

Echtzeitsysteme

Übungen zur Vorlesung

Messung von Ausführungszeiten & Antwortzeiten

Florian Schmaus Peter Wägemann

Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)
Lehrstuhl für Informatik 4 (Verteilte Systeme und Betriebssysteme)
<https://www4.cs.fau.de>

09.11.2017



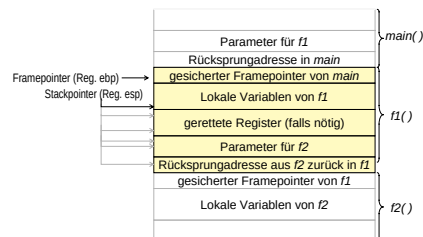
Übersicht

- 1 Wiederholung: Stack-Aufbau
- 2 Einflüsse der Ausführungszeit
- 3 Zeitmessung
 - Zeitgeber
 - Probleme von Messungen
- 4 Was bedeutet Antwortzeit?
- 5 Aufgabe: Antwortzeit



Wiederholung: Stack-Aufbau

```
1 int main() {  
2     int a, b, c;  
  
3     a = 10;  
4     b = 20;  
  
5     f1(a, b);  
  
6     return (a);  
7 }
```



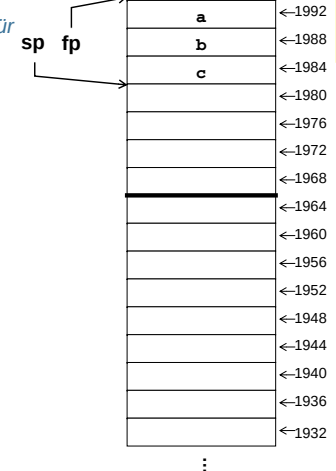
- Stack-Frame zur Verwaltung des Stacks
 - Lokale Variablen
 - Funktionsparameter
 - Rücksprungadressen



Ablauf Funktionsaufruf

```
int main() {  
    int a, b, c;  
    a = 10;  
    b = 20;  
    f1(a, b);  
    return(a);  
}
```

Stack-Frame für
main erstellen
&a = fp-4
&b = fp-8
&c = fp-12



```
int main() {  
    int a, b, c;  
    a = 10;  
    b = 20;  
    f1(a, b);  
    return(a);  
}
```



Übersicht

- 1 Wiederholung: Stack-Aufbau
- 2 Einflüsse der Ausführungszeit
- 3 Zeitmessung
 - Zeitgeber
 - Probleme von Messungen
- 4 Was bedeutet Antwortzeit?
- 5 Aufgabe: Antwortzeit



Beispiel: Ausführungszeit

```
1 uint64_t count_positive(uint64_t* array, size_t size){
2     uint64_t count = 0;
3     for(int i = 0; i < size; i++){
4         if (array[i] > 0){
5             count++;
6         }
7     }
8     return count;
9 }
```

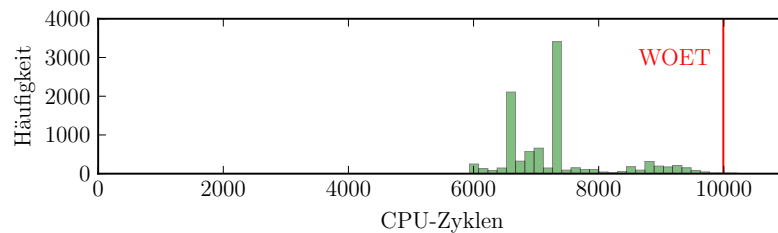
Benchmark-Programm

- Iteration über Array mit variabler Größe
- Zählen der positiven Zahlen in Array

👉 Wie verhält sich die **Ausführungszeit** bei 10.000 Messungen?



Histogramm: Ausführungszeiten

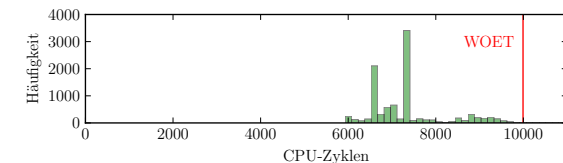


- Maximum der beobachteten Ausführungszeiten
(engl. worst-observed execution time **WOET**)
- 10.000 Ausführungen der Funktion count_positive()
- Maximum: 9992 CPU-Zyklen
- Hohe Streuung der Ausführungszeiten

Warum?



Setup der Messung: Applikation



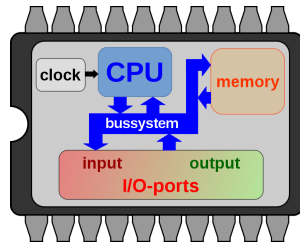
```
1 void main(void){
2     uint64_t array[ARRAY_SIZE];
3     memset(array, 0, sizeof(array)); // alle Eingabedaten mit 0 initialisiert!
4     start = get_time();
5     positive = count_positive(array, ARRAY_SIZE);
6     stop = get_time();
7     printf("%lu", stop-start);
8 }
```

- Ausführungszeit unabhängig von unterschiedlichen Eingabedaten
 - Feste Länge
 - Immer mit 0 initialisiert
- 👉 **kein Einfluss** der Eingabedaten

Woher kommen dann die Schwankungen?



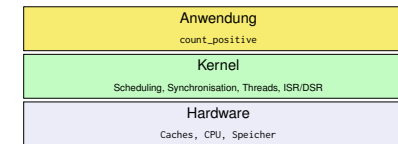
Setup der Messung: Hardware



- CPU: Intel Core i7
- Takt: ≤ 3.3 GHz
- Cache: 4 MB *Smart* Cache
- Universalbetriebssystem
- Aufgaben-System: zusätzliche Last (`stress --cpu 8 --io 8 --hdd 8 --vm 8`)



Einflüsse der Hardware & des Betriebssystems

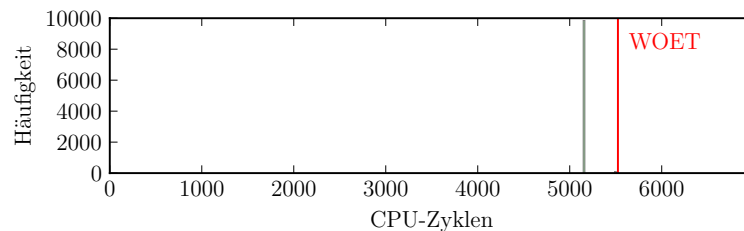


- Pipelining: spekulative Ausführung (engl. Branch Prediction)
- Takt: dynamische Änderung möglich (engl. Dynamic Frequency Voltage Scaling, DVFS)
- Cache: heuristische Strategien
- Scheduling
 - Keine Priorisierung von Aufgabe: Completely Fair Scheduler
 - Timer-Interrupts möglich
 - **Verdrängung** möglich

Wie verhält sich die Messung auf dem **EZS-Board**?



count_positive auf EZS-Board



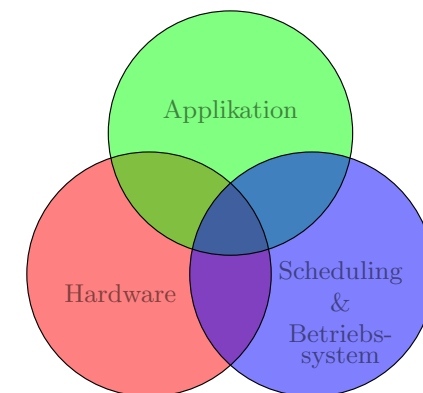
- Geringere Komplexität $\leadsto$ *weniger* Streuung
- Trotzdem Unterschiede
 - Hardware
 - 3-stufige Pipeline
 - Spekulative Ausführung: ART Accelerator™
 - Betriebssystem: *Schwankungen der Ausführungszeit trotz eines Threads?*

Wissen über **Hard- & Software** essenziell für Echtzeitsysteme



Was beeinflusst die Ausführungszeit?

Zusammenfassung



- 1 **Applikation:** Eingabedaten, ...
 - 2 **Hardware:** Caches, Pipelining, ...
 - 3 **Scheduling:** Höherprioritäre Aufgaben, Interrupts, Overheads, ...
- (gegenseitige) Einflüsse **kaum vermeidbar**, aber reduzierbar



Übersicht

- 1 Wiederholung: Stack-Aufbau
- 2 Einflüsse der Ausführungszeit
- 3 Zeitmessung
 - Zeitgeber
 - Probleme von Messungen
- 4 Was bedeutet Antwortzeit?
- 5 Aufgabe: Antwortzeit



Zähler in Mikrocontrollern

Zähler (Counter) zählen Hardware-Ereignisse

- Externer Drehgeber (Radumdrehung)
- Interner Prozessortakt (hohe Auflösung)
- Externer Quarz (Real-Time Clock)

Äquidistante Ereignisse ermöglichen einen *Zeitgeber (Timer)* für

- Periodische Aktivierung
- *Messen von Zeitabständen*
- *Kontrolliertes Verbrennen von Prozessorzeit*



Zähler-Betriebsmodi

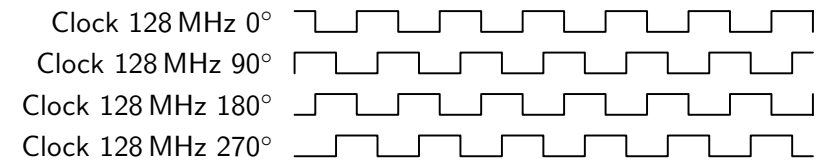
Zähler bzw. Zeitgeber bieten zwei Betriebsmodi:

Abfragebetrieb (Polling) Aktives Auslesen des Zählers
 ~> bis Erreichen eines vorgegebenen Wertes

Unterbrecherbetrieb (Interrupt) Zähler unterbricht System
 ~> Erreichen eines konfigurierten Zählerstandes.



Probleme von Taktgebern



- Clock-Drift
 - Abweichung der internen Uhr von Realzeit
 - Temperaturabhängiger Phasenunterschied
 - Äußerst kritisch in verteilten Echtzeitsystemen
 - Quarz: $\approx 10^{-6}$ sec/sec = 1 sec in 11,6 Tagen

Lösung

- Messung mit externer, hochauflösender Uhr

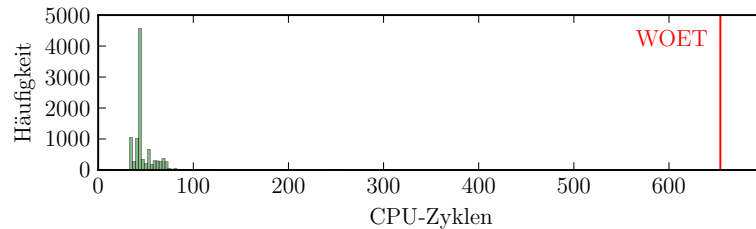
Oszilloskop ~> Übungsaufgabe



Overhead der Messung

Wer misst misst Mist!

```
1 start = get_ticks(); // ticks ~ processor cycles/timer values
2 // nothing executed here
3 stop = get_ticks();
4 printf("%lu", stop-start);
```



- Overhead-Bestimmung für Intel-CPU; EZS-Board $\leadsto$ *erweiterte Übungen*
- Overhead-Bestimmung vernachlässigt
 - Instruktionen bis zum 1. Abfragen des Timer-Registers
 - Instruktionen nach dem 2. Abfragen des Timer-Registers
 - *Weitere Störfaktoren?*



Einschub: Zusammenfassung von Ergebnissen

- *How Not To Lie With Statistics: The Correct Way To Summarize Benchmark Results* [1]

- Wichtige Regeln

1. Für **normalisierte Werte** nicht das **arithmetische Mittel** verwenden
2. Für **normalisierte Werte** das **geometrische Mittel** verwenden
3. Für **Rohdaten (mit Einheiten)** das **arithmetische Mittel** verwenden

- Arithmetisches Mittel: $x_{arith} = \frac{1}{n} \sum_{i=1}^N x_i$

- Geometrisches Mittel: $x_{geom} = \sqrt[n]{\prod_{i=1}^N x_i}$

- Für Messungen in Echtzeitsystemen

4. Alle Standardabweichungen müssen **weniger als 1 %** betragen [2]

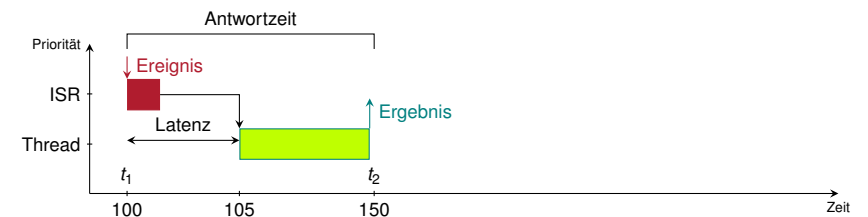


Übersicht

- 1 Wiederholung: Stack-Aufbau
- 2 Einflüsse der Ausführungszeit
- 3 Zeitmessung
 - Zeitgeber
 - Probleme von Messungen
- 4 Was bedeutet Antwortzeit?
- 5 Aufgabe: Antwortzeit



Ausführungszeit & Antwortzeit

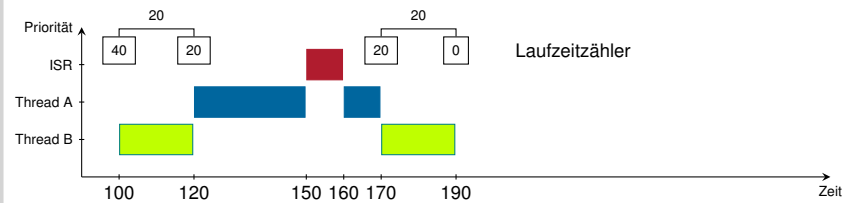


Stoppuhr

- Punkte auf der Zeitachse t_1 und $t_2 \leadsto$ Ereignis und Ergebnis
- Antwortzeit ist $\Delta t = t_2 - t_1$ (Beispiel: $150 - 100 = 50$ Zählerticks)



Messung der ausgeführten Zeit



Rechenzeitsimulation

- Verbrauchte *Laufzeit* eines Threads
- Vorgegebene Zeit aktiv warten $\leadsto$ Laufzeit verbrauchen
- Umsetzung
- Funktion, die *aktiv* t_{wct} wartet $\leadsto$ Schleife auf Zählerwert
- HW-Zähler läuft bei Unterbrechungen weiter! $\leadsto$ *lokaler* Zähler
- Dekrement bei jeder Änderung? Beispiel: Sprung von 120 $\rightarrow$ 170



Übersicht

- 1 Wiederholung: Stack-Aufbau
- 2 Einflüsse der Ausführungszeit
- 3 Zeitmessung
 - Zeitgeber
 - Probleme von Messungen
- 4 Was bedeutet Antwortzeit?
- 5 Aufgabe: Antwortzeit



libEZS-Überblick

Plattformunabhängige Hilfsfunktionen

- Timer-Zugriff (Zeitmessung)
- DAC-Zugriff
- GPIO-Zugriff
- ...

```
<aufgabe>
|-- CMakeLists.txt
|-- app.c
|-- ecos
\-- libEZS
    |-- include
    |   |-- ezs_dac.h
    |   |-- ezs_gpio.h
    |   \-- ezs_stopwatch.h
    |-- src
    |   |-- ezs_stopwatch.c
    |   \-- drivers
    \-- stm32
        |-- ezs_dac.c
        |-- ezs_counter.c
        \-- ezs_gpio.c
```

Die *libEZS* wird im Laufe der Übungen erweitert



Zeitmessung in *ezs_stopwatch.c/h*

Die Zeitmessung wird durch zwei Funktionen implementiert:

```
void ezs_watch_start(cyg_uint32 *state);
cyg_uint32 ezs_watch_stop(cyg_uint32 *state);
```

- Parameter: Zeiger auf *globale* Variable
 $\rightarrow$ viele unabhängige Messzeitpunkte
- `ezs_watch_stop(cyg_uint32 *state)` gibt Zeitdifferenz in *Ticks* zurück

Hinweis

`ezs_counter_get()` in `drivers/include/ezs_counter.h`

Hinweis

Auflösung der Zähler in Bruchteilen von Nanosekunden:
 $\rightarrow$ `ezs_counter_get_resolution()`



WCET-Simulator in `ezs_stopwatch.c/.h`

Zu implementieren:

```
void ezs_lose_time(cyg_uint32 wcet, cyg_uint8 percentage);
```

■ Parameter:

- 1 Gewünschte WCET in *Ticks*
- 2 *Maximum* des zufällig zu subtrahierenden *WCET-Anteils*

■ Implementierung muss internen Zähler verwalten

↪ Bei welcher Änderung des Systemzählers anpassen?

↪ Welche Auflösung ist erreichbar

- Jeder Thread besitzt einen eigenen Stack!
- Thread-übergreifende Messungen möglich

■ *Abfragebetrieb*

Hinweis

Auflösung des Zählers in Pikosekunden:

→ `ezs_counter_resolution_ps()`

Kann das problematisch sein?



Rechnen mit Timer-Auflösungen

```
1 // CPU_SPEED is 84MHz
2 // RCC_CLOCK is CPU_SPEED / 2 = 42MHz
3 // timer increment frequency is RCC_CLOCK / (PRESCALER+1)
4 // PRESCALE is configured as 1
5 // counter will overflow after ??? sec
```

■ **Überläufe:** *Dauer der Messung* beachten

- 32-Bit Timer in auf EZS-Board

■ **Rundungsfehler:** *Auflösungen* beachten

- Auflösung in „Mikro-Sekunden-Ticks“
- $1000000/42 = 23809.5238...$ vs. $\text{floor}(1000000/42)$
- $\approx 15 \mu\text{s}$ Rundungsfehler bei 1 s Zeitmessung

Lösung

```
cyg_resolution_t ezs_counter_get_resolution(void);
```

- $\frac{\text{Dividend}}{\text{Divisor}}$ beinhaltet die Auflösung in Nanosekunden ↪ siehe `make doc`



Literatur

- [1] Philip J Fleming and John J Wallace.
How not to lie with statistics: the correct way to summarize benchmark results.
Communications of the ACM, 29(3):218–221, 1986.
- [2] Gernot Heiser.
Systems benchmarking crimes, 2016.
<http://gernot-heiser.org/benchmarking-crimes.html>.

