Fachprojekt „Systemsoftwaretechnik”

02 - Debugging im Linux-Kern (extern)

Alexander Krause

AG Systemsoftware

Veranstaltungswebseite

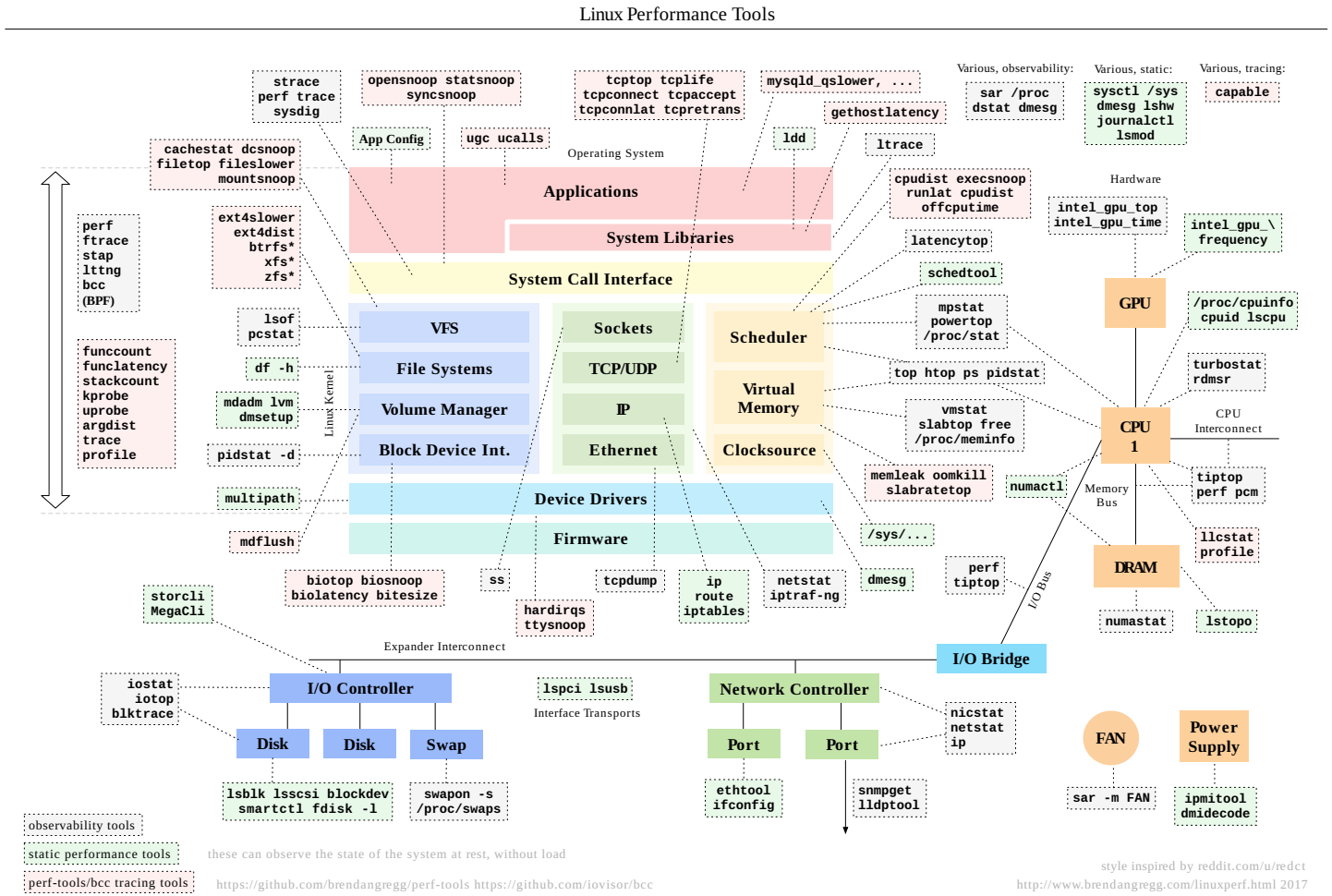
21.05.24

Debugging

„**printk()** is the king of all debuggers, […].” (LWN)

Was kennt Ihr sonst noch?

„Externe” Werkzeuge - Was heißt das?



Debugger

GNU Debugger (1/2)

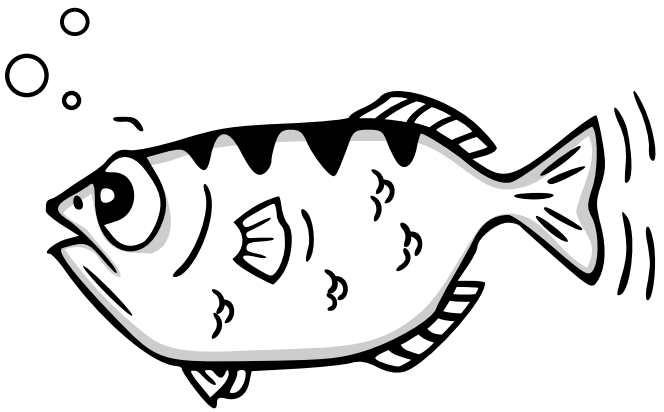
- Erlaubt das Nachverfolgen des Kontrollflusses
- Untersuchung (und Manipulation) des Programmzustandes während der Ausführung
- Untersuchung von Fehlern wie z. B. *segmentation fault*
- Bildet Threads ab und können einzeln untersucht werden
- Programm kann **jeder Zeit** mit `strg+c` unterbrochen werden

```
al@ganymed:~/coding$ gdb ./a.out
GNU gdb (Debian 13.2-1) 13.2
Copyright (C) 2023 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/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-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 ./a.out...
(gdb) run
Starting program: /home/local/al/coding/a.out
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Program received signal SIGSEGV, Segmentation fault.
__strlen_avx2 () at ../sysdeps/x86_64/multiarch/strlen-avx2.S:76
76      ../sysdeps/x86_64/multiarch/strlen-avx2.S: No such file or directory.
(gdb)
```

GNU Debugger (2/2)

- Unterstützt verschiedene Ziele (*targets*), wie z. B. i386, amd64, …
- Erweiterbar um weitere Ziele mittels GDB-Stub

  (Ziel muss nur das GDB-Protokoll implementieren)



Können wir den GNU Debugger einfach für einen Betriebssystemkern (hier: Linux) nutzen?

Nein!

Problem: GDB läuft selbst ein Anwendungsprozess und soll der Betriebssystemkern untersuchen. Wir müssen von von außen an den Kern. Aber wie?

in vivo-Untersuchung

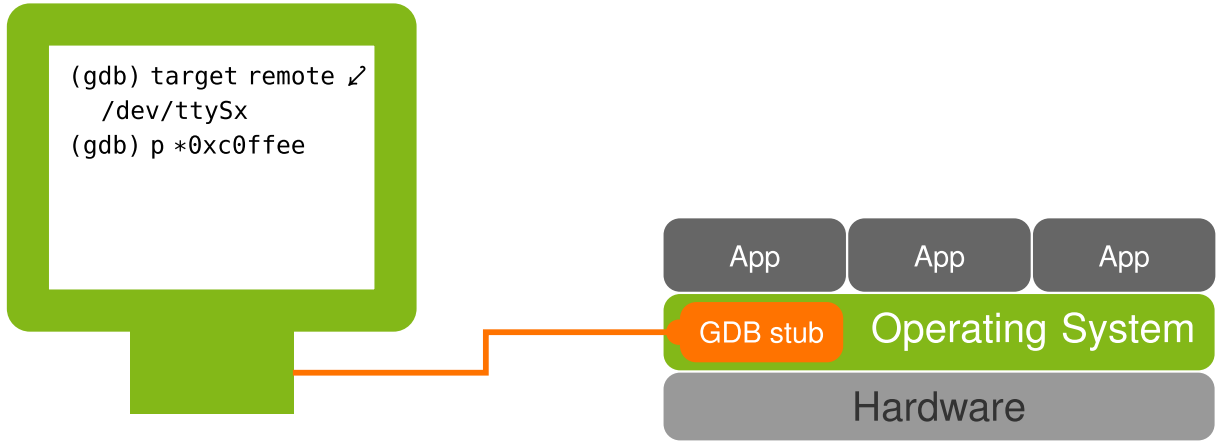
Wie kann ich einen laufenden Betriebssystemkern oder einen laufenden Computer untersuchen?

- Oszilloskop
- LED(s)
- Externer Debugger, wie z. B. der Fa. Lauterbach
- Serielle Schnittstelle

Kernel GNU Debugger [1]

- Lässt sich „einfach” über einen GDB bedienen
- Benötigt *zwei* Computer:

  Device under Test (DUT) und Entwicklungsmaschine



- Implementiert GDB-Stub über serielle Schnittstelle
- Kleine Schwester ist kdb über Tastatur am *DUT*
- Kennt auch Threads (= **alle** im System laufende Threads/Prozesse)
- **Aber** eine beliebige Unterbrechung mittels `strg+c` ist **nicht** möglich

Verwendung des *kgdb*

- Aus dem untersuchten System die Kontrolle an den GDB übergeben

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

- Kernparameter `kgdbwait`
  - Unterbricht Startvorgang des Kerns
  - Übergibt Kontrolle an GDB
  - Möglichkeit, um u. a. Breakpoints zu setzen
- Verweilt nicht zu lange im GDB, bevor Ihr `c` drückt. Ansonsten beschwert sich der Linux-Kern

  ```
  [ 116.011870][ C0] INFO: NMI handler (kgdb_nmi_handler) took
  too long to run: 50930.300 msecs
  ```

Vor- und Nachteile vom *Kernel* GNU Debugger

Pro

- Tiefen Einblick in den Zustand des BS-Kerns
- Gute Abbildung von GDB-Entitäte BS-Interna
- Unterbrechung bei Ausnahmen im BS-Kern
- Funktioniert sowohl mit emulierter als auch echter Hardware

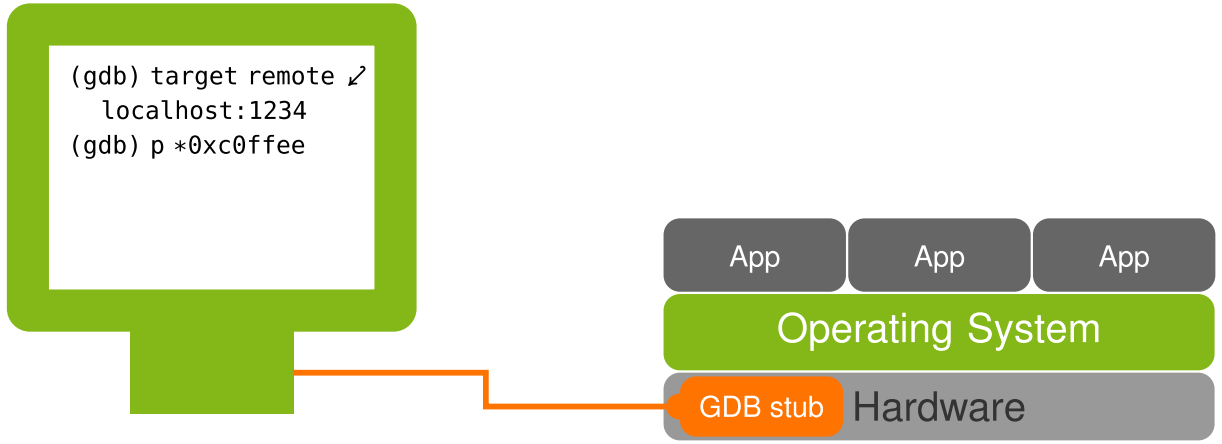
Contra

- zur Laufzeit von außen nicht unterbrechbar
- ängt von *fehlerfreier kgdb*-Komponente n Kern ab

Und nu?

QEMU-GDB-Stub

- Qemu implementiert GDB-Stub
- GDB sitzt daher in der Hardware → erinnert an ext. Debugger
- Jetzt sind beliebige Unterbrechungen (`strg+c`) möglich



- Kennt keine Threads im eigentlichen Sinne → dargestellte Threads entsprechen emulierten CPU-Kernen
- Funktioniert auch, wenn das untersuchte System steht(!)

(Geht bei uns auch mit dem Parameter `-g XX/vmlinux` für das Skript `boot.sh`)

Tracing

dtrace - Die Mutter aller Werkzeuge

- *dtrace* wurde von Sun Microsystems für Solaris entwickelt [2]
- Erlaubt die dynamische Instrumentierung von BS-Kern **und** Nutzeranwendungen
- Kern selbst wird **nicht** instrumentiert
- Daten und Anknüpfungspunkte (*probe points*) werden zur Laufzeit durch sog. Provider geladen
- Prädikate beschreiben Auslösebedingung, siehe Zeile 2

```
1 syscall::write:entry
2 /execname == "sshd" && arg0 == 5/
3 {
4         @[ustack()] = quantize(arg2);
5 }
```

Und was kann Linux?

Quellen

- *kprobes* (LWN-Artikel)
- *uprobes*
- *tracepoints*
- *function tracer*

  (GCC-Erweiterung *mcount*)

Extraktion

- *perf* (Beispiele)
- *ftrace*
- *SystemTap*
- *eBPF* (Mehr Doku)
- *LTTng*

Frontend

- perf
- *bcc*
- *kernelshark*
- siehe Extraktion

Wichtige Erkenntnis: Abwägung zwischen spezifische Informationen sind direkt verfügbar vs. universell im Kern einsetzbar

(Danke an Julia Evans für ihre schöne Einteilung und Brendan Gregg)

Datenquelle *kprobe*

- Zwei Typen
  - *kprobe*: Kann an quasi **jeder** Assemblerinstruktion wirken
  - *kretprobe*: Kann bei der Rückkehr einer Funktion wirken
- Prinzip
  1. Kopiere zu untersuchende Instruktion
  2. Ersetze zu untersuchende Instruktion mit `int3` (auf x86)
- Bei Auslösen der Trap (`int3`)
  1. Führe *pre*-Handler aus
  2. Führe ersetzte Instruktion aus
  3. Führe *post*-Handler aus

Datenquelle *kprobe*

- Optimierungen
  - Nutze `jmp`-Instruktion statt `int3`
  - Nutze *ftrace*-Infrastruktur bei Funktionen als Ziel
- Beispiel

```
1 static void __kprobes handler_pre(struct kprobe *p, struct pt_regs *regs,
2                 unsigned long flags) {
3     printk("before execution\n");
4 }
5 static void __kprobes handler_post(struct kprobe *p, struct pt_regs *regs,
6                 unsigned long flags) {
7     printk("after execution\n");
8 }
```

Datenaufbereitung

- Moderne Werkzeuge können …
  - Daten aggregieren,
  - Fehler abfangen,
  - Kern- und Anwend
  - den Mehraufwand
- Weitere Details zu WN-Artikel [3]

Weil ich „nur” SystemTap kann.
😎

SystemTap

- Ursprüngliches Konzept von Frank Ch. Eigler et al. [4]
- Nutzt verschiedene Backends:

  kprobes, *uprobes*, *ebpf*, …
- C-ähnliche Sprache zum Erstellen von Skripten
- Kapselung von häufig-genutzten Funktionen in Tapsets
- Zustand modifizieren oder echten C-Code einbauen mit dem Guru-Mode (`-g`)
- Schaut gerne in den Beginners Guide

SystemTap - Das erste Skript

```
1 probe begin {
2     printf("Starting stap...\n")
3 }
4
5 probe syscall.open {
6   printf ("%s(%d) open\n", execname(), pid())
7 }
```

- Gebe beim Start den Text „Starting stap...\n” aus
- Instrumentiere den Systemaufruf open( )
- Bei jedem Öffnen einer Datei gebe den Namen des Programms und die Prozess-ID aus

SystemTap - Übersetzung

```
root@debian:~# stap -vt -r /mnt/linux/ test.stp
Pass 1: parsed user script and 484 library scripts using 132404virt/107252res/11404shr/94996data kb, in 140usr/20sys/166real ms.
Pass 2: analyzed script: 1 probe, 4 functions, 0 embeds, 0 globals using 194428virt/170444res/12876shr/157020data kb, in 790usr/940sys/128
Pass 3: translated to C into "/tmp/stap0a74pc/stap_9bf5fa33360a9fd150b4098db40097b6_1408_src.c" using 194428virt/170636res/13068shr/157020
Pass 4: compiled C into "stap_9bf5fa33360a9fd150b4098db40097b6_1408.ko" in 1660usr/430sys/4530real ms.
Pass 5: starting run.
sshd (440) wrote 36 bytes
sshd (440) wrote 76 bytes
----- probe hit report:
kernel.function("vfs_write@fs/read_write.c:564").return, (test.stp:1:1), hits: 6, cycles: 152min/2273avg/7049max, variance: 18695040, from
----- refresh report:
Pass 5: run completed in 40usr/30sys/2789real ms.
```

- `-v`: Erhöhe das Log-Level
- `-t`: Erhebe Timing-Informationen während der Ausführung und gebe sie aus
- `-r`: Pfad zum Linux-Quellcode-Verzeichnis, hier `/mnt/linux`

SystemTap - Übersetzung

- `test.stp`

```
1 probe kernel.function("vfs_write").return {
2         if (stp_pid() == pid()) next
3         printf("%s (%d) wrote %d bytes, expected\n", execname(), pid(), $return)
4 }
```

SystemTap - Grundprinzip

- zu untersuchende Stelle (Funktion, …) spezifizieren
- Lokale Variablen, Funktionsargumente oder Rückgabewert stehen zur Verfügung
- Ausführen von beliebigen Aktionen – siehe *language reference*

```
probe kernel.function("vfs_write").return {
        if (stp_pid() == pid()) next
        printf("%s (%d) wrote %d bytes, expected %d\n", execname(), pid(), $return, @entry($count))
}
```

SystemTap - Ort spezifizieren

- Siehe *language reference* Abschnitt 4.2 „Built-in probe point types”
- Siehe syscall-Tapsets
  - Wrapper für die Systemaufrufe
  - Zugriff über z. B. `syscall.open`
- `kernel.function("foo@bar.c")` → Funktion `foo()` in Datei `bar.c`
- `kernel.function("*@bar.c")` → **jede** Funktion in Datei `bar.c`
- `kernel.statement("*@fs/namei.c:3080")` → Zeile 3080 in Datei `fs/namei.c`

SystemTap - Funktionen

- Aggregation von Daten

```
1  global bytes
2  probe kernel.function("vfs_write").return {
3          if (stp_pid() == pid() || $return < 0) next
4          bytes <<< $return
5  }
6  probe end {
7          print(@hist_linear(bytes, 0, 2048, 128))
8  }
```

Ausgabe:

```
1  value |-------------------------------------------------- count
2      0 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@    572
3    128 |@@@                                                 36
4    256 |@@@                                                 36
```

- Periodisch Aufgaben erledigen

```
1  probe timer.ms(1000) {
2      printf("%d\n", gettimeofday_s())
3  }
```

SystemTap - „guru mode”

- *guru mode* gestattet zusätzliche und invasive Funktionen (Parameter `-g`)
- Zustand sowie das Verhalten des Kerns modifzieren (Beispiele: 1, 2)

```
1 // Beispiel 1
2 probe kernel.function("do_fsync").return {
3     # override result code, just in case kernel sent back -EINVAL or somesuch
4     try { $return = 0 } catch { }
5 }
```

- C-Code einbetten (Beispiel)

```
1 probe syscall.open %{
2     int i = 1;
3     printk("i ist gleich %d\n", i);
4 %}
```

It’s your turn! 😎

Referenzen

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

[2]
B. M. Cantrill, M. W. Shaprio, und A. H. Leventhal, „Dynamic Instrumentation of Production Systems“, in *Proceedings of the 2004 USENIX Annual Technical Conference*, The USENIX Association, Juni 2004.

[3]
E. Rocca, „Comparing SystemTap and bpftrace“. April 2021. Verfügbar unter: https://lwn.net/Articles/852112/

[4]
F. Ch. Eigler *u. a.*, „Architecture of systemtap: a linux trace/probe tool“, in *Proceedings of the Linux Symposium*, 2005.