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

1.2 Tracing

1.3 Ursprung

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.

```
1 # GET TRACERS
2 $ cat /sys/kernel/debug/tracing/available_tracers
3
4 # GET AVAILABLE EVENT LIST
5 $ cd /sys/kernel/debug/tracing
6 $ ls -1 events/
7
8 # sched
9 # irq
```

2.4 Beispiel für Events

- file operartionen
- netzzwerk zugriffe

2.4.1 Scheduler

2.4.2 IRQ

2.5 Abfangen von Events

```
1  $ cd /sys/kernel/debug/tracing
2  $ echo 1 > events/sched/enable
```

2.5.1 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.

2.5.2 kretprobes

2.5.3 CPU-Traps

2.5.4 uprobes

- für anwendungn

- system libs

2.6 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 Tracing auf Mikrokontrollern ?????

4 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.

4.1 Trace-Log Aufzeichnung

4.1.1 trace-cmd

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

Die `trace-cmd` Konsolenanwendung dient nicht nur zur Aufzeichnung der Trace-Events, sondern bietet auch die Möglichkeit augezeichnetet Reports visuell darzustellen. Die Ausgabe erfolgt als Tabelle in der Konsole und ist somit rein Textbasiert.

4.1.2 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 # ...
```

4.1.3 Netzwerk Paketanalyse Beispiel

- mit bfttrac verworfene netzwerkpakete zählen wenn zu viel trafic

4.1.4 Kernelshark

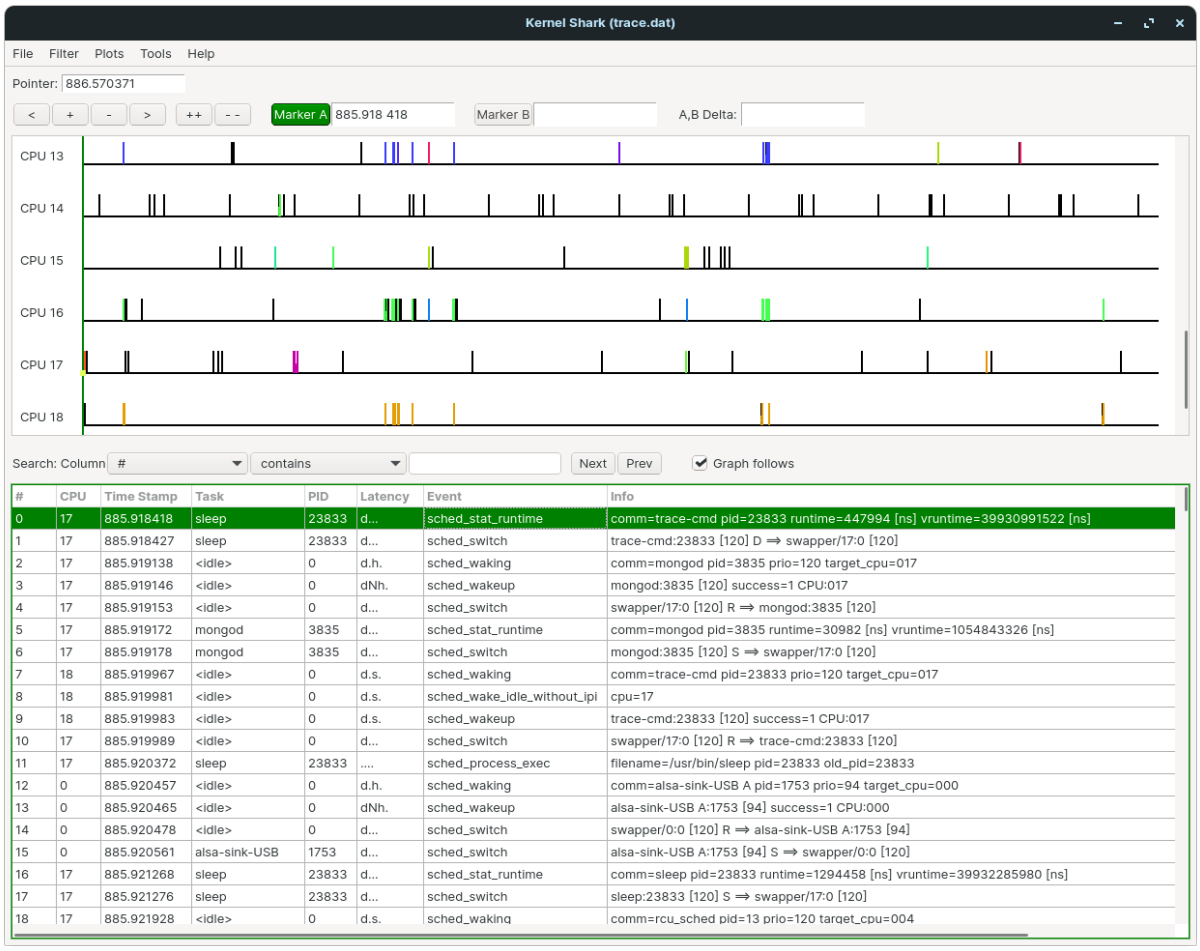

Bild **4**-1: Kernelshark

5 Interpretation des Kernel-Trace Ergebnisses

6 Beispiel der Identifikation von Laufzeitproblemen

In diesem Abschnitt soll an einem einfache Beispiel gezeigt werden ## 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`.

6.1 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.

6.1.1 Aktivierung der Events

```
1 $ echo 1 > /sys/kernel/debug/tracing/tracing_on
2 $ cat /sys/kernel/debug/tracing/trace
3 $echo > /sys/kernel/debug/tracing/trace
```

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

6.2 Visualisierung und Beurteilung des Trace-Logs mittels kernelshark

Literaturverzeichnis