Echtzeitsysteme

Übungen zur Vorlesung

Cyclic Scope

Simon Schuster **Peter Wägemann**

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

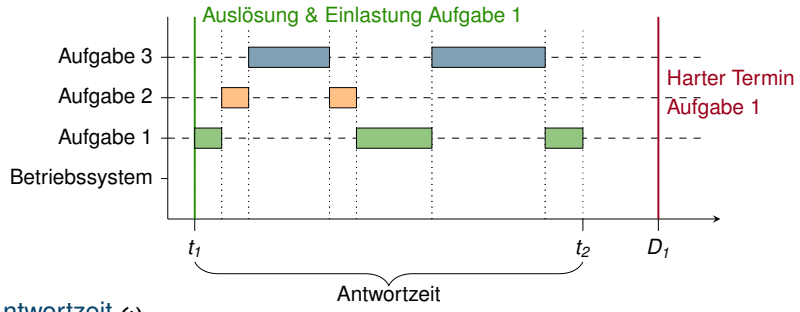
30. November 2018

Hinweise: Simple Scope

- Hyperperiode fest auf 100 ms kodiert
- Wenn längere Zeit kein Plot der Ergebnisse
  - 256 Scheduling-Entscheidungen bis Plot angezeigt wird
    event-triggered: `ms_to_cyg_ticks`, `ms_to_ezs_ticks`
    time-triggered: `ms_to_ticks`
- Zu langes Sampling: Überlauf des Timers

Übersicht

Antwortzeitanalyse



- Antwortzeit ω_i
  - Zeitdauer zwischen Auslösezeit und Terminationszeitpunkt
- Idee: Antwortzeitanalyse
  - Terminationszeitpunkt vor dem absoluten Termin d_i
  - Antwortzeit ω_i kürzer als der relative Termin D_i

  → Für jeden Auftrag $J_{i,j}$ in der Aufgabe: $T_i : \omega_{i,j} \leq D_{i,j}$
- Voraussetzungen
  - Bedingungen A1 - A7 müssen eingehalten werden
  - Konzept ist jedoch erweiterbar

Berechnung der Antwortzeit

- Antwortzeit ω_i der Aufgabe T_i berechnet sich zu:

$$\omega_i(t) = e_i + \sum_{k=1}^{i-1} \left\lceil \frac{t}{p_k} \right\rceil e_k; 0 < t \leq p_i$$

  - Aufgabe terminiert bevor das Ereignis (Periode) erneut eintritt
  - Setzt sich zusammen aus:
    - WCET e_i von T_i
    - WCETs $e_1, \ldots, e_{i-1}$ der Aufgaben $T_1, \ldots, T_{i-1}$ höherer Priorität

- Prüfung: $\omega_i(t) \leq t$
  - $t = jp_k; \quad k = 1, 2, \ldots, i; \quad j = 1, 2, \ldots, \lfloor \min(p_i, D_i)/p_k \rfloor$
  - Zeitbedarf erhöht sich nur bei Auslösung dringlicherer Aufgaben
  - Bis das Ereignis erneut eintritt/der Termin der Aufgabe erreicht ist

⚠ Ist die Ungleichung für **einen** Zeitpunkt t erfüllt, ist T_i zulässig

Restriktionen des periodischen Modells

⚠ Mathematische Ansätze zur zeitlichen Analyse periodischer Echtzeitsysteme bedingen häufig starke Einschränkungen:

A1 Alle Aufgaben sind periodisch

A2 Alle Arbeitsaufträge können an ihren Auslösezeitpunkten eingeplant und ausgeführt werden

A3 Termine und Perioden sind identisch

A4 Kein Arbeitsauftrag gibt die Kontrolle über den Prozessor ab

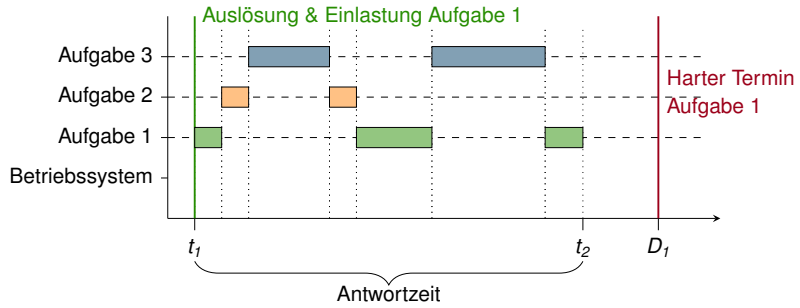
A5 Alle Aufgaben sind unabhängig[1]

A6 Die Kosten durch Unterbrechungen, Ablaufplanung und Verdrängung sind vernachlässigbar

A7 Alle Aufgaben verhalten sich voll-präemptiv

[1] D.h. die einzige gemeinsame Ressource ist die CPU und es existieren keine Einschränkungen hinsichtlich der Auslösezeiten der Arbeitsaufträge voneinander.

Antwortzeitanalyse



Praktische Betrachtung

Beruht auf Voraussetzungen A1 – A7:

A7 Alle Aufgaben verhalten sich voll-präemptiv

⇝ hochpriorer Aufgaben können jederzeit verdrängen

A6 Kosten von Unterbrechungen, Ablaufplanung und Verdrängung vernachlässigbar

Übersicht

Randbedingungen für die Rahmenlänge

Lang genug und so kurz wie möglich halten...

Terminüberwachung unterstützen $\rightsquigarrow$ f hinreichend kurz

1. Erfordert eine rechtzeitige Auslösung: $f \leq p_i$, für alle $1 \leq i \leq n$
2. Möglich unter der Bedingung: $2f - ggT(p_i, f) \leq D_i$, für alle $1 \leq i \leq n$
3. f teilt die Hyperperiode H so, dass gilt: $\lfloor p_i/f \rfloor - p_i/f = 0$, für ein i mit $1 \leq i \leq n$

Jobverdrängung vermeiden $\rightsquigarrow$ f hinreichend lang

4. Erfüllt, wenn gilt: $f \geq \max(e_i^f)$, für $1 \leq i \leq H/f$
   - e_i^f gibt die WCET aller Aufträge im Rahmen i an
   - Jeder Auftrag läuft in der durch f gegebenen Zeitspanne komplett durch
   - Erste Abschätzung nach unten: $f \geq \max(e_i)$, für $1 \leq i \leq n$

⚠ Ermittlung von $\max(e_i^f)$ erfolgt nachgelagert:

- Kann erst durch konkrete Ablaufplanung beantwortet werden
- Iterativer Prozess $\mapsto$ Wiederholung für jedes potentielle f

Beispielsystem

Aufgabe T_i	Periode p_i / ms	WCET e_i / ms	Termin D_i / ms
T_1	9	2	5
T_2	18	3	8
T_3	45	3	45

Übersicht

Busy Loop

```
1 void main(void) {
2   while (true) {
3     Task0();
4     Task1();
5     Task2();
6     Task3();
7   }
8 }
```

Vorteile:

- Geringe Verwaltungsallgemeinkosten
- Simpel, übersichtlich, ...

Nachteile:

- Nur *eine Periode*, *keine Deadline-Überprüfung* möglich
- Mathematische *Analyse unmöglich*

Multi-Perioden-Hauptschleife

Anforderung: wir wollen unterschiedliche Perioden haben

Lösung:

- Jede Aufgabe hat ein *Aktivierungs-Flag*
- Feste Abarbeitungsreihenfolge innerhalb einse Durchlaufs

Multiraten-Hauptschleife

```
1 void main(void) {
2   while (true) {
3     wait_for_timer_tick();
4     if      (activated0) { activated0 = false; Task0(); }
5     else if (activated1) { activated1 = false; Task1(); }
6     else if (activated2) { activated2 = false; Task2(); }
7     else if (activated3) { activated3 = false; Task3(); }
8   }
9 }
```

Setzen der Flags in der Hauptschleife problematisch

⇝ Lang laufender Task kann Flag-Setzen/*Deadlineüberprüfung* verzögern

Multi-Perioden-Zeitgeber

Setzen der Flags in der Hauptschleife problematisch

⇝ Lang laufender Task kann Flag-Setzen/*Deadlineüberprüfung* verzögern

Lösung: Setzen der Flags in Zeitgeber-Interruptbehandlung

```
1  volatile uint8_t timer = 0;
2  ...
3  ++timer; // Interrupt alle 1ms
4  ...
5  if ((timer %   5) == 0) { activated0 = true; } // Task0 alle 5ms
6  if ((timer %  10) == 0) { activated1 = true; } // Task1 alle 10ms
7  if ((timer %  20) == 0) { activated2 = true; } // Task2 alle 20ms
8  if ((timer % 100) == 0) { activated3 = true; } // Task3 alle 100ms
9
10 if (timer >= 100) { timer = 0; } // Ueberlaufbehandlung
```


Einschub: Schlüsselwort volatile

- Bei einem Interrupt wird `timer_event = 1` gesetzt
- Aktive Warteschleife wartet, bis `timer_event != 0`
- Flag (scheinbar) in Schleife nicht verändert ⇝ Compiler-Optimierung
  - `timer_event` wird einmalig vor der Warteschleife in Register geladen
    ☞ Endlosschleife
- `volatile` erzwingt das Laden bei jedem Lesezugriff

```
1 static uint8_t timer_event = 0;
2 ISR (INT0_vect) { timer_event = 1; }
3
4 void main(void) {
5   while(1) {
6     while(timer_event == 0) { /* warte auf Timer-Event */ }
7     /* bearbeite Timer-Event */
```

```
1 volatile static uint8_t timer_event = 0;
2 ISR (INT0_vect) { timer_event = 1; }
3
4 void main(void) {
5   while(1) {
6     while(timer_event == 0) { /* warte auf Timer-Event */ }
7     /* bearbeite Timer-Event */
```


Einschub: Lost-Update-Problematik

- Tastendruckzähler: Zählt mittels Variable `zaehler`
  - Inkrementierung in der Unterbrechungsbehandlung
  - Dekrementierung im Hauptprogramm zum Start der Verarbeitung

Hauptprogramm **H**

```
1 ; volatile uint8_t zaehler;
2 ; C-Anweisung: zaehler--;
3 lds r24, zaehler
4 dec r24
5 sts zaehler, r24
```

Interruptbehandlung **I**

```
1 ; C-Anweisung: zaehler++
2 lds r25, zaehler
3 inc r25
4 sts zaehler, r25
```

Zeile	zaehler	r24	r25
-	5		
3 H	5	5	-
4 H	5	4	-
2 I	5	4	5
3 I	5	4	6
4 I	6	4	6
5 H	4	4	-

Fazit Cyclic Executive

Vorteile

- Einfach, übersichtlich, wenige Ressourcen notwendig, ...
- Mehrere Perioden, Deadlineüberprüfung, erleichtert WCET-Analyse
- Mathematische Analyse anwendbar

Probleme der Implementierung: *Nebenläufige Zugriffe*

(Sichtbarkeits-)Synchronisation:

1. zwischen Zeitgeberunterbrechung und `main`-if/else
2. beim Setzen der Flags

Andere Namen in der Literatur:

Main Loop Scheduling, Main Loop Tasker, Prioritized Cooperative Multitasker, Non-preemptive Scheduler, ...

Übersicht

Aufgabe 5 - Hinweise

Wichtige Hinweise

Basisübung: Reine Textaufgabe, *Denksportaufgabe*

$\rightsquigarrow$ keine Implementierung notwendig

- Kern der Aufgabe: Auswirkung der Rahmenlänge

Erweiterte Übung: Implementierung einer *Cyclic Executive*

- Überprüfung der Lauffähigkeit und Deadlines von Jobs
- Vereinfachte Ausnahmebehandlung:
  Ausgabe welcher Task Deadline überschritten hat
- Verwendung *eines* eCos Alarms