Events und Mailboxen

Florian Franzmann, Martin Hoffmann, Tobias Klaus

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)
www4.informatik.uni-erlangen.de

16. Dezember 2013

Rekapitulation der Vorlesung

Kapitel 5-1: Grundlegende Abfertigung nicht-periodischer Echtzeitsysteme

Nicht-periodische Aufgaben

- Definiert durch $T_i = (i_i, e_i, D_i)$
- 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, WCET, ...)
- Herausforderung Mischbetrieb: Erhaltung statischer Garantien
- Abweisung (spor. Aufg.): Schwerwiegende Ausnahmesituation

Rekapitulation der Vorlesung (Forts.)

Kapitel 5-1: Grundlegende Abfertigung nicht-periodischer Echtzeitsysteme

Basistechniken zur Umsetzung

- Unterbrecherbetrieb $\rightsquigarrow$ Bevorzugt nicht-periodische Aufgaben
- Hintergrundbetrieb $\rightsquigarrow$ Stellt nicht-period. Aufgaben hinten an
- Zusteller $\rightsquigarrow$ Konvertieren nicht-period. Aufgaben in Periodische
  - Spezielle periodische Aufgabe $T_s = (p_s, e_s)$
  - Ausführungsbudget, Auffüllperiode und -regeln
  - Zustände $\rightsquigarrow$ idle, backlogged, eligible, consumes, exhausted
  - Abbildung auf Prioritätswarteschlange (z.B. AJQ)

Periodische Zusteller

- Verschiedene Ausführungen, z.B.: Polling, Deferrable und Sporadic Server
- Unterscheiden sich (lediglich) im Regelwerk
- i.d.R. für mehrere Aufgaben zuständig

Rekapitulation der Vorlesung (Forts.)

Kapitel 5-1: Grundlegende Abfertigung nicht-periodischer Echtzeitsysteme

Beispiel: Abfragender Zusteller (Polling Server)

- Periodische Aufgabe $T_P = (p_s, e_s)$
- Budget e_s verfällt
- Im Falle sporadischer Aufgaben schwierig:
  - $p_P \leq D_s/2$, wobei $D_s \leq i_s$ (quasi Abtasttheorem)
  - $\rightsquigarrow$ hohe Abtastfrequenz, Überlastgefahr

Slack Stealing

- Termin ist maßgeblich $\rightsquigarrow$ Verschieben periodischer Aufgaben möglich
- Problem: Schlupfzeit bestimmen
- Taktsteuerung (mit Rahmen): Einfach $\rightsquigarrow f - x_k$
- Vorrangsteuerung: Schwierig $\rightsquigarrow$ dynamischen Berechnung

Rekapitulation der Vorlesung (Forts.)

Kapitel 5-2: Zustellerkonzepte und Übernahmeprüfung

Bandweite-bewahrende Zusteller

- Budget bleibt erhalten $\rightsquigarrow$ Verbesserung des Abfragebetriebs
- Regelwerk wird erweitert $\rightsquigarrow$ Auffüll- und Konsumregeln
- Betriebssystem (Scheduler) wacht über Budget

Auslegung

- Größe Budget $\rightsquigarrow$ Berücksichtigung aller periodischer Aufgaben
- Verbesserung Antwortzeit $\rightsquigarrow$ Kombination mit Hintergrundbetrieb

Rekapitulation der Vorlesung (Forts.)

Kapitel 5-2: Zustellerkonzepte und Übernahmeprüfung

Beispiel: Aufschiebbarer Zusteller (Deferrable Server)

- Verbrauchsregel: Verbraucht $\frac{1}{\textit{Zeiteinheit}}$ Budget bei Tätigkeit
- Auffüllregel: periodisches Auffüllen von e_s mit p_s
- keine Akkumulation

Achtung

- aufschiebbarer Zusteller $\neq$ periodische Aufgabe
- *Double hit* $\rightsquigarrow$ Kritischer Zeitpunkt und Auffüllzeitpunkt fallen zusammen
- $\rightsquigarrow$ Störung ist bis zu e_s größer als bei periodischer Aufgabe

Rekapitulation der Vorlesung (Forts.)

Kapitel 5-2: Zustellerkonzepte und Übernahmeprüfung

Lösungsansatz: Sporadischer Zusteller (Sporadic Server)

- Verschiedene Ausprägungen
- Beansprucht niemals mehr Zeit als periodische Aufgabe

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

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

Rekapitulation der Vorlesung (Forts.)

Kapitel 5-2: Zustellerkonzepte und Übernahmeprüfung

Forts.: SpSL Sporadic Server, Auffüllregeln

- R1: Initiales Budget ist e_s
- R2: Zeitpunkt $rt_s = t_b + p_s$, wobei:
  - T_s besitzt Budget, dann $t_b = P_s$ wird tätig
  - T_s hat kein Budget, dann $t_b = P_s$ ist/wird 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

Rekapitulation der Vorlesung

Kapitel 6: Rangfolgen

Rangfolge

- Abhängigkeit von Kontrollfluss $\rightsquigarrow$ Reihenfolge
- oft in Datenabhängigkeiten begründet
  - Produzent/Konsument Verhältnis
  - Konsumierbare Betriebsmittel
  - begrenzte Puffer
- Hinweis auf unterschiedliche zeitliche Domänen!

Kausalordnung

- Relation: Ursache, Wirkung, Nebenläufigkeit
- Nebenläufigkeit vs. Gleichzeitigkeit
- Abhängigkeits- und Aufgabengraphen

Rekapitulation der Vorlesung (Forts.)

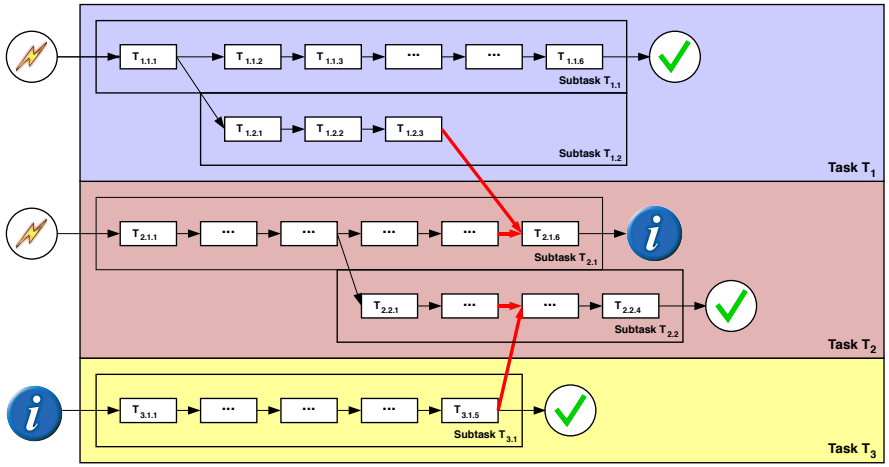
Kapitel 6: Rangfolgen

Koordinierung

- Unnötig $\rightsquigarrow$ Rangfolge egal
  - Neuester Wert ist ausreichend
- Durch Einplanung $\rightsquigarrow$ analytische Verfahren
  - periodische Aufgaben $\rightsquigarrow$ Passende Raten!!!
  - Ablauftabelle, Phasenversatz
  - Keine Kontrolle zur Laufzeit
- Durch Kooperation $\rightsquigarrow$ konstruktive Verfahren
  - periodische und nicht-periodische Aufgaben
  - Synchronisation $\rightsquigarrow$ Vielzahl von Möglichkeiten
  - in zeitgesteuerten Systemen unmöglich!

Rekapitulation der Vorlesung (Forts.)

Kapitel 6: Rangfolgen



Gerichtete Abhängigkeiten: UND, ODER und zeitliche Abhängigkeiten

Signalisierung von Ereignissen - eCos Event Flags[1]

Grundlagen

- Signale unterstützen Produzent-Konsument Muster
- Ein Thread/DSR kann ein Ereignis (z.B. Tastendruck) signalisieren auf das ein konsumierender Thread wartet
- Umsetzung: 32-bit Integer $\rightarrow$ 32 Einzelsignale pro Flag
  - Ein Flag erlaubt somit $2^{32} - 1$ Signalkombinationen
  - Threads können ein Signalmuster blockierend warten, bzw. pollen
- Achtung: Flags zählen keine Ereignisse! (vgl. HW-Interrupts)

[1] `ecos.sourceware.org/docs-latest/ref/kernel-flags.html`

Signalisierung von Ereignissen - eCos Event Flags[2]

API

- Produzenten/Konsumenten teilen sich eine Flagobjekt
- Dieses wird von der Anwendung bereitgestellt (vgl. Alarmobjekt)
- Flagobjekt muss initialisiert werden:

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

- Signal(e) im Flag setzen:

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

- Bzw. zurücksetzen:

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

- Auf Signal warten/pollen:

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

[2] `ecos.sourceware.org/docs-latest/ref/kernel-flags.html`

Signalisierung von Ereignissen - eCos Event Flags[3]

API - Konsumieren von Signalen

- `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 dem 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.

[3] ecos.sourceware.org/docs-latest/ref/kernel-flags.html

Signalisierung von Ereignissen - eCos Event Flags[4]

Beispiel

```
cyg_flag_t flag0;

void my_dsr(cyg_vector_t v, cyg_ucount32 c, cyg_addrword_t d){
  cyg_flag_setbits(&flag0, 0x02);
}

void user_thread(cyg_addr_t data){
  while(1){
    cyg_flag_wait(&flag0, 0x22,
      CYG_FLAG_WAITMODE_OR | CYG_FLAG_WAITMODE_CLR);
    printf("Event!\n");
  }
}

void cyg_user_start(){
...
  cyg_flag_init(&flag0);
...
}
```

[4] ecos.sourceware.org/docs-latest/ref/kernel-flags.html

Versenden von Nachrichten - eCos Mail Boxes[5]

Grundlagen

- Zwischen Threads können Nachrichten versendet werden
- Ein Konsument erzeugt einen Briefkasten (mailbox) einer gewissen Größe (Default: 10)
- Produzenten können Nachrichten dort ablegen
  - Inhalt: Zeiger auf beliebige Datenstruktur
  - Konsument kann auf Nachrichtenempfang blockieren
  - Produzent blockiert, falls Briefkasten voll
  - Aber auch nicht-blockierende Aufrufvarianten
- API $\rightarrow$ Selbststudium!

[5] ecos.sourceware.org/docs-latest/ref/kernel-mail-boxes.html

Versenden von Nachrichten - Beispiel

- Initialisierung

```
static cyg_handle_t mailbox_handle;
static cyg_mbox     mailbox;

void cyg_user_start(void) {
  cyg_mbox_create(&mailbox_handle, &mailbox);
  ... }
```

- Produzent (Sender)

```
void producer_entry(cyg_addrword_t data) {
  ...
  cyg_mbox_put(mailbox_handle, &my_message);
  ... }
```

- Konsument (Empfänger)

```
void consumer_entry(cyg_addrword_t data) {
  ...
    void *message = cyg_mbox_get(mailbox_handle);
  ... }
```