

Echtzeitsysteme

Übungen zur Vorlesung

Zustellerkonzepte: Extended Scope (Teil 2)

Simon Schuster **Peter Wägemann**

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

14. Dezember 2018



Nicht-periodische Aufgaben

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

Nicht-periodische Arbeitsaufträge

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



Basistechniken zur Umsetzung

- **Unterbrecherbetrieb** \leadsto Bevorzugt nicht-periodische Aufgaben
- **Hintergrundbetrieb** \leadsto Stellt nicht-periodische Aufgaben hinten an
- **Slack Stealing**
 - Idee: Termin ist maßgeblich
 \leadsto *Verschieben* periodischer Aufgaben möglich
 - *Erfordert Unterbrecherbetrieb*
 - Problem: Schlupfzeit bestimmen
 - Zeitsteuerung (mit Rahmen): einfach $\leadsto f - x_k$
 - Ereignissteuerung: schwierig \leadsto dynamischen Berechnung
- **Zusteller** \leadsto Konvertieren nicht-period. in periodische Aufgaben
 - Spezielle periodische Aufgabe $T_s = (p_s, e_s)$
 - Ausführungsbudget, Auffüllperiode und -regeln
 - Abbildung auf Prioritätswarteschlange (z. B. AJQ)



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 $D_s \leq j_s \rightsquigarrow$ Abtasttheorem
- hohe Abtastfrequenz, Überlastgefahr



Bandweite-bewahrende Zusteller

- Budget bleibt erhalten
 ~> Verbesserung des Abfragebetriebs
- Regelwerk wird erweitert
 ~> Auffüll- und Konsumregeln
- Betriebssystem (Scheduler) wacht über Budget

Auslegung

- Größe Budget
 ~> Berücksichtigung aller periodischer Aufgaben
- Verbesserung Antwortzeit
 ~> Kombination mit Hintergrundbetrieb



Beispiel: Aufschiebbarer Zusteller (Deferrable Server)

- Verbrauchsregel: verbraucht $\frac{1}{\text{Zeiteinheit}}$ Budget bei Tätigkeit
- Auffüllregel: periodisches Auffüllen von e_s mit p_s
- Keine Akkumulation

Achtung: aufschiebbarer Zusteller \neq periodische Aufgabe

- **Double hit**
 - \leadsto Kritischer Zeitpunkt und Auffüllzeitpunkt fallen zusammen
- \leadsto Störung ist bis zu e_s größer als bei periodischer Aufgabe



Lösungsansatz: Sporadischer Zusteller (Sporadic Server)

- Verschiedene Ausprägungen
- Beansprucht niemals mehr Zeit als periodische Aufgabe

Beispiel: SpSL Sporadic Server (Sprunt, Sha & Lehoczky)

- Verbraucht $\frac{1}{\text{Zeiteinheit}}$ Budget bei Tätigkeit
- Aufgefüllt wird entsprechend dem Verbrauchsmuster
 - Nächster Auffüllzeitpunkt wird zu Beginn der Tätigkeit bestimmt
 - Aufzufüllendes Budget zum Ende der Tätigkeit
 - \leadsto Auffüllregeln R1 – R3
- SpSL Sporadic Server
 - \leadsto Menge von Aufgaben T_i mit $p_i = p_s$ und $\sum e_i = e_s$



Forts.: SpSL Sporadic Server, Auffüllregeln

- R1: initiales Budget ist e_s
- R2: Auffüllzeitpunkt $rt_s = t_b + p_s$, wobei:
 - T_s besitzt Budget, dann $t_b = P_s$ wird tätig
 - T_s hat kein Budget, dann $t_b = P_s$ *ist* tätig und T_s *erhält Budget*
- R3: Budgetberechnung
 - Sobald P_s untätig wird oder T_s kein Budget mehr hat
 - Budget für $rt_s =$ Verbrauch von T_s seit t_b

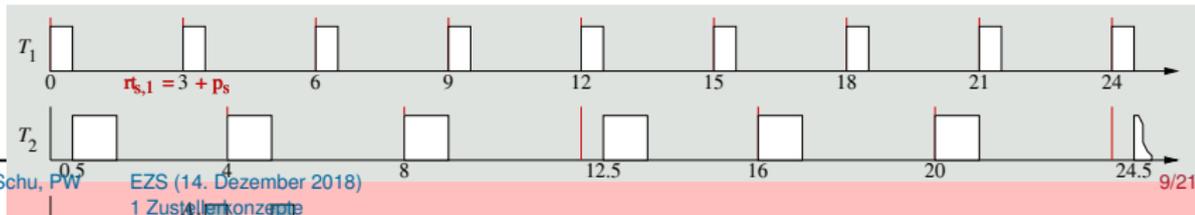
Achtung

- P_s bezeichnet das **Tasksystem** ab der Priorität s (und höher)
- Im Beispiel: kleinere Zahl \rightsquigarrow höherer Priorität



Beispiel: SpSL

$T_1 = (3, 0.5)$, $T_2 = (4, 1)$, $T_3 = (19, 4.5)$ und $T_s = (5, 1.5)$; RM-Ablaufplanung



1 Zustellerkonzepte

2 Rangfolge & Synchronisation

3 Ereignisse in eCos

- Events
- Mailbox



Kausalordnung

- **Relation:** Ursache, Wirkung
- Mehrere Ursache-Wirkungspaare überlappend: Nebenläufigkeit
- Nebenläufigkeit vs. Gleichzeitigkeit
- ☞ Sequentialisierung von Aufgaben

Rangfolge

- Abhängigkeit von Kontrollfluss \rightsquigarrow Reihenfolge
- Oft in Datenabhängigkeiten begründet
 - Produzent/Konsument Verhältnis
 - Konsumierbare Betriebsmittel (Nachrichten, Interrupts, ...)
 - **Begrenzte** Puffer limitieren die Anzahl häufig
- ☞ Beachtung unterschiedlicher **zeitlicher Domänen**



Koordinierung

- Unnötig falls Rangfolge egal
 - Neuester Wert ist ausreichend
- Durch Einplanung \leadsto analytische Verfahren
 - Periodische Aufgaben \leadsto **Passende Raten!**
 - Ablauftabelle
 - Keine Kontrolle zur Laufzeit
- Durch Kooperation \leadsto konstruktive Verfahren
 - Periodische und nicht-periodische Aufgaben
 - Synchronisation \leadsto Vielzahl von Möglichkeiten
 - **In zeitgesteuerten Systemen unsinnig!**



1 Zustellerkonzepte

2 Rangfolge & Synchronisation

3 Ereignisse in eCos

- Events
- Mailbox



Signalisieren von Ereignissen

- Signale unterstützen *Produzent-Konsument Muster*
- Thread/DSR *signalisiert* Ereignis (z. B. Tastendruck)
... konsumierender Thread *wartet*
- Umsetzung: 32-bit Integer \rightsquigarrow 32 *Einzel-signale* pro Flag
 - Ein Flag erlaubt somit $2^{32} - 1$ Signalkombinationen
 - Threads können auf ein Signalmuster blockierend warten oder pollen

Achtung:

Flags zählen keine Ereignisse! (vgl. HW-Interrupts)

¹<http://ecos.sourceware.org/docs-latest/ref/kernel-flags.html>



- Produzenten/Konsumenten teilen sich eine Flag-Objekt
- Dieses wird von der *Anwendung* bereitgestellt (vgl. Alarmobjekt)

- Flag-Objekt muss initialisiert werden:

```
1 cyg_flag_init(cyg_flag_t* flag)
```

- Signal(e) im Flag setzen:

```
1 cyg_flag_setbits(cyg_flag_t* flag, cyg_flag_value_t value)
```

- Bzw. zurücksetzen:

```
1 cyg_flag_maskbits(cyg_flag_t* flag, cyg_flag_value_t value)
```

- Auf Signal warten/pollen:

```
1 cyg_flag_value_t cyg_flag_wait/poll(cyg_flag_t* flag,  
2 cyg_flag_value_t pattern,  
3 cyg_flag_mode_t mode);
```



- `cyg_flag_value_t` pattern setzt gewünschte Signalkombination
- `cyg_flag_mode_t` legt Weckmuster fest
 - `CYG_FLAG_WAITMODE_AND`: alle konfigurierten Signale müssen aktiv sein; sie bleiben nach Aufwachen gesetzt
 - `CYG_FLAG_WAITMODE_OR`: mindestens eines der konfigurierten Signale muss aktiv sein; alle Signale bleiben nach dem Aufwachen gesetzt
 - `CYG_FLAG_WAITMODE_OR` | `CYG_FLAG_WAITMODE_CLR`: mindestens eines der konfigurierten Signale muss aktiv sein; alle gesetzten Signale werden nach dem Aufwachen gelöscht



```
1 static cyg_flag_t flag0;
2
3 void my_dsr(cyg_vector_t v,
4             cyg_ucount32 c,
5             cyg_addrword_t d){
6     cyg_flag_setbits(&flag0, 0x02);
7 }
8
9 void user_thread(cyg_addr_t data){
10    while(true) {
11        cyg_flag_wait(&flag0, 0x22,
12                    CYG_FLAG_WAITMODE_OR | CYG_FLAG_WAITMODE_CLR);
13        ezs_printf("Event!\n");
14    }
15 }
16
17 void cyg_user_start(void){
18    ...
19    cyg_flag_init(&flag0);
20    ...
21 }
```





- Zwischen Threads können *Nachrichten* versendet werden
- Konsument erzeugt einen Briefkasten (engl. mailbox) fester Größe
- Produzenten legt Nachrichten dort ab
 - Inhalt: *Zeiger* auf beliebige Datenstruktur
 - Konsument kann auf *Nachrichteneingang* blockieren
 - Produzent blockiert, falls Briefkasten *voll*
 - Aber auch *nicht-blockierende* Aufrufvarianten

²<http://ecos.sourceforge.org/docs-latest/ref/kernel-mail-boxes.html>

■ Mailbox anlegen:

```
1 void cyg_mbox_create(cyg_handle_t* handle, cyg_mbox* mbox);
```

■ Nachricht verschicken:

```
1 cyg_bool_t cyg_mbox_put(cyg_handle_t mbox, void* item);
```

■ Nachricht empfangen:

```
1 void* cyg_mbox_get(cyg_handle_t mbox);
```

■ Empfang *und* Versand können blockieren

■ *try*-Versionen: Würde ich blockieren?

■ *timed*-Versionen: Blockieren für bestimmte Zeit

→ Selbststudium!

³<http://ecos.sourceware.org/docs-latest/ref/kernel-mail-boxes.html>

■ Initialisierung:

```
1 static cyg_handle_t mailbox_handle;  
2 static cyg_mbox mailbox;  
3 void cyg_user_start(void) {  
4     cyg_mbox_create(&mailbox_handle, &mailbox);  
5     ...  
6 }
```

■ Produzent (Sender):

```
1 void producer_entry(cyg_addrword_t data) {  
2     ...  
3     cyg_mbox_put(mailbox_handle, &my_message);  
4     ...  
5 }
```

■ Konsument (Empfänger):

```
1 void consumer_entry(cyg_addrword_t data) {  
2     ...  
3     void *message = cyg_mbox_get(mailbox_handle);  
4     ...  
5 }
```

