LDD3, Ch 4 & 11

ECEN 427

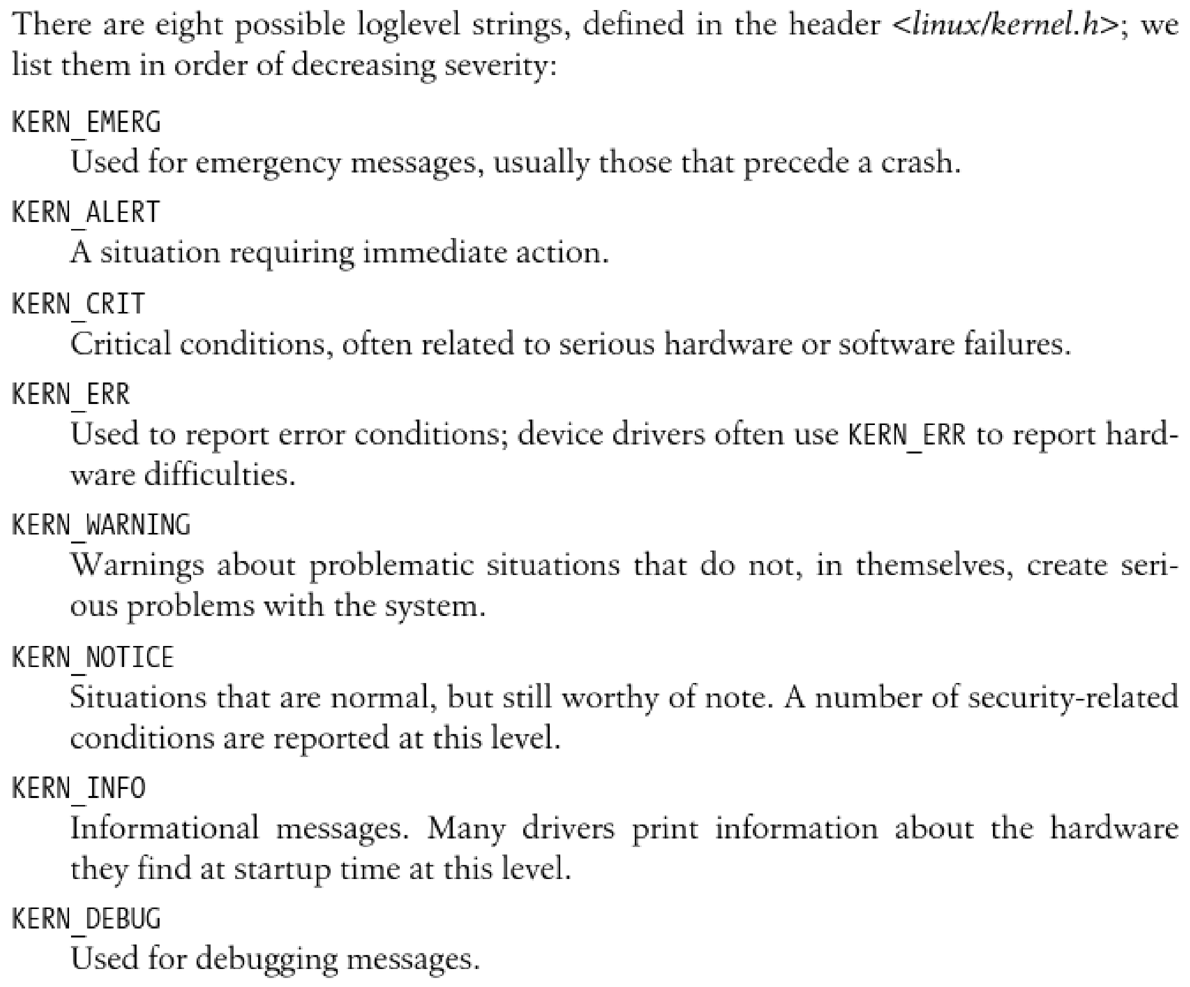
BYU Electrical & Computer Engineering
IRA A. FULTON COLLEGE OF ENGINEERING

Ch 4 Debugging

BYU Electrical & Computer Engineering
IRA A. FULTON COLLEGE OF ENGINEERING

printk

- One of the differences is that printk lets you classify messages according to their severity by associating different loglevels, or priorities, with the messages. You usually indicate the loglevel with a macro.



There are eight possible loglevel strings, defined in the header *<linux/kernel.h>*; we list them in order of decreasing severity:

`KERN_EMERG`
: Used for emergency messages, usually those that precede a crash.

`KERN_ALERT`
: A situation requiring immediate action.

`KERN_CRIT`
: Critical conditions, often related to serious hardware or software failures.

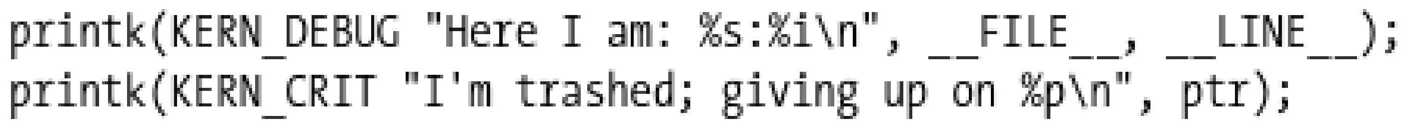
`KERN_ERR`
: Used to report error conditions; device drivers often use `KERN_ERR` to report hardware difficulties.

`KERN_WARNING`
: Warnings about problematic situations that do not, in themselves, create serious problems with the system.

`KERN_NOTICE`
: Situations that are normal, but still worthy of note. A number of security-related conditions are reported at this level.

`KERN_INFO`
: Informational messages. Many drivers print information about the hardware they find at startup time at this level.

`KERN_DEBUG`
: Used for debugging messages.



```
printk(KERN_DEBUG "Here I am: %s:%i\n", __FILE__, __LINE__);
printk(KERN_CRIT "I'm trashed; giving up on %p\n", ptr);
```

Kernel Log

- The printk function writes messages into a circular buffer that is __LOG_BUF_LEN bytes long: a value from 4 KB to 1 MB chosen while configuring the kernel.
- The **dmesg** command can be used to look at the content of the buffer without flushing it; actually, the command returns to stdout the whole content of the buffer, whether or not it has already been read.
- If the circular buffer fills up, printk wraps around and starts adding new data to the beginning of the buffer, overwriting the oldest data

Kernel or No Kernel?

BYU Electrical & Computer Engineering
IRA A. FULTON COLLEGE OF ENGINEERING

- How to globally disable printing in your kernel module?

```
#undef PDEBUG             /* undef it, just in case */
#ifdef SCULL_DEBUG
#  ifdef __KERNEL__
     /* This one if debugging is on, and kernel space */
#    define PDEBUG(fmt, args...) printk( KERN_DEBUG "scull: " fmt, ## args)
#  else
     /* This one for user space */
#    define PDEBUG(fmt, args...) fprintf(stderr, fmt, ## args)
#  endif
#else
#  define PDEBUG(fmt, args...) /* not debugging: nothing */
#endif

#undef PDEBUGG
#define PDEBUGG(fmt, args...) /* nothing: it's a placeholder */
```

Makefile changes

```
# Comment/uncomment the following line to disable/enable debugging
DEBUG = y

# Add your debugging flag (or not) to CFLAGS
ifeq ($(DEBUG),y)
  DEBFLAGS = -O -g -DSCULL_DEBUG # "-O" is needed to expand inlines
else
  DEBFLAGS = -O2
endif

CFLAGS += $(DEBFLAGS)
```

Rate Limiting

- "If you are not careful, you can find yourself generating thousands of messages with printk, overwhelming the console and, possibly, overflowing the system log file."

- "When using a slow console device (e.g., a serial port), an excessive message rate can also slow down the system or just make it unresponsive.

- What should you do?
  - In many cases, the best behavior is to set a flag saying, "I have already complained about this," and not print any further messages once the flag gets set
  - In others, though, there are reasons to emit an occasional "the device is still broken" notice. The kernel has provided a function that can be helpful in such cases:

```
int printk_ratelimit(void);
```

  - This function should be called before you consider printing a message that could be repeated often

```
if (printk_ratelimit())
    printk(KERN_NOTICE "The printer is still on fire\n");
```

pr_*, dev_*

Don't call printk directly!

https://www.kernel.org/doc/html/latest/process/coding-style.html#printing-kernel-messages

There are a number of driver model diagnostic macros in <linux/dev_printk.h> which you should use to make sure messages are matched to the right device and driver, and are tagged with the right level:

- `dev_err(), dev_warn(), dev_info(),` and so forth
- https://elixir.bootlin.com/linux/v5.4/source/include/linux/device.h#L1740

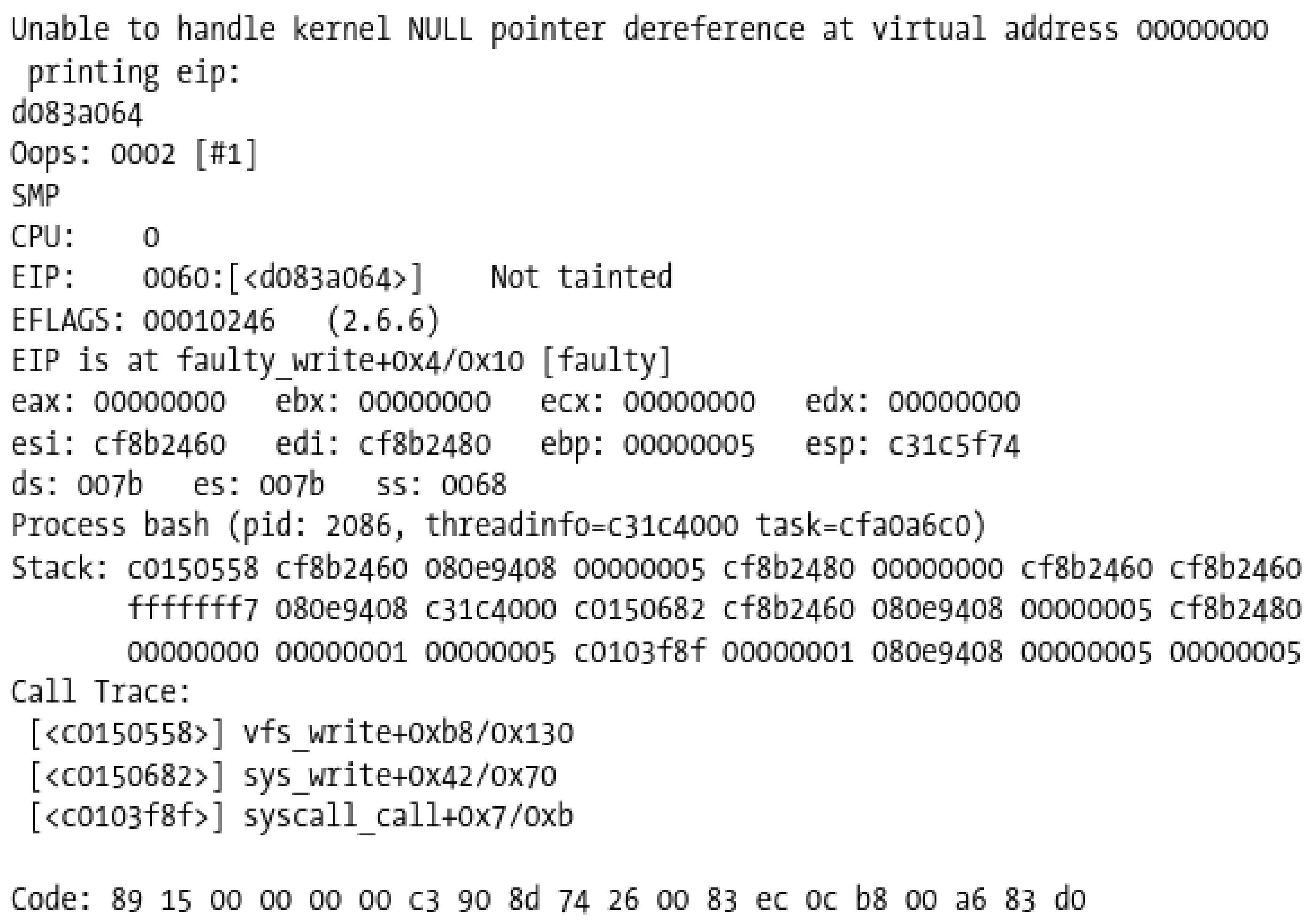
For messages that aren't associated with a particular device, <linux/printk.h> defines:

- `pr_notice(), pr_info(), pr_warn(), pr_err()`, etc.
- https://elixir.bootlin.com/linux/v5.4/source/include/linux/printk.h#L303

Oops Messages

BYU Electrical & Computer Engineering
IRA A. FULTON COLLEGE OF ENGINEERING

- “Most bugs show themselves in NULL pointer dereferences or by the use of other incorrect pointer values. The usual outcome of such bugs is an oops message.”



```
Unable to handle kernel NULL pointer dereference at virtual address 00000000
 printing eip:
d083a064
Oops: 0002 [#1]
SMP
CPU:    0
EIP:    0060:[<d083a064>]    Not tainted
EFLAGS: 00010246   (2.6.6)
EIP is at faulty_write+0x4/0x10 [faulty]
eax: 00000000   ebx: 00000000   ecx: 00000000   edx: 00000000
esi: cf8b2460   edi: cf8b2480   ebp: 00000005   esp: c31c5f74
ds: 007b   es: 007b   ss: 0068
Process bash (pid: 2086, threadinfo=c31c4000 task=cfa0a6c0)
Stack: c0150558 cf8b2460 080e9408 00000005 cf8b2480 00000000 cf8b2460 cf8b2460
       fffffff7 080e9408 c31c4000 c0150682 cf8b2460 080e9408 00000005 cf8b2480
       00000000 00000001 00000005 c0103f8f 00000001 080e9408 00000005 00000005
Call Trace:
 [<c0150558>] vfs_write+0xb8/0x130
 [<c0150682>] sys_write+0x42/0x70
 [<c0103f8f>] syscall_call+0x7/0xb

Code: 89 15 00 00 00 00 c3 90 8d 74 26 00 83 ec 0c b8 00 a6 83 d0
```

Ch 11: Data Types

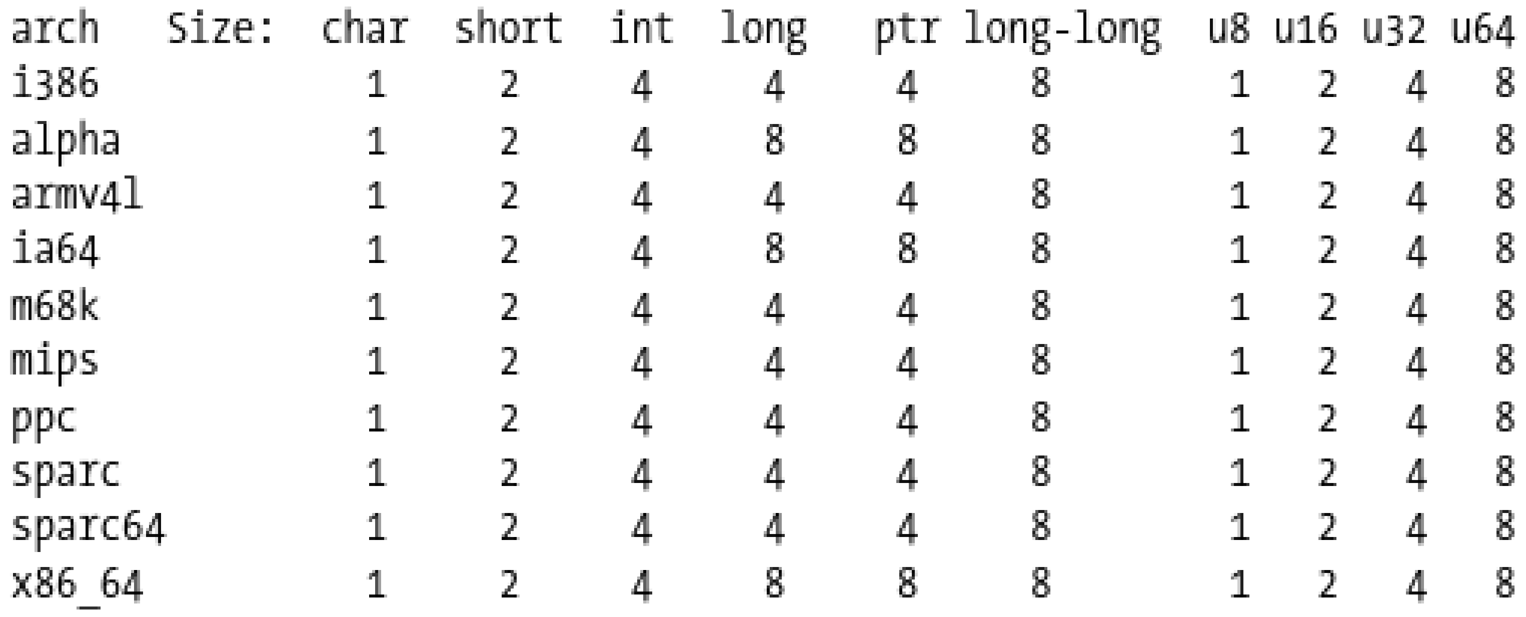
BYU Electrical & Computer Engineering
IRA A. FULTON COLLEGE OF ENGINEERING

Data Types

- Data types used by kernel data are divided into three main classes:

1. standard C types such as int,
2. explicitly sized types such as u32, and
3. types used for specific kernel objects, such as pid_t.

- The problem is that you can’t use the standard types when you need “a 2-byte filler” or “something representing a 4-byte string,” because the normal C data types are not the same size on all architectures.



```
arch    Size:  char  short  int  long   ptr long-long  u8 u16 u32 u64
i386             1     2     4     4     4     8        1   2   4   8
alpha            1     2     4     8     8     8        1   2   4   8
armv4l           1     2     4     4     4     8        1   2   4   8
ia64             1     2     4     8     8     8        1   2   4   8
m68k             1     2     4     4     4     8        1   2   4   8
mips             1     2     4     4     4     8        1   2   4   8
ppc              1     2     4     4     4     8        1   2   4   8
sparc            1     2     4     4     4     8        1   2   4   8
sparc64          1     2     4     4     4     8        1   2   4   8
x86_64           1     2     4     8     8     8        1   2   4   8
```

- It’s interesting to note that the SPARC 64 architecture runs with a 32-bit user space, so pointers are 32 bits wide there, even though they are 64 bits wide in kernel space.

Although you must be careful when mixing different data types, sometimes there are good reasons to do so. One such situation is for memory addresses, which are special as far as the kernel is concerned.

Although, conceptually, addresses are pointers, memory administration is often better accomplished by using an unsigned integer type; the kernel treats physical memory like a huge array, and a memory address is just an index into the array.

Furthermore, a pointer is easily dereferenced; when dealing directly with memory addresses, you almost never want to dereference them in this manner. Using an integer type prevents this dereferencing, thus avoiding bugs.

Therefore, generic memory addresses in the kernel are usually **unsigned long**, exploiting the fact that **pointers and long integers are always the same size, at least on all the platforms currently supported by Linux**.

Explicit Size Types

- Sometimes kernel code requires data items of a specific size, perhaps to match predefined binary structures,* to communicate with user space, or to align data within structures by inserting “padding” fields

- The kernel offers the following data types to use whenever you need to know the size of your data. All the types are declared in <asm/types.h>, which, in turn, is included by <linux/types.h>:

```
u8;    /* unsigned byte (8 bits) */
u16;   /* unsigned word (16 bits) */
u32;   /* unsigned 32-bit value */
u64;   /* unsigned 64-bit value */
```

- If a user-space program needs to use these types, it can prefix the names with a double underscore: __u8 and the other types are defined independent of __KERNEL__. If, for example, a driver needs to exchange binary structures with a program running in user space by means of ioctl, the header files should declare 32-bit fields in the structures as __u32.

- It's important to remember that these types are Linux specific, and using them hinders porting software to other Unix flavors. Systems with recent compilers support the C99-standard types, such as uint8_t and uint32_t; if portability is a concern, those types can be used in favor of the Linux-specific variety.

Interface Specific types

- Some of the commonly used data types in the kernel have their own typedef statements, thus preventing any portability problems. For example, a process identifier (pid) is usually pid_t instead of int.

- Note that, in recent times, relatively few new interface-specific types have been defined. Use of the typedef statement has gone out of favor among many kernel developers, who would rather see the real type information used directly in the code, rather than hidden behind a user-defined type. Many older interface-specific types remain in the kernel, however, and they will not be going away anytime soon.

Printing Interface Specific Types

BYU Electrical & Computer Engineering
IRA A. FULTON COLLEGE OF ENGINEERING

- The main problem with _t data items is that when you need to print them, it’s not always easy to choose the right printk or printf format, and warnings you resolve on one architecture reappear on another.

- For example, how would you print a size_t, that is unsigned long on some platforms and unsigned int on some others?

- Whenever you need to print some interface-specific data, the best way to do it is by **casting the value to the biggest possible type** (usually long or unsigned long) and then printing it through the corresponding format.

Pointer Return Types

- Many internal kernel functions return a pointer value to the caller. Many of those functions can also fail. In ***most*** cases, failure is indicated by returning a NULL pointer value. This technique works, but it is unable to communicate the exact nature of the problem. Some interfaces really need to return an actual error code so that the caller can make the right decision based on what actually went wrong.

- A number of kernel interfaces return this information by **encoding the error code in a pointer value**. Such functions must be used with **care**, since their return value cannot simply be compared against NULL. To help in the creation and use of this sort of interface, a small set of functions has been made available (in <linux/err.h>).

- The caller can use IS_ERR to test whether a returned pointer is an error code or not:
  - `long IS_ERR(const void *ptr);`
- If you need the actual error code, it can be extracted with:
  - `long PTR_ERR(const void *ptr);`