Linux Device Driver
(Debugging Techniques)

Amir Hossein Payberah
payberah@yahoo.com
Contents

- Debugging by Printing
- Debugging by Querying
Debugging by Printing

- The most common debugging technique is monitoring, which in *applications* programming is done by calling `printf` at suitable points.

- When you are debugging *kernel* code, you can accomplish the same goal with `printk`. 
printk

- It works like `printf`.
- One of the differences is that printk lets you classify messages according to their severity by associating different `loglevels`, or priorities, with the messages.
Loglevels

- **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 will often use KERN_ERR to report hardware difficulties.
Loglevels

- **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.
Loglevels

- Each string represents an integer in angle brackets.
- Integers range from 0 to 7, with smaller values representing higher priorities.
Loglevels

- A printk statement with no specified priority defaults to `DEFAULT_MESSAGE_LOGLEVEL`, specified in `kernel/printk.c` as an integer.

- The default loglevel value has changed several times during Linux development, so we suggest that you always specify an explicit loglevel.
Loglevels

- If the priority is less than the integer variable `console_loglevel`, the message is displayed.
- If both `klogd` and `syslogd` are running on the system, kernel messages are appended to `/var/log/messages` independent of `console_loglevel`. 
Kernel loglevel

- It is possible to read and modify the console loglevel using the text file /proc/sys/kernel/printk.
- The file hosts four integer values.
Kernel loglevels

- **console_loglevel**
  - Messages with a higher priority than console_loglevel will be printed to the console.

- **default_message_loglevel**
  - Messages without an explicit priority will be printed with priority default_message_level.

- **minimum_console_level**
  - It is the minimum (highest) value to which console_loglevel can be set.

- **default_console_loglevel**
  - It is the default value for console_loglevel.
Changing loglevel

- klogd
  - `klogd -c <loglevel>`

- echo
  - `echo <loglevel> > /proc/sys/kernel/printk`
Turning the Messages On and Off

```c
#ifdef SCULL_DEBUG
    # define PDEBUG(fmt, args...) printk(KERN_DEBUG "scull: " fmt, ## args)
#endif
```

Turning the Messages On and Off

DEBUG = y

ifeq ($(DEBUG),y)
  DEBFLAGS = -O -g -DSCULL_DEBUG
else
  DEBFLAGS = -O2
Endif

CFLAGS += $(DEBFLAGS)
Contents

- Debugging by Printing
- Debugging by Querying
Debugging by querying

- A massive use of printk can slow down the system noticeably.
  - because syslogd keeps syncing its output files.
  - thus, every line that is printed causes a disk operation.
Debugging by querying

- This problem can be solved by prefixing the name of your log file as it appears in `/etc/syslogd.conf` with a minus.

- Two main techniques are available to driver developers for querying the system:
  - Creating a file in the `/proc` filesystem.
  - Using the `ioctl` driver method.
Using the /proc

- The /proc filesystem is a special, software-created filesystem that is used by the kernel to export information to the world.

- Each file under /proc is tied to a kernel function that generates the file’s “contents” on the fly when the file is read.
Using the /proc

- /proc is heavily used in the Linux system.
- Many utilities on a modern Linux distribution, such as ps, top, and uptime, get their information from /proc.
Create /proc file

- All modules that work with /proc should include `<linux/proc_fs.h>`.
- To create a read-only /proc file, your driver must implement a function to produce the data when the file is read.
Read_proc

- `int (*read_proc)(char *page, char **start, off_t offset, int count, int *eof, void *data);`
  - `page` pointer is the buffer where you’ll write your data.
  - `start` is used by the function to say where the interesting data has been written in `page`.
  - `offset` and `count` have the same meaning as in the read implementation.
  - `eof` argument points to an integer that must be set by the driver to signal that it has no more data to return.
  - `data` is a driver specific data pointer you can use for internal bookkeeping.
Create_proc_read_entry

- int create_proc_read_entry(char *entry_name, int mode, char *proc_dir_entry, int *proc_func, char *client_data);

  - entry_name is the name of the /proc entry.
  - mode is the file permissions to apply to the entry
  - proc_dir_entry is a pointer to the parent directory for this file.
  - proc_func is the pointer to the read_proc function,
  - client_data is data pointer that will be passed back to the read_proc function.
int remove_proc_entry(char *entry_name, char *proc_dir_entry);

- `entry_name` is the name of the /proc entry.
- `parent_dir` is a pointer to the parent directory for this file.
Question?