

Wiederholung

Zustellerkonzepte

Tobias Klaus Florian Schmaus Peter Wägemann

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

16. Januar 2017





Evaluation der Veranstaltung

- Eure Meinung (Lob/Kritik) ist uns wichtig!
 - Eure Rückmeldung hat Konsequenzen (z.B. Folien-Redesign)
- Bitte evaluiert Vorlesung und Übungen





Evaluation der Veranstaltung

- Eure Meinung (**Lob/Kritik**) ist uns wichtig!
 - Eure Rückmeldung hat Konsequenzen (z.B. Folien-Redesign)
- Bitte evaluiert **Vorlesung** und Übungen



Typische Rückläuferquote → **2 – 10%**

- Zu wenig für eine sinnvolle Einschätzung
- Aber: Typische Rückläuferquote in EZS → **60 – 80%**

Motivationsanreiz zur Evaluation



- **Traditionell:** Kaffee und Kekse in der letzten Vorlesung
- **Feste Bedingung:** $\geq 60\%$ der ausgegebenen TANs werden evaluiert!



1 Organisatorisches

2 Zustellerkonzepte

3 Rangfolge

4 Ereignisse in eCos

- Events

- Mailbox



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 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



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



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 $D_s \leq i_s \leadsto$ Abtasttheorem
- hohe Abtastfrequenz, Überlastgefahr



Bandweite-bewahrende Zusteller

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



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



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

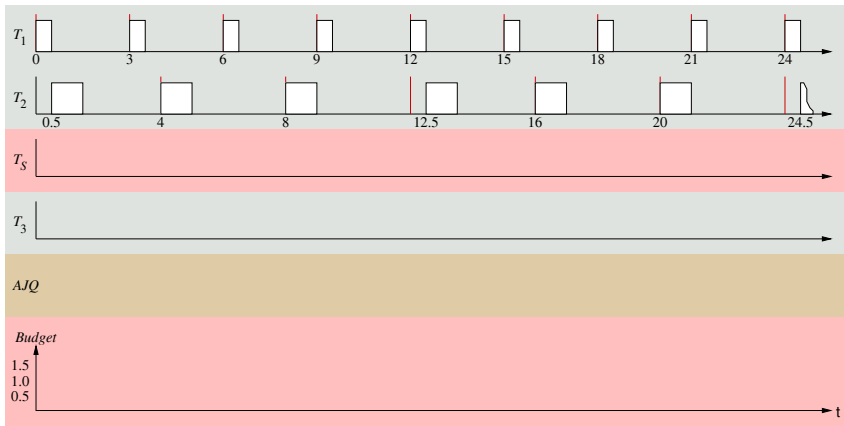
- R1: Initiales Budget ist e_s
- R2: Auffüllzeitpunkt $rt_s = t_b + p_s$, wobei:
 - T_s besitzt Budget, dann $t_b = P_s$ wird tätig
 - T_s hat kein Budget, dann $t_b = P_s$ *ist* tätig und T_s *erhält Budget*
- R3: Budgetberechnung
 - Sobald P_s untätig wird oder T_s kein Budget mehr hat
 - Budget für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

Achtung

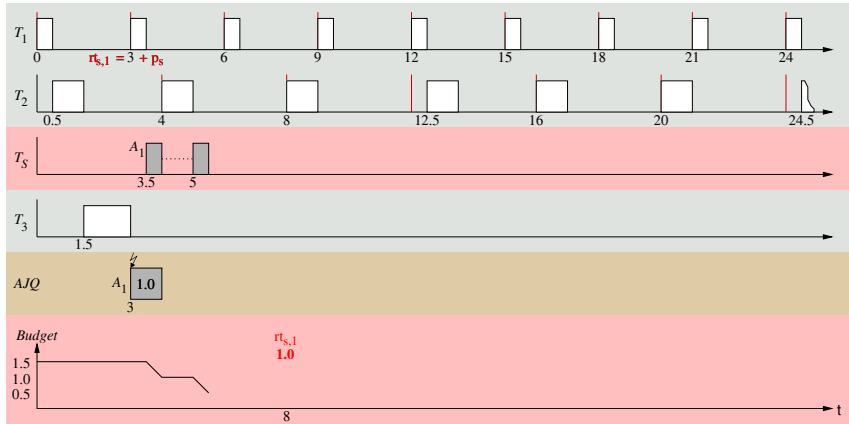
- P_s bezeichnet das **Tasksystem** ab der Priorität s (und höher)
- Im Beispiel: Kleinere Zahl \rightsquigarrow höherer Priorität



Beispiel: SpSL



Beispiel: SpSL

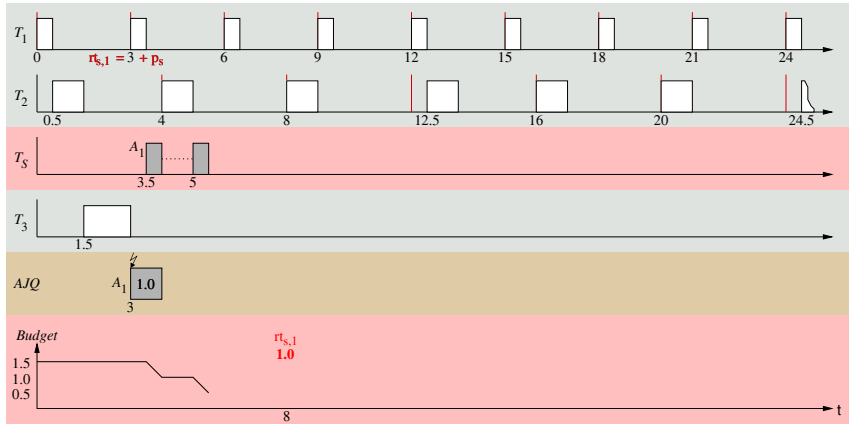


$t_{3.5}$ T_S startet: $t_b = 3 \leadsto rt_{s,1} = 8$ (R2)

- T_1 startet zum Zeitpunkt $t_3 \leadsto P_s$ wurde tätig



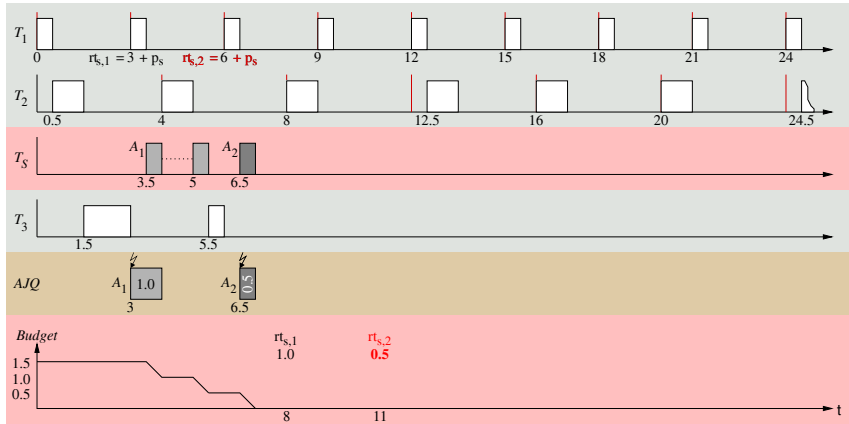
Beispiel: SpSL



$t_{5.5}$ T_s wird untätig, an $rt_{s,1}$ wird 1 Zeiteinheit aufgefüllt (R3)



Beispiel: SpSL

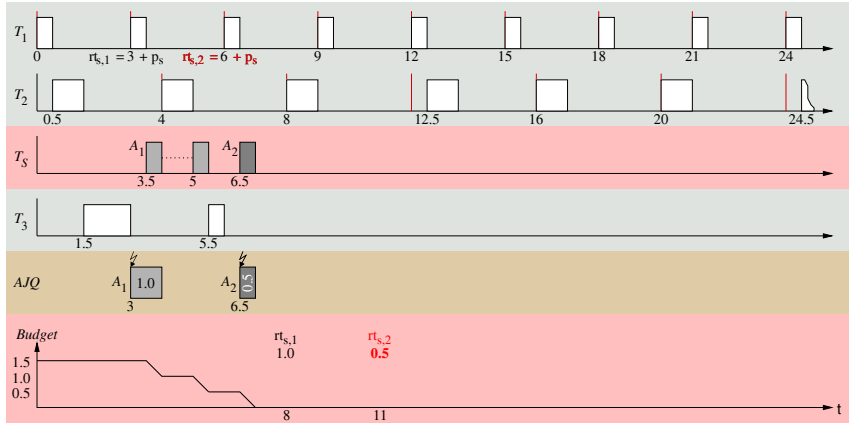


$t_{6.5}$ T_s startet: $t_b = 6 \rightsquigarrow r_{s,2} = 11$ (R2)

- T_1 startet zum Zeitpunkt $t_6 \rightsquigarrow P_s$ wurde tätig



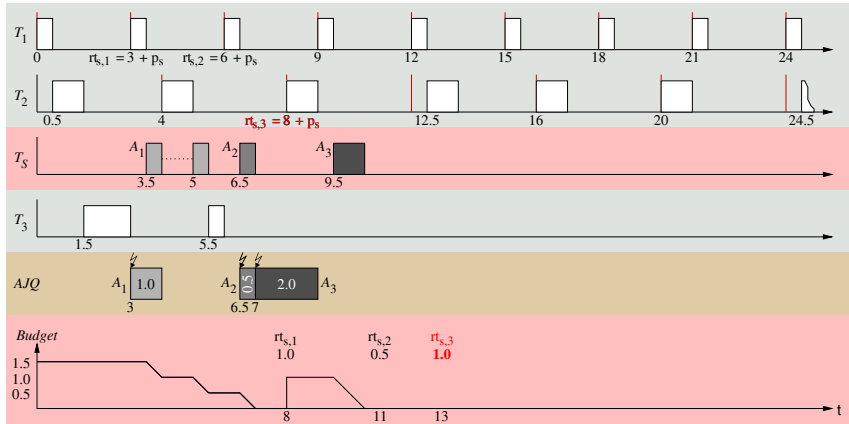
Beispiel: SpSL



t_7 Budget erschöpft, an $rt_{s,2}$ werden 0.5 Einheiten aufgefüllt (R3)



Beispiel: SpSL

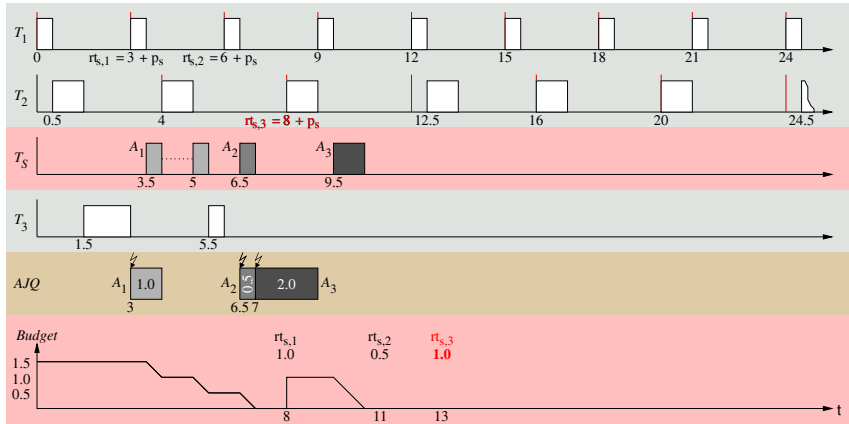


$rt_{s,1} = t_8$ Budgetauffüllung, T_s wird ausführungsbereit

- T_1 und T_2 mit höherer Priorität $\leadsto T_s$ wird noch nicht ausgeführt



Beispiel: SpSL

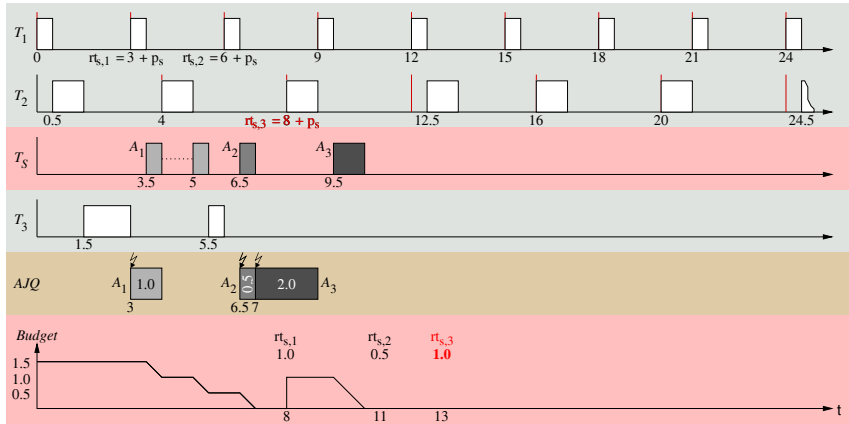


$t_{9.5}$ T_S startet: $t_b = 8 \leadsto rt_{s,3} = 13$ (R2)

■ T_2 startet zum Zeitpunkt $t_8 \leadsto P_s$ wurde tätig



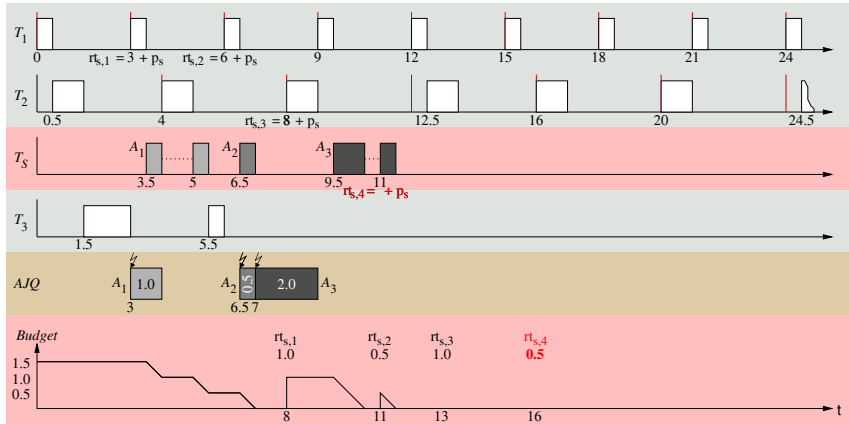
Beispiel: SpSL



$t_{10.5}$ Budget erschöpft, an $rt_{s,3}$ wird 1 Einheit aufgefüllt (R3)



Beispiel: SpSL



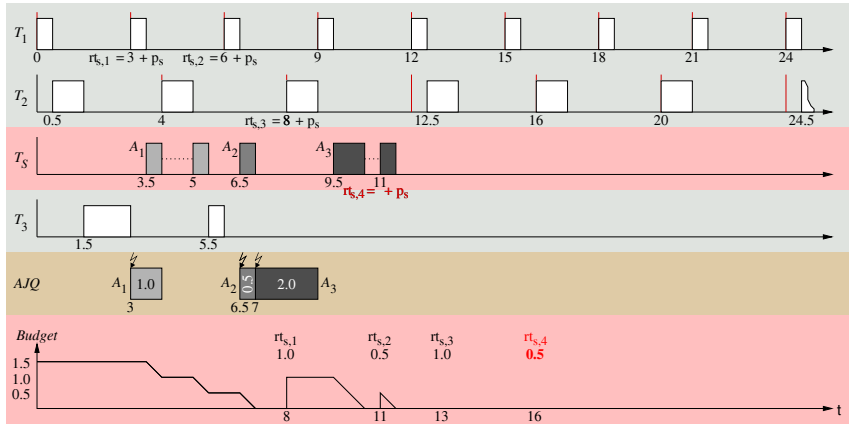
$rt_{s,2} = t_{11}$ Budgetauffüllung, T_s wird ausführungsbereit: $t_b = 11 \leadsto rt_{s,4} = 16$ (R2)

■ T_1 und T_2 nicht ausführungsbereit $\leadsto T_s$ startet

$\leadsto T_s$ wird zum Zeitpunkt t_{11} tätig



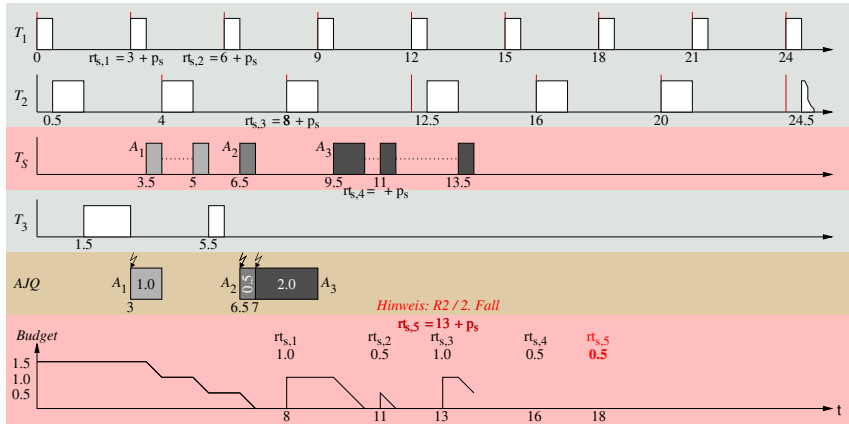
Beispiel: SpSL



$t_{11.5}$ Budget erschöpft, an $rt_{s,4}$ werden 0.5 Einheiten aufgefüllt (R3)



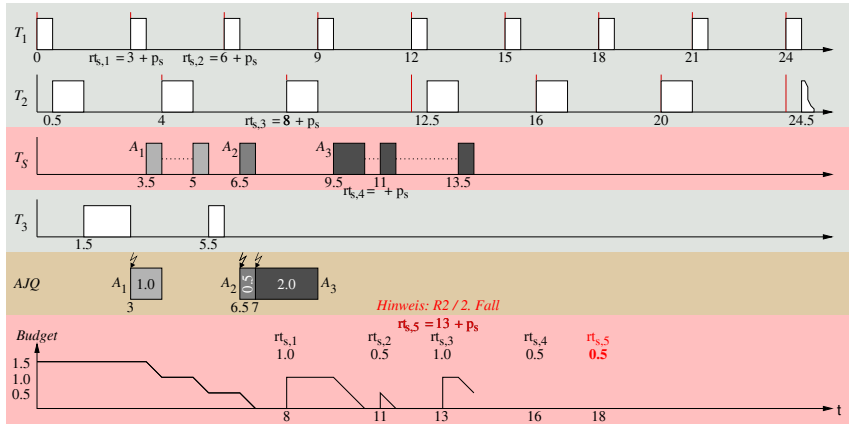
Beispiel: SpSL



$rt_{s,3} = t_{13}$ Budgetauffüllung, T_s wird ausführungsbereit



Beispiel: SpSL

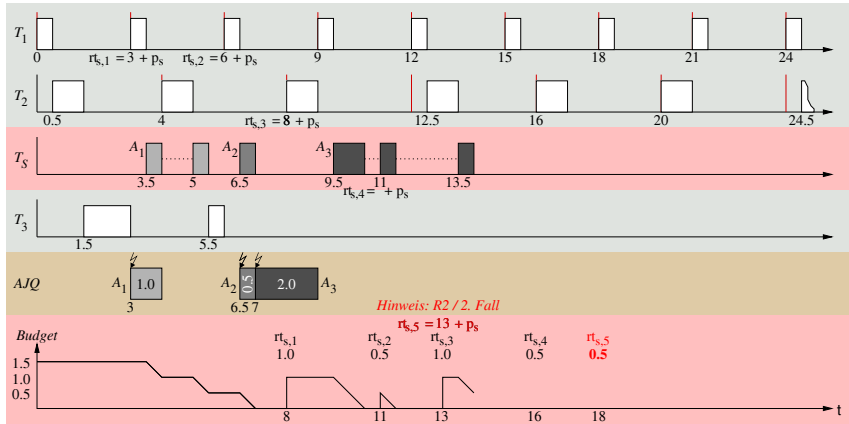


$t_{13.5}$ T_s startet: $t_b = 13 \leadsto rt_{s,5} = 18$ (R2)

- zwar ist P_s bereits seit t_{12} tätig, aber T_s besitzt kein Budget
- ↪ Auffüllzeitpunkt $rt_{s,3}$ dient als Basis für $rt_{s,5}$



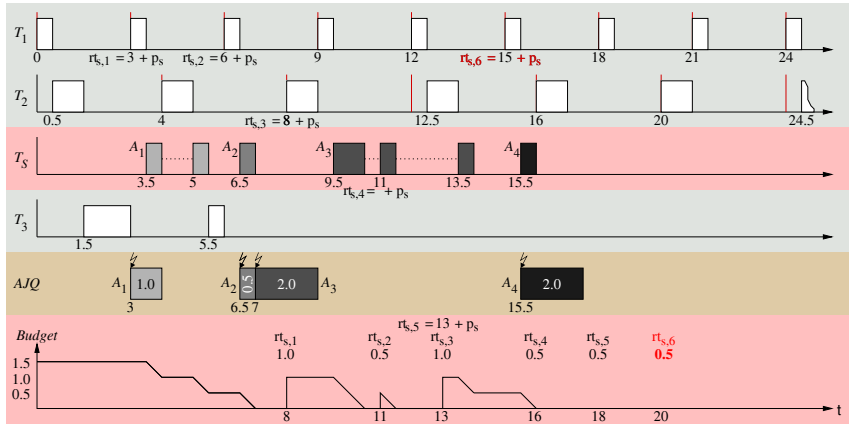
Beispiel: SpSL



t_{14} T_s wird untätig, an $rt_{s,5}$ werden 0.5 Einheiten aufgefüllt (R3)



Beispiel: SpSL

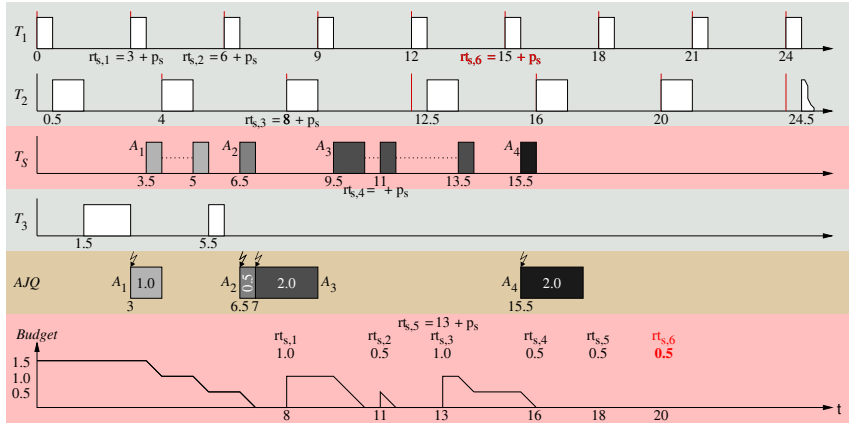


$t_{15.5}$ T_5 startet: $t_b = 15 \leadsto rt_{s,6} = 20$ (R2)

■ T_1 startet zum Zeitpunkt $t_{15} \leadsto P_s$ wurde tätig



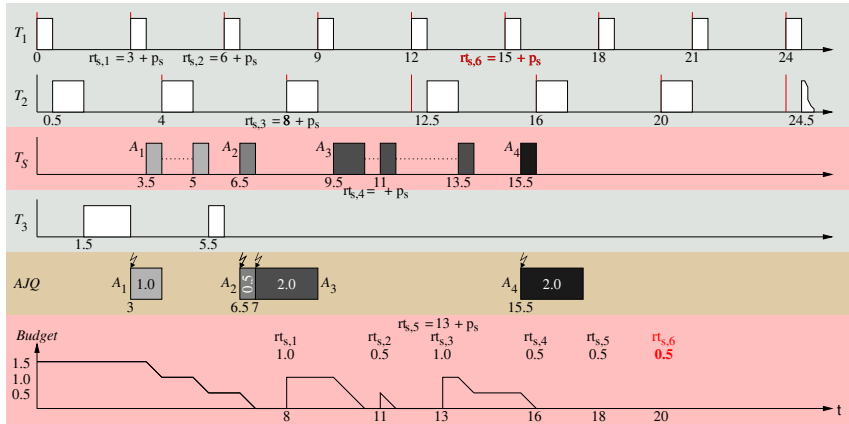
Beispiel: SpSL



t_{16} Budget erschöpft, an $rt_{s,6}$ werden 0.5 Einheiten aufgefüllt (R3)



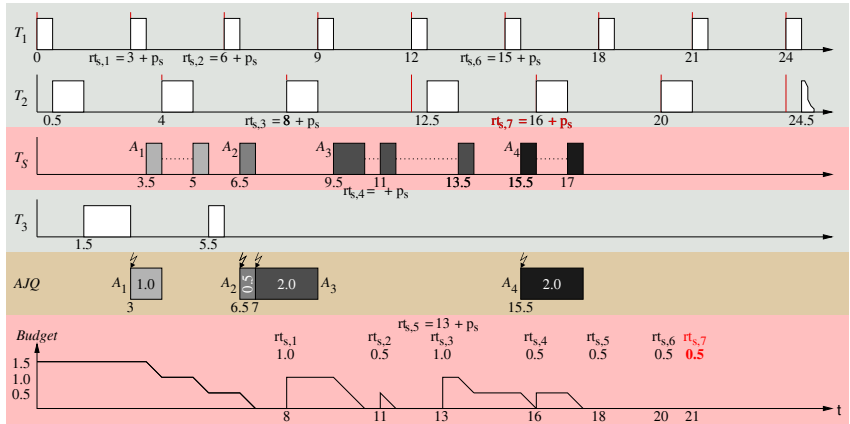
Beispiel: SpSL



$rt_{s,4} = t_{16}$ Budgetauffüllung, T_s wird ausführungsbereit



Beispiel: SpSL

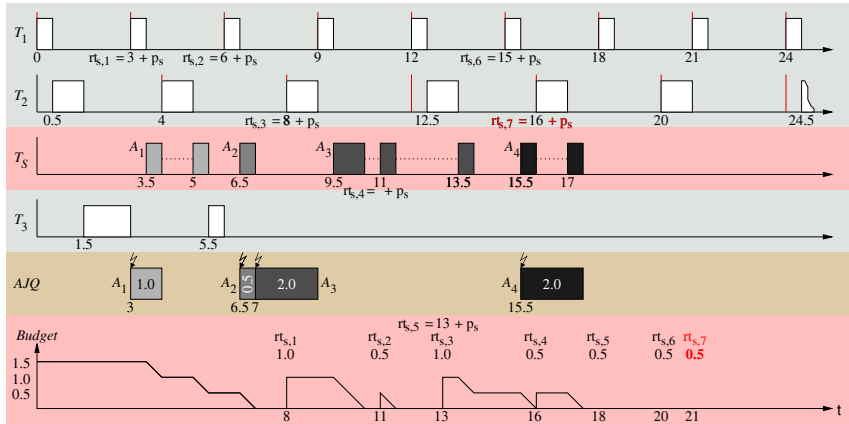


t_{17} T_s startet: $t_b = 16 \leadsto rt_{s,7} = 21$ (R2)

■ T_2 startet zum Zeitpunkt $t_{16} \leadsto P_s$ wurde tätig



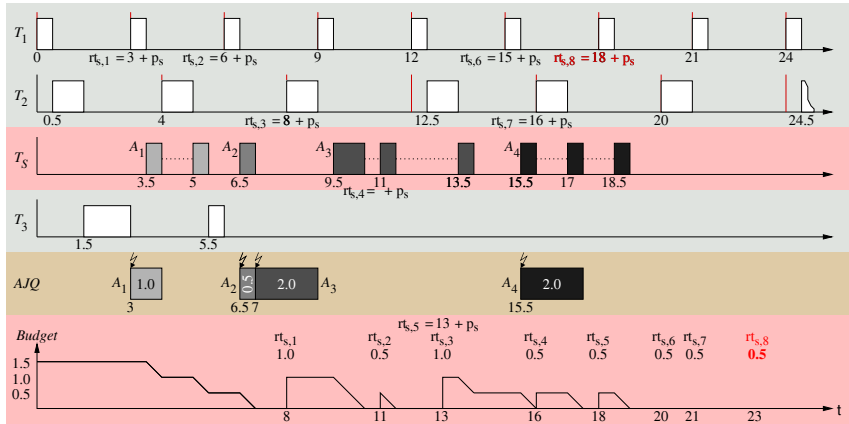
Beispiel: SpSL



$t_{17.5}$ Budget erschöpft, an $rt_{s,7}$ werden 0.5 Einheiten aufgefüllt (R3)



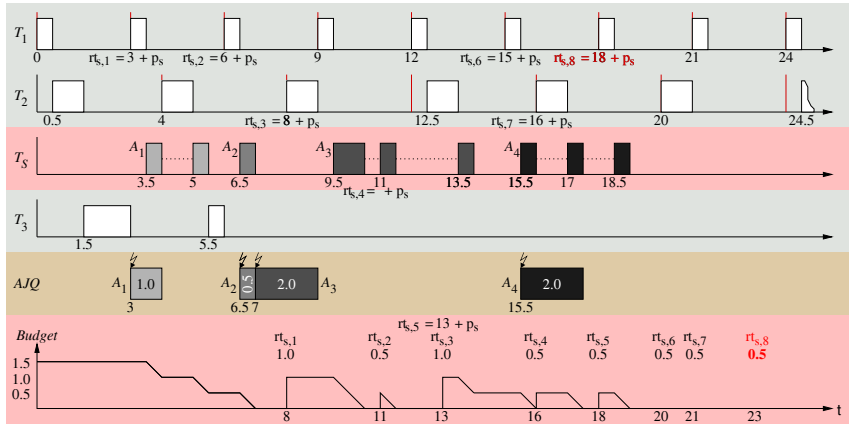
Beispiel: SpSL



$rt_{s,5} = t_{18}$ Budgetauffüllung, T_s wird ausführungsbereit



Beispiel: SpSL

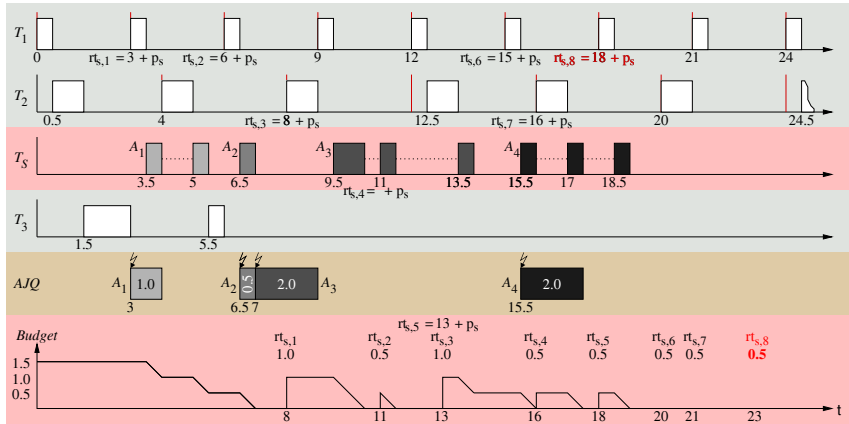


$t_{18.5}$ T_s startet: $t_b = 18 \leadsto rt_{s,8} = 23$ (R2)

■ T_1 startet zum Zeitpunkt $t_{18} \leadsto P_s$ wurde tätig



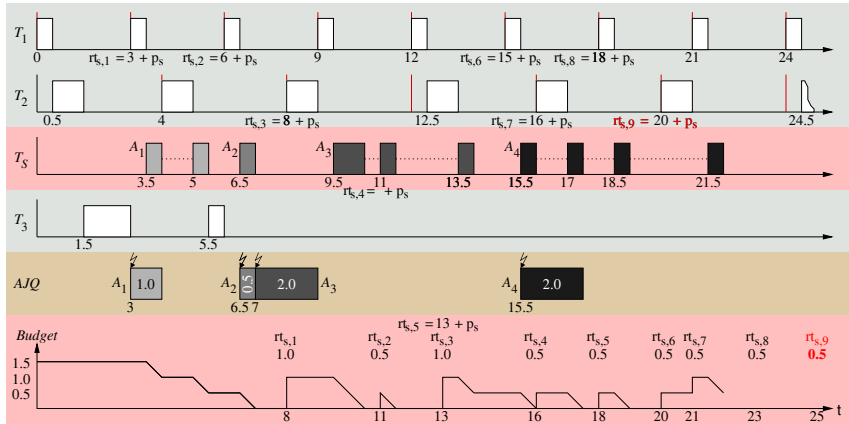
Beispiel: SpSL



t_{19} Budget erschöpft, an $rt_{s,8}$ werden 0.5 Einheiten aufgefüllt (R3)



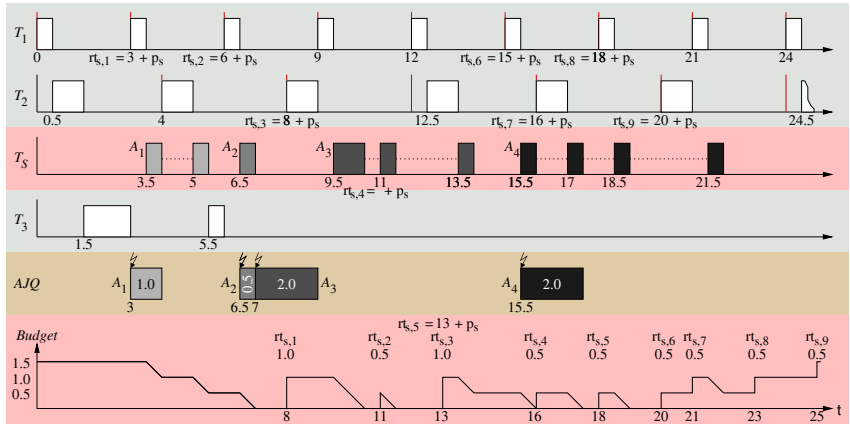
Beispiel: SpSL



t_{23}, t_{25} Budgetauffüllung bis zum ursprünglichen Serverbudget ...



Beispiel: SpSL



t_{23}, t_{25} Budgetauffüllung bis zum ursprünglichen Serverbudget ...



1 Organisatorisches

2 Zustellerkonzepte

3 Rangfolge

4 Ereignisse in eCos

■ Events

■ Mailbox



Rangfolge

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



Rangfolge

- Abhängigkeit von Kontrollfluss \leadsto 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

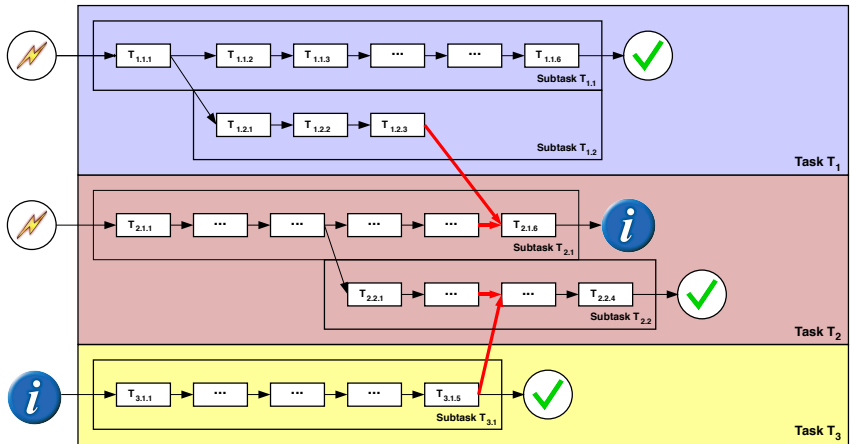


Koordinierung

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



Gerichtete Abhängigkeiten



Gerichtete Abhängigkeiten: **UND**, **ODER** und **zeitliche** Abhängigkeiten

- 1 Organisatorisches
- 2 Zustellerkonzepte
- 3 Rangfolge
- 4 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 \leadsto 32 *Einzelsignale* 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:

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

- Bzw. zurücksetzen:

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

- Auf Signal warten/pollen:

```
4  cyg_flag_value_t cyg_flag_wait/poll(cyg_flag_t* flag,  
5                                     cyg_flag_value_t pattern,  
6                                     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 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.



```
1  static cyg_flag_t flag0;

2  void my_dsr(cyg_vector_t v,
3             cyg_ucount32 c,
4             cyg_addrword_t d){
5      cyg_flag_setbits(&flag0, 0x02);
6  }

7  void user_thread(cyg_addr_t data){
8      while(true) {
9          cyg_flag_wait(&flag0, 0x22,
10                      CYG_FLAG_WAITMODE_OR | CYG_FLAG_WAITMODE_CLR);
11          printf("Event!\n");
12      }
13  }

14  void cyg_user_start(void){
15      ...
16      cyg_flag_init(&flag0);
17      ...
18  }
```





- Zwischen Threads können *Nachrichten* versendet werden
- Konsument erzeugt einen Briefkasten (mailbox) fester Größe
- Produzenten legt Nachrichten dort ab
 - Inhalt: Zeiger auf beliebige Datenstruktur
 - Konsument kann auf *Nachrichtenenpfang* 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 cyg_mbox_create(cyg_handle_t* handle, cyg_mbox* mbox);
```

■ Nachricht verschicken:

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

■ Nachricht empfangen:

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

■ Empfang *und* Versand können blockieren.

■ *try*-Versionen: Würde ich blockieren?

■ *timed*-Versionen: Blockieren, aber nur für bestimmte Zeit.

→ Selbststudium!

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



Versenden von Nachrichten – Beispiel

■ 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 }
```



Besprechung der Übungsaufgabe

„Extended Scope – Teil 2“



Fragen?

