Linux Device Drivers - debugging

Jernej Vičič

March 15, 2018

Overview

Introduction

- kernel code cannot be executed in a debugger,
- an error can "lay down" the whole system,

Debugging support in the kernel

- compiling proprietary kernel,
  - "debugging features" in the core,
  - these properties are slow (additional writing),
  - is usually turned off in "production" kernels,
  - "kernel hacking" menu,
  - kernel configuration tool.
    - https://en.wikipedia.org/wiki/Menuconfig

Configuration possibilities

- CONFIG_DEBUG_KERNEL: enables for debugging possibilities, does not do anything,
- CONFIG_DEBUG_SLAB: enables to find memory overrun and missing initialization errors, each byte of memory is addressed to $0xa5$ and is set to $0x6b$ after the command *free*,
- CONFIG_DEBUG_PAGEALLOC: enables searching for errors of type "memory corruption", slows down the system considerably,
- CONFIG_DEBUG_SPINLOCK: enables search errors of type: "double open lock",
- CONFIG_DEBUG_SPINLOCK_SLEEP: complains if a function is called that can sleep in critical section,
- CONFIG_INIT_DEBUG: enables code search for parts that use

Configuration possibilities

- CONFIG_DEBUG_INFO: "full debugging information", this option is needed by gdb,
- CONFIG_FRAME_POINTER: also gdb,
- CONFIG_MAGIC_SYSRQ: enables "magic SysRq",
- CONFIG_DEBUG_STACKOVERFLOW, CONFIG_DEBUG_STACK_USAGE: enables search of "kernel stack overflows",
- CONFIG_KALLSYMS: "kernel symbol information" is inserted into kernel that enables a nedugger a "human readable" output,
- CONFIG_IKCONFIG ,CONFIG_IKCONFIG_PROC: usable if the kernel is not compiled by us,

Configuration possibilities

- CONFIG_ACPI_DEBUG: "verbose ACPI debugging information", used with ACPI,
- CONFIG_DEBUG_DRIVER: "debugging information in the driver core",
- CONFIG_SCSI_CONSTANTS: "verbose SCSI error messages",
- CONFIG_INPUT_EVBUG: "verbose logging of input events" for drivers of I/O devices,
- CONFIG_PROFILING: "tracking down some kernel hangs and related problems".

Iskanje napak z izpisi

- *printk*,
- almost *printf*.
- but only almost *printf*.

printk

- ordering messages according to loglevel,
- loglevel is defined by a macro,
- KERN_INFO" already seen,
- loglevel macro expands to a string, that is added to the message,
- The first number in the output is the console log level, the second is the default log level, third is the minimum log level and fourth is the maximum log level.

```
cat /proc/sys/kernel/printk
4 4 1 7
```

printk

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

printk

- KERN_EMERG: this printout is usually followed by a crash,
- KERN_ALERT: requires immediate action,
- KERN_CRIT: "Critical conditions", bad hw, sw errors,
- KERN_ERR: an error message, the drivers should report hw errors this way,
- KERN_WARNING: nothing terrible,
- KERN_NOTICE:normal events, which we still log, security - access,
- KERN_INFO: often on startup,
- KERN_DEBUG: "debugging messages".

printk

- Depending on loglevel, the kernel outputs on:
  - console
  - text-mode terminal
  - serial port
  - parallel printer
- if smaller than console_loglevel:
  - if klogd and syslogd are started, the messages are written to: /var/log/messages,
  - if klogd is not started, the messages will be only in: /proc/kmsg

Print device numbers

```
int print_dev_t(char *buffer, dev_t dev);
char *format_dev_t(char *buffer, dev_t dev);
```

- Print device numbers

Iskanje napak s povpraševanji

- Debugging by Querying,
- frequent use of printk slows down the system,
- some options without using printk:
  - file in /proc file system,
  - use the ioctl driver,
  - export attributes via sysfs.
- last item will not be looked at (we need too much knowledge).

Using /proc file system

- special, programmed file system (not hw),
- used by the kernel,
- each file is bound to a core function that generates the contents of this file when it is read (data is not written),
- examples: /proc/ modules displays all the registered modules,
- some drivers print their content in this way,
- our driver can also work,

Using /proc file system

- /proc should not be used for these purposes,
- sysfs should be used instead (demanding),
- we will use /proc :)

File implementation in /proc

- include <linux/proc_fs.h>
- "read-only", need a read function,
- process starts read system call
- kernel allocates one page of memory at the read call,
- PAGE_SIZE bytes

File implementation in /proc

```
int (*read_proc)(char *page, char **start, off_t offset,
int count, int *eof, void *data);
```

- page - kernel allocates at the command,
- data - (not relevant) - "driver specific, internal usage",
- returns the number of bytes that are placed in the page,
- eof is a simple flag,
- start is used for commands that return more than one data page.

File implementation in /proc

```
int scull_read_procmem(char *buf, char **start, off_t offset,
int count, int *eof, void *data)
{
  int i, j, len = 0;
  int limit = count - 80; /* Don't print more than this */
  for (i = 0; i < scull_nr_devs && len <= limit; i++) {
    struct scull_dev *d = &scull_devices[i];
    struct scull_qset *qs = d->data;
    if (down_interruptible(&d->sem))
      return -ERESTARTSYS;
    len += sprintf(buf+len,"\nDevice %i: qset %i, q %i, sz %li\n",
    i, d->qset, d->quantum, d->size);
    for (; qs && len <= limit; qs = qs->next) { /* scan the list */
      len += sprintf(buf + len, " item at %p, qset at %p\n",
      qs, qs->data);
      if (qs->data && !qs->next) /* dump only the last item */
      for (j = 0; j < d->qset; j++) {
        if (qs->data[j])
          len += sprintf(buf + len,
        " % 4i: %8p\n",
        j, qs->data[j]);
      }
    }
    up(&scull_devices[i].sem);
  }
```

File implementation in /proc

- assume that: only one data page,
- therefore we do not need *start*, *offset*,
- ... but ... we look at the buffer overrun.

File implementation in /proc

- a function was defined: *read_proc*
- function is linked into hierarchy */proc*
- call *create_proc_read_entry*

File implementation in /proc

```
struct proc_dir_entry *create_proc_read_entry(
const char *name, mode_t mode,
struct proc_dir_entry *base,
read_proc_t *read_proc, void *data);
```

- *name* – filename
- *mode* – security mask (0 default)
- *base* – directory in which the file will reside (NULL je root=/proc)
- *read_proc* – function that implements the file, in our case: *scull_read_procmem*
- *data* – kernel ignores this

Example create_proc_read_entry

```
create_proc_read_entry("scullmem", 0 /* default mode */,
NULL /* parent dir */, scull_read_procmem,
NULL /* client data */);
```

- *scullmem* – file name
- default security (world readable)

/proc implementation

```
remove_proc_entry("scullmem", NULL /* parent dir */);
```

- unload module: delete links,
- use: *remove_proc_entry*

Method ioctl

- system call, which behaves like a "file descriptor",
- gets a number that identifies which command is executed,
- we need an additional program,
  - executes the *ioctl* commands,
  - displays the results,
- code in the driver is practically the same.
- More below.

Debugging by observing

- observe the applications behaviour in user space,
- simple/small problems.

Debugging by observing

Use:

1. debugger (for applications),
2. additional printf in the code,
3. *strace*

Debugging by observing – strace

- observe a program from user space;
- shows all system calls,
- system call arguments,
- "return values" of system calls.
- Cmdline options:
  - -*t* – (time) when the call was executed,
  - -*T* – (time spent) time the call used,
  - -*e* – (limit types) filter calls,
  - -*o* – (output) to a file.

Debugging by observing – strace

1. *strace* gets information directly from kernel,
2. no need for "debugging support",
3. can connect to a working process,

Debugging by observing – strace

```
open("/dev", O_RDONLY|O_NONBLOCK|O_LARGEFILE|O_DIRECTORY) = 3
fstat64(3, {st_mode=S_IFDIR|0755, st_size=24576, ...}) = 0
fcntl64(3, F_SETFD, FD_CLOEXEC) = 0
getdents64(3, /* 141 entries */, 4096) = 4088
[...]
getdents64(3, /* 0 entries */, 4096) = 0
close(3) = 0
[...]
fstat64(1, {st_mode=S_IFCHR|0664, st_rdev=makedev(254, 0), ...}) = 0
write(1, "MAKEDEV\nadmmidi0\nadmmidi1\nadmmid"..., 4096) = 4000
write(1, "b\nptywc\nptywd\nptywe\nptywf\nptyx0\n"..., 96) = 96
write(1, "b\nptyxc\nptyxd\nptyxe\nptyxf\nptyy0\n"..., 4096) = 3904
write(1, "s17\nvcs18\nvcs19\nvcs2\nvcs20\nvcs21"..., 192) = 192
write(1, "\nvcs47\nvcs48\nvcs49\nvcs5\nvcs50\nvc"..., 673) = 673
close(1) = 0
exit_group(0) = ?
```

Debugging by observing – strace

- this was started: *ls /dev > /dev/scull0*,
- *ls* after reading the contents of directory starts *write*,
- tries to write only 4000 bytes,
- restarts the call,
- *scull0* writes *quantum* data (4000) in one go.

Debugging by observing – strace

```
[...]
open("/dev/scull0", O_RDONLY|O_LARGEFILE) = 3
fstat64(3, {st_mode=S_IFCHR|0664, st_rdev=makedev(254, 0), ...}) = 0
read(3, "MAKEDEV\nadmmidi0\nadmmidi1\nadmmid"..., 16384) = 4000
read(3, "b\nptywc\nptywd\nptywe\nptywf\nptyx0\n"..., 16384) = 4000
read(3, "s17\nvcs18\nvcs19\nvcs2\nvcs20\nvcs21"..., 16384) = 865
read(3, "", 16384) = 0
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1), ...}) = 0
write(1, "8865 /dev/scull0\n", 17) = 17
close(3) = 0
exit_group(0) = ?
```

Debugging by observing – strace

- started call: *wc -l /dev/scull0*,
- *wc* is used for fast read,
- posktries to read more data: 16384 bytes,
- gets only 4000 bytes,
- restarts the call,
- *scull0* reads *quantum* data (4000) in one go.

System error search

- When we run a driver, and the system error occurs;
- We want to gather as much information as possible;
- "fault" $\neq$ "panic";
- usually panic does not happen, only:
- a process that triggered the driver's code with a bug dies;
- unfreed memory space remains allocated with *kmalloc*,
- the whole system works normally.

System error search

- *oops* : The term does not stand for anything, other than that it is a simple mistake.
- *oops* is a deviation from the normal behaviour of the kernel,
- when the kernel detects a problem, it outputs a *oops* message and kills all the processes that are connected,
- usually also *oops* does not crash the entire system.
- usually, we can only remove the driver from the kernel.
- if the kernel/system behaves strangely: reboot.

System error search, oops

- most bugs: dereferencing *NULL pointer* or using the wrong values of the pointer;
- this causes the *oops* message;
- *oops* displays the state of the processor when the error happened:
  - content of registers,
  - description of the rest of the environment.
- generated using *printk* in the error handler (*fault handler: arch/*/kernel/traps.c*).

System error search, oops primer

```
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
```

System error search, oops primer

- example: *faulty*,
- it has an error,
- code *faulty.c*.

System error search, oops primer

```
ssize_t faulty_write (struct file *filp, const char __user *buf, size_t count,
loff_t *pos)
{
  /* make a simple fault by dereferencing a NULL pointer */
  *(int *)0 = 0;
  return 0;
}
```

- dereference a *NULL pointer*,
- 0 is never a valid poiter address,
- kernel signals oops, process, that called dies.

System error search, oops primer

```
ssize_t faulty_read(struct file *filp, char __user *buf,
size_t count, loff_t *pos)
{
  int ret;
  char stack_buf[4];
  /* Let's try a buffer overflow */
  memset(stack_buf, 0xff, 20);
  if (count > 4)
  count = 4; /* copy 4 bytes to the user */
  ret = copy_to_user(buf, stack_buf, count);
  if (!ret)
  return count;
  return ret;
}
```

System error search, oops primer

- *read* function,
- copies the string into a local variable,
- string is longer than the allocated space,
- *buffer overflow*,
- oops on the return function is displayed,
- *instruction pointer* is set to *nowhereland*.
- Unusable oops:

System error search, oops primer

```
EIP: 0010:[<00000000>]
Unable to handle kernel paging request at virtual address ffffffff
printing eip:
ffffffff
Oops: 0000 [#5]
SMP
CPU: 0
EIP: 0060:[<ffffffff>] Not tainted
EFLAGS: 00010296 (2.6.6)
EIP is at 0xffffffff
eax: 0000000c ebx: ffffffff ecx: 00000000 edx: bfffda7c
esi: cf434f00 edi: ffffffff ebp: 00002000 esp: c27fff78
ds: 007b es: 007b ss: 0068
Process head (pid: 2331, threadinfo=c27fe000 task=c3226150)
Stack: ffffffff bfffda70 00002000 cf434f20 00000001 00000286 cf434f00 ffffffff7
bfffda70 c27fe000 c0150612 cf434f00 bfffda70 00002000 cf434f20 00000000
00000003 00002000 c0103f8f 00000003 bfffda70 00002000 00002000 bfffda70
Call Trace:
[<c0150612>] sys_read+0x42/0x70
[<c0103f8f>] syscall_call+0x7/0xb
Code: Bad EIP value.
```

System error search, oops primer

```
EIP is at faulty_write+0x4/0x10 [faulty]
```

- this line shows the address (first example oops), where the error occurred,
- error happened in function *faulty_write*,
- in module *faulty*,
- *instruction pointer* shows to (0x4)=4 bytes in function, that is (0x10)=16 bytes long.

System error search, when system hangs

- most bugs are presented as *oops*,
- some bugs hang a system,
  - then we do not get *oops*,
    tem code is in an infinite loop,
  - scheduler does not work,
  - system is unresponsive.
- how to get to the description?
- two options:
  - we make sure that the system does not hang,
  - enable to search for errors anyway.

System error search, when system hangs

- How do we get to the description?
- Two options:
  - we make sure that the system does not hang,
  - enable to search for errors anyway.

System error search, when system hangs

- make sure the system does not hang:
  - in strategic places, add a system call *schedule*, which ensures that the task scheduler is called,
  - we can review the process that has been looping,
  - we can kill the process that was looping.
- We can still look for errors:
  - we use this if we do not know where our code hangs,
  - add printouts to the console (also change *console_loglevel*)

Debuggers and similar tools

- last resort)- slow,
- sometimes it is the only way,
- using the debugger at the core is a special problem:
  - difficult to use *breakpoints*,
  - it is difficult to use the running of a single step *single step*,

Format ELF

- Executable and Linkable Format (ELF),
- before: Extensible Linking Format,
- standard file format for:
  - executables,
  - object code,
  - shared libraries,
  - core dumps.

gdb

- debugger is started as if the kernel is just one of the applications,
- give it a kernel image - (ELF kernel image),
- give it the name of the *core* file,
- for the kernel that is currently running is this file in */proc/kcore*.

gdb – usage

```
gdb /usr/src/linux/vmlinux /proc/kcore
```

- first argument: uncompressed ELF image (not: *zImage* ali *bzImage* – boot),
- second argument: name of *core* file, that is */proc* datoteka.

gdb – what does not work

- can not modify kernel data,
- we can not set *breakpoints* or *watchpoints*,
- we can not execute individual steps *single-step*,
- If we want a symbolic table, we need to compile the kernel with the *CONFIG_DEBUG_INFO* tag on,
- can not access the module code.

gdb – new kernels and modules (current)

- from version 2.6.7,
- kernel modules are in ELF format,
- 3 interesting parts:
  - *.text* – executable module code,
  - *.bss* – global module variables, uninitialized at compile time,
  - *.dat* – global module variables, initialized at compile time,,

gdb – new kernels and modules (current)

- present to the debugger where are the sections,
- this is written in */sys/module*,
- for our example in: */sys/module/scull/sections*,
- command that we use: *add-symbol-file*.

gdb – example scull

```
(gdb) add-symbol-file .../scull.ko 0xd0832000 \
-s .bss 0xd0837100 \
-s .data 0xd0836be0
```

- search for the addresses in */sys/module/scull/sections*,
- *0xd0832000* – .text base address,
- *0xd0837100* – .bss base address,
- *0xd0836be0* – .data base address.

gdb – example scull

```
(gdb) add-symbol-file scull.ko 0xd0832000 \
-s .bss 0xd0837100 \
-s .data 0xd0836be0
add symbol table from file "scull.ko" at
.text_addr = 0xd0832000
.bss_addr = 0xd0837100
.data_addr = 0xd0836be0
(y or n) y
Reading symbols from scull.ko...done.
(gdb) p scull_devices[0]
$1 = {data = 0xcfd66c50,
quantum = 4000,
qset = 1000,
size = 20881,
access_key = 0,
...}
```

- device *scull* takes 20,881 bytes,
- *data* contains data ...

gdb – example scull

```
(gdb) print *(address)
```

- displays the code located at *address*,
- helps to locate a function (in memory).

kdb – Kernel Debugger

- *nonofficial patch*,
- `oss.sgi.com`,
- usage:
  - find *patch*, that suits our kernel,
  - use this *patch*,
  - compile the kernel,
  - install new kernel.

kdb

```
Entering kdb (0xc0347b80) on processor 0 due to Keyboard Entry
[0]kdb>
```

- when we have a repaired kernel,
- in the console, press *Pause (or Break) key*,
- the item above appears:

kdb – on module *scull*

```
[0]kdb> bp scull_read
Instruction(i) BP #0 at 0xcd087c5dc (scull_read)
is enabled globally adjust 1
[0]kdb> go
```

- *bp* – kdb stops the kernelat next visit to function *scull_read*,
- *go* – continues the execution,
- start *cat /dev/scull0* which starts the method *scull_read*.

kdb – on module *scull*

```
Instruction(i) breakpoint #0 at 0xd087c5dc (adjusted)
0xd087c5dc scull_read: int3
Entering kdb (current=0xcf09f890, pid 1575) on processor 0 due to
Breakpoint @ 0xd087c5dc
[0]kdb>
[0]kdb> bt
ESP EIP Function (args)
0xcdbddf74 0xd087c5dc [scull]scull_read
0xcdbddf78 0xc0150718 vfs_read+0xb8
0xcdbddfa4 0xc01509c2 sys_read+0x42
0xcdbddfc4 0xc0103fcf syscall_call+0x7
[0]kdb> mds scull_devices 1
0xd0880de8 cf36ac00 ....
```

- *bt* – stack trace
- *mds* – manipulate data, this example shows the value of the pointer *scull_devices*,

kgdb – kgdb Patches

- enables most of the useful functionality of normal debuggers:
  - changing of variables,
  - breakpoints, ...
- `http://kgdb.linsyssoft.com/`,
- just info ...

User-Mode Linux Port

- User-Mode Linux Port (UML),
- works on the basis of a virtual machine,
- does not talk directly with hardware,
- allows you to run the kernel in the user space,
- why you should use (pros):
  - gdb can be used,
  - different hardware,
  - if crashed just an "application" died.

Linux Trace Toolkit

- Linux Trace Toolkit (LTT),
- kernel patch +,
- a collection of tools,
- `http://www.opersys.com/LTT`.

Dynamic Probes

- Dynamic Probes (DProbes),
- debugging tool,
- allows the probe to be mounted,
- code in a special language,
- can send information to the user space.