SYS2

Linux kernel

20xx

Version 1

Jules AUBERT <jules1.aubert@epita.fr>

Contents

1 Introduction

1.1 Goal

The goal of this course is to dive into the Linux kernel and play around with it.

1.2 Get the code

You can clone the Linux repo with **git(1)**:

```
$ git clone https://github.com/torvalds/linux.git
(...)
$
```

Be aware that it will download **the entirely of the history of the repo with all the commits, branches, tags, ...** It takes some gigabytes. Once you are inside the repo, you can execute the following command to change the repo into the last version of the kernel (v6.14 as I am writing this document).

```
$ git checkout tags/v6.14
...
$
```

If you want to download the commit of the last version, you can execute the following command:

```
$ git clone --depth 1 -b v6.14 https://github.com/torvalds/linux.git
(...)
$
```

1.3 Documentation

1.3.1 Online

Here are some useful links for you.

- https://kernel.org ; Linux kernel official website
- https://docs.kernel.org ; Linux official documentation
- https://github.org/torvalds/linux ; Linux official repo

1.3.2 Offline

Once you have cloned the Linux repo, you can execute

```
$ make htmldocs
(...)
$
```

to have the documentation offline in ./Documentation/output/index.html

Be aware that you need specific packets, they're given by the **make(1)** command when it failed to build the doc.

You will also encounter errors because of functions already implemented, do not worry, this is a known bug, nothing to fear about.

1.4 Virtualization

You are grown up now, you will use **QEMU** as the virtualization software in this tutorial. You will experiment its strength.

If it's not done, install GDB aswell.

```
$ sudo apt install -y qemu-system gdb
(...)
$
```

1.5 Environment

Since EPITA has decided to install Ubuntu (sigh) on your laptops, I may have to write some specific instructions for you.

Why?

Because Ubuntu is installed with Secure Boot and compiles some compilation tools (gcc, ld, ...) with some flags to keep security. This added security is a PITA (Pain In The ... what? You thought I was writing EPITA without the 'E'?) for us.

Know that the PIE[1] uses NixOS for the environment. As for myself, I am using the former distro on the PIE, Arch[2] Linux. Both are great to use, you should consider testing other Linux distros on your free time.

[1] Parc Informatique de l'EPITA
[2] (I use Arch btw)

2 Compile

2.1 Goal

The goal of this chapter is to compile the Linux kernel.

2.2 Config

Once you have checked out the version you want to use, you can configure the compilation.

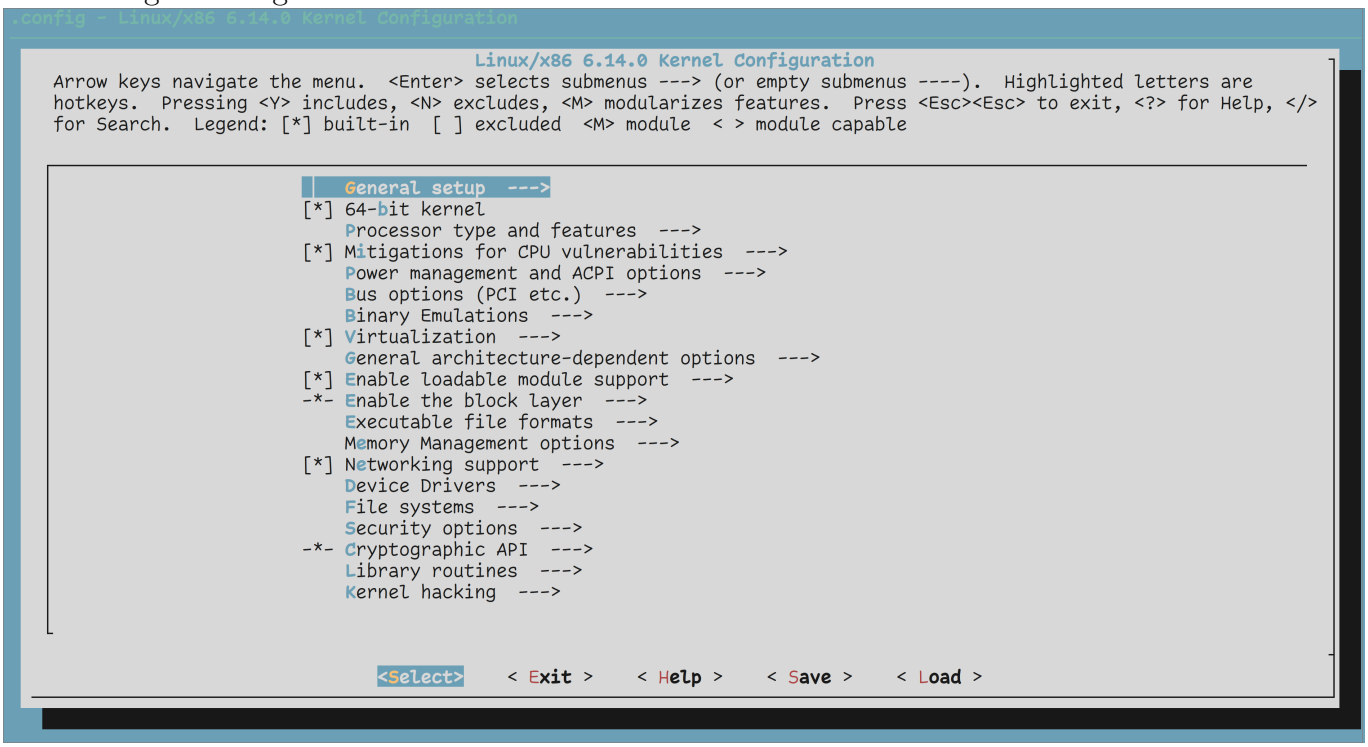
You can have the **default** configuration by calling:

```
$ make defconfig
(...)
$
```

Execute it. We will configure our own configuration from this default configuration.

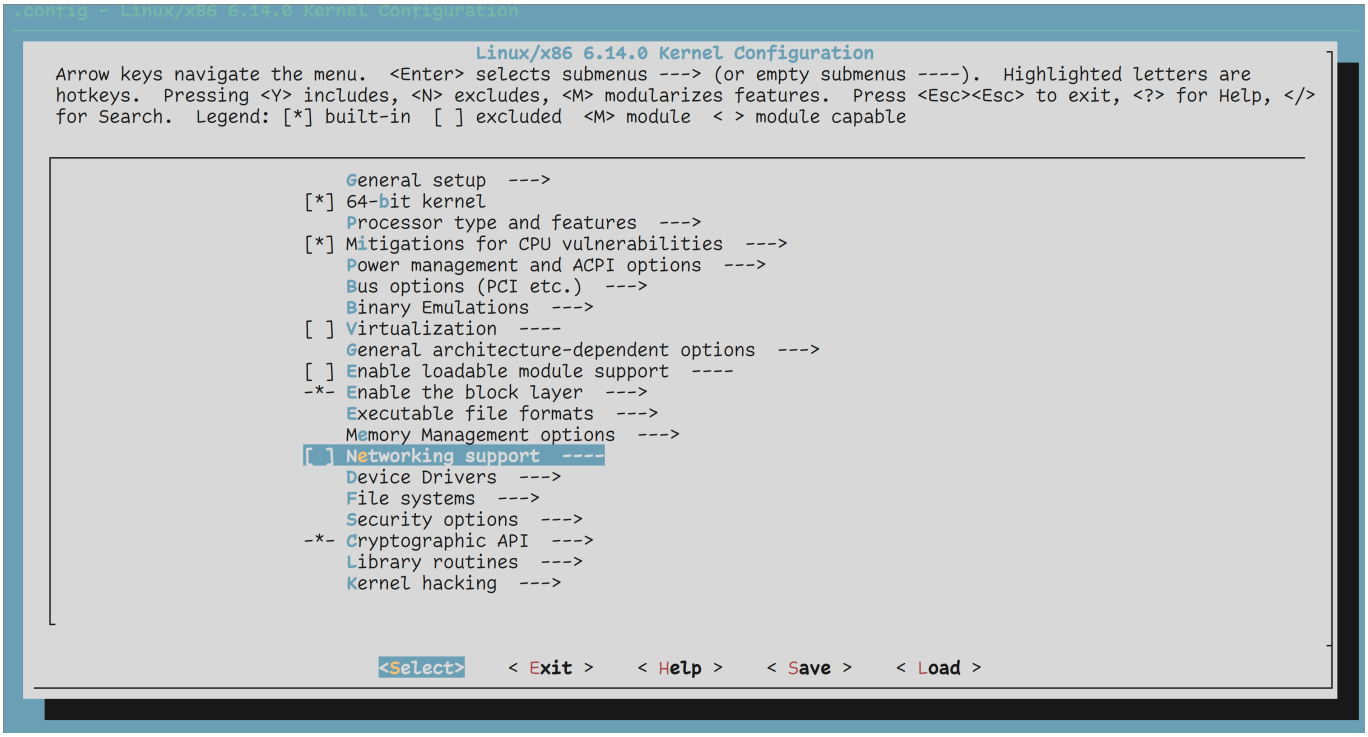
```
$ make menuconfig
(...)
$
```

Let's configure it together!

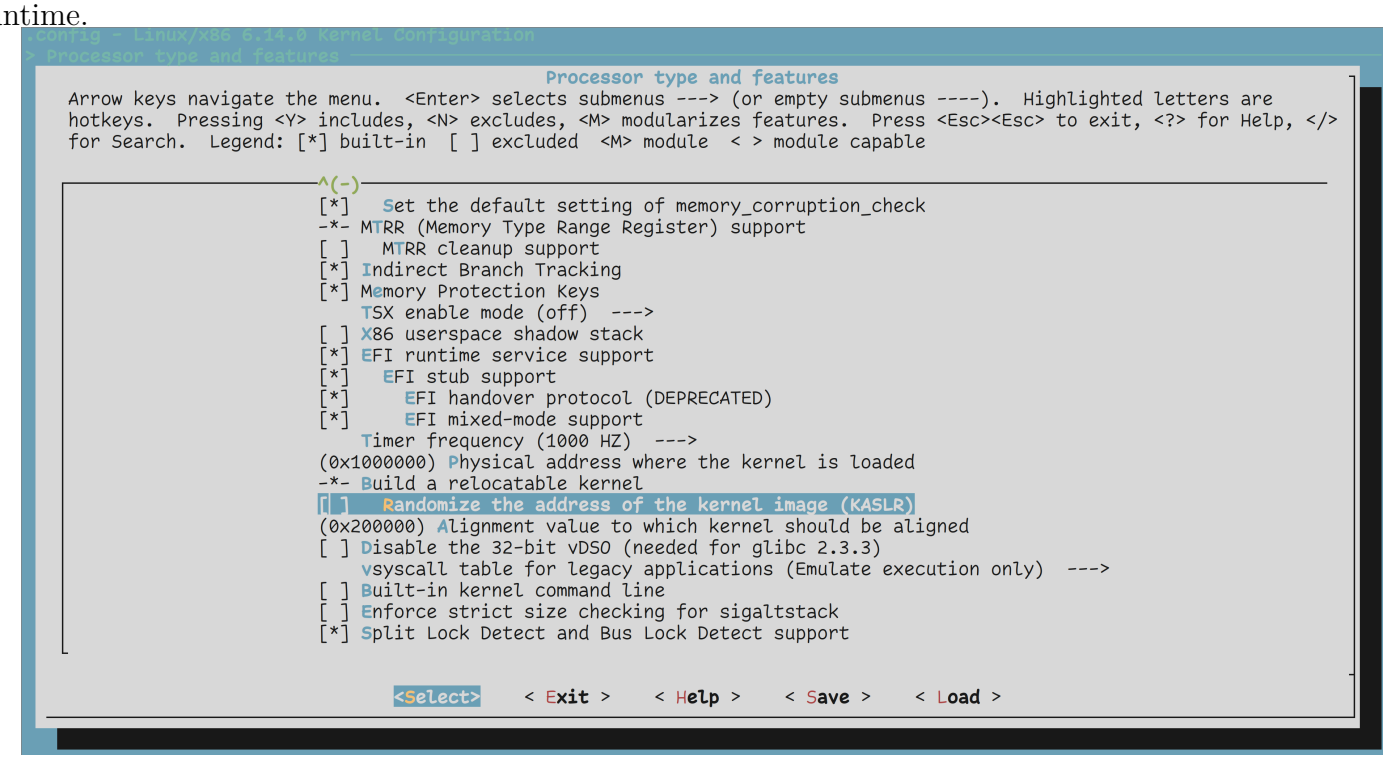


This is a Text-based User Interface (TUI). You can go from menu to menu with the arrow keys and the Return key, and going back with the Escape key. You can change selection below with the Tab key. I let you read the first paragraph above the list of menus to help you use this program.

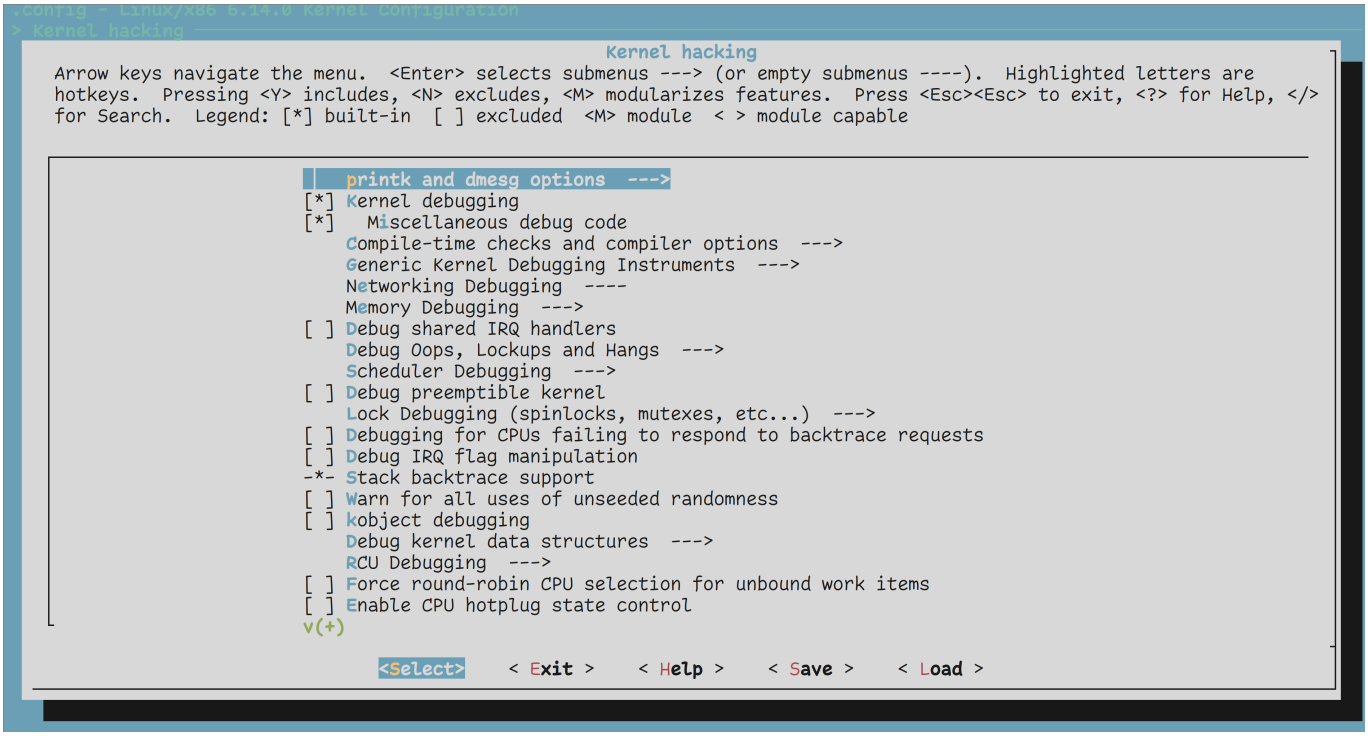
First of all, from the root menu, uncheck **Virtualization**, **Enable loadable module support** and **Networking support**. Press the Space key to check or uncheck selections.



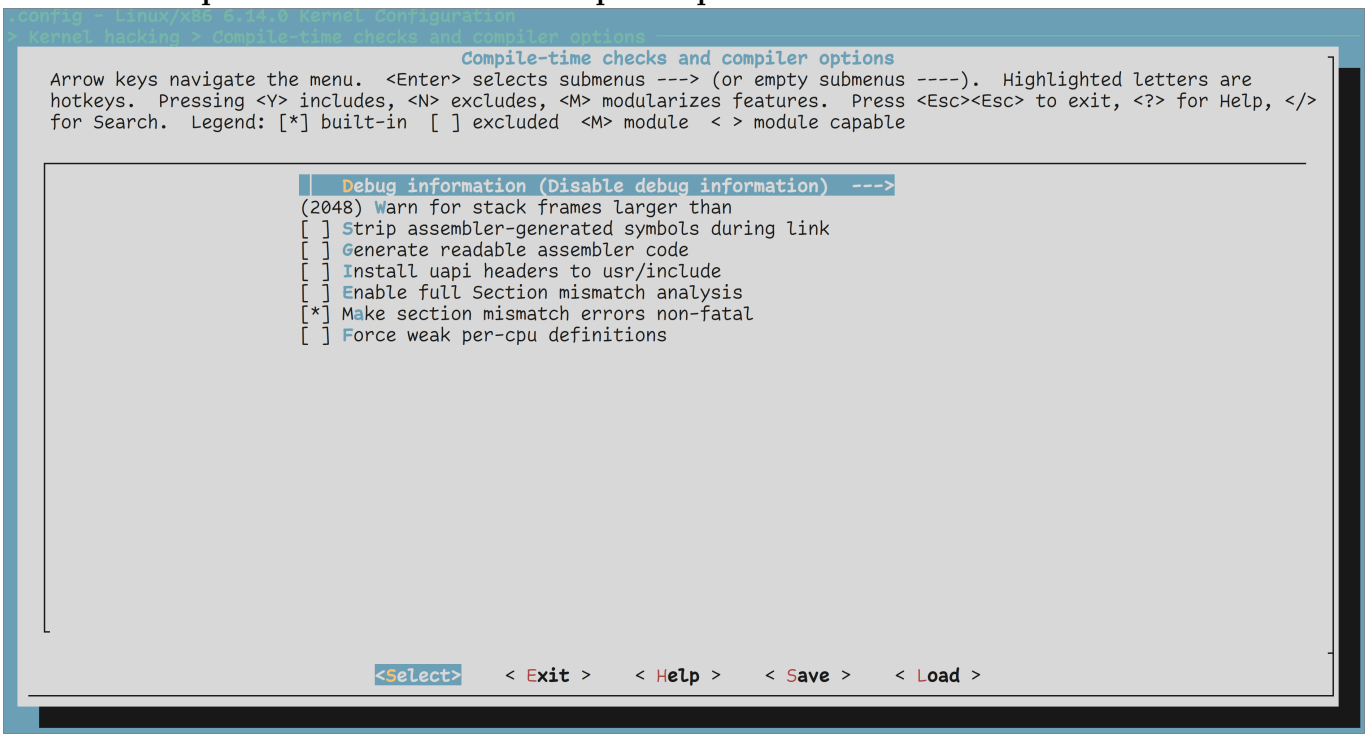
Now, go to **Processor type and features**, and at the end of this menu, uncheck **Randomize the address of the kernel image (KASLR)**. This is to help **gdb(1)** to debug the kernel at runtime.



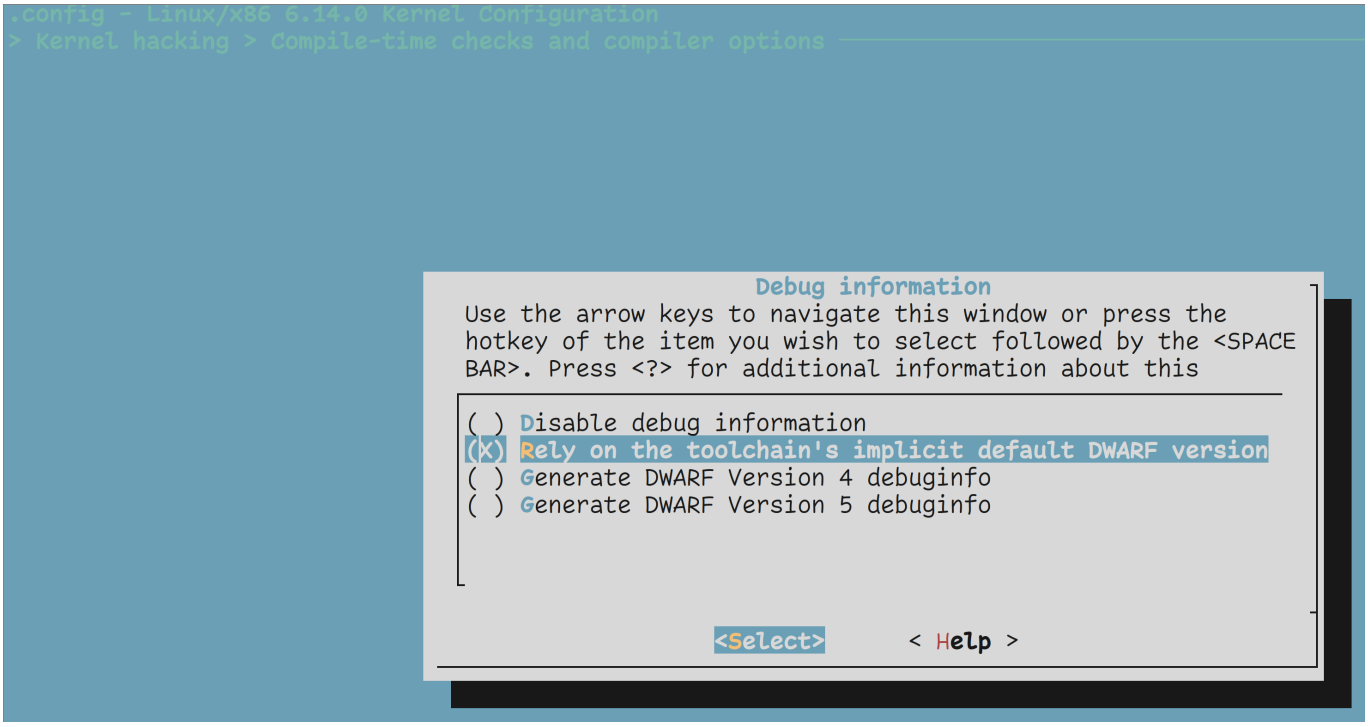
Exit this menu to go back to the root menu, and head into **Kernel hacking** submenu.



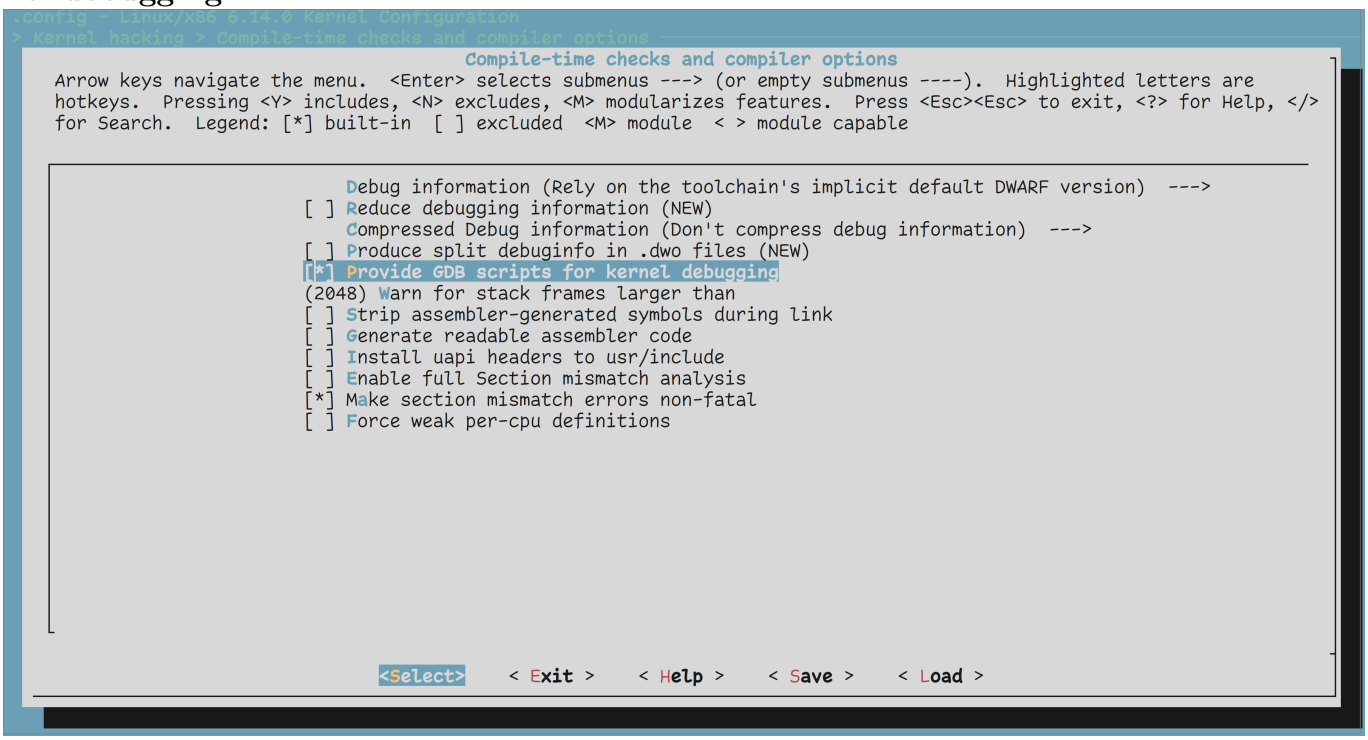
Get into **Compile-time checks and compiler options**



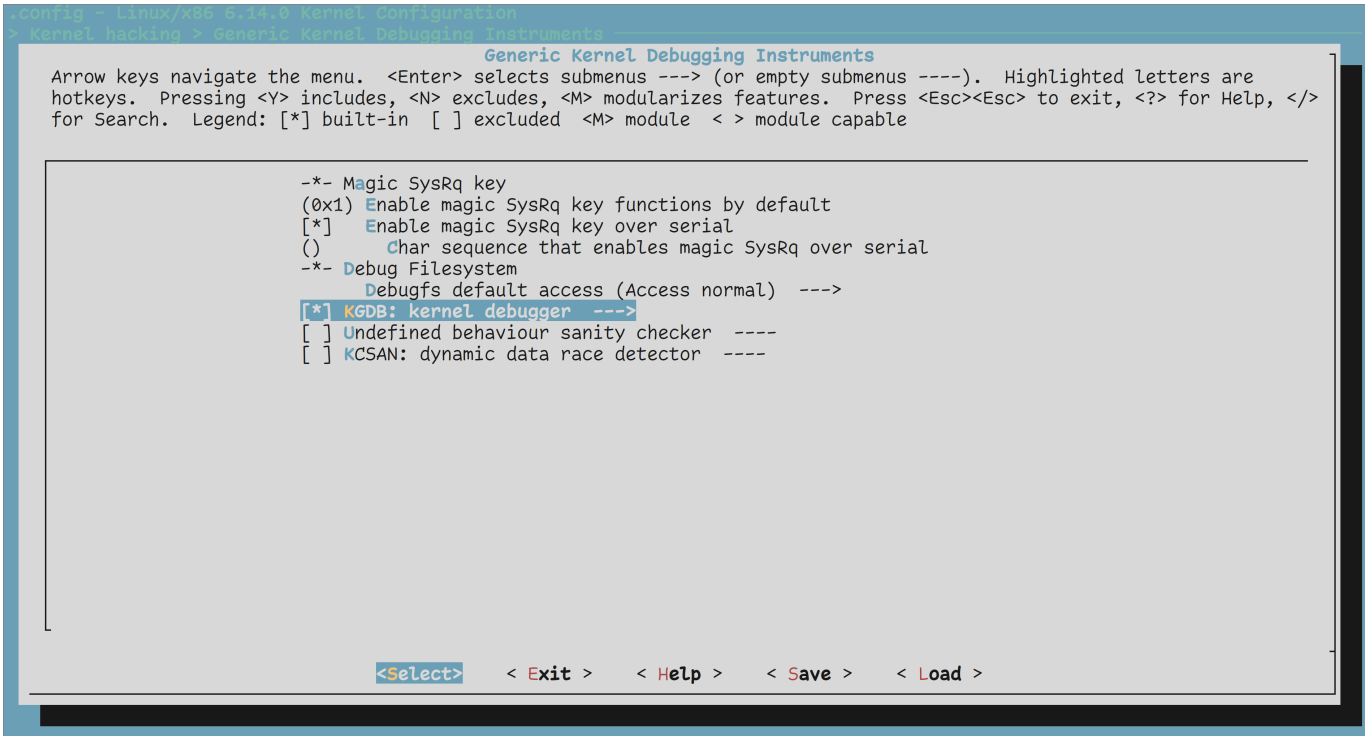
and then get into **Debug information** and check **Rely on the toolchain's implicit default DWARF version**.



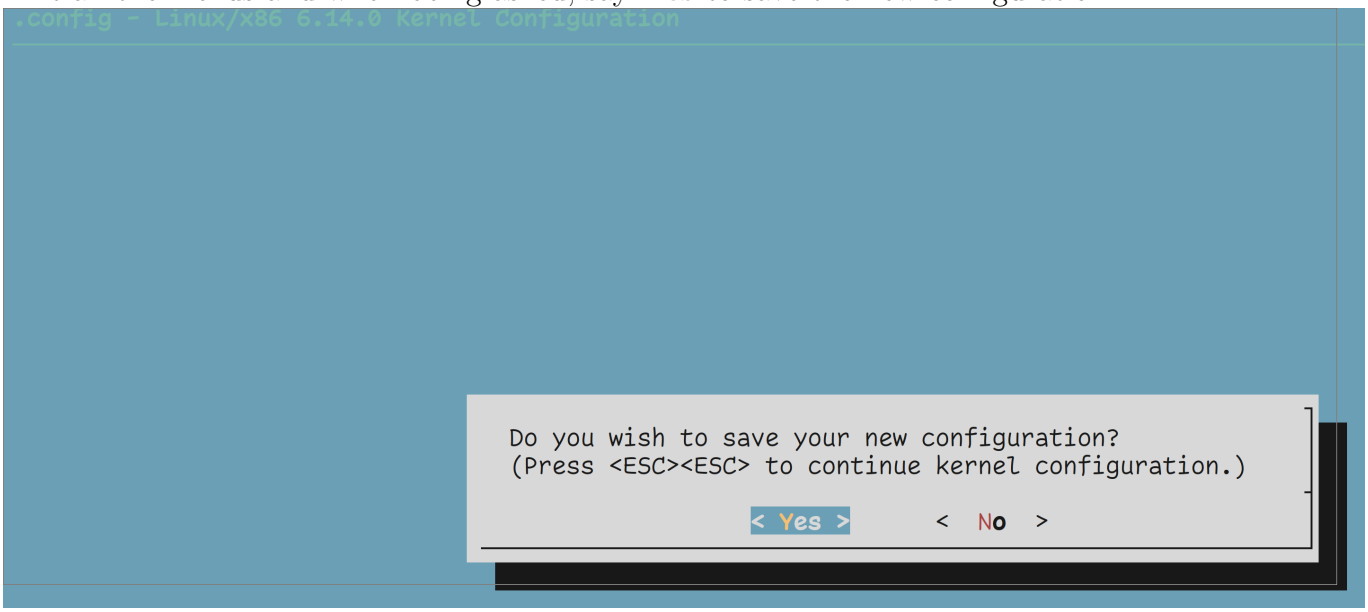
Still in the **Compile-time checks and compiler options**, check **Provide GDB scripts for kernel debugging**.



Go back to the **Kernel hacking** submenu, get into **Generic Kernel Debugging Instruments** and check **KGDB: Kernel debugger**.



Exit all the menus and when being asked, say **Yes** to save the new configuration.




```
configuration written to .config

*** End of the configuration.
*** Execute 'make' to start the build or try 'make help'.
```

Your configuration has been saved into the **.config** hidden file. Maybe you noticed it in the menuconfig looking up the name of the config file on the top left corner of the screen.

2.2.1 Ubuntu config

Before compiling the kernel, we need to discuss about the Ubuntu configuration. You need to uncheck two options to deactivate the use of security keys. If you do not uncheck them, the compilation will ask you to give certificates to continue the compilation.

We could search for the options to deactivate within the menuconfig... or just call a script to deactivate them from the command-line.

From the root repo directory, execute:

```
$ ./scripts/config --disable SYSTEM_TRUSTED_KEYS
$ ./scripts/config --disable SYSTEM_REVOCATION_KEYS
```

2.3 gdb scripts

Before compiling the kernel, you need to generate the scripts for gdb.

```
$ make scripts_gdb
(...)
$
```

2.4 Compile

Time to compile the kernel! Depending of the power of your computer, you can add jobs to compile several files at the same time. Here, I am using 16 jobs. You can add the **time(1)** command to see how long it takes for your computer to compile Linux.

```
$ time make -j 16
(...)
Kernel: arch/x86/boot/bzImage is ready  (#9)
make -j 16  1118.59s user 332.27s system 358% cpu 6:44.59 total
$
```

It took me 6 minutes and 44 seconds to compile my kernel. :)

2.5 The kernels

Among all the generated files, two are important:

- arch/x86/boot/bzImage: the compressed kernel to boot on
- vmlinux: the kernel containing all the debug information

You can also use arch/x86_64/boot/bzImage, it is a symbolic link to the x86 file.
I let you check the file info and sizes of the two generated files.

```
$ file vmlinux arch/x86/boot/bzImage
(...)
$ ls -lh vmlinux arch/x86/boot/bzImage
(...)
```

3 Boot

3.1 Goal

The goal of this chapter is to boot the compiled Linux kernel and join a gdb server to debug the execution.

3.2 QEMU

QEMU (Quick Emulator) is a free and open-source emulator and virtualizer that allows you to run programs and operating systems for one hardware architecture on another. To call QEMU on the Linux kernel on an x64 architecture (Intel 64 bits), execute:

```
$ qemu-system-x86_64 -kernel arch/x86/boot/bzImage &
$
```

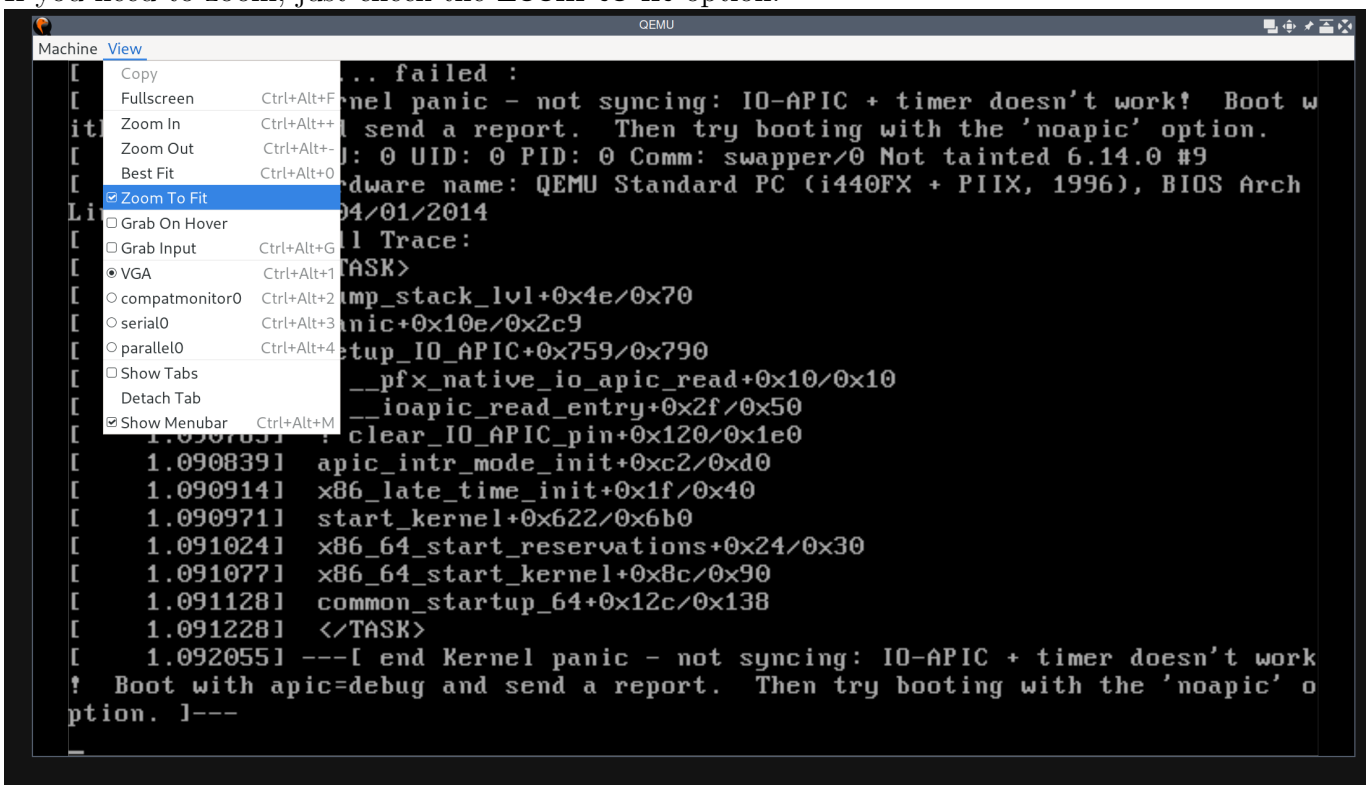
If QEMU shows any problem, it might be because you are on Ubuntu and/or have generated a big Linux kernel. Replace the command with this:

```
$ qemu-system-x86_64 -kernel arch/x86/boot/bzImage -m 1G &
$
```

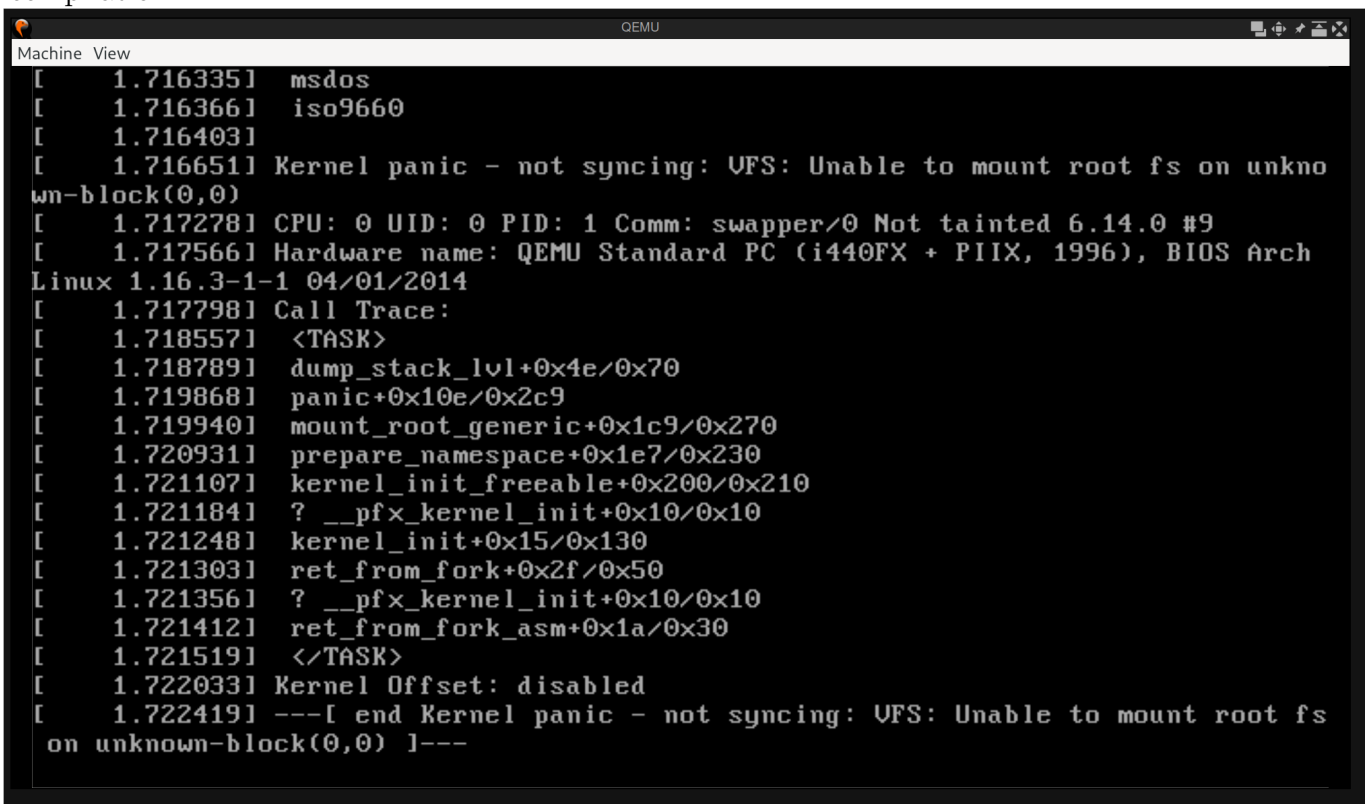
By default, QEMU emulates 128 MB of RAM, the **-m** parameter is to tell QEMU to use 1 gigabyte of memory. With a small Linux you do not need it. But for the same compile configuration file, two Linux distros will generate different sized kernels. For example, Arch will generate a bzImage smaller than 10 MB while on Ubuntu it will generate a bzImage of 42 MB. The decompression makes the kernel compiled on Ubuntu too big while on Arch it works just fine :).

If you ever click with your cursor in the QEMU window and cannot get it back to your system, press **Ctrl+Alt+G** to release grab (written on the window of QEMU).

If you need to zoom, just check the **Zoom to fit** option.



What you seen now is a **kernel panic**. It is a state where Linux is in a critical error and cannot recover. The only part of code still in execution is the one to make the cursor blink. If you do not see the kernel panic or a blinking cursor, it may comes from some weird Ubuntu shenanigans during the compilation.



3.3 gdb server

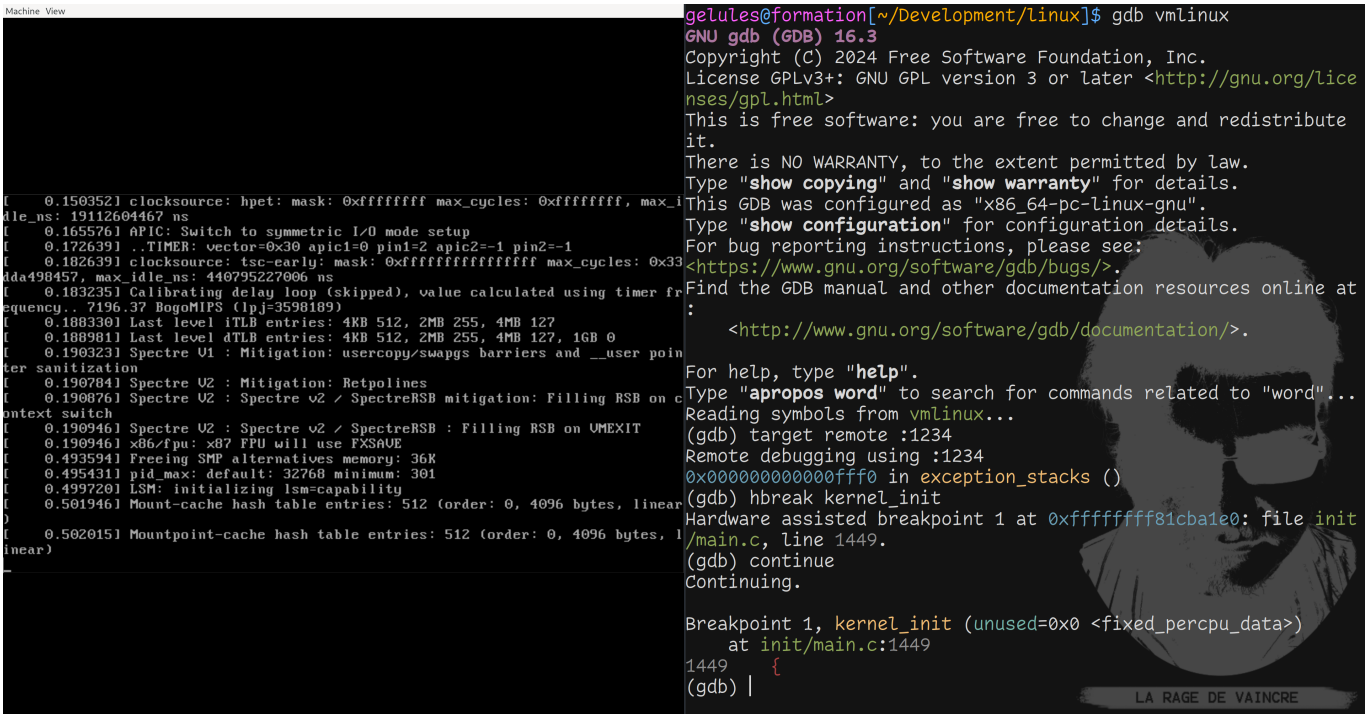
We are going to execute the kernel with a gdb server in QEMU on the port 1234. We can then attach gdb to the remote server and debug Linux inside QEMU using **vmlinux** debug information.

```
$ qemu-system-x86_64 -kernel arch/x86/boot/bzImage -s -S -append nokaslr &
$
```

- -s: It is the parameter to execute the gdb server
- -S: It is the parameter to execute QEMU in freeze mode until we continue the execution through gdb
- -append nokaslr: Append the **No Kernel ASLR** to avoid kernel address space layout randomization and help gdb to have hardcoded address to ease the debugging

To connect to the remote server, execute gdb this way:

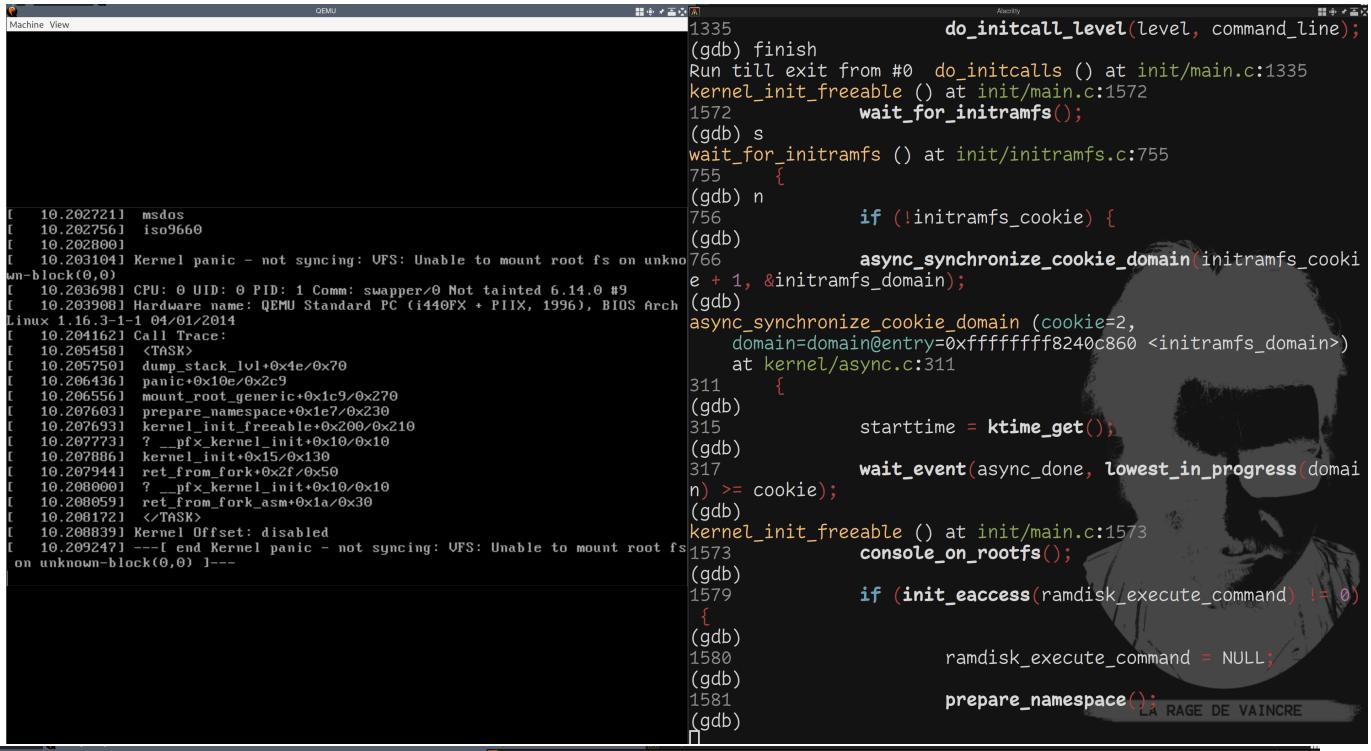
```
$ gdb vmlinux
(gdb) target remote :1234
(gdb) hbreak kernel_init
(gdb) continue
$
```

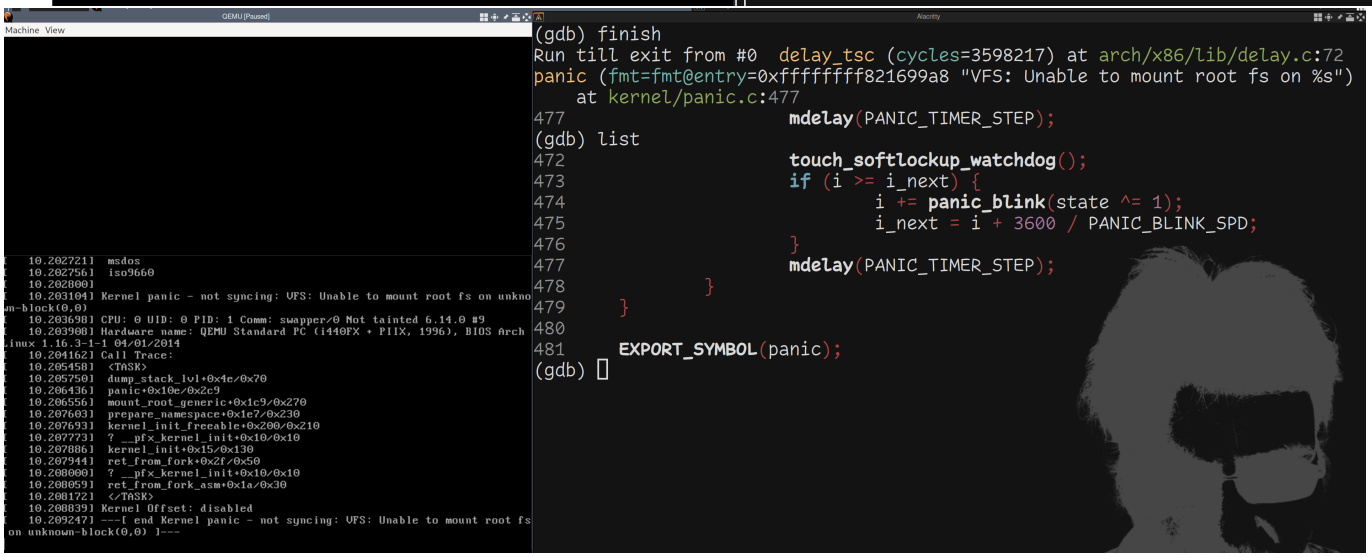


If you want to see the coming instructions, you can execute **list** in gdb. Remember, if you press the Return key, it will reexecute the last instruction.

You can continue to see how to boot works. There's nothing to see here really. I just want you to learn how to debug a Linux kernel on QEMU and investigate into the code. Anyway you will end up on a kernel panic.

You can see the kernel booting at each function you **next** or **step-in**. Can you find the instructions where the kernel panic occurs and the cursor starts blinking?


QEMU
Machine View
[10.202721] msdos
[10.202756] iso9660
[10.202800]
[10.203104] Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(0,0)
[10.203698] CPU: 0 UID: 0 PID: 1 Comm: swapper/0 Not tainted 6.14.0 #9
[10.203908] Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS Arch Linux 1.16.3-1-1 04/01/2014
[10.204162] Call Trace:
[10.205458] <TASK>
[10.205750] dump_stack_lvl+0x4e/0x70
[10.206436] panic+0x10e/0x2c9
[10.206556] mount_root_generic+0x1c9/0x270
[10.207603] prepare_namespace+0x1e7/0x230
[10.207693] kernel_init_freeable+0x200/0x210
[10.207773] ? __pfx_kernel_init+0x10/0x10
[10.207886] kernel_init+0x15/0x130
[10.207944] ret_from_fork+0x2f/0x50
[10.208000] ? __pfx_kernel_init+0x10/0x10
[10.208059] ret_from_fork_asm+0x1a/0x30
[10.208172] </TASK>
[10.208839] Kernel Offset: disabled
[10.209247] ---[end Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(0,0)]---
1335 do_initcall_level(level, command_line);
(gdb) finish
Run till exit from #0 do_initcalls () at init/main.c:1335
kernel_init_freeable () at init/main.c:1572
1572 wait_for_initramfs();
(gdb) s
wait_for_initramfs () at init/initramfs.c:755
755 {
(gdb) n
756 if (!initramfs_cookie) {
(gdb)
766 async_synchronize_cookie_domain(initramfs_cookie + 1, &initramfs_domain);
(gdb)
async_synchronize_cookie_domain (cookie=2, domain=domain@entry=0xffffffff8240c860 <initramfs_domain>)
at kernel/async.c:311
311 {
(gdb)
315 starttime = ktime_get();
(gdb)
317 wait_event(async_done, lowest_in_progress(domain) >= cookie);
(gdb)
kernel_init_freeable () at init/main.c:1573
1573 console_on_rootfs();
(gdb)
1579 if (init_eaccess(ramdisk_execute_command) != 0) {
(gdb)
1580 ramdisk_execute_command = NULL;
(gdb)
1581 prepare_namespace();
(gdb)
LA RAGE DE VAINCRE


QEMU [Paused]
Machine View
[10.202721] msdos
[10.202756] iso9660
[10.202800]
[10.203104] Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(0,0)
[10.203698] CPU: 0 UID: 0 PID: 1 Comm: swapper/0 Not tainted 6.14.0 #9
[10.203908] Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS Arch Linux 1.16.3-1-1 04/01/2014
[10.204162] Call Trace:
[10.205458] <TASK>
[10.205750] dump_stack_lvl+0x4e/0x70
[10.206436] panic+0x10e/0x2c9
[10.206556] mount_root_generic+0x1c9/0x270
[10.207603] prepare_namespace+0x1e7/0x230
[10.207693] kernel_init_freeable+0x200/0x210
[10.207773] ? __pfx_kernel_init+0x10/0x10
[10.207886] kernel_init+0x15/0x130
[10.207944] ret_from_fork+0x2f/0x50
[10.208000] ? __pfx_kernel_init+0x10/0x10
[10.208059] ret_from_fork_asm+0x1a/0x30
[10.208172] </TASK>
[10.208839] Kernel Offset: disabled
[10.209247] ---[end Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(0,0)]---
(gdb) finish
Run till exit from #0 delay_tsc (cycles=3598217) at arch/x86/lib/delay.c:72
panic (fmt=fmt@entry=0xffffffff821699a8 "VFS: Unable to mount root fs on %s")
at kernel/panic.c:477
477 mdelay(PANIC_TIMER_STEP);
(gdb) list
472 touch_softlockup_watchdog();
473 if (i >= i_next) {
474 i += panic_blink(state ^= 1);
475 i_next = i + 3600 / PANIC_BLINK_SPD;
476 }
477 mdelay(PANIC_TIMER_STEP);
478 }
479 }
480
481 EXPORT_SYMBOL(panic);
(gdb)

4 Init

4.1 Goal

The goal of this chapter is to create an initrd with a shell and be able to look around when the kernel executes it.

4.2 initrd

initrd (initial RAM disk) is a temporary filesystem loaded into memory by the bootloader and used by the Linux kernel during the early stages of the boot. Its main purpose is to provide the necessary drivers and tools the kernel needs to access the real distro filesystem. It also detects and prepare storage devices.

To create the initrd, you will use **cpio(1)**.

4.2.1 cpio

cpio (Copy In Copy Out) is a simple archive format to package several files. You know the tar format? Now you know a new format: cpio. You do not need to know how it works, just how to make one.

First of all, download the **source of bash**.

https://ftp.gnu.org/gnu/bash/

Download the latest version.

To ease the next chapter, along the **linux repo directory**, create the following directories: **bash** and **init**. The first one will contain the source of bash, the second one is where you will create your initrd.

Download the source of bash, untar it and compile it to have a **static** binary.

```
$ tar xf bash.tar.gz
$ cd bash
$ ./configure --enable-static-link
(...)
$ make -j 16
(...)
$ ldd bash
  not a dynamic executable
$ file bash
bash: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux), statically linked, BuildID[sha1]=
    e7af2f7c07ece1965112ab98534b446b55bef75c, for GNU/Linux 4.4.0, with debug_info, not stripped
```

As you can see, bash is not dynamic, it is a static binary. Remember your Piscine? A static binary is a binary with no dependancies, with no libraries that must be present on the system. You can execute your compiled bash into the initrd without putting the shared objects.

4.3 Creating the initrd

Copy the bash binary in your **init** folder. Now, execute **cpio(1)** to create the initrd.

```
$ mv bash init
$ echo init | cpio -o -H newc > initrd.img
$ ls
(...)
$ file initrd.img
initrd.img: ASCII cpio archive (SVR4 with no CRC)
```

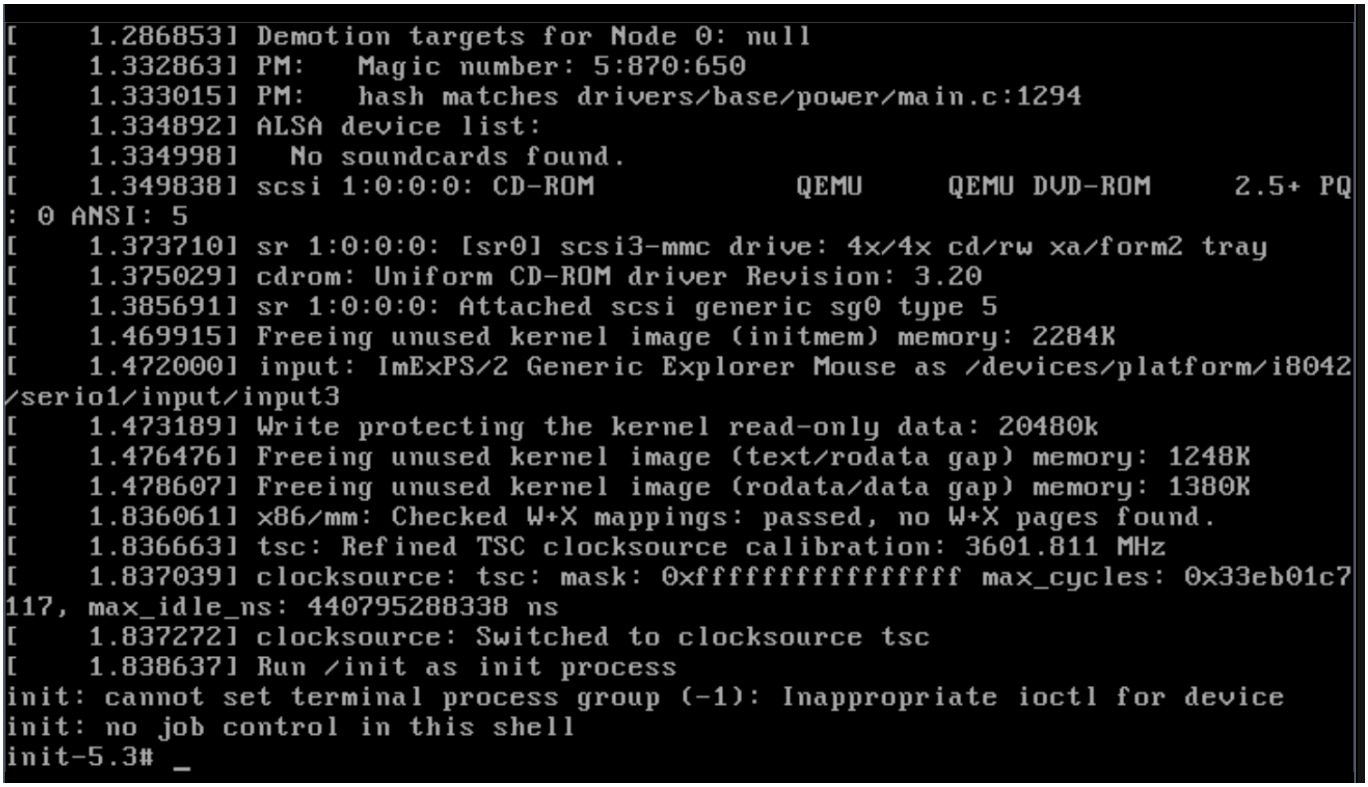
The **newc** is the format you will use for the initrd. It is one of the latest format used in cpio.

If you want to unarchive your initrd, just execute:

```
$ cpio -i < initrd.img
(...)
$
```

Go back to the **linux repo** and execute QEMU this way:

```
$ qemu-system-x86_64 -kernel arch/x68/boot/bzImage -initrd ../init/initrd.img -append "init=/init"
$
```



If you do not see the same result, press the Return key, you should get your shell. Sometimes kernel logs take over the stdout.

Hurray! We got a shell! Now what? Try to look around, is there only the **init** binary file? Just execute **ls(1)** to list your directory. :)

Oh... wait? **ls(1)** is not present, right? How can we achieve such a simple task as listing a directory?

With builtin commands!

Just execute **echo *** to list all the files present.

```
# echo *
dev init root
#
```

- dev: the **device** directory
- init: the binary file you are currently executing
- root: the placeholder for the root filesystem of the disto initrd should mount and pivot_root(2) (chroot(1)) in

The only interesting place here is **dev**. Let's dive in!

```
# cd dev
# echo *
console
# echo "Hello APPING" > console
Hello APPING
#
```

console is a special character file linked to your current **tty(4)**.

You now know how to create an initrd file and boot on it. In the following chapters, we use a C program as an init, boot on it and debug it from the gdb server to see what is happening inside the Linux kernel when executing C programs.

4.4 APPING init

Here is the file **apping_init.c** containing the code we will use as our init:

```c
#include <unistd.h>

int main(void)
{
    write(1, "APPING\n", 7);
    _exit(0);
}
```

And this is how you compile it:

```
$ gcc -static --entry main -o init apping_init.c
$ file init
(...)
$ ldd init
(...)
$
```

What is the interest of **–entry main**? And why do we have to use **_exit(2)** (the syscall, not the value "2")?

When you compile a C code into a binary, the main function is not really the entry point. It was a lie all along. The real entry function is **__init**. You can see it as a **constructor**. It initializes the program environment and then calls the **main** function.

The reason we use **__exit(2)** is that we want to avoid the call to the destructor, which is a **__fini**. It is called when the program exits and it is not our interest here.

Change the call to _exit with a simple return, you will have a segfault. Why? Because we have not called _init, so the program is deallocating memory never initialized in the first place.

4.4.1 Constructor and destructor in C

You thought constructors and destructors where only used by OOP langages? Que nenni! They are not. They are used by all languages, even C.

The **__init** function is called before the main function and it is used to initialize the program environment. It is a good place to do some low level initialization like setting up the stack pointer, initializing the heap, etc...

The **__fini** function is called after the main function and it is used to cleanup the program environment. It is a good place to do some low level cleanup like freeing memory, closing file descriptors, etc...

Here is a simple example of a library defining a constructor and destructor in C:

```
#include <stdio.h>

void __attribute__((constructor)) my_init(void)
{
    printf("Hello from the constructor!\n");
}

void __attribute__((destructor)) my_fini(void)
{
    printf("Goodbye from the destructor!\n");
}
```

You can compile it with:

```
$ gcc -c -o apping_lib.o apping_lib.c
$
```

Now let's code a program "*using*" our library:

```
#include <stdio.h>

int main(void)
{
    printf("Hello from the program!\n");
    return 0;
}
```

You can compile it with:

```
$ gcc -o apping_program apping_program.c apping_lib.o
$
```

And execute it this way:

```
$ ./apping_program
Hello from the constructor!
Hello from the program!
Goodbye from the destructor!
$
```

4.5 Booting on APPING init

Now let's boot on this new init.

First of all, create the new initrd with our apping_init.

```
$ mv apping_init init
$ echo init | cpio -o -H newc > initrd.img
```

And boot on the kernel and our new initrd with QEMU.

```
$ qemu-system-x86_64 -kernel arch/x86/boot/bzImage -initrd ../init/initrd.img -append "nokaslr init=/init"
$
```

Damn! The execution occurs too fast and only the kernel panic is finally displayed. Impossible to go back above the logs. Do not worry, we will make the program wait after the call to **write(2)**.

Update the code this way:

```
#include <limits.h>
#include <unistd.h>

int main(void)
{
    write(1, "APPING\n", 7);

    for (unsigned long long i = 0; i < LONG_MAX; i++);

    _exit(0);
}
```

This avoid a call to **sleep(3)**. You have plenty of time before **i** reaches **LONG_MAX**.

```
[    0.788260] Demotion targets for Node 0: null
[    0.790349] PM:   Magic number: 5:879:728
[    0.791251] ALSA device list:
[    0.791567]   No soundcards found.
[    0.910568] ata2: found unknown device (class 0)
[    0.914488] ata2.00: ATAPI: QEMU DVD-ROM, 2.5+, max UDMA/100
[    0.921250] scsi 1:0:0:0: CD-ROM            QEMU     QEMU DVD-ROM     2.5+ PQ
: 0 ANSI: 5
[    0.932555] sr 1:0:0:0: [sr0] scsi3-mmc drive: 4x/4x cd/rw xa/form2 tray
[    0.932737] cdrom: Uniform CD-ROM driver Revision: 3.20
[    0.937989] sr 1:0:0:0: Attached scsi generic sg0 type 5
[    0.979436] Freeing unused kernel image (initmem) memory: 2348K
[    0.979969] Write protecting the kernel read-only data: 20480k
[    0.981825] Freeing unused kernel image (text/rodata gap) memory: 1320K
[    0.982511] Freeing unused kernel image (rodata/data gap) memory: 928K
[    1.144132] x86/mm: Checked W+X mappings: passed, no W+X pages found.
[    1.144753] Run /init as init process
APPING
[    1.183604] input: ImExPS/2 Generic Explorer Mouse as /devices/platform/i8042
/serio1/input/input3
[    1.670222] tsc: Refined TSC clocksource calibration: 3399.942 MHz
[    1.670493] clocksource: tsc: mask: 0xffffffffffffffff max_cycles: 0x31021837
eaf, max_idle_ns: 440795290740 ns
[    1.670665] clocksource: Switched to clocksource tsc
```

For the following chapters, go back to the original apping_init. I made you wait just to see that it works.

```c
#include <unistd.h>

int main(void)
{
    write(1, "APPING\n", 7);
    _exit(0);
}
```

5 Exec

5.1 Goa

The goal of this chapter is to see how Linux loads a binary into memory for execution. We will use the **apping_init**

5.2 Execve

To put a binary into memory for execution, you use **execve(2)**. But what is happening inside execve? Let's find out!

From the linux repo, open the file **./fs/exec.c** and search for these lines:

```
SYSCALL_DEFINE3(execve,
    const char __user *, filename,
    const char __user *const __user *, argv,
    const char __user *const __user *, envp)
{
  return do_execve(getname(filename), argv, envp);
}
```

As you can see, the syscall definition for execve is calling **do_execve**.

We will not follow the code from the Linux source code, we will follow the code from the gdb server.

By the way? Do you know why is it using **DEFINE3**? Because there are **3** parameters for **write(2)**.

Before that, we need to make a new initrd, containing **bash** as an init and our apping_init program.

Go back to your init directory and place bash and apping_init together in this directory.

```
$ mv bash init
$ echo init > list
$ echo apping\_init >> list
$ cat list | cpio -o -H newc > initrd.img
```

Now, let's boot using this new initrd!

From now on, because the command line to boot is pretty long, I will only use the name of the bzImage and initrd.img instead of the whole path.

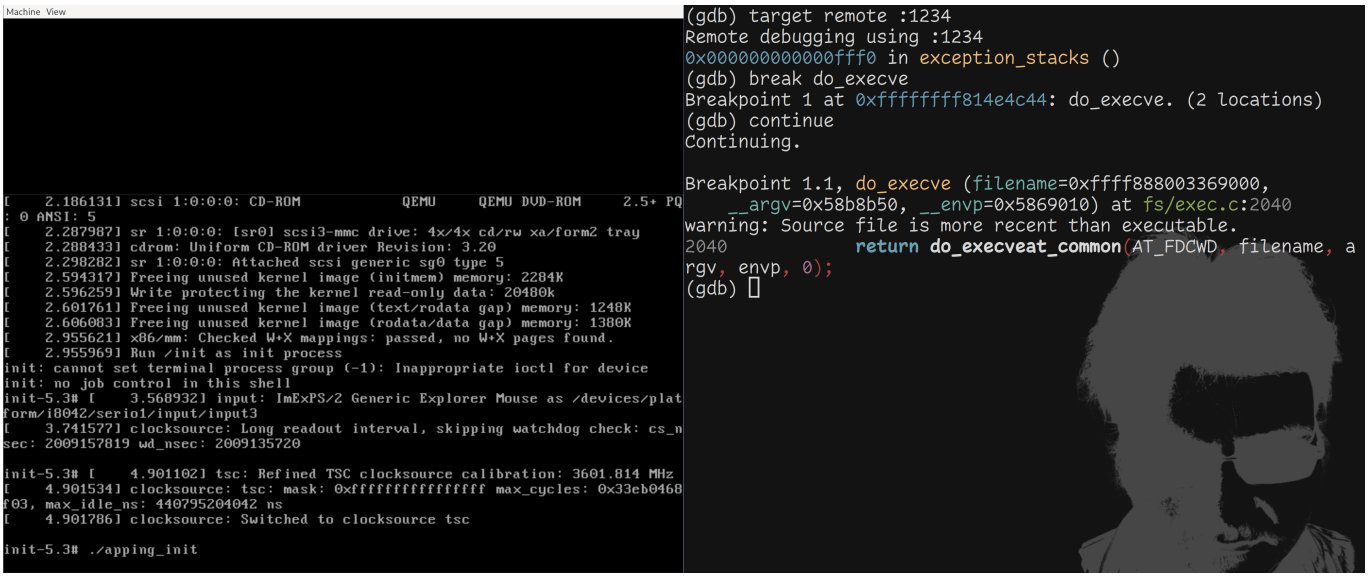
```
$ qemu-system-x86_64 -kernel bzImage -initrd initrd.img -append "nokaslr init=/init" -s -S &
$ gdb vmlinux
(gdb) target remote :1234
(gdb) break do_execve
(gdb) continue
(...)
```

Now from QEMU, execute **apping_init**

```
# ./apping_init
(...)
```

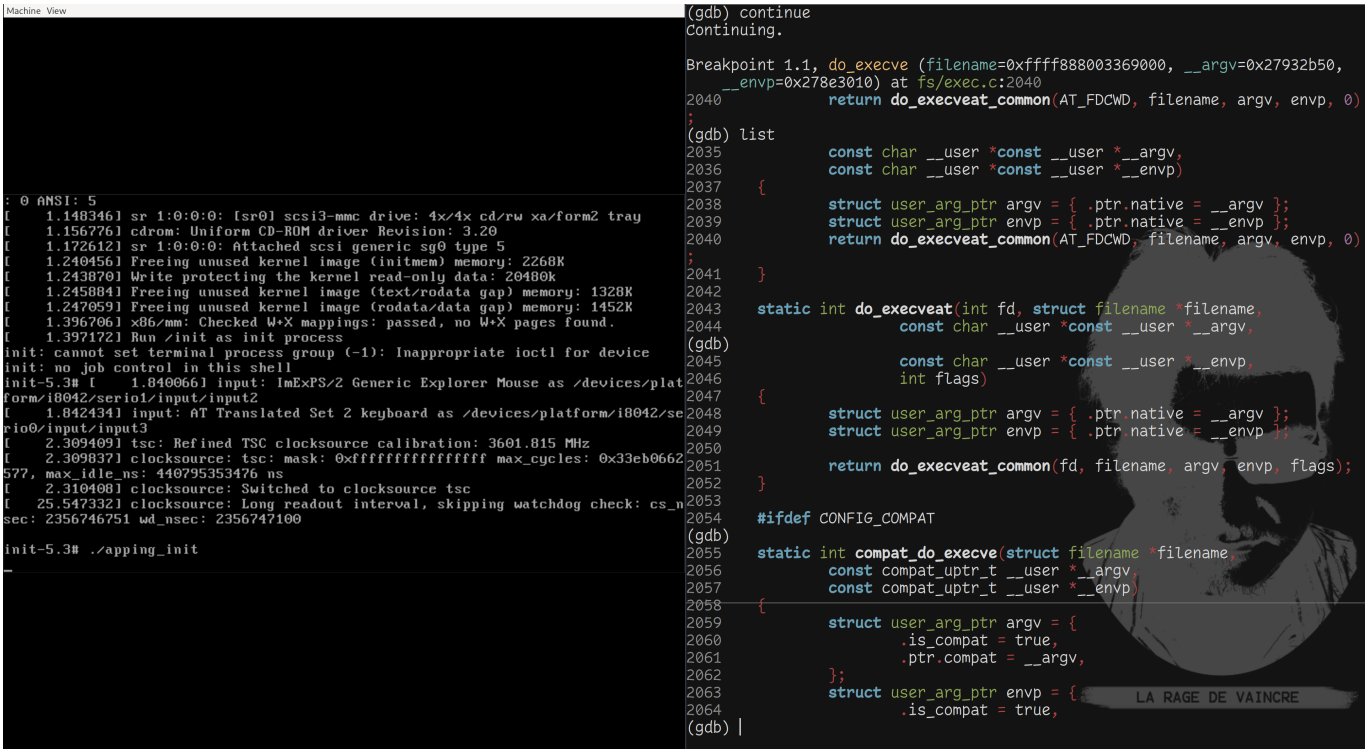
gdb will catch the breakpoint.

```
(gdb) # Breakpoint reached
```



You can use the **list** gdb command to see what instructions are coming. Remember that pressing the Return key will execute the last instruction.

```
(gdb) # Breakpoint reached
(gdb) list
(...)
(gdb)
(...)
(gdb)
(...)
```



To not make this PDF a +1024 pages document, you will break on some interesting functions and not follow the whole flow of execve which you would finish for your graduation day if the flow was completly explained her if the flow was completly explained here.

break on the **search_binary_handler** function and continue the execution.

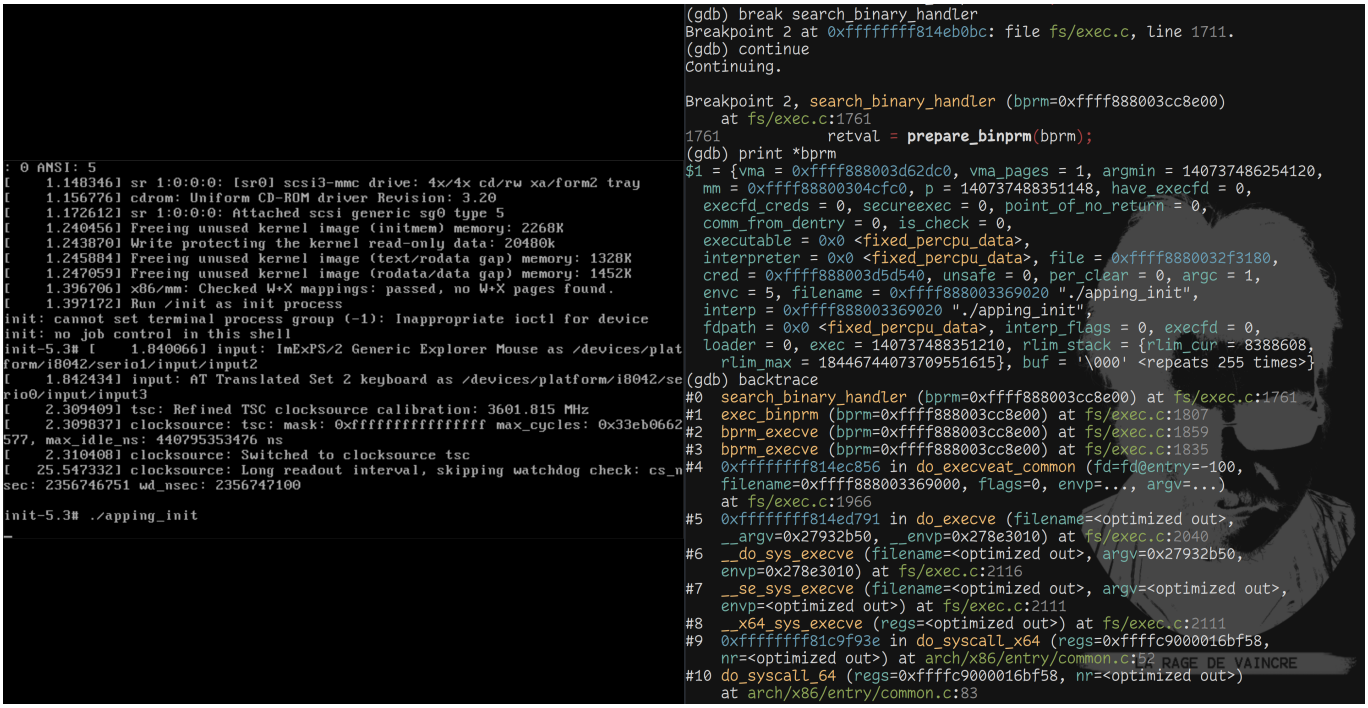
```
(gdb) break search_binary_handler
(gdb) continue
(...)
(gdb) # Breakpoint reached
```

You are now in **search_binary_handler** with a **bprm** struct as an argument. **bprm** means **binary parameters**. You can print its value in gdb. Since it is a pointer, you need to dereference it using an asterisk.

```
(gdb) print *bprm
(...)
```

If you wish to know how you ended-up here, you can execute the **backtrace** (or **bt**) command. As always, feel free to **list** the following instructions.

```
(gdb) bt
(...)
(gdb) list
(...)
(gdb)
(...)
```



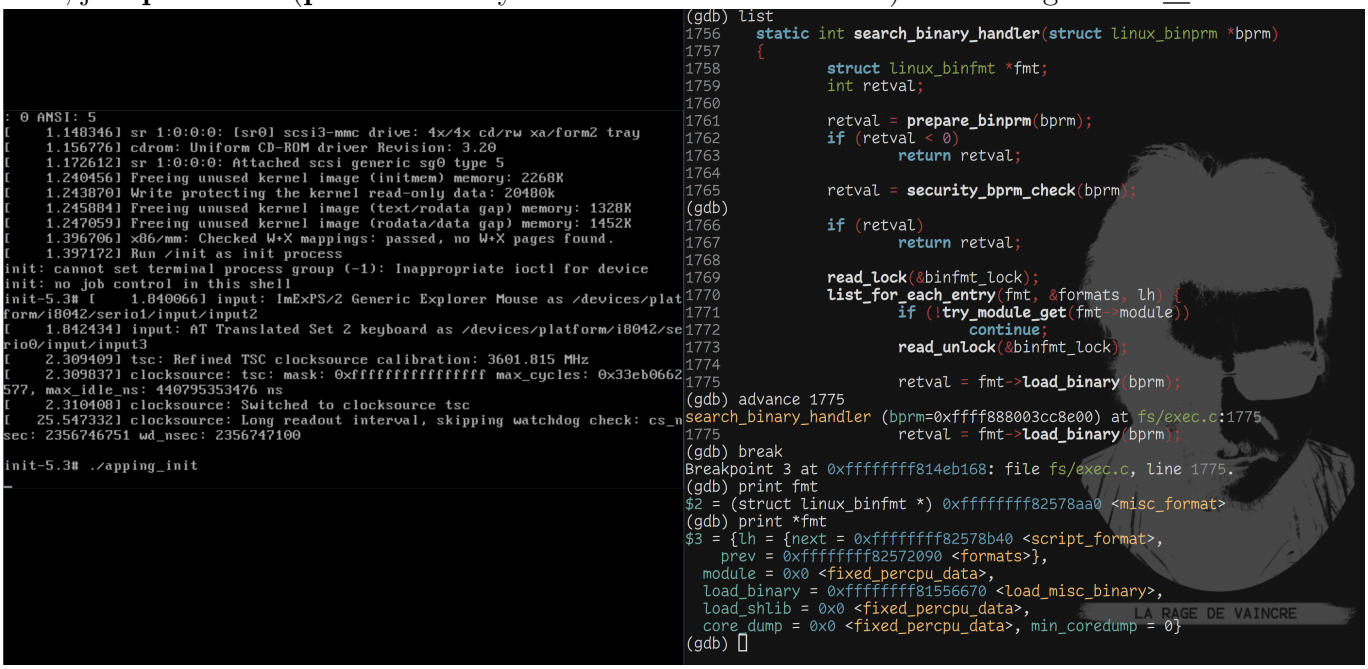
The interesting line in the next instructions is this one:

```
1775      retval = fmt->load_binary(bprm);
```

You can **advance** to this line. **break** on the line, we will go through this line several times.

```
(gdb) advance 1775
(gdb) break
```

Now, just **print fmt** (**print *fmt** if you want to see the structure). You will get **misc_format**[3].



Now that you have break on the line 1775, you can **continue** and re-**print fmt** each time you end-up on your breakpoint.

[3]learn more about it on the next chapter

```
(gdb) advance 1775
search_binary_handler (bprm=0xffff888003cc8e00) at fs/exec.c:1775
1775                    retval = fmt->load_binary(bprm);
(gdb) break
Breakpoint 3 at 0xffffffff814eb168: file fs/exec.c, line 1775.
(gdb) print fmt
$2 = (struct linux_binfmt *) 0xffffffff82578aa0 <misc_format>
(gdb) print *fmt
$3 = {lh = {next = 0xffffffff82578b40 <script_format>,
    prev = 0xffffffff82572090 <formats>},
  module = 0x0 <fixed_percpu_data>,
  load_binary = 0xffffffff81556670 <load_misc_binary>,
  load_shlib = 0x0 <fixed_percpu_data>,
  core_dump = 0x0 <fixed_percpu_data>, min_coredump = 0}
(gdb) continue
Continuing.

Breakpoint 3, search_binary_handler (bprm=0xffff888003cc8e00)
    at fs/exec.c:1775
1775                    retval = fmt->load_binary(bprm);
(gdb) print fmt
$4 = (struct linux_binfmt *) 0xffffffff82578b40 <script_format>
(gdb) continue
Continuing.

Breakpoint 3, search_binary_handler (bprm=0xffff888003cc8e00)
    at fs/exec.c:1775
1775                    retval = fmt->load_binary(bprm);
(gdb) print fmt
$5 = (struct linux_binfmt *) 0xffffffff82578ba0 <elf_format>
(gdb)
```

As you can see, the first, searched handler is the **script_format**. It will search for the "**#!**" hard-coded value to interpret it with the program placed after. Then it will search for the **elf_format**. I am sure you know this one :). The file begins with "**0x7fELF**".

continue the execution, the string **APPING** should appear on the console, then you will get the hands on **bash** again.

```
init-5.3# ./apping_init
APPING
[   30.318085] apping_init (42) used greatest stack depth: 14176 bytes left
init-5.3# _
```

```
(gdb) print fmt
$4 = (struct linux_binfmt *) 0xffffffff82578b40 <script_format>
(gdb) continue
Continuing.

Breakpoint 3, search_binary_handler (bprm=0xffff888003cc8e00)
    at fs/exec.c:1775
1775                    retval = fmt->load_binary(bprm);
(gdb) print fmt
$5 = (struct linux_binfmt *) 0xffffffff82578ba0 <elf_format>
(gdb) continue
Continuing.
```

You can replay this scenario, **breaking**, **backtracing**, **listing**, **advancing** and **finishing**[4] anytime you want to understand better the path for an execution. I cannot explain it entirely here, it would take too much time for you to read. Make it your own journey.

[4] the **finish** command is used to continue the execution of the current function until the end where gdb break

5.2.1 Misc format

Maybe the **misc format** is still mysterious to you. You will understand it better here. You are going to make a change in your system: execute any Python script without explicitly specifying the interpreter.

Execute this line:

```
$ echo ':Python:E::py::/usr/bin/python3:' | sudo tee /proc/sys/fs/binfmt_misc/register
$
```

The structure is the following: We register a Python format based on the Extension using the /usr/bin/python3 interpreter.

Of course, put the path to your Python interpreter if it is located somewhere else.

From now on, you can execute a Python script without calling or writing the interpreter. The kernel will load it because it will recognize first the extension of the file and then call the interpreter.

```
$ chmod +x apping.py
$ cat apping.py
print("I am an APPING Python script")
$ ./apping.py
I am an APPING Python script
$
```

You can check the registration of your new format this way (since we used **Python** in the data, the filename is **Python**):

```
$ cat /proc/sys/fs/binfmt_misc/Python
enabled
interpreter /usr/bin/python3
flags:
extension .py
$
```

If you want to get rid of this, write **-1** in the Python file:

```
$ echo -1 | sudo tee /proc/sys/fs/binfmt_misc/Python
$
```

You can learn more at this page of the documentation: https://docs.kernel.org/admin-guide/binfmt-misc.html. You will have more explanation on other fields you can use to register a new format.

Also on the manpage: `man 5 binfmt.d`

6 Write

6.1 Goal

The goal of this chapter is to see how Linux prints data on stdout. You just debugged the loading of a binary in the kernel, now let's debug a binary loaded in the kernel.

6.2 The initrd for write

To ease the debug, you will make a new initrd. Just take **apping_init** and make it the initrd.

```
$ mv apping_init init
$ echo init | cpio -o -H newc > initrd.img
```

6.3 Rewrite write

First of all, how to break on write? Let's dive into Linux source code. Open the file **./fs/read_write.c** and search these lines:

```
ssize_t ksys_write(unsigned int fd, const char __user *buf, size_t count)
{
  CLASS(fd_pos, f)(fd);
  ssize_t ret = -EBADF;

  if (!fd_empty(f)) {
    loff_t pos, *ppos = file_ppos(fd_file(f));
    if (ppos) {
      pos = *ppos;
      ppos = &pos;
    }
    ret = vfs_write(fd_file(f), buf, count, ppos);
    if (ret >= 0 && ppos)
      fd_file(f)->f_pos = pos;
  }

  return ret;
}

SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf,
    size_t, count)
{
  return ksys_write(fd, buf, count);
}
```

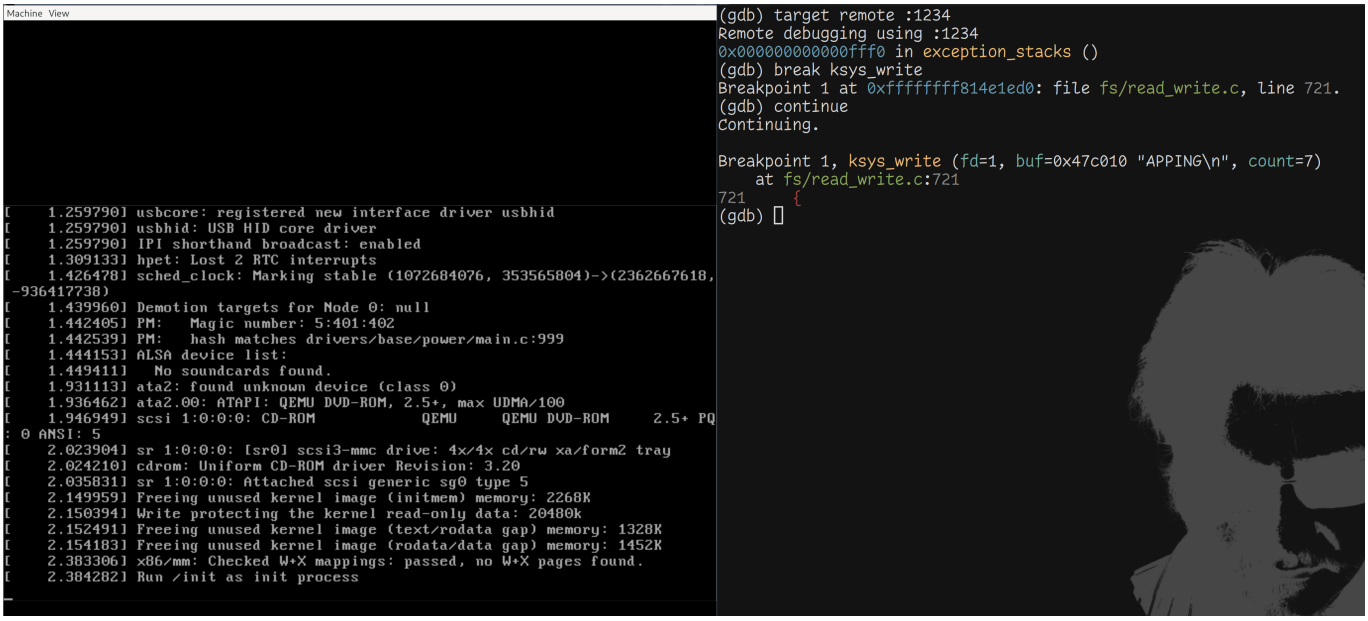
The syscall **write(2)** is executing **ksys_write**.

6.4 Debugging write

Let's break on **ksys_write**.

Because the command line to boot is pretty long, I will only use the name of the bzImage and initrd.img instead of the whole path.

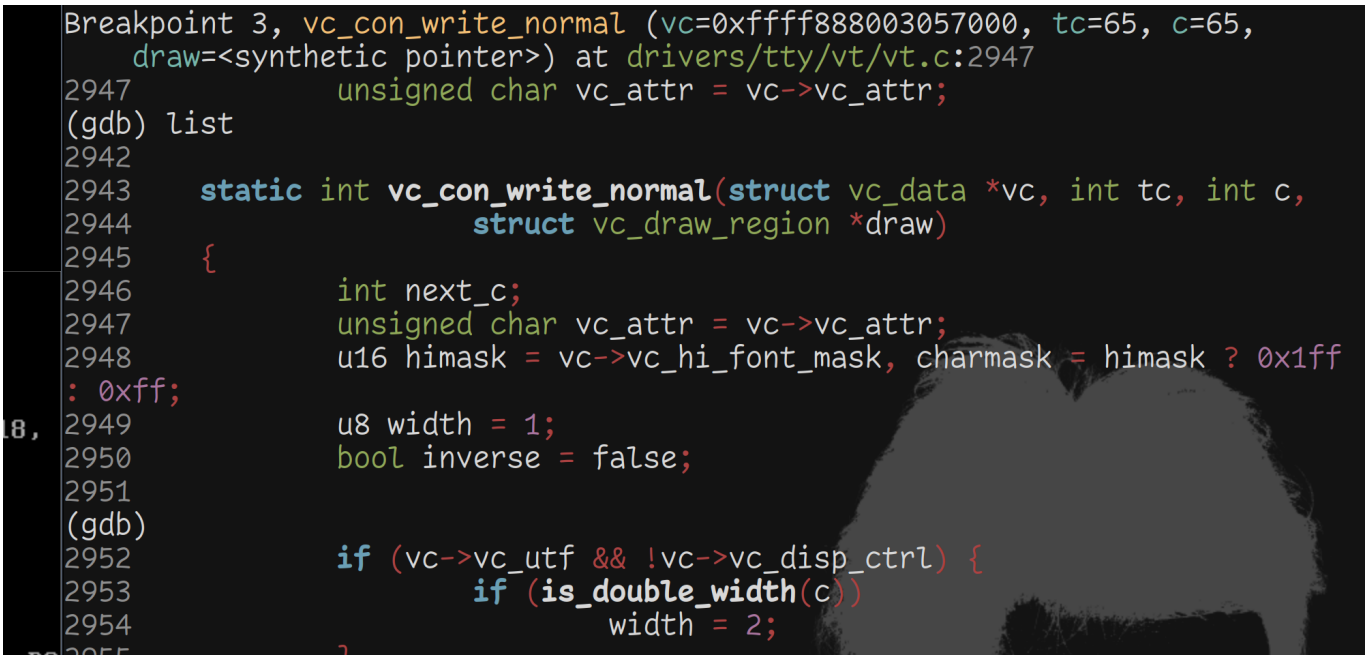
```
$ qemu-system-x86_64 -kernel bzImage -initrd initrd.img -append "nokaslr init=/init" -s -S &
$ gdb vmlinux
(gdb) target remote :1234
(gdb) break ksys_write
(gdb) continue
(...)
(gdb) # Breakpoint reached
```

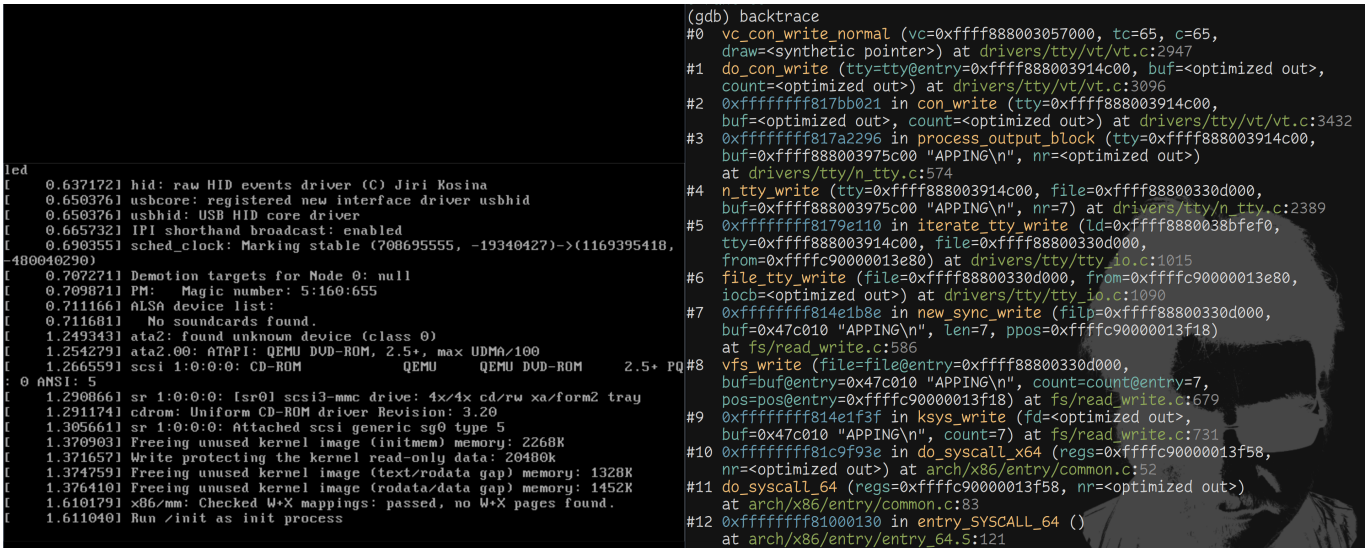


Because I do not want to make a full documentation about the source code of **write(2)** (just read the source by yourself), you will break directly to the interesting part. But, remember, you can always **backtrace** at any moment to see from where you come.

break on **vc__con__write__normal**. It is the final function before printing a single letter. Then, continue the execution.

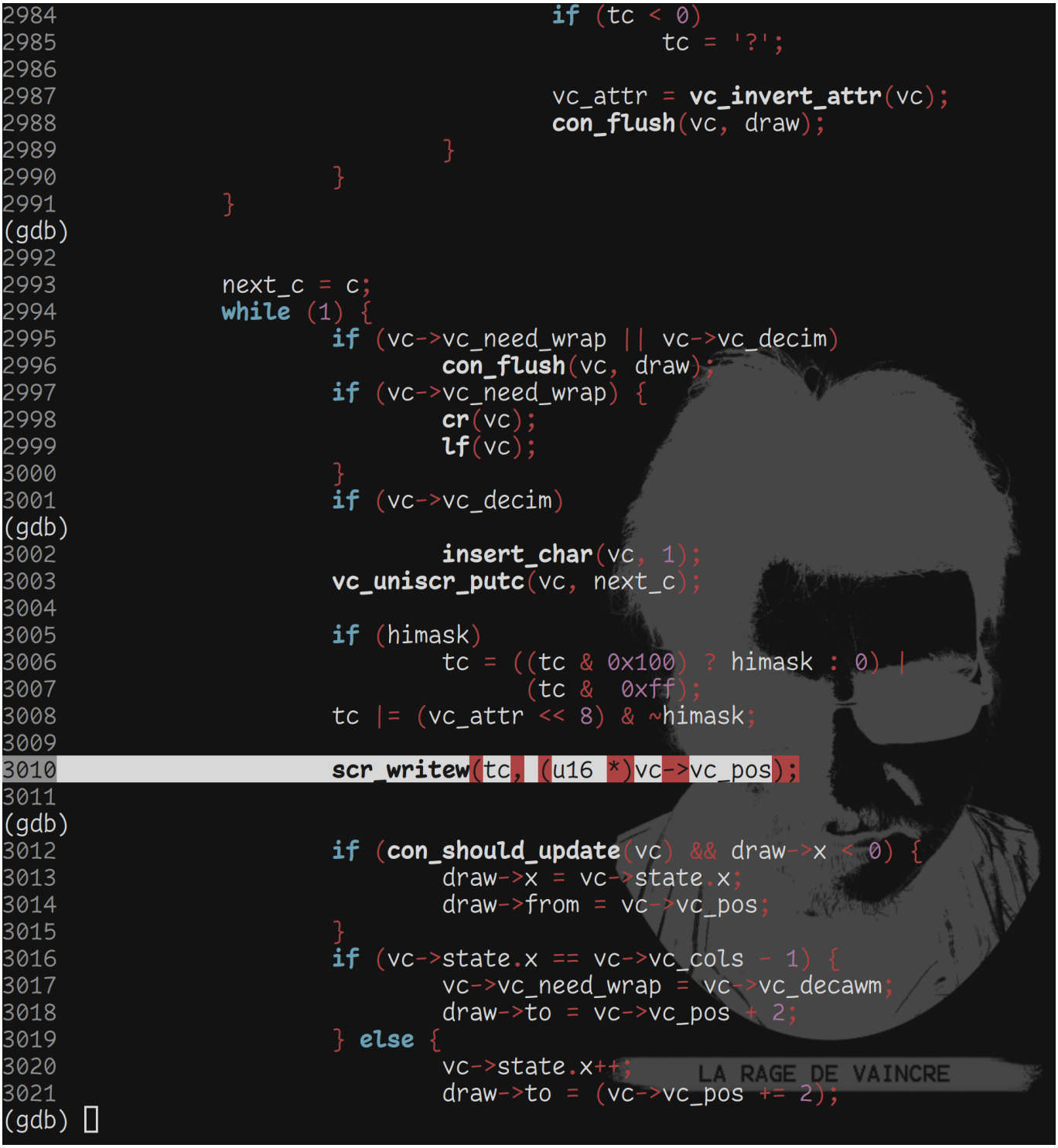
```
(gdb) break vc_con_write_normal
(gdb) continue
(...)
(gdb) # Breakpoint reached
(gdb) backtrace
(...)
```





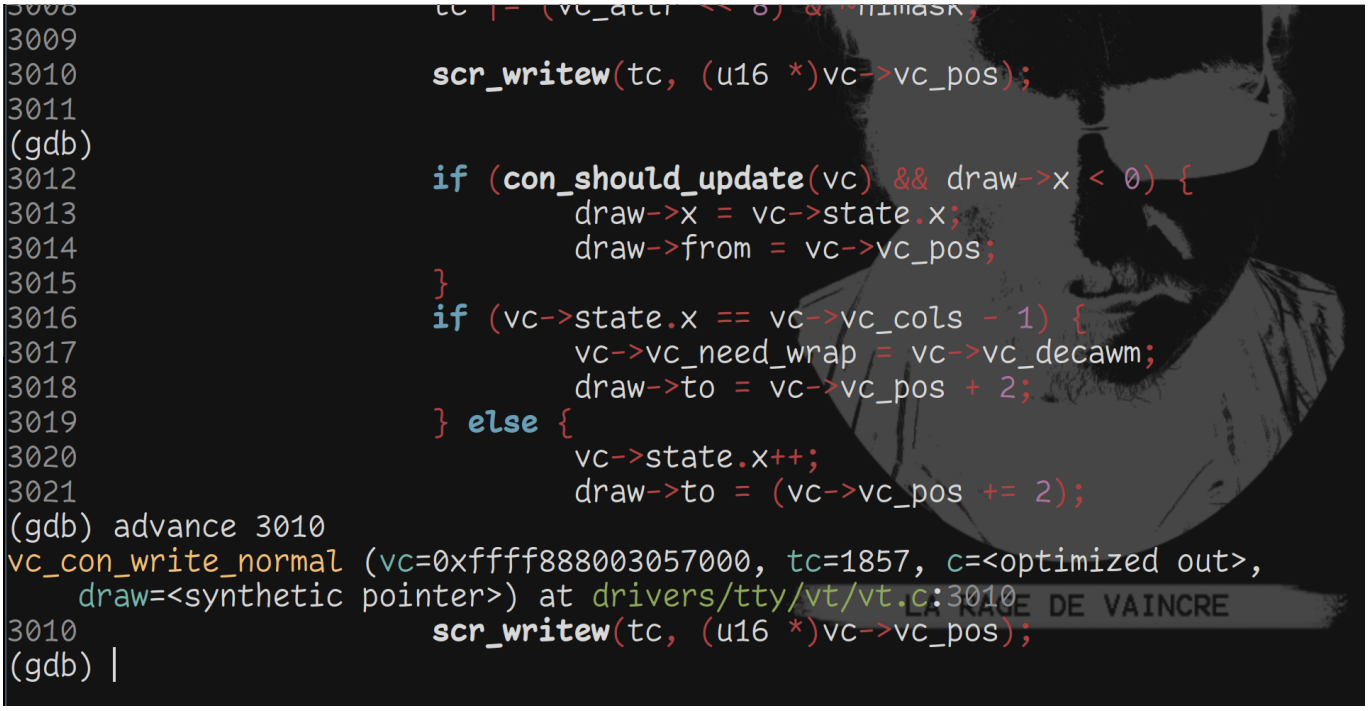
list the following instructions. Can you spot this line?

```
scr_writew(tc, (u16 *)vc->vc_pos);
```



scr_writew is a macro, not a function. You cannot break on it. But you can still **advance** to the line number of the instruction.

```
(gdb) advance 3010
```



If you follow the backtrace, you will understand that to write data on the terminal and reach **scr_writew**, the call from **write** will iterate on the data to write on the screen byte by byte.

To continue, we need to dive deeper. What is lower than C? Assembly! For convenience, you will use Intel syntax. Execute on gdb:

```
(gdb) set disassembly-flavor intel
(gdb) x/i $rip
=> 0xffffffff817a6a24 <do_con_write+1236>:  mov    r10d,DWORD PTR [rsp+0x8]
(gdb) x/i $rip # to print more instructions
=> 0xffffffff817a6a24 <do_con_write+1236>:  mov    r10d,DWORD PTR [rsp+0x8]
   0xffffffff817a6a29 <do_con_write+1241>:  mov    rax,QWORD PTR [r14+0x200]
   0xffffffff817a6a30 <do_con_write+1248>:  mov    rdi,r14
   0xffffffff817a6a33 <do_con_write+1251>:  or     r10d,r13d
   0xffffffff817a6a36 <do_con_write+1254>:  mov    WORD PTR [rax],r10w
   0xffffffff817a6a3a <do_con_write+1258>:  call   0xffffffff817a22c0 <con_is_visible>
   0xffffffff817a6a3f <do_con_write+1263>:  mov    rdx,QWORD PTR [r14+0x200]
   0xffffffff817a6a46 <do_con_write+1270>:  test   al,al
(gdb)
```

```
(gdb) x/i $rip
=> 0xffffffff817b907b <do_con_write+1323>:      mov    0x10(%rsp),%r10d
(gdb) set disassembly-flavor intel
(gdb) x/i $rip
=> 0xffffffff817b907b <do_con_write+1323>:
    mov    r10d,DWORD PTR [rsp+0x10]
(gdb) x/8i $rip
=> 0xffffffff817b907b <do_con_write+1323>:
    mov    r10d,DWORD PTR [rsp+0x10]
   0xffffffff817b9080 <do_con_write+1328>:
    mov    rax,QWORD PTR [r14+0x200]
   0xffffffff817b9087 <do_con_write+1335>:      mov    rdi,r14
   0xffffffff817b908a <do_con_write+1338>:      or     r10d,r15d
   0xffffffff817b908d <do_con_write+1341>:      mov    WORD PTR [rax],r10w
   0xffffffff817b9091 <do_con_write+1345>:
    call   0xffffffff817b4760 <con_is_visible>
   0xffffffff817b9096 <do_con_write+1350>:
    mov    rdx,QWORD PTR [r14+0x200]
   0xffffffff817b909d <do_con_write+1357>:      test   al,al
(gdb)
```

Among all these lines, only one is modifying the memory.

```
mov WORD PTR [rax], r10w
```

It will put the value inside **r10w** at the adress pointed by **rax**. advance to this line and break on it.

```
(gdb) advance *do_con_write+1341
(gdb) break
```

```
   0xffffffff817b908d <do_con_write+1341>:      mov    WORD PTR [rax],r10w
   0xffffffff817b9091 <do_con_write+1345>:
    call   0xffffffff817b4760 <con_is_visible>
   0xffffffff817b9096 <do_con_write+1350>:
    mov    rdx,QWORD PTR [r14+0x200]
   0xffffffff817b909d <do_con_write+1357>:      test   al,al
(gdb) advance *do_con_write+1341
0xffffffff817b908d in vc_con_write_normal (vc=0xffff888003057000, tc=1857,
    c=<optimized out>, draw=<synthetic pointer>)
    at drivers/tty/vt/vt.c:3010
3010                    scr_writew(tc, (u16 *)vc->vc_pos);
(gdb) break
Breakpoint 4 at 0xffffffff817b908d: file drivers/tty/vt/vt.c, line 3010.
(gdb)
```

If you look at the function **do__con__write**, you can see some parameters, including a buffer containing the string:

`"APPING\n"`

```
Breakpoint 1, do_con_write (tty=tty@entry=0xffff888003840800, buf=0xffff888003843c00 "APPING\n", count=6)
    at drivers/tty/vt/vt.c:3043
```

The first character is 'A'. What do we have in the **r10w** register?

```
(gdb) print/x $r10w
$2 = 0x741
```

What is this value? Open the **ascii(7)** manpage and search the value 0x41. It's an A!

Oct	Dec	Hex	Char
100	64	40	@
101	65	41	A
102	66	42	B
103	67	43	C
104	68	44	D

You can also use Python to get this information.

```
$ python
(python) chr(0x41)
'A'
(python) exit()
$
```

```
Python 3.13.3 (main, Apr  9 2025, 07:44:25) [GCC 14.2.1 20250207] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> chr(0x41)
'A'
>>>
```

But what is the first value? 0x7?

Time for some OSDEV. 8)

https://wiki.osdev.org/Printing_To_Screen
https://wiki.osdev.org/Text_UI

Basics

Assuming that you are in protected mode and not using the BIOS to write text to screen, you will have write directly to "video" memory.

This is quite easy. The text screen video memory for colour monitors resides at `0xB8000`, and for monochrome monitors it is at address `0xB0000` (see Detecting Colour and Monochrome Monitors for more information).

Text mode memory takes two bytes for every "character" on screen. One is the *ASCII code* byte, the other the *attribute* byte. so the text "HeLlo" would be stored as:

```
0x000b8000: 'H', colour_for_H
0x000b8002: 'e', colour_for_e
0x000b8004: 'L', colour_for_L
0x000b8006: 'l', colour_for_l
0x000b8008: 'o', colour_for_o
```

The *attribute* byte carries the *foreground colour* in its lowest 4 bits and the *background color* in its highest 3 bits. The interpretation of bit #7 depends on how you (or the BIOS) configured the hardware (see VGA Resources for additional info).

For instance, using `0x00` as attribute byte means black-on-black (you'll see nothing). `0x07` is lightgrey-on-black (DOS default), `0x1F` is white-on-blue (Win9x's blue-screen-of-death), `0x2a` is for green-monochrome nostalgics.

For colour video cards, you have 32 KB of text video memory to use. Since 80x25 mode does not use all 32 KB (80 x 25 x 2 = 4,000 bytes per screen), you have 8 display pages to use.

When you print to any other page than 0, it will *not* appear on screen until that page is *enabled* or *copied* into the page 0 memory space.

Colours

Each character has a colour byte. This colour byte is split up in forecolour and backcolour.

The layout of the byte, using the standard colour palette:

```
Bit 76543210
    ||||||||
    |||||^^^-fore colour
    ||||^----fore colour bright bit
    |^^^-----back colour
    ^--------back colour bright bit OR enables blinking Text
```

Its easy to write to BL, the Colour Nibbles(4Bit), in a Hex Value.
For Example:

```
0x01 sets the background to black and the fore colour to blue

0x10 sets the background to blue and the fore colour to black

0x11 sets both to blue.
```

The default display colours set by the BIOS upon booting are 0x0F: 0 (black) for the background and 7 (White) + 8 (Bright) for the foreground.

In text mode 0, the following standard colour palette is available for use. You can change this palette with VGA commands.

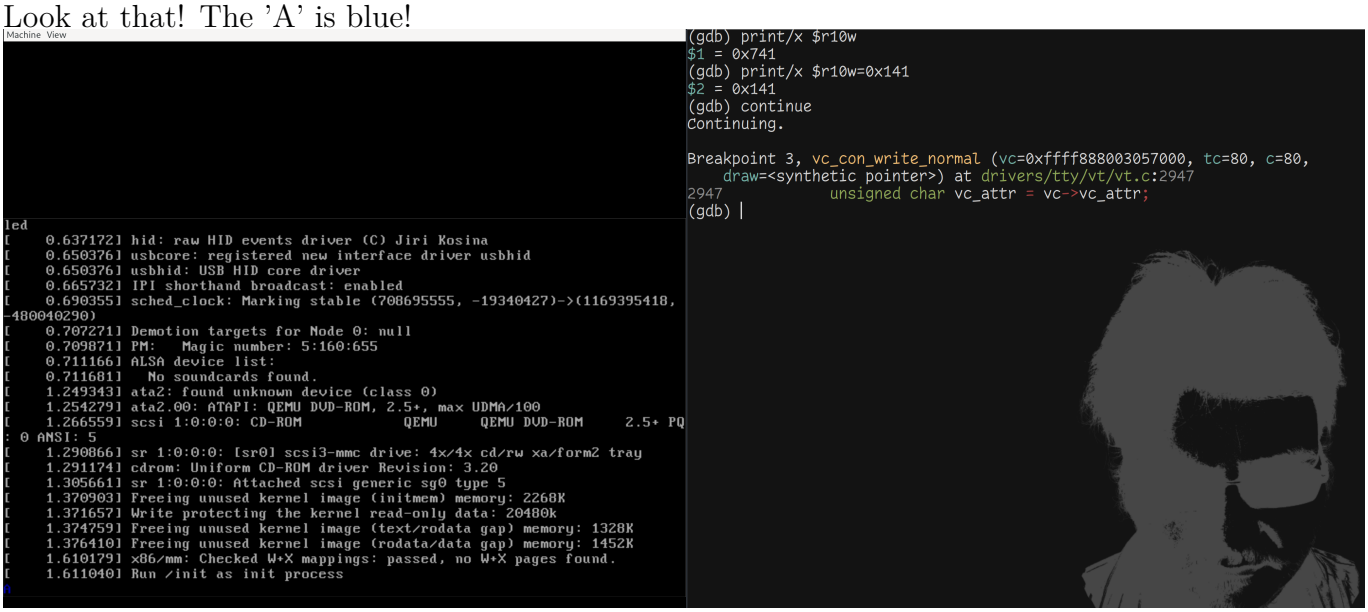
Number	Colour	Name	Number + bright bit	bright Colour	Name
0		Black	0+8=8		Dark Gray
1		Blue	1+8=9		Light Blue
2		Green	2+8=A		Light Green
3		Cyan	3+8=B		Light Cyan
4		Red	4+8=C		Light Red
5		Magenta	5+8=D		Light Magenta
6		Brown	6+8=E		Yellow
7		Light Gray	7+8=F		White

We are using two bytes for a letter. One for the letter itself, the other one for the colour. We can see that the colour 0x7 is light gray. It is a match. The text on the screen is not really white, that would be too constrasted and hurtful for the eyes. And the byte for the colour is dispatched into two different layers. Background and foreground. For the 0x7 colour, we can read the byte as follow: 0x07. So the back colour is black and the fore colour is light gray.

Let's get something fancier!

```
(gdb) print/x $r10w
$2 = 0x741 # 'A'
(gdb) print/x $r10w=0x141
$3 = 0x141 # fore blue 'A'
(gdb) continue
(gdb) # Breakpoint reached
```

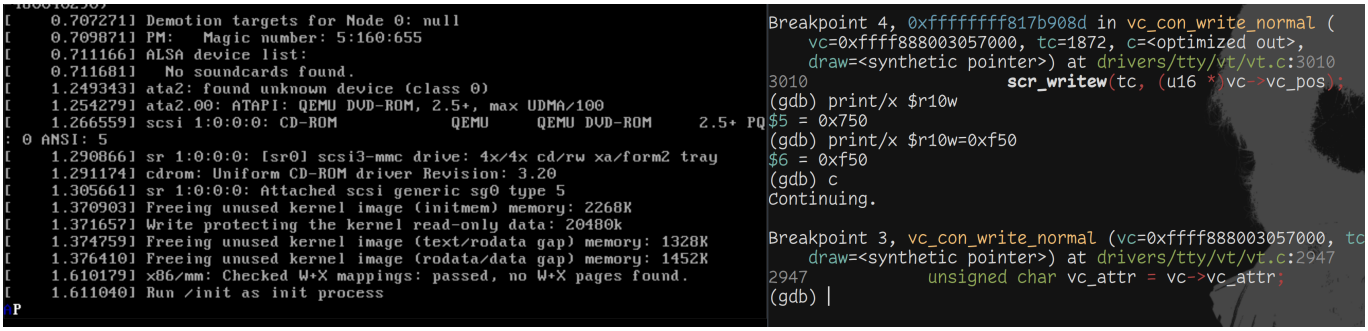
Look at that! The 'A' is blue!



Let's continue!

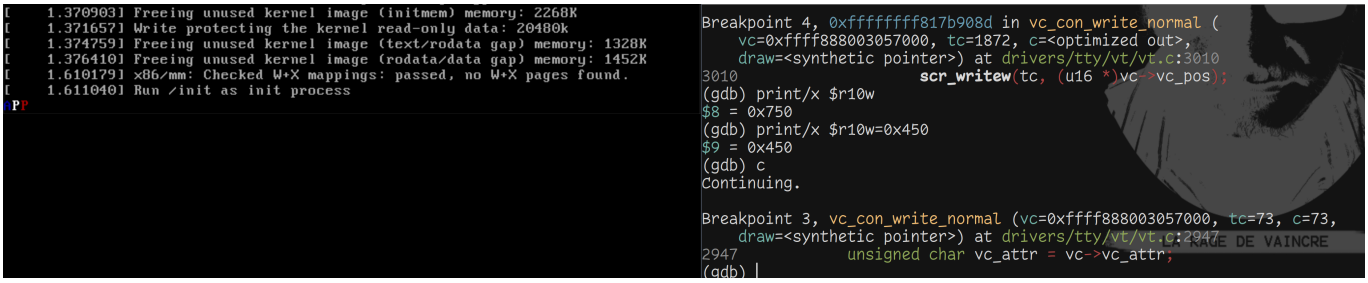
The next letter should be 'P'. Can you make a fore white P on your own?

```
(gdb) print/x $r10w
$3 = 0x750 # 'P'
(gdb) print/x $r10w=0xF50
$4 = 0xF50 # fore white 'P'
(gdb) continue
(gdb) # Breakpoint reached
```



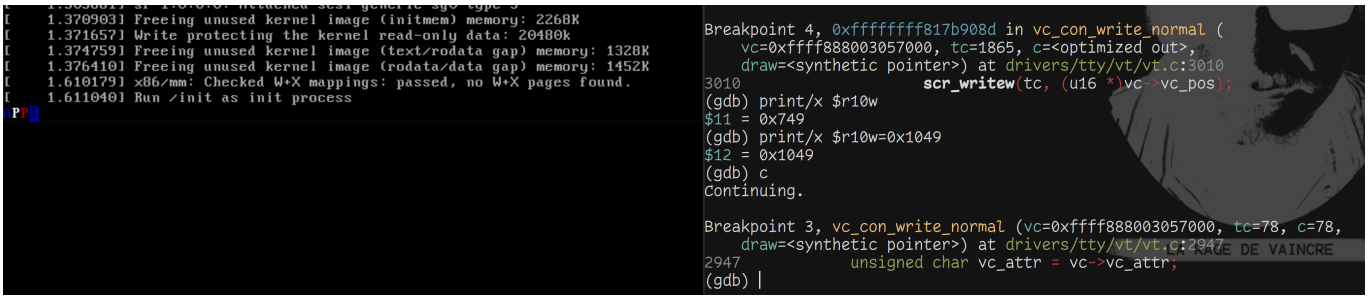
The next letter should be 'P'. Can you make a fore red P on your own?

```
(gdb) print/x $r10w
$5 = 0x750 # 'P'
(gdb) print/x $r10w=0x450
$6 = 0x450 # fore red 'P'
(gdb) continue
(gdb) # Breakpoint reached
```



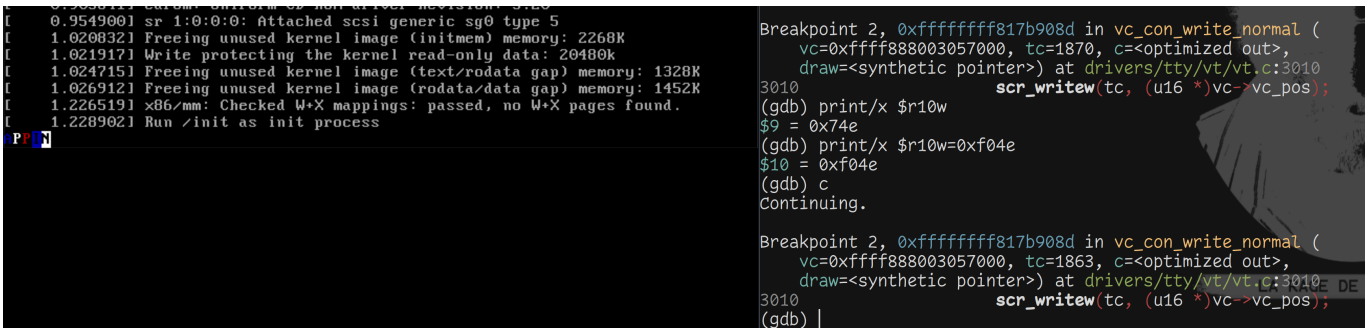
The next letter should be 'I'. Can you make a back blue I on your own?

```
(gdb) print/x $r10w
$7 = 0x749 # 'I'
(gdb) print/x $r10w=0x1049
$8 = 0x1049 # back blue 'I'
(gdb) continue
(gdb) # Breakpoint reached
```



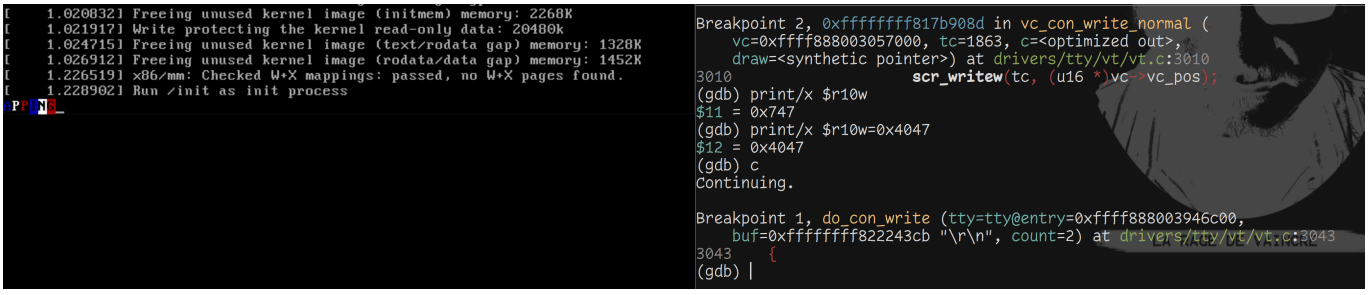
The next letter should be 'N'. Can you make a back white N on your own?

```
(gdb) print/x $r10w
$9 = 0x74e # 'N'
(gdb) print/x $r10w=0xF04e
$10 = 0xF04E # back white 'N'
(gdb) continue
(gdb) # Breakpoint reached
```



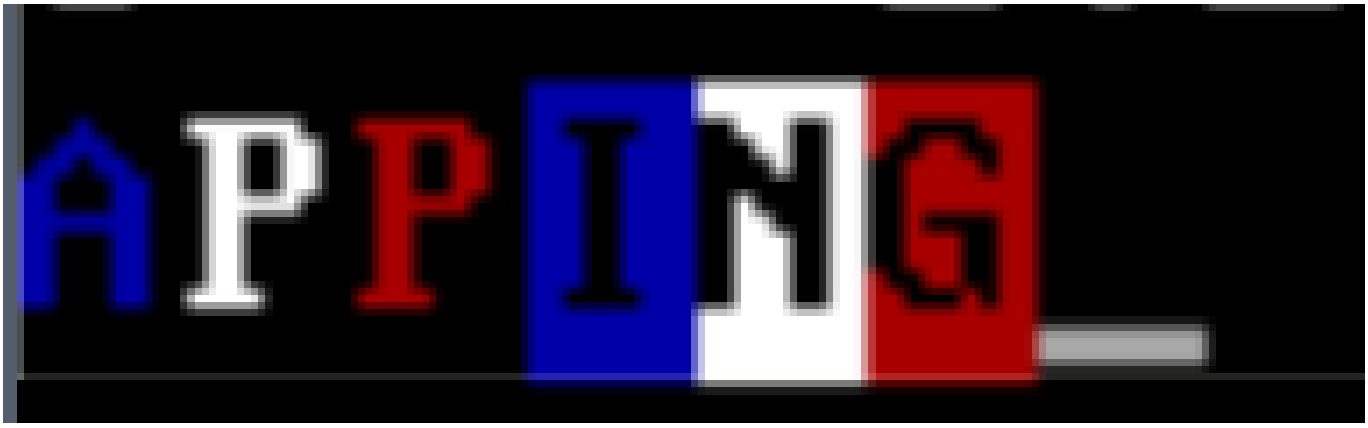
The next and last letter should be 'G'. Can you make a back red G on your own?

```
(gdb) print/x $r10w
$11 = 0x747 # 'G'
(gdb) print/x $r10w=0x4047
$12 = 0x4047 # back red 'G'
(gdb) continue
(gdb) # Breakpoint reached
```



What a beautiful APPING we have here!

The last letter will be the return line, and after printing it, the code will continue until the kernel panic.


APING