Echtzeitsysteme

Übungen zur Vorlesung

Cyclic Scope

Simon Schuster **Phillip Raffeck**

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

Wintersemester 2019/20

Übersicht

Übersicht

Idee strukturierter Ablaufplanung

Herkömmliche Zeitsteuerung

- potentiell komplexe Ablaufpläne
  - ⇝ eventuell inklusive Verdrängung
- Einplanung zu jedem Takt möglich
- Terminüberprüfung für jeden Task

→ Busy-Loop mit Entscheidung zu jedem Takt

Implementierung ohne regelmäßigen Zeitgeber (tickless)

+ weniger Zeitgeberereignisse
– Einplanung/Terminüberprüfung weiterhin zu jedem Takt möglich

Strukturierte Ablaufplanung

- Einführung künstlicher, zeitlicher Struktur
- weniger Zeitpunkte für Einplanung/Terminüberprüfung

Vor-/Nachteile zyklischer Ablaufpläne

☞ Zyklisches Ablaufmodell liefert wohlgeordnete Ablaufpläne
- Eine feste Rahmengröße mit definierten Schranken
- Ablaufplanung (↦ Zuteilung Aufträge zu Rahmen) findet offline statt

→ Einlastung und Terminüberwachung zu definierten Zeitpunkten

– Busy-Loop-Verhalten innerhalb eines Rahmens (vgl. IV-2/12)
- Sequentielle, kooperative Abarbeitung der Aufträge
- Keine individuelle Laufzeitüberwachung und Ausnahmebehandlung
- Anfällig für Jitter und mangelnde Periodizität

\+ Niedrige Verwaltungsgemeinkosten
- Einlastung und Terminüberwachung findet nur an den Rahmengrenzen statt
- Keine Verdrängung (engl. *preemption*) (vgl. III-2/13)
- Minimalistisches Laufzeitsystem (Dispatcher+Terminprüfung genügt)

\+ Hohe Vorhersagbarkeit
- Einziger Interrupt ist der Zeitgeber an den Rahmengrenzen

→ Unterbrechungsfreier Durchlauf innerhalb der Rahmen

→ Vereinfacht die WCET-Analyse ungemein (vgl. Kapitel III-3)

Randbedingungen für die Rahmenlänge

Lang genug und so kurz wie möglich halten...

Terminüberwachung unterstützen ⇝ 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 ⇝ 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 ↦ 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

Vorgriff aus der Vorlesung

Nicht-periodische Aufgaben

- Definiert durch $T_i = (i_i, e_i, D_i)$
- *Aperiodische* vs. *sporadische* Aufgabe
- *Mischbetrieb*: periodisch ⟷ sporadisch/aperiodisch
  - *dynamische* Einplanung
  - Beeinflussung periodischer Aufgaben?
  - Übernahmeprüfung ⟷ Antwortzeitminimierung

Nicht-periodische Arbeitsaufträge

- Kaum a-priori Wissen (Zeitpunkt, . . .)
- Herausforderung Mischbetrieb: Erhaltung **statischer Garantien**
- Abweisung (spor. Aufg.): schwerwiegende Ausnahmesituation

Vorgriff aus der Vorlesung (Forts.)

Basistechniken zur Umsetzung

- Unterbrecherbetrieb ⇝ bevorzugt nicht-periodische Aufgaben
- Hintergrundbetrieb ⇝ stellt nicht-periodische Aufgaben hinten an
- Zusteller ⇝ konvertiert nicht-period. in periodische Aufgaben
  - Spezielle periodische Aufgabe $T_s = (p_s, e_s)$
  - Ausführungsbudget, Auffüllperiode und -regeln
  - Abbildung auf Prioritätswarteschlange (z. B. AJQ)

Slack Stealing

- Idee: Termin ist maßgeblich
  ⇝ *Verschieben* periodischer Aufgaben möglich
- *Erfordert Unterbrecherbetrieb*
- Problem: **Schlupfzeit** bestimmen
  - Zeitsteuerung (mit Rahmen): Einfach ⇝ $f - x_k$
  - Ereignissteuerung: schwierig ⇝ dynamischen Berechnung

Vorgriff aus der Vorlesung (Forts.)

Periodische Zusteller

- Verschiedene Ausführungen
  z. B.: Polling, Deferrable, Sporadic Server
- Unterscheiden sich im **Regelwerk**
- I. d. R. für mehrere Aufgaben zuständig

Beispiel: Abfragender Zusteller (Polling Server)

- Periodische Aufgabe $T_P = (p_s, e_s)$
- Budget e_s verfällt
- Im Falle sporadischer Aufgaben schwierig:
  - $p_P \leq \frac{D_s}{2}$, wobei $D_s \leq i_s$ ⇝ Abtasttheorem

→ hohe Abtastfrequenz, Überlastgefahr

Übersicht

Busy Loop

```
void main(void) {
  while (true) {
    Task0();
    Task1();
    Task2();
    Task3();
  }
}
```

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 eines Durchlaufs

Multiraten-Hauptschleife

```
void main(void) {
  while (true) {
    wait_for_timer_tick();
    if (activated0) { activated0 = false; Task0(); }
    if (activated1) { activated1 = false; Task1(); }
    if (activated2) { activated2 = false; Task2(); }
    if (activated3) { activated3 = false; Task3(); }
  }
}
```

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

```
volatile uint8_t timer = 0;
...
++timer; // Interrupt alle 1ms
...
if ((timer %   5) == 0) { activated0 = true; } // Task0 alle 5ms
if ((timer %  10) == 0) { activated1 = true; } // Task1 alle 10ms
if ((timer %  20) == 0) { activated2 = true; } // Task2 alle 20ms
if ((timer % 100) == 0) { activated3 = true; } // Task3 alle 100ms

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

```
static uint8_t timer_event = 0;
ISR (INT0_vect) { timer_event = 1; }

void main(void) {
  while(1) {
    while(timer_event == 0) { /* warte auf Timer-Event */ }
    /* bearbeite Timer-Event */
volatile static uint8_t timer_event = 0;
ISR (INT0_vect) { timer_event = 1; }

void main(void) {
  while(1) {
    while(timer_event == 0) { /* warte auf Timer-Event */ }
    /* 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*

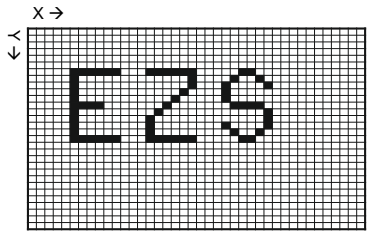
⇝ keine Implementierung notwendig

- Kern der Aufgabe: Auswirkung der Rahmenlänge

Erweiterte Übung: Implementierung einer *Cyclic Executive*

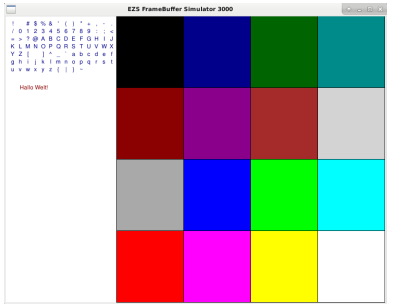
- Überprüfung der Lauffähigkeit und Deadlines von Jobs
- Vereinfachte Ausnahmebehandlung: Ausgabe welcher Task Deadline überschritten hat
- Verwendung *eines* eCos Alarms

Grafische Ausgabe – Das Display



- **Framebuffer** ⇝ Grafische Ausgabe
- eCos bietet einheitliche, *hardwareunabhängig Schnittstelle*
- Darstellung auf LCD: `ezs_fb_*` ⇝ `make doc`
- Auflösung in unserem Fall: 320x240
  - Zwecks Portabilität immer Makros nutzen:
  - Breite: `CYG_FB_WIDTH(FRAMEBUF)`
  - Höhe: `CYG_FB_HEIGHT(FRAMEBUF)`

Grafische Ausgabe – Das Display



Farbpalette

- 16 unterstützte Farben
- `CYG_FB_DEFAULT_PALETTE_*`

⇝ `libEZS/include/ezs_fb.h`