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 eigenen Tracing-System.

Linux bietet einige eigenständige Tools, welche ermöglichen, Vorgänge innerhalb eines Linux-Systems nachzuvollziehen und analysieren zu können. Die Linux-Tracing Funktionalität und die zur Verfügung stehenden Tools helfen so bei der Identifikation von Laufzeiten, Nebenläufigkeiten und der Untersuchung von Latenzproblemen. Hierzu sind bereits alle nötigen Tools und Funktionalitäten im Linux-Kernel integriert. [2]

1.1 Relevanz

Bei Mikrokontrollern und auch im Zusammenhang mit Echtzeit-Betriebssystemen ist jede Aktion, die Ausgeführt wird, von hoher Bedeutung. Moderne Linux Systeme sind sehr komplex und bestehen aus vielen Softwaremodulen, welche auf unterschiedlichsten Weisen untereinander interagieren. Um diese Interaktionen nachzuvollziehen können und um zu verstehen, wie sich Softwarekomponenten im Verbund verhalten, ist es wichtig, das System systemnah zu debuggen, um diese analysieren zu können. Oft können Fehler reproduziert und mit solchen Analysen identifiziert werden. Zusätzlich besteht auch die Möglichkeit während des Bootvorgangs zu debuggen, dies kann zum Beispiel bei Kernel-Modulen hilfreich sein.

2 Grundlagen

Um die Tracing-Funktionalität auf einem Linux-System verwenden zu können, muss das System für deren Verwendung vorbereitet werden. Hierzu muss unter anderem das Debug-Filesystem (FS) auf dem Ziel-System aktiviert werden und die entsprechende Art des Tracings entsprechend der Anwendung gewählt werden.

2.1 Ringbuffer

Bei einem Ringbuffer handelt sich um eine Datenstruktur, die das asynchrone Lesen und Schreiben von Informationen erleichtert und ist die Basis für das Debug-FS 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 Ringbuffer-Implementierung die Debug-Informationen gespeichert und ein Auslesen dieser ist mittels der Einträge im Debug-FS möglich.

2.2 Debug-Filesystem

Das Debug-FS 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-FS `/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.

```
1 # ENABLE DEBUG FS
```

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

2.3 Tracing

Durch das Debug-FS ist der Zugriff auf die Debug und insbesondere auf die Tracing-Daten möglich. Im Debug-FS ist nach der Aktivierung die `Tracing`-Ordnerstruktur vorhanden. In diesem werden verfügbaren Events in gruppiert in Ordnern dargestellt, auf welche im späteren Verlauf reagiert und aufgezeichnet werden können. 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[4].

Der `function_graph`-Tracer gibt bei Verwendung zusätzliche Informationen, wie z.B. die Laufzeit von einzelnen Funktionen[6] aus.

Auch kann dieser den Stacktrace und den Call-Stack übersichtlich darstellen, indem hier zusätzlich 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 Inter-Integrated Circuit (I2C)-Bus vom Kernel ausgelöst werden. Wenn das Event im Debug-FS aktiviert wurde, stellt dieses die Informationen des Events bereit. Je nach Typ können unterschiedlichste Informationen dem Nutzer bereitgestellt werden. In der `ls`-Ausgabe des `events`-Ordners des Debug-FS ist zu sehen, welche Events abgefangen und mittels der Linux-Tracing-Tools3 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`-FS 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 diese Event-Gruppe aufgezeichnet werden soll oder nur spezifische.

```
1  $ cd /sys/kernel/debug/tracing/events/ext4
2  $ ls -lah | awk '{print $9}'
3  # EVENTS FOR EXT4
4  ext4_write_end
5  ext4_writepage
6  ext4_readpage
7  ext4_error
8  [...]
9  # INTERFACE FOR EVENT SETUP
10 enable
11 filter
12 format
```

Die optionale `format`-Datei kann zusätzliche Informationen geben, wie durch das Event bereitgestellte Daten in der Ausgabe zu interpretieren sind. 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 ausgelöst 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      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 gleichen Methode aktiviert werden. Hierzu wird die enable-Datei im eigentlichen Event-Ordner verwendet anstatt jene, welche sich in der 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
9  # GET EVENT LOG
10 $ cat /sys/kernel/debug/tracing/trace
```

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

2.5 Probes

2.5.1 Kernel-Probes

kprobes[5] können dazu verwendet werden, Laufzeit und Performance-Daten des Kernels zu sammeln. Der Vorteil dieser ist, dass Daten ohne Unterbrechung der Ausführung auf Prozessor-Instruktions-Ebene aggregiert werden können, anders als 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 sammeln zu können. Die kretprobes[1] ermöglichen es 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[3]. Mit diesen können zur Laufzeit Events in einer Applikation an Instruktionen registriert werden. Diese können zum Beispiel beim Aufruf von Funktionen oder Instruktionen in einem Programm registriert werden. Wenn das Programm ausgeführt wird, wird im Trace-Log mittels der zuvor registrierten uprobes, jeder Aufruf festgehalten.

```
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 der Tastsache, dass alle Applikationen ihren eigenen virtuellen Adressraum besitzen, haben diese auch eine andere 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 Debug-FS registriert werden unter Angabe des Offsets. Als letzter Schritt, muss das neu erstellte uprobe-Event noch aktiviert werden und die Aufzeichnung oder Ausgabe der uprobe.

```
1  # BUILD APPLICATION
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 uprobe_event
7  $ echo "p:my_uprobe /tmp/test:0x1149" > /sys/kernel/debug/
       tracing/uprobe_events
8  # ACTIVATE UPROBE EVENTS
9  $ echo 1 > /sys/kernel/tracing/events/uprobes/enable
10 # EXECUTE PROGRAM
11 $ /root/hello
12 Hello uprobe
13 [...]
14 # PRINT TRACED EVENTS
15 $ cat /sys/kernel/debug/tracing/trace
16 # tracer: nop
17 # TASK-PID   CPU#   TIMESTAMP   FUNCTION
```

```
18 #  |    |      |          |           |
19 test-24842 [012] 258544.995456: printf: [...]
20 [...]
```

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 Echtzeitbetriebsystemen können diese zu Problemen führen, wenn dieses bereits mit den maximalen Ressourcen arbeitet.

Die Aufzeichnung eines Trace-Logs benötigt je nach aktivierten Events, eine nicht unerhebliche Menge an Speicherplatz auf dem System. Zusätzlich muss das Medium auf welchem die Logs gespeichert werden sollen, ein Mindestmaß an Bandbreite zur verfügung stellen.

Wird zusätzlich die Auswertung auf dem zu analysierenden Gerät durchgeführt, benötigt diese weitere Ressourcen und kann somit den Betrieb und die Aufzeichnung beeinflussen.

Um diese nachteiligen Effekte zu minimieren, sollte die Auswertung auf einem seperaten Gerät durchgeführt werden. Vor der Aufzeichnung sollen nur die Events aktiviert werden, welche im Fokus der analyse stehen. Somit kann Bandbreite und Speicherplatz verringert werden, welche die zu analysierende Anwendung ggf. selbst benötigt.

3 Tools

Allgemein sind keine speziellen Programme notwendig, um die Laufzeiteigenschaften eines Programms aufzuzeichnen. Der Linux-Kernel bringt bereits alle nötigen Funktionalitäten mit, jedoch gibt es Tools, welche 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 erfolgen.

3.1 Trace-Log Aufzeichnung

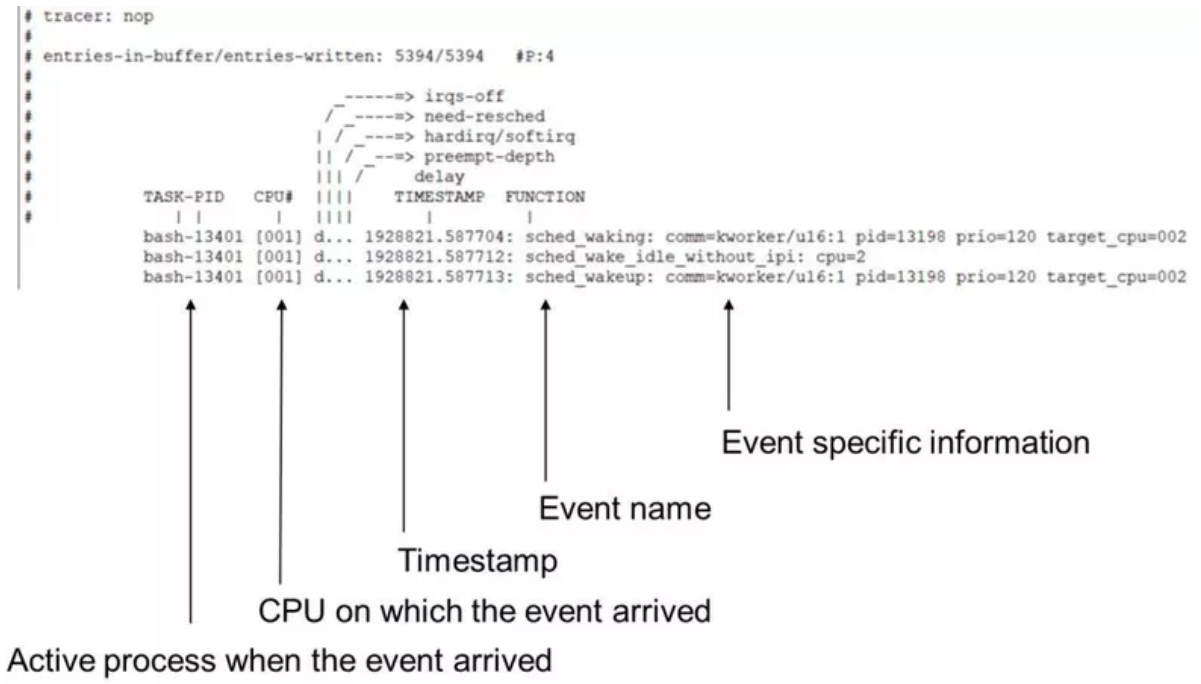
Für die Log-Aufzeichnung wird der zuvor beschriebene Ringbuffer genutzt. Das Aufzeichnen in den Ringpuffer ist standardmäßig aktiviert. Kann aber bei Bedarf deaktiviert werden.

```
1 # ENABLE TRACING
2 $ echo 1 > tracing_on
3 # DISBALE TRACING
4 $ echo 0 > tracing_on
```

Mit dem folgenden Befehl kann der Inhalt des Ringbuffers auch während einer Aufzeichnung, 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, sind ausreichend.**3**-1

```
1 # GET LAST TRACELOG EVENTS
2 $ less /sys/kernel/tracing/trace
```

Das Lesen während einer Aufzeichnung mittels Trace hat keinerlei Einfluss auf den Inhalt des Ringbuffers. Die bisherigen Aufzeichnungen der Ereignisse können mit dem Leeren der `trace`-Datei entfernt werden.




```
# 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

```
1 # EMPTY TRACELOG
2 $ echo '' > trace
```

Um einen Überlauf an Informationen zu verhindern kann die Aufzeichnung auch konsumierend gelesen werden. Somit werden beim Lesen zeitgleich diese aus dem Ringbuffer entfernt. Ein weiterer 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 angelegt werden. Dies erfolgt im Untervereziechnis `instances/`. Das Debug-FS 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`[8] ist das bekannteste und meistgenutzte Hilfsmittel zur Aufzeichnung. Dies ist ein Kommandozeilenwerkzeug, das auf den meisten gängigen Linux-Distributionen bereits vorinstalliert ist. Mit dem letzten Befehl werden alle Events vom

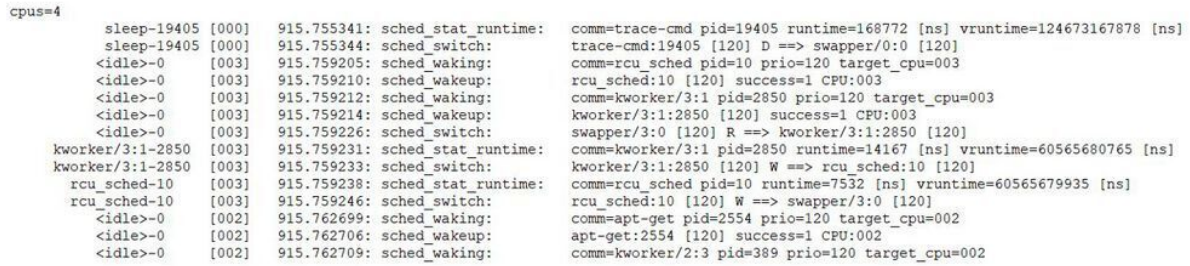


Bild **3**-2: trace-cmd Report

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 Systemen 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 aufgezeichnete 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 bestimmten Ereignissen einzugrenzen. Mit dem Tool ist es einfach 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 Skriptsprache, mit welcher

nicht nur Aggregation, 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 Trace-Aufzeichnungen 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 aktuellen Verzeichnis gesucht.

```
# OPEN KERNELSHARK WITH trace.dat
$ kernelshark
# OPEN KERNELSHAR WITH SPECIFIED TRACELOG
$ kernelshark -i other_trace.dat
```

Im Folgenden ist die grafische Darstellung**3**-3 zu sehen. Dabei besitzt jeder Task einen eigenen Farbton. Für jeden Prozessorkern wird eine eigene Zeile dargestellt.

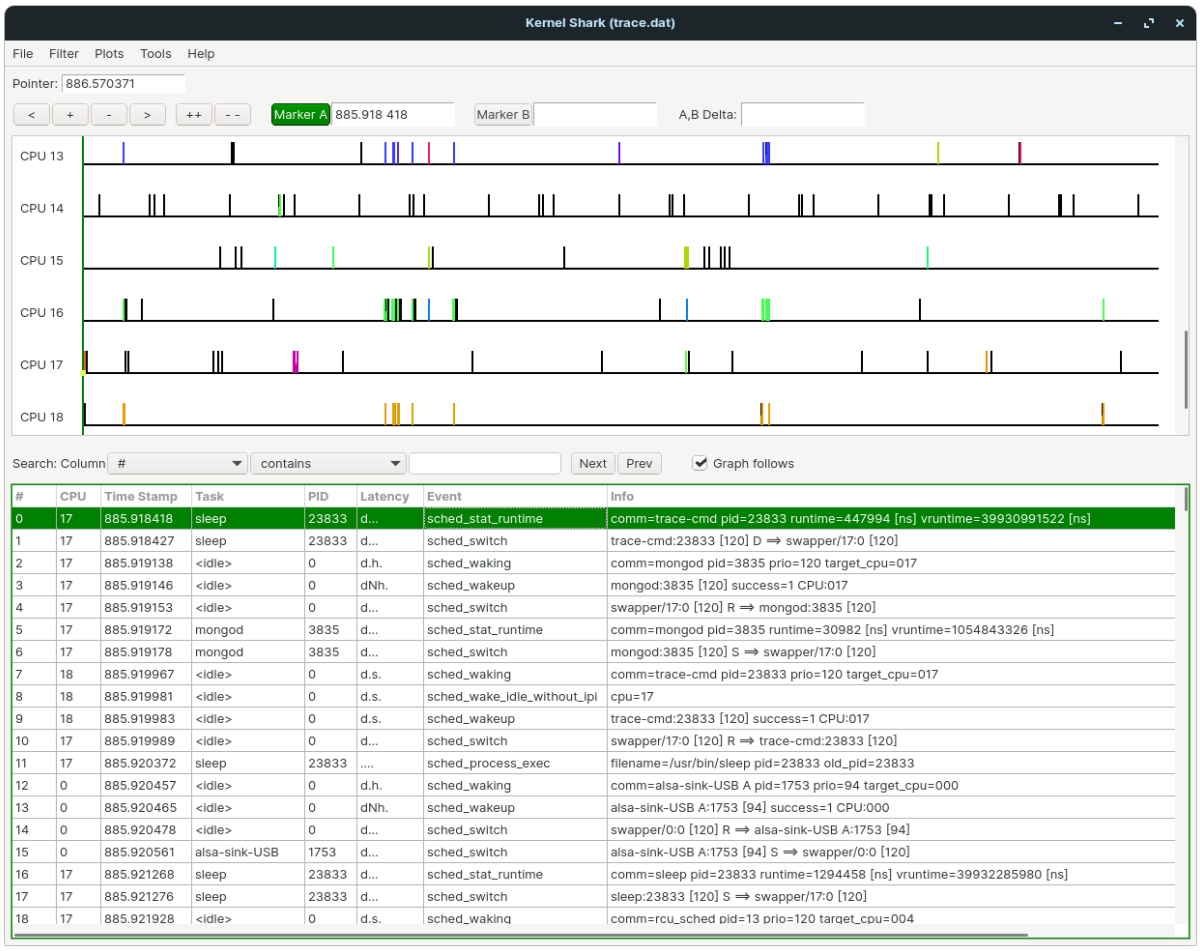


Bild **3**-3: Kernelshark

4 Beispiel - TCP Paketanalyse

Dieses Beispiel zeigt, wie der Empfang von Transmission Control Protocol (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 Internet of Things (IOT)-Anwendungen der Fall sein, bei denen das Message Queuing Telemetry Transport (mqtt)-Protokoll verwendet wird. Hierbei können viele kleine Netzwerkpakete von IOT-Sensoren einen starken Traffic am Server bzw mqtt-Broker zur Folge haben.

4.1 bpftrace Installation

Dabei 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 erfordert 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` Skript, 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.

```
1  // tcpdrop.bt - SIMPLIFIED
2  kprobe:tcp_drop
3  {
4      // GET SOCKET INFORMATION
5      $sk = ((struct sock *) arg0);
6      $inet_family = $sk->__sk_common.skc_family;
7      //ADRESSES
8      $daddr = ntop($sk->__sk_common.skc_daddr);
9      $saddr = ntop($sk->__sk_common.skc_rcv_saddr);
10     // PORTS
11     $dport = $sk->__sk_common.skc_dport;
12     $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`[7] verwendet. Mit diesem ist es möglich, User Datagram Protocol (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 `16Byte` gestellt um somit viele kleine Pakete in kurzer Zeit erzeugen zu können. Im Anschluss wurde `bpftrace` gestartet, welches die Events als Logdatei `tcpdrop_log` in einem lesbaren Textformat ausgeben soll.

```
1  # START SERVER
2  $ ntttcp -r
```

```
3  NTTTCP for Linux 1.4.0
4  -----------------------------------------------------------
5  21:27:58 INFO: 17 threads created
6
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 einiger Zeit wurde `ntttcp` und `bpftrace` die Aufzeichnung manuell gestoppt. Das aufgezeichnete Trace-Log 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-Identifikator (ID) und dem Programm eingeleitet unter welches das Event ausgelöst hat. In diesem Fall wurde der Paketverlust durch ein Empfangenes Paket der `ntttcp`-Anwendung ausgelöst. Die Senderichtung des Pakets kann anhand der Quell und Empfangs Internet Protocol (IP)-Adresse ermittelt werden. Danach folgt der Kernel-Stacktrace, in welchem der Funktionsaufruf-Verlauf bis zum Auslösen des überwachten Events 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. Diese traten zudem vermehrt aus, je länger das System getestet wurde. Eine tiefergehende Untersuchung des Kernel-Stacktrace kann hierzu genauere Informationen bereitstellen. Das Beispiel zeigt auch, dass nicht nur das Auslösen von Events protokolliert werden kann, sondern auch mittels einfacher Skript-Befehle komplexe Debug-Informationen systematisch gewonnen werden können.

5 Beispiel - Reverse Engineering

In diesem Abschnitt soll an einem einfachen Beispiel gezeigt werden, wie es mittels Tracing möglich ist, eine Analyse der Nachrichten auf einem I2C-Bus durchzuführen.

5.1 Ausgangsszenario

Als Ausgabgspunkt für dieses Beispiel, kommuniziert ein Eingebettetes-System mit einem Sensor über den I2C-Bus. Das Programm welches mit dem Sensor kommuniziert ist eine Black-Box. Somit steht kein Quellcode zur Verfügung. Hier soll das Protokoll analysiert werden, um dieses in einer späteren Anwendung nachbauen zu können. Auch gibt es hier keinen Zugriff auf die elektrische Ebene des Bus-Systems, somit kann kein Logic-Analyzer verwendet werden.

5.2 Aufzeichnung Trace-Log

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 `i2c`-Events aktiviert.

```
1  # DANGER: RUN ALL NEXT COMMANDS AS ROOT
2  $ sudo su
3  # ACTIVATE DEBUG FS
4  $ mount -t debugfs none /sys/kernel/debug
5  # USE NOP TRACER
6  $ echo nop > current_tracer
7  # CLEAR RECENT EVENT LOG
8  $ echo > /sys/kernel/debug/tracing/trace
9  # ENABLE ALL I2C-BUS EVENTS
10 echo 1 > /sys/kernel/debug/tracing/events/i2c/enable
```

```
# ENABLE TRACING
$ echo 1 > /sys/kernel/debug/tracing/tracing_on
```

Nachdem das Tracing aktiviert wurde, wurde das Trace-Log manuell analysiert.

```
$ cat /sys/kernel/debug/tracing/trace
# tracer: nop
#
# entries-in-buffer/entries-written: 96/96
#           _-----=> irqs-off
#          / _----=> need-resched
#         | / _---=> hardirq/softirq
#         || / _--=> preempt-depth
#         ||| / _-=> migrate-disable
#         |||| /     delay
#TASK-PID |||||  TIMESTAMP  FUNCTION
# |   |   |||||     |         |
7127   987.236090: i2c_write: i2c-1 #0 a=020 f=0000 l=1 [01]
7127   987.236656: i2c_result: i2c-1 n=1 ret=1
7127   987.737266: i2c_write: i2c-1 #0 a=020 f=0000 l=1 [02]
7127   987.737827: i2c_result: i2c-1 n=1 ret=1
7127   988.238418: i2c_write: i2c-1 #0 a=020 f=0000 l=1 [04]
7127   988.238977: i2c_result: i2c-1 n=1 ret=1
7127   988.739545: i2c_write: i2c-1 #0 a=020 f=0000 l=1 [08]
7127   988.740132: i2c_result: i2c-1 n=1 ret=1
[...]
```

5.3 Auswertung

Das Tracelog zeigt nun einige Events vom Typ `i2c_write`. Diese werden für die Analyse benötigt, da hier die vom System über den I2C-Bus gesendeten Nachrichten stehen. Zu sehen ist, dass über den `i2c-1` Bus des Systems gesendet wird. Die Zieladresse ist dabei `0x20` und wird im Log mit dem Präfix `a=` angegeben. Die Länge der gesendeten Bytes ist mit `l=1` angegeben. Somit wurde nur ein Byte an den Slave gesendet. Die eigentlichen Daten sind in Hex-Array-Schreibweise angegeben. Hier wurde nacheinander `[01]`, `[02]`, `[04]`, `[08]` gesendet.[11]

Die korrektheit der Adresse kann zusätzlich mittels des `i2cdetect`-Befehls überprüft werden. Die Ausgabe zeigt, dass am System nur ein I2C-Slave mit der Adresse `0x20` angeschlossen ist, welches somit mit der Ausgabe im Trace-Log übereinstimmt.

```
# SCAN FOR I2C BUS CLIENTS
$ i2cdetect -y 1
```

```
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:                          -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: 20 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
```

Anhand der I2C-Adresse und der einfachheit der gesendeten Daten, kann auf einen `PCF8574` Port-Expander-Integrated Circuit (IC) geschlossen werden. Dieser nimmt jeweils an Adresse `0x20` ein Byte entgegen und schaltet somit die jeweiligen Ausgangspins. Der Quellcode bestätigt diese Erkenntnisse ebenfalls. Die in Python geschriebene "Black-Box" verwendet das `smbus`-Modul um auf den I2C-Bus-1 zuzugreifen und sendet in einer Dauerschleife jeweils ein Byte.

```python
#!/bin/env python3
import smbus
import time
# USE I2C-BUS 1
bus = smbus.SMBus(1)

while True:
    bus.write_byte(0x20, 0x01)
    time.sleep(0.5)
    bus.write_byte(0x20, 0x02)
    time.sleep(0.5)
    bus.write_byte(0x20, 0x04)
    time.sleep(0.5)
    bus.write_byte(0x20, 0x08)
    time.sleep(0.5)
```

Somit konnte das Protokoll des I2C-Slave mittels Tracing ohne Kenntnis der Software analysiert und reverse engineered werden.

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] DRONAMRAJU, Srikar: *Uprobe-tracer: Uprobe-based Event Tracing*. https://docs.kernel.org/trace/uprobetracer.html. Version: 01.05.2022

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

[5] 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

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

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

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

[11] TV, Linux: *Bus snooping/sniffin*. https://linuxtv.org/wiki/index.php/Bus_snooping/sniffing. Version: 01.05.2022

Akronyme

FS Filesystem. 2–4, 7, 10

I2C Inter-Integrated Circuit. 4, 18–20

IC Integrated Circuit. 20

ID Identifikator. 16

IOT Internet of Things. 14

IP Internet Protocol. 16

mqtt Message Queuing Telemetry Transport. 14

TCP Transmission Control Protocol. 14–17

UDP User Datagram Protocol. 15

Abbildungsverzeichnis