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 hinzugefügten Print-Messages der Code um Debug-Ausgaben erweitert. 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 Linux-System nachvollziehen und analysieren zu können. Die Linux-Tracing Funktionalität und die zur Verfügung stehenden Tools, helfen so dabei bei der Identifikation von Laufzeiten, Nebenläufigkeiten und der Untersuchung von Latenzproblemen. Hierzu sind bereits alle nötigen Tools und Funktionaltitäten im Linux-Kernel integriert.[2]

1.1 Relevanz

Bei Mikrokontrollern und meist in Zusammenhang mit Echtzeit-Betriebsystemen ist jede Aktion die Ausgeführt wird von hoher Bedeutung. Moderne Linux Systeme sind sehr komplex und bestehen aus vielen Softwaremodulen, welche auf unterschiedlichesten Weißen untereinander interagieren. Um diese Interaktionen nachzuvollziehen können und um zu verstehen wie sich Softwarekomponenten im Verbund verhalten, ist es wichtig das System Systemnah debuggen und diese analysieren zu können. Oft können Fehler repoduziert und mit solchen Anaylsen identifiziert werden. Zusätzlich besteht bei Custom Driver die Möglichkeit während des Bootvorgangs zu debuggen.

2 Grundlagen

Um die Tracing-Funktionalität auf einem Linux-System verwenden zu können, muss das System für deren Verwenung vorbereitet werden. Hierzu müssen unter das Debug-Filesytem auf dem Ziel-System aktiviert werden und die entsprechende Art des zu Tracings, entsprechen der Anwendung gewählt werden.

2.1 Ringbuffer

Bei einem Ringbuffer handelt sich um eine Datenstruktur, die das asynchrone Lesen und Schreiben von Informationen erleichtert. Der Puffer wird in der Regel als Array mit zwei Zeigern implementiert. Einem Lesezeiger und einem Schreibzeiger. Man liest aus dem Puffer, indem man den Inhalt des Lesezeigers liest und dann den Zeiger auf das nächste Element erhöht, und ebenso beim Schreiben in den Puffer mit dem Schreibzeiger. So werden in der eingesetzten Rinbbufferimplementierung die Debug-Informationen gespeichert und ein Auslesen dieser ist mittels der Einträge im Debug-Filesystem möglich.

2.2 Debug-Filesystem

Das Debug-Filesystem, wurde in der Kernel-Version `2.6.10-rc3`[2] eingeführt. Es bietet Zugriff auf Diagnose und Debug-Informationen auf Kernel-Ebene.

Ein Vorteil gegenüber des Prozess-Filesystem `/proc` ist, dass jeder Entwickler hier auch eigene Daten zur späteren Diagnose einpflegen kann. Um das Dateisystem nutzen zu können, muss dies zuerst aktiviert werden. Nach der aktivierung, stehen die Ordner unter dem angegebenen Pfad zur Verfügung.

```
# ENABLE DEBUG FS
$ sudo mount -t debugfs debugfs /sys/kernel/debug
$ ls -lah /sys/kernel/debug | awk '{print $9}'
hid
usb
tracing
[...]
```

2.3 Tracing

Durch das Debug-Filesystem ist der Zugriff auf die Debug und insbesondere auf die Tracing-Daten möglich. Im Debug-Filesystem ist nach aktivierung die `tracing`-Ordnerstruktur vorhanden. In diesem werden verfügbaren Events in gruppiert in Ordnern dargestellt, auf welche im späteren Verlauf reagiert werden kann. Zudem können hier auch die Verfügbaren `tracer` angezeigt und aktiviert werden, welche noch weitere debugging Optionen bereitstellen.

2.3.1 Tracer

```
# GET TRACERS
$ cat /sys/kernel/debug/tracing/available_tracers
hwlat blk mmiotrace function_graph wakeup_dl wakeup_rt wakeup
   function nop
# USE SPECIFIC TRACER
$ echo function_graph > /sys/kernel/debug/tracing/
   current_tracer
# DISABLE TRACER USAGE
$ echo nop > /sys/kernel/debug/tracing/current_tracer
```

`tracer` sind zusätzliche Tracing-Tools, welche eine gezieltere Aggregierung von Events z.B. Filterung und somit tiefergehende Analyse erlauben. Zum Beispiel erlaubt der `ftrace`-Tracer eine detaillierte Ereignis-Filterung auf spezifizierte Events[3].

Der `function_graph`-Tracer gibt bei Verwendung zusätzliche Informationen, wie z.B. die Laufzeit von einzelnen Funktionen[5].

Auch kann dieser den Stacktrace und den Call-Stack übersichtlich darstellen, indem hier die Namen der aufgerufenen Funktionen ausgegeben werden.

```
# CALL STACK USING FUNCTION_GRAPH TRACER
$ echo function_graph > /sys/kernel/debug/tracing/
   current_tracer
$ cat /sys/kernel/debug/tracing/trace
# tracer: function_graph
# CPU-  DURATION           FUNCTION CALLS
# |     |   |              |  |   |
 0)                |       getname() {
 0)   4.442 us     |          kmem_cache_alloc() {
 0)   1.382 us     |              __might_sleep();
 0)   2.478 us     |          }
 [...]
```

2.3.2 Events

Ein Event kann zum Beispiel durch das Lesen und Schreiben auf den System I2C-Bus vom Kernel ausgelößt werden. Wenn das Event im Debug-Filesystem aktiviert wurde, stellt dieses die Informationen des Events bereit. Je nach Typ können unterschiedlichste Informationen dem Nutzer bereitgestellt werden.

Im Beispiel des I2C-Events kann nachvollzogen werden, welche Nachrichten an eine bestimmte Adresse gesendet wurden.

```
$ mount -t debugfs none /sys/kernel/debug
$ cd /sys/kernel/debug/tracing/
$ echo nop > current_tracer
$ echo 1 > events/i2c/enable
$ echo 1 > tracing_on
$ cat /sys/kernel/debug/tracing/trace
i2c_write: i2c-5 0 a=048 f=0001 l=8
i2c_read: i2c-5 1 a=048 f=0002 l=9
```

In der `ls`-Ausgabe des `events`-Ordners des Debug-Filesystems ist zu sehen, welche Events abgefangen 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
drm
exceptions
ext4 #READPAGE, WRITEPAGE, ERROR, FREE_BLOCKS
filelock
filemap
```

```
gpio #GPIO_DIRECTION, GPIO_VALUE
hda
i2c
irq
net
smbus #READ, WRITE, REPLY
sock #STATE_CHANGED, EXCEED_BUFFER_LIMIT, 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
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 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.

```
$  cat /sys/kernel/debug/tracing/events/sched/sched_wakeup/
   format
ID: 318
format:
    field:unsigned short common_type;   offset:0;   size:2;
       signed:0;
    field:unsigned char common_flags;   offset:2;   size:1;
       signed:0;
    field:unsigned char common_preempt_count;   offset:3;
       size:1; signed:0;
    field:int common_pid;   offset:4;   size:4; signed:1;
```

```
8       field:char comm[16];    offset:8;   size:16;    signed:1;
9       field:pid_t pid;    offset:24;  size:4; signed:1;
10      field:int prio; offset:28;  size:4; signed:1;
11      field:int success;  offset:32;  size:4; signed:1;
12      field:int target_cpu;   offset:36;  size:4; signed:1;
```

2.4 Abfangen von Events

Um ein `event`[10] 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 # ENBABLE SPECIFIC EVENT
7 $ echo 1 > ./ext4_readpage/enable
8 $ echo 1 > ./ext4_writepage/enable
```

Nach dem Aktivieren der Events, können diese z.B. `Trace-Log` Aufgezeichnet oder anderweitig ausgegeben werden.

2.5 Probes

2.5.1 Kernel-Probes

`kprobes`[4] 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. Somit ist es möglich zu verschiedenen Laufzeiten des zu analysierenden Systems oder Programms, Daten zu verschiedenen Laufzeiten gezielt sammen zu können.

Der kretprobes[1] ermöglicht uns auf den Rückgabewert jeder Kernel- oder Modulfunktion zuzugreifen! Die Möglichkeit, den Rückgabewert einer bestimmten Funktion dynamisch nachzuschlagen, kann in einem Debug-Szenario ein entscheidender Vorteil sein.

2.5.2 User-Level-Probes

Eine Weiterentwicklung zu den kprobes sind die uprobes. Mit diesem können zur Laufzeit Events in eine Applikation eingebunden werden. Wenn ein uprobes hinzugefügt werden soll, muss davor noch was gemacht werden.

```
1 //test.c
2 #include <stdio.h>
3 int main(void)
4 {
5     int i;
6     for (i = 0; i < 5; i++)
7         printf("Hello uprobe\n");
8     return 0;
9 }
```

Bei der Nutzung von kprobes kann ein einfacher Symbolnamen spezifiziert werden. Aufgrund das alle Applikationen ihren eigenen virtuellen Adressraum besitzen, haben diese auch einen anderen Adressbasis. Beim Erzeugen eines uprobes wird das Adressoffset im Textsegment der jeweiligen Applikation benötigt. Der obere C++-Code, stellt ein einfaches Beispiel dar, indem die printf-Anweisung, mittels einer uprobe aufgezeichnet werden soll. Der Adressoffset kann mittels objdump und dem Pfad des zu analysierenden Programms. Danach kann die uprobe im Debuf-Filesystem registriert werden, unter Angabe des Offsets. Als letzer Schritt, muss das neu erstellte uprobe-Event noch aktiviert werden und die Aufzechnung oder Ausgabe der uprobe.

```
1 # CREATE EXECUTE OBJECT
2 $ gcc ./test.c -o ./tmp/test
3 # GET OFFSET
4 $ objdump -F -S -D ./test | less | grep main
5 0000000000001149 <main> (File Offset: 0x1149):
6 # REGISTER A uprobe_event
```

```
$ echo "p:my_uprobe /tmp/test:0x1149" > /sys/kernel/debug/
   tracing/uprobe_events
# ACTIVATE UPROBE EVENTS
$ echo 1 > /sys/kernel/tracing/events/uprobes/enable
# EXECUTE PROGRAM
$ /root/hello
Hello uprobe
[...]
# PRINT TRACED EVENTS
$ cat /sys/kernel/debug/tracing/trace
# tracer: nop
# TASK-PID   CPU#   TIMESTAMP   FUNCTION
#  |    |      |         |          |
test-24842 [006] 258544.995456: printf: [...]
[...]
```

Ein weiterer Anwendungsfall ist die Inspektion von System-Bibliotheken.

2.6 Ressourcen

Beim tracing werden zusätzliche ressourcen benötigt, die Auswirkungen auf die reale Ausführzeit haben. Bei Realtime OS können diese problemeatisch werden, wenn diese bereits mit dem maximalen ressourcen arbeitet.

- 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
- grosse dateigrößen bei langen logs

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. Somit kann die aufzeichnung headless auf dem Ziel-System geschehen und die spätere Analyse mit entsprechenden Tools auf einem anderen System.

3.1 Trace-Log Aufzeichnung

Für die Log-Aufzeichnung wird der zuvor beschriebene Ringbuffer genutzt. Das Aufzeichnen in den Ringpuffer ist standardmäßig aktiviert. Kann aber bei Bedarf deaktiviert werden.

```
1 $ echo 1 > tracing on
2 $ echo 0 > tracing on
```

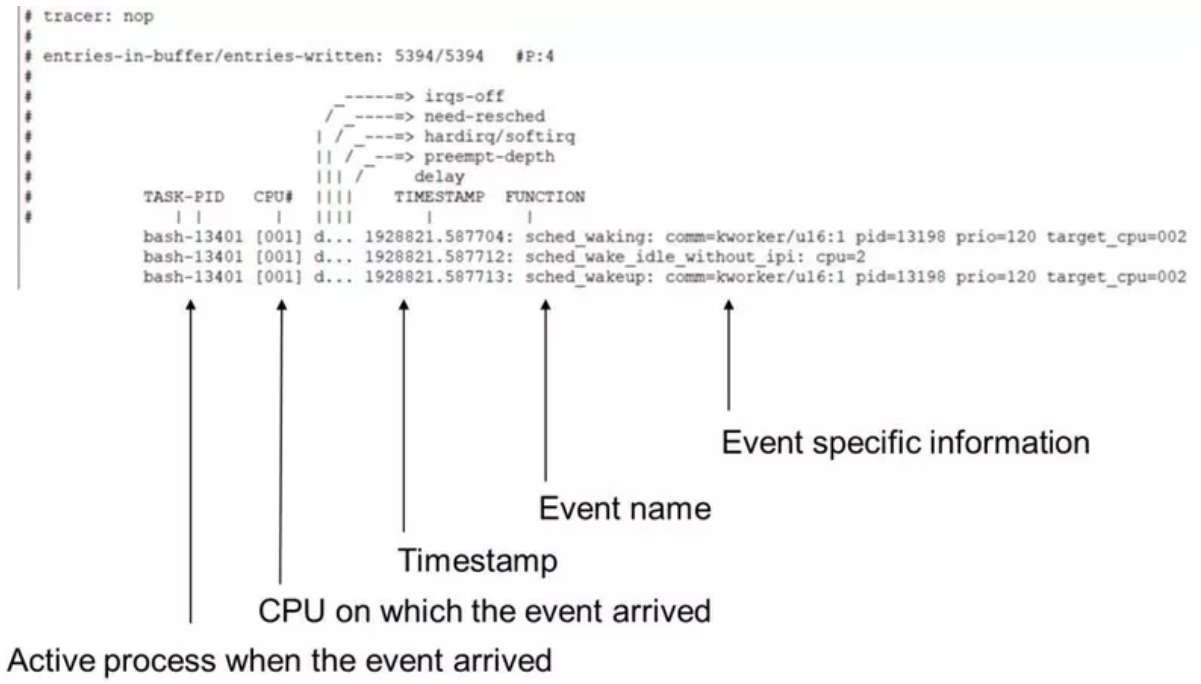
Mit dem folgenden Befehl kann der Inhalt des Ringuffers, auch während einer Aufzeichung, ausgebeben werden. Somit sind im Allgemeinen keine besonderen Tools notwendig. Anwendungen zum Ausgeben von Dateien wie z.B.`cat` oder `less`, welche sich auch auf kleinen Systemen befinden reichen aus.**3**-1

```
1 $ less /sys/kernel/tracing/trace
```

Das Lesen während einer Aufzeichnung mit trace hat keinerlei Einfluss auf den Inhalt des Ringpuffers.

Die bisheirgen Aufzeichnungen der Ereignisse können mit dem leeren der `trace`-Datei entfernt werden:

```
1 $ echo > trace
```




```
# tracer: nop
#
# entries-in-buffer/entries-written: 5394/5394   #P:4
#
#                              _-----=> irqs-off
#                             / _----=> need-resched
#                            | / _---=> hardirq/softirq
#                            || / _--=> preempt-depth
#                            ||| /     delay
#           TASK-PID   CPU#  ||||    TIMESTAMP  FUNCTION
#              | |       |   ||||       |         |
          bash-13401 [001] d... 1928821.587704: sched_waking: comm=kworker/u16:1 pid=13198 prio=120 target_cpu=002
          bash-13401 [001] d... 1928821.587712: sched_wake_idle_without_ipi: cpu=2
          bash-13401 [001] d... 1928821.587713: sched_wakeup: comm=kworker/u16:1 pid=13198 prio=120 target_cpu=002
```

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

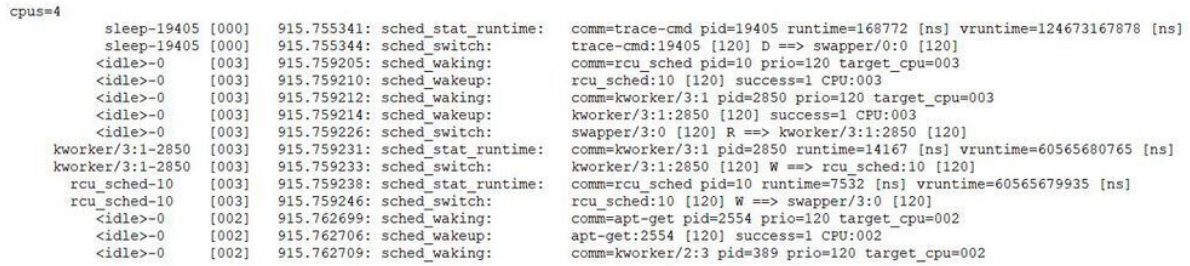
Um einen Überlauf an Informationen zu verhinden kann die Aufzeichnung auch konsumierend gelesen werden. Somit werden beim Lesen zeitgleich diese aus dem Ringbuffer entfernt.

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/`. Das Debug-Filesystem legt nach dem Anlegen des Ordners, die benötigten Dateien wie die `trace`-Datei automatisch an. Alle weiteren Operationen können dann auch auf dieser ausgeführt werden.

```
1 $ cd  /sys/kernel/tracing/instances
2 $ mkdir ./inst0
3 $ mkdir ./inst1
```

3.1.1 trace-cmd

Das Tool `trace-cmd`[7] ist das bekannteste und am meisten genutzte Hilfmittel zur Aufzeichnung. Dies ist ein Kommandozeilenwerkzeug, das auf den meisten gängingen Linux-Distrubitionen bereits vorinstalliert ist.



```
cpus=4
          sleep-19405 [000]   915.755341: sched_stat_runtime:   comm=trace-cmd pid=19405 runtime=168772 [ns] vruntime=124673167878 [ns]
          sleep-19405 [000]   915.755344: sched_switch:         trace-cmd:19405 [120] D ==> swapper/0:0 [120]
          <idle>-0    [003]   915.759205: sched_waking:         comm=rcu_sched pid=10 prio=120 target_cpu=003
          <idle>-0    [003]   915.759210: sched_wakeup:         rcu_sched:10 [120] success=1 CPU:003
          <idle>-0    [003]   915.759212: sched_waking:         comm=kworker/3:1 pid=2850 prio=120 target_cpu=003
          <idle>-0    [003]   915.759214: sched_wakeup:         kworker/3:1:2850 [120] success=1 CPU:003
          <idle>-0    [003]   915.759226: sched_switch:         swapper/3:0 [120] R ==> kworker/3:1:2850 [120]
    kworker/3:1-2850  [003]   915.759231: sched_stat_runtime:   comm=kworker/3:1 pid=2850 runtime=14167 [ns] vruntime=60565680765 [ns]
    kworker/3:1-2850  [003]   915.759233: sched_switch:         kworker/3:1:2850 [120] W ==> rcu_sched:10 [120]
      rcu_sched-10    [003]   915.759238: sched_stat_runtime:   comm=rcu_sched pid=10 runtime=7532 [ns] vruntime=60565679935 [ns]
      rcu_sched-10    [003]   915.759246: sched_switch:         rcu_sched:10 [120] W ==> swapper/3:0 [120]
         <idle>-0     [002]   915.762699: sched_waking:         comm=apt-get pid=2554 prio=120 target_cpu=002
         <idle>-0     [002]   915.762706: sched_wakeup:         apt-get:2554 [120] success=1 CPU:002
         <idle>-0     [002]   915.762709: sched_waking:         comm=kworker/2:3 pid=389 prio=120 target_cpu=002
```

Bild **3**-2: trace-cmd Report

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.

```
1  # CHECK IF TRACING IS ENABLED
2  $ sudo mount | grep tracefs
3  none on /sys/kernel/tracing type tracefs (rw,relatime,seclabel
       )
4  ## ONLY SCHEDULER EVENTS
5  $ echo sched_wakeup >> /sys/kernel/debug/tracing/set_event
6  ## ALL EVENTS USING set_event
7  $ echo *:* > /sys/kernel/debug/tracing/set_event
8  # RECORD
9  $ trace-cmd record ./program_executable
10 # RECORD SPECIFIC EVENT
11 $ trace-cmd record -e sched ./program_executable
12 # USING A ADDITIONAL TRACER
13 $ trace-cmd -t function ./program_executable
```

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**3**-2. Auf Aufzeichnungen können zusätzliche 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[9] 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.

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

3.1.3 Kernelshark

Das zuvor vorgestellte tace-cmd ist wie oben erwähnt nur ein textbasiertes Analysetool. 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 Distributionen 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.

```
# OPEN KERNELSHARK WITH trace.dat
$ kernelshark
# OPEN KERNELSHAR WITH SPECIFIED TRACELOG
$ kernelshark -i <Dateiname>
```

Im folgenden ist die grafische Darstellung**3**-3 zu sehen. Dabei besitzt jeder Task ein eigenen Farbton. Für jede CPU wird eine eigene Zeile dargestellt.

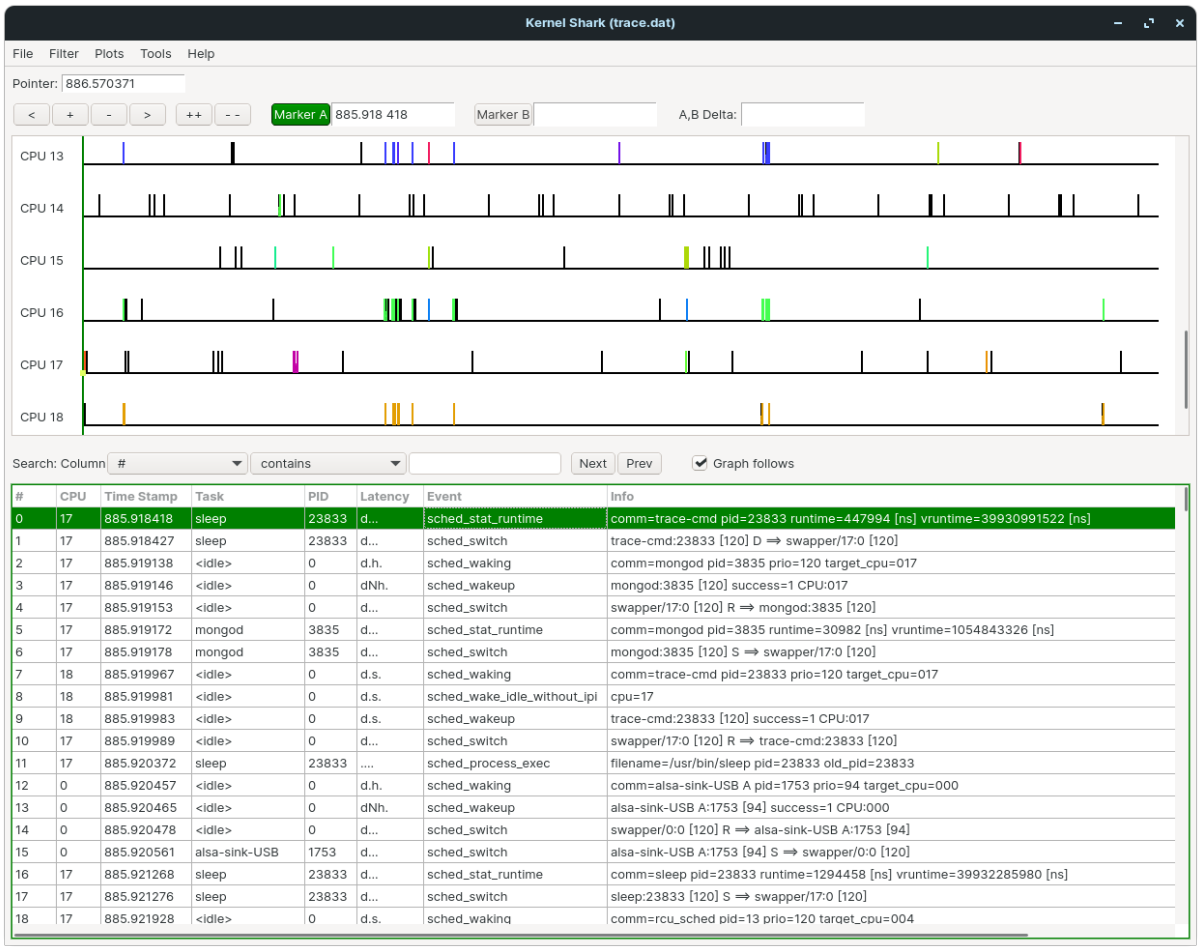


Bild **3**-3: Kernelshark

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. Dies kann zum Beispiel bei IOT-Anwendungen der Fall sein, bei denen das MQTT-Protokoll verwendet wird. Hierbei können viele kleine Netzwerkpakete von IOT-Sensoren, einen starken Traffik am Server zur folge haben.

4.1 bpftrace Installation

Hierbei wird auf dem zu analysierenden System `bpftrace`[9] 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.

```
1 # INSTALL FROM SOURCE
2 $ git clone https://github.com/iovisor/bpftrace ./bpftrace
3 $ cd ./bpftrace && mkdir -p build
4 $ cmake -DCMAKE_BUILD_TYPE=Release . && make -j20
5 $ sudo make install
6 # GET TCP DROP EXAMPLE
7 $ cp ./bpftrace/tools/tcpdrop.bt ~/
```

4.2 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;
    //LOG INTO USERSPACE
    printf("%39s:%-6d %39s:%-6d %-10s\n", $saddr, $lport,
        $daddr, $dport, $statestr);
}
```

Um eine Lastspitze auf dem System zu erzeugen wurde das Netzwerkbenchmark-Tool `ntttcp`[6] 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, welche auf dem gleichen System aber auch auf verschiedenen Systemen ausgeführt werden können.

4.3 Aufzeichnung Trace-Log

Um die Messung zu starten, wurde zuerst der `ntttcp`-Server gestartet, dieser empfängt die vom Sender gesendeten Pakete. Im zweiten Schritt wurde der `ntttcp`-Client auf dem anderen System gestartet. Hier wurde mittels `-t` Parameter die Laufzeit auf unendlich gestellt, somit werden durchgehend Pakete an den Server gesendet. Die Paketgröße wurde hier auf `4096Kbyte` gestellt um so eine Fragmentierung des TCP-Paketes bei einer MTU von `1500byte` zu erzwingen.

Im Anschluss wurde `bpftrace` gestartet, welches die Events als Logdatei `tcpdrop_log` in einem lesabren Textformat ausgeben soll.

```
# START SERVER
$ ntttcp -r
NTTTCP for Linux 1.4.0
-----------------------------------------------------------
21:27:58 INFO: 17 threads created

```

```
 7
 8 # RUN bpftrace RECORD
 9 $ sudo bpftrace -o ~/tcpdrop_log -f text -v ~/tcpdrop.bt
10 INFO: node count: 171
11 Program ID: 146
12 The verifier log:
13 processed 374 insns (limit 1000000) max_states_per_insn 0
14 Attaching BEGIN
15 [...]
16
17 # START CLIENT # PACKET SIZE 16Byte
18 $ ntttcp -s10.11.12.1 -t -l 16
19 NTTTCP for Linux 1.4.0
20 ---------------------------------------------------------
21 21:28:52 INFO: running test in continuous mode.
22 21:28:52 INFO: 64 threads created
23 21:28:52 INFO: 64 connections created in 5656 microseconds
24 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.

4.4 Ausgabe

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.

```
 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_v4_do_rcv+196
 9         __release_sock+120
10         __tcp_close+444
```

```
11          tcp_close+37
12          inet_release+72
13          __sock_release+66
14          sock_close+21
15          __fput+156
16          ____fput+14
17          task_work_run+112
18          exit_to_user_mode_prepare+437
19          syscall_exit_to_user_mode+39
20          do_syscall_64+110
21          entry_SYSCALL_64_after_hwframe+68
22          # FIRST FUNCTION CALL
23  [...]
```

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 `24` und zu testende Programm toggelt dabei den GPIO in einer Dauerschleife.

```
1  //gpio_test.cpp
2  #include <iostream>
3  #include <pigpio.h>
4  #include <csignal>
5  using namespace std;
6
7  volatile bool running = true;
8  const int GPIO = 24;
9  void signal_callback_handler(int signum) {
10     running = false;
11 }
12
```

```
13 int main(int argc, char *argv[])
14 {
15     //REGISTER SIGNAL HANDLER
16     signal(SIGINT, signal_callback_handler);
17
18     if (gpioInitialise() < 0){
19         return -1;
20     }
21     gpioSetMode(GPIO, PI_OUTPUT);
22     bool state = false;
23     while(running){
24         state = !state;
25         gpioWrite(GPIO, (int)state);
26     }
27     return 0;
28 }
```

Das Programm kann mittels `g++` Compiler für das Zielsystem übersetzt werden. Da zur Ansteuerung des GPIO Ports die `pigpio` Bibliothek verwendet wurde, welche eine Alternative zur obsoltenen `WiringPi` Bibliothek darstellt, muss diese noch installiert werden.

```
1 # INSTALL PIGPIO LIB
2 $ git clone https://github.com/joan2937/pigpio.git ~/pigpio
3 $ cd ~/pigpio && make && sudo make install && cd ~
4 # COMPILE
5 $ g++ -Wall -pthread -o gpio_test gpio_test.cpp -lpigpio -lrt
6 # TEST
7 $ sudo ./gpio_test
```

5.2 Kernel des Testsystems

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-Protokoll 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[8] 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 installiert

kernel=kernel7_rt.img.

5.3 Aufzeichnung Trace-Log

Zur Aufzeichnung des Trace-Logs wurde trace-cmd[7] verwendet. Auf dem Zielsystem wurde dabei nur die Aufzeichnung vorgenommen und die Analyse der Logs erfolgte auf einem seperaten System. Für den Test wird zuerst die tracing-Funktionalität aktiviert und alle sched und gpio-Events aktiviert.

```
1  # ENABLE TRACING
2  $ echo 1 > /sys/kernel/debug/tracing/tracing_on
3  $ cat /sys/kernel/debug/tracing/trace
4  # CLEAR RECENT EVENT LOG
5  $ echo > /sys/kernel/debug/tracing/trace
6  # ENABLE ALL SCHEDULER EVENTS
7  echo 1 > /sys/kernel/debug/tracing/events/sched/enable
8  # ENABLE ALL GPIO EVENTS
9  echo 1 > /sys/kernel/debug/tracing/events/gpio/enable
10 # RUN TEST
11 sudo trace-cmd record -e sched -e gpio -o ./
      gpio_test_trace_lts ./gpio_test
```

Im Anschluss wurde das Programm gestartet und die Events mittels trace-cmd aufgezeichnet. Da das Testprogramm nicht automatisch terminiert (wie z.B. sleep 5), muss dieses mittels Ctl+C manuell beendet werden. Die resultierende Ausgabedatei gpio_test_trace_* enthält die von trace-cmd geloggten Daten.

5.4 Auswertung

Literaturverzeichnis

[1] BILLIMORIA, Kaiwan N.: *Linux Kernel Debugging*. Packt Publishing, 2022 https://subscription.packtpub.com/book/cloud-and-networking/9781801075039/5/ch05lvl1sec33/kprobes-limitations-and-downsides. – ISBN 1801075034

[2] CORBET, Jonathan: *DebugFS*. https://www.kernel.org/doc/html/latest/filesystems/debugfs.html. Version: 01.05.2022

[3] EA, Ulia: *trace: trace your kernel functions!* https://jvns.ca/blog/2017/03/19/getting-started-with-ftrace/. Version: 01.05.2022

[4] JIM KENISTON, Masami H. Prasanna S Panchamukhi P. Prasanna S Panchamukhi: *Kernel Probes (Kprobes)*. https://docs.kernel.org/trace/kprobes.html. Version: 01.05.2022

[5] KAMATHE, Gaurav: *Analyze the Linux kernel with ftrace*. https://opensource.com/article/21/7/linux-kernel-ftrace. Version: 01.05.2022

[6] MICROSOFT: *NTTTCP-for-Linux*. https://github.com/microsoft/ntttcp-for-linux. Version: 01.05.2022

[7] RED HAT, Inc: *trace-cmd*. https://man7.org/linux/man-pages/man1/trace-cmd.1.html. Version: 01.05.2022

[8] RIVA, Mauro: *Real-Time System using Preempt-RT (kernel 4.19.y)*. https://lemariva.com/blog/2019/09/raspberry-pi-4b-preempt-rt-kernel-419y-performance-test. Version: 01.05.2022

[9] ROBERTSON., Alastair: *bpftrace*. `https://github.com/iovisor/bpftrace`. Version: 01.05.2022

[10] THEODORE TSO, Li Z. ; ZANUSSI, Tom: *Event Tracing*. `https://docs.kernel.org/trace/events.html`. Version: 01.05.2022

Abbildungsverzeichnis