FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

FH Aachen

Fachbereich Elektrotechnik und Informationstechnik

Embedded Systems SS2022

Referat

Linux Tracing

Marcel Ochsendorf, Muhammed Parlak

Inhalt

1 Einleitung

Tracing ist die spezielle Verwendung der Protokollierung zur Aufzeichnung von Informationen über den Ausführungsablauf eines Programms. Oft werden mit eigenständig hinzugefügte Print-Messages der Code debuggt. Somit verfolgt man die Anweisungen mit einem eigenem tracing-System. Linux bringt einige eigenständige Tools mit, mit denen es möglich ist Vorgänge innerhalb von einem Embedded-System nachvollziehen und analysieren zu können. Die Linux-Tracing Funktionalität und die bestehenden Tools, welche im Linux-Kernel integriert sind, helfen so dabei bei der Identifikation von Laufzeiten, Nebenläufigkeiten und der Untersuchung von Latenzproblemen,

1.1 Relevanz

- multicore systeme unterscheid mikotrkontroller
- moderne linux systeme sind sehr komplex und bestehen aud vielen Softwaremodulen, welche untereinander interagieren
- systemnahes debugging, kann dann dabei helfen, schwer zu erkennde und nicht einfach reproduzierbare fehler aufzufinden
- auch möglichkeit bei custom driver deugging wärend des bootvorgangsb ## Tracing

2 Grundlagen

2.1 Ringbuffer

2.2 Debug-Filesystem

- möglichkeiten

```
1 #ENABLE DEBUG FS
2 $ sudo mount -t debugfs debugfs /sys/kernel/debug
```

2.3 Trace Events

Durch das Debug-Filesystem ist jetzt der Zugriff auf die Debug und insbesondere auf die Tracing-Daten möglich. Im Debug-Filesystem ist nach aktivierung der `tracing`-Ordner vorhanden. In diesem werden die verfügbaren Events in Gruppen (Ordnern) dargestellt, auf welche im späteren Verlauf reagiert werden können.

- was sind tracers

```
1 # GET TRACERS
2 $ cat /sys/kernel/debug/tracing/available_tracers
3 hwlat blk mmiotrace function_graph wakeup_dl wakeup_rt wakeup
     function nop
```

In der `ls` Ausgabe des `events`-Ordners des Debug-Filesystems ist zu sehen, welche Events abgefangen werden können und mittels der Linux-Tracing-Tools protokoliert werden können.

```
# GET AVAILABLE EVENT LIST
$ cd /sys/kernel/debug/tracing/events
$ ls -lah | awk '{print $9}'
alarmtimer
clk
cpuhp
drm
exceptions
ext4
# =>  EXT4_READPAGE, EXT4_WRITEPAGE, EXT4_ERROR,
   EXT4_FREE_BLOCKS
filelock
filemap
fs_dax
ftrace
gpio
# => GPIO_DIRECTION, GPIO_VALUE
hda
i2c
irq
net
smbus
# => READ, WRITE, REPLY
sock
# => SOCKET_STATE_CHANGED, SOCK_EXCEED_BUFFER_LIMIT,
   SOCK_REC_QUEUE_FULL
spi
tcp
timer
# => TIMER_STOP, TIMER_INIT, TIMER_EXPIRED
```

Alle Events sind in Gruppen gebündelt. Alle Events, welche das `ext4`-Filesystem betreffen, befinden sich im `ext4`-Ordner. Die Auflistung zeigt einige der für das `ext4` zur Verfügung stehenden Events. Zudem befinden sich zwei zusätzliche Dateien `enable`, `filter`in diesem Ordner. Durch diese ist es später möglich anzugeben, ob dieses Event aufgezeichnet werden soll.

```
$ cd /sys/kernel/debug/tracing/events/ext4
$ ls -lah | awk '{print $9}'

# EVENTS FOR EXT4
ext4_write_end
ext4_writepage
ext4_readpage
ext4_error

# INTERFACE FOR EVENT SETUP
enable
filter
```

```
13 format
```

Die optionale `format`-Datei kann zusätzliche Informationen bereitstellen über das, durch das Event bereitgestellt Format der Ausgabe. Das folgende Beispiel zeigt das Ausgabeformat für die Formatbeschreibung für das Scheduler-Wakeup `sched_wakeup`-Event. Somit kann nicht nur in Erfahrung gebracht werden, wann und ob das Event ausgelößt hat, sondern es können auch weitere Event-Spezifische Informationen durch das Event gemeldet werden.

```
1  $  cat /sys/kernel/debug/tracing/events/sched/sched_wakeup/
     format
2  ID: 318
3  format:
4      field:unsigned short common_type;   offset:0;   size:2;
         signed:0;
5      field:unsigned char common_flags;   offset:2;   size:1;
         signed:0;
6      field:unsigned char common_preempt_count;   offset:3;
         size:1; signed:0;
7      field:int common_pid;   offset:4;   size:4; signed:1;
8
9      field:char comm[16];    offset:8;   size:16;    signed:1;
10     field:pid_t pid;    offset:24;  size:4; signed:1;
11     field:int prio; offset:28;  size:4; signed:1;
12     field:int success;  offset:32;  size:4; signed:1;
13     field:int target_cpu;   offset:36;  size:4; signed:1;
```

2.4 Abfangen von Events

Um ein Event abfangen zu können, muss dies zuerst für die gewünschten Events aktiviert werden. Hierzu werden die Event-Interface-Dateien verwendet, welche sich in jeder Event-Gruppe befinden. Die einfachste Methode ist es, eine `1` oder `0` in die `enable`-Datei der Gruppe zu schreiben. Ein spezifisches Event kann mit der gleiche Methode aktiviert werden. Hierzu wirde die `enable`-Datei im eigentlichen Event-Ordner verwendet anstatt jene, welche ich in de Event-Gruppe befindet.

```
1 $ cd /sys/kernel/debug/tracing/events/ext4
2  # ENABLE ALL EVENTS FROM THIS GROUP
3 $ echo 1 > ./enable
4  # DISBALE ALL EVENTS
5 $ echo 0 > ./enable
6
7  # ENBABLE SPECIFIC EVENT
```

```
8 $ echo 1 > ./ext4_readpage/enable
9 $ echo 1 > ./ext4_writepage/enable
```

2.4.1 kprobes

Kprobes können dazu verwendet werden, Laufzeit und Performance-Daten des Kernels zu sammeln. Der Vorteil and diesen ist, dass diese Daten ohne Unterbrechnung der Ausführung auf CPU-Instruktions-Ebene aggregiert werden können, anders wie bei dem Debuggen eines Programms mittels Breakpoints. Ein weiterer Vorteil ist, dass das Registrieren der Kprobes dynamisch zur Laufzeit und ohne Änderungen des Programmcodes geschieht.

2.4.2 kretprobes

2.4.3 CPU-Traps

2.4.4 uprobes

- für anwendungn
- system libs

2.5 Nachteile / verscfälschung

- welche effekte können entstehen
- tracing bracuth ressoucen
- last minimieren auf traget minimieren
- nur aufzeichnen und später analysieren z.B. auf einem anderen system
- wie verhindern

3 Tools

Allgemein sind keine speziellen Programme notwending um die Laufzeiteigenschaften eines Programms aufzuzeichnen. Der Linux-Kernel bringt bereits alle nötigen Funktionalitäten mit. Jedoch gibt es Tools die eine visuelle Darstellung der aufgezeichneten Events ermöglichen.

3.1 Trace-Log Aufzeichnung

Für die Log-Aufzeichnung wird ein Ringbuffer genutzt. Das Aufzeichnen in den Ringpuffer ist Standardmäßig aktiviert.

```
1 # Disable the Recording on the ringbuffer
2 $ echo 0 > tracing on
```

Mit dem folgenden Befehl kann der Inhalt des Ringuffers, auch während einer Aufzeichung, ausgebeben werden:

```
1 $ less trace
```

Das Lesen während einer Aufzeichnung mit trace hat keinerlei Einfluss auf den Inhalt des Ringpuffers. Die Ausgabe des letzten Kommandos wird dabei in einem menschenlesbaren Format dargestellt:

Die bisheirgen Aufzeichnungen der Ereignisse können mit einem einfachen Befehl entfernt werden:

```
1 $ echo > trace
```

Um einen Überlauf an Informationen zu verhinden kann die Aufzeichnung auch konsumierend gelesen werden. Somit werden beim Lesen zeitgleich diese aus dem Ringbuffer

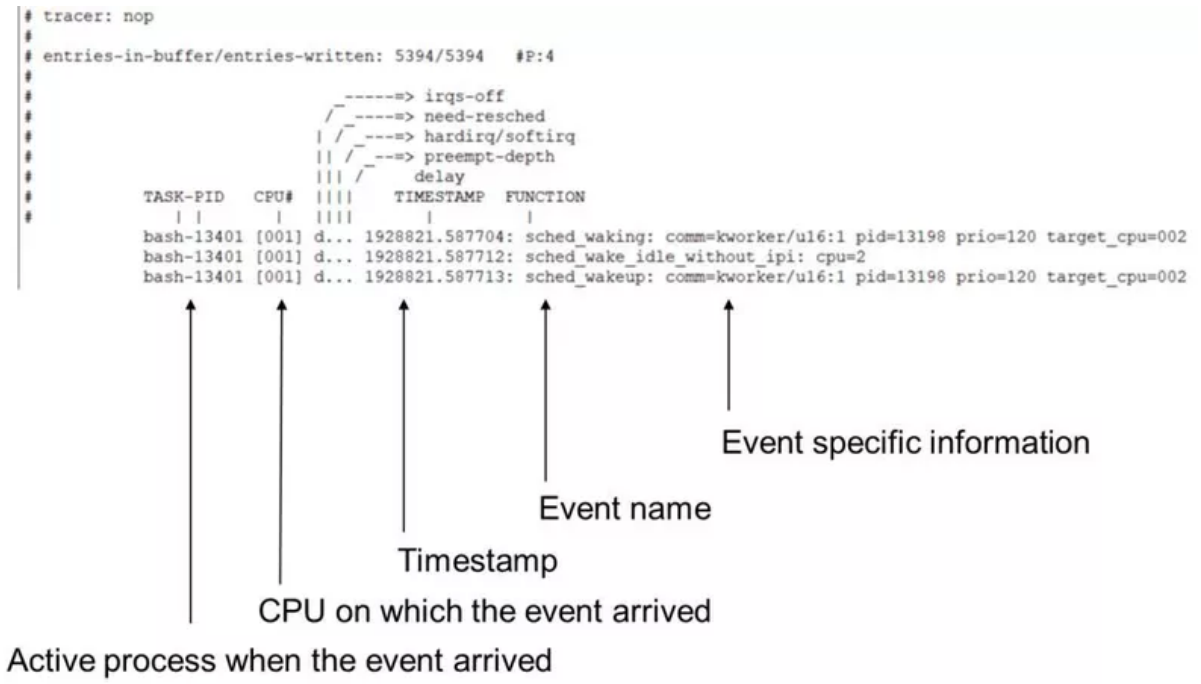

Bild **3**-1: Trace-Log

entfernt.

Eine weitere Kernpunkt ist, dass in Mehrkernsystemen für jeden einzelnen Core ein separater Ringbuffer existiert. Damit die Analyse von verschiedenen Events getrennt werden kann, kann mit jeder weiteren Instanz pro Core ein weiterer Ringbuffer angelgt werden. Dies erfolgt im Unterverezichnis `instances/`.

```
1 $ cd instances
2 $ mkdir inst0
3 $ mkdir inst1
4
5 # Remove if the instances is not needed anymore
6 # rmdir inst0
```

3.1.1 trace-cmd

Das Tool `trace-cmd` ist das bekannteste und am meisten genutzte Hilfmittel zur Aufzeichnung. Dies ist ein kommandozeilenwerkzeug, das auf den meisten gängingen Linux Distrubitionen bereits vorinstalliert ist.

```
1 # CHECK IF TRACING IS ENABLED
2 # RUN AS SUDO
3 $ mount | grep tracefs
```

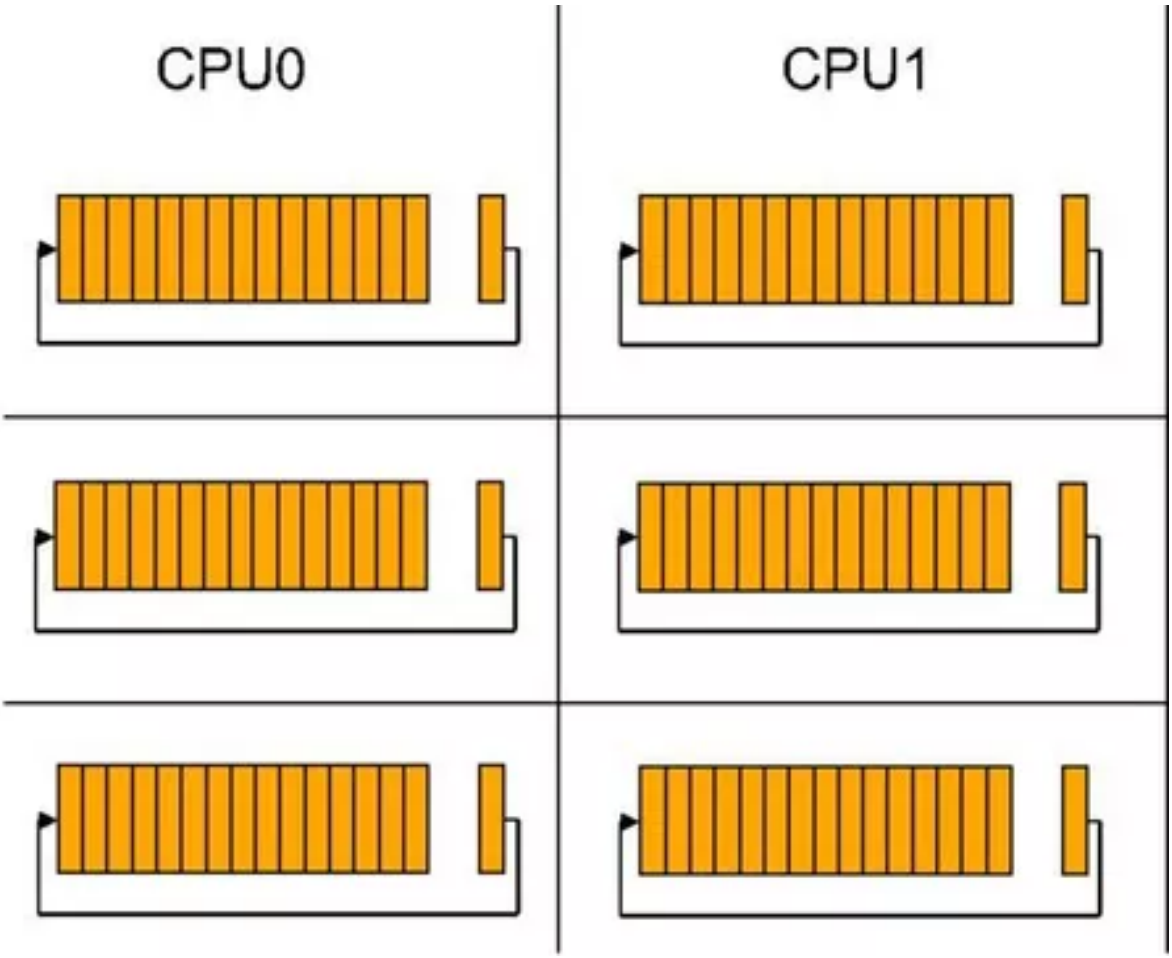

Bild **3**-2: Ringbuffer

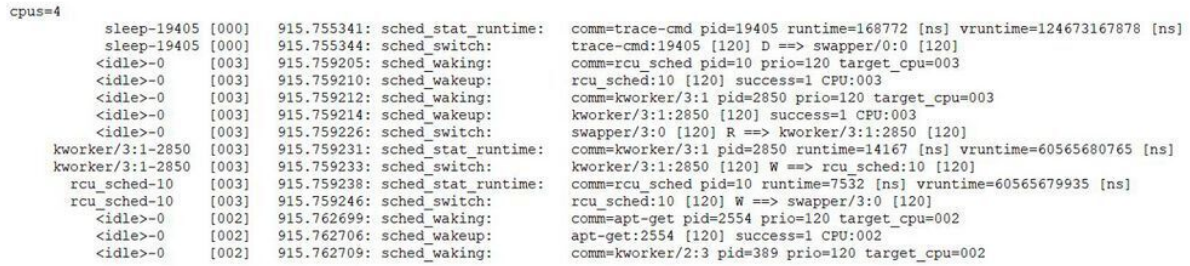

Bild **3**-3: trace-cmd Report

```
4  ## => none on /sys/kernel/tracing type tracefs (rw,relatime,
       seclabel)
5
6  # IF TRACING IS NOT ACTIVE, ENABLE IT FIRST
7
8  ## ONLY SCHEDULER EVENTS
9  $ echo sched_wakeup >> /sys/kernel/debug/tracing/set_event
10 ## ALL EVENTS
11 $ echo *:* > /sys/kernel/debug/tracing/set_event
12
13
14 # RECORD
15 $ trace-cmd record -e sched ./program_executable
```

- output erklörung

Mit dem letzten Befehl werden die ganzen Events zu Scheduler aufgezeichnet. Dabei werden während der Aufzeichnung kontinuierlich die Ringbuffer in konsumierender Form ausgelesen und in die Datei `trace.dat` geschrieben, falls mit dem `-o` keine eigene Datei eingegeben wurde. Als Informationen werden zu dem Inhalt des Ringbuffers auch zusätzlich notwendige Informationen über das Target, für die Auswertung auf beliebigen System gespeichert.

Die `trace-cmd` Konsolenanwendung dient nicht nur zur Aufzeichnung der Trace-Events, sondern bietet auch die Möglichkeit augezeichnetet Reports visuell darzustellen. Die Ausgabe erfolgt mit dem Befehl `trace-cmd report [-i <Dateiname>]` als Tabelle in der Konsole und ist somit rein Textbasiert, siehe dazu die nächste Abbildung.

Auf diese Aufzeichnung zusätzlich ein Filter angewendet werden, um die Suche auf bestimten Erreignissen einzugrenzen.

Mit dem Tool ist es einfach die Teilschritte zu automatisieren.

3.1.2 bpftrace

Seit der Kernelversion >4.x, kann ein weiteres Tool mit dem Namen bpftrace verwendet werden. Dieses bietet jedoch zusätzlich eine eigene Skripsprache mit der nicht nur Aggreation, sondern auch die Eventfilter und die Verarbeitung der Ergebnisse automatisiert werden können.

```
1  # Block I/O latency as a histogram EXAMPLE
2  $ wget https://raw.githubusercontent.com/iovisor/bpftrace/
       master/tools/biolatency.bt
3  $ bftrace ./biolatency.bt
4  # @usecs:
5  # [512, 1K)              10 |@                    |
6  # [ 1K, 2K)             426 |@@@@@@@@@@@@@@@@@@@@ |
7  # [2K, 4K)              230 |@@@@@@@@@@@@@@@@     |
8  # [4K, 8K)                9 |@                    |
9  # [8K, 16K)             128 |@@@@@@@@@@@@@@@@     |
10 # [16K, 32K)             68 |@@@@@@@@             |
11 # ...
```

3.1.3 Kernelshark

Das zuvor erklärte tace-cmd ist wie oben erwähnt nur ein textbasiertes Analysetool. Das kommende Kernelshark Tool bietet dem Anwender die Möglichkeit die Traceaufzeichnungen grafisch zu analysieren. Dabei sind die beiden Tools aufeinander abgestimmt und werden gemeinsam entwickelt. Auch dieses Tool ist in den meisten Linux Distrubutionen vorinstalliert.

Das vom trace-cmd erzeugte trace.dat-Format wird im Kernelshark als Eingabe erwartet. Wenn im folgendem ersten Befehl nichts eingegeben, dann wird nach der entsprechenden trace.dat im Verzeichnis gesucht.

```
1  $ kernelshark
2  $ kernelshark -i <Dateiname>
```

Im folgenden ist die grafische Darstellung zu sehen. Dabei besitzt jeder Task ein eigenen Farbton. Für jede CPU wird eine eigene Zeile dargestellt. Dieses Tool hat eine gewisse Ähnlichkeit mit der von uns genutzten Logic 2 Software.

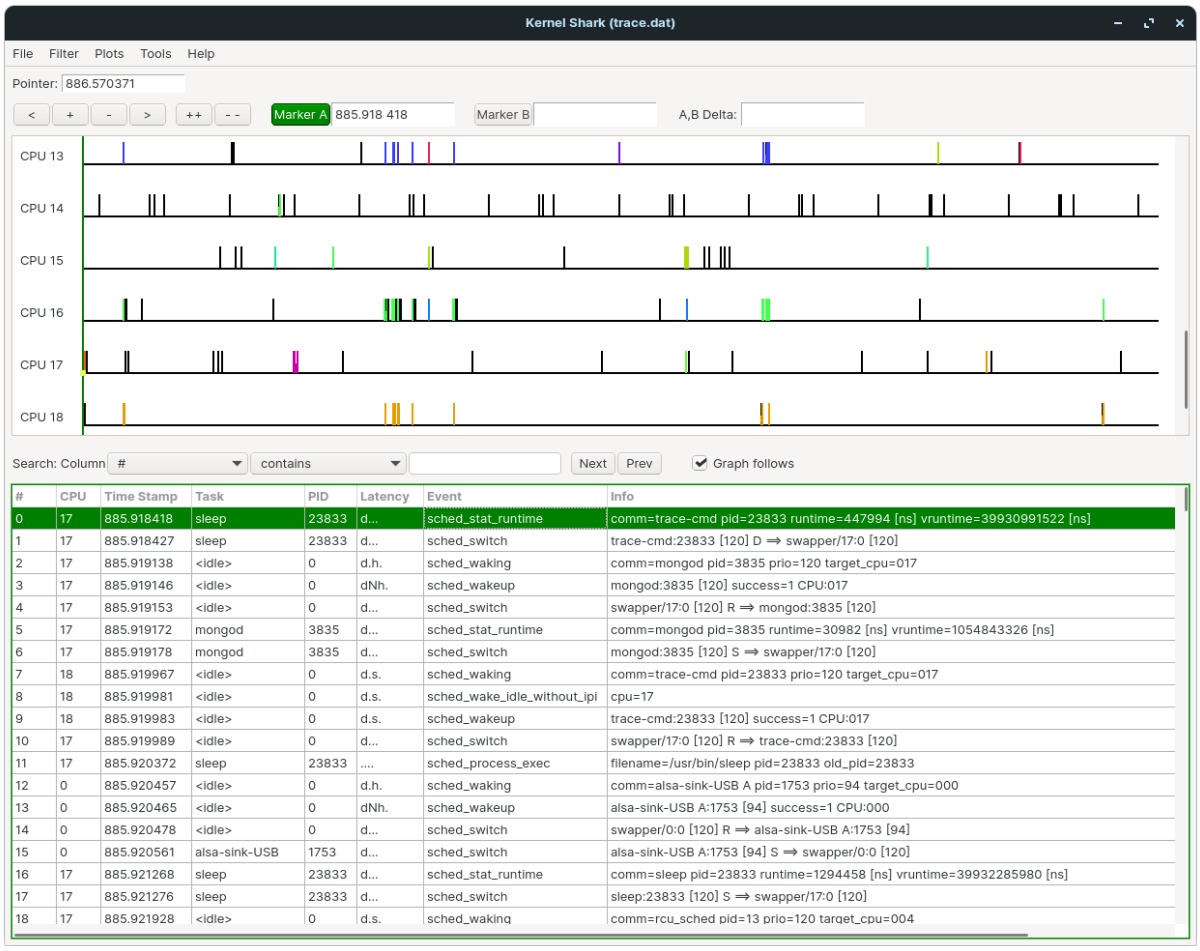

Bild **3**-4: Kernelshark

4 Beispiel - TCP Paketanalyse

Dieses Beispiel soll zeigen, wie der Empfang von TCP-Netzwerkpaketen auf Paketverlust auf einem System überprüft werden kann. Hierbei soll analysiert werden, wie das System auf eine unerwartet große Menge an TCP-Paketen reagiert.

Hierbei wird auf dem zu analysierenden System `bpftrace` verwendet. Unter Debian-Systemen kann dies einfach über den APT-Package-Manager installiert werden. Jedoch ist diese Version welche in der Registry hinterlegt ist meist nicht aktuell. Das folgende Beispiel erfodert die Version `>= 0.14`. Somit muss `bpftrace` aus den Quellen gebaut werden, da in der APT-Registry nur die Version `0.11` zur Verfügung stand.

```
# INSTALL FROM SOURCE
$ git clone https://github.com/iovisor/bpftrace ./bpftrace
$ cd ./bpftrace && mkdir -p build
$ cmake -DCMAKE_BUILD_TYPE=Release . && make -j20
$ sudo make install
# GET TCP DROP EXAMPLE
$ cd ~
$ wget https://raw.githubusercontent.com/iovisor/bpftrace/
   master/tools/tcpdrop.bt
```

Das `tcpdrop.bt` Script, welches in diesem Beispiel verwendet wird, registriert eine `kprobe` auf die `tcp_drop()` Funktion und nutzt anschließend `printf` Funktion um die Informationen in den Userspace zu loggen.

```
// tcpdrop.bt - SIMPLIFIED
kprobe:tcp_drop
{
    // GET SOCKET INFORMATION
    $sk = ((struct sock *) arg0);
    $inet_family = $sk->__sk_common.skc_family;
    //ADRESSES
    $daddr = ntop($sk->__sk_common.skc_daddr);
    $saddr = ntop($sk->__sk_common.skc_rcv_saddr);
    // PORTS
    $dport = $sk->__sk_common.skc_dport;
    $dport = $sk->__sk_common.skc_dport;
```

```
13      //LOG INTO USERSPACE
14      printf("%39s:%-6d %39s:%-6d %-10s\n", $saddr, $lport,
           $daddr, $dport, $statestr);
15  }
```

Um eine Lastspitze auf dem System zu erzeugen wurde das Netzwerkbenchmark-Tool `ntttcp` verwendet. Mit diesem ist es möglich UDP und TCP Pakete mit verschiedenen Paketgrößen zu generieren. Hierzu werden zwei Instanzen benötigt, der Server und der Client.

```
1  # START SERVER
2  $ ntttcp -r
3  NTTTCP for Linux 1.4.0
4  ---------------------------------------------------------
5  21:27:58 INFO: 17 threads created
6
7  # RUN bpftrace RECORD
8  $ sudo bpftrace -o ~/tcpdrop_log -f text -v ~/tcpdrop.bt
9  INFO: node count: 171
10 Program ID: 146
11 The verifier log:
12 processed 374 insns (limit 1000000) max_states_per_insn 0
      total_states 7 peak_states 7 mark_read 1
13 Attaching BEGIN
14
15 # START CLIENT # PACKET SIZE 4096K
16 $ ntttcp -s10.11.12.1 -t -l 4096K
17 NTTTCP for Linux 1.4.0
18 ---------------------------------------------------------
19 21:28:52 INFO: running test in continuous mode. please monitor
       throughput by other tools
20 21:28:52 INFO: 64 threads created
21 21:28:52 INFO: 64 connections created in 5656 microseconds
22 21:28:52 INFO: Network activity progressing...
```

Nach einigen Sekunden wurde `ntttcp` und `bpftrace` die Aufzeichnung manuell gestoppt. Der aufgezeichnete Trace für das `tcp_drop`-Event befindet sich in der `tcpdrop_log` Datei

```
1  $ cat ~/tcpdrop_log
2  [..]
3  # tcp_drop() TIME   PID  APPLICATION    SOURCE  DESTINATION
4  21:36:57 18157    ntttcp        10.11.12.1:5014
      10.11.12.2:59012
5          #CALLSTACK
6          # LAST FUNCTION CALL
7          tcp_drop+1
8          tcp_rcv_established+562
9          tcp_v4_do_rcv+328
```

```
10          tcp_v4_rcv+3325
11          ip_protocol_deliver_rcu+48
12          ip_local_deliver_finish+72
13          ip_local_deliver+250
14          ip_rcv_finish+182
15          ip_rcv+204
16          __netif_receive_skb_one_core+136
17          __netif_receive_skb+24
18          process_backlog+169
19          __napi_poll+46
20          net_rx_action+575
21          __do_softirq+204
22          irq_exit_rcu+164
23          sysvec_apic_timer_interrupt+124
24          asm_sysvec_apic_timer_interrupt+18
25          clear_page_erms+7
26          get_page_from_freelist+730
27          __alloc_pages+379
28          alloc_pages+135
29          skb_page_frag_refill+128
30          sk_page_frag_refill+33
31          tcp_sendmsg_locked+1049
32          tcp_sendmsg+45
33          inet_sendmsg+67
34          sock_sendmsg+94
35          __sys_sendto+275
36          __x64_sys_sendto+41
37          do_syscall_64+97
38          entry_SYSCALL_64_after_hwframe+68
39          # FIRST FUNCTION CALL
40  [...]
```

Die Ausgabe der Logdatei stellt Textbasiert nicht nur dar ob ein TCP-Paket verloren wurde, sondern gibt auch zusätzliche Informationen aus. Jeder Event-Trigger des `tcp_drop()` Events wird dabei mit der Systemzeit, Prozess-ID und dem Programm eingeleitet unter welches das Event ausgelößt hat. In diesem Fall wurde der Paketverlust durch ein Empfangenes Paket der `ntttcp`-Anwendung ausgelößt. Die Senderichtung des Pakets kann anhand der Quell- und Empfangs-IP-Adresse ermittelt werden. Danach folgt der Kernel-Stacktrace, in welchem der Funktionsaufruf-Verlauf bis zum Auslösen des überwachten Event aufgeführt ist.

Somit ist aus den Logs zu entnehmen, dass unter den getesteten Bedingungen auf dem System TCP Pakete verloren gingen, eine tiefergehende Untersuchung des Kernel-Stacktrace kann hierzu genauere Informationen bereitstellen. Das Beispiel zeigt auch, wie nicht nur das Auslösen von Events protokolliert werden kann, sondern auch mittels einfacher Script-Befehle komplexe Debug-Informationen systematisch gewonnen werden können.

5 Beispiel - Identifikation von Laufzeitproblemen

In diesem Abschnitt soll an einem einfache Beispiel gezeigt werden, wie es mittels Tracing möglich ist, eine Laufzeitanalyse auf verschiedenen Systemen für eine Anwendung durchzuführen.

5.1 Ausgangsszenario

Als Ausgangspunkt dieses Beispiels, soll das Laufzeitverhalten eines Programms auf einem Linux-System analysiert werden. Die zugrunde liegende Software wurde bisher nur auf einem Linux-Realtime Kernel verwendet, jedoch erfordert die Implementation neuer Features eine neuere Kernel-Version, welche noch nicht als RT-Version auf dem System zur Verfügung steht. Somit soll ermittelt werden, ob die unmodifizierte Software eins zu eins auf dem neuen System lauffähig ist und die Laufzeitandorderungen erfüllt.

Das System besteht hier aus einem `RaspberryPi 4B` mit einer angeschlossenen LED am GPIO-Port `25` und zu testende Programm lässt diese dabei in 100ms Abständen Blinken.

```
1  #include <iostream>
2  #include <wiringPi.h>
3  #include <csignal>
4  using namespace std;
5
6  void signal_callback_handler(int signum) {
7
8  }
9
10 int main(int argc, char *argv[])
11 {
12     //REGISTER SIGNAL HANDLER
```

```
13      signal(SIGINT, signal_callback_handler);
14
15      wiringPiSetup();              // Setup the library
16      pinMode(0, OUTPUT);      // Configure GPIO0 as an output
17      pinMode(1, INPUT);       // Configure GPIO1 as an input
18      bool state = false;
19
20      while(1)
21      {
22          state = !state;
23          digitalWrite(0, state);
24          delay(500);
25      }
26      return 0;
27  }
```

Für den Test wurde als RT Kernel die Version `4.19.59-rt23-v7l+` verwendet, welche nicht alle Funktionaltitäten des aktuellen `5.10` Kernel besitzt. In diesem fiktiven Beispiel, wird die `systemd-networking >V.248` Funktionalität für das Batman-Prokoll benötigt, welche den Grund für die Umstellung darstellt und nicht trivial in den `4.x` Kernel integriert werden kann.

Die Messungen wurden zuerst auf dem aktuellen `5.10 LTS` Kernel aufgezeichnet und im Anschluss wurde der RT-Kernel auf einem anderen System per Cross-Compilation aus dem `rpi-4.19.y-rt` Branch des `raspberrypi/linux` Repository gebaut. Dieser Schritt war notwendig, da es kein fertiges RT-Kernel Image zur Verfügung stand. Die erzeugten Dateien wurden dann auf die Boot-Partition der SD Karte geschrieben und in der `/boot/config.txt` Datei wurde der neue Kernel hinterlegt `kernel=kernel7_rt.img`.

5.2 Aufzeichnung Trace-Log

Zur Aufzeichnung des Trace-Logs wurde `trace-cmd` verwendet. Auf dem Zielsystem wurde dabei nur die Aufzeichnung vorgenommen und die Analyse der Logs erfolgte auf einem seperaten System.

5.2.1 Aktivierung der Events

```
1  $ echo 1 > /sys/kernel/debug/tracing/tracing_on
2  $ cat /sys/kernel/debug/tracing/trace
```

```
$ echo > /sys/kernel/debug/tracing/trace
```

```
trace-cmd record -e sched ./blink
```

5.3 Visualisierung und Beurteilung des Trace-Logs mittels kernelshark

Literaturverzeichnis