Phase 4: Komposition – ProOSEK

Echtzeitsysteme 2 - Vorlesung/Übung

Fabian Scheler
Peter Ulbrich
Wolfgang Schröder-Preikschat

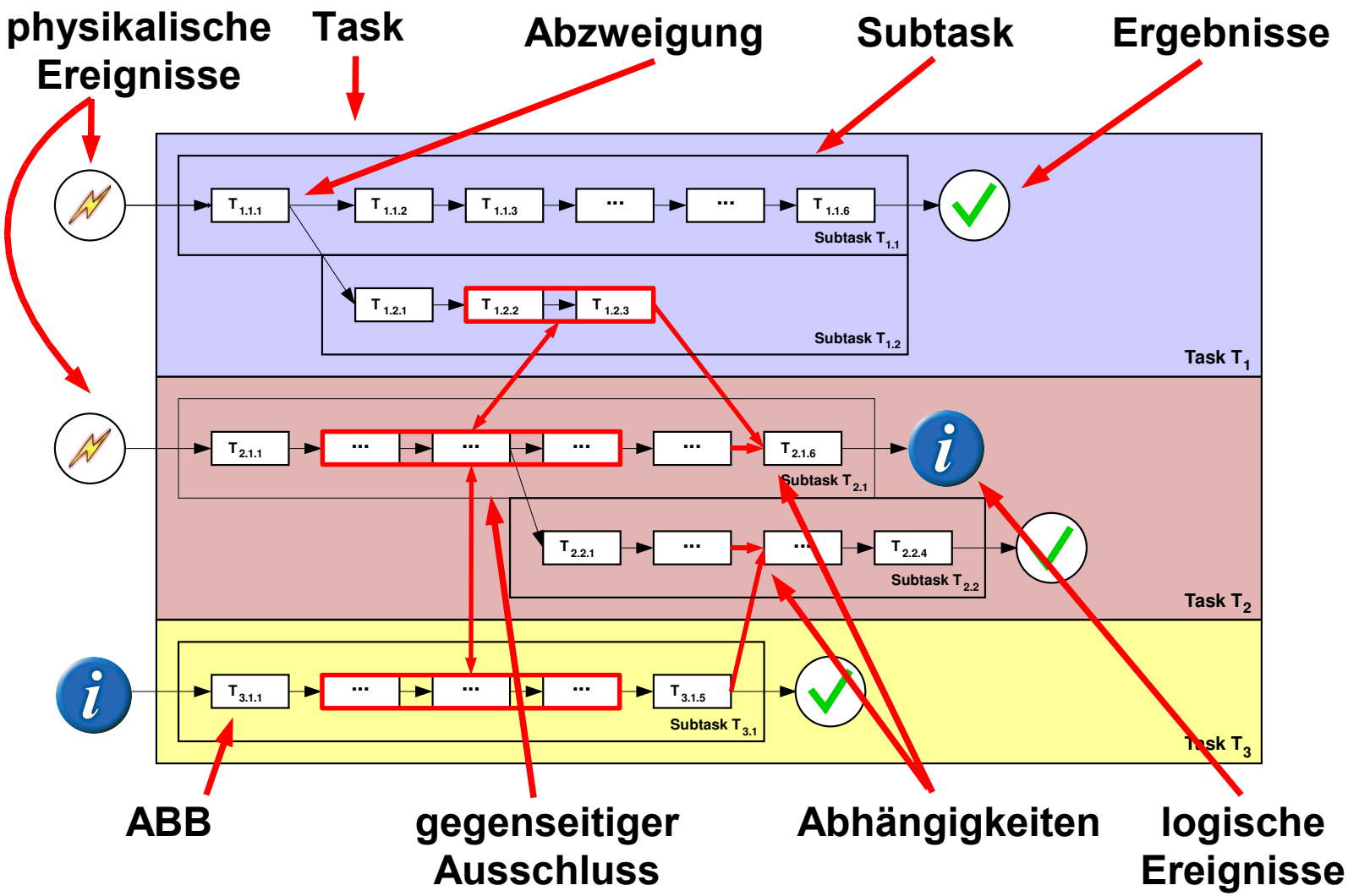
Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme
Friedrich-Alexander Universität Erlangen-Nürnberg

http://www4.cs.fau.de/~{scheler,ulbrich,wosch}
{scheler,ulbrich,wosch}@cs.fau.de

Übersicht

- Wiederholung
  - Ereignisbehandlungen
  - Abbildung
  - Antwortzeitanalyse
- ProOSEK
  - Abbildung
  - Antwortzeitanalyse
  - Hinweise

Wiederholung: Ereignisbehandlungen



Wiederholung: Abbildung

- Implementierung von
  - Tasks, Subtasks und ABBs
  - Konkatenation von ABBs
  - gegenseitigem Ausschluss
  - Verzweigungen
  - Abhängigkeiten
- Implementierung mit den Mitteln des Betriebssystems

Wiederholung: Antwortzeitanalyse

- **Auslösezeitpunkt**
  - Zeitpunkt an dem ein Ereignis eintritt
- **Antwortzeit**
  - Abschlusszeitpunkt der Behandlung des Ereignisses
  - mit anderen Worten: das Ende eines Subtask

- **maximale Antwortzeit**:

$$w_i(t) = e_i + \sum_{k=1}^{i-1} ceiling(\frac{t}{p_k}) e_k$$

$$w_i(t) \leq t \; ; t = jp_k \; ; k = 1, 2, \ldots, i \; ; j = 1, 2, \ldots, floor(min(d_i, p_i)/p_k)$$

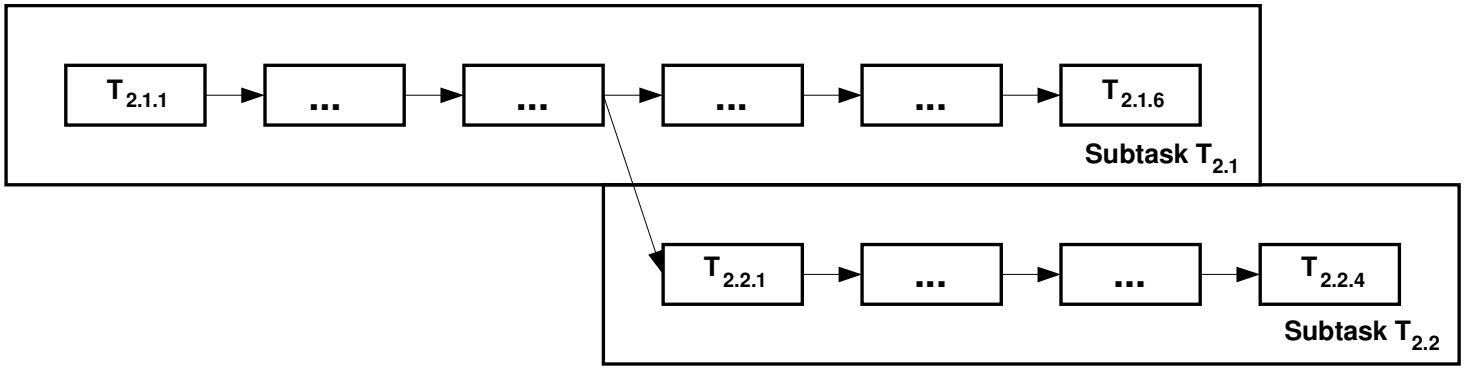
- hinreichendes und notwendiges **Kriterium für Planbarkeit**:

$$max_t w_i(t) \leqslant d_i \; \forall \; Taskst_i$$

ProOSEK: Abbildung

- Verzweigungen
- Abhängigkeiten
- gegenseitiger Ausschluss
- aperiodische Ereignisbehandlungen

ProOSEK: Verzweigungen

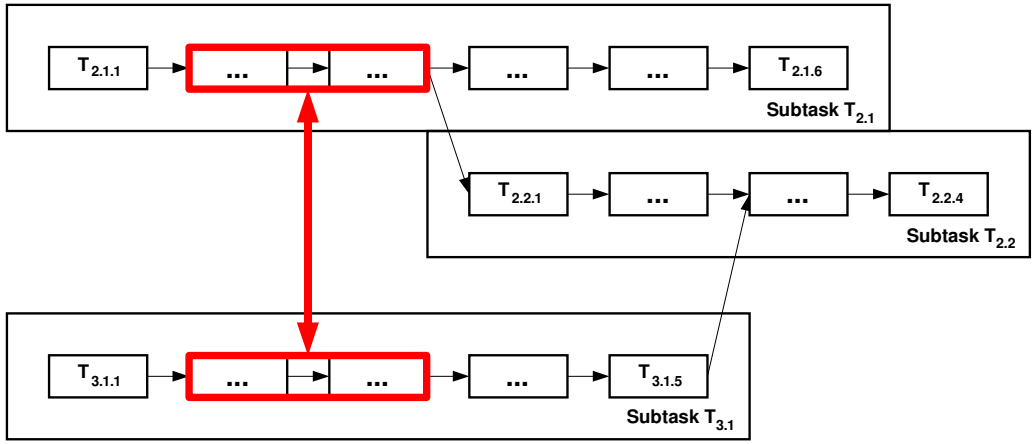


- Task spaltet sich in unabhängige Kontrollflüsse
- Subtasks *münden nicht wieder zurück*

```
TASK(A) {
   ...
   ActivateTask(B);
   ...
   ChainTask(C);
}
```

- ein Subtask entsteht nur wenn
  - Task A nicht-präemptiv ist oder
  - Priorität(B) < Priorität(A)
- nieder-priore Subtasks zweigen immer von hoch-prioren Subtasks ab

ProOSEK: gegenseitiger Ausschluss



- **Ansatz:** nicht verdrängbarer Task
  - TASK B: nicht-präemptiv und Prio(B) > Prio(A)
  - TASK A: verdrängbar
  - ✗ **Overhead durch Kontextwechsel**

```
TASK(A) {
  ...
  ActivateTask(B);
  ...
  TerminateTask();
}
```

```
TASK(B) {
  ...
  criticalStuff();
  ...
  TerminateTask();
}
```

ProOSEK: gegenseitiger Ausschluss

- **Ansatz:** *Kernelized Monitor*
  - spezielle Resource: RES_SCHEDULER
  - verhindert Verdrängung komplett
  - Synchronisation nur auf TASK-Ebene
  - ✔ kein Kontextwechsel
  - ✗ **Blockade potentiell unbeteiligter TASKs**

```
TASK(A) {
  ...
  GetResource(RES_SCHEDULER);
  doCriticalStuff();
  ReleaseResource(RES_SCHEDULER);
  ...
  TerminateTask();
}
```

ProOSEK: gegenseitiger Ausschluss

- **Ansatz:** Schlossvariablen (RESOURCEn)
  - OSEK Priority Ceiling Protocol (= *Stack-based Priority Ceiling*)
  - RESOURCEn statisch konfiguriert
  - Implementierung (optional) auch für ISR Cat. 2
  - ✔ keine unnötigen Blockaden
  - ✗ für ISR Cat. 2 häufig mittels `Suspend-/ResumeOSInterrupts()`

```
TASK(A) {
  ...
  GetResource(LOCK);
  doCriticalStuff();
  ReleaseResource(LOCK);
  ...
  TerminateTask();
}
```

```
ISR(ISRA) {
  ...
  GetResource(LOCK);
  doCriticalStuff();
  ReleaseResource(LOCK);
  ...
  return;
}
```

ProOSEK: gegenseitiger Ausschluss

- **Ansatz:** harte Synchronisation
  - sperren der Unterbrechungen
  - Selektiv:
    1) ISR Cat. 1 + ISR Cat. 2 → `Enable-/DisableAllInterrupts()`
    2) ISR Cat. 2 → `Resume-/SuspendOSInterrupts()`
  - immer **paarweise**, Schachtelung möglich
  - ✗ **verhindert jegliche Entgegennahme von Ereignissen**
    - Ereignisse können verloren gehen (auch Timerinterrupts)

```
TASK(A) {
  ...
  SuspendOSInterrupts();
  doCriticalStuff();
  ResumeOSInterrupts();
  ...
  TerminateTask();
}
```

```
ISR(ISRA) {
  ...
  DisableAllInterrupts();
  doCriticalStuff();
  EnableAllInterrupts();
  ...
  return;
}
```

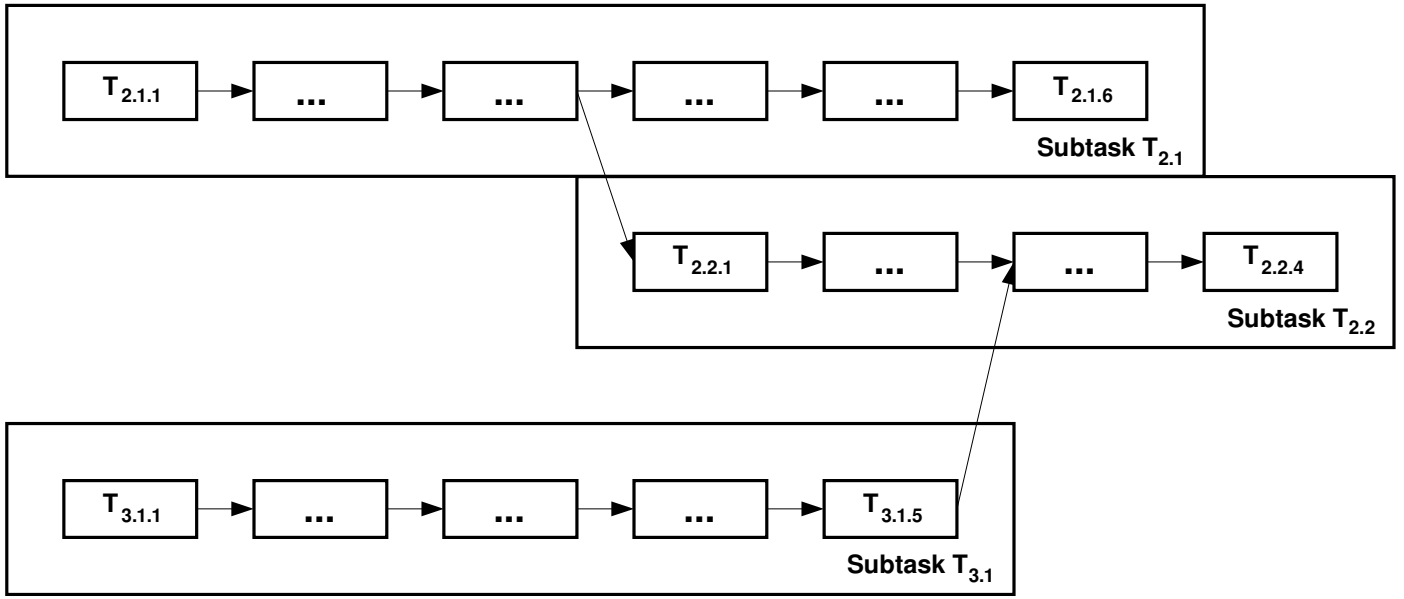
ProOSEK: gegenseitiger Ausschluss

- **Ansatz:** Phasenverschiebung
  - Subtasks werden in geeigneter Reihenfolge gestartet
  - Auslösung erfolgt *„zeitgesteuert“* durch einen Alarm
  - → `SetRelAlarm()` bzw. `SetAbsAlarm()`
    - ALARM A1 aktiviert Subtask $T_{3.1}$, ALARM A2 Subtask $T_{2.1}$
    - `phase2` = WCET(Subtask $T_{3.1}$) → **keine Überlappung!**
    - beide Subtasks haben eine identische Periode `period`
    - **keine gesonderte Synchronisation notwendig!**
  - → **Achtung:** gegenseitiger Ausschluss ist nicht mehr sichtbar!

```
TASK(Init) {
  ...
  SetRelAlarm(A1,0,period);
  SetRelAlarm(A2,phase2,period);
  ...
  TerminateTask();
}
```

```
Task(Task3) {
  ...
  doCriticalStuff();
  ...
  TerminateTask;
}
```

ProOSEK: Abhängigkeiten



- Arten von Abhängigkeiten
  - ODER-Abhängigkeiten
  - UND-Abhängigkeiten
- Nachfolger hängt von mehreren Vorgängern ab

ProOSEK: ODER-Abhängigkeiten

- **Ansatz:** mehrfache Auslösbarkeit des Nachfolgers
  - jeder Vorgänger löst jeweils den Nachfolger aus
- **zu beachten:**
  - Prioritätenvergabe
  - Anzahl der Auslösungen

ProOSEK: UND-Abhängigkeiten

- **Ansatz:** EVENTs
  - Nachfolger wartet zyklisch bis mehrere EVENTs gesetzt sind
  - Vorgänger setzen jeweils ein EVENT

```
Task(Successor) {
  EventMaskType eventsAwaited;
  EventMaskType eventsNotSet;
  eventsAwaited = e1 | e2 | e3;
  eventsNotSet = eventsAwaited;

  do {
    EventMaskType eventsSet = 0;
    WaitEvent(eventsNotSet);
    GetEvent(Successor,&eventsSet);
    eventsNotSet ^=
      (eventsSet & eventsAwaited);
  } while(eventsNotSet != 0);
  ...
  TerminateTask();
}
```

```
Task(Predecessor1) {
  ...
  SetEvent(Successor,e1);
  TerminateTask()
}
Task(Predecessor2) {
  ...
  SetEvent(Successor,e2);
  TerminateTask()
}
Task(Predecessor3) {
  ...
  SetEvent(Successor,e3);
  TerminateTask()
}
```

ProOSEK: UND-Abhängigkeiten

- **zu beachten:**
  - Overhead durch Kontextwechsel
  - erschwert Antwortzeitanalyse
  - Prioritätenvergabe

ProOSEK: UND-Abhängigkeiten

- **Ansatz:** ALARM & USERCOUNTER
  - Nachfolger wird von einem ALARM ausgelöst
  - ALARM wird durch einen USERCOUNTER gesteuert
  - Vorgänger inkrementieren den USERCOUNTER

```
TASK(Successor) {
  ...
  SetRelAlarm(alarm,3,0);
  TerminateTask();
}

Task(Predecessor1) {
  ...
  AdvanceCounter(counter);
  TerminateTask()
}

Task(Predecessor2) {
  ...
  AdvanceCounter(counter);
  TerminateTask()
}

Task(Predecessor3) {
  ...
  AdvanceCounter(counter);
  TerminateTask()
}
```

ProOSEK: UND-Abhängigkeiten

- **zu beachten:**
  - Prioritätenvergabe
  - Wann soll der ALARM ablaufen?
    - Vorgänger können unterschiedliche Periode besitzen
  - der ALARM muss initialisiert werden

ProOSEK: aperiodische Ereignisse

- verfügbare Methoden: *deferred handling*
- Vorgehen
  - Ereignis wird initial durch ISR behandelt
  - ISR löst eine TASK aus
  - Auslösung *speichert* das Ereignis
- nicht implementierbar:
  - *deferred server*
  - *sporadic server*
  - ...
- zu beachten:
  - minimale Zwischenankunftszeit
  - pessimistische Antwortzeitanalyse
  - Anzahl der Auslösungen

ProOSEK → eCos

- Verzweigungen
  - einen Faden aktivieren
    - **`cyg_thread_create`**
    - **`cyg_thread_resume`**
- gegenseitiger Ausschluss
  - **Schlossvariablen**
    - **`cyg_mutex_lock`** und **`cyg_mutex_unlock`**
    - unterstützt eine Variante von PIP bzw. PCP
  - **Semaphore**
    - **`cyg_semaphore_wait`** und **`cyg_semaphore_post`**
  - **Kernelized Monitor**
    - **`cyg_scheduler_lock`** und **`cyg_scheduler_unlock`**
    - verhindert die Ausführung von DSRs
  - **Unterbrechungen sperren**
    - **`cyg_interrupt_disable`** und **`cyg_interrupt_enable`**

ProOSEK → eCos

- ODER-Abhängigkeiten
  - **Semaphore** → siehe „gegenseitiger Ausschluss“
  - **Flags**
    - **`cyg_flag_wait`**, **`cyg_set_bits`** und **`cyg_mask_bits`**
    - einzelne Bits lassen sich mit ODER bzw. UND verknüpfen
  - **Mail Boxes**
    - **`cyg_mbox_get`** und **`cyg_mbox_put`**
    - **Achtung:** hier kann auch der Sender blockieren
    - kombiniert Kontroll- und Datenflussabhängigkeit
  - **Achtung:** Fäden sind in eCos **nicht mehrfach aktivierbar**
    - → der Faden muss **alle eingetretenen Vorgänger** behandeln
    - z.B. bis der Semaphor den Wert 0 erreicht hat

ProOSEK → eCos

- UND-Abhängigkeiten
  - **Semaphore** → siehe „gegenseitiger Ausschluss“
  - **Flags** → siehe „ODER-Abhängigkeiten“
  - **Condition Variables**
    - **`cyg_cond_wait`**, **`cyg_cond_signal`** und **`cyg_cond_broadcast`**

- gegenseitiger Ausschluss, UND-/ODER-Abhängigkeiten
  - **Phasenverschiebung**
    - Counter, Clocks und Alarme
    - Verwendung analog zu ProOSEK

ProOSEK: Ablaufplanung

- Begriffe
- Algorithmus
- einfache Antwortzeitanalyse
- erweiterte Antwortzeitanalyse

Begriffe

- Task, Subtask, ABB
- **Ausführungspfad**

  Ein Ausführungspfad ist ein Kontrollfluss innerhalb eines Subtask und besteht aus ABBs die sequentiell hintereinander ausgeführt werden. Derselbe ABB kann auch mehrmals in einem Ausführungspfad auftreten.
- **Abschnitt**

  Ein Abschnitt ist eine Menge von ABBs, die in einem Ausführungspfad direkt aufeinander folgen.
- **WCET** der Subtask $T_{n.m}$:

$$e_{n.m} = max_{Pa_{n.m.l} \in T_{n.m}} \sum_{T_{n.m.i} \in Pa_{n.m.l}} e_{n.m.i} k_{n.m.l}(T_{n.m.i})$$

$Pa_{n.m.l}$	Pfad l der Subtask $T_{n.m}$
$k_{n.m.l}(T_{n.m.i})$	Auftreten des ABB $T_{n.m.i}$ im Pfad $Pa_{n.m.l}$

Begriffe

- Termin, Periode, minimale Zwischenankunftszeit
- Planbarkeit
- **kritischer Zeitpunkt**

  Eine Subtask erreicht ihre **maximale Antwortzeit** genau dann, wenn sie ihre maximale Ausführungszeit erreicht und an einem **kritischen Zeitpunkt** ausgelöst wird.

Algorithmus

- OSEK
  - Vorrangsteuerung
  - MLQ-Scheduler
  - statisch bestimmte Prioritäten

→ zur Auswahl stehende Algorithmen
  - RMA
  - DMA

→ Entscheidung: DMA
  - Begründung: weniger restriktiv als RMA

Einfach Ablaufplanung

- trivialer Fall
- TASKs
- kritische Abschnitte
- ALARMe
- Self-Suspension
- beliebige Termine

Trivialer Fall

- **Ereignisse**
  - **strikt periodisch**
  - **Termine ≤ Periode**
  - **keine Abhängigkeiten**
- **OSEK**
  - ausschließlich **ISRs Cat. 1** und **Cat. 2**
  - ISRs verhalten sich **präemptiv**
  - keine Sperrung von Unterbrechungen
  - ausreichende Anzahl von Prioritätsebenen
  - **keine**
    - **TASKs**
    - **RESOURCEn**
    - **ALARMe**
    - **EVENTs**

Trivialer Fall

- **allgemein:**

$$r_{n.m} = e_{n.m} + i_{n.m} + b_{n.m}$$

$i_{n.m}$ — Zeit in der $T_{n.m}$ durch höher-priore Subtasks unterbrochen wird

$b_{n.m}$ — Zeit in der $T_{n.m}$ durch nieder-priore Subtasks blockiert wird

- **hier:**

$$i_{n.m} = \sum_{T_{i.j} \in H_{n.m}} ceiling(r_{n.m} / p_i) e_{i.j}$$

$H_{n.m}$ — Menge der höher-prioren Subtasks

$ceiling(r_{n.m} / p_i)$ — Auftreten des Ereignisses T_i während der Ausführung der Subtasks $T_{n.m}$

- **ingesamt:**

$$r_{n.m} = e_{n.m} + \sum_{T_{i.j} \in H_{n.m}} ceiling(r_{n.m} / p_i) e_{i.j} + b_{n.m}$$

TASKs

- **keine nicht-präemptiven** TASKs
- kein TASK sperrt die Unterbrechungen
- ein TASK **aktiviert maximal einen Nachfolger**
- **keine RESOURCEn, EVENTs oder `Schedule()`**

- **Problematik:**
  - **ISRs nieder-priorer Ereignisse** unterbrechen
  - **TASKs hoch-priorer Ereignisse**
  - → ***Rate-monotonic Priority Inversion***

TASKS

- hier:

$$i_{n.m} = \underbrace{\sum_{T_{i.j} \in H_{n.m}} ceiling(r_{n.m}/p_i) e_{i.j}}_{(1)} + \underbrace{\sum_{T_{i.j} \in L_{n.m}} ceiling(r_{n.m}/p_i) e^{isr}_{i.j}}_{(2)}$$

(1) Verdrängung durch höher-priore Subtasks

(2) Verdrängung durch ISRs nieder-priorer Subtasks

$L_{n.m}$ Menge der nieder-prioren Subtasks

$e^{isr}_{i.j}$ Ausführungszeit des ISR-Anteils der Subtask $T_{n.m}$

Gegenseitiger Ausschluss

- **Implementierungsvarianten**
  - gesperrte Unterbrechungen
  - nicht-präemptive TASKs
  - RESOURCEn

- hier: $b_{n.m} = b^{non}_{n.m} + b^{isr}_{n.m} + b^{res}_{n.m}$

$b^{non}_{n.m}$ Blockade durch nicht-präemptive TASKs

$b^{isr}_{n.m}$ Blockade durch gesperrte Unterbrechungen

$b^{res}_{n.m}$ Blockade durch belegte RESOURCEn

Kritische Abschnitte

- Begründung
  - TASK A belegt eine RESOURCE
  - TASK A wird von einer nicht-präemptiven TASK B verdrängt
  - ISR C unterbricht TASK B und sperrt die Interrupts
- Bestimmung der einzelnen Komponenten durch
  - ABBs
  - Ausführungspfade
  - Abschnitte

ALARMe

- ALARM werden von COUNTERn gesteuert
  - USERCOUNTER betrachten wir hier nun nicht
  - hier: TIMERCOUNTER
- TIMERCOUNTER
  - Hardware-Timer
  - löst zyklisch Interrupt aus
  - je nach Zählerstand läuft ALARM ab
- Problem
  - Timer-ISR
    - unterbricht andere Subtransaktionen
    - welche Priorität soll die ISR haben

ALARMe

- **insgesamt:**

$$w_{n.m}(t) = e_{n.m} + tc^{isr}_{n.m}(t) + i_{n.m} + b_{n.m}$$

$tc^{isr}_{n.m}(t)$ Unterbrechungen der Subtask $T_{n.m}$ durch Timer-ISRs

- **hier:**

$$tc^{isr}_{n.m}(t) = \sum_{tc \in TC} tc^{tc}_{n.m}(t)$$

$tc^{tc}_{n.m}(t) = 0$ Subtask $T_{n.m}$ enthält keine TASKs und alle ISRs haben eine höhere Priorität als der ISR des Hardware-Timers

$tc^{tc}_{n.m}(t) = ceiling(t / p_{TC}) e^{isr}_{TC}$ sonst

TC Menge aller Timer-Counter

- **Priorität der Timer-ISR**
  - so hoch wie nötig, so niedrig wie möglich
  - **es darf kein Tick verloren gehen**
  - iterative Annäherung

Self-Suspension

- in OSEK durch
  - Warten auf ein EVENT: `WaitEvent()`
  - Aufruf des Schedulers: `Schedule()`

- **Problematik:**
  - Blockade durch nieder-priore Subtasks kann nach jedem Verzicht auf den Prozessor erneut auftreten

- **insgesamt:** $w_{n.m}(t) = e_{n.m} + t\,c^{isr}_{n.m}(t) + i_{n.m} + s_{n.m} + (s^{k}_{n.m} + 1)\,b_{n.m}$

  $s_{n.m}$ so lange verzichtet Subtask $T_{n.m}$ auf den Prozessor

  $s^{k}_{n.m}$ so oft verzichtet Subtask $T_{n.m}$ auf den Prozessor

Self-Suspension

- Alternative: aktives Warten

```
Task(UglyTask) {
   ...
   SetRelAlarm(alarm1,time_to_wait,0);
   while(GetAlarm(alarm1) != E_OS_NO_FUNC);
   ...
   TerminateTask();
}
```

- **Problematisch**
  - erhöht Prozessorauslastung unnötigerweise
  - nieder-priore Subtasks werden lange verzögert
  - → kann zu **Lifelocks** führen

- → Solche Probleme auf Entwurfsebene vermeiden

beliebige Deadlines

- **bisher:** Deadline ≤ Periode
- **jetzt:** Deadline > Periode
  - Ereignis kann erneut eintreten bevor es fertig behandelt wurde
  - mehrere Inkarnationen einer Subtask können existieren
  - TASKs **müssen mehrfach aktivierbar** sein
  - ISRs **können nicht mehrfach** aktiviert werden

- Analyse am kritischen Zeitpunkt reicht nicht mehr aus
  - Analyse muss sich über ein **Auslastungsintervall** erstrecken

Ein **Priorität-A-Auslastungsintervall** [a,b] ist ein Zeitintervall in dem nur Subtasks mit einer Priorität ≥ A ausgeführt werden. In den Zeiträumen]a – ε,a[und]b,b + ε[werden nur Subtasks mit einer Priorität < A ausgeführt.

Erweiterte Ablaufplanung

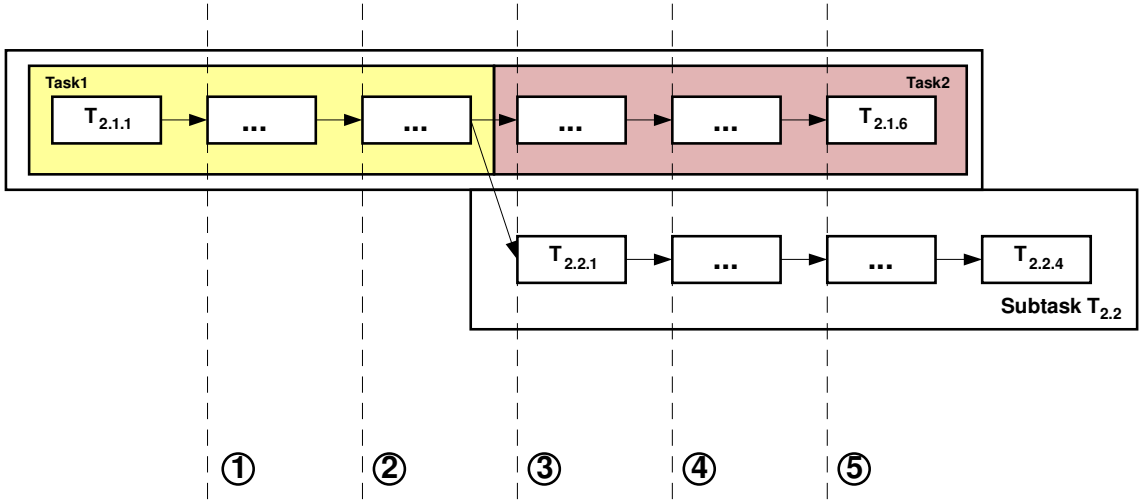
- Laufzeitprioritäten
- Verzweigungen
- abhängige Ereignisse

Laufzeitprioritäten

- bisherige Verwendung von ABBs
  - Berechnung der WCET
  - Berechnung der Belegungszeit von RESOURCEn
  - Sperrung von Unterbrechungen

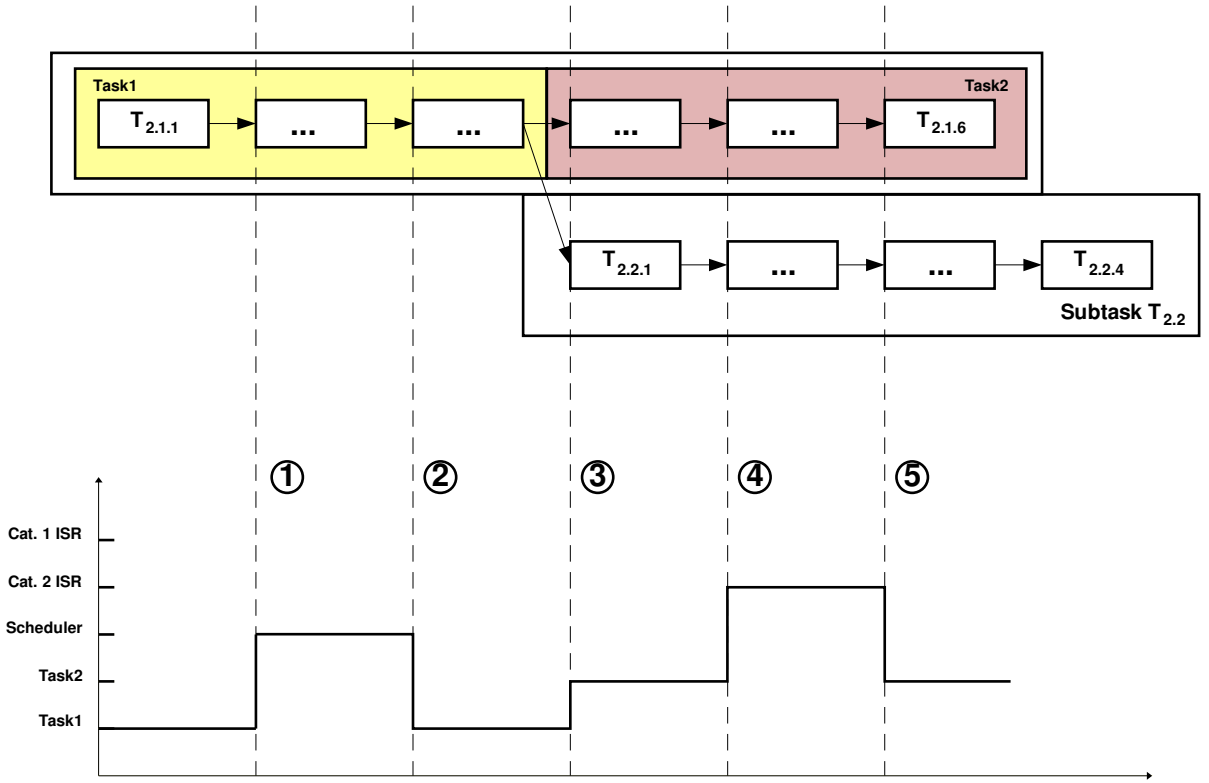
- **jetzt:** Priorität kann zwischen ABBs *variieren*

Laufzeitprioritäten




```
1 GetResource(RES_SCHEDULER)
2 ReleaseResource(RES_SCHEDULER)
3 ChainTask(Task2)
4 SuspendOSInterrupts()
5 ResumeOSInterrupts()
```

Laufzeitprioritäten



1	`GetResource(RES_SCHEDULER)`
2	`ReleaseResource(RES_SCHEDULER)`
3	`ChainTask(Task2)`
4	`SuspendOSInterrupts()`
5	`ResumeOSInterrupts()`

Laufzeitprioritäten: Antwortzeitanalyse

- Begriffe
  - **Unterbrechungsblock** eines ABB

    Ein **Unterbrechungsblock** eines ABB ist ein längst möglicher **Abschnitt eines Ausführungspfades** dessen ABBs alle eine **größere Priorität** haben als dieser ABB.

  - **führender Unterbrechungsblock**

    Ein **führender Unterbrechungsblock** ist ein Unterbrechungsblock dessen erster ABB zugleich der **erste ABB eines Ausführungspfades** ist.

  - **Verzögerungsblock** eines ABB

    Ein **Verzögerungungsblock** eines ABB ist ein längst möglicher **Abschnitt eines Ausführungspfades** dessen ABBs alle eine **größere/gleiche Priorität** als/wie dieser ABB haben.

Laufzeitprioritäten: Antwortzeitanalyse

- Klassen von Subtasks:
  - $L_{n.m.l}$

    diese Subtasks enthalten keinen Unterbrechungsblock des ABB $T_{n.m.l}$.
  - $U_{n.m.l}$

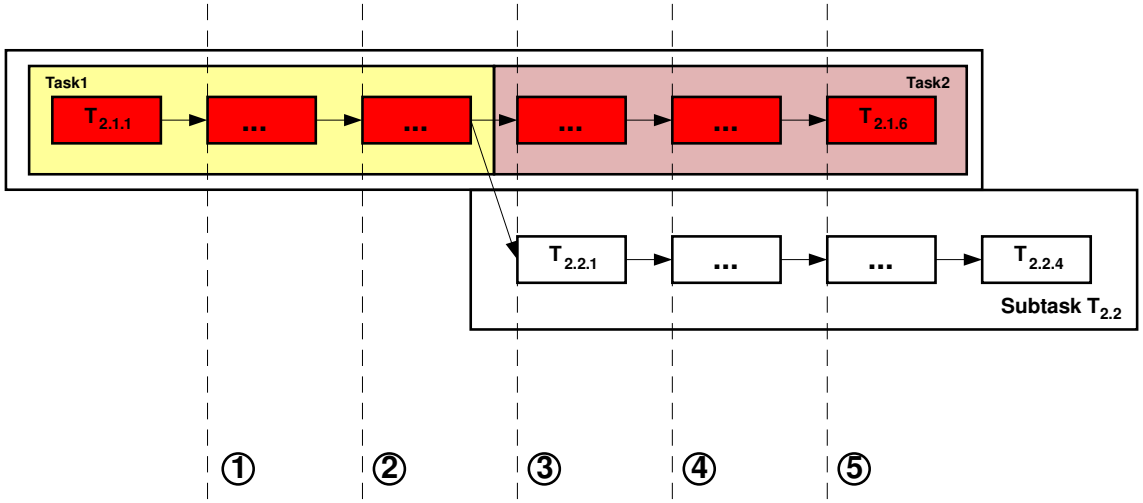
    diese Subtasks enthalten einen führenden Unterbrechungsblock des ABB $T_{n.m.l}$.
  - $V/L_{n.m.l}$

    diese Subtasks enthalten einen mindestens einen Verzögerungsblock des ABB $T_{n.m.l}$.

- **Zweck**
  - führender Unterbrechungsblock: **mehrmalige Unterbrechung**
  - Unterbrechungs-/Verzögerungsblock: **einmalige Verzögerung**

Laufzeitprioritäten: Antwortzeitanalyse

- Analyse erfolgt für einen bestimmten Ausführungspfad



- alle Ausführungspfade müssen analysiert werden
- Analyse erfolgt sukzessive
  - Berechnung der Antwortzeit des ersten ABB
  - Berechnung der Antwortzeit des zweiten ABB ausgehend von diesem Ergebnis

Laufzeitprioritäten: Antwortzeitanalyse

- Antwortzeit des ABB $T_{n.m.1}$

$$w^{k}_{n.m.1}(t^{(l)}) = b_{n.m} + e_{n.m.1} + (k-1)\,e_{n.m} + \sum_{T_{i,j} \in U_{n.m.1}} ceiling\left(\frac{t^{(l)}}{p_i}\right) e^{u}_{i.j\,,\,n.m.1} + \sum_{T_{i,j} \in V / L_{n.m.1}} e^{u}_{i.j\,,\,n.m.1}$$

Laufzeitprioritäten: Antwortzeitanalyse

- Antwortzeit des ABB $T_{n.m.1}$

Blockade

wiederholte Verzögerung durch führende Unterbrechungsblöcke

$$w^{k}_{n.m.1}(t^{(l)}) = b_{n.m} + e_{n.m.1} + (k-1)\,e_{n.m} + \sum_{T_{i.j} \in U_{n.m.1}} ceiling\left(\frac{t^{(l)}}{p_i}\right) e^{u}_{i.j\,,\,n.m.1} + \sum_{T_{i.j} \in V / L_{n.m.1}} e^{u}_{i.j\,,\,n.m.1}$$

WCET der Unterbrechungs – bzw. Verzögerungsblöcke

Zeitbedarf von $T_{n.m.1}$ und der (k -1) vorherigen Auftreten von T_n

einmalige Verzögerungen durch Verzögerungsblöcke

Laufzeitprioritäten: Antwortzeitanalyse

- Antwortzeit des ABB $T_{n.m.r}$, r > 1

$$w^k_{n.m.r}(t^{(l)}) = \underbrace{f^k_{n.m.(r-1)} + e_{n.m.r}}_{(1)} + (2) + (3) + (4) + (5)$$

(1) Antwortzeit des vorhergehenden ABB und WCET des gegenwärtigen Subtask

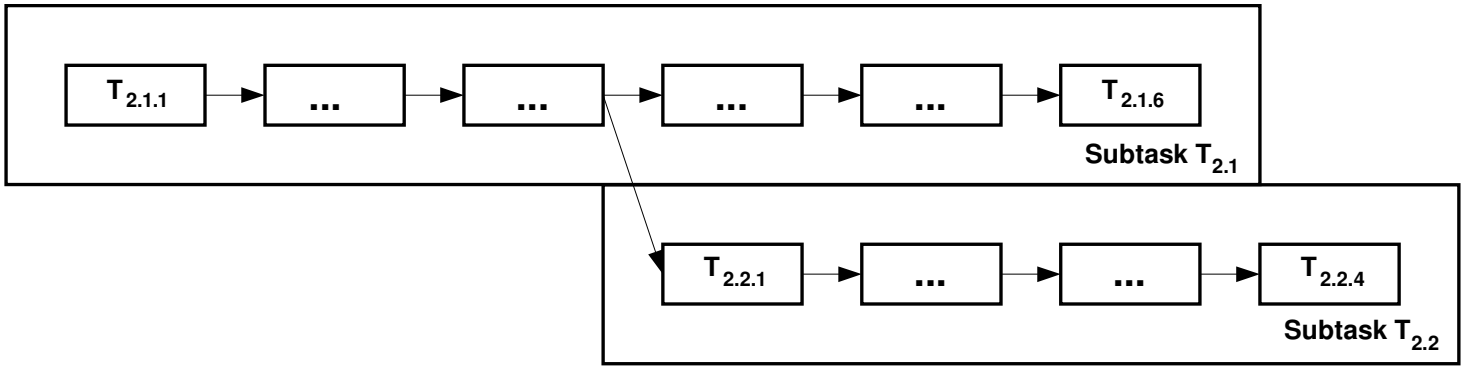
(2) Unterbrechung durch führende Unterbrechungsblöcke – unter bestimmten Umständen (Prio($T_{n.m.r}$) < Prio($T_{n.m.(r-1)}$)) können sich Aktivierungen für Unterbrechungsblöcke aufstauen – diese aufgestauten Aktivierungen werden hier nicht berücksichtigt

(3) Aufgestaute, frühere Aktivierungen führender Unterbrechungsblöcke

(4) Verzögerungsblöcke, die seit Beginn der Subtask aktiv sind und nicht durch einen vorhergehenden Unterbrechungsblock aktiviert wurden.

(5) Verzögerungsblöcke, die durch einen vorhergehenden Unterbrechungsblock aktiviert wurden.

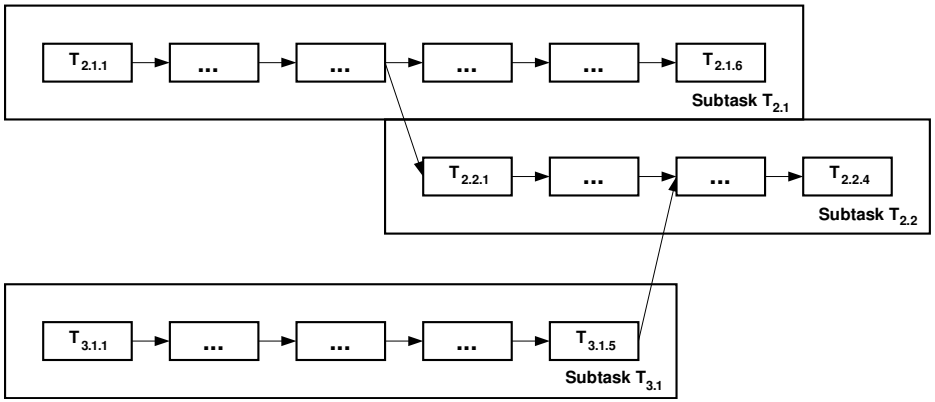
Verzweigungen



- **Prioritäten**
  - Priorität des Vorgängers richtet sich nach den Nachfolgern
  - Prio(Vorgänger) = max Prio(Nachfolger)

- **Antwortzeitanalyse**
  - Vorgänger: erweiterte Antwortzeitanalyse
  - Nachfolger: erweiterte Antwortzeitanalyse für alle Nachfolger ausgehend vom Vorgänger

ODER-Abhängigkeiten



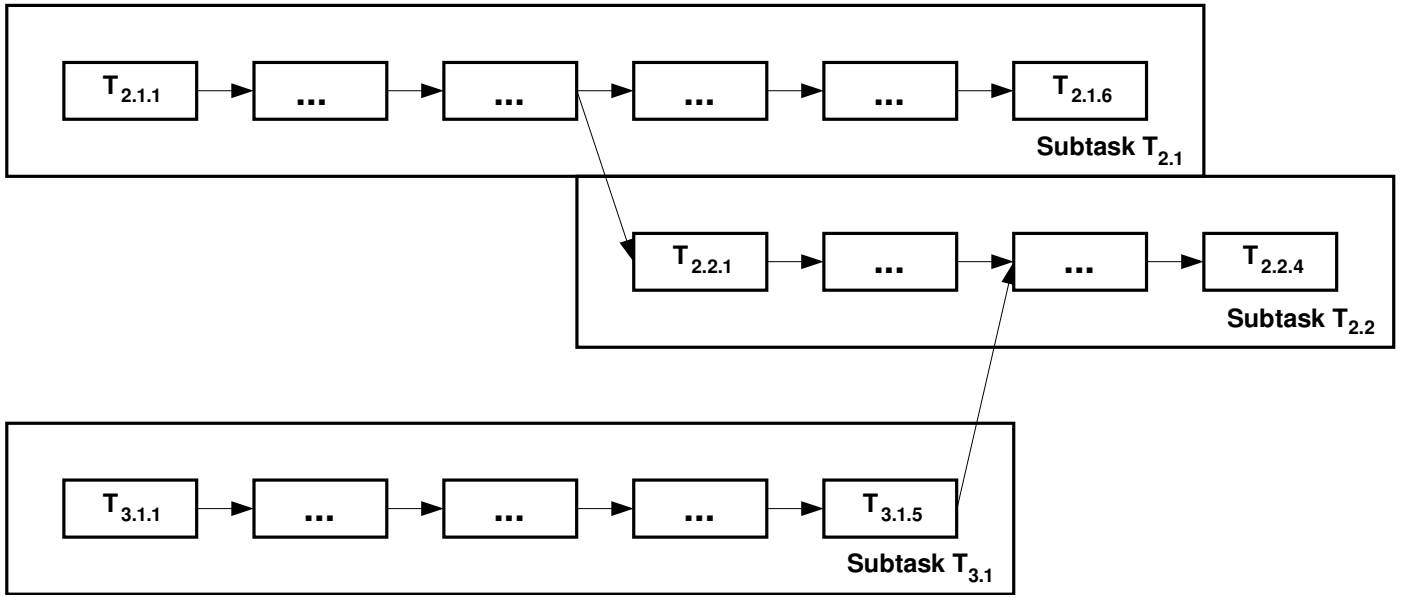
- **Antwortzeitanalyse**
  - für alle Vorgänger: erweiterte Antwortzeitanalyse
  - Nachfolger wird für jeden Vorgänger analysiert
- **Achtung**
  - Aktivierungen des Nachfolgers
  - ein Nachfolger muss mehrere Vorgänger behandeln

ODER-Abhängigkeiten

- **Prioritäten**
  - mehrere Subtasks *teilen* sich *denselben Nachfolger*
  - diese Subtasks können *verschiedene Termine* haben
  - → der Nachfolger müsste *verschiedene Prioritäten* besitzen

- **Lösung**
  - Priorität des Nachfolgers richtet sich nach den Vorgängern
  - → Prio(Nachfolger) = max Prio(Vorgänger)

UND-Abhängigkeiten



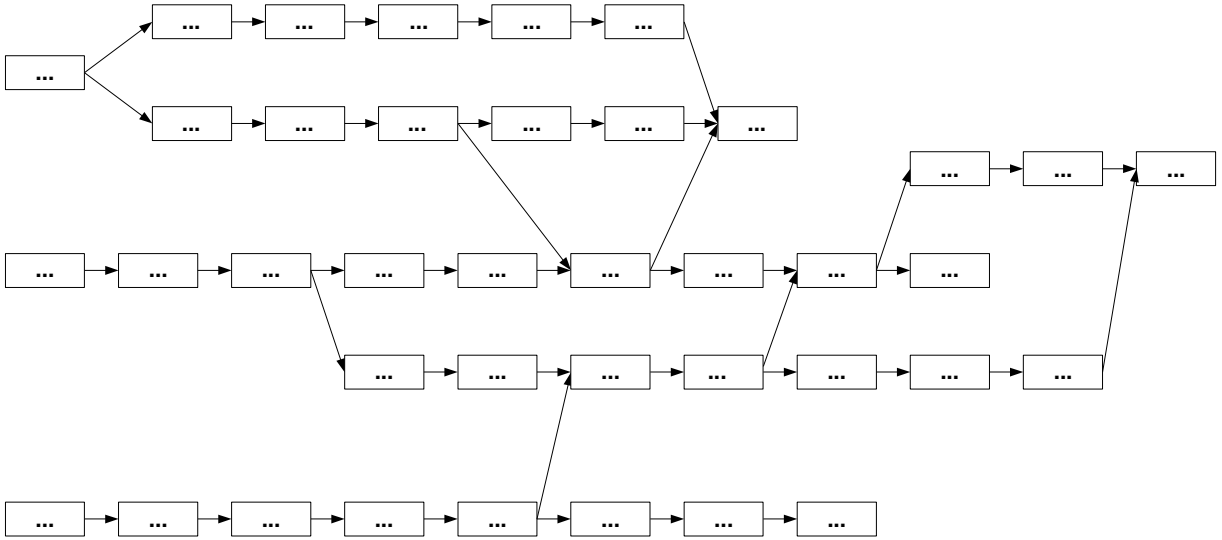
- **Antwortzeitanalyse**
  - für alle Vorgänger: erweiterte Antwortzeitanalyse
  - Nachfolger wird für den spätesten Vorgänger analysiert

UND-Abhängigkeiten

- **Prioritäten**
  - alle Vorgänger haben einen *gemeinsamen Nachfolger*
  - alle Vorgänger haben einen *gemeinsame Termin*
  - *innerhalb* der Vorgänger kann es auch *separate Termine* geben

- **Lösung**
  - Prioritäten der Vorgänger richten sich nach
    - der Priorität des Nachfolgers: Prio(Vorgänger) = Prio(Nachfolger)
    - den separaten Terminen

Probleme - Prioritäten



- Problematik: viele Subtasks, viele Abhängigkeiten
- → hohe Prioritäten
- → evtl. Konflikte

- → nach Möglichkeit im Entwurf vermeiden

Zusammenfassung

- ProOSEK Abbildung
  - Verzweigungen
  - ODER-Abhängigkeiten
  - UND-Abhängigkeiten
  - aperiodische Ereignisse
- ProOSEK: einfache Ablaufplanung
  - Algorithmus
  - trivialer Fall
  - TASKs
  - kritische Abschnitte
  - ALARMe
  - Self-Suspension
  - beliebige Deadlines
- ProOSEK: erweiterte Ablaufplanung
  - Prinzip
  - Verzweigungen
  - ODER/UND-Abhängigkeiten

Ergebnis

- voll ausformulierte Steuerung
  - eCos Konfiguration/Initialisierung
  - Implementierung der Anwendung
  - Abbildung der Komponenten auf Tasks und Subtasks
  - Abbildung und Implementierung der Abhängigkeiten
- Antwortzeitanalyse