

# Phase 4: Komposition – ProOSEK

## Echtzeitsysteme 2 - Vorlesung/Übung

Peter Ulbrich  
Fabian Scheler  
Wolfgang Schröder-Preikschat

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

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



1

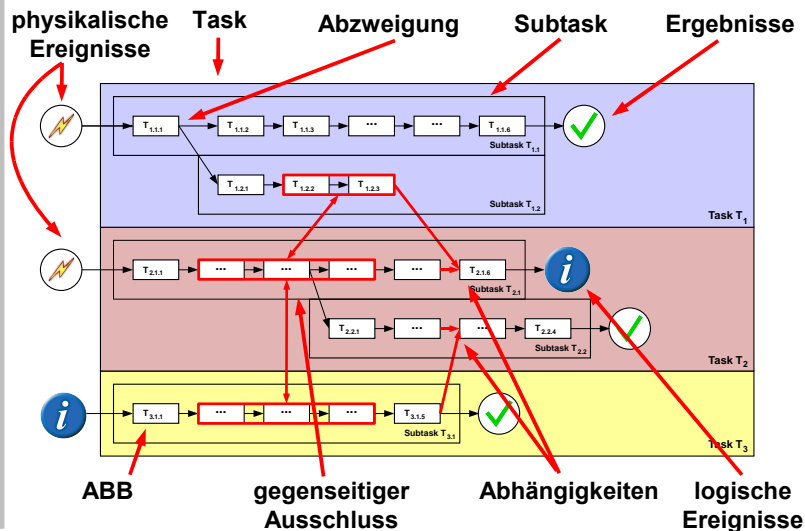
## Übersicht

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

2

© {ulbrich, scheler, wosch}@cs.fau.de - EZL (SS 2010)

## Wiederholung: Ereignisbehandlungen



3

## Wiederholung: Abbildung

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

4

© {ulbrich, scheler, wosch}@cs.fau.de - EZL (SS 2010)

## Wiederholung: Antwortzeitanalyse

### Auslösezeitpunkt

- Zeitpunkt an dem ein Ereignis eintritt

### Antwortzeit

- Abschlusszeitpunkt der Behandlung des Ereignisses
- mit anderen Worten: das Ende eines Subtask

### maximale Antwortzeit:

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

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

### hinreichendes und notwendiges Kriterium für Planbarkeit:

$$\forall \text{Tasks } t_i: \exists t: w_i(t) \leq t \wedge t \leq d_i$$

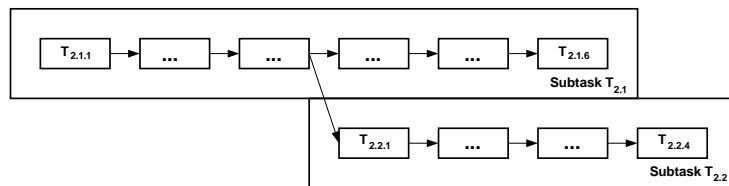


## ProOSEK: Abbildung

- Verzweigungen
- Abhängigkeiten
- gegenseitiger Ausschluss
- aperiodische Ereignisbehandlungen



## ProOSEK: Verzweigungen



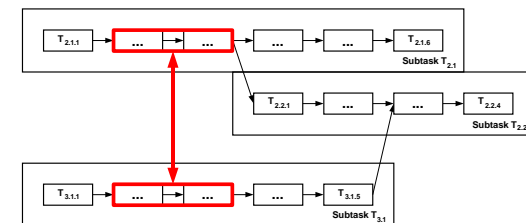
- Task spaltet sich in unabhängige Kontrollflüsse
- Subtasks *münden nicht wieder zurück*

```
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  $T_{31}$ , ALARM A2 Subtask  $T_{21}$

- phase2 = WCET(Subtask  $T_{31}$ ) → **keine Überlappung!**

- beide Subtasks haben eine identische Periode period

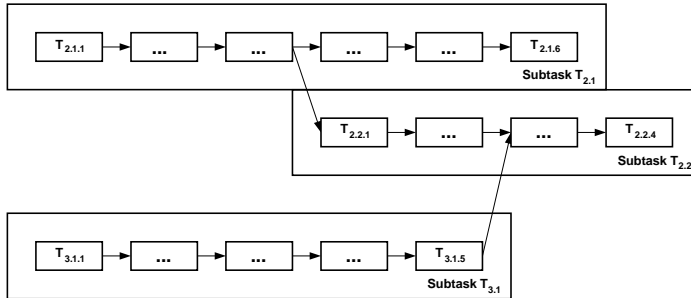
- **keine gesonderte Synchronisation notwendig!**

→ **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 → PxROS-HR

### ■ Nachrichten & Mailboxen

- Favorisiertes Konzept in PxROS-HR
- `PxMbxCreate()` und `PxMsgSend()`, `PxMsgReceive()`
- **Achtung:** Verschiedene Varianten (blockierend / nicht-blockierend), Envelop Mechanismus, usw.
- kombiniert ggf. Kontroll- und Datenflussabhängigkeit

### ■ Verzweigungen

- einen Faden aktivieren
  - `PxTaskCreate()` / `PxTaskResume(taskid)`
  - `PxDelayCreate(obj)` / `PxDelaySched()`
- **Achtung:** Tasks sind in PxROS-HR **nicht mehrfach aktivierbar**
  - der Task muss **alle eingetretenen Vorgänger** behandeln
  - z.B. bis der Semaphor den Wert 0 erreicht hat



- gegenseitiger Ausschluss
  - **Semaphore**
    - `PxSemaRequest()` und `PxSemaRelease()`
  - **Mutex, Kernelized Monitor & Unterbrechungen sperren**
    - So nicht vorgesehen
    - „Missbrauchen“ von `PxTaskSetPrio()`
    - Abbilden auf Nachrichten `PxMsgAwaitRel()`
    - Abbilden auf Remote Handler (Epilog) `PxTaskSchedRemhnd()`
- ODER-Abhängigkeiten
  - **Semaphore** → siehe „gegenseitiger Ausschluss“
  - **Nachrichten** → siehe „Mailboxen & Nachrichten“
  - **Events**
    - `PxAwaitEvents()`, `PxTaskSignalEvents()`, `PxResetEvents()`
    - einzelne Events lassen sich mit ODER bzw. UND verknüpfen



- UND-Abhängigkeiten
  - **Semaphore** → siehe „gegenseitiger Ausschluss“
  - **Events** → siehe „ODER-Abhängigkeiten“
  - **Nachrichten** → siehe „Mailboxen & Nachrichten“
- gegenseitiger Ausschluss, UND-/ODER-Abhängigkeiten
  - **Phasenverschiebung**
    - Clocks und Timeouts `PxPeRequest()`, `PxToRequest()`
    - **Achtung:** Kann relativ und absolut verwendet werden!



## 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  $T_{nm}$  :

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

$Pa_{n,m,l}$  Pfad l der Subtask  $T_{nm}$

$k_{n,m,i}(T_{n,m,i})$  Auftreten des ABB  $T_{nmi}$  im Pfad  $Pa_{nmi}$



## 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  $\leq$  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  $T_{nm}$  durch höher-priore Subtasks unterbrochen wird

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

### hier:

$$i_{n,m} = \sum_{T_{ij} \in H_{n,m}} \text{ceiling}(r_{n,m} / p_i) e_{ij}$$

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

$\text{ceiling}(r_{n,m} / p_i)$  Auftreten des Ereignisses  $T_i$  während der Ausführung der Subtasks  $T_{nm}$

$b_{n,m} = 0$  Keine Blockierung durch nieder-priore ISRs

### ingesamt:

$$r_{n,m} = e_{n,m} + \sum_{T_{ij} \in H_{n,m}} \text{ceiling}(r_{n,m} / p_i) e_{ij} + b_{n,m}$$



## 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:

$$i_{n,m} = \underbrace{\sum_{T_{ij} \in H_{n,m}} \text{ceiling}(r_{n,m} / p_i) e_{ij}}_{(1)} + \underbrace{\sum_{T_{ij} \in L_{n,m}} \text{ceiling}(r_{n,m} / p_i) e_{ij}^{isr}}_{(2)}$$

(1) Verdrängung durch höher-priore Subtasks

(2) Verdrängung durch ISRs nieder-priorer Subtasks

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

$e_{ij}^{isr}$  Ausführungszeit des ISR-Anteils der Subtask  $T_{nm}$



## Gegenseitiger Ausschluss

### Implementierungsvarianten

- gesperrte Unterbrechungen
- nicht-präemptive TASKs
- RESOURCEn

### hier:

$$b_{n,m} = b_{n,m}^{non} + b_{n,m}^{isr} + b_{n,m}^{res}$$

$b_{n,m}^{non}$  Blockade durch nicht-präemptive TASKs

$b_{n,m}^{isr}$  Blockade durch gesperrte Unterbrechungen

$b_{n,m}^{res}$  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:  $w_{n,m}(t) = e_{n,m} + tc_{n,m}^{isr}(t) + i_{n,m} + b_{n,m}$   
 $tc_{n,m}^{isr}(t)$  Unterbrechungen der Subtask  $T_{nm}$  durch Timer-ISRs
- hier:  $tc_{n,m}^{isr}(t) = \sum_{tc \in TC} tc_{n,m}^{tc}(t)$   
 $tc_{n,m}^{tc}(t) = 0$  Subtask  $T_{nm}$  enthält keine TASKs und alle ISRs haben eine höhere Priorität als der ISR des Hardware-Timers  
 $tc_{n,m}^{tc}(t) = \text{ceiling}(t / p_{tc}) e_{tc}^{isr}$  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:  $w_{n,m}(t) = e_{n,m} + tc_{n,m}^{isr}(t) + i_{n,m} + s_{n,m} + (s_{n,m}^k + 1)b_{n,m}$   
 $s_{n,m}$  so lange verzichtet Subtask  $T_{nm}$  auf den Prozessor  
 $s_{n,m}^k$  so oft verzichtet Subtask  $T_{nm}$  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-priorie Subtasks werden lange verzögert
- kann zu **Lifelocks** führen

→ Solche Probleme auf Entwurfsebene vermeiden



## beliebige Deadlines

- **bisher:** Deadline  $\leq$  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  $\geq A$  ausgeführt werden. In den Zeiträumen  $]a - \epsilon, a[$  und  $]b, b + \epsilon[$  werden nur Subtasks mit einer Priorität  $< A$  ausgeführt.



## Erweiterte Ablaufplanung

- Laufzeitprioritäten
- Verzweigungen
- abhängige Ereignisse



## Laufzeitprioritäten

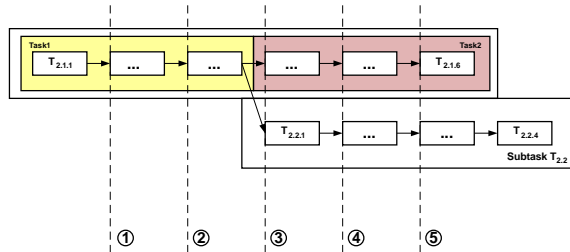
- bisherige Verwendung von ABBs

- Berechnung der WCET
- Berechnung der Belegungszeit von RESOURCEn
- Sperrung von Unterbrechungen

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

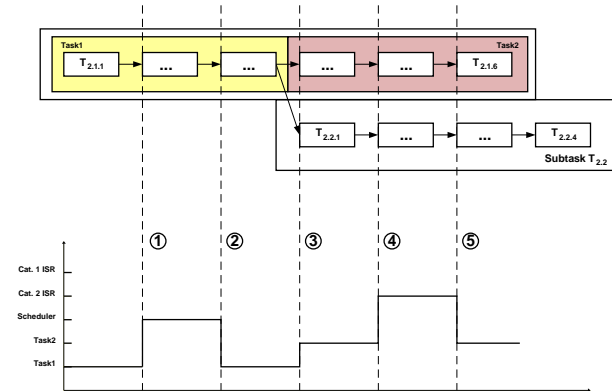


# Laufzeitprioritäten



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 **Verzögerungsblock** 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:

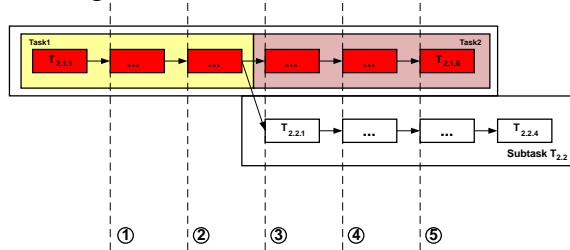
- $L_{n,m}$   
diese Subtasks enthalten keinen Unterbrechungsblock des ABB  $T_{nrd}$ .
- $U_{n,m}$   
diese Subtasks enthalten einen führenden Unterbrechungsblock des ABB  $T_{nrd}$ .
- $V/L_{n,m}$   
diese Subtasks enthalten einen mindestens einen Verzögerungsblock des ABB  $T_{nrd}$ .

### 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  $T_{nm1}$

$$w_{n,m,l}^k(t^{(l)}) = b_{n,m} + e_{n,m,l} + (k-1)e_{n,m} + \sum_{T_{ij} \in U_{n,m,l}} \text{ceiling}\left(\frac{t^{(l)}}{p_i}\right) e_{ij,n,m,l}^u + \sum_{T_{ij} \in V/L_{n,m,l}} e_{ij,n,m,l}^u$$



## Laufzeitprioritäten: Antwortzeitanalyse

- Antwortzeit des ABB  $T_{nm1}$

Blockade

wiederholte Verzögerung durch führende Unterbrechungsblöcke

$$w_{n,m,l}^k(t^{(l)}) = b_{n,m} + e_{n,m,l} + (k-1)e_{n,m} + \sum_{T_{ij} \in U_{n,m,l}} \text{ceiling}\left(\frac{t^{(l)}}{p_i}\right) e_{ij,n,m,l}^u + \sum_{T_{ij} \in V/L_{n,m,l}} e_{ij,n,m,l}^u$$

WCET der Unterbrechungs – bzw. Verzögerungsblöcke

Zeitbedarf von  $T_{nm1}$  und der (k-1) vorherigen Auftreten von  $T_n$

einmalige Verzögerungen durch Verzögerungsblöcke



## Laufzeitprioritäten: Antwortzeitanalyse

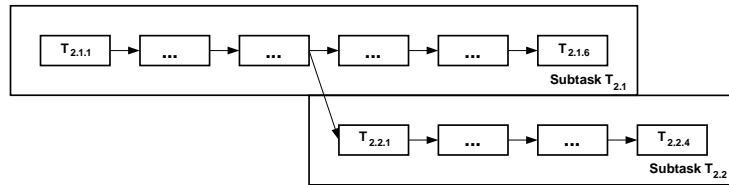
- Antwortzeit des ABB  $T_{nmr}$ ,  $r > 1$

$$w_{n,m,r}^k(t^{(l)}) = \underbrace{f_{n,m,(r-1)}^k + e_{n,m,r}}_{(1)} + (2) + (3) + (4) + (5)$$

- (1) Antwortzeit des vorhergehenden ABB und WCET des gegenwärtigen Subtask
- (2) Unterbrechung durch führende Unterbrechungsblöcke – unter bestimmten Umständen ( $\text{Prio}(T_{nmr}) < \text{Prio}(T_{nm(r-1)})$ ) 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

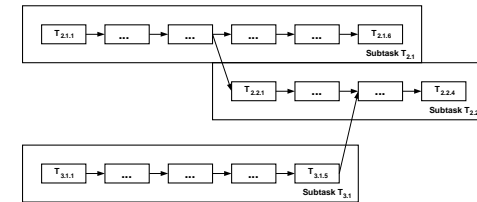
- Priorität des Vorgängers richtet sich nach den Nachfolgern
- $\text{Prio}(\text{Vorgänger}) = \max \text{Prio}(\text{Nachfolger})$

### ■ 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

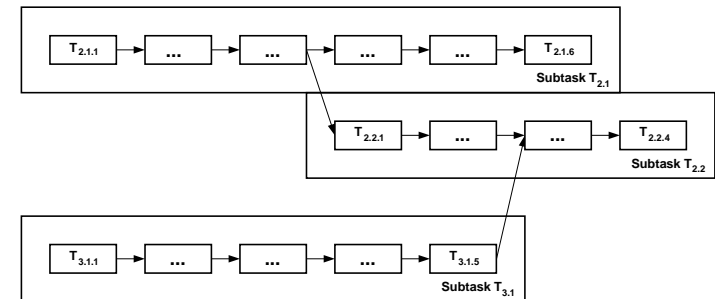
- mehrere Subtasks *teilen* sich *denselben* Nachfolger
- diese Subtasks können *verschiedene Termine* haben
- der Nachfolger müsste *verschiedene Prioritäten* besitzen

### ■ Lösung

- Priorität des Nachfolgers richtet sich nach den Vorgängern
- $\text{Prio}(\text{Nachfolger}) = \max \text{Prio}(\text{Vorgänger})$



## UND-Abhängigkeiten



### ■ Antwortzeitanalyse

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



## UND-Abhängigkeiten

### ■ Prioritäten

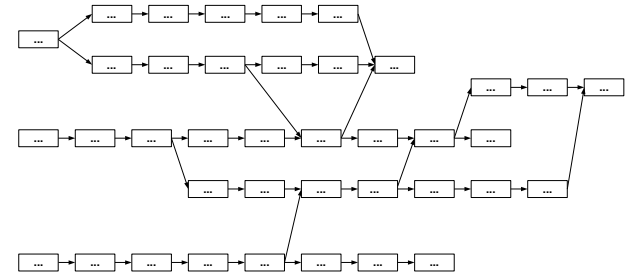
- alle Vorgänger haben einen *gemeinsamen Nachfolger*
- alle Vorgänger haben einen *gemeinsame Termin*
- *innerhalb* der Vorgänger kann es auch *separate Termine* geben

### ■ Lösung

- Prioritäten der Vorgänger richten sich nach
  - der Priorität des Nachfolgers:  $\text{Prio}(\text{Vorgänger}) = \text{Prio}(\text{Nachfolger})$
  - 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

