

FH Aachen

Fachbereich Elektrotechnik und Informationstechnik

Embedded Systems SS2022

Referat

Linux Tracing

Marcel Ochsendorf, Muhammed Parlak

Inhalt

1	Einleitung	1
1.1	Relevanz	1
2	Grundlagen	2
2.1	Ringbuffer	2
2.2	Debug-Filesystem	2
2.3	Tracing	2
2.3.1	Tracer	3
2.3.2	Events	3
2.4	Abfangen von Events	5
2.5	Probes	6
2.5.1	uprobes	6
2.6	Nachteile / verschärfung	7
3	Tools	8
3.1	Trace-Log Aufzeichnung	8
3.1.1	trace-cmd	9
3.1.2	bpftrace	12
3.1.3	Kernelshark	12
4	Beispiel - TCP Paketanalyse	14
4.1	Ausgangsszenario	14
4.2	bpftrace Installation	14
4.3	TCPDROP.BT	15
4.4	Aufzeichnung Trace-Log	15
4.5	Ausgabe	16
5	Beispiel - Identifikation von Laufzeitproblemen	18
5.1	Ausgangsszenario	18
5.2	Kernel des Testsystems	19
5.3	Aufzeichnung Trace-Log	20

5.4 Visualisierung und Beurteilung des Trace-Logs	21
5.4.1 RT Kernel	21
5.4.2 LTS Kernel	21
Literaturverzeichnis	22
Abbildungsverzeichnis	23

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 nachzuvollziehen 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 aus vielen Softwaremodulen, welche untereinander interagieren
- systemnahes debugging, kann dann dabei helfen, schwer zu erkennende und nicht einfach reproduzierbare fehler aufzufinden
- auch möglichkeit bei custom driver deugging während des bootvorgangsb ## Tracing

2 Grundlagen

2.1 Ringbuffer

Bei einem Ringbuffer handelt sich um eine Datenstruktur, die es READER und WRITER erleichtert, Informationen asynchron auszutauschen. 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.

2.2 Debug-Filesystem

- möglichkeiten

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

2.3 Tracing

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. Zudem können hier auch die Verfügbaren `tracer` angezeigt und aktiviert werden, welche noch mehr debugging Optionen bereitstellen.

2.3.1 Tracer

```
1 # GET TRACERS
2 $ cat /sys/kernel/debug/tracing/available_tracers
3 hwlat blk mmiotrace function_graph wakeup_dl wakeup_rt wakeup
        function nop
4 # USE SPECIFIC TRACER
5 $ echo function_graph > /sys/kernel/debug/tracing/
        current_tracer
6 # DISABLE TRACER USAGE
7 $ 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. Der `function_graph`-Tracer gibt bei Verwendung zusätzliche Informationen, wie z.B. die Laufzeit von einzelnen Funktionenftrace. Auch kann dieser den Stacktrace und den Call-Stack übersichtlich darstellen, indem hier die Namen der aufgerufenen Funktionen ausgegeben werden.

```
1 # CALL STACK USING FUNCTION_GRAPH TRACER
2 $ echo function_graph > /sys/kernel/debug/tracing/
   current_tracer
3 $ cat /sys/kernel/debug/tracing/trace
4 # tracer: function_graph
5 # CPU- DURATION           FUNCTION CALLS
6 # |    |    |
7 0)          |      getname() {
8 0)  4.442 us  |      kmem_cache_alloc() {
9 0)  1.382 us  |      __might_sleep();
10 0) 2.478 us  |
11 [...]
```

2.3.2 Events

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

```
1 # GET AVAILABLE EVENT LIST
2 $ cd /sys/kernel/debug/tracing/events
3 $ ls -lah | awk '{print $9}'
4 alarmtimer
5 clk
6 cpuhp
7 drm
```

```

8 exceptions
9 ext4
10 # => EXT4_READPAGE, EXT4_WRITEPAGE, EXT4_ERROR,
     EXT4_FREE_BLOCKS
11 filelock
12 filemap
13 fs_dax
14 ftrace
15 gpio
16 # => GPIO_DIRECTION, GPIO_VALUE
17 hda
18 i2c
19 irq
20 net
21 smbus
22 # => READ, WRITE, REPLY
23 sock
24 # => SOCKET_STATE_CHANGED, SOCK_EXCEED_BUFFER_LIMIT,
     SOCK_REC_QUEUE_FULL
25 spi
26 tcp
27 timer
28 # => TIMER_STOP, TIMER_INIT, TIMER_EXPIRED
29 [...]

```

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.

```

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

Um ein eventevents 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 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 # DISABLE ALL EVENTS
5 $ echo 0 > ./enable
6 # ENABLE SPECIFIC EVENT
7 $ echo 1 > ./ext4_readpage/enable
8 $ echo 1 > ./ext4_writepage/enable
```

Nach dem Aktivieren der Events, können

2.5 Probes

- was sind probes ### kprobes

kprobeskprobes können dazu verwendet werden, Laufzeit und Performance-Daten des Kernels zu sammeln. Der Vorteil an diesen ist, dass diese Daten ohne Unterbrechung 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 zusammen zu können.

kretprobes

kretprobeskprobes

2.5.1 uprobes

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. 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.

```
1 // hello.c
2 #include <stdio.h>
3
4 int main(void)
5 {
6     int i;
7
8     for (i = 0; i < 5; i++)
9         printf("Hello uprobe\n");
10
```

```
11     return 0;  
12 }
```

```
1 # CREATE EXECUTE OBJECT  
2 $ /root/gcc hello.c -o hello  
3  
4 # GET OFFSET  
5 $ objdump -F -S -D hello | less  
6  
7 # CREATE A uprobe_event  
8 $ echo "p:my_uprobe /path_to_application/hello:<0xOffset>" >  
      uprobe_events  
9  
10 # ACTIVATE UPROBE EVENTS  
11 $ echo 1 > /sys/kernel/tracing/events/uprobes/my_uprobe/enable  
12  
13 # EXECUTE  
14 $ /root/hello  
15 # Hello uprobe  
16 # Hello uprobe  
17 # Hello uprobe  
18 # Hello uprobe  
19 # Hello uprobe  
20  
21 # PRINT TRACED EVENTS  
22 $ cat /sys/kernel/debug/tracing/trace
```

- für anwendungn
- system libs

2.6 Nachteile / verschärfung

- 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 notwendig 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 Ringpuffers, auch während einer Aufzeichnung, ausgegeben 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 **3-1**:

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

```
1 $ 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

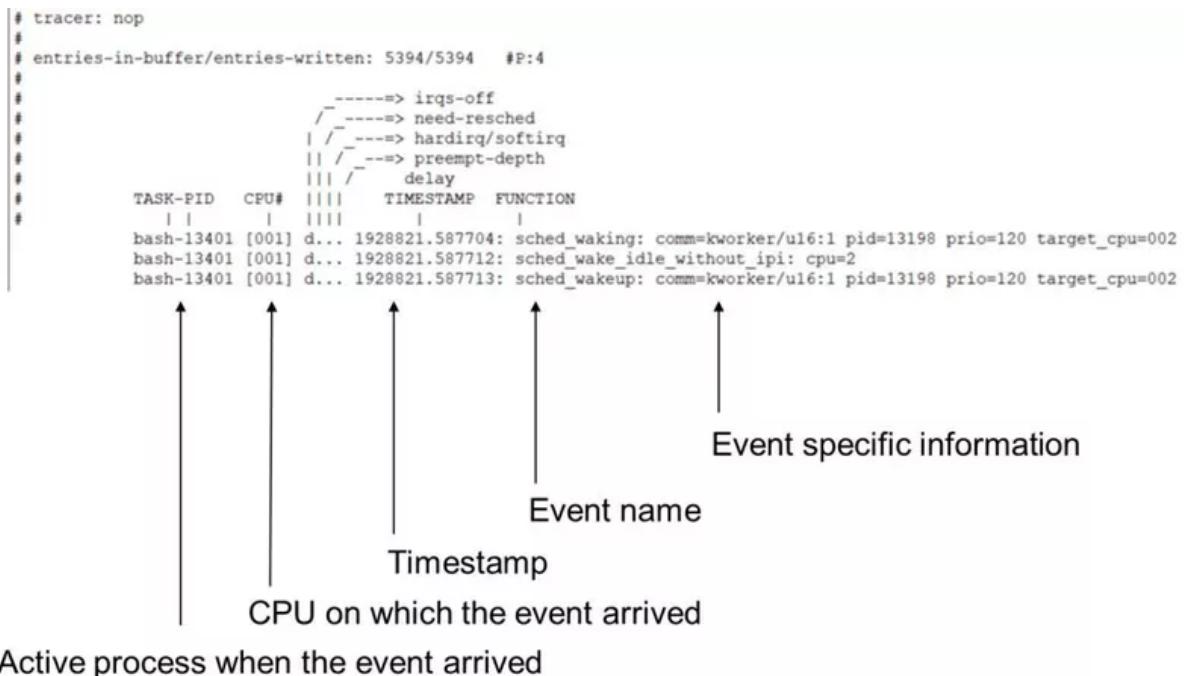


Bild 3-1: Trace-Log

entfernt 3-2.

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 Unterverzeichnis `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 gängigen Linux Distributionen bereits vorinstalliert ist.

```

1 # CHECK IF TRACING IS ENABLED
2 $ sudo mount | grep tracefs

```

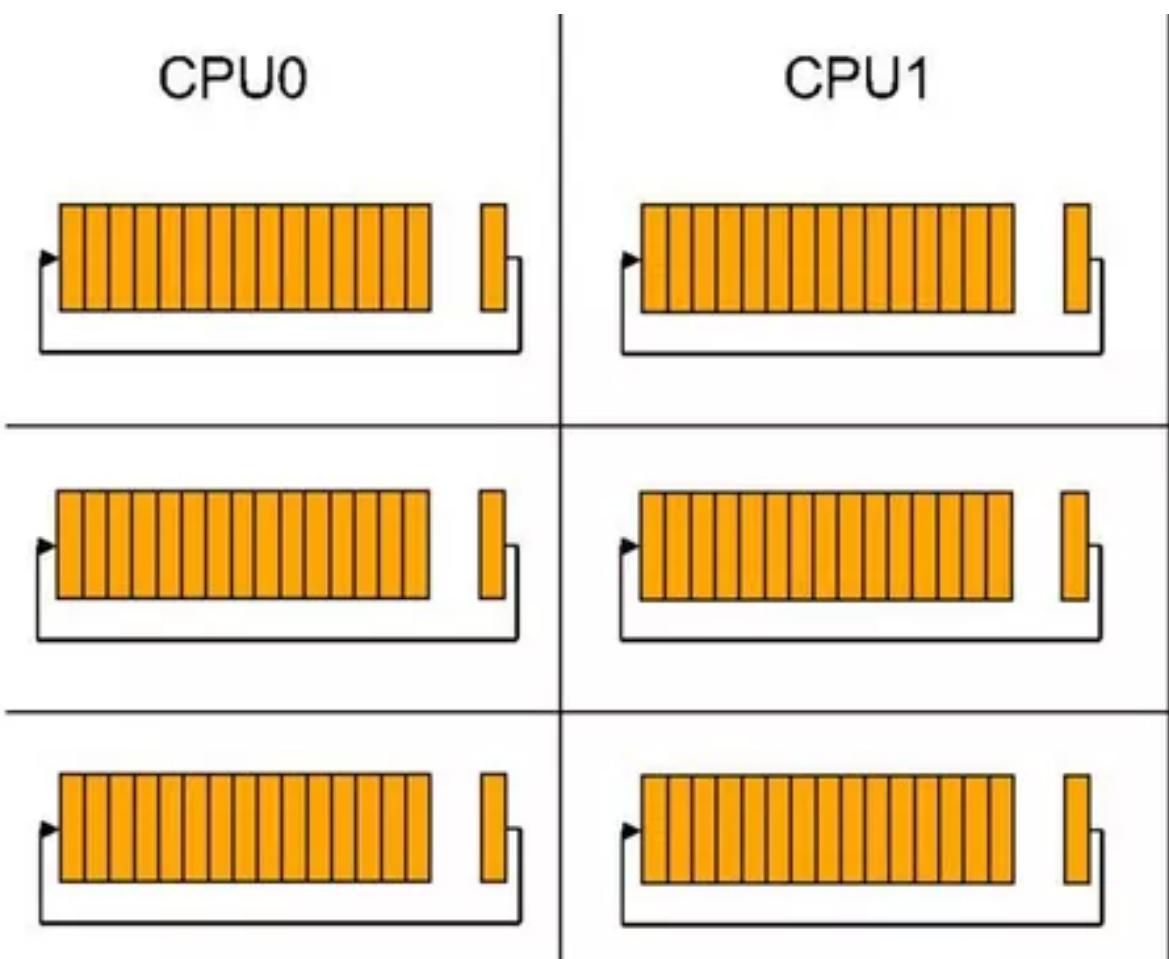


Bild 3-2: Ringbuffer

```
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 3-3: trace-cmd Report

```
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
```

- output erklärung

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 augezeichnet Reports visuell darzustellen. Die Ausgabe erfolgt mit dem Befehl `trace-cmd report [-i <Dateiname>]` als Tabelle in der Konsole und ist somit rein Textbasiert3-3.

Auf diese Aufzeichnung zusätzlich ein Filter angewendet werden, um die Suche auf bestimmten Ereignissen einzuschränken.

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 Skriptsprache mit der nicht nur Aggregation, 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 $ bpftrace ./biolatency.bt
4 @usecs:
5 [512 , 1K)          10 | @
6 [ 1K , 2K)          426 | 000000000000000000000000
7 [2K , 4K)           230 | 000000000000000000000000
8 [4K , 8K)            9 | @
9 [8K , 16K)          128 | 000000000000000000000000
10 [16K , 32K)         68 | 000000000000000000000000
11 [...]
```

3.1.3 Kernelshark

Das zuvor vorgestellte `trace-cmd` ist wie oben erwähnt nur ein textbasiertes Analyse-Tool. 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.

```
1 # OPEN KERNELSHARK WITH trace.dat
2 $ kernelshark
3 # OPEN KERNELSHARK WITH SPECIFIED TRACELOG
4 $ kernelshark -i <Dateiname>
```

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

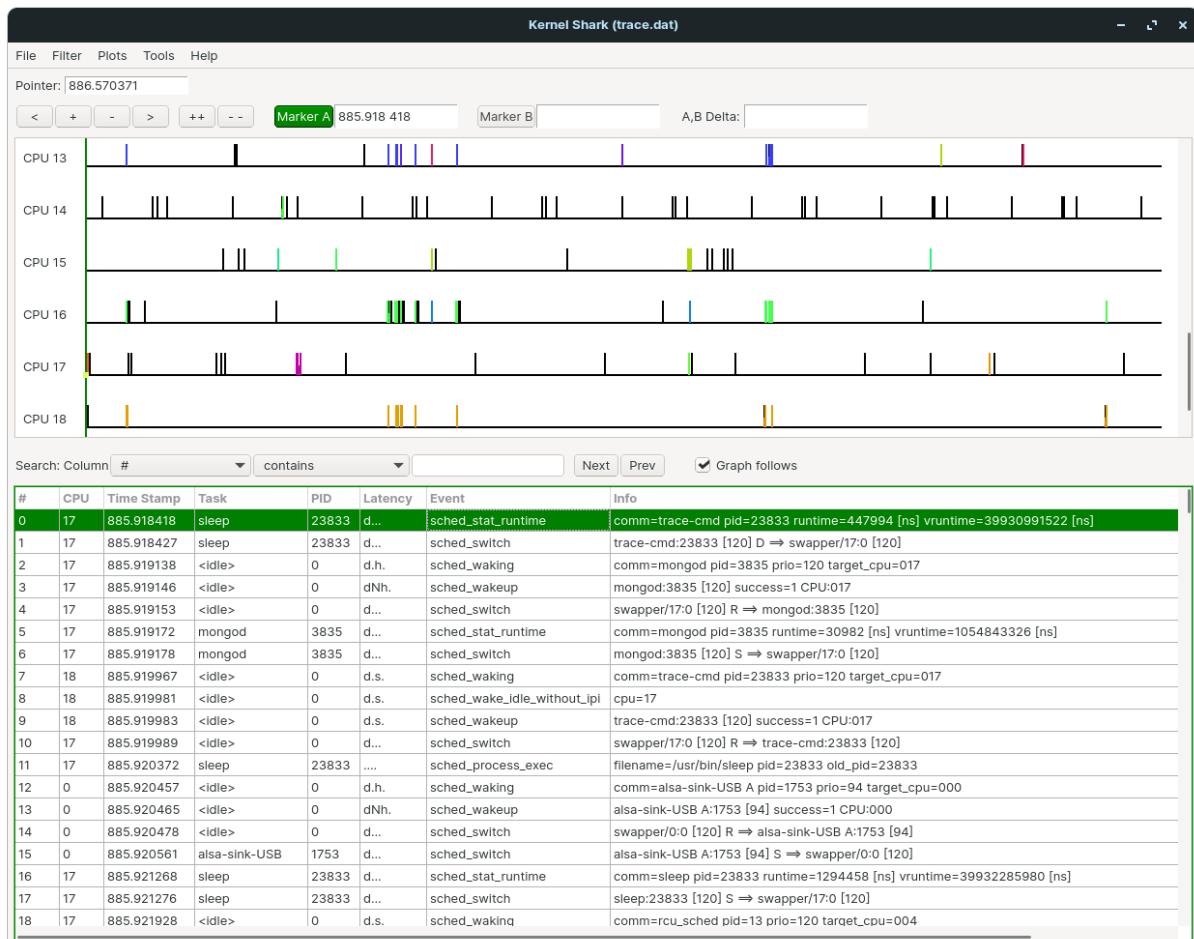


Bild 3-4: Kernelshark

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.

4.1 Ausgangsszenario

- iot anwendungen
- viele kleine pakete

4.2 bpftrace Installation

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 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 ~
```

4.3 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.

```
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     // ADDRESSES
8     $daddr = ntohs($sk->_sk_common.skc_daddr);
9     $saddr = ntohs($sk->_sk_common.skc_rcv_saddr);
10    // PORTS
11    $dport = $sk->_sk_common.skc_dport;
12    $lport = $sk->_sk_common.skc_dport;
13    // LOG INTO USERSPACE
14    printf("%39s:%-6d %39s:%-6d %-10s\n", $saddr, $lport,
15           $daddr, $dport, $statestr);
15 }
```

Um eine Lastspitze auf dem System zu erzeugen wurde das Netzwerkbenchmark-Tool `nttcp` 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.

4.4 Aufzeichnung Trace-Log

Um die Messung zu starten, wurde zuerst der `nttcp`-Server gestartet, dieser empfängt die vom Sender gesendeten Pakete. Im zweiten Schritt wurde der `nttcp`-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 lesabaren 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_perInsn 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 `nttcp` und `bpftrace` die Aufzeichnung manuell gestoppt. Der aufgezeichnete Trace für das `tcp_drop`-Event befindet sich in der `tcpdrop_log` Datei.

4.5 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 ausgelöst hat. In diesem Fall wurde der Paketverlust durch ein Empfangenes Paket der `nttcp`-Anwendung ausgelöst. 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
22     gpioSetMode(GPIO, PI_OUTPUT);
23
24     bool state = false;
25
26     while(running)
27     {
28         state = !state;
29         gpioWrite(GPIO, (int)state);
30     }
31     return 0;
32 }
```

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 Funktionalitäten 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-Compilationrpi4rt aus dem `rpi-4.19.y-rt` Branch des `raspberrypi/linux` Repo-

sitory 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` verwendet. Auf dem Ziel-System wurde dabei nur die Aufzeichnung vorgenommen und die Analyse der Logs erfolgte auf einem separaten 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 `Ctrl+C` manuell beendet werden. Die resultierende Ausgabedatei `gpio_test_trace_*` enthält die von `trace-cmd` geloggten Daten.

5.4 Visualisierung und Beurteilung des Trace-Logs

5.4.1 RT Kernel

5.4.2 LTS Kernel

“ # Fazit

- aufzeichnugn auf device => headless
- analyse visuell

Literaturverzeichnis

Abbildungsverzeichnis

3-1	Trace-Log	9
3-2	Ringbuffer	10
3-3	trace-cmd Report	11
3-4	Kernelshark	13