

# SYS2

*Linux kernel*

20xx

---

Version 1



Jules AUBERT <[jules1.aubert@epita.fr](mailto:jules1.aubert@epita.fr)>

---

# Contents

<b>1</b>	<b>Introduction</b>	<b>II</b>
1.1	Goal	II
1.2	Get the code	II
1.3	Documentation	II
1.3.1	Online	II
1.3.2	Offline	II
1.4	Virtualization	III
1.5	Environment	III
<b>2</b>	<b>Compile</b>	<b>IV</b>
2.1	Goal	IV
2.2	Config	IV
2.2.1	Ubuntu config	VIII
2.3	gdb scripts	IX
2.4	Compile	IX
2.5	The kernels	IX
<b>3</b>	<b>Boot</b>	<b>X</b>
3.1	Goal	X
3.2	QEMU	X
3.3	gdb server	XI
<b>4</b>	<b>Init</b>	<b>XV</b>
4.1	Goal	XV
4.2	initrd	XV
4.2.1	cpio	XV
4.3	Creating the initrd	XV
4.4	APPING init	XVII
4.4.1	Constructor and destructor in C	XVIII
4.5	Booting on APPING init	XIX
<b>5</b>	<b>Exec</b>	<b>XXI</b>
5.1	Goa	XXI
5.2	Execve	XXI
5.2.1	Misc format	XXVI
<b>6</b>	<b>Write</b>	<b>XXVII</b>
6.1	Goal	XXVII
6.2	The initrd for write	XXVII
6.3	Rewrite write	XXVII
6.4	Debugging write	XXVII

# 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 entirety 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<sup>1</sup> uses NixOS for the environment. As for myself, I am using the former distro on the PIE, Arch<sup>2</sup> Linux. Both are great to use, you should consider testing other Linux distros on your free time.

---

<sup>1</sup>Parc Informatique de l'EPITA

<sup>2</sup>(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!

```
config - Linux/x86 6.14.0 Kernel Configuration
Linux/x86 6.14.0 Kernel Configuration
Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenus ----). Highlighted letters are
hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </>
for Search. Legend: [*] built-in [ ] excluded <M> module < > module capable

[*] General setup --->
  [*] 64-bit kernel
  Processor type and features --->
  [*] Mitigations for CPU vulnerabilities --->
  Power management and ACPI options --->
  Bus options (PCI etc.) --->
  Binary Emulations --->
  [*] Virtualization --->
  General architecture-dependent options --->
  [*] Enable loadable module support --->
  -* Enable the block layer --->
  Executable file formats --->
  Memory Management options --->
  [*] Networking support --->
  Device Drivers --->
  File systems --->
  Security options --->
  -* Cryptographic API --->
  Library routines --->
  Kernel hacking --->

<Select> <Exit> <Help> <Save> <Load>
```

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.

```

config - Linux/x86 6.14.0 Kernel Configuration
Linux/x86 6.14.0 Kernel Configuration
Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenu ----). Highlighted letters are
hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </>
for Search. Legend: [*] built-in [ ] excluded <M> module < > module capable

  General setup --->
[*] 64-bit kernel
  Processor type and features --->
[*] Mitigations for CPU vulnerabilities --->
  Power management and ACPI options --->
  Bus options (PCI etc.) --->
  Binary Emulations --->
[ ] Virtualization ----
  General architecture-dependent options --->
[ ] Enable loadable module support ----
-* Enable the block layer --->
  Executable file formats --->
  Memory Management options --->
[ ] Networking support ----
  Device Drivers --->
  File systems --->
  Security options --->
-* Cryptographic API --->
  Library routines --->
  Kernel hacking --->

<Select> < Exit > < Help > < Save > < Load >

```

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.

```

config - Linux/x86 6.14.0 Kernel Configuration
Processor type and features
Processor type and features
Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenu ----). Highlighted letters are
hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </>
for Search. Legend: [*] built-in [ ] excluded <M> module < > module capable

^(-)
[*] Set the default setting of memory_corruption_check
-* MTRR (Memory Type Range Register) support
[ ] MTRR cleanup support
[*] Indirect Branch Tracking
[*] Memory Protection Keys
  TSX enable mode (off) --->
[ ] X86 userspace shadow stack
[*] EFI runtime service support
[*] EFI stub support
[*] EFI handover protocol (DEPRECATED)
[*] EFI mixed-mode support
  Timer frequency (1000 HZ) --->
(0x1000000) Physical address where the kernel is loaded
-* Build a relocatable kernel
[ ] Randomize the address of the kernel image (KASLR)
(0x200000) Alignment value to which kernel should be aligned
[ ] Disable the 32-bit vDSO (needed for glibc 2.3.3)
  vsyscall table for legacy applications (Emulate execution only) --->
[ ] Built-in kernel command line
[ ] Enforce strict size checking for sigaltstack
[*] Split Lock Detect and Bus Lock Detect support

<Select> < Exit > < Help > < Save > < Load >

```

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

```

config - Linux/x86_64 6.14.0 Kernel Configuration
* Kernel hacking
Kernel hacking
Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenus ----). Highlighted letters are
hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </>
for Search. Legend: [*] built-in [ ] excluded <M> module < > module capable

| printk and dmesg options --->
[*] Kernel debugging
[*] Miscellaneous debug code
Compile-time checks and compiler options --->
Generic Kernel Debugging Instruments --->
Networking Debugging ----
Memory Debugging --->
[ ] Debug shared IRQ handlers
Debug Oops, Lockups and Hangs --->
Scheduler Debugging --->
[ ] Debug preemptible kernel
Lock Debugging (spinlocks, mutexes, etc...) --->
[ ] Debugging for CPUs failing to respond to backtrace requests
[ ] Debug IRQ flag manipulation
-* Stack backtrace support
[ ] Warn for all uses of unseeded randomness
[ ] kobject debugging
Debug kernel data structures --->
RCU Debugging --->
[ ] Force round-robin CPU selection for unbound work items
[ ] Enable CPU hotplug state control
v(+)

<Select> < Exit > < Help > < Save > < Load >

```

## Get into Compile-time checks and compiler options

```

config - Linux/x86_64 6.14.0 Kernel Configuration
* Kernel hacking > Compile-time checks and compiler options
Compile-time checks and compiler options
Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenus ----). Highlighted letters are
hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </>
for Search. Legend: [*] built-in [ ] excluded <M> module < > module capable

| Debug information (Disable debug information) --->
(2048) Warn for stack frames larger than
[ ] Strip assembler-generated symbols during link
[ ] Generate readable assembler code
[ ] Install uapi headers to usr/include
[ ] Enable full Section mismatch analysis
[*] Make section mismatch errors non-fatal
[ ] Force weak per-cpu definitions

<Select> < Exit > < Help > < Save > < Load >

```

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

```

config - Linux/x86 6.14.0 Kernel Configuration
> kernel hacking > Compile-time checks and compiler options

```

**Debug information**

Use the arrow keys to navigate this window or press the hotkey of the item you wish to select followed by the <SPACE BAR>. Press <?> for additional information about this

- ( ) Disable debug information
- (X)** **Rely on the toolchain's implicit default DWARF version**
- ( ) Generate DWARF Version 4 debuginfo
- ( ) Generate DWARF Version 5 debuginfo

**<Select>**      < Help >

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

```

config - Linux/x86 6.14.0 Kernel Configuration
> kernel hacking > Compile-time checks and compiler options

```

**Compile-time checks and compiler options**

Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenus ----). Highlighted letters are hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [\*] built-in [ ] excluded <M> module < > module capable

- Debug information (Rely on the toolchain's implicit default DWARF version) --->
- [ ] Reduce debugging information (NEW)
- Compressed Debug information (Don't compress debug information) --->
- [ ] Produce split debuginfo in .dwo files (NEW)
- [\*] Provide GDB scripts for kernel debugging**
- (2048) Warn for stack frames larger than
- [ ] Strip assembler-generated symbols during link
- [ ] Generate readable assembler code
- [ ] Install uapi headers to usr/include
- [ ] Enable full Section mismatch analysis
- [\*] Make section mismatch errors non-fatal
- [ ] Force weak per-cpu definitions

**<Select>**      < Exit >      < Help >      < Save >      < Load >

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

```

.config - Linux/x86 6.14.0 Kernel Configuration
* Kernel hacking > Generic Kernel Debugging Instruments
                                     Generic Kernel Debugging Instruments
Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty submenus ----). Highlighted letters are
hotkeys. Pressing <Y> includes, <N> excludes, <M> modularizes features. Press <Esc><Esc> to exit, <?> for Help, </>
for Search. Legend: [*] built-in [ ] excluded <M> module < > module capable

--* Magic SysRq key
(0x1) Enable magic SysRq key functions by default
[*] Enable magic SysRq key over serial
( ) char sequence that enables magic SysRq over serial
--* Debug Filesystem
    Debugfs default access (Access normal) --->
[*] KGDB: kernel debugger --->
[ ] Undefined behaviour sanity checker ----
[ ] KCSAN: dynamic data race detector ----

<Select> < Exit > < Help > < Save > < Load >

```

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

```

.config - Linux/x86 6.14.0 Kernel Configuration

Do you wish to save your new configuration?
(Press <ESC><ESC> to continue kernel configuration.)

< Yes > < No >

```

```

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 &
$
```

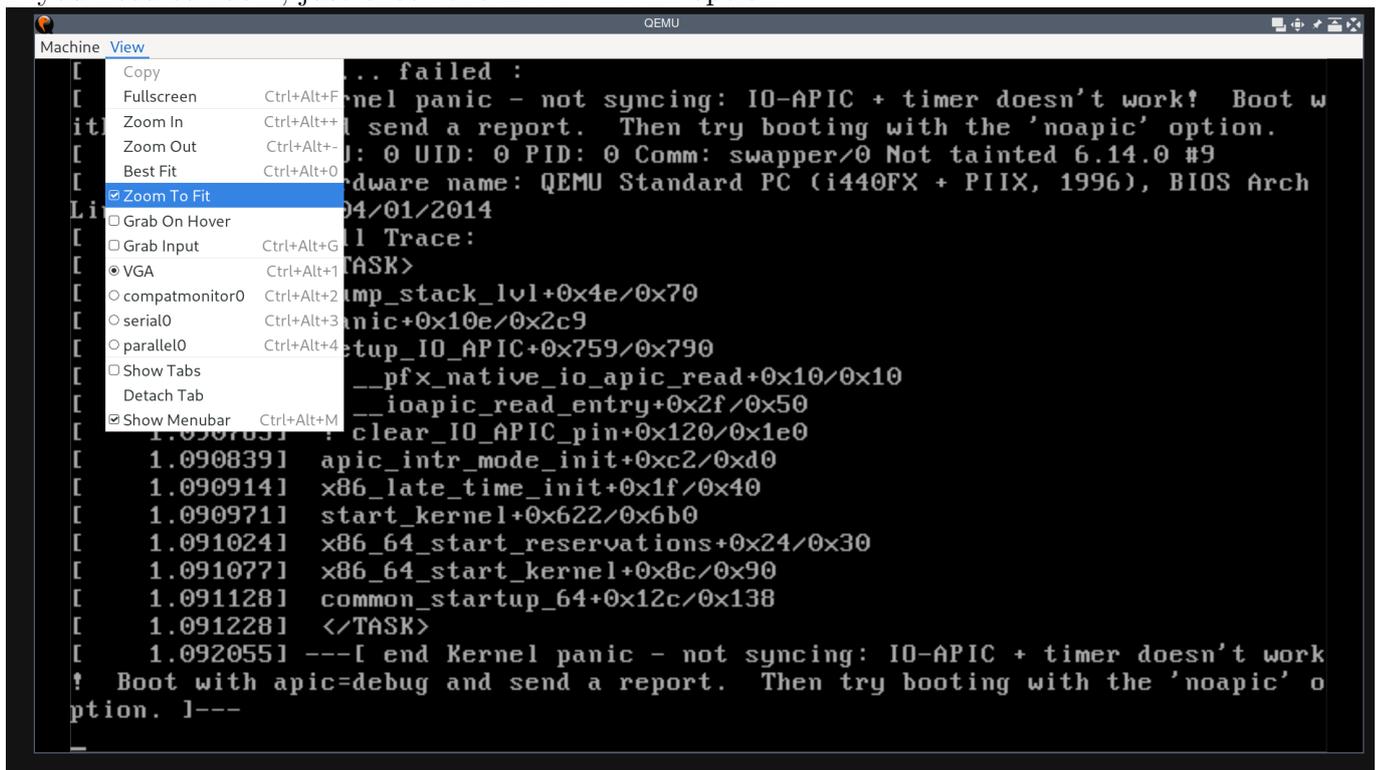
If QEMU shows any problem, it might be because you are on Ubuntu and/or have generated a big Linux kernel. Replace the command with this:

```
$ 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.

```

Machine View
[ 1.716335] msdos
[ 1.716366] iso9660
[ 1.716403]
[ 1.716651] Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(0,0)
[ 1.717278] CPU: 0 UID: 0 PID: 1 Comm: swapper/0 Not tainted 6.14.0 #9
[ 1.717566] Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS Arch Linux 1.16.3-1-1 04/01/2014
[ 1.717798] Call Trace:
[ 1.718557] <TASK>
[ 1.718789] dump_stack_lvl+0x4e/0x70
[ 1.719868] panic+0x10e/0x2c9
[ 1.719940] mount_root_generic+0x1c9/0x270
[ 1.720931] prepare_namespace+0x1e7/0x230
[ 1.721107] kernel_init_freeable+0x200/0x210
[ 1.721184] ? __pfx_kernel_init+0x10/0x10
[ 1.721248] kernel_init+0x15/0x130
[ 1.721303] ret_from_fork+0x2f/0x50
[ 1.721356] ? __pfx_kernel_init+0x10/0x10
[ 1.721412] ret_from_fork_asm+0x1a/0x30
[ 1.721519] </TASK>
[ 1.722033] Kernel Offset: disabled
[ 1.722419] ---[ end Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(0,0) ]---

```

### 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
$
```

```
Machine View
[ 0.1509352] clocksource: hpet: mask: 0xffffffff max_cycles: 0xffffffff, max_i
idle_ns: 19112604467 ns
[ 0.1655761] APIC: Switch to symmetric I/O mode setup
[ 0.1726391] .TIMER: vector=0x30 apic1=0 pin1=2 apic2=-1 pin2=-1
[ 0.1826391] clocksource: tsc-early: mask: 0xffffffffffffffff max_cycles: 0x33
ada498457, max_idle_ns: 440795227006 ns
[ 0.1832351] Calibrating delay loop (skipped), value calculated using timer fr
equency... 7196.37 BogoMIPS (lpj=3598189)
[ 0.1883301] Last level iTLB entries: 4KB 512, 2MB 255, 4MB 127
[ 0.1889811] Last level dTLB entries: 4KB 512, 2MB 255, 4MB 127, 16B 0
[ 0.1903231] Spectre V1 : Mitigation: usercopy/swapgs barriers and __user poin
ter sanitization
[ 0.1907841] Spectre V2 : Mitigation: Retpolines
[ 0.1908761] Spectre V2 : Spectre v2 / SpectreRSB mitigation: Filling RSB on c
ontext switch
[ 0.1909461] Spectre V2 : Spectre v2 / SpectreRSB : Filling RSB on UMEXIT
[ 0.1909461] x86/fpu: x87 FPU will use FXSAVE
[ 0.4935941] Freeing SMP alternatives memory: 36K
[ 0.4954311] pid_max: default: 32768 minimum: 301
[ 0.4997201] LSM: initializing lsm=capability
[ 0.5019461] Mount-cache hash table entries: 512 (order: 0, 4096 bytes, linear)
[ 0.5020151] Mountpoint-cache hash table entries: 512 (order: 0, 4096 bytes, l
inear)

gelules@formation[~/Development/Linux]$ gdb vmlinux
GNU gdb (GDB) 16.3
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/lice
nses/gpl.html>
This is free software: you are free to change and redistribute
it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-pc-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at
:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from vmlinux...
(gdb) target remote :1234
Remote debugging using :1234
0x0000000000000000 in exception_stacks ()
(gdb) hbreak kernel_init
Hardware assisted breakpoint 1 at 0xffffffff81cba1e0: file init
/main.c, line 1449.
(gdb) continue
Continuing.

Breakpoint 1, kernel_init (unused=0x0 <fixed_percpu_data>)
  at init/main.c:1449
1449  {
(gdb) |
LA RAGE DE VAINCRE
```

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.

```
(gdb) continue
Continuing.

Breakpoint 1, kernel_init (unused=0x0 <fixed_percpu_data>)
  at init/main.c:1449
1449  {
(gdb) list
1444  {
1445      free_initmem_default(POISON_FREE_INITMEM);
1446  }
1447
1448  static int __ref kernel_init(void *unused)
1449  {
1450      int ret;
1451
1452      /*
1453       * Wait until kthreadd is all set-up.
(gdb) █
LA RAGE DE VAINCRE
```

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.

```
(gdb) next
1455         wait_for_completion(&kthreadd_done);
(gdb)
1457         kernel_init_freeable();
(gdb) s
kernel_init_freeable () at init/main.c:1541
1541         gfp_allowed_mask = __GFP_BITS_MASK;
(gdb) next
1546         set_mems_allowed(node_states[N_MEMORY]);
(gdb)
arch_local_irq_save ()
    at ./arch/x86/include/asm/irqflags.h:124
124         arch_local_irq_disable();
(gdb)
set_mems_allowed (nodemask=...)
    at ./include/linux/cpuset.h:167
167         write_seqcount_begin(&current->mems_allowed_seq
);
(gdb)
set_mems_allowed (nodemask=...)
    at ./arch/x86/include/asm/current.h:47
47         return this_cpu_read_const(const_pcpu_h
ot.current_task);
(gdb)
set_mems_allowed (nodemask=...)
    at ./include/linux/cpuset.h:169
169         write_seqcount_end(&current->mems_allowed_seq);
(gdb)
170         local_irq_restore(flags);
(gdb)
171         task_unlock(current);
(gdb)
kernel_init_freeable () at init/main.c:1548
1548         cad_pid = get_pid(task_pid(current));
(gdb) □
```

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?

```

Machine View
[ 10.2027211] msdos
[ 10.2027561] iso9660
[ 10.2028001] Kernel panic - not syncing: UFS: Unable to mount root fs on unknown-block(0,0)
[ 10.2036981] CPU: 0 UID: 0 PID: 1 Comm: swapper/0 Not tainted 6.14.0 #9
[ 10.2039081] Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS Arch Linux 1.16.3-1-1 04/01/2014
[ 10.2041621] Call Trace:
[ 10.2054581] <TASK>
[ 10.2057501] dump_stack_lvl+0x4e/0x70
[ 10.2064361] panic+0x10e/0x2c9
[ 10.2065561] mount_root_generic+0x1c9/0x270
[ 10.2076031] prepare_namespace+0x1e7/0x230
[ 10.2076931] kernel_init_freeable+0x200/0x210
[ 10.2077731] ? __pfx_kernel_init+0x10/0x10
[ 10.2078861] kernel_init+0x15/0x130
[ 10.2079441] ret_from_fork+0x2f/0x50
[ 10.2080001] ? __pfx_kernel_init+0x10/0x10
[ 10.2080591] ret_from_fork_asm+0x1a/0x30
[ 10.2081721] </TASK>
[ 10.2088391] Kernel Offset: disabled
[ 10.2092471] ---[ end Kernel panic - not syncing: UFS: 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)

(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 EXPORT_SYMBOL(panic);
481 (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"
$
```

```
[ 1.286853] Demotion targets for Node 0: null
[ 1.332863] PM: Magic number: 5:870:650
[ 1.333015] PM: hash matches drivers/base/power/main.c:1294
[ 1.334892] ALSA device list:
[ 1.334998] No soundcards found.
[ 1.349838] scsi 1:0:0:0: CD-ROM QEMU QEMU DVD-ROM 2.5+ PQ
: 0 ANSI: 5
[ 1.373710] sr 1:0:0:0: [sr0] scsi3-mmc drive: 4x/4x cd/rw xa/form2 tray
[ 1.375029] cdrom: Uniform CD-ROM driver Revision: 3.20
[ 1.385691] sr 1:0:0:0: Attached scsi generic sg0 type 5
[ 1.469915] Freeing unused kernel image (initmem) memory: 2284K
[ 1.472000] input: ImExPS/2 Generic Explorer Mouse as /devices/platform/i8042
/serio1/input/input3
[ 1.473189] Write protecting the kernel read-only data: 20480k
[ 1.476476] Freeing unused kernel image (text/rodata gap) memory: 1248K
[ 1.478607] Freeing unused kernel image (rodata/data gap) memory: 1380K
[ 1.836061] x86/mm: Checked W+X mappings: passed, no W+X pages found.
[ 1.836663] tsc: Refined TSC clocksource calibration: 3601.811 MHz
[ 1.837039] clocksource: tsc: mask: 0xffffffffffffffff max_cycles: 0x33eb01c7
117, max_idle_ns: 440795288338 ns
[ 1.837272] clocksource: Switched to clocksource tsc
[ 1.838637] Run /init as init process
init: cannot set terminal process group (-1): Inappropriate ioctl for device
init: no job control in this shell
init-5.3# _
```

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`:

```
#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 were only used by OOP languages? 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.

```
#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.

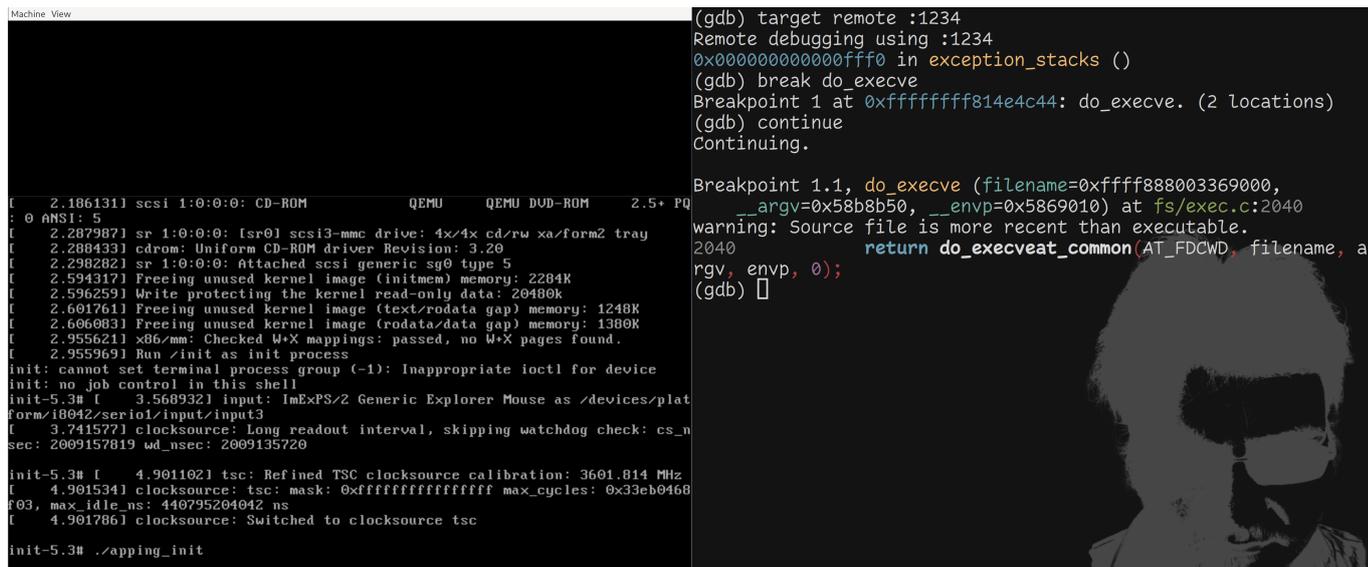
```
$ qemu-system-x86_64 -kernel bzImage -initrd initrd.img -append "nokaslr init=/init" -s -S &
$ gdb vmlinux
(gdb) target remote :1234
(gdb) break do_execve
(gdb) continue
(...)
```

Now from QEMU, execute `apping_init`

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

gdb will catch the breakpoint.

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



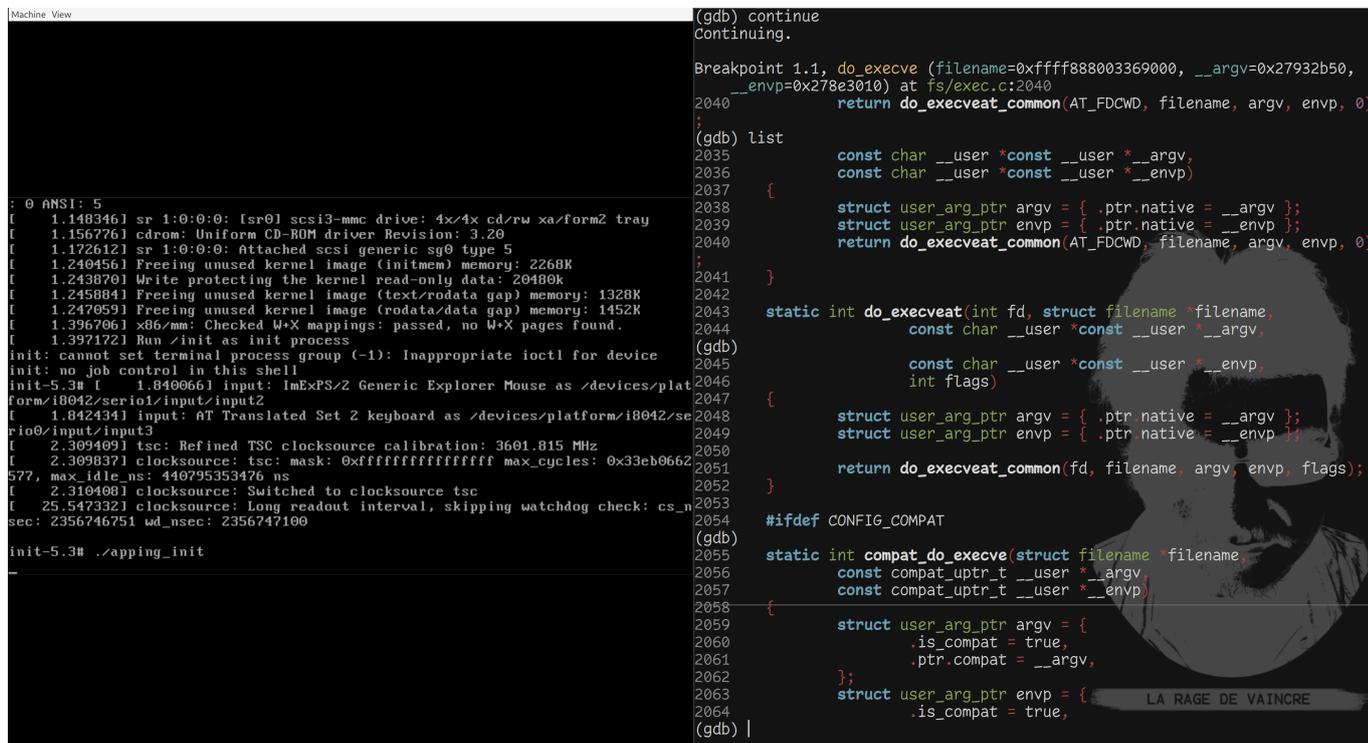
```
Machine View
[  2.186131] scsi 1:0:0:0: CD-ROM          QEMU      QEMU DVD-ROM    2.5+ PQ
: 0 ANSI: 5
[  2.287987] sr 1:0:0:0: [sr0] scsi3-mmc drive: 4x/4x cd/rw xa/form2 tray
[  2.288433] cdrom: Uniform CD-ROM driver Revision: 3.20
[  2.298282] sr 1:0:0:0: Attached scsi generic sg0 type 5
[  2.594317] Freeing unused kernel image (initmem) memory: 2284K
[  2.596259] Write protecting the kernel read-only data: 20480k
[  2.601761] Freeing unused kernel image (text/rodata gap) memory: 1248K
[  2.606083] Freeing unused kernel image (rodata/data gap) memory: 1380K
[  2.955621] x86/mm: Checked W*X mappings: passed, no W*X pages found.
[  2.955969] Run /init as init process
init: cannot set terminal process group (-1): Inappropriate ioctl for device
init: no job control in this shell
init-5.3# [  3.568932] input: ImExPS/2 Generic Explorer Mouse as /devices/plat
form/i8042/serio1/input/input3
[  3.741577] clocksource: Long readout interval, skipping watchdog check: cs_n
sec: 2009157819 wd_nsec: 2009135720
init-5.3# [  4.901102] tsc: Refined TSC clocksource calibration: 3601.814 MHz
[  4.901534] clocksource: tsc: mask: 0xffffffffffff max_cycles: 0x33eb0468
003, max_idle_ns: 440795204042 ns
[  4.901786] clocksource: Switched to clocksource tsc
init-5.3# ./apping_init

(gdb) target remote :1234
Remote debugging using :1234
0x0000000000000000 in exception_stacks ()
(gdb) break do_execve
Breakpoint 1 at 0xffffffff814e4c44: do_execve. (2 locations)
(gdb) continue
Continuing.

Breakpoint 1.1, do_execve (filename=0xffff888003369000,
__argv=0x58b8b50, __envp=0x5869010) at fs/exec.c:2040
warning: Source file is more recent than executable.
2040      return do_execveat_common(AT_FDCWD, filename, a
rgv, envp, 0);
(gdb) |
```

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)
(...)
```



```
Machine View
[  1.148346] sr 1:0:0:0: [sr0] scsi3-mmc drive: 4x/4x cd/rw xa/form2 tray
[  1.156776] cdrom: Uniform CD-ROM driver Revision: 3.20
[  1.172612] sr 1:0:0:0: Attached scsi generic sg0 type 5
[  1.240456] Freeing unused kernel image (initmem) memory: 2268K
[  1.243879] Write protecting the kernel read-only data: 20480k
[  1.245884] Freeing unused kernel image (text/rodata gap) memory: 1328K
[  1.247059] Freeing unused kernel image (rodata/data gap) memory: 1452K
[  1.396706] x86/mm: Checked W*X mappings: passed, no W*X pages found.
[  1.397172] Run /init as init process
init: cannot set terminal process group (-1): Inappropriate ioctl for device
init: no job control in this shell
init-5.3# [  1.840961] input: ImExPS/2 Generic Explorer Mouse as /devices/plat
form/i8042/serio1/input/input2
[  1.842434] input: AT Translated Set 2 keyboard as /devices/platform/i8042/ser
io0/input/input3
[  2.309409] tsc: Refined TSC clocksource calibration: 3601.815 MHz
[  2.309837] clocksource: tsc: mask: 0xffffffffffff max_cycles: 0x33eb0662
577, max_idle_ns: 440795353476 ns
[  2.310400] clocksource: Switched to clocksource tsc
[  2.547332] clocksource: Long readout interval, skipping watchdog check: cs_n
sec: 2356746751 wd_nsec: 2356747100
init-5.3# ./apping_init

(gdb) continue
Continuing.

Breakpoint 1.1, do_execve (filename=0xffff888003369000, __argv=0x27932b50,
__envp=0x278e3010) at fs/exec.c:2040
2040      return do_execveat_common(AT_FDCWD, filename, argv, envp, 0)
;
(gdb) list
2035      const char __user *const __user * __argv,
2036      const char __user *const __user * __envp)
2037  {
2038      struct user_arg_ptr argv = { .ptr.native = __argv };
2039      struct user_arg_ptr envp = { .ptr.native = __envp };
2040      return do_execveat_common(AT_FDCWD, filename, argv, envp, 0)
;
2041  }
2042
2043  static int do_execveat(int fd, struct filename *filename,
2044      const char __user *const __user * __argv,
2045      const char __user *const __user * __envp,
2046      int flags)
2047  {
2048      struct user_arg_ptr argv = { .ptr.native = __argv };
2049      struct user_arg_ptr envp = { .ptr.native = __envp };
2050
2051      return do_execveat_common(fd, filename, argv, envp, flags);
2052  }
2053
2054  #ifdef CONFIG_COMPAT
2055  (gdb)
2056  static int compat_do_execve(struct filename *filename,
2057      const compat_uptr_t __user * __argv,
2058      const compat_uptr_t __user * __envp)
2059  {
2060      struct user_arg_ptr argv = {
2061          .is_compat = true,
2062          .ptr.compat = __argv,
2063      };
2064      struct user_arg_ptr envp = {
2065          .is_compat = true,
2066      };
2067      return do_execveat_common(AT_FDCWD, filename, argv, envp, 0);
2068  }
2069  #endif
(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 completely explained here.

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

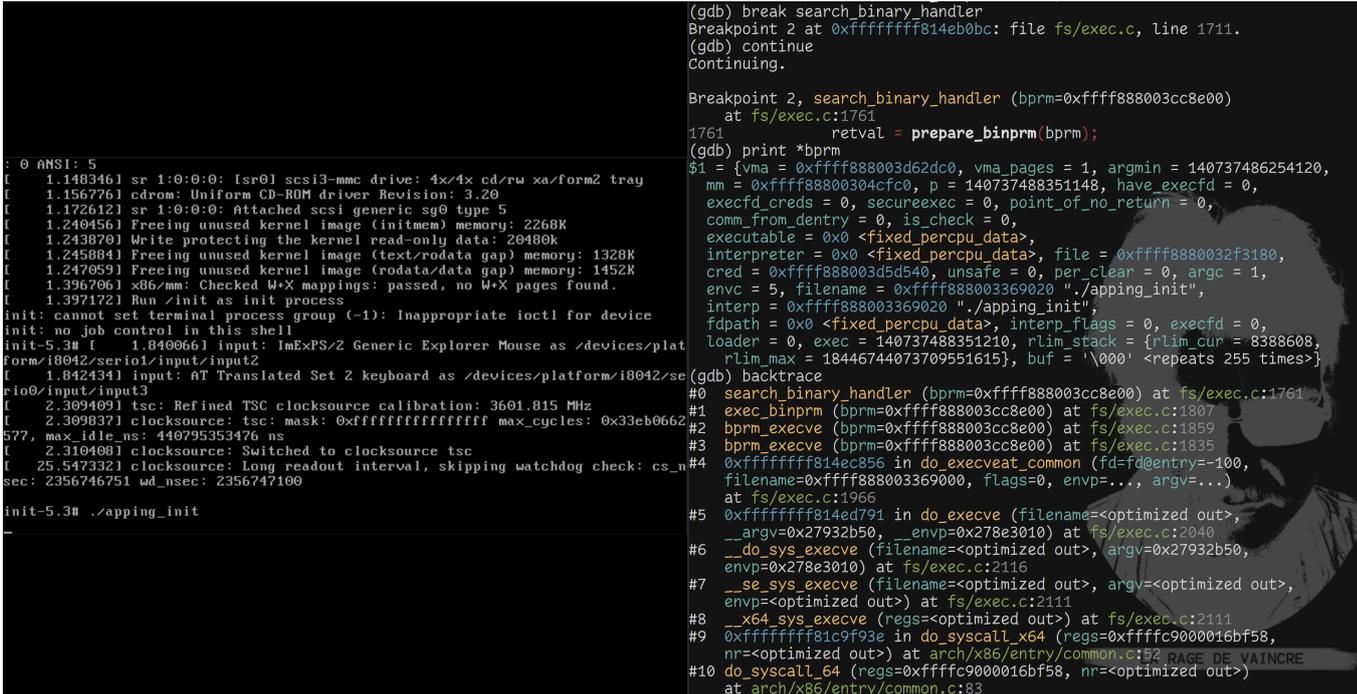
```
(gdb) break search_binary_handler
(gdb) continue
(...)
(gdb) # Breakpoint reached
```

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)
(...)
```



```
(gdb) break search_binary_handler
Breakpoint 2 at 0xffffffff814eb0bc: file fs/exec.c, line 1711.
(gdb) continue
Continuing.

Breakpoint 2, search_binary_handler (bprm=0xffff888003cc8e00)
at fs/exec.c:1761
1761         retval = prepare_binprm(bprm);
(gdb) print *bprm
$1 = {vma = 0xffff888003d62dc0, vma_pages = 1, argmin = 140737486254120,
mm = 0xffff88800304cfc0, p = 140737488351148, have_execfd = 0,
execfd_creds = 0, secureexec = 0, point_of_no_return = 0,
comm_from_dentry = 0, is_check = 0,
executable = 0x0 <fixed_percpu_data>,
interpreter = 0x0 <fixed_percpu_data>, file = 0xffff8880032f3180,
cred = 0xffff888003d5d540, unsafe = 0, per_clear = 0, argc = 1,
envc = 5, filename = 0xffff888003369020 "/.apping_init",
interp = 0xffff888003369020 "/.apping_init",
fdpath = 0x0 <fixed_percpu_data>, interp_flags = 0, execfd = 0,
loader = 0, exec = 140737488351210, rlim_stack = {rlim_cur = 8388608,
rlim_max = 18446744073709551615}, buf = '\000' <repeats 255 times>}
(gdb) backtrace
#0 search_binary_handler (bprm=0xffff888003cc8e00) at fs/exec.c:1761
#1 exec_binprm (bprm=0xffff888003cc8e00) at fs/exec.c:1807
#2 bprm_execve (bprm=0xffff888003cc8e00) at fs/exec.c:1859
#3 bprm_execve (bprm=0xffff888003cc8e00) at fs/exec.c:1835
#4 0xffffffff8142c856 in do_execveat_common (fd=fdentry=-100,
filename=0xffff888003369000, flags=0, envp=..., argv=...)
at fs/exec.c:1966
#5 0xffffffff814ed791 in do_execve (filename=<optimized out>,
__argv=0x27932b50, __envp=0x278e3010) at fs/exec.c:2040
#6 __do_sys_execve (filename=<optimized out>, argv=0x27932b50,
envp=0x278e3010) at fs/exec.c:2116
#7 __se_sys_execve (filename=<optimized out>, argv=<optimized out>,
envp=<optimized out>) at fs/exec.c:2111
#8 __x64_sys_execve (regs=<optimized out>) at fs/exec.c:2111
#9 0xffffffff81c9f93e in do_syscall_x64 (regs=0xffffc9000016bf58,
nr=<optimized out>) at arch/x86/entry/common.c:52
#10 do_syscall_64 (regs=0xffffc9000016bf58, nr=<optimized out>)
at arch/x86/entry/common.c:83
```

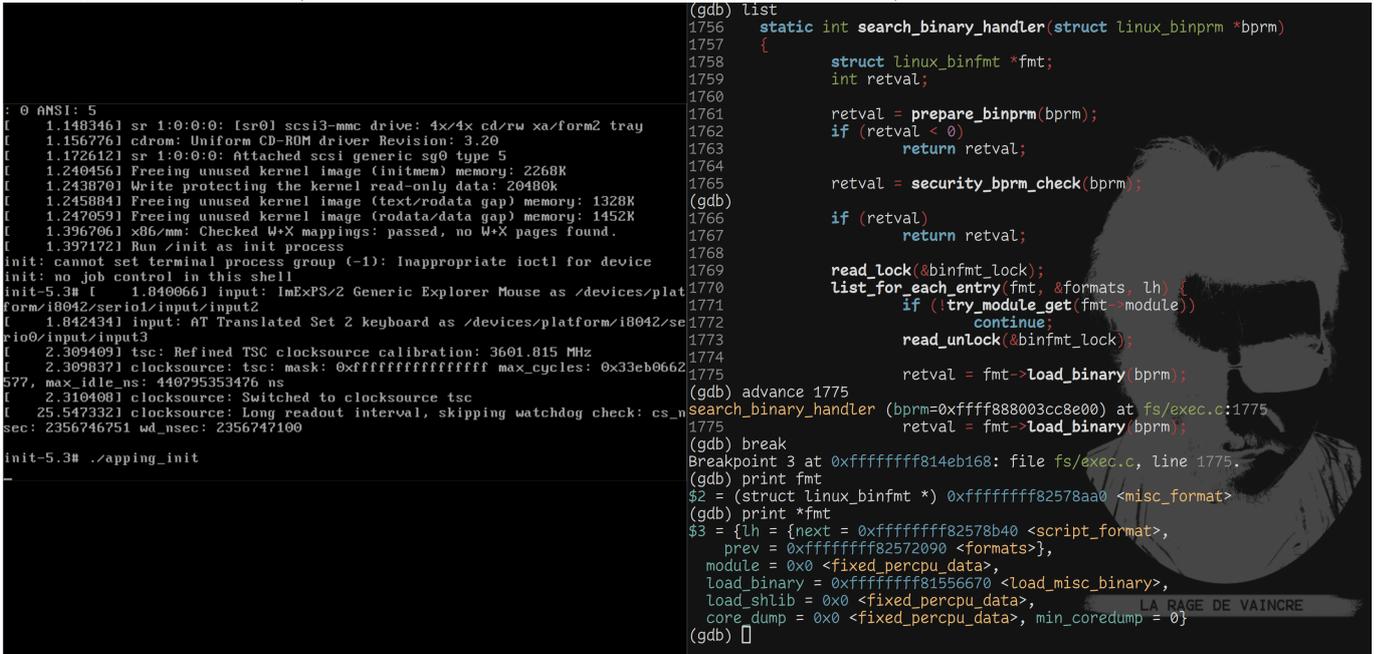
The interesting line in the next instructions is this one:

```
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`<sup>3</sup>.



```
(gdb) list
1756 static int search_binary_handler(struct linux_binprm *bprm)
1757 {
1758     struct linux_binfmt *fmt;
1759     int retval;
1760
1761     retval = prepare_binprm(bprm);
1762     if (retval < 0)
1763         return retval;
1764
1765     retval = security_bprm_check(bprm);
(gdb)
1766     if (retval)
1767         return retval;
1768
1769     read_lock(&binfmt_lock);
1770     list_for_each_entry(fmt, &formats, lh) {
1771         if (!try_module_get(fmt->module))
1772             continue;
1773         read_unlock(&binfmt_lock);
1774
1775         retval = fmt->load_binary(bprm);
(gdb) advance 1775
search_binary_handler (bprm=0xfffff888003cc8e00) at fs/exec.c:1775
1775         retval = fmt->load_binary(bprm);
(gdb) break
Breakpoint 3 at 0xfffff814eb168: file fs/exec.c, line 1775.
(gdb) print fmt
$2 = (struct linux_binfmt *) 0xfffff82578aa0 <misc_format>
(gdb) print *fmt
$3 = {lh = {next = 0xfffff82578b40 <script_format>,
prev = 0xfffff82572090 <formats>},
module = 0x0 <fixed_percpu_data>,
load_binary = 0xfffff81556670 <load_misc_binary>,
load_shlib = 0x0 <fixed_percpu_data>,
core_dump = 0x0 <fixed_percpu_data>, min_coredump = 0}
(gdb) □
```

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

<sup>3</sup>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`<sup>4</sup> any-time 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.

<sup>4</sup>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
```

```
Machine View
[ 1.259790] ushcore: registered new interface driver usbhid
[ 1.259790] usbhid: USB HID core driver
[ 1.259790] IPI shorthand broadcast: enabled
[ 1.309133] hpet: Lost 2 RTC interrupts
[ 1.426478] sched_clock: Marking stable (1072684076, 353565804)-(2362667618,
-936417738)
[ 1.439960] Demotion targets for Mode 0: null
[ 1.442405] PM: Magic number: 5:401:402
[ 1.442539] PM: hash matches drivers/base/power/main.c:999
[ 1.444153] ALSA device list:
[ 1.449411] No soundcards found.
[ 1.931113] ata2: found unknown device (class 0)
[ 1.936462] ata2.00: ATAPI: QEMU DUD-ROM, 2.5+, max UDMA/100
[ 1.946949] scsi 1:0:0:0: CD-ROM          QEMU      QEMU DUD-ROM    2.5+ PQ
: 0 ANS1: 5
[ 2.023904] sr 1:0:0:0: [sr0] scsi3-mmc drive: 4x4x cd/rw xa/form2 tray
[ 2.024210] cdrom: Uniform CD-ROM driver Revision: 3.20
[ 2.035831] sr 1:0:0:0: Attached scsi generic sg0 type 5
[ 2.149959] Freeing unused kernel image (initmem) memory: 2268K
[ 2.150394] Write protecting the kernel read-only data: 20480k
[ 2.152491] Freeing unused kernel image (text/rodata gap) memory: 1328K
[ 2.154183] Freeing unused kernel image (rodata/data gap) memory: 1452K
[ 2.383306] x86/mm: Checked W*X mappings: passed, no W*X pages found.
[ 2.384282] Run /init as init process

(gdb) target remote :1234
Remote debugging using :1234
0x00000000000000ff0 in exception_stacks ()
(gdb) break ksys_write
Breakpoint 1 at 0xffffffff814e1ed0: file fs/read_write.c, line 721.
(gdb) continue
Continuing.

Breakpoint 1, ksys_write (fd=1, buf=0x47c010 "APPING\n", count=7)
at fs/read_write.c:721
721      {
(gdb) 
```

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
(...)
```

```
Breakpoint 3, vc_con_write_normal (vc=0xffff888003057000, tc=65, c=65,
draw=<synthetic pointer>) at drivers/tty/vt/vt.c:2947
2947     unsigned char vc_attr = vc->vc_attr;
(gdb) list
2942
2943     static int vc_con_write_normal(struct vc_data *vc, int tc, int c,
2944                                 struct vc_draw_region *draw)
2945     {
2946         int next_c;
2947         unsigned char vc_attr = vc->vc_attr;
2948         u16 himask = vc->vc_hi_font_mask, charmask = himask ? 0x1ff
: 0xff;
8, 2949         u8 width = 1;
2950         bool inverse = false;
2951
(gdb)
2952         if (vc->vc_utf && !vc->vc_disp_ctrl) {
2953             if (is_double_width(c))
2954                 width = 2;
2955     }
```

```

(gdb) backtrace
#0  vc_con_write_normal (vc=0xffff888003057000, tc=65, c=65,
    draw=<synthetic pointer>) at drivers/tty/vt/vt.c:2947
#1  do_con_write (tty=tty@entry=0xffff888003914c00, buf=<optimized out>,
    count=<optimized out>) at drivers/tty/vt/vt.c:3095
#2  0xffffffff817bb021 in con_write (tty=0xffff888003914c00,
    buf=<optimized out>, count=<optimized out>) at drivers/tty/vt/vt.c:3432
#3  0xffffffff817a2296 in process_output_block (tty=0xffff888003914c00,
    buf=0xffff888003975c00 "APPING\n", nr=<optimized out>)
    at drivers/tty/n/tty.c:574
#4  n_tty_write (tty=0xffff888003914c00, file=0xffff88800330d000,
    buf=0xffff888003975c00 "APPING\n", nr=7) at drivers/tty/n/tty.c:2389
#5  0xffffffff8179e110 in iterate_tty_write (ld=0xffff8880038bfef0,
    tty=0xffff888003914c00, file=0xffff88800330d000,
    from=0xffffc90000013e80) at drivers/tty/tty_io.c:1015
#6  file_tty_write (file=0xffff88800330d000, from=0xffffc90000013e80,
    iocb=<optimized out>) at drivers/tty/tty_io.c:1090
#7  0xffffffff814e1b8e in new_sync_write (filp=0xffff88800330d000,
    buf=0x47c010 "APPING\n", len=7, ppos=0xffffc90000013f18)
    at fs/read_write.c:586
#8  vfs_write (file=file@entry=0xffff88800330d000,
    buf=buf@entry=0x47c010 "APPING\n", count=count@entry=7,
    pos=pos@entry=0xffffc90000013f18) at fs/read_write.c:679
#9  0xffffffff814e1f3f in ksys_write (fd=<optimized out>,
    buf=0x47c010 "APPING\n", count=7) at fs/read_write.c:731
#10  0xffffffff81c9f93e in do_syscall_x64 (regs=0xffffc90000013f58,
    nr=<optimized out>) at arch/x86/entry/common.c:52
#11  do_syscall_64 (regs=0xffffc90000013f58, nr=<optimized out>)
    at arch/x86/entry/common.c:83
#12  0xffffffff81000130 in entry_SYSCALL_64 ()
    at arch/x86/entry/entry_64.S:121
led
[ 0.637172] hid: raw HID events driver (C) Jiri Kosina
[ 0.650376] usbcore: registered new interface driver usbhid
[ 0.650376] usbhid: USB HID core driver
[ 0.665732] IPI shorthand broadcast: enabled
[ 0.690355] sched_clock: Marking stable (700695555, -19340427)->(1169395418,
    480040290)
[ 0.707271] Demotion targets for Node 0: null
[ 0.709871] PM: Magic number: 5:160:655
[ 0.711166] ALSA device list:
[ 0.711681] No soundcards found.
[ 1.249343] ata2: found unknown device (class 0)
[ 1.254279] ata2.00: ATAPI: QEMU DVD-ROM, 2.5+, max UDMA/100
[ 1.266559] scsi 1:0:0:0: CD-ROM          QEMU    QEMU DVD-ROM    2.5+ PQ
: 0 ANSII: 5
[ 1.290866] sr 1:0:0:0: [sr0] scsi3-mmc drive: 4x/4x cd/rw xa/form2 tray
[ 1.291174] cdrom: Uniform CD-ROM driver Revision: 3.20
[ 1.305661] sr 1:0:0:0: Attached scsi generic sg0 type 5
[ 1.370903] Freeing unused kernel image (initmem) memory: 2268K
[ 1.371657] Write protecting the kernel read-only data: 20480k
[ 1.374759] Freeing unused kernel image (text/rodata gap) memory: 1328K
[ 1.376410] Freeing unused kernel image (rodata/data gap) memory: 1452K
[ 1.610179] x86/mm: Checked W*X mappings: passed, no W*X pages found.
[ 1.611040] Run /init as init process

```

list the following instructions. Can you spot this line?

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

```

2984         if (tc < 0)
2985             tc = '?';
2986
2987         vc_attr = vc_invert_attr(vc);
2988         con_flush(vc, draw);
2989     }
2990 }
2991 }
(gdb)
2992
2993     next_c = c;
2994     while (1) {
2995         if (vc->vc_need_wrap || vc->vc_decim)
2996             con_flush(vc, draw);
2997         if (vc->vc_need_wrap) {
2998             cr(vc);
2999             lf(vc);
3000         }
3001         if (vc->vc_decim)
(gdb)
3002             insert_char(vc, 1);
3003         vc_unisr_putc(vc, next_c);
3004
3005         if (himask)
3006             tc = ((tc & 0x100) ? himask : 0) |
3007                 (tc & 0xff);
3008         tc |= (vc_attr << 8) & ~himask;
3009
3010         scr_writew(tc, (u16 *)vc->vc_pos);
3011
(gdb)
3012         if (con_should_update(vc) && draw->x < 0) {
3013             draw->x = vc->state.x;
3014             draw->from = vc->vc_pos;
3015         }
3016         if (vc->state.x == vc->vc_cols - 1) {
3017             vc->vc_need_wrap = vc->vc_decawm;
3018             draw->to = vc->vc_pos + 2;
3019         } else {
3020             vc->state.x++;
3021             draw->to = (vc->vc_pos += 2);
(gdb) □

```

`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
```

```

3008         tc |= (vc_attr << 8) & ~TIMASK;
3009
3010         scr_writew(tc, (u16 *)vc->vc_pos);
3011
(gdb)
3012         if (con_should_update(vc) && draw->x < 0) {
3013             draw->x = vc->state.x;
3014             draw->from = vc->vc_pos;
3015         }
3016         if (vc->state.x == vc->vc_cols - 1) {
3017             vc->vc_need_wrap = vc->vc_decawm;
3018             draw->to = vc->vc_pos + 2;
3019         } else {
3020             vc->state.x++;
3021             draw->to = (vc->vc_pos += 2);
(gdb) advance 3010
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) |

```

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 address 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/Printing_To_Screen)

[https://wiki.osdev.org/Text\\_UI](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 colour* 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 ( $80 \times 25 \times 2 = 4,000$  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 contrasted 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!

```
Machine View
(gdb) print/x $r10w
$1 = 0x741
(gdb) print/x $r10w=0x141
$2 = 0x141
(gdb) continue
Continuing.

Breakpoint 3, vc_con_write_normal (vc=0xffff888003057000, tc=80, c=80,
draw=<synthetic pointer>) at drivers/tty/vt/vt.c:2947
2947 unsigned char vc_attr = vc->vc_attr;
(gdb) |

led
[ 0.637172] hid: raw HID events driver (C) Jiri Kosina
[ 0.650376] usbcore: registered new interface driver usbhid
[ 0.650376] usbhid: USB HID core driver
[ 0.665732] IPi shorthand broadcast: enabled
[ 0.690355] sched_clock: Marking stable (708695555, -19340427)->(1169395418,
-480040290)
[ 0.707271] Demotion targets for Node 0: null
[ 0.709871] PM: Magic number: 5:160:655
[ 0.711166] ALSA device list:
[ 0.711681] No soundcards found.
[ 1.249343] ata2: found unknown device (class 0)
[ 1.254279] ata2.00: ATAPI: QEMU DVD-ROM, 2.5+, max UDMA/100
[ 1.266559] scsi 1:0:0:0: CD-ROM QEMU QEMU DVD-ROM 2.5+ PQ
: 0 ANSI: 5
[ 1.290866] sr 1:0:0:0: [sr0] scsi3-mmc drive: 4x/4x cd/rw xa/form2 tray
[ 1.291174] cdrom: Uniform CD-ROM driver Revision: 3.20
[ 1.305661] sr 1:0:0:0: Attached scsi generic sg0 type 5
[ 1.370903] Freeing unused kernel image (initmem) memory: 2268K
[ 1.371657] Write protecting the kernel read-only data: 20480k
[ 1.374759] Freeing unused kernel image (text/rodata gap) memory: 1328K
[ 1.376410] Freeing unused kernel image (rodata/data gap) memory: 1452K
[ 1.610179] x86/mm: Checked W*X mappings: passed, no W*X pages found.
[ 1.611040] Run /init as init process
```

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
```

```

[ 0.707271] Demotion targets for Node 0: null
[ 0.709871] PM: Magic number: 5:160:655
[ 0.711166] ALSA device list:
[ 0.711681] No soundcards found.
[ 1.249343] ata2: found unknown device (class 0)
[ 1.254279] ata2.00: ATAPI: QEMU DVD-ROM, 2.5+, max UDMA/100
[ 1.266559] scsi 1:0:0:0: CD-ROM QEMU QEMU DVD-ROM 2.5+ PQ
: 0 ANSI: 5
[ 1.290866] sr 1:0:0:0: [sr0] scsi3-mmc drive: 4x/4x cd/rw xa/form2 tray
[ 1.291174] cdrom: Uniform CD-ROM driver Revision: 3.20
[ 1.305661] sr 1:0:0:0: Attached scsi generic sg0 type 5
[ 1.370903] Freeing unused kernel image (initmem) memory: 2268K
[ 1.371657] Write protecting the kernel read-only data: 20480k
[ 1.374759] Freeing unused kernel image (text/rodata gap) memory: 1328K
[ 1.376410] Freeing unused kernel image (rodata/data gap) memory: 1452K
[ 1.610179] x86/mm: Checked W*X mappings: passed, no W*X pages found.
[ 1.611040] Run /init as init process
P

Breakpoint 4, 0xfffffff817b908d in vc_con_write_normal (
vc=0xffff888003057000, tc=1872, c=<optimized out>,
draw=<synthetic pointer>) at drivers/tty/vt/vt.c:3010
3010 scr_writew(tc, (u16 *)vc->vc_pos);
(gdb) print/x $r10w
$5 = 0x750
(gdb) print/x $r10w=0xf50
$6 = 0xf50
(gdb) c
Continuing.

Breakpoint 3, vc_con_write_normal (vc=0xffff888003057000, tc=
draw=<synthetic pointer>) at drivers/tty/vt/vt.c:2947
2947 unsigned char vc_attr = vc->vc_attr;
(gdb) |
```

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
```

```

[ 1.370903] Freeing unused kernel image (initmem) memory: 2268K
[ 1.371657] Write protecting the kernel read-only data: 20480k
[ 1.374759] Freeing unused kernel image (text/rodata gap) memory: 1328K
[ 1.376410] Freeing unused kernel image (rodata/data gap) memory: 1452K
[ 1.610179] x86/mm: Checked W*X mappings: passed, no W*X pages found.
[ 1.611040] Run /init as init process
PP

Breakpoint 4, 0xfffffff817b908d in vc_con_write_normal (
vc=0xffff888003057000, tc=1872, c=<optimized out>,
draw=<synthetic pointer>) at drivers/tty/vt/vt.c:3010
3010 scr_writew(tc, (u16 *)vc->vc_pos);
(gdb) print/x $r10w
$8 = 0x750
(gdb) print/x $r10w=0x450
$9 = 0x450
(gdb) c
Continuing.

Breakpoint 3, vc_con_write_normal (vc=0xffff888003057000, tc=73, c=73,
draw=<synthetic pointer>) at drivers/tty/vt/vt.c:2947
2947 unsigned char vc_attr = vc->vc_attr;
(gdb) |
```

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.

