

# Echtzeitsysteme

## Abfertigung periodischer Echtzeitsysteme

Fabian Scheler

Lehrstuhl Informatik 4

04. November 2014

# Gliederung

- 1 Überblick
- 2 Periodische Aufgaben
- 3 Ereignisgesteuerte Ausführung
- 4 Zeitgesteuerte Ausführung
- 5 Zusammenfassung

# Fragestellungen

- Was zeichnet **periodische Echtzeitsysteme** aus?
  - Welches **Vorabwissen** ist in solchen Systemen verfügbar?
  - Reicht dies aus, um **sinnvolle Anwendungen** umzusetzen?
  - Welchen **Restriktionen** unterliegen solche Echtzeitsysteme?
- Basismechanismen für die Abarbeitung periodischer Jobs
  - **Ereignisgesteuerte Ausführung**
    - ereignisorientierte Einplanung
    - feste und dynamische Prioritäten
    - Berechnungskomplexität: Ablauftabelle vs. Ablaufliste
  - **Zeitgesteuerte Ausführung**
    - „*Busy Loop*“ vs. Ablauftabellen
    - zeitgesteuerte Abfertigung von Arbeitsaufträgen

# Gliederung

- 1 Überblick
- 2 Periodische Aufgaben
- 3 Ereignisgesteuerte Ausführung
- 4 Zeitgesteuerte Ausführung
- 5 Zusammenfassung

# Periodische Aufgabe (engl. *periodic task*)

## Vorabwissen

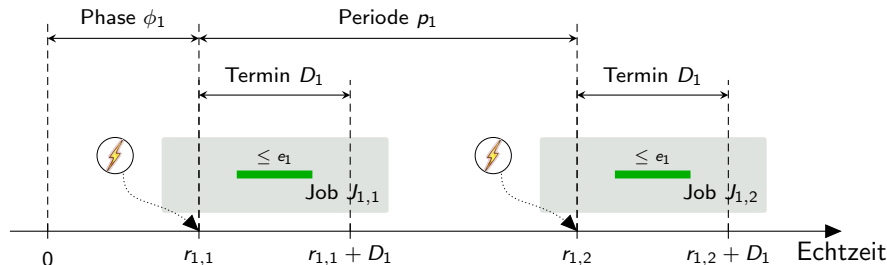
Aufgaben, die in (halb-) regelmäßigen Zeitintervallen kontinuierlich eine vorgegebene Systemfunktion erbringen<sup>1</sup>. Jede periodische Aufgabe  $T_i = (p_i, e_i, D_i, \phi_i)$  ist eine Abfolge von Arbeitsaufträgen mit

- $p_i$  der Periode (engl. *period*)
- $e_i$  der maximalen Ausführungszeit (WCET)
- $D_i$  dem relativen Termin (engl. *deadline*)
- $\phi_i$  der Phase (engl. *phase*)

---

<sup>1</sup>Nach [1] ist eine periodische Aufgabe nicht wirklich periodisch, da die Abstände zwischen den Auslösezeiten (engl. *interrelease time*) eines Arbeitsauftrags einer periodischen Aufgabe nicht der Periode selbst entsprechen müssen. Anderswo werden solche Aufgaben verschiedentlich als sporadische Aufgaben bezeichnet.

# Periodische Aufgaben auf der Echtzeitachse



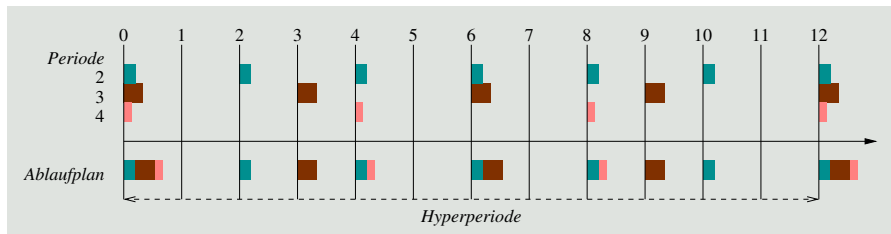
- Ausführungszeit  $e_i$**  maximale Ausführungszeit aller Jobs  $J_{i,j}$  in  $T_i$
- relativer Termin  $D_i$**  maximale Zeitspanne zwischen Auslösezeit  $r_{i,j}$  und Fertigstellung eines Jobs  $J_{i,j}$  in  $T_i$
- Periode  $p_i$**  minimale Länge aller Zeitintervalle  $[r_{i,j}; r_{i,j+1}]$  zwischen den Auslösezeiten der Jobs in  $T_i$
- Phase  $\phi_i$**  Auslösezeit  $r_{i,1}$  des ersten Jobs  $J_{i,1}$  in  $T_i$

# Hyperperiode

Mix verschiedener Aufgaben mit unterschiedlichen Perioden

Zeitintervall, in dem alle periodischen Aufgaben (mindestens einmal) durchgelaufen sind und erneut zusammen zur Ausführung anstehen:

- Hyperperiode  $H$ , das kleinste gemeinsame Vielfache aller Perioden
- führt ggf. zu **Schwankungen in den Einlastungszeiten**



- maximale Anzahl aller Arbeitsaufträge in  $H$  ist  $\sum_{i=1}^n H/p_i$ 
  - im vorliegenden Beispiel:  $(12/2) + (12/3) + (12/4) = 13$

# Periodische Echtzeitsysteme in der Praxis

Kann man Echtzeitsysteme ausschließlich aus periodischen Aufgaben aufbauen?

Echtzeitrechensysteme berechnen Stellwerte für Aktoren oft ohne ein zugrunde liegendes konventionelles (analoges) Kontrollsystem

- das kontrollierte Objekt erfährt dann eine **direkte digitale Kontrolle**
- die Kontrollanwendungen zeigen dabei eine hohe **Regelmäßigkeit**
  - eine meist endlose Sequenz von Kontrollperioden

## Rückgekoppelte Kontrollschleife (engl. *feedback control loop*)

initialisiere Stellwert;

initialisiere Zeitgeber und Unterbrecher;

**bei** Zeitgeberunterbrechung **erledige** /\* *abtasten, regeln, steuern* \*/

A/D-Wandlung der Echtzeitinstanz, Echtzeitabbild ziehen;

Echtzeitdatenbasis aktualisieren, neuen Stellwert berechnen;

D/A-Wandlung des Stellwerts, Echtzeitinstanz verändern;

**basta.**



# Periodische Echtzeitsysteme in der Praxis (Forts.)

Kann man Echtzeitsysteme ausschließlich aus periodischen Aufgaben aufbauen?



periodische Regelungsaufgaben:

alle 3 ms z.B. Sensorabtastung, Sensordatenfusion, Fluglageregelung

alle 9 ms z.B. Höhenregelung

☞ Die **zeitliche Auflösung** ( $d^{sample}$ ) der Regelung richtet sich nach der **Objektdynamik** ( $d^{object}$  bzw.  $d^{rise}$ ) (siehe Folie III-1/6).

# Restriktionen des periodischen Modells

Verzicht auf Entwicklungskomfort zugunsten einer realistischeren Analyse

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.

# Implikationen

Einschränkungen, die Einfluss auf Anwendungen ausüben

**Betriebsmittel:** *gemeinsame Betriebsmittel sind **nicht möglich!***

- gemeinsame Betriebsmittel  $\leadsto$  implizieren Synchronisation
- Aufgaben sind nicht mehr unabhängig

⚡ **I4Copter:** mehrere Sensoren am selben SPI-Bus

**Rangordnung:** *komplexe Aufgaben können **nicht geteilt werden!***

- eine komplexe Aufgabe wird in mehrere, einfachere Aufgaben aufgeteilt, die kooperativ einen Dienst zur Verfügung stellen
- Aufgaben sind nicht mehr unabhängig

⚡ **I4Copter:** Filter, Fusion und Regelung sind aufgeteilt

**Kommunikation:** *Aufgaben können **nicht synchron kommunizieren!***

- synchroner Nachrichtenversand/-empfang
- Aufgaben sind nicht mehr unabhängig

# Gliederung

- 1 Überblick
- 2 Periodische Aufgaben
- 3 Ereignisgesteuerte Ausführung**
- 4 Zeitgesteuerte Ausführung
- 5 Zusammenfassung

# Ereignisorientierte Einplanung

(engl. *event-driven scheduling*)

Einplanung von Arbeitsaufträgen erfolgt zu Ereigniszeitpunkten, die zufällig auftreten und somit nicht vorhersehbar sind

- die Ereignisverarbeitung unterliegt einer gewissen **Dringlichkeit**
- Ereignisauslöser sind kontrollierte Objekte/andere Arbeitsaufträge


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

**feste Zuordnung**  $\mapsto$  Ereignisverarbeitung/-auslöser

- gibt Arbeitsaufträgen eine **absolute Priorität**

**variable Zuordnung**  $\mapsto$  Ereignisverarbeitung

- gibt Arbeitsaufträgen eine **relative Priorität**

 auch: **prioritätsorientierte Einplanung** (engl. *priority-driven scheduling*)

# Prioritätsorientierte Algorithmen

## Klassifikation

Verfahren zur prioritätsorientierten Einplanung periodischer Jobs werden in zwei Gruppen eingeteilt:

**feste Priorität** (engl. *fixed priority* oder *static priority*)

- alle Jobs einer Task haben dieselbe unveränderliche Priorität
  - d.h., die Taskpriorität ist relativ zu anderen Tasks fest, unabhängig von der Auslösung bzw. Beendigung von Jobs
- ↪ Prioritäten werden **zum Entwurfszeitpunkt fest** vergeben

**dynamische Priorität** (engl. *dynamic priority*)

- die Priorität der Jobs einer Task sind veränderlich
  - d.h., die Taskpriorität variiert relativ zu anderen Tasks, wenn Jobs ausgelöst bzw. beendet werden
- ↪ Prioritäten werden **dynamisch zur Laufzeit** vergeben

# Verfeinerte Klassifikation

Ein Frage der Betrachtungsebene. . .

**Praxisrelevanz** haben Verfahren, die den Jobs feste Prioritäten zuweisen

- die Zuweisung erfolgt jedoch zum Auslösezeitpunkt eines Jobs
  - wenn er ereignisbedingt auf die **Bereitliste** (engl. *ready list*) kommt
- einmal zugewiesen, bleibt die Priorität eines ausgelösten Jobs gleich
  - jedoch immer nur in Relation zu allen anderen Jobs auf der Bereitliste
- auf Jobebene sind die Prioritäten fest, auf Taskebene aber variabel


Konsequenz daraus: verschiedene Kategorien von Einplanungsalgorithmen

**feste Priorität** wie gehabt (S. IV-1/14)

**dynamische Priorität auf Taskebene** (engl. *task-level dynamic-priority*)

**feste Priorität auf Jobebene** (engl. *job-level fixed-priority*)

**dynamische Priorität auf Jobebene** (engl. *job-level dynamic-priority*)

 **dynamische Priorität**  $\mapsto$  dynamisch auf Task- und fest auf Jobebene

# Verdrängbarkeit

Verschränkung (engl. *interleaving*) von Arbeitsaufträgen

Arbeitsaufträge könn(t)en verschränkt ausgeführt werden, wenn:

- diese verdrängbar sind (typischerweise durch den Planer),
- die Zeitbedingungen (engl. *time constraints*) es erlauben.

**Präemptivität** (engl. *preemptivity*) ist eine Eigenschaft des jeweiligen Arbeitsauftrags:


**verdrängbar** (engl. *preemptable*) ist ein Arbeitsauftrag, wenn seine Ausführung suspendiert werden darf

an beliebigen Stellen (engl. *fully preemptive*)

an ausgewiesenen Stellen (engl. *preemption points*)

**unverdrängbar** (engl. *non-preemptable*), sonst

- der Job läuft durch

 ggf. Mischbetrieb  $\leadsto$  Präemptivität als **Jobattribut** implementiert



# Ereignisorientierter Planer

(engl. *event-driven scheduler*)

Einplanung ereignisbedingt ausgelöster Arbeitsaufträge resultiert in einer **dynamischen Datenstruktur**  $\mapsto$  „sortierte Liste“

- der Vorgang ist gekoppelt mit der Einlastung: **online scheduling**
- kritisch ist die **Berechnungskomplexität** und wann sie anfällt
  - konstant oder variabel, dann jedoch mit oberer Schranke  $\mapsto$  WCET
  - zum Auslöse- oder Auswahlzeitpunkt von Arbeitsaufträgen

**Sortierschlüssel** (engl. *sort key*) zur Reihung eines Arbeitsauftrags ist die ihm zugeordnete Priorität

- ergibt sich ggf. erst zum Ereigniszeitpunkt aus der Priorität der von ihm zu verarbeitenden **Klasse von Ereignissen**
- ist eindeutig abzubilden auf einen endlichen Wertebereich

 auch: **prioritätsorientierter Planer** (engl. *priority-driven scheduler*)

# Berechnungskomplexität

## Zeitpunkte ihrer Wirksamwerdung

### Auslösezeitpunkt (nach dem Ereigniszeitpunkt)

- konstanter Aufwand, im Falle einer Ablaufabelle
  - Jobs durch indizierte Adressierung in die Tabelle aufnehmen
  - ggf. ist ein Tabelleneintrag eine Jobliste (FIFO) gleicher Priorität
- (beschränkter) linearer Aufwand, im Falle einer Ablaufliste
  - Vorabwissen zur **WCET des Sortiervorgangs** ist gefordert

### Auswahlzeitpunkt (nach dem Auslöse-, vor dem Einlastungszeitpunkt)

- nahezu konstanter Aufwand, im Falle einer Ablaufliste
  - Jobs vom Kopf her der (ggf. einfach verketteten) Liste entnehmen
- beschränkter linearer Aufwand, im Falle einer Ablaufabelle
  - 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** nahezu  $O(1)$

### Ablauftabelle

```
Job* table[Jobs];

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

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

    return 0;
}
```

**release**  $O(1)$

**extract**  $O(n)$ , **obere Schranke** Jobs

# 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

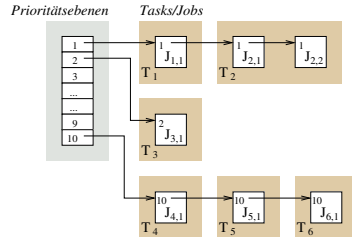
```
Job* table[Jobs];

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

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

    return 0;
}
```

- mehrere Tasks pro Priorität
- mehrere Jobs pro Task
- Reihenfolge der Auslösung



# Welche Datenstrukturen für welchen Einsatzzweck?

Feste/Dynamische Prioritäten und Ablauftabellen/-listen

Ablauf**ta**bellen haben eine **statische** Struktur

- Prioritäten werden fest auf Tabellenindizes abgebildet
- ~> das Prioritätsgefüge ist zur Laufzeit unveränderbar
  - es werden absolute Prioritäten von Aufgaben verwaltet
- ~> Eignung für die Implementierung **fester Prioritäten**

Ablauf**li**sten haben eine **dynamische** Struktur

- Prioritäten entsprechen der Position innerhalb der Ablaufliste
- ~> das Prioritätsgefüge passt sich zur Laufzeit an
  - Prioritäten werden relativ zu anderen Arbeitsaufträgen verwaltet
- ~> Eignung für die Implementierung **dynamischer Prioritäten**
  - auch andere dynamische, (partiell) sortierte Datenstrukturen sind hier denkbar, z.B. binäre Haufen (engl. *binary heap*)

# Prioritätsorientierter $O(1)$ -Scheduler

!?

Die Tücke liegt oft im Detail...

**Jobauslösung** mit **konstantem Aufwand**,  $O(1)$ , ist möglich, sofern gilt:

- ① der Ablaufplan ist eine auf mehrere Prioritätsebenen aufgeteilte dynamische Datenstruktur, repräsentiert als Tabelle von...
  - Wartelisten  $\leadsto$  LIFO
  - Warteschlangen  $\leadsto$  FIFO
- ② alle Jobs, die über denselben Tabelleneintrag erfasst werden, besitzen auch dieselbe Priorität  $\leadsto$  **Prioritätsschlange**
  - sonst könnte LIFO/FIFO **Prioritätsverletzung** zur Folge haben
- ③ die Anzahl der Tabelleneinträge entspricht mindestens der Anzahl statisch zugewiesener Prioritäten
  - ggf. werden dann nahezu alle Tabelleneinträge nur einen Job erfassen
  - hängt ab von der Echtzeitanwendung und dem Einplanungsverfahren

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

- begrenzt viele leere Tabelleneinträge sind ggf. zu überspringen

# Prioritätsorientierter $O(1)$ -Scheduler (Forts.)

!?

Eine Abwägungsfrage...

Vorrangsteuerung ist mit einem grundsätzlichen Konflikt konfrontiert:

- **entweder** Jobauslösung **oder** Jobauswahl mit  $O(1)$  zu versehen
  - beides zugleich geht nicht

... für **Jobauslösung in  $O(1)$**  spricht:

- ereignisgesteuerte Einplanung/Einlastung benötigen konstante Zeit
  - als Folge eines *Interrupts* oder der Zustellung eines „Zeitsignals“
  - bedeutsam für voll-verdrängbare Systeme
- ereignisbedingte Jobverzögerungen lassen sich exakt bestimmen

... für **Jobauswahl in  $O(1)$**  spricht:

- der Übergang zum jew. nachfolgenden Job benötigt konstante Zeit
  - wenn z.B. der aktuelle Job durchgelaufen ist oder blockiert

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

# Gliederung

- 1 Überblick
- 2 Periodische Aufgaben
- 3 Ereignisgesteuerte Ausführung
- 4 Zeitgesteuerte Ausführung**
- 5 Zusammenfassung



# Die „Busy Loop“

Die wirklich einfachste Variante für die Implementierung zyklischer Systeme?

Periodische Aufgaben wiederholt in einer Schleife ausführen

```
int main(void) {
    unsigned long cnt = 0;

    while(1) {
        warte_durchlauf();

        kontrolle_start();
        aufgabe1();
        kontrolle_stop();

        if(counter % 2 == 0) {
            aufgabe2_1();
        }

        10ms_nach_aufgabe1();

        if(counter % 2 == 0) {
            aufgabe2_2();
        }

        cnt++;
    }

    return 0;
}
```

- längere Perioden lassen sich durch einen **Rundenzähler** ableiten
  - die Schleife definiert einen **Rahmen**  
~> ausrichtendes Raster für **alle Aktivitäten**
- Explizite Überwachung der **Rahmendauer**
  - Ausführungszeit ist i.d.R. **nicht konstant**
- Schwierige Spezifikation **zeitlichen Versatzes**
  - Abhängigkeit von der **tats. Ausführungszeit**
- Konflikte durch **lange andauernde Jobs**
  - evtl. ist eine **manuelle Aufteilung** nötig
- **Überwachung** der Ausführungszeit
  - schwieriger **Abbruch** des betroffenen Jobs

# Ein Ablaufplan gibt den Takt vor

Falls alle Parameter der Arbeitsaufträge im Voraus bekannt sind ...

**Vorwissen** bahnt den Weg, um Ablaufpläne *off-line* erstellen zu können

- alle Programme und das System verhalten sich **deterministisch**
  - oder noch besser  $\leadsto$  **vorhersagbar**

**statischer Ablaufplan**  $\mapsto$  exakter Jobfahrplan; enthält feste Angaben darüber, wann welche Arbeitsaufträge auszuführen sind

- die jedem Arbeitsauftrag zugeteilte Prozessorzeit ist gleich seiner maximalen Ausführungszeit  $\leadsto$  WCET
- Einlastung der Arbeitsaufträge geschieht streng nach Fahrplan
  - alle Termine werden im Normalfall sicher eingehalten
  - unvorhergesehene Ausnahmen<sup>2</sup> führen zu Terminüberschreitungen
- da die Einplanung *off-line* geschieht, können Algorithmen mit hoher Berechnungskomplexität zum Einsatz kommen

---

<sup>2</sup>Gemeint sind hier die synchronen Programmunterbrechungen (d.h., *Traps*), z.B. aufgrund von Berechnungs- und/oder Adressierungsfehlern.

# Abarbeitung statischer Ablaufpläne

## Tabellengesteuerte Einlastung von Arbeitsaufträgen

Repräsentation vorberechneter (statischer) Ablaufpläne  $\leadsto$  **Tabelle**

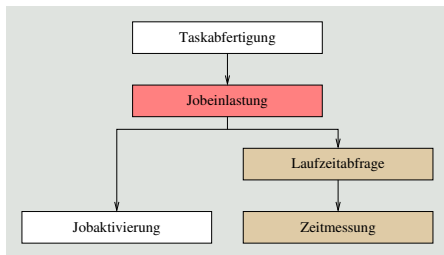
- jeder Tabelleneintrag entspricht einer Einplanungsentscheidung zu einem (vorab) bestimmten Zeitpunkt auf der Echtzeitachse
- bei Einlastung wird ein **Zeitgeber** (engl. *timer*) programmiert und der Arbeitsauftrag wird gestartet
  - „Kurzzeitwecker“ auf nächsten Entscheidungszeitpunkt stellen
  - einzustellender Wert ist im aktuellen Tabelleneintrag zu finden
- ein **Zeitgebersignal** schaltet zum nächsten Tabelleneintrag weiter

**Reihumverfahren**: am Tabellenende wird wieder zum -anfang gesprungen

- **zyklischer Ablaufplan** (engl. *cyclic schedule*) periodischer Aufgaben

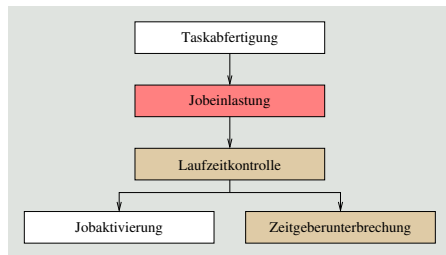
# Abfertigung von Arbeitsaufträgen

Abfragebetrieb (engl. *polling mode*) vs. Unterbrecherbetrieb (engl. *interrupt mode*)



## Abfragebetrieb

(S. IV-1/30 bis IV-1/31)



## Unterbrecherbetrieb

(S. IV-1/32 bis IV-1/34)

## Benutzthierarchie

Die Benutzbeziehung [3] in einer funktionalen Hierarchie drückt Abhängigkeiten von der Verfügbarkeit korrekter Implementierungen von Funktionen aus. *A benutzt B*, wenn die korrekte Ausführung von *B* zwingend ist für die Korrektheit von *A*: d.h., die Korrektheit von *A* hängt ab von der Korrektheit von *B* (*A* liegt über *B*).

# Tabellengesteuerte Einlastung zyklischer Arbeitsaufträge

Taskabfertigung: Grundsätzliche Verfahrensweise

```
erledige Dispatcher (Ablaufabelle, Tabellenlänge):  
  setze Laufzähler auf ersten Eintrag der Ablaufabelle;  
  solange der Betrieb läuft tue  
    erledige  
      laste Ablaufabelle[Laufzähler].Arbeitsauftrag ein;  
      wenn Laufzähler < Tabellenlänge dann erhöhe Laufzähler um 1  
      sonst setze Laufzähler auf ersten Eintrag der Ablaufabelle;  
  basta;  
basta.
```

Einlastung der Arbeitsaufträge verläuft in drei grundsätzlichen Schritten:

- 1 Laufzeitüberwachung des anstehenden Arbeitsauftrags aufsetzen
- 2 anstehenden Arbeitsauftrag starten und ausführen
- 3 sich auf den nächsten Entscheidungszeitpunkt **synchronisieren**

# Synchronisation durch Abfrage eines Taktzählers

Jobeinlastung, Laufzeitabfrage und Zeitmessung

erledige laste ein (Arbeitsauftrag):

interpretiere Arbeitsauftrag. Entscheidungszeitpunkt als Taktzahl;

aktiviere Arbeitsauftrag;

solange Taktzähler < Taktzahl tue nichts;

basta.

Grundlage bildet ein **Taktzähler** (engl. *clock counter*) in der Hardware

- der Entscheidungszeitpunkt muss als Taktzahl vorliegen oder in eine Taktzahl umgerechnet werden können
  - diese Taktzahl wird nach Beendigung des Arbeitsauftrags abgewartet
- gezählt werden z.B. die CPU-Takte bei Befehlsausführung



Verzögerung von Arbeitsaufträgen kann Spätfolgen nach sich ziehen

# Abfragebetrieb im Rückblick

## Verzögerungsproblematik bei Taktzähler und Zeitkontrolle

Abtastung des Zeitgebers durch das **im Vordergrund** laufende Programm

- nachdem ein aktivierter Arbeitsauftrag komplett durchgelaufen ist
  - Arbeitsaufträge erhalten einen gewissen Vertrauensvorschuss
  - evtl. Terminüberschreitungen werden erst im Nachhinein erkannt
- schwache/strikte Echtzeitfähigkeit liegt ganz in Anwendungshand  
**schwach** bei Terminüberschreitung, Ergebnis findet Verwendung
  - der nachfolgende Arbeitsauftrag startet verspätet
  - als Folge kann das System komplett aus den Takt geraten
- **strikt** sonst, d.h., wenn Termineinhaltung jederzeit garantiert ist
- die WCET muss die Behandlung evtl. Fehlersituationen einschließen

 Alternative: **Zeitgeberunterbrechung** (engl. *timer interrupt*)

# Synchronisation durch unterbrechenden Zeitgeber

Jobeinlastung: Einseitige Synchronisation mit Zeitgeberunterbrechung

**erledige** laste ein (Arbeitsauftrag):  
stelle Zeitgeber ein auf Arbeitsauftrag. Entscheidungszeitpunkt;  
kontrolliere Arbeitsauftrag;  
**solange** Zeitgebersignalmarke ungesetzt ist **tue** nichts;  
setze Zeitgebersignalmarke zurück;  
**basta**.

Anzeige des Zeitgebersignals durch ein **im Hintergrund** arbeitendes Gerät

- Ausführungsfreigabe durch **Softwaresignal** der Behandlungsroutine
  - hier: die Zeitgebersignalmarke, die beim Konsumieren gelöscht wird
  - der *Dispatcher* synchronisiert sich mit dem Zeitgeber
- Abbruch des Arbeitsauftrags als Folge einer Zeitgeberunterbrechung
  - sofern der Arbeitsauftrag dann noch in Ausführung befindlich war
  - ist in Bezug auf die WCET des Arbeitsauftrags ein Ausnahmefall



# Synchronisation durch unterbrechenden Zeitgeber (Forts.)

Laufzeitkontrolle, Zeitgeberunterbrechung: Bedingter Jobabbruch

**erledige** Behandlungsroutine zum *Timer Interrupt*:

**wenn** Arbeitsauftrag.Zustand = laufend **dann** breche Arbeitsauftrag ab;  
setze Zeitgebersignalmarke;

**basta**.

Erfüllung der Wartebedingung für den (aktiv wartenden) *Dispatcher*

- ggf. Abbruch eines seinen Termin überschreitenden Arbeitsauftrags

**erledige** kontrolliere (Arbeitsauftrag):

setze Arbeitsauftrag.Zustand auf laufend;  
aktiviere Arbeitsauftrag;  
setze Arbeitsauftrag.Zustand auf beendet;

**basta**.

„Schönheitsfehler“:

- Zustand
- Signalmarke
- unnötiger *Interrupt*

# Synchronisation durch unterbrechende Zeitkontrolle

Jobeinlastung, Laufzeitkontrolle, Zeitgeberunterbrechung: Unbedingter Jobabbruch

**erledige** Behandlungsroutine zum *Timer Interrupt*:  
    breche Arbeitsauftrag ab;  
**basta**.

**erledige** kontrolliere (Arbeitsauftrag):  
    lasse Unterbrechung durch Zeitkontrolle zu;  
    aktiviere Arbeitsauftrag;  
    wehre Unterbrechung durch Zeitkontrolle ab;  
**basta**.

**Ausnahmefall** die  
    Zeitkontrolle läuft  
    bei Überschreitung  
    der WCET des  
    Arbeitsauftrags ab

**erledige** laste ein (Arbeitsauftrag):  
    richte Zeitkontrolle aus auf Arbeitsauftrag. Entscheidungszeitpunkt;  
    kontrolliere Arbeitsauftrag;  
    **solange** Zeitkontrolle  $\neq 0$  **tue** nichts;  
**basta**.

# Stapelbasierte Abarbeitung von Ablauftabellen

Mischung aus lang andauernden und häufig wiederkehrenden Jobs unterstützen

**Batch Processing** führt einen Job nach dem anderen aus

- lang andauernde Jobs verzögern kurze, häufig wiederkehrende Jobs
- ~ diese Jobs verpassen u.U. deshalb ihre Termine
  - oder lange andauernde Jobs werden in handliche „Happen“ zerteilt

**Stapelbasierte Abarbeitung von Ablauftabellen** erlaubt Verdrängung

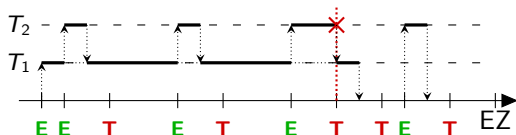
- der eingelastete Job verdrängt den aktuell ausgeführten Job
  - der ausgeführte Job wird nicht abgebrochen
- ~ mehrere *kurze* Jobs **während** eines *langen* Jobs ausführen
  - die Kontrolle des ausgeführten Jobs wird schwieriger
    - ein Entscheidungszeitpunkt ermöglicht die Einlastung **oder** die Kontrolle eines Jobs, **beides zugleich ist i.A. nicht möglich**
    - die Ausführungszeit eines Jobs muss explizit protokolliert werden
    - alternativ wird eine **Terminüberwachung** statt einer Laufzeitkontrolle durchgeführt (z.B. OSEKtime [2])

# Stapelbasierte Abarbeitung von Ablauftabellen

Beispiel –  $T_1 = (20, 10, 15, 0)$ ,  $T_2 = (5, 1, 2, 1)$

mögliche Ablauftabelle:

Aktion	Aufgabe	Zeit
E	$T_1$	0
E	$T_2$	1
T	$T_2$	3
E	$T_2$	6
T	$T_2$	8
E	$T_2$	11
T	$T_2$	13
T	$T_1$	15
E	$T_2$	16
T	$T_2$	18



$t = 0$   $T_1$  einlasten

$t = 1, 6, 11$   $T_2$  einlasten,  $T_1$  verdrängen

$t = 2, 7$   $T_2$  terminiert,  $T_1$  fortsetzen

$t = 13$   $T_2$  verfehlt seinen Termin  
 $\leadsto$  Ausnahme auslösen,  $T_2$  abbrechen

$t = 14$   $T_1$  terminiert

$t = 16$   $T_2$  einlasten

$t = 17$   $T_2$  terminiert

# Gliederung

- 1 Überblick
- 2 Periodische Aufgaben
- 3 Ereignisgesteuerte Ausführung
- 4 Zeitgesteuerte Ausführung
- 5 Zusammenfassung**

# Resümee

**Periodische Aufgaben** haben in Echtzeitsystemen eine weite Verbreitung

- Periode, Phase, Hyperperiode, digitale Kontrollschleife
- Restriktionen periodischer Aufgaben und ihre Einschränkungen

**Ereignisgesteuerte Ausführung** periodischer Aufgaben

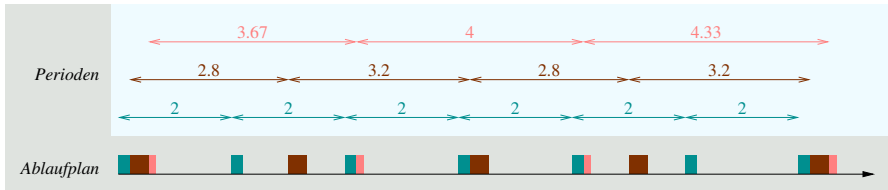
- Ereignis- bzw. prioritätsorientierte Einplanung
- Feste und dynamische Prioritäten auf Task- bzw. Job-Ebene
- Auslösung vs. Auswahl, Ablaufliste vs. Ablauftabelle
- *Multi-Level-Queue-Scheduler*, Prioritätsorientierter  $O(1)$ -Scheduler

**Zeitgesteuerte Ausführung** periodischer Aufgaben

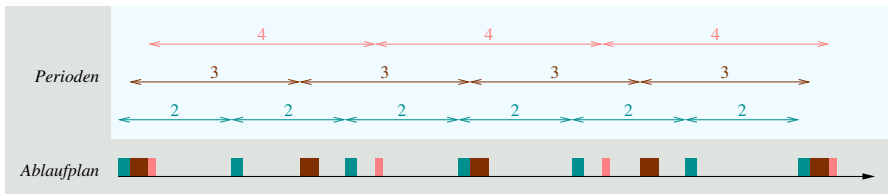
- naive „*Busy Loop*“-Implementierung und Ablauftabellen
- Laufzeitkontrolle im Abfrage- und Unterbrecherbetrieb
- stapelbasierte Ablaufplanung

# Genauigkeit periodischer Aufgaben

## Einfluss der Einplanung auf Schwankungen in der Einlastung



- bis auf Periode 2 sind alle anderen Jobs nicht wirklich periodisch



- alle Jobs laufen wirklich periodisch ab: Jobabstand = Periode

# Literaturverzeichnis

- [1] LIU, J. W. S.:  
*Real-Time Systems*.  
Prentice-Hall, Inc., 2000. –  
ISBN 0-13-099651-3
  
- [2] OSEK/VDX GROUP:  
Time Triggered Operating System Specification 1.0 / OSEK/VDX Group.  
2001. –  
Forschungsbericht. –  
<http://portal.osek-vdx.org/files/pdf/specs/ttos10.pdf>
  
- [3] PARNAS, D. L.:  
Some Hypotheses About the “Uses” Hierarchy for Operating Systems / TH Darmstadt.  
1975 (BS I 75/2). –  
Forschungsbericht