Fachprojekt „Systemsoftwaretechnik”

01 - Debugging im Linux-Kern (built-in)

Alexander Krause

AG Systemsoftware

Veranstaltungswebseite

30.04.24

Code of Conduct

- Wir in der Fakultät für Informatik fördern ein motivierendes und chancengleiches Umfeld, in welchem wir gemeinsam lernen, lehren und forschen.
- Diskriminierung und sexualisierte Gewalt werden nicht toleriert, intensiv verfolgt und im Rahmen der rechtlichen Möglichkeiten sanktioniert.
- Zu inakzeptablem Verhalten zählt u.a. belästigendes, beleidigendes, diskriminierendes, einschüchterndes, abwertendes und erniedrigendes Verhalten und Sprache.

Ansprechpartner*innen

Wenn Sie Diskriminierung oder sexualisierte Gewalt erleben oder beobachten oder Fragen zum Thema haben, können Sie sich beispielsweise an folgende Stellen wenden:

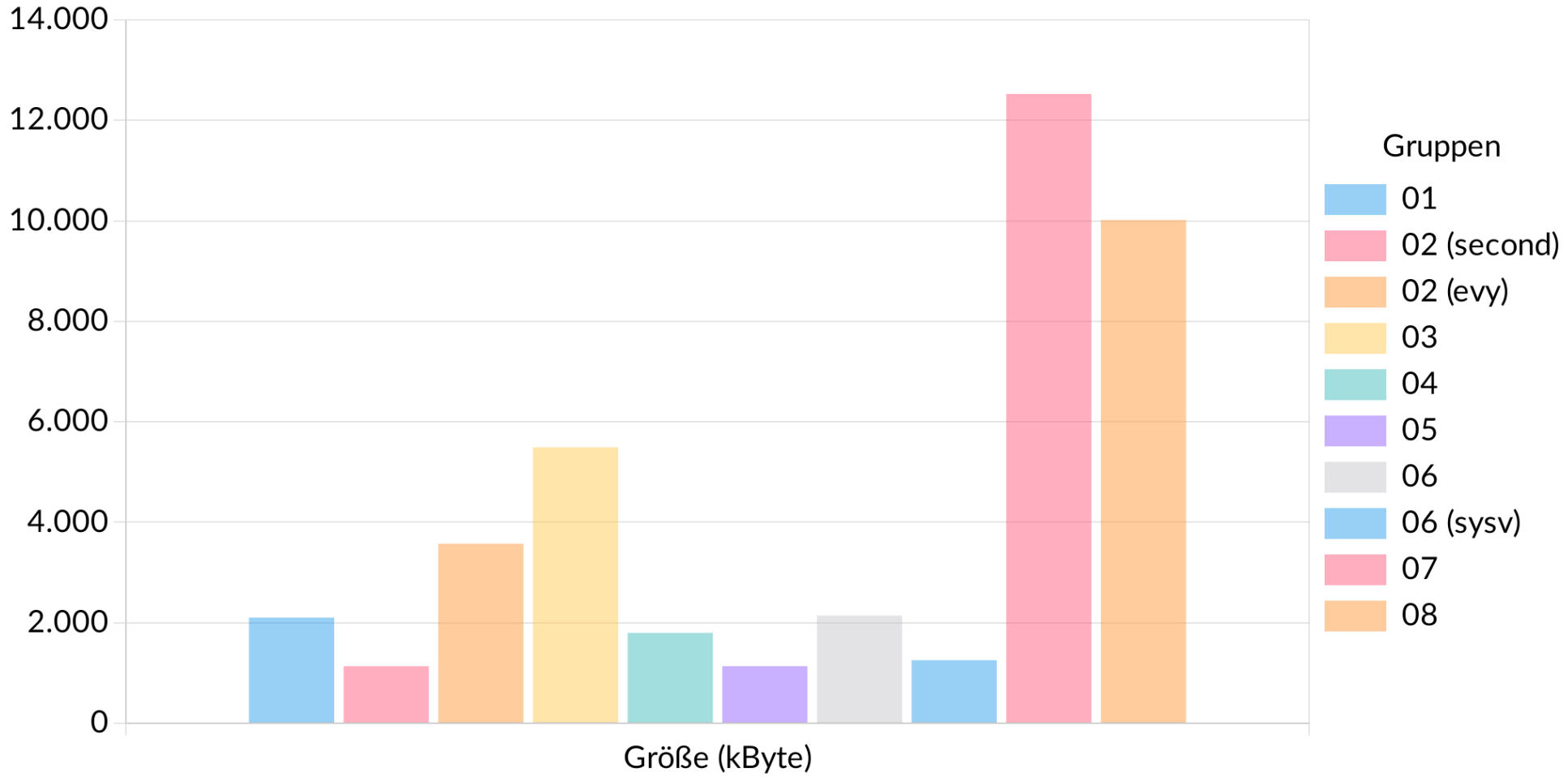
- Die Lehrperson(en) Ihrer Lehrveranstaltung
- Alle Mitglieder der Diversitätskommission der Fakultät
- Fachschaftsrat, auch anonym unter vertraulich@oh14.de
- Zentrale Beratungsstelle zum Schutz vor Diskriminierung und sexualisierter Gewalt
- Prorektorin Diversität

Besprechung A0

A0-Resumee

- Was habt Ihr beobachtet und wie seid Ihr damit umgegangen?
- Was hatte ich mit Euch vor?
  - Installation und Einrichtung einer eigenen Debian-Installation
  - Vor allem aber: Das Bauen eines **eigenen** Linux-Kerns
  - Kontakt mit dem Build-Umgebung
  - Erstes Gefühl für die Zusammenhänge im Linux-Kern (Stichwort: Modul-Unterstützung)

Challenge: Kernel-Größe



- And the winner is …

- 🎊 Gruppe 02 **und** 05 mit 1136 kBytes! 🎉

Ooooops ...

```
[  28.399265] sysrq: Trigger a crash
[  28.400808] Kernel panic - not syncing: sysrq triggered crash
[  28.402661] CPU: 1 PID: 423 Comm: bash Not tainted 6.1.55 #1
[  28.404340] Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS 1.16.3-debian-1.16.3-2 04/01/2014
[  28.407068] Call Trace:
[  28.407904]      <TASK>
[  28.408640]      dump_stack_lvl+0x33/0x46
[  28.409829]      panic+0x10d/0x298
[  28.414116]      ? _printk+0x63/0x7e
[  28.415266]      sysrq_handle_crash+0x11/0x20
[  28.416503]      __handle_sysrq.cold+0x44/0x11c
...
[  28.422665]      ? __do_sys_newfstatat+0x4e/0x80
[  28.424027]      ksys_write+0x66/0xe0
[  28.425089]      do_syscall_64+0x3b/0x90
[  28.426394]      entry_SYSCALL_64_after_hwframe+0x64/0xce
[  28.436010] RIP: 0033:0x7f2ce5c97240
[  28.437316] Code: 40 00 48 8b 15 c1 9b 0d 00 f7 d8 64 89 02 48 c7 c0 ff ff ff ff eb b7 0f 1f 00 80 3d a1 23 0e 00 00 74 17 b8 01 00 00 0
[  28.442991] RSP: 002b:00007ffc1e8f92b8 EFLAGS: 00000202 ORIG_RAX: 0000000000000001
[  28.446632] RAX: ffffffffffffffda RBX: 0000000000000002 RCX: 00007f2ce5c97240
[  28.448944] RDX: 0000000000000002 RSI: 000055b536e4c840 RDI: 0000000000000001
[  28.456037] RBP: 000055b536e4c840 R08: 00007f2ce5d71c68 R09: 0000000000000073
[  28.458338] R10: 0000000000001000 R11: 0000000000000202 R12: 0000000000000002
[  28.461827] R13: 00007f2ce5d72760 R14: 0000000000000002 R15: 00007f2ce5d6d9e0
[  28.464128]      </TASK>
[  28.465561] Kernel Offset: 0x1b200000 from 0xffffffff81000000 (relocation range: 0xffffffff80000000-0xffffffffbfffffff)
[  28.468976] ---[ end Kernel panic - not syncing: sysrq triggered crash ]---
```

Kernel Panic

„Don’t panic!” - Fehlertypen

- Verschiedene Typen von Fehlern
  - *panic*
    - Ein unerwartete Fehler
    - Der Betriebssystemkern befindet sich in einer Sackgasse
    - Schwächere Form: *oops*; wird durch Beenden des Prozesses behoben
    - siehe kernel/panic.c
  - *bug*
    - Ein unerwarteter Fehler, der von der Programmierer:in entdeckt wird
    - Meist über `BUG_ON(X)` mit einer Bedingung `X`versehen ausgelöst
    - siehe bug.h
  - *warn*
    - Ein unerwarteter Fehler, der aber **behebbar** ist
    - Meist über `WARN_ON(X)` mit einer Bedingung `X` versehen ausgelöst
    - siehe bug.h

„Don’t panic!” - Automatische Fehlerbehandlung

- Verhalten des Kerns bei Fehlern ist konfigurierbar
- Zugriff über das Werkzeug `sysctl` (`man sysctl`) auf Kernelparameter in `/proc/sys`
- Setzen der Parameter in `kernel.panic` bestimmt
  - `kernel.panic`: Sekunden nach einer Panik bis zum automatischen Neustart (0 = kein Neustart)
  - `kernel.panic_on_oops`: Behandle ein `Oops` wie eine Panik, wenn auf 1 gesetzt
  - `kernel.panic_on_warn`: Behandle eine Warnung wie eine Panik, wenn auf 1 gesetzt

Beispiele für das Setzen der Kern-Parameter

Setzen der Kern-Parameter mittels `sysctl`

- Neustart bei Panik nach 3 Sekunden:

```
stud@sst:~$ sysctl kernel.panic
kernel.panic = 10
stud@sst:~$ sysctl -w kernel.panic=3
kernel.panic = 3
```

- Behandle Warnung als Panik:

```
stud@sst:~$ sysctl kernel.panic_on_warn
kernel.panic_on_warn = 0
stud@sst:~$ sysctl -w kernel.panic_on_warn=1
kernel.panic_on_warn = 1
```

Und wenn das System mal steht?

Dann helfen *Linux Sys-Requests* (*sysrq*) [1]

- *magische* Tastenkombination, die vom Linux-Kern (fast) immer akzeptiert wird
- `Alt Gr + Druck + X`
- **Achtung:** Bitte versucht dies **nicht** in der VM auf dem eigenen Notebook oder unseren Rechnern zu probieren. Die Kommandos treffen das Hostsystem und **nicht** die VM.
- Einfacher Merksatz für den Notfall:
  - **r**aising = Wechsel in den Raw-Modus bei der Eingabe
  - **e**lephants = Sende *SIGTERM* an alle Prozesse
  - **i**s = Sende *SIGKILL* an alle Prozesse
  - **s**o = Synchronisiere alle Dateisysteme
  - **u**tterly = Binde alle Dateisysteme als *read-only* ein

- boring = Starte neu

Laufzeitfehler

Man sieht ja den Wald vor lauter Bäumen nicht…

```
[  +0,147076] usb 3-1.3: New USB device found, idVendor=17ef, idProduct=30a9, bcdDevice=30.80
[  +0,000012] usb 3-1.3: New USB device strings: Mfr=1, Product=2, SerialNumber=3
[  +0,000007] usb 3-1.3: Product: 40AY
[  +0,000004] usb 3-1.3: Manufacturer: Lenovo
[  +0,000004] usb 3-1.3: SerialNumber: 1S40AY0090EUZVW0E6XW
[  +0,332044] usbcore: registered new device driver r8152-cfgselector
[  +0,118921] r8152-cfgselector 4-1.1: reset SuperSpeed USB device number 3 using xhci_hcd
[  +0,034908] r8152 4-1.1:1.0 (unnamed net_device) (uninitialized): Invalid header when reading pass-thru MAC addr
[  +0,011818] r8152 4-1.1:1.0: load rtl8153b-2 v2 04/27/23 successfully
[  +0,032424] r8152 4-1.1:1.0 eth0: v1.12.13
[  +0,000070] usbcore: registered new interface driver r8152
[  +0,008583] usbcore: registered new interface driver cdc_ether
[  +0,002117] usbcore: registered new interface driver r8153_ecm
[  +0,107748] r8152 4-1.1:1.0 enxf4a80d20224c: renamed from eth0
[  +0,696365] r8169 0000:06:00.0 enp6s0: Link is Down
[  +1,972187] r8152 4-1.1:1.0 enxf4a80d20224c: carrier on
[  +0,032709] r8152 4-1.1:1.0 enxf4a80d20224c: carrier off
[  +0,499472] usb 4-1.2: new SuperSpeed USB device number 5 using xhci_hcd
[  +0,032679] usb 4-1.2: New USB device found, idVendor=0bda, idProduct=0411, bcdDevice= 1.21
[  +0,000011] usb 4-1.2: New USB device strings: Mfr=1, Product=2, SerialNumber=0
[  +0,000005] usb 4-1.2: Product: 4-Port USB 3.0 Hub
[  +0,000003] usb 4-1.2: Manufacturer: Generic
[  +0,028545] hub 4-1.2:1.0: USB hub found
[  +0,001410] hub 4-1.2:1.0: 4 ports detected
[  +0,092498] usb 3-1.1.2: new high-speed USB device number 13 using xhci_hcd
[  +0,168774] usb 3-1.1.2: New USB device found, idVendor=0bda, idProduct=5411, bcdDevice= 1.21
[  +0,000010] usb 3-1.1.2: New USB device strings: Mfr=1, Product=2, SerialNumber=0
[  +0,000006] usb 3-1.1.2: Product: 4-Port USB 2.0 Hub
```

- Und nu?

Dynamic debug [2]

- Erlaubt das dynamische und selektive Einschalten von Ausgaben
- Es können die Ausgaben in …
  - einzelnen Zeilen,
  - ganzen Dateien oder
  - kompletten Modulen (aka Treibern) gesteuert werden.
- Code muss zur Ausgabe das Makro `pr_debug` verwenden (siehe linux/printk.h)
- Zugriff über `/sys/kernel/debug/dynamic_debug/control` im *debugfs*

Was ist das *debug*-Dateisystem (*debugfs*)?

- Pseudo-Dateisystem analog zu *procfs* oder *sysfs* [3]
- Richtet sich speziell an Kern-Entwickler
- Stellt Interna des Kerns als virtuelle Dateien bereit
- Bereitgestellte Schnittstelle gilt als nicht stabil [4]
- Separat eingebunden über `mount -t debugfs none /sys/kernel/debug`

Benutzung von *Dynamic debug*

- Aktuellen Zustand anschauen: `cat /sys/kernel/debug/dynamic_debug/control` (Nehmt lieber `less`)

▶ Ausgabe

- Alle 12 Nachrichten in der Funktion `svc_process()` aktivieren:

  ```
  echo 'func svc_process +p' > /sys/[...]/dynamic_debug/control
  ```

- Aktiviere die Nachricht in Zeile 1603 in Datei `vcsock.c`:

  ```
  echo 'file svcsock.c line 1603 +p' > ...
  ```

- Diese und weitere Beispiele finden sich in der Dokumentation ([2])

„C ist toll!” 😎

- Wo ist hier das Problem?

```c
void my_function(void) {
    int *bar = malloc(sizeof(int) * 42);
    // ...
    // three thousand lines later ...
    // ...
    bar[4] = 42;
}
```

- Die 168 Bytes auf dem Heap werden nie freigegeben!
- Je nach Ausführungshäufigkeit läuft der Hauptspeicher sukzessive voll.

Speicherlecks finden mit *kmemleak*

- Was ist *kmemleak*?
  - Detektor zum Auffinden von Speicherlecks im Linux-Kern [5]
  - Muss über Kconfig aktiviert werden
  - Sucht im Hintergrund nach nicht-referenzierten Speicherbereichen (-> Laufzeit-Overhead)
- Wie funktioniert *kmemleak*?
  - Funktionen zur Allokation und Freigabe werden instrumentiert
  - Zustand der Allokationen wird separat mitgeführt
  - Regelmäßiges Scannen des kompletten Hauptspeichers
  - Kontrolle über *debugfs*

Steuerung von *kmemleak*

- Auslesen der aktuellen Speicherlecks: `cat /sys/kernel/debug/kmemleak`
- Einen neuen Scan anstoßen: `echo scan > /sys/kernel/debug/kmemleak`
- Scanintervall setzen: `echo scan=<secs> > /sys/kernel/debug/kmemleak`
- Zusätzliche Information zu einem Speicherleck: `echo dump=<addr> > /sys/kernel/debug/kmemleak`

„C ist toll! … Wirklich!” 😏

- Wo ist hier das Problem?

```c
int bar[3];
int my_secret_key_for_hdd_encryption;

void my_function(void) {
    ...
    // three thousand lines later ...
    ...
    bar[4] = 42;
}
```

- Es erfolgt ein unerlaubter Zugriff über die Array-Grenzen hinaus
- Drei Fälle sind zu unterscheiden:
  - Bester Fall: Dahinter liegt nichts im Speicher
  - Mittelschlimmer Fall: Dahinter liegen unwichtige Daten
  - Schlimmster Fall: Dahinter liegen **wichtige** Daten, s. o.

Kernel Address Sanitizer [6]

- Setzt Compiler-Erweiterungen ein, um Validitätschecks vor **jedem** Speicherzugriff einzufügen
- Auf *arm64* auch mittels Hardwareunterstützung möglich
- Detektiert *out-of-bounds*- und *use-after-free*-Fehler
- Hält über die Laufzeitvariable `panic_on_warn` den Kernel bei einem Fehler an
- Verwandtes Werkzeug zum Auffinden von uninitialisierten Variablen: Kernel Memory Sanitizer (KMSAN)

„C ist toll! … Wirklich! Echt jetzt!” 😭

- Wo ist hier das Problem?

```c
int bar;
spinlock_t lock;

void my_thread_a(void) {
    spin_lock(&lock);
    bar = 42;
    spin_unlock(&lock);
}

void my_thread_b(void) {

    bar = 4711;

}
```

- Es erfolgt ein **unsychronisierter** Zugriff auf die Variable *bar*

„C ist toll! … Wirklich! Echt jetzt!” 😭

- Wo ist hier das Problem?

```c
int bar;
spinlock_t lock_a, lock_b;

void my_thread_a(void) {
    spin_lock(&lock_a);
    spin_lock(&lock_b);
    bar = 42;
    spin_unlock(&lock_b);
    spin_unlock(&lock_a);
}

void my_thread_b(void) {
    spin_lock(&lock_b);
    spin_lock(&lock_a);
    bar = 4711;
    spin_unlock(&lock_b);
    spin_unlock(&lock_a);
}
```

- Die Belegungsreihenfolge der Spinlocks ist **verschieden**

„C ist toll! … Wirklich! Echt jetzt!” 😭

- Wo ist hier das Problem?

```c
int bar;
spinlock_t lock_a;

void my_thread_a(void) {
    spin_lock(&lock_a);
    bar = 42;
    spin_unlock(&lock_a);
}

void interrupt_handler(void) {
    spin_lock(&lock_a);
    bar = 4711;
    spin_unlock(&lock_a);
}
```

- Die Unterbrechungen werden beim Zugriff aus dem Faden nicht gesperrt!

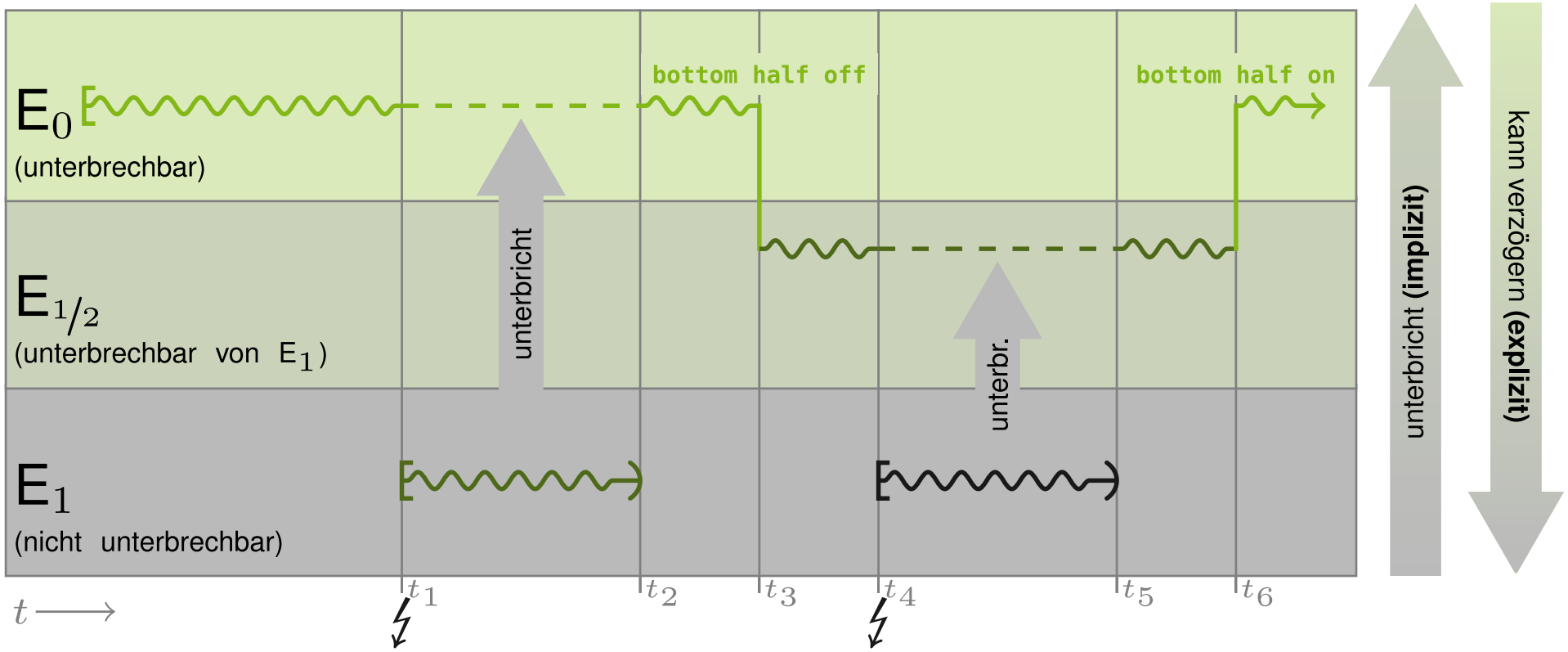
  🚨 Verklemmungsgefahr 🚨

- So ist es besser:

```c
// ...
void my_thread_a(void) {
```

```
    spin_lock_irq(&lock_a);
    bar = 42;
    spin_unlock_irq(&lock_a);
}
// ...
```

Unterbrechungssynchronisation in Betriebssystemen



(Für weitere Informationen siehe Foliensatz 05-IRQ-Synchronisation zur Veranstaltung „Betriebssystembau" (WS23))

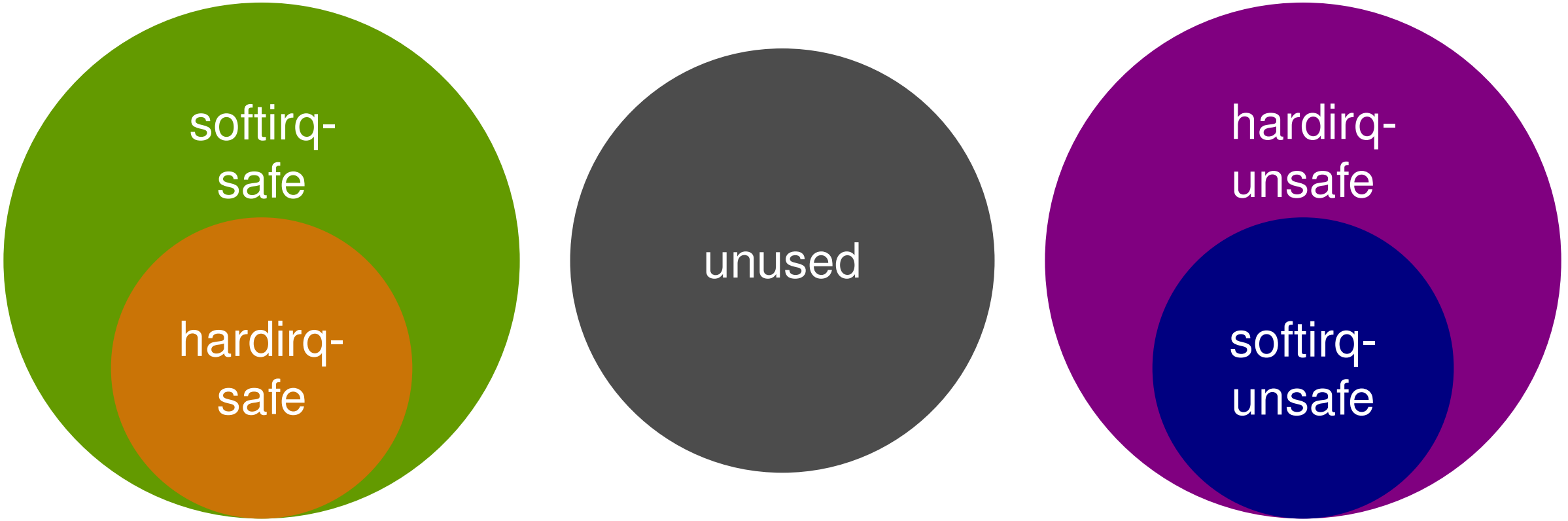
Runtime locking correctness validator (1/3)

- *lockdep* [7], [8] stellt Korrektheit des Sperrens sicher
- Prüft, ob eine Sperre *X-safe* ist, wobei X =
  - *hardirq*
  - *softirq*
- *X-safe* bedeutet, dass eine Sperre
  a. in Kontext *X* belegt wurde oder
  b. in Kontext eines Tasks mit *X* deaktiviert belegt wurde.
- Prüft Sperren-Reihenfolgen. Muss eine feste Abfolge geben. Folgendes ist verboten:
  - L1 -> L2
  - L2 -> L1

Runtime locking correctness validator (2/3)

Wie verhalten sich die Zustände zueinander?

- Jede Klasse von Locks darf **nur** in einem Zustand sein
- Transition in einen anderen Zustand → 🚨 Fehler! 🚨



Beziehungen der Zustände einer Sperre

Runtime locking correctness validator (3/3)

- Betrachtet nicht einzelne Instanzen von Sperren, sondern Klassen von Sperren
- **Beispiel:** Jede Instanz der Sperre `inode_lock` aus der Datenstruktur `struct inode` gehört zu der gleichen Klasse
- Weitere, verwandte Werkzeuge
  - zur Detektion von Wettlaufsituationen (*race condition*) zur Laufzeit Kernel Concurrency Sanitizer (KCSAN)
  - sowie zur statischen Sperren-Analyse sparse

Referenzen

[1]
Verfügbar unter: https://www.kernel.org/doc/html/v6.1/admin-guide/sysrq.html

[2]
Verfügbar unter: https://www.kernel.org/doc/html/v6.1/admin-guide/dynamic-debug-howto.html

[3]
Verfügbar unter: https://www.kernel.org/doc/html/v6.1/filesystems/debugfs.html

[4]
Jonathan Corbet, „Debugfs and the making of a stable ABI“. https://lwn.net/Articles/309298/, Dezember 2008.

[5]
Verfügbar unter: https://www.kernel.org/doc/html/v6.1/dev-tools/kmemleak.html?highlight=kmemleak

[6]
Verfügbar unter: https://www.kernel.org/doc/html/v6.1/dev-tools/kasan.html

[7]
J. Corbet, „The kernel lock validator“. März 2006. Verfügbar unter: https://lwn.net/Articles/185666/

[8]