# Echtzeitsysteme

Übungen zur Vorlesung

Simple Scope

**Simon Schuster** **Peter Wägemann**

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

Sommersemester 2022



## Übersicht



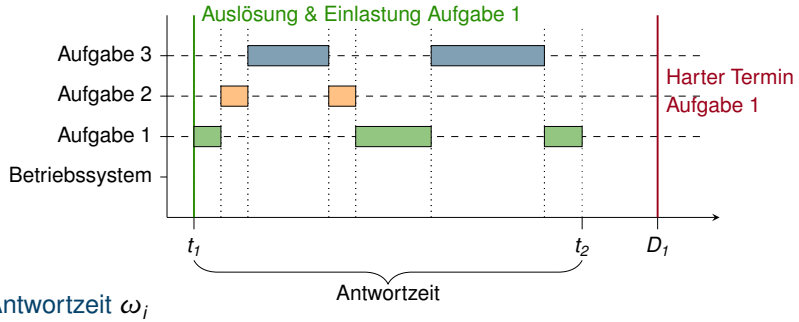


## Übersicht





## Antwortzeitanalyse




- Antwortzeit $\omega_i$
  - Zeitdauer zwischen Auslösezeit und Terminationszeitpunkt
- Idee: Antwortzeitanalyse
  - Terminationszeitpunkt vor dem absoluten Termin $d_i$
  - Antwortzeit $\omega_i$ 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 $\omega_i$ 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



## Beispielsystem

| Aufgabe $T_i$ | Periode $p_i$ ms | WCET $e_i$ ms | Termin $D_i$ ms |
|---|---|---|---|
| $T_1$ | 3 | 1 | 3 |
| $T_2$ | 5 | 1.5 | 5 |
| $T_3$ | 7 | 1.25 | 7 |
| $T_4$ | 9 | 0.5 | 9 |

Alle Aufgaben weisen einen Phasenversatz $\phi_i$ von 0 auf



## Beispiel: Berechnung der maximalen Antwortzeit

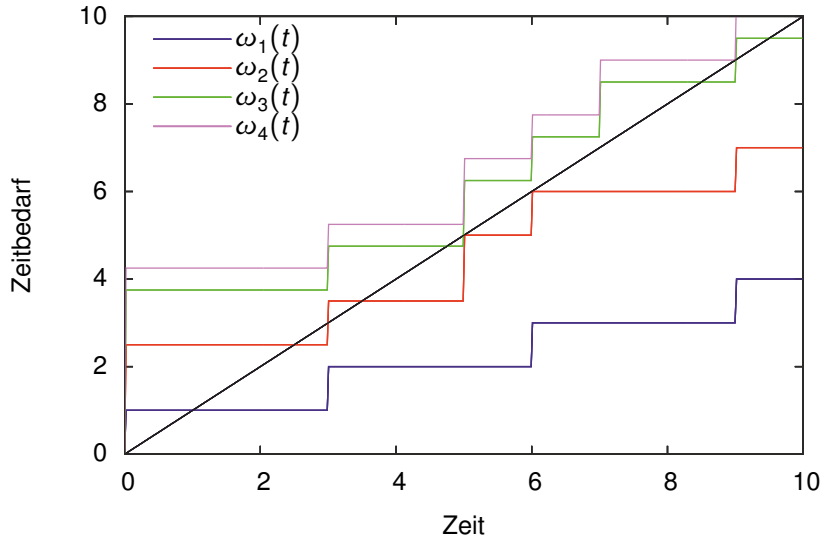
Aufgaben: $T_1 = (3, 1, 3, \phi_1)$, $T_2 = (5, 1.5, 5, \phi_2)$, $T_3 = (7, 1.25, 7, \phi_3)$, $T_4 = (9, 0.5, 9, \phi_4)$

- Antwortzeit $\omega_1$ von $T_1$
  - $\omega_1(3) = 1 \leq 3 \rightsquigarrow$ zulässig
- Antwortzeit $\omega_2$ von $T_2$
  - $\omega_2(3) = 1.5 + \lceil \frac{3}{3} \rceil 1 = 2.5 \leq 3 \rightsquigarrow$ zulässig
- Antwortzeit $\omega_3$ von $T_3$
  - $\omega_3(3) = 1.25 + \lceil \frac{3}{3} \rceil 1 + \lceil \frac{3}{5} \rceil 1.5 = 3.75 > 3$
  - $\omega_3(5) = 1.25 + \lceil \frac{5}{3} \rceil 1 + \lceil \frac{5}{5} \rceil 1.5 = 4.75 \leq 5 \rightsquigarrow$ zulässig
- Antwortzeit $\omega_4$ von $T_4$
  - $\omega_4(3) = 0.5 + \lceil \frac{3}{3} \rceil 1 + \lceil \frac{3}{5} \rceil 1.5 + \lceil \frac{3}{7} \rceil 1.25 = 4.25 > 3$
  - $\omega_4(5) = 0.5 + \lceil \frac{5}{3} \rceil 1 + \lceil \frac{5}{5} \rceil 1.5 + \lceil \frac{5}{7} \rceil 1.25 = 5.25 > 5$
  - $\omega_4(6) = 0.5 + \lceil \frac{6}{3} \rceil 1 + \lceil \frac{6}{5} \rceil 1.5 + \lceil \frac{6}{7} \rceil 1.25 = 6.75 > 6$
  - $\omega_4(7) = 0.5 + \lceil \frac{7}{3} \rceil 1 + \lceil \frac{7}{5} \rceil 1.5 + \lceil \frac{7}{7} \rceil 1.25 = 7.75 > 7$
  - $\omega_4(9) = 0.5 + \lceil \frac{9}{3} \rceil 1 + \lceil \frac{9}{5} \rceil 1.5 + \lceil \frac{9}{7} \rceil 1.25 = 9.00 \leq 9 \rightsquigarrow$ zulässig



## Beispiel: Berechnung der Zeitbedarfsfunktionen

Aufgaben: $T_1 = (3, 1, 3, \phi_1)$, $T_2 = (5, 1.5, 5, \phi_2)$, $T_3 = (7, 1.25, 7, \phi_3)$, $T_4 = (9, 0.5, 9, \phi_4)$




## 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]

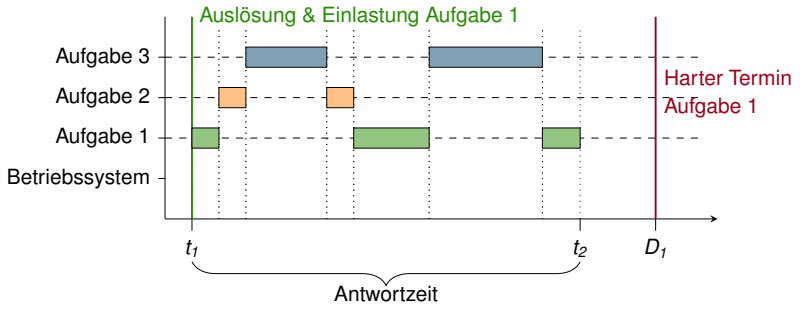
**A6** Die Kosten durch Unterbrechungen, Ablaufplanung und Verdrängung sind vernachlässigbar

**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

⇝ Höherpriorere Aufgaben können jederzeit verdrängen

**A6** Kosten von Unterbrechungen, Ablaufplanung und Verdrängung vernachlässigbar



## Ereignisorientierte Einplanung

(engl. *event-driven scheduling*)

⚠ Einplanung von Arbeitsaufträgen erfolgt zu Ereigniszeitpunkten
- Ihr Auftreten ist nicht (exakt) vorhersehbar
- Ereignisauslöser sind kontrollierte Objekte/andere Arbeitsaufträge

→ Die Ereignisverarbeitung unterliegt einer gewissen Dringlichkeit

☞ Ereignisse haben Prioritäten die dem Ereignisauslöser und/oder der Ereignisverarbeitung zugeordnet sind

Feste Zuordnung ↦ Ereignisverarbeitung/-auslöser
- Arbeitsaufträge erhalten absolute Priorität

Variable Zuordnung ↦ Ereignisverarbeitung
- Arbeitsaufträge erhalten relative Priorität

Auch prioritätsorientierte Einplanung (engl. *priority-driven scheduling*)



## Ereignisorientierter Planer

(engl. *event-driven scheduler*)

- Einplanung ereignisbedingt ausgelöster Arbeitsaufträge resultiert in einer dynamischen Datenstruktur ↦ sortierte Liste

⚠ Kritisch ist die Berechnungskomplexität und wann sie anfällt
- Gekoppelt mit der Einlastung: *online scheduling* (siehe III-2/15 ff)
- Konstant oder variabel, dann jedoch mit oberer Schranke ↦ WCET

→ Zum Auslöse- oder Auswahlzeitpunkt von Arbeitsaufträgen

☞ Priorität bildet den Sortierschlüssel (engl. *sort key*)
- Ergibt sich ggf. erst zum Ereigniszeitpunkt aus der Priorität der von ihm zu verarbeitenden Ereignissen
- Ist eindeutig abzubilden auf einen endlichen Wertebereich

# Aufbau und Berechnungskomplexität

Feste/Dynamische Prioritäten und Ablauftabellen/-listen

- Ablaufliste ↦ Dynamische Datenstruktur
  - Prioritäten entsprechen der Position innerhalb der Ablaufliste
  - Das (relative) Prioritätsgefüge passt sich zur Laufzeit an
  - → Eignung für die Implementierung dynamischer Prioritäten
  - Linearer Berechnungsaufwand zum Auslösezeitpunkt
    - Vorabwissen zur WCET des Sortiervorgangs ist gefordert
  - Nahezu konstanter Berechnungsaufwand zum Auswahlzeitpunkt
    - Aufträge vom Kopf her der (ggf. einfach verketteten) Liste entnehmen
- Ablauftabelle ↦ Statische Datenstruktur
  - Prioritäten werden fest auf Tabellenindizes abgebildet
  - Zur Laufzeit unveränderliches Gefüge absoluter Prioritäten
  - → Eignung für die Implementierung fester Prioritäten
  - Konstanter Berechnungsaufwand zum Auslösezeitpunkt
    - Aufträge durch indizierte Adressierung in die Tabelle aufnehmen
    - Ggf. ist ein Tabelleneintrag eine Auftragliste (FIFO) gleicher Priorität
  - Linearer Berechnungsaufwand zum Auswahlzeitpunkt
    - Vorabwissen zur WCET des Suchvorgangs ist gefordert
    - Tabelleneinträge können leer sein und sind zu überspringen



# Berechnungskomplexität (Forts.)

Ablaufliste vs. Ablauftabelle

**Ablaufliste**

```
Job *list = 0;

void release(Job *item) {
  Job* last = 0, tail = list;

  while(tail && outrank(tail,item)) {
    last = tail;
    tail = last->next;
  }

  if(!last) {
    item->next = list; list = item;
  } else {
    item->next = tail;
    last->next = item;
  }
}

Job* extract() {
  Job* item = list;
  if(item) list = item->next;
  return item;
}
```

`release` $O(n)$

`extract` $O(1)$

**Ablauftabelle**

```
Job* table[Jobs];

void release(Job *item) {
  assert((priority(item) >= 0)
      && (priority(item) <= Jobs - 1));
  item->state = Ready;
}

Job* extract() {
  for(uint slot = 0;slot < Jobs;slot++)
    if(table[slot]->state == Ready) {
       table[slot]->state = Selected;
       return table[slot];
    }
  return 0;
}
```

⚠ Feste Anzahl an Aufträgen

`release` $O(1)$

`extract` $O(n)$



# *Multi-Level-Queue-Scheduler*, MLQ-Scheduler

Häufig anzutreffende Sonderform der Ablauftabelle

- Eine Ablaufliste je Priorität, organisiert als FIFO
- Ablauflisten werden in einer Ablauftabelle verwaltet

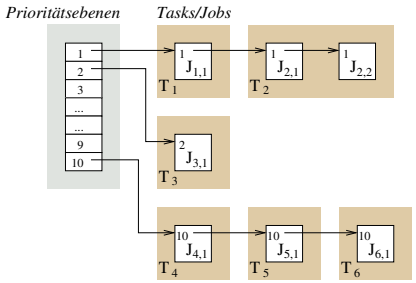
**Multi-Level-Queue**

```
Job* table[Jobs];

void release(Job *item) {
  assert((prio(item) >= 0)
      && (prio(item) <= Jobs - 1));
  item->state = Ready;
  append(table[prio(item)],item);
}

Job* extract() {
  for(uint slot = 0;slot < prios; slot++)
    if(!empty(table[slot])) {
      Job *item = head(table[slot]);
      item->state = Selected;
      return item;
    }
  return 0;
}
```

- Mehrere Tasks pro Priorität
- Mehrere Aufträge pro Task
- Reihenfolge der Auslösung






# Prioritätsorientierter $O(1)$-*Scheduler* !?

Die Tücke liegt oft im Detail...

Auftragauslösung mit konstantem Aufwand $O(1)$ möglich?

1. Ablaufplan ist dynamische Datenstruktur (Tabelle) aus mehreren Prioritätsebenen

   Wartelisten ↦ LIFO

   Warteschlangen ⇝ FIFO
2. Aufträge die über denselben Tabelleneintrag erfasst werden besitzen dieselbe Priorität
   - Sonst könnte LIFO/FIFO Prioritätsverletzung zur Folge haben
3. Anzahl der Tabelleneinträge entspricht mindestens der Anzahl statisch zugewiesener Prioritäten
   - Ggf. werden dann nahezu alle Tabelleneinträge nur einen Auftrag erfassen
   - Abhängig von der Echtzeitanwendung und dem Einplanungsverfahren

Auftragauswahl ist unter diesen Bedingungen nicht in $O(1)$ möglich:

- Leere Tabelleneinträge sind ggf. zu überspringen

## Prioritätsorientierter $O(1)$-*Scheduler* (Forts.) !?

Eine Abwägungsfrage...

⚠ Vorrangsteuerung ist mit grundsätzlichen Konflikt konfrontiert:
- Entweder Auftragauslösung oder Auftragauswahl mit $O(1)$ zu versehen
  - Beides zugleich geht nicht

☞ Für Auftragauslösung in $O(1)$ spricht:
- Ereignisgesteuerte Auslösung benötigen konstante Zeit
  - Z.B. als Folge eines *Interrupts* oder der Zustellung eines *Zeitsignals*
  - Bedeutsam für voll-verdrängbare Systeme

→ Ereignisbedingte Auftragverzögerungen lassen sich exakt bestimmen

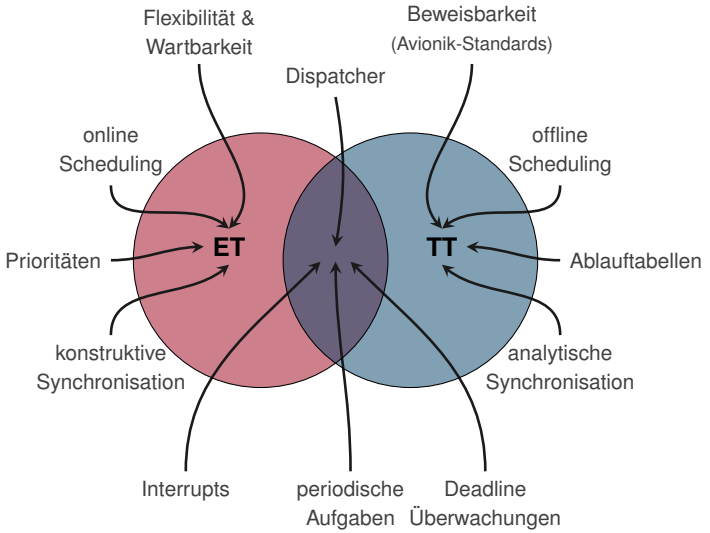
☞ Für Auftragauswahl in $O(1)$ spricht:
- Übergang zum nachfolgenden Auftrag benötigt konstante Zeit
  - Z.B. wenn der aktuelle Auftrag durchgelaufen ist oder blockiert

☞ Linux (bis 2.6), Mach, QNX, ..., VxWorks verhelfen Auftragauslösung zu $O(1)$





## Ereignisgesteuerte vs. zeitgesteuerte EZS






## Übersicht





## Alarme und periodische Aktivierung

eCos-Zähler-Abstraktion

eCos-*Alarme* basieren auf eCos-*Zählern* (Counter[2])

- Anlegen eines Zähler für bestimmtes Ereignis

```
1 void cyg_counter_create(cyg_handle_t* handle,
2                         cyg_counter* counter)
```

- Inkrementieren:

```
1 void cyg_counter_tick(cyg_handle_t counter)
```

  - Zeitgeberunterbrechung (→ DSR)
    → eCos-interne Uhr als Zähler
  - externes Ereignis (Taster, etc.)
    → Zähler wird „in Software“ inkrementiert (→ DSR, → Faden)
- eCos verwaltet Zählerstand intern
- Zugriff auf Zählerstand:

```
1 cyg_tick_count_t cyg_counter_current_value(cyg_handle_t ctr);
2 void cyg_counter_set_value(cyg_handle_t ctr, cyg_tick_count_t val);
```

[2] http://ecos.sourceware.org/docs-1a/ref/kernel-counters.html

# Alarme und periodische Aktivierung

eCos-Uhr

eCos-*Uhren* (Clocks[3]) sind spezialisierte *Zähler*

- Basierend auf *Zeitgeberunterbrechung*
- Festgelegte Zeitauflösung beim Erstellen
- Einzige vorgegebene Uhr: die eCos Uhr

```
1 cyg_handle_t cyg_real_time_clock(void);
```

  - Abfragen:

```
1 cyg_tick_count_t cyg_current_time(void)
```

  - Handle auf Uhr-internen Zähler holen

```
1 void cyg_clock_to_counter(cyg_handle_t clock,
2                           cyg_handle_t* counter);
```

  - Inkrementieren:

```
1 void cyg_counter_tick(cyg_handle_t counter);
```

- Uhr anlegen: `cyg_clock_create()`

[3] http://ecos.sourceware.org/docs-latest/ref/kernel-clocks.html



# Alarme und periodische Aktivierung – 1

Anlegen eines Alarms

eCos-*Alarm*[4] führt Aktion bei Erreichen eines Zählerstandes aus

1 Anlegen:

```
1 void cyg_alarm_create(cyg_handle_t counter,
2                       cyg_alarm_t* alarmfn,
3                       cyg_addrword_t data,
4                       cyg_handle_t* handle,
5                       cyg_alarm* alarm);
```

- `counter`: zugeordneter Zähler
- `alarmfn`: Alarmbehandlung (Funktionspointer)
- `data`: Parameter für Alarmbehandlung
- `handle`: Alarm Handle (vgl. Threaderzeugung)
- `alarm`: Speicher für Alarmobjekt (vgl. Threaderzeugung)

☞ alarmfn wird im DSR-Kontext ausgeführt
→ `cyg_thread_resume()`

[4] http://ecos.sourceware.org/docs-latest/ref/kernel-alarms.html



# Alarme und periodische Aktivierung – 2

eCos-Alarm

eCos-*Alarm*[5] führt Aktion bei Erreichen eines Zählerstandes aus

2 Alarminitialisierung:

```
1 void cyg_alarm_initialize(cyg_handle_t alarm,
2                           cyg_tick_count_t trigger,
3                           cyg_tick_count_t interval);
```

- `alarm`: Alarmhandle
- `trigger`: *Absolute* Zählerticks der *ersten* Aktivierung
  - Nutze `cyg_current_time() + x` ⇝ *Phase*
  - Vorsicht: `cyg_current_time()` kann bei jedem Aufruf anderen Wert liefern
  - trigger muss in der Zukunft liegen. Warum?
- `interval`: Zählerintervall für folgende *periodische* Aktivierungen

3 Alarm freischalten

```
1 void cyg_alarm_enable(cyg_handle_t alarm)
```

[5] http://ecos.sourceware.org/docs-latest/ref/kernel-alarms.html



# tt-eCos

**eCos ist eigentlich *ereignisgesteuert***

⇝ Studienarbeit: Time-Triggered eCos

- Zeitgesteuerte Ausführung von Tasks in Ablauftabellen
- Terminüberwachung mit Ausnahmebehandlung
- Angelehnt an *OSEKtime* (Automobilstandard)

**Ausführliche Dokumentation**

→ Ausarbeitung der Studienarbeit von Michael Lang:
`https://opus4.kobv.de/opus4-fau/files/674/sa_michael_lang.pdf`

## tt-eCos Taskkonstruktion

Ablauftabellen und Tasks werden statisch (global) angelegt:

1. Definition der Ablauftabellen unter Angabe der maximalen Ereigniseinträge. (Makro!)

```
tt_DispatcherTable(string    <Tabellenname>,
                   tt_uint32 <Eintragsanzahl>)
```

2. Definition der Task(s) und Implementierung des Task-Programms

```
tt_Task ( string <Task-Name>) { .. Programm ..}
```

3. Definition des Idletasks und optionaler Hook-Routinen.

```
tt_IdleTask{.. Programm ..}
```



## tt-eCos Initialisierung

Initialisierung zur Laufzeit (in `cyg_user_start()`):

1. Initialisierung der Tasks unter Angabe ihrer Terminüberwachungsmethode

```
tt_InitTask (tt_tasktype       <Task-Name>,
             tt_deadlinemethod <Terminmethode>);
```

2. Initialisierung der Ablauftabelle(n)

```
tt_InitDispatcherTable( tt_dispatchertabletype <Tabellenname>)
```

- Mehrere Tabellen möglich

⇝ Wechsel zur Laufzeit

```
tt_statustype tt_switchtable(tt_dispatchertabletype <Tabellenname>)
```



## tt-eCos Tabelleneinträge

- Definition der Ereignisse der einzelnen Tabellen.

```
tt_bool tt_DispatcherTableEntry(
  tt_dispatchertabletype <Tabellenname>,
  tt_ticktype            <Zeitpunkt>,
  tt_action              <Ereignis>,
  tt_tasktype            < Task-ID> )
```

- Starten des Betriebssystems.

```
void tt_startos( tt_dispatchertabletype <Anfangstabelle> )
```

**Zeitpunkte?**

Schon wieder Ticks...:

```
cyg_resolution_t ttEcos_get_resolution(void)
```



## Terminüberwachung

Für jeden Thread mittels `tt_deadlinemethod` konfigurierbar:

- TT_STRINGENT: Strikte Terminüberprüfung
  - *direkt* nach Ablauf des Termins
- TT_NONSTRINGENT: Nicht-Strikte Terminüberprüfung
  - einem späteren Zeitpunkt
  - Terminverletzung möglich
  - Mehr Flexibilität und Effizienz

Einplanung von Taskstart oder Terminüberwachung (`tt_action`):

- TT_START_TASK: Task-Einplanung
- TT_DEADLINE: Terminüberprüfung

```
tt_bool tt_DispatcherTableEntry(
      tt_dispatchertabletype <Tabellenname>,
      tt_ticktype            <Zeitpunkt>,
      tt_action              <Ereignis>,
      tt_tasktype            <Task-ID>)
```

# Ausführungsmodell in tt-eCos

- *Auftragsorientiertes* Ausführungsmodell ($\neq$ prozessorientiert!)
  - *keine* Endlosschleife in der Anwendung

  → Betriebssystem startet Faden, der Jobs abarbeitet und sich beendet
- Faden blockiert sich nie selbst
  - sonst würde kein Fortschritt mehr stattfinden

  → *run-to-completion-Semantik*

## Vergleich mit eCos: *Prozessorientiertes* Ausführungsmodell

- Anwendungsthread implementiert *Endlosschleife* . . .
- . . . die sich blockiert und auf Ereignis wartet

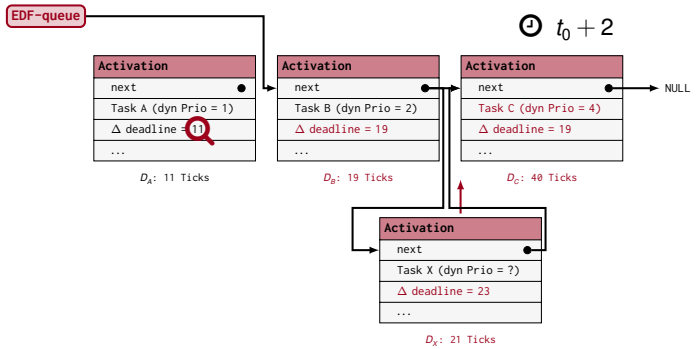


# Übersicht





# Earliest-Deadline First – Implementierungsskizze




## Aktivierungsliste

- verwaltet Aktivierungen der einzelnen Prozesse
- sortiert nach der verbleibenden Zeit zum **relativen Termin**
- speichert Deltas

## `ezs_thread_resume(task_x)`

- Einfügeposition finden



# Earliest-Deadline First – EZS und eCos

- Relative Deadlines von Tasks Speichern und Abrufen
  - Deadline mit einem Handle verknüpfen (während Systeminitialisierung)

```
1 void ezs_set_deadline(cyg_handle_t thread,
2                       cyg_addrword_t deadline);
```

  - Deadline abfragen

```
1 cyg_addrword_t ezs_get_deadline(cyg_handle_t thread);
```

- Prozesspriorität eines Tasks anpassen

```
1 void cyg_thread_set_priority(cyg_handle_t thread,
2                               cyg_priority_t priority);
```

- Eigene Funktionen für Taskstart und Ende bei EDF

  ⇝ "Hook" für Kernfunktionen, durch euch zu erweitern

  ⇝ Userland-Implementierung eines Einplaners

```
1 // Wraps cyg_thread_resume(cyg_handle_t);
2 void ezs_thread_resume(cyg_handle_t handle);
```

```
1 // Wraps cyg_thread_suspend(cyg_handle_t);
2 void ezs_thread_suspend(cyg_handle_t handle);
```

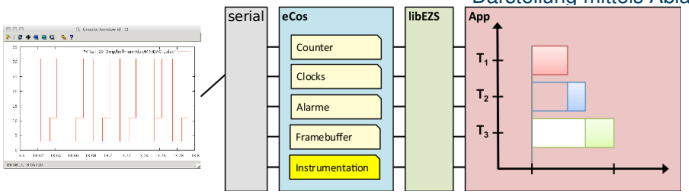
# Übersicht





# Software-Tracing

Darstellung mittels Ablaufgraph




- **Visualisierung der Threads** ⇝ Softwarebasiertes Tracing
- Threadidentifier je nach Schedulertyp: Priorität, ID, Threadhandle
  ⇝ ET: **eindeutige Zuordnung** Faden → Priorität notwendig
- Hyperperiode standardmäßig auf 100 ms kodiert
- Verwendet den `ezs_counter`
  ⇝ zu langes Sampling: Überlauf des Timers
- 256 Scheduling-Entscheidungen bis Plot angezeigt wird
  - Keine Anzeige nach wenigen Hyperperioden (einigen Sekunden)?
    ⇝ Deadlock, Systemabsturz, zu wenige Aktivierungen
- Verwendet serielle Schnittstelle ☞ Cutecom schließen, bei Bedarf:
  `/tmp/$USER-ezs-serial`