Kernel modules

Lab objectives

- creating simple modules
- describing the process of kernel module compilation
- presenting how a module can be used with a kernel
- simple kernel debugging methods

Kernel Modules Overview

A monolithic kernel, though faster than a microkernel, has the disadvantage of lack of modularity and extensibility. On modern monolithic kernels, this has been solved by using kernel modules. A kernel module (or loadable kernel mode) is an object file that contains code that can extend the kernel functionality at runtime (it is loaded as needed); When a kernel module is no longer needed, it can be unloaded. Most of the device drivers are used in the form of kernel modules.

For the development of Linux device drivers, it is recommended to download the kernel sources, configure and compile them and then install the compiled version on the test / development tool machine.

An example of a kernel module

Below is a very simple example of a kernel module. When loading into the kernel, it will generate the message `"Hi"`. When unloading the kernel module, the `"Bye"` message will be generated.

```
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>

MODULE_DESCRIPTION("My kernel module");
MODULE_AUTHOR("Me");
MODULE_LICENSE("GPL");

static int dummy_init(void)
{
        pr_debug("Hi\n");
        return 0;
}

static void dummy_exit(void)
{
        pr_debug("Bye\n");
}

module_init(dummy_init);
module_exit(dummy_exit);
```

The generated messages will not be displayed on the console but will be saved in a specially reserved memory area for this, from where they will be extracted by the logging

daemon (syslog). To display kernel messages, you can use the **dmesg** command or inspect the logs:

```
# cat /var/log/syslog | tail -2
Feb 20 13:57:38 asgard kernel: Hi
Feb 20 13:57:43 asgard kernel: Bye

# dmesg | tail -2
Hi
Bye
```

Compiling kernel modules

Compiling a kernel module differs from compiling an user program. First, other headers should be used. Also, the module should not be linked to libraries. And, last but not least, the module must be compiled with the same options as the kernel in which we load the module. For these reasons, there is a standard compilation method (`kbuild`). This method requires the use of two files: a `Makefile` and a `Kbuild` file.

Below is an example of a `Makefile`:

```
KDIR = /lib/modules/`uname -r`/build

kbuild:
	make -C $(KDIR) M=`pwd`

clean:
	make -C $(KDIR) M=`pwd` clean
```

And the example of a `Kbuild` file used to compile a module:

```
EXTRA_CFLAGS = -Wall -g

obj-m        = modul.o
```

As you can see, calling **make** on the `Makefile` file in the example shown will result in the **make** invocation in the kernel source directory (`/lib/modules/`uname -r`/build`) and referring to the current directory (`M = `pwd``). This process ultimately leads to reading the `Kbuild` file from the current directory and compiling the module as instructed in this file.

Note

For labs we will configure different **KDIR**, according to the virtual machine specifications:

```
KDIR = /home/student/src/linux
[...]
```

A `Kbuild` file contains one or more directives for compiling a kernel module. The easiest

example of such a directive is `obj-m = module.o`. Following this directive, a kernel module (`ko` - kernel object) will be created, starting from the `module.o` file. `module.o` will be created starting from `module.c` or `module.S`. All of these files can be found in the `Kbuild`'s directory.

An example of a `Kbuild` file that uses several sub-modules is shown below:

```
EXTRA_CFLAGS = -Wall -g

obj-m        = supermodule.o
supermodule-y = module-a.o module-b.o
```

For the example above, the steps to compile are:

- compile the `module-a.c` and `module-b.c` sources, resulting in module-a.o and module-b.o objects
- `module-a.o` and `module-b.o` will then be linked in `supermodule.o`
- from `supermodule.o` will be created `supermodule.ko` module

The suffix of targets in `Kbuild` determines how they are used, as follows:

- M (modules) is a target for loadable kernel modules
- Y (yes) represents a target for object files to be compiled and then linked to a module (`$(mode_name)-y`) or within the kernel (`obj-y`)
- any other target suffix will be ignored by `Kbuild` and will not be compiled

Note

These suffixes are used to easily configure the kernel by running the **make menuconfig** command or directly editing the `.config` file. This file sets a series of variables that are used to determine which features are added to the kernel at build time. For example, when adding BTRFS support with **make menuconfig**, add the line `CONFIG_BTRFS_FS = y` to the `.config` file. The BTRFS kbuild contains the line `obj-$(CONFIG_BTRFS_FS):= btrfs.o`, which becomes `obj-y:= btrfs.o`. This will compile the `btrfs.o` object and will be linked to the kernel. Before the variable was set, the line became `obj:=btrfs.o` and so it was ignored, and the kernel was build without BTRFS support.

For more details, see the `Documentation/kbuild/makefiles.txt` and `Documentation/kbuild/modules.txt` files within the kernel sources.

Loading/unloading a kernel module

To load a kernel module, use the **insmod** utility. This utility receives as a parameter the path to the `*.ko` file in which the module was compiled and linked. Unloading the module from the kernel is done using the **rmmod** command, which receives the module name as a parameter.

```
$ insmod module.ko
$ rmmod module.ko
```

When loading the kernel module, the routine specified as a parameter of the `module_init` macro will be executed. Similarly, when the module is unloaded the routine specified as a parameter of the `module_exit` will be executed.

A complete example of compiling and loading/unloading a kernel module is presented below:

```
faust:~/lab-01/modul-lin# ls
Kbuild  Makefile  modul.c

faust:~/lab-01/modul-lin# make
make -C /lib/modules/`uname -r`/build M=`pwd`
make[1]: Entering directory `/usr/src/linux-2.6.28.4'
  LD      /root/lab-01/modul-lin/built-in.o
  CC [M]  /root/lab-01/modul-lin/modul.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /root/lab-01/modul-lin/modul.mod.o
  LD [M]  /root/lab-01/modul-lin/modul.ko
make[1]: Leaving directory `/usr/src/linux-2.6.28.4'

faust:~/lab-01/modul-lin# ls
built-in.o  Kbuild  Makefile  modul.c  Module.markers
modules.order  Module.symvers  modul.ko  modul.mod.c
modul.mod.o  modul.o

faust:~/lab-01/modul-lin# insmod modul.ko

faust:~/lab-01/modul-lin# dmesg | tail -1
Hi

faust:~/lab-01/modul-lin# rmmod modul

faust:~/lab-01/modul-lin# dmesg | tail -2
Hi
Bye
```

Information about modules loaded into the kernel can be found using the **lsmod** command or by inspecting the `/proc/modules`, `/sys/module` directories.

Kernel Module Debugging

Troubleshooting a kernel module is much more complicated than debugging a regular program. First, a mistake in a kernel module can lead to blocking the entire system. Troubleshooting is therefore much slowed down. To avoid reboot, it is recommended to use a virtual machine (qemu, virtualbox, vmware).

When a module containing bugs is inserted into the kernel, it will eventually generate a kernel oops. A kernel oops is an invalid operation detected by the kernel and can only be generated by the kernel. For a stable kernel version, it almost certainly means that the module contains a bug. After the oops appears, the kernel will continue to work.

Very important to the appearance of a kernel oops is saving the generated message. As noted above, messages generated by the kernel are saved in logs and can be displayed with the **dmesg** command. To make sure that no kernel message is lost, it is recommended to insert/test the kernel directly from the console, or periodically check the kernel messages. Noteworthy is that an oops can occur because of a programming error, but also a because of hardware error.

If a fatal error occurs, after which the system can not return to a stable state, a kernel panic is generated.

Look at the kernel module below that contains a bug that generates an oops:

```
/*
 * Oops generating kernel module
 */

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>

MODULE_DESCRIPTION ("Oops");
MODULE_LICENSE ("GPL");
MODULE_AUTHOR ("PSO");

#define OP_READ         0
#define OP_WRITE        1
#define OP_OOPS         OP_WRITE

static int my_oops_init (void)
{
        int *a;

        a = (int *) 0x00001234;
#if OP_OOPS == OP_WRITE
        *a = 3;
#elif OP_OOPS == OP_READ
        printk (KERN_ALERT "value = %d\n", *a);
#else
#error "Unknown op for oops!"
#endif

        return 0;
}

static void my_oops_exit (void)
{
}

module_init (my_oops_init);
module_exit (my_oops_exit);
```

Inserting this module into the kernel will generate an oops:

```
faust:~/lab-01/modul-oops# insmod oops.ko
[...]

faust:~/lab-01/modul-oops# dmesg | tail -32
BUG: unable to handle kernel paging request at 00001234
IP: [<c89d4005>] my_oops_init+0x5/0x20 [oops]
  *de = 00000000
Oops: 0002 [#1] PREEMPT DEBUG_PAGEALLOC
last sysfs file: /sys/devices/virtual/net/lo/operstate
Modules linked in: oops(+) netconsole ide_cd_mod pcnet32 crc32 cdrom [last unloaded: modul]

Pid: 4157, comm: insmod Not tainted (2.6.28.4 #2) VMware Virtual Platform
EIP: 0060:[<c89d4005>] EFLAGS: 00010246 CPU: 0
EIP is at my_oops_init+0x5/0x20 [oops]
EAX: 00000000 EBX: fffffffc ECX: c89d4300 EDX: 00000001
ESI: c89d4000 EDI: 00000000 EBP: c5799e24 ESP: c5799e24
 DS: 007b ES: 007b FS: 0000 GS: 0033 SS: 0068
Process insmod (pid: 4157, ti=c5799000 task=c665c780 task.ti=c5799000)
Stack:
 c5799f8c c010102d c72b51d8 0000000c c5799e58 c01708e4 00000124 00000000
 c89d4300 c5799e58 c724f448 00000001 c89d4300 c5799e60 c0170981 c5799f8c
 c014b698 00000000 00000000 c5799f78 c5799f20 00000500 c665cb00 c89d4300
Call Trace:
 [<c010102d>] ? _stext+0x2d/0x170
 [<c01708e4>] ? __vunmap+0xa4/0xf0
 [<c0170981>] ? vfree+0x21/0x30
 [<c014b698>] ? load_module+0x19b8/0x1a40
 [<c035e965>] ? __mutex_unlock_slowpath+0xd5/0x140
 [<c0140da6>] ? trace_hardirqs_on_caller+0x106/0x150
 [<c014b7aa>] ? sys_init_module+0x8a/0x1b0
 [<c0140da6>] ? trace_hardirqs_on_caller+0x106/0x150
 [<c0240a08>] ? trace_hardirqs_on_thunk+0xc/0x10
 [<c0103407>] ? sysenter_do_call+0x12/0x43
Code: <c7> 05 34 12 00 00 03 00 00 00 5d c3 eb 0d 90 90 90 90 90 90 90 90
EIP: [<c89d4005>] my_oops_init+0x5/0x20 [oops] SS:ESP 0068:c5799e24
---[ end trace 2981ce73ae801363 ]---
```

Although relatively cryptic, the message provided by the kernel to the appearance of an oops provides valuable information about the error. First line:

```
BUG: unable to handle kernel paging request at 00001234
EIP: [<c89d4005>] my_oops_init + 0x5 / 0x20 [oops]
```

Tells us the cause and the address of the instruction that generated the error. In our case this is an invalid access to memory.

Next line

```
Oops: 0002 [# 1] PREEMPT DEBUG_PAGEALLOC
```

Tells us that it's the first oops (#1). This is important in the context that an oops can lead to other oopses. Usually only the first oops is relevant. Furthermore, the oops code (`0002`) provides information about the error type (see `arch/x86/include/asm/trap_pf.h`):

- Bit 0 == 0 means no page found, 1 means protection fault
- Bit 1 == 0 means read, 1 means write
- Bit 2 == 0 means kernel, 1 means user mode

In this case, we have a write access that generated the oops (bit 1 is 1).

Below is a dump of the registers. It decodes the instruction pointer (`EIP`) value and notes that the bug appeared in the `my_oops_init` function with a 5-byte offset (`EIP: [<c89d4005>] my_oops_init+0x5`). The message also shows the stack content and a backtrace of calls until then.

If an invalid read call is generated (`#define OP_OOPS OP_READ`), the message will be the same, but the oops code will differ, which would now be `0000`:

```
faust:~/lab-01/modul-oops# dmesg | tail -33
BUG: unable to handle kernel paging request at 00001234
IP: [<c89c3016>] my_oops_init+0x6/0x20 [oops]
  *de = 00000000
Oops: 0000 [#1] PREEMPT DEBUG_PAGEALLOC
last sysfs file: /sys/devices/virtual/net/lo/operstate
Modules linked in: oops(+) netconsole pcnet32 crc32 ide_cd_mod cdrom

Pid: 2754, comm: insmod Not tainted (2.6.28.4 #2) VMware Virtual Platform
EIP: 0060:[<c89c3016>] EFLAGS: 00010292 CPU: 0
EIP is at my_oops_init+0x6/0x20 [oops]
EAX: 00000000 EBX: fffffffc ECX: c89c3380 EDX: 00000001
ESI: c89c3010 EDI: 00000000 EBP: c57cbe24 ESP: c57cbe1c
 DS: 007b ES: 007b FS: 0000 GS: 0033 SS: 0068
Process insmod (pid: 2754, ti=c57cb000 task=c66ec780 task.ti=c57cb000)
Stack:
 c57cbe34 00000282 c57cbf8c c010102d c57b9280 0000000c c57cbe58 c01708e4
 00000124 00000000 c89c3380 c57cbe58 c5db1d38 00000001 c89c3380 c57cbe60
 c0170981 c57cbf8c c014b698 00000000 00000000 c57cbf78 c57cbf20 00000580
Call Trace:
 [<c010102d>] ? _stext+0x2d/0x170
 [<c01708e4>] ? __vunmap+0xa4/0xf0
 [<c0170981>] ? vfree+0x21/0x30
 [<c014b698>] ? load_module+0x19b8/0x1a40
 [<c035d083>] ? printk+0x0/0x1a
 [<c035e965>] ? __mutex_unlock_slowpath+0xd5/0x140
 [<c0140da6>] ? trace_hardirqs_on_caller+0x106/0x150
 [<c014b7aa>] ? sys_init_module+0x8a/0x1b0
 [<c0140da6>] ? trace_hardirqs_on_caller+0x106/0x150
 [<c0240a08>] ? trace_hardirqs_on_thunk+0xc/0x10
 [<c0103407>] ? sysenter_do_call+0x12/0x43
Code: <a1> 34 12 00 00 c7 04 24 54 30 9c c8 89 44 24 04 e8 58 a0 99 f7 31
EIP: [<c89c3016>] my_oops_init+0x6/0x20 [oops] SS:ESP 0068:c57cbe1c
---[ end trace 45eeb3d6ea8ff1ed ]---
```

objdump

Detailed information about the instruction that generated the oops can be found using the **objdump** utility. Useful options to use are **-d** to disassemble the code and **-S** for interleaving C code in assembly language code. For efficient decoding, however, we need the address where the kernel module was loaded. This can be found in `/proc/modules`.

Here's an example of using **objdump** on the above module to identify the instruction that generated the oops:

```
faust:~/lab-01/modul-oops# cat /proc/modules
oops 1280 1 - Loading 0xc89d4000
netconsole 8352 0 - Live 0xc89ad000
pcnet32 33412 0 - Live 0xc895a000
ide_cd_mod 34952 0 - Live 0xc8903000
crc32 4224 1 pcnet32, Live 0xc888a000
cdrom 34848 1 ide_cd_mod, Live 0xc886d000

faust:~/lab-01/modul-oops# objdump -dS --adjust-vma=0xc89d4000 oops.ko

oops.ko:     file format elf32-i386


Disassembly of section .text:

c89d4000 <init_module>:
#define OP_READ         0
#define OP_WRITE        1
#define OP_OOPS         OP_WRITE

static int my_oops_init (void)
{
c89d4000:       55                      push   %ebp
#else
#error "Unknown op for oops!"
#endif

        return 0;
}
c89d4001:       31 c0                   xor    %eax,%eax
#define OP_READ         0
#define OP_WRITE        1
#define OP_OOPS         OP_WRITE

static int my_oops_init (void)
{
c89d4003:       89 e5                   mov    %esp,%ebp
        int *a;

        a = (int *) 0x00001234;
#if OP_OOPS == OP_WRITE
        *a = 3;
c89d4005:       c7 05 34 12 00 00 03    movl   $0x3,0x1234
c89d400c:       00 00 00
#else
#error "Unknown op for oops!"
#endif

        return 0;
}
c89d400f:       5d                      pop    %ebp
c89d4010:       c3                      ret
c89d4011:       eb 0d                   jmp    c89c3020 <cleanup_module>
c89d4013:       90                      nop
c89d4014:       90                      nop
c89d4015:       90                      nop
c89d4016:       90                      nop
c89d4017:       90                      nop
c89d4018:       90                      nop
c89d4019:       90                      nop
c89d401a:       90                      nop
c89d401b:       90                      nop
c89d401c:       90                      nop
c89d401d:       90                      nop
c89d401e:       90                      nop
c89d401f:       90                      nop

c89d4020 <cleanup_module>:

static void my_oops_exit (void)
{
c89d4020:       55                      push   %ebp
c89d4021:       89 e5                   mov    %esp,%ebp
}
c89d4023:       5d                      pop    %ebp
c89d4024:       c3                      ret
```

```
c89d4025:       90                      nop
c89d4026:       90                      nop
c89d4027:       90                      nop
```

Note that the instruction that generated the oops (`c89d4005` identified earlier) is:

```
C89d4005: c7 05 34 12 00 00 03 movl $ 0x3,0x1234
```

That is exactly what was expected - storing value 3 at 0x0001234.

The `/proc/modules` is used to find the address where a kernel module is loaded. The **--adjust-vma** option allows you to display instructions relative to `0xc89d4000`. The **-l** option displays the number of each line in the source code interleaved with the assembly language code.

addr2line

A more simplistic way to find the code that generated an oops is to use the **addr2line** utility:

```
faust:~/lab-01/modul-oops# addr2line -e oops.o 0x5
/root/lab-01/modul-oops/oops.c:23
```

Where `0x5` is the value of the program counter (`EIP = c89d4005`) that generated the oops, minus the base address of the module (`0xc89d4000`) according to `/proc/modules`

minicom

Minicom (or other equivalent utilities, eg **picocom**, **screen**) is a utility that can be used to connect and interact with a serial port. The serial port is the basic method for analyzing kernel messages or interacting with an embedded system in the development phase. There are two more common ways to connect:

- a serial port where the device we are going to use is `/dev/ttyS0`
- a serial USB port (FTDI) in which case the device we are going to use is `/dev/ttyUSB`.

For the virtual machine used in the lab, the device that we need to use is displayed after the virtual machine starts:

```
char device redirected to /dev/pts/20 (label virtiocon0)
```

Minicom use:

```
#for connecting via COM1 and using a speed of 115,200 characters per second
minicom -b 115200 -D /dev/ttyS0

#For USB serial port connection
minicom -D /dev/ttyUSB0

#To connect to the serial port of the virtual machine
minicom -D /dev/pts/20
```

netconsole

Netconsole is a utility that allows logging of kernel debugging messages over the network. This is useful when the disk logging system does not work or when serial ports are not available or when the terminal does not respond to commands. **Netconsole** comes in the form of a kernel module.

To work, it needs the following parameters:

- port, IP address, and the source interface name of the debug station
- port, MAC address, and IP address of the machine to which the debug messages will be sent

These parameters can be configured when the module is inserted into the kernel, or even while the module is inserted if it has been compiled with the `CONFIG_NETCONSOLE_DYNAMIC` option.

An example configuration when inserting **netconsole** kernel module is as follows:

```
alice:~# modprobe netconsole netconsole=6666@192.168.191.130/
eth0,6000@192.168.191.1/00:50:56:c0:00:08
```

Thus, the debug messages on the station that has the address `192.168.191.130` will be sent to the `eth0` interface, having source port `6666`. The messages will be sent to `192.168.191.1` with the MAC address `00:50:56:c0:00:08`, on port `6000`.

Messages can be played on the destination station using **netcat**:

```
bob:~ # nc -l -p 6000 -u
```

Alternatively, the destination station can configure **syslogd** to intercept these messages. More information can be found in `Documentation/networking/netconsole.txt`.

Printk debugging

`The two oldest and most useful debugging aids are Your Brain and Printf`.

For debugging, a primitive way is often used, but it is quite effective: `printk` debugging. Although a debugger can also be used, it is generally not very useful: simple bugs (uninitialized variables, memory management problems, etc.) can be easily localized by control messages and the kernel-decoded oop message.

For more complex bugs, even a debugger can not help us too much unless the operating system structure is very well understood. When debugging a kernel module, there are a lot of unknowns in the equation: multiple contexts (we have multiple processes and threads running at a time), interruptions, virtual memory, etc.

You can use `printk` to display kernel messages to user space. It is similar to `printf`'s functionality; the only difference is that the transmitted message can be prefixed with a string of `"<n>"`, where `n` indicates the error level (loglevel) and has values between `0` and `7`. Instead of `"<n>"`, the levels can also be coded by symbolic constants:

```
KERN_EMERG - n = 0
KERN_ALERT - n = 1
KERN_CRIT - n = 2
KERN_ERR - n = 3
KERN_WARNING - n = 4
KERN_NOTICE - n = 5
KERN_INFO - n = 6
KERN_DEBUG - n = 7
```

The definitions of all log levels are found in `linux/kern_levels.h`. Basically, these log levels are used by the system to route messages sent to various outputs: console, log files in `/var/log` etc.

Note

To display `printk` messages in user space, the `printk` log level must be of higher priority than *console_loglevel* variable. The default console log level can be configured from `/proc/sys/kernel/printk`.

For instance, the command:

```
echo 8 > /proc/sys/kernel/printk
```

will enable all the kernel log messages to be displayed in the console. That is, the logging level has to be strictly less than the `console_loglevel` variable. For example, if the `console_loglevel` has a value of `5` (specific to `KERN_NOTICE`), only messages with loglevel stricter than `5` (i.e `KERN_EMERG`, `KERN_ALERT`, `KERN_CRIT`, `KERN_ERR`, `KERN_WARNING`) will be shown.

Console-redirected messages can be useful for quickly viewing the effect of executing the kernel code, but they are no longer so useful if the kernel encounters an irreparable error and the system freezes. In this case, the logs of the system must be consulted, as they keep the information between system restarts. These are found in `/var/log` and are text files, populated by `syslogd` and `klogd` during the kernel run. `syslogd` and `klogd` take the information from the virtual file system mounted in `/proc`. In principle, with `syslogd` and `klogd` turned on, all messages coming from the kernel will go to `/var/log/kern.log`.

A simpler version for debugging is using the `/var/log/debug` file. It is populated only with the `printk` messages from the kernel with the `KERN_DEBUG` log level.

Given that a production kernel (similar to the one we're probably running with) contains only release code, our module is among the few that send messages prefixed with KERN_DEBUG . In this way, we can easily navigate through the `/var/log/debug` information by finding the messages corresponding to a debugging session for our module.

Such an example would be the following:

```
# Clear the debug file of previous information (or possibly a backup)
$ echo "New debug session" > /var/log/debug
# Run the tests
# If there is no critical error causing a panic kernel, check the output
# if a critical error occurs and the machine only responds to a restart,
  restart the system and check /var/log/debug.
```

The format of the messages must obviously contain all the information of interest in order to detect the error, but inserting in the code `printk` to provide detailed information can be as time-consuming as writing the code to solve the problem. This is usually a trade-off between the completeness of the debugging messages displayed using `printk` and the time it takes to insert these messages into the text.

A very simple way, less time-consuming for inserting `printk` and providing the possibility to analyze the flow of instructions for tests is the use of the predefined constants `__FILE__`, `__LINE__` and `__func__`:

- `__FILE__` is replaced by the compiler with the name of the source file it is currently being compiled.
- `__LINE__` is replaced by the compiler with the line number on which the current instruction is found in the current source file.
- `__func__` / `__FUNCTION__` is replaced by the compiler with the name of the function in which the current instruction is found.

Note

`__FILE__` and `__LINE__` are part of the ANSI C specifications: `__func__` is part of specification C99; `__FUNCTION__` is a GNU `C` extension and is not portable; However, since we write code for the `Linux` kernel, we can use it without any problems.

The following macro definition can be used in this case:

```
#define PRINT_DEBUG \
       printk (KERN_DEBUG "[% s]: FUNC:% s: LINE:% d \ n", __FILE__,
               __FUNCTION__, __LINE__)
```

Then, at each point where we want to see if it is "reached" in execution, insert PRINT_DEBUG; This is a simple and quick way, and can yield by carefully analyzing the output.

The **dmesg** command is used to view the messages printed with `printk` but not appearing on the console.

To delete all previous messages from a log file, run:

```
cat /dev/null > /var/log/debug
```

To delete messages displayed by the **dmesg** command, run:

```
dmesg -c
```

Dynamic debugging

Dynamic dyndbg debugging enables dynamic debugging activation/deactivation. Unlike `printk`, it offers more advanced `printk` options for the messages we want to display; it is very useful for complex modules or troubleshooting subsystems. This significantly reduces the amount of messages displayed, leaving only those relevant for the debug context. To enable `dyndbg`, the kernel must be compiled with the `CONFIG_DYNAMIC_DEBUG` option. Once configured, `pr_debug()`, `dev_dbg()` and `print_hex_dump_debug()`, `print_hex_dump_bytes()` can be dynamically enabled per call.

The `/sys/kernel/debug/dynamic_debug/control` file from the debugfs (where `/sys/kernel/debug` is the path to which debugfs was mounted) is used to filter messages or to view existing filters.

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

Debugfs is a simple file system, used as a kernel-space interface and user-space interface to configure different debug options. Any debug utility can create and use its own files / folders in debugfs.

For example, to display existing filters in `dyndbg`, you will use:

```
cat /debug/dynamic_debug/control
```

And to enable the debug message from line `1603` in the `svcsock.c` file:

```
echo 'file svcsock.c line 1603 +p' > /debug/dynamic_debug/control
```

The `/debug/dynamic_debug/control` file is not a regular file. It shows the `dyndbg` settings on the filters. Writing in it with an echo will change these settings (it will not actually make a write). Be aware that the file contains settings for `dyndbg` debugging messages. Do not log in this file.

Dyndbg Options

- `func` - just the debug messages from the functions that have the same name as the one

defined in the filter.

```
echo 'func svc_tcp_accept +p' > /debug/dynamic_debug/control
```

- `file` - the name of the file(s) for which we want to display the debug messages. It can be just the source name, but also the absolute path or kernel-tree path.

```
file svcsock.c
file kernel/freezer.c
file /usr/src/packages/BUILD/sgi-enhancednfs-1.4/default/net/sunrpc/svcsock.c
```

- `module` - module name.

```
module sunrpc
```

- `format` - only messages whose display format contains the specified string.

```
format "nfsd: SETATTR"
```

- `line` - the line or lines for which we want to enable debug calls.

```
# Triggers debug messages between lines 1603 and 1605 in the svcsock.c file
$ echo 'file svcsock.c line 1603-1605 +p' > /sys/kernel/debug/dynamic_debug/control
# Enables debug messages from the beginning of the file to line 1605
$ echo 'file svcsock.c line -1605 +p' > /sys/kernel/debug/dynamic_debug/control
```

In addition to the above options, a series of flags can be added, removed, or set with operators `+`, `-` or `=`:

- `p` activates the pr_debug() .
- `f` includes the name of the function in the printed message.
- `l` includes the line number in the printed message.
- `m` includes the module name in the printed message.
- `t` includes the thread id if it is not called from interrupt context
- `_` no flag is set.

KDB: Kernel debugger

The kernel debugger has proven to be very useful to facilitate the development and debugging process. One of its main advantages is the possibility to perform live debugging. This allows us to monitor, in real time, the accesses to memory or even modify the memory while debugging. The debugger has been integrated in the mainline kernel starting with version 2.6.26-rci. KDB is not a *source debugger*, but for a complete analysis it can be used in parallel with gdb and symbol files -- see the GDB debugging section

To use KDB, you have the following options:

- non-usb keyboard + VGA text console
- serial port console
- USB EHCI debug port

For the lab, we will use a serial interface connected to the host. The following command will activate GDB over the serial port:

```
echo hvc0 > /sys/module/kgdboc/parameters/kgdboc
```

KDB is a *stop mode debugger*, which means that, while it is active, all the other processes are stopped. The kernel can be *forced* to enter KDB during execution using the following SysRq command

```
echo g > /proc/sysrq-trigger
```

or by using the key combination `Ctrl+O g` in a terminal connected to the serial port (for example using **minicom**).

KDB has various commands to control and define the context of the debugged system:

- lsmod, ps, kill, dmesg, env, bt (backtrace)
- dump trace logs
- hardware breakpoints
- modifying memory

For a better description of the available commands you can use the `help` command in the KDB shell. In the next example, you can notice a simple KDB usage example which sets a hardware breakpoint to monitor the changes of the `mVar` variable.

```
# trigger KDB
echo g > /proc/sysrq-trigger
# or if we are connected to the serial port issue
Ctrl-O g
# breakpoint on write access to the mVar variable
kdb> bph mVar dataw
# return from KDB
kdb> go
```

Exercises

Important

We strongly encourage you to use the setup from this repository.

To solve exercises, you need to perform these steps:

- prepare skeletons from templates
- build modules
- start the VM and test the module in the VM.

The current lab name is kernel_modules. See the exercises for the task name.

The skeleton code is generated from full source examples located in `tools/labs/templates`. To solve the tasks, start by generating the skeleton code for a complete lab:

```
tools/labs $ make clean
tools/labs $ LABS=<lab name> make skels
```

You can also generate the skeleton for a single task, using

```
tools/labs $ LABS=<lab name>/<task name> make skels
```

Once the skeleton drivers are generated, build the source:

```
tools/labs $ make build
```

Then, start the VM:

```
tools/labs $ make console
```

The modules are placed in /home/root/skels/kernel_modules/<task_name>.

You DO NOT need to STOP the VM when rebuilding modules! The local *skels* directory is shared with the VM.

Review the Exercises section for more detailed information.

Warning

Before starting the exercises or generating the skeletons, please run **git pull** inside the Linux repo, to make sure you have the latest version of the exercises.

If you have local changes, the pull command will fail. Check for local changes using `git status`. If you want to keep them, run `git stash` before `pull` and `git stash pop` after. To discard the changes, run `git reset --hard master`.

If you already generated the skeleton before `git pull` you will need to generate it again.

0. Intro

Using **cscope** or LXR find the definitions of the following symbols in the Linux kernel source code:

- `module_init()` and `module_exit()`
  - what do the two macros do? What is `init_module` and `cleanup_module`?

- `ignore_loglevel`
  - What is this variable used for?

Warning

If you have problems using **cscope**, it is possible that the database is not generated. To generate it, use the following command in the kernel directory:

```
make ARCH=x86 cscope
```

Note

When searching for a structure using **cscope**, use only the structure name (without `struct`). So, to search for the structure `struct module`, you will use the command

```
vim -t module
```

or, in **vim**, the command

```
:cs f g module
```

Note

For more info on using **cscope**, read the cscope section in the previous lab.

1. Kernel module

To work with the kernel modules, we will follow the steps described above.

Generate the skeleton for the task named 1-2-test-mod then build the module,

by running the following command in `tools/labs`.

```
$ LABS=kernel_modules make skels
$ make build
```

These command will build all the modules in the current lab skeleton.

Warning

Until after solving exercise 3, you will get a compilation error for `3-error-mod`. To avoid this issue, remove the directory `skels/kernel_modules/3-error-mod/` and remove the corresponding line from `skels/Kbuild`.

Start the VM using **make console**, and perform the following tasks:

- load the kernel module.
- list the kernel modules and check if current module is present
- unload the kernel module
- view the messages displayed at loading/unloading the kernel module using **dmesg** command

Note

Read Loading/unloading a kernel module section. When unloading a kernel module, you can specify only the module name (without extension).

2. Printk

Watch the virtual machine console. Why were the messages displayed directly to the virtual machine console?

Configure the system such that the messages are not displayed directly on the serial console, and they can only be inspected using `dmesg`.

Hint

One option is to set the console log level by writting the desired level to `/proc/sys/kernel/printk`. Use a value smaller than the level used for the prints in the source code of the module.

Load/unload the module again. The messages should not be printed to the virtual machine console, but they should be visible when running `dmesg`.

3. Error

Generate the skeleton for the task named **3-error-mod**. Compile the sources and get the corresponding kernel module.

Why have compilation errors occurred? **Hint:** How does this module differ from the previous module?

Modify the module to solve the cause of those errors, then compile and test the module.

4. Sub-modules

Inspect the C source files `mod1.c` and `mod2.c` in `4-multi-mod/`. Module 2 contains only the definition of a function used by module 1.

Change the `Kbuild` file to create the `multi_mod.ko` module from the two C source files.

Hint

Read the Compiling kernel modules section of the lab.

Compile, copy, boot the VM, load and unload the kernel module. Make sure messages are properly displayed on the console.

5. Kernel oops

Enter the directory for the task **5-oops-mod** and inspect the C source file. Notice where the problem will occur. Add the compilation flag `-g` in the Kbuild file.

Hint

Read Compiling kernel modules section of the lab.

Compile the corresponding module and load it into the kernel. Identify the memory address at which the oops appeared.

Hint

Read `Debugging`_ section of the lab. To identify the address, follow the oops message and extract the value of the instructions pointer (`EIP`) register.

Determine which instruction has triggered the oops.

Hint

Use the `proc/modules` information to get the load address of the kernel module. Use, on the physical machine, objdump and/or addr2line . Objdump needs debugging support for compilation! Read the lab's objdump and addr2line sections.

Try to unload the kernel module. Notice that the operation does not work because there are references from the kernel module within the kernel since the oops; Until the release of those references (which is almost impossible in the case of an oops), the module can not be unloaded.

6. Module parameters

Enter the directory for the task **6-cmd-mod** and inspect the C `cmd_mod.c` source file. Compile and copy the associated module and load the kernel module to see the printk message. Then unload the module from the kernel.

Without modifying the sources, load the kernel module so that the message shown is `Early bird gets tired` .

Hint

The str variable can be changed by passing a parameter to the module. Find more information here.

7. Proc info

Check the skeleton for the task named **7-list-proc**. Add code to display the Process ID (`PID`) and the executable name for the current process.

Follow the commands marked with `TODO` . The information must be displayed both when loading and unloading the module.

Note

- In the Linux kernel, a process is described by the `struct task_struct`. Use LXR or `cscope` to find the definition of `struct task_struct`.
- To find the structure field that contains the name of the executable, look for the "executable" comment.
- The pointer to the structure of the current process running at a given time in the kernel is given by the `current` variable (of the type `struct task_struct*`).

Hint

To use `current` you'll need to include the header in which the `struct task_struct` is defined, i.e `linux/sched.h`.

Compile, copy, boot the VM and load the module. Unload the kernel module.

Repeat the loading/unloading operation. Note that the PIDs of the displayed processes differ. This is because a process is created from the executable `/sbin/insmod` when the module is loaded and when the module is unloaded a process is created from the executable `/sbin/rmmod`.

Extra Exercises

1. KDB

Go to the **8-kdb** directory. Activate KDB over the serial port and enter KDB mode using **SysRq**. Connect to the pseudo-terminal linked to virtiocon0 using **minicom**, configure KDB to use the hvc0 serial port:

```
echo hvc0 > /sys/module/kgdboc/parameters/kgdboc
```

and enable it using SysRq (**Ctrl** + **O g**). Review the current system status (**help** to see the available KDB commands). Continue the kernel execution using the **go** command.

Load the `hello_kdb` module. The module will simulate a bug when writing to the `/proc/hello_kdb_bug` file. To simulate a bug, use the below command:

```
echo 1 > /proc/hello_kdb_bug
```

After running the above command, at every oops/panic the kernel stops the execution and enters debug mode.

Analyze the stacktrace and determine the code that generated the bug. How can we find out from KDB the address where the module was loaded?

In parallel, use GDB in a new window to view the code based on KDB information.

Hint

Load the symbol file. Use **info line**.

When writing to `/proc/hello_kdb_break`, the module will increment the `kdb_write_address` variable. Enter KDB and set a breakpoint for each write access of the `kdb_write_address` variable. Return to kernel to trigger a write using:

```
echo 1 > /proc/hello_kdb_break
```

2. PS Module

Update the created kernel module at proc-info in order to display information about all the processes in the system, when inserting the kernel module, not just about the current process. Afterwards, compare the obtained result with the output of the **ps** command.

Hint

- Processes in the system are structured in a circular list.
- `for_each _...` macros (such as `for_each_process`) are useful when you want to navigate the items in a list.
- To understand how to use a feature or a macro, use LXR or Vim and **cscope** and search for usage scenarios.

3. Memory Info

Create a kernel module that displays the virtual memory areas of the current process; for each memory area it will display the start address and the end address.

Hint

- Start from an existing kernel module.
- Investigate the structures `struct task_struct`, `struct mm_struct` and `struct vm_area_struct`. A memory area is indicated by a structure of type `struct vm_area_struct`.
- Don't forget to include the headers where the necessary structures are defined.

4. Dynamic Debugging

Go to the **9-dyndbg** directory and compile the `dyndbg.ko` module.

Familiarize yourself with the `debugfs` file system mounted in `/debug` and analyze the contents of the file `/debug/dynamic_debug/control`. Insert the `dyndbg.ko` module and notice the new content of the `dynamic_debug/control` file.

What appears extra in the respective file? Run the following command:

```
grep dyndbg /debug/dynamic_debug/control
```

Configure **dyndbg** so that only messages marked as "Important" in `my_debug_func()` function are displayed when the module is unloaded. The exercise will only filter out the `pr_debug()` calls; `printk()` calls being always displayed.

Specify two ways to filter.

Hint

Read the Dynamic debugging section and look at the **dyndbg** options (for example, **line**, **format**).

Perform the filtering and revise the `dynamic_debug/control` file. What has changed? How do you know which calls are activated?

Hint

Check the **dyndbg** flags. Unload the kernel module and observe the log messages.

5. Dynamic Debugging During Initialization

As you have noticed, `pr_debug()` calls can only be activated /filtered after module insertion. In some situations, it might be helpful to view the messages from the initialization of the module. This can be done by using a default (fake) parameter called **dyndbg** that can be passed as an argument to initialize the module. With this parameter you can add /delete **dyndbg** flags.

Hint

Read the last part of the Dynamic debugging section and see the available flags (e.g.: +/-**p**).

Read the Debug Messages section at Module Initialization Time and insert the module so that the messages in `my_debug_func()` (called `dyndbg_init()`) are also displayed during initialization.

Warning

In the VM from the lab, you will need to use **insmod** instead of **modprobe**.

Without unloading the module, deactivate `pr_debug()` calls.

Hint

You can delete the set flags. Unload the kernel module.