

Echtzeitsysteme

Rangfolge

Lehrstuhl Informatik 4

12. Dezember 2013

Gliederung

- 1 Überblick
- 2 Rangfolge und gerichtete Abhängigkeiten
 - Datenabhängigkeiten
 - Nebenläufigkeit
 - Abhängigkeits- und Aufgabengraphen
 - Koordinierung
- 3 Umsetzung
 - Naive Implementierung
 - Physikalisch und logische Ereignisse
 - Implementierungsvarianten gerichteter Abhängigkeiten
- 4 Ablaufplanung
- 5 Zusammenfassung

Fragestellungen

- Was bedeutet **Rangfolge**?
 - Was ist die Ursache von Rangfolge?
 - Wie beschreibt man Rangfolge?
- Wie kann man **Rangfolge implementieren**?
 - Welche Implementierungsvarianten gibt es?
 - Welche Implikationen haben sie?
- Wie geht man in der **Ablaufplanung** mit Rangfolgebeziehungen um?

Gliederung

1 Überblick

2 Rangfolge und gerichtete Abhangigkeiten

- Datenabhangigkeiten
- Nebenlaufigkeit
- Abhangigkeits- und Aufgabengraphen
- Koordinierung

3 Umsetzung

- Naive Implementierung
- Physikalisch und logische Ereignisse
- Implementierungsvarianten gerichteter Abhangigkeiten

4 Ablaufplanung

5 Zusammenfassung

Rangfolge (engl. *precedence*)

Abhängigkeit von Kontrollflüssen

Arbeitsaufträge können gezwungen sein, in einer ganz bestimmten Reihenfolge ausgeführt werden zu müssen

- Beispiel Radarüberwachungsanlage ...
 - Signalaufbereitungsauftrag muss vor Nachführauftrag gelaufen sein
- Beispiel Kommunikationssystem ...
 - Sendeauftrag muss vor Empfangsauftrag gelaufen sein
 - Empfangsauftrag muss vor Bestätigungsauftrag gelaufen sein
- Beispiel Anfragesystem ...
 - Authentifizierungsauftrag muss vor Eingabeauftrag gelaufen sein
 - Eingabeauftrag muss vor Suchauftrag gelaufen sein
 - Suchauftrag muss vor Ausgabeauftrag gelaufen sein

 die Rangfolge ist oft in Datenabhängigkeiten begründet

Datenabhangigkeit (engl. *data dependency*)

Abhangigkeit von konsumierbaren Betriebsmitteln

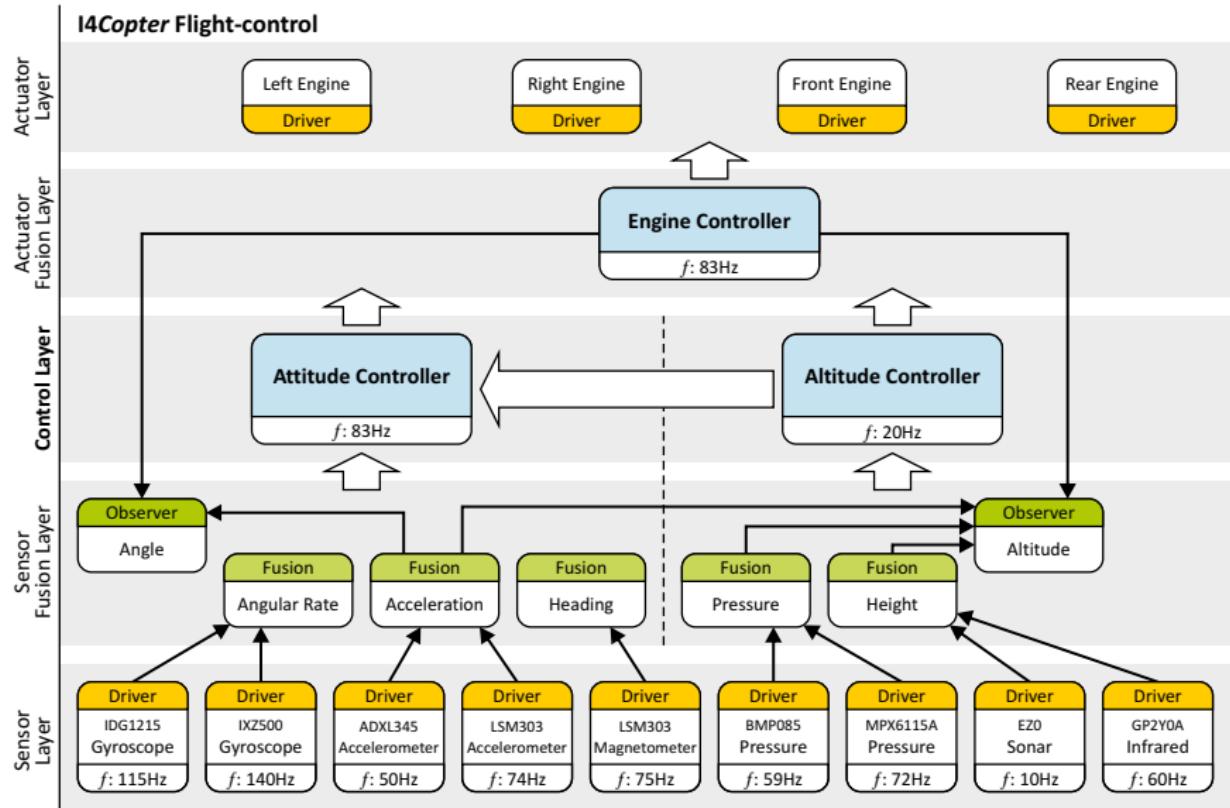
Arbeitsauftrage brauchen zum Ablauf ggf. **konsumierbare Betriebsmittel**

- ihre Anzahl ist (log.) unbegrenzt: Nachrichten, Signale, Interrupts
Produzent kann beliebig viele davon erzeugen
Konsument zerstort sie wieder bei Inanspruchnahme

Produzent und Konsument sind voneinander abhangige Entitaten

- zwischen ihnen besteht eine **gerichtete Abhangigkeit**
- der Konsument vom Produzenten ...
 - weil ein konsumierbares Betriebsmittel erst bereitgestellt werden muss, um es in Anspruch nehmen zu konnen
- der Produzent vom Konsumenten (zumindest manchmal) ...
 - weil konsumierbare Betriebsmittel auf endlich viele wiederverwendbare Betriebsmittel abgebildet werden
 - weil der Produzent dazu erst ein wiederverwendbares Betriebsmittel anfordern muss, das vom Konsumenten spater wieder freizugeben ist
 - Beispiel: **begrenzter Puffer** (engl. *bounded buffer*)

Datenabhängigkeiten im I4Copter



Nebenlufige Aktivitaten

Nichtsequentielles Programm

Nebenlufigkeit (engl. *concurrency*) bezeichnet das Verhaltnis von nicht kausal abhangigen Ereignissen, die sich also nicht beeinflussen

- Aktionen konnen nebenlufig ausgefuhrt werden, wenn keine das Resultat des anderen benigt

```
1:     foo = 4711;  
2:     bar = 42;  
3: foobar = foo + bar;  
4: barfoo = bar + foo;  
5:     hal = foobar + barfoo;
```

- Zeile 1 kann nebenlufig zu Zeile 2 ausgefuhrt werden
- Zeile 3 kann nebenlufig zu Zeile 4 ausgefuhrt werden

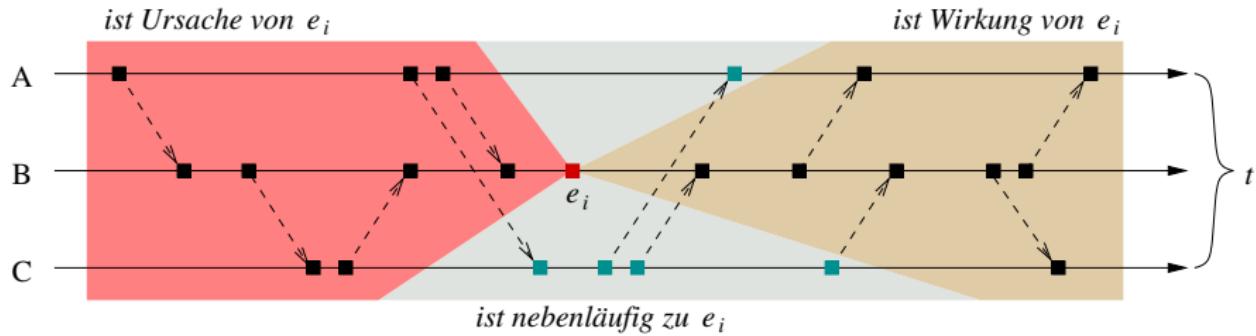
Kausalitat (lat. *causa*: Ursache) ist die Beziehung zwischen **Ursache** und **Wirkung**, d.h., die ursachliche Verbindung zweier Ereignisse

- Ereignisse sind nebenlufig, wenn keines Ursache des anderen ist

Kausalordnung

Nebenlufigkeit als relativistischer Begriff von Gleichzeitigkeit

Relationen „ist Ursache von“, „ist Wirkung von“, „ist nebenlufig zu“:



- ein Ereignis ist nebenlufig zu einem anderen, wenn es im **Anderswo** des anderen Ereignisses liegt
 - d.h., weder in der Zukunft noch in der Vergangenheit des anderen
- das Ereignis ist weder Ursache oder Wirkung des anderen Ereignisses

Kausalordnung (Forts.)

Rangfolge aus Grunden von Daten- und Zeitabhangigkeit

„ist Ursache von“
„ist Wirkung von“
„ist nebenlufig zu“ } ~ Sequentialisierung von Ereignissen/Aktionen

Aktionen konnen im Echtzeitbetrieb nebenlufig stattfinden, wenn . . .

- keine das Resultat der anderen benigt (s. Folie VI/8) ✓
- keine die (strikten) Zeitbedingungen der anderen verletzt
 - Zeitpunkte drfen nicht bzw. nur selten verpasst werden
 - Zeitintervalle drfen nicht bzw. nur begrenzt zeitlich gedehnt werden
 - Abstand zwischen Ursache (Ereigniszeitpunkt) und Wirkung (Termin)

. . . Abhangigkeiten hingegen erfordern das Herstellen von Gleichzeitigkeit

- z.B. durch den Austausch von Zeitsignalen (s. Folie VI/15)
 - implizit im Falle analytischer Koordinierung
 - explizit im Falle konstruktiver Koordinierung

Beispiel: Serieller Empfang von Nachrichten

Implementierung orientiert sich an OSEK OS [7] bzw. AUTOSAR OS [2]

Die Nachrichtenverarbeitung besteht aus zwei getrennten Aufgaben:

Empfang Abholen einzelner Bytes und Zusammensetzen von Nachrichten
Verarbeitung Nachricht verarbeiten und Behandlung aktivieren

Empfang

```
Pool *msgPool; Buffer *msgBuffer; Message *msg;

ISR(SerialByte) {
    unsigned char rec = rs232_get();
    msg_addTo(msg, rec);

    if(msg_isComplete(msg)) {
        buffer_ins(msgBuffer, msg);
        msg = pool_getfree(msgPool);
    }

    return;
}
```

Verarbeitung

```
TASK(MsgHandler) {
    Message *cMsg = 0;

    InitHandler();

    cMsg = buffer_get(msgBuffer);
    msg_prepare(cMsg);
    handle(cMsg);

    TerminateTask();
}
```

Datenabhängigkeit \leadsto gemeinsamer Puffer `msgBuffer`

Rangfolge \leadsto Wann kann die Nachricht verarbeitet werden?

???

- Wann wird `TASK(MsgHandler)` aktiv?

Abhangigkeits- und Aufgabengraphen [6, S. 43]

Notationen fur Abhangigkeiten zwischen verschiedenen Arbeitsauftragen

Die Kausalordnung ist eine Halbordnung und wird durch eine **Vorgangerrelation** (engl. *precedence relation*) → beschrieben:

- $J_i \rightarrow J_k$: Job J_i ist **Vorganger** (engl. *predecessor*) von J_k
- die Ausführung des **Nachfolgers** (engl. *successor*) J_k erfordert die Fertigstellung des Vorgängers J_i

Beispiel auf Folie VI/11

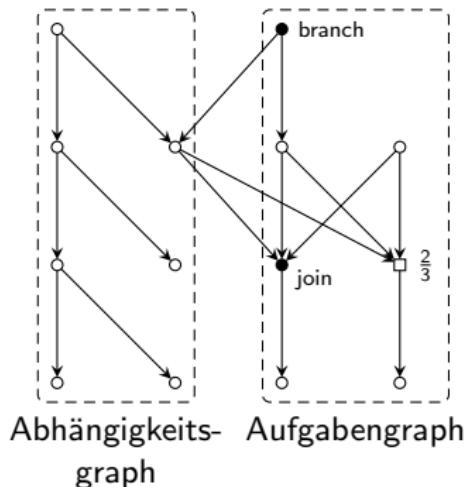
- **ISR(SerialByte)** ist der Vorganger
 - Zuerst muss die Nachricht vollständig empfangen werden, ...
 - **TASK(MsgHandler)** ist der Nachfolger
 - ... anschlieend findet die eigentliche Nachrichtenbehandlung statt.
- ~ fur eine korrekte Funktion ist eine **koordinierte Ausführung** von **ISR(SerialByte)** und **TASK(MsgHandler)** notwendig

Abhangigkeits- und Aufgabengraphen [6, S. 43] (Forts.)

Notationen fur Abhangigkeiten zwischen verschiedenen Arbeitsauftragen

Graph $\mathcal{G} = (\mathcal{J}, \rightarrow)$ dient als Beschreibung der Vorgangerrelation:

- Knoten sind Arbeitsauftrage, Pfeile sind Abhangigkeiten



- **Abhangigkeitsgraph**
 - engl. *precedence graph*
 - beschreibt nur die Vorgangerrelation
- **Aufgabengraph**
 - engl. *task graph*
 - umfasst mehr Abhangigkeitstypen
 - zeitliche Abhangigkeiten
 - UND/ODER-Vorgangerrelationen
 - bedingte Verzweigungen
 - ...

☞ Eingabe fur die statische Ablaufplanung bzw. Implementierung

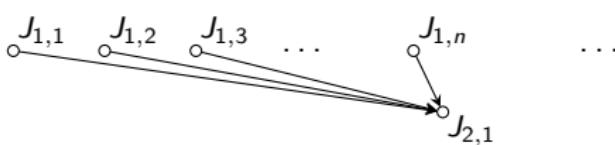
Beispiel: Serieller Empfang von Nachrichten (Forts.)

Abhangigkeitsbeziehungen der einzelnen Arbeitsauftrage

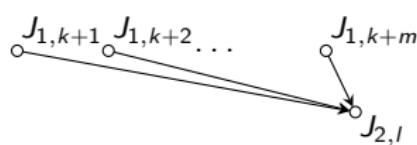
Aufgabe T_1 Empfang einzelner Bytes \leadsto Jobs $J_{1,1}, J_{1,2}, \dots$

Aufgabe T_2 Bearbeitung der Nachrichten \leadsto Jobs $J_{2,1}, J_{2,2}, \dots$

1. Nachricht = n Bytes



1. Nachricht = m Bytes



- **keine Abhangigkeiten** zwischen den einzelnen Jobs von T_1 bzw. T_2
 - auch wenn der Termin $D_{1,1}$ die Fertigstellung von $J_{1,1}$ vor dem Beginn von Job $J_{1,2}$ erzwingt: $D_{1,1} \leq r_{1,2}$
- die Jobs $J_{1,1}, \dots, J_{1,n}$ ermoglichen aber die Ausführung von $J_{2,1}$
 - erst wenn die Nachricht komplett ist, kann sie verarbeitet werden
 - ~ die Jobs $J_{1,1}, \dots, J_{1,n}$ sind Vorganger von $J_{2,1}$
- endgultige Abhangigkeitsbeziehungen erst zur Laufzeit bekannt
 - Nachrichten konnen unterschiedlich viele Bytes umfassen
 - ~ unterschiedlich viele Vorganger von $J_{2,1}$ und $J_{2,I}$

Koordinierung (engl. *coordination*)

Gerichtete Abhangigkeiten analytisch/konstruktiv behandeln

durch Einplanung \leadsto analytische Verfahren

- Ablaufplane berucksichtigen Rangfolgen und Datenabhangigkeiten
 - *a priori Wissen* \mapsto periodische Aufgaben
- Arbeitsauftrage laufen komplett durch (engl. *run to completion*)
 - sie warten weder ex- noch implizit, durfen jedoch verdrangt werden
- Ergebnis ist ein System von ausschlielich einfachen Aufgaben

durch Kooperation \leadsto konstruktive Verfahren

- Synchronisationspunkte in den Programmen explizit machen
 - d.h., *Zeitsignale austauschen* \mapsto Semaphor
- Arbeitsauftrage sind Produzenten/Konsumenten von Ereignissen
 - physikalische Ereignisse von den kontrollierten Objekten
 - logische Ereignisse von anderen Arbeitsauftragen
- Ergebnis ist ein System von (ggf. vielen) komplexen Aufgaben

Gliederung

1 Überblick

2 Rangfolge und gerichtete Abhangigkeiten

- Datenabhangigkeiten
- Nebenlaufigkeit
- Abhangigkeits- und Aufgabengraphen
- Koordinierung

3 Umsetzung

- Naive Implementierung
- Physikalisch und logische Ereignisse
- Implementierungsvarianten gerichteter Abhangigkeiten

4 Ablaufplanung

5 Zusammenfassung

Naive Implementierung

Statische Anordnung im Quelltext: Rangfolge \mapsto Reihenfolge von Unterprogrammen

Gerichtete Abhängigkeiten können statisch im Quelltext kodiert werden:

- falls Vorgänger und Nachfolger \rightarrow priori bekannt und fix sind
- ~ Behandlung wird nur aufgerufen, falls die Nachricht vollständig ist

```
Message *msg;  
  
ISR(SerialByte) {  
    unsigned short received = rs232_getByte();  
    msg_addTo(msg, received);  
  
    if(msg_isComplete(msg)) {  
        InitHandler();  
  
        msg_prepare(currentMsg);  
        handle(currentMsg);  
  
        msg_clear(msg);  
    }  
}
```

Die Implementierung wird so sichtbar vereinfacht:

- nur ein Aktivitätsträger
- Rangfolge ist unmittelbar ablesbar und muss nicht explizit geregelt werden
- keine Pufferung notwendig

☞ Allerdings hat diese Variante auch gravierende Nachteile!

Nachteile implizit kodierter Abhangigkeiten

- die statische Sequentialisierung **verletzt zeitliche Domanen**
 - innerhalb einer zeitlichen Domane ist das zeitliche Verhalten bekannt
 - unterschiedliche zeitliche Domanen besitzen oft auch verschiedene auslosende Ereignisse mit unterschiedlichen zeitlichen Eigenschaften
~ sie sind daher auch Kandidaten fur verschiedene Aufgaben
 - im betrachteten Beispiel existieren folgende zeitliche Domanen:
 - Empfang ~ z.B. nicht-periodische Aufgabe $T_1 = (i_1, e_1)$
 - Verarbeitung ~ z.B. nicht-periodische Aufgabe $T_2 = (i_2, e_2)$
- Beziehung zwischen diesen zeitlichen Domanen:
 - Empfang mehrere Bytes pro Nachricht ~ $i_1 < i_2$
 - Verarbeitung ist komplexer als deren Empfang ~ $e_2 > e_1$
- die naive Implementierung **verschmilzt zeitlichen Domanen**
 - Ergebnis ist eine Aufgabe $T'_1 = (i_1, e_1 + e_2)$
 - das ist **unrealistisch**, schlielich wird T_2 weniger haufig aktiviert

☞ gerichtete Abhangigkeiten: Hinweis auf versch. zeitliche Domanen

Übergang zwischen zeitlichen Domänen

Produzenten und Konsumenten werden mit unterschiedlichen Raten aktiviert

Innerhalb eines Echtzeitssystems können verschiedene zeitliche Domänen existieren (s. Folie VI/7, Beispiel I4Copter).

- in Anlehnung an „clock domains“ eines Hardwarebausteins
 - Bereiche eines Chips, die mit unterschiedlichen Taktraten arbeiten
- ☞ gerichtete Abhängigkeiten erfordern ihre Angleichung
 - die produzierten Daten müssen ...
 - in einem gemeinsamen Puffer zwischengespeichert werden
 - für die weitere Verarbeitung fusioniert und gefiltert werden
 - abhängig von den zeitlichen Eigenschaften dieser Domänen
 - Puffergröße hängt von der Rate von Produzent und Konsument ab
 - der Fusions- bzw. Filteralgorithmus nutzt eine Vorausschau (engl. *lookahead*) des Produzenten im Vergleich zum Konsumenten
 - häufig genügt auch ein einfaches „last is best“
 - ~ Verwende einfach immer den ausreichend aktuellsten Wert!

Übergang zwischen zeitlichen Domänen (Forts.)

Produzenten und Konsumenten werden mit unterschiedlichen Raten aktiviert

- ☞ eine Verschmelzung **zeitlich identischer Domänen** ist möglich
 - stellt aber immer noch eine Optimierung dar
 - ~ letzter Schritt beim Entwurf des Echtzeitsystems [3, 4]
 - zunächst werden die zeitlichen Domänen identifiziert
 - jede zeitliche Domäne wird auf eine Aufgabe abgebildet
 - anschließend werden bestimmte zeitliche Domänen vereinigt
 - sonst bekäme man **zu viele verschiedene Aufgaben**
 - **zeitliche Kohäsion:** Aufgaben werden immer gleichzeitig aktiviert
 - **sequentielle Kohäsion:** Aufgaben laufen immer nacheinander ab
- ☞ die naive Implementierung nimmt diese Optimierung vorweg
 - auch wenn die zeitlichen Domänen verschieden sind
- ☞ nutze **logische Ereignisse**, um zeitliche Domänen zu entkoppeln

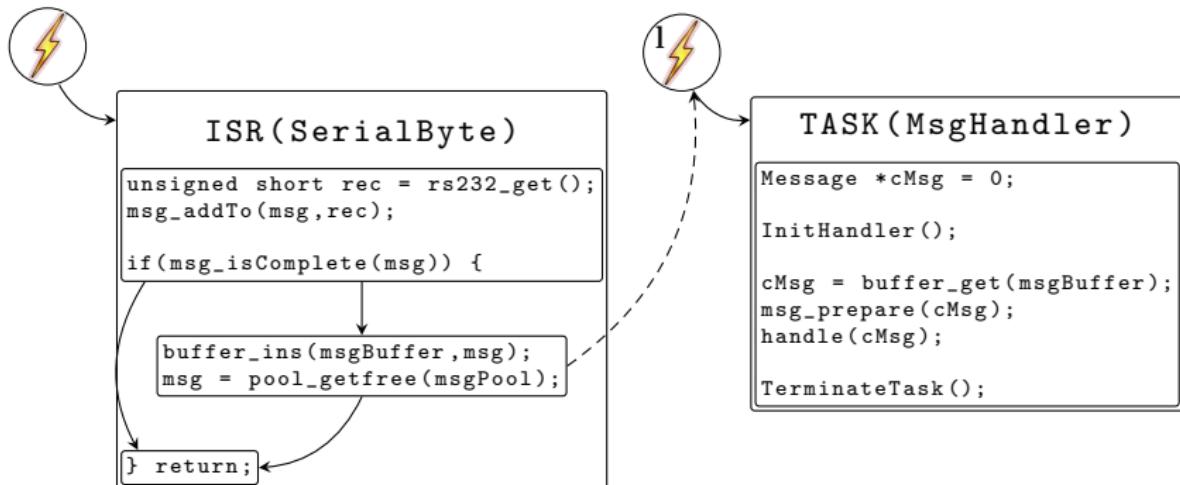
Physikalische und logische Ereignisse

physikalische Ereignisse resultieren aus Zustandsänderungen der Umwelt

- wenn die serielle Schnittstelle den Empfang eines Byte anzeigt
 - ~ infolgedessen wird eine Unterbrechung auslöst

logische Ereignisse ruft die Echtzeitanwendung selbst hervor

- wenn eine Nachricht vollständig empfangen wurde
 - ~ das logische Ereignis entkoppelt Empfang und Verarbeitung zeitlich



Implementierungsvarianten gerichteter Abhängigkeiten

Rangfolge sicherstellen, ohne eine zeitliche Kopplung vorwegzunehmen

Ziel ist die Herstellung der Rangfolge, ohne die zeitliche Nähe durch eine entsprechende Anordnung im Quelltext zu erzwingen:

ohne Koordinierung \leadsto Rangfolge bewusst vernachlässigen

- oft reicht es aus, dass Daten einfach aktuell sind

analytische Koordinierung \leadsto mit Hilfe der Ablaufplanung

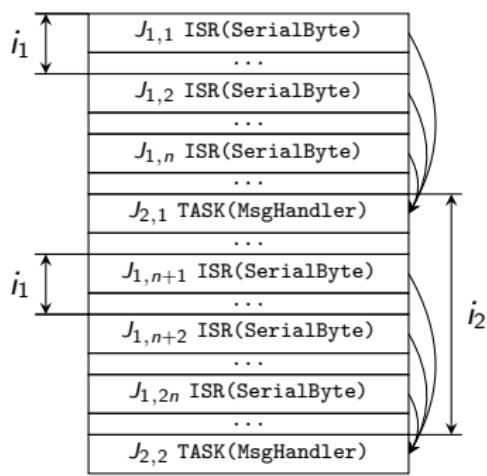
- nur für Abhängigkeiten zwischen **periodischen Aufgaben** anwendbar
 - Taktsteuerung geeignete Anordnung der Jobs in der Ablauftabelle
 - Vorrangsteuerung erreiche Anordnung durch Phasenversatz

konstruktive Koordinierung \leadsto mit Hilfe einseitiger Synchronisation

- für **nicht-periodischen Aufgaben** \leadsto unumgänglich
- in **zeitgesteuerten Systemen** \leadsto unmöglich
- für Synchronisation existieren eine Vielzahl von Möglichkeiten
 - z.B. Aktivierung des Nachfolgers oder expliziter Signalaustausch

Analytische Umsetzung der Rangordnung

Eingabe für die statische Ablaufplanung (s. Folie IV-3/18 ff.) ist ein Abhängigkeits- oder Aufgabengraph (s. Folie VI/13). Die erzeugte Ablauftabelle muss die entsprechenden Randbedingungen einhalten.



- überführe nicht-periodische Aufgaben T_1 und T_2 (s. Folie VI/18) in entsprechende periodische Aufgaben
 - Periode $p_n = \text{Zwischenankunftszeit } i_n$
- ordne Jobs nach den Abhängigkeiten an
 - $r_{i,j} + e_i \leq r_{n,m} \Leftrightarrow J_{i,j} \mapsto J_{n,m}$
- **phasenverschobene Ausführung** von $J_{m,n}$ in vorranggesteuerten System ist analog
 - Rangfolge impliziert passende Phase ϕ_m :

$$\phi_m = \max_{J_{i,j} \mapsto J_{m,n}} r_{i,j} + \omega_{i,j}$$



Einhaltung dieser Phase wird zur Laufzeit nicht überwacht!

- Laufzeitüberschreitungen führen u.U. auch zu Verletzungen der Rangfolge!

Rangfolge durch Bereitstellung des Nachfolgers

AUTOSAR OS [2]

```
ISR(SerialByte) {
    unsigned char rec = rs232_get();
    msg_addTo(msg,rec);

    if(msg_isComplete(msg)) {
        buffer_ins(msgBuffer,msg);
        msg = pool_getfree(msgPool);
        ActivateTask(MsgHandler);
    }
    return;
}

TASK(MsgHandler) { /* ... */ }
```

POSIX [5]

```
void i_serialbyte(void) {
    unsigned char rec = rs232_get();
    msg_addTo(msg,rec);

    if(msg_isComplete(msg)) {
        buffer_ins(msgBuffer,msg);
        msg = pool_getfree(msgPool);
        pthread_create(thread,attr,t_msghandler,NULL);
    }
    return;
}

void t_msghandler(void* arg) { /* ... */ }
```

- Nachfolger wird explizit durch den Vorgänger aktiviert
 - obige Beispiele: Systemaufrufe `ActivateTask` bzw. `pthread_create`
 - der Planer stellt automatisch die richtige Reihenfolge sicher
- **Nachteil:** komplette Sequentialisierung von Vorgänger u. Nachfolger
 - auch wenn dies nicht unbedingt erforderlich wäre
 - erschwert die Umsetzung komplexer Abhängigkeitsszenarien
 - $J_{1,1} \mapsto J_{2,1} \mapsto J_{1,1}$ wäre beispielsweise nicht implementierbar

Rangfolge durch den Austausch von Zeitsignalen

Der Konsument wartet explizit auf das Eintreten der Abhangigkeit

POSIX

```
void i_serialbyte(void) {
    unsigned char rec = rs232_get();
    msg_addTo(msg, rec);

    if(msg_isComplete(msg)) {
        buffer_ins(msgBuffer, msg);
        msg = pool_getfree(msgPool);
        sem_post(&msg_sem);
    }
    return;
}

void t_msghandler(void* arg) {
    Message *cMsg = 0;
    InitHandler();

    do {
        sem_wait(&msg_sem);
        cMsg = buffer_get(msgBuffer);
        msg_prepare(cMsg);
        handle(cMsg);
    } while(1);

    pthread_exit(NULL);
}
```

- Betriebssystemabstraktion:
der **Semaphor** (engl. *semaphore*)
 - **sem_wait()** wartet **blockierend** auf das Eintreten einer Abhangigkeit
 - **sem_post()** zeigt das Eintreten der Abhangigkeit an
- haufig in Verbindung mit sog. **Do-While-Prozessen**
Do \leadsto **InitHandler()**
While \leadsto Nachrichten verarbeiten
- ermoglicht eine **teilweise nebenlaufige Abarbeitung** der beteiligten Jobs
 - Ausführung von **InitHandler()**, bevor eine Nachricht ansteht

Nachrichtenversand (engl. *message passing*)

Kombination aus Rangfolge und Datenaustausch

AUTOSAR OS

```
Message msg, rcvMsg;

ISR(SerialByte) {
    unsigned char rcv = rs232_get();
    msg_addTo(&msg, rec);

    if(msg_isComplete(&msg))
        SendMessage(serialMsg, &msg);
    return;
}

TASK(MsgHandler) {
    Message *cMsg = 0;
    InitHandler();

    do {
        WaitEvent(msgEvent);
        ClearEvent(msgEvent);
        ReceiveMessage(serialMsg, &rcvMsg);
        msg_prepare(&rcvMsg);
        handle(&rcvMsg);
    } while(1);

    TerminateTask();
}
```

- Übermittlung des Zwischenergebnisses durch den Versand einer Nachricht
 - Vorgänger \leadsto SendMessage()
 - Nachfolger \leadsto ReceiveMessage()
- eigenhändige Verwaltung/Pufferung der Daten entfällt unter Umständen
 - \leadsto oft Aufgabe des Kommunikationssystems
- Besonderheit in AUTOSAR OS: keine Rangfolge durch Nachrichtenversand
 - \leadsto ReceiveMessage() blockiert nicht
 - erfordert Kombination mit Signalen
 - Ereignisse (engl. events) in AUTOSAR
 - ein zur Nachricht gehörendes Ereignis, wird bei ihrem Versand gesetzt

Gliederung

1 Überblick

2 Rangfolge und gerichtete Abhängigkeiten

- Datenabhängigkeiten
- Nebenläufigkeit
- Abhängigkeits- und Aufgabengraphen
- Koordinierung

3 Umsetzung

- Naive Implementierung
- Physikalisch und logische Ereignisse
- Implementierungsvarianten gerichteter Abhängigkeiten

4 Ablaufplanung

5 Zusammenfassung

Weitere Lockerung der Restriktionen

Aufhebung der Einschränkungen A2 und A5, A4 bleibt weiter bestehen

Mathematische Ansätze zur Analyse periodischer Echtzeitsysteme schränken solche Systeme häufig stark ein:

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 voneinander, d.h. die einzige gemeinsame Ressource ist die CPU und es existieren keine Einschränkungen hinsichtlich der Auslösezeiten der Arbeitsaufträge.~~

A6 Der Overhead durch Unterbrechungen, Ablaufplanung oder Verdrängung ist vernachlässigbar.

A7 Alle Aufgaben verhalten sich voll-präemptiv.

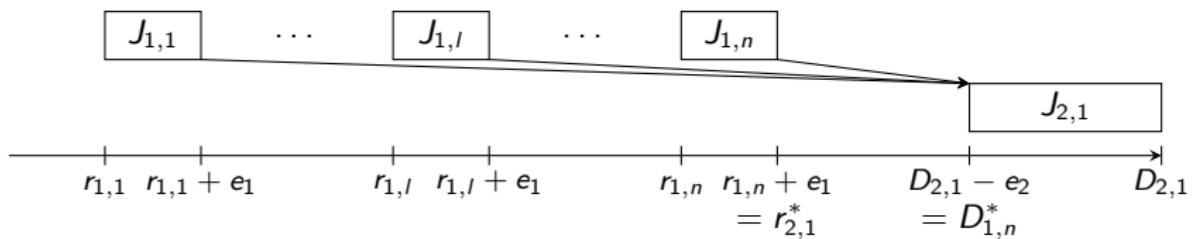
Abhängigkeiten \rightsquigarrow phasenverschobene Ausführung

Gerichtete Abhängigkeiten durch eine Modifikation des Planungsproblems auflösen

Verfahren analog zur Berechnung statischer Ablaufpläne (s. Folie VI/23):

- Abhängigkeiten schränken den zeitlichen Ablauf ein
- \rightsquigarrow formuliere **Auslösezeiten und Termine** so um, dass sie mit den Abhängigkeiten der Halbordnung übereinstimmen [1]

Beispiel: **ISR(SerialByte)** und **TASK(MsgHandler)** (s. Folie VI/11)



- $J_{2,1}$ kann frühestens nach $J_{1,n}$ starten
 - \rightsquigarrow angepasste Auslösezeit des Nachfolgers $r_{2,1}^* = \max_{1 \leq j \leq n} r_{1,j} + e_1$
- $J_{2,1}$ benötigt noch genügend Ausführungszeit
 - \rightsquigarrow angepasster Termin des Vorgängers $D_{1,n}^* = D_{2,1} - e_2$

Abhangigkeiten \rightsquigarrow phasenverschobene Ausführung (Forts.)

Gerichtete Abhangigkeiten durch eine Modifikation des Planungsproblems auflosen

- ① der Nachfolger J_i kann seine Ausführung erst dann beginnen, wenn seine Vorganger fertiggestellt wurden
 - \rightsquigarrow modifizierte die Auslosezeit des Nachfolgers

$$r_i^* = \max \{ r_i, \{ r_j^* + e_j | J_j \rightarrow J_i \} \}$$

- ② die Vorganger J_i mussen rechtzeitig fertig werden, so dass der Nachfolger seinen Termin einhalten kann
 - \rightsquigarrow modifizierte die Termine der Vorganger

$$D_i^* = \min \{ D_i, \{ D_j^* - e_j | J_i \rightarrow J_j \} \}$$

- ☞ anschlieend erfolgt die Ablaufplanung mit EDF
 - EDF ist auch fur derartige Systeme optimal (s. Folie IV-2/24)
 - fur Systeme mit statischen Prioritaten ist die Sache kniffliger ...
- ☞ Wirklich gut funktioniert das nur mit Abhangigkeitsgraphen!
 - Muster wie „2 von 3 Vorgangern“ erfordern angepasste Abbildungen

Gliederung

1 Überblick

2 Rangfolge und gerichtete Abhängigkeiten

- Datenabhängigkeiten
- Nebenläufigkeit
- Abhängigkeits- und Aufgabengraphen
- Koordinierung

3 Umsetzung

- Naive Implementierung
- Physikalisch und logische Ereignisse
- Implementierungsvarianten gerichteter Abhängigkeiten

4 Ablaufplanung

5 Zusammenfassung

Resümee

Rangfolge \leadsto gerichtete Abhängigkeiten

- resultieren oft aus Datenabhängigkeiten
- Abhängigkeitsgraphen und Aufgabengraphen
- gerichtete Abhängigkeiten in nebenläufigen Ausführungsumgebungen erfordern Koordinierung

Umsetzung gerichteter Abhängigkeiten \leadsto Koordinierung

- wohlgeordneter Ablauf von Produzent und Konsument
- Übergang zwischen zeitlichen Domänen
- Implementierung gerichteter Abhängigkeiten
 - implizit \leadsto statische Ablauftabellen, Phasenverschiebung
 - explizit \leadsto Aktivierung, Zeitsignale, Nachrichten

Ablaufplanung nutzt die Einschränkung des Ablaufverhaltens

- Nachfolger \leadsto modifizierte Auslösezeiten
- Vorgänger \leadsto modifizierte Termine

Literaturverzeichnis

- [1] ABDELZAHER, T. F. ; SHIN, K. G.:
Combined Task and Message Scheduling in Distributed Real-Time Systems.
In: *IEEE Transactions on Parallel and Distributed Systems* 10 (1999), Nr. 11, S.
1179–1191.
<http://dx.doi.org/10.1109/71.809575>. –
DOI 10.1109/71.809575. –
ISSN 1045–9219
- [2] AUTOSAR:
Specification of Operating System (Version 4.0.0) / Automotive Open System
Architecture GbR.
2009. –
Forschungsbericht
- [3] GOMAA, H. :
A software design method for real-time systems.
In: *Communications of the ACM* 27 (1984), Nr. 9, S. 938–949.
<http://dx.doi.org/10.1145/358234.358262>. –
DOI 10.1145/358234.358262. –
ISSN 0001–0782

Literaturverzeichnis (Forts.)

- [4] GOMAA, H. :
Structuring criteria for real time system design.
In: *Proceedings of the 10th International Conference on Software Engineering (ICSE '88)*.
New York, NY, USA : ACM Press, 1989. –
ISBN 0-8186-1941-4, S. 290–301
- [5] IEEE:
ISO/IEC IEEE/ANSI Std 1003.1-1996 Information Technology — Portable Operating System Interface (POSIX®) — Part 1: System Application: Program Interface (API) [C Language].
IEEE, New York : IEEE, 1996. –
784 S. –
ISBN 1-55937-573-6
- [6] LIU, J. W. S.:
Real-Time Systems.
Prentice-Hall, Inc., 2000. –
ISBN 0-13-099651-3

Literaturverzeichnis (Forts.)

- [7] OSEK/VDX GROUP:
Operating System Specification 2.2.3 / OSEK/VDX Group.
2005. –
Forschungsbericht. –
<http://portal.osek-vdx.org/files/pdf/specs/os223.pdf>, visited 2009-09-09