
#define pr_warn_once(fmt, ...)                                     \
    printk_once(KERN_WARNING pr_fmt(fmt), ##__VA_ARGS__)
```

Controlling printk() behaviour (cont.)

- printk itself is fast. It just prints into string and places the string in circular buffer. Message printing might be quite slow especially for serial console.
  - printk to buffer: a few microseconds (but this time can be sufficient if there are a lot of messages)
  - printk to serial console: a few **milli**seconds
- There is also ratelimited version of printk. Use it in code paths where message might be generated too often (e.g. packet coming from network and we print message to ring buffer).

```
#define printk_ratelimited(fmt, ...)                              \
({                                                                \
     static DEFINE_RATELIMIT_STATE(_rs,                           \
                                   DEFAULT_RATELIMIT_INTERVAL,    \
                                   DEFAULT_RATELIMIT_BURST);      \
                                                                  \
     if (__ratelimit(&_rs))                                       \
      printk(fmt, ##__VA_ARGS__);                                 \
})
```

Controlling printk() behaviour (cont.)

- Also, there are corresponding helpers

```
#define pr_err_ratelimited(fmt, ...)                              \
    printk_ratelimited(KERN_ERR pr_fmt(fmt), ##__VA_ARGS__)
...
```

- For device drivers there is a special set of printk() functions
- They are prefixed with dev_*()
- They defined in `<linux/device.h>`
- Use them in your device driver
- Same applies to network subsystem where net_ratelimited_*() functions should be used
- They defined in <linux/net.h>
- Use them in your network device driver

Controlling printk() behaviour (cont.)

- There is also interface in proc filesystem to control printk() behaviour
  - One can delay each message printing by specified number of milliseconds via sysctl -w kernel.printk_delay = <msec> knob
  - One can control message rate limiting by means of sysctl -w kernel.printk_ratelimit = <jiffies> and sysctl -w kernel.printk_ratelimit_burst = <number>
- Also console log level can be specified
  - Via Linux command line parameter loglevel=
  - Via sysctl -w kernel.printk = c d m f

■ (c) console_loglevel	messages with higher priority will be printed
■ (d) default_message_loglevel	messages without explicit priority assigned this one
■ (m) minimum_console_loglevel	minimum (highest) that console_loglevel can be set
■ (f) default_console_loglevel	default value of console_loglevel

  - Via dmesg(1) -n or --console-level parameter

Debugfs: A memory mapped file system

- It is used by kernel developers to put/get information from kernel code to user space
- There is no rules that force any restriction on format or contents of data in debugfs: developers are free to put information that they wish in format whey think is best
- As such none of interfaces in debugfs should be considered as API and to be used in user space for that purpose. The only purpose of interfaces in debugfs is to provide information from kernel to user space useful for debugging kernel interfaces
- Assuming above debugfs root available only to superuser by default

Debugfs: A memory mapped file system (cont.)

- It is mounted by super user from user space via following command (see /etc/init.d/rcS in our busybox directories)

  ```
  mount -t debugfs none /sys/kernel/debug
  ```

- However there is options uid, gid and mode to make debugfs root available to non super user. Also it is possible to mount it to location other than `/sys/kernel/debug`
- There is another restriction with using debugfs from your kernel code:
  - it’s API available to GPL-only code, which means that modules with proprietary or “dual” licensing can not use debugfs API
- All debugfs API is available via inclusion of `<linux/debugfs.h>` in your code

Debugfs: A memory mapped file system (cont.)

- There are number of places in Linux Kernel where debugfs is used, as well as in many external third-party modules
- We will not cover debugfs API for kernel code right now, as it requires bit understanding of Linux Virtual File System (VFS) interface
- However we will use debugfs interfaces provided by various subsystems and features in Linux Kernel.
- One of such wonderful feature is dynamic debugging facility that permits for low overhead debug messages insertion in source code. It uses debugfs to control debug messages generation scopes, severity levels and more.
- You may refer to Documentation/filesystems/debugfs.txt for more information.
- We will not cover debugfs API for kernel code right now, as it requires bit understanding of Linux Virtual File System (VFS) interface.
- However we will use debugfs interfaces provided by various subsystems and features in Linux Kernel.

dyndbg: Dynamic debugging

- It is quite common to have various messages in user space source code that are print to either console or to dedicated log file(s)
- Developing code for kernel does not differ in this approach from user space too much. However it is not specific to write debug messages to file(s)
- On the other hand there is need to control when to output this kind of messages from code
- Also printing huge amount of messages may affect system performance
- Assuming all the above flexible mechanism to generate and control generation is added to Linux Kernel. That mechanism is called dynamic debugging (dyndbg for short)

dyndbg: Dynamic debugging (cont.)

- Previously printk() is used to implement debug message output
- And each piece of code in-tree and out-of-tree kernel code invent their own wheel to do that
- With old, still supported, but not preferred approach code did something like

```
#ifdef DEBUG
#define DPRINTF(fmt, args...)                           \
    do {                                                \
        if (printk_ratelimit())                         \
            printk(KERN_DEBUG fmt, ## args);            \
    } while (0);                                        \
#endif /* DEBUG */
```

- With new all this stuff replaced with pr_debug() and family functions

dyndbg: Dynamic debugging (cont.)

- Here is `pr_debug()` definition in `<linux/printk.h>`

```
/* If you are writing a driver, please use dev_dbg instead */
#if defined(CONFIG_DYNAMIC_DEBUG)
/* dynamic_pr_debug() uses pr_fmt() internally so we don't need it here
*/
#define pr_debug(fmt, ...) \
    dynamic_pr_debug(fmt, ##__VA_ARGS__)
#elif defined(DEBUG)
#define pr_debug(fmt, ...) \
    printk(KERN_DEBUG pr_fmt(fmt), ##__VA_ARGS__)
#else
#define pr_debug(fmt, ...) \
    no_printk(KERN_DEBUG pr_fmt(fmt), ##__VA_ARGS__)
#endif
```

dyndbg: Dynamic debugging (cont.)

- There is also pr_devel() preprocessor macro that evaluates to printk() if DEBUG is on and to no_printk() when not defined

```
/* pr_devel() should produce zero code unless DEBUG is defined */
#ifdef DEBUG
#define pr_devel(fmt, ...) \
    printk(KERN_DEBUG pr_fmt(fmt), ##__VA_ARGS__)
#else
#define pr_devel(fmt, ...) \
    no_printk(KERN_DEBUG pr_fmt(fmt), ##__VA_ARGS__)
#endif
```

dyndbg: Dynamic debugging (cont.)

- and no_printk() is just a dummy helper inline function to let gcc to check fmt arguments. It is used when CONFIG_PRINTK isn’t active

```
/*
 * Dummy printk for disabled debugging statements to use whilst
maintaining
 * gcc's format and side-effect checking.
 */
static inline __printf(1, 2)
int no_printk(const char *fmt, ...)
{
    return 0;
}
```

- It is defined in `<linux/printk.h>` too

dyndbg: Dynamic debugging (cont.)

- Dynamic debug is compiled when CONFIG_DYNAMIC_DEBUG is active
- By default no pr_debug() messages would be printed
- To control what messages to print at early boot stages where debugfs isn’t initialized one can pass parameters via Linux command line from bootloader

```
# dyndbg parameter is used to activate pr_debug() in non-module code
# module.dyndbg format is used to activate pr_debug() in modules
# (including built-in)

… dyndbg=QUERY module_name.dyndbg=QUERY ...
```

- When debugfs is available one can control where pr_debug() is active by reading/writing to /sys/kernel/debug/dynamic_debug/control.

dyndbg: Dynamic debugging (cont.)

- There is special format of QUERY messages going to either command line parameter or to dynamic_debug/control file

```
$ sysfs='/sys/kernel/debug'
$ echo "command; command; command; ..." >$sysfs/dynamic_debug/control

# command ::= match-spec* flags-spec
#
# match-spec ::= 'func' string |
#                'file' string |
#                'module' string |
#                'format' string |
#                'line' line-range
#
# line-range ::= lineno | '-'lineno | lineno'-' | lineno'-'lineno
#
# lineno ::= unsigned-int
```

dyndbg: Dynamic debugging (cont.)

- The flags specification comprises a change operation followed by one or more flag characters. The change operation is one of the characters:

-	remove the given flags
+	add the given flags
=	set the flags to the given flags

- The flags are:

p	enables the pr_debug() callsite.
f	Include the function name in the printed message
l	Include line number in the printed message
m	Include module name in the printed message
t	Include thread ID in messages not generated from interrupt context
_	No flags are set. (Or'd with others on input)

- Refer to Documentation/dynamic-debug-howto.txt as more complete source of information about dynamic debugging feature

dyndbg: Dynamic debugging (cont.)

- Let’s replace pr_info() by pr_debug() in hello_exit()
- Do not enable debug messages:

```
~ # insmod hello.ko count=2
[   20.750420] hello: loading out-of-tree module taints kernel.
[   20.757132] hello_init: Hello, world!
[   20.759883] hello_init: Hello, world!
~ # cat /sys/kernel/debug/dynamic_debug/control  | grep hello.c
..../hello.c:85 [hello]hello_exit =_ "%s: %lld\012"
~ # rmmod hello
~ # dmesg | tail -2
[   20.757132] hello_init: Hello, world!
[   20.759883] hello_init: Hello, world!
~ #
```

dyndbg: Dynamic debugging (cont.)

- Enable debug messages:

```
~ # insmod hello.ko count=2
[   54.008406] hello_init: Hello, world!
[   54.011169] hello_init: Hello, world!
~ # echo 'file hello.c line 85 +p' > /sys/kernel/debug/dynamic_debug/control
~ # cat /sys/kernel/debug/dynamic_debug/control  | grep hello.c
..../hello.c:85 [hello]hello_exit =p "%s: %lld\012"
~ # rmmod hello
~ # dmesg | tail -3
[   54.011169] hello_init: Hello, world!
[   78.001123] hello_exit: 54005378298
[   78.001138] hello_exit: 54008150089
~ #
```