

Überblick

Grundlagen

Überblick

Einplanungseinheit

Programmunterbrechung

Unterbrechungstechnik

Zusammenfassung

Fragestellungen

- ▶ Echtzeitrechner nehmen **Arbeitsaufträge** entgegen und bearbeiten die an sie gestellten **Aufgaben** fristgemäß
 - ▶ was ist ein Arbeitsauftrag?
 - ▶ was ist eine Aufgabe?
- ▶ während der Ausführung von Arbeitsaufträgen dürfen ggf. auftretende **Programmunterbrechungen** die Einhaltung von Fristen nicht verletzen
 - ▶ was ist eine Programmunterbrechung?
 - ▶ was sind in dem Zusammenhang die Verwaltungsgemeinkosten?
- ▶ die **Erkennung** und **Behandlung** von Programmunterbrechungen hat großen Einfluss auf die Robustheit von Rechnersystemen
 - ▶ was bedeutet Unterbrechungserkennung?
 - ▶ was muss Unterbrechungsbehandlung immer leisten?

Ausführungsstrang

Logische Einheit der Ablaufplanung (engl. *unit of scheduling*)

Abstraktion (des Betriebssystems) von einem beliebigen Programm in Abarbeitung bzw. Ausführung ist der **Prozess**

- ▶ im Prozess existieren ggf. mehrere Ausführungsstränge gleichzeitig
 - ▶ nebenläufiges Programm (engl. *concurrent program*)
 - ▶ durchzogen mit mehr als einen **Programmfa**den (engl. *thread*)
- ▶ der Kontext eines Fadens manifestiert sich im **Prozessorstatus**
 - ▶ physisch die Inhalte der Arbeits- und Statusregister der CPU
 - ▶ ggf. erweitert um Segment-/Seitendesriptoren von MMU bzw. TLB
- ▶ Einlastung (engl. *dispatching*) eines Fadens bedeutet **Kontextwechsel**

Instanzen von Prozessen repräsentieren sich als Fäden mit oder ohne eigenen Adressräumen, aber auch als bloße „Prozeduren“

- ▶ bei kooperativer Einplanung und Verzicht auf Synchronisationspunkte

Aufgabe (engl. *task*)

A *task* is the execution of a sequential program. It starts with reading of the input data and of the internal state of the task, and terminates with the production of the results and updating the internal state. [1, S. 75]

einfache Aufgabe (engl. *simple task*) ohne Synchronisationspunkt

- ▶ läuft durch, ohne zu blockieren
- ▶ ist von wichtigeren Aufgaben ggf. verdrängbar

komplexe Aufgabe (engl. *complex task*) mit Synchronisationspunkt

- ▶ erwartet die Zuteilung von Betriebsmitteln
 - ▶ wiederverwendbare und/oder konsumierbare BM
- ▶ hängt ab von der Bearbeitung anderer Aufgaben

Arbeitsauftrag (engl. *job*)

*We call each unit of work that is scheduled and executed by the system a **job** and a set of related jobs which jointly provide some system function a **task**. [2, S. 26]^a*

^aNach [1, S. 75] müssten dann die Arbeitsaufträge einer Aufgabe strikt nacheinander ablaufen, damit der sequentielle Charakter der Aufgabe gewahrt ist. In [2] entsteht aber ein anderes Bild: Arbeitsaufträge werden ggf. nebenläufig ausgeführt.

Einheit der Einplanung ist der Arbeitsauftrag und nicht der Faden, denn...

- ▶ alle Arbeitsaufträge könnten von nur einem Faden ausgeführt werden
- ▶ bei einfädigen Systemen ist Fadeneinplanung irrelevant und sinnlos

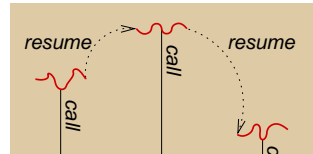
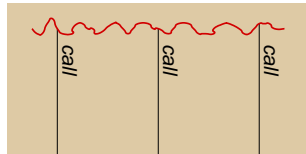
Ausführungsstränge und Arbeitsaufträge

Dualität von Betriebssystemstrukturen

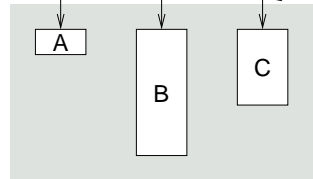
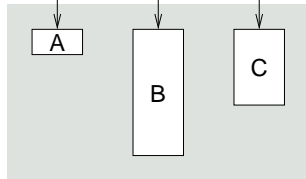
einfädige Aufgabe

mehrfädige Aufgabe

Ausführungsstränge



Arbeitsaufträge



- ▶ die Arbeitsaufträge A, B und C sind Prozeduren (der Anwendung)
- ▶ die Prozeduren werden nacheinander aufgerufen bzw. aktiviert
 - ▶ alle von einem Faden oder jede von einem eigenen Faden

Generische Systemsoftware

Abarbeitung einer Ablaufabelle (engl. *schedule table*)

Schablone (engl. *template*) für Arbeitsaufträge

```
template<class Dingus> class Role : public Dingus { ... };
```

```
class Action {  
public:  
    void operator() () { ... }  
    ...  
};
```

```
template<class Thingy> class Thread : private Coroutine {  
public:  
    static Thread* master();    // dispatcher  
  
    void operator() () { master()->resume(*this); }  
    ...  
};
```

```
typedef Role<Action> Job;
```

```
typedef Role< Thread<Action> > Job;
```

Abfertiger (engl. *dispatcher*)

```
void execute (Job item[], unsigned short high) {  
    unsigned short next = 0;  
    while (item) (item[next < high ? next++ : next = 0])();  
}
```

Generische Systemsoftware (Forts.)

Wiederverwendung (engl. *reuse*) der Implementierung eines Arbeitsauftrags

Arbeitsauftrag \mapsto aktives Objekt

```
template<class Thingy> class Thread : private Coroutine {
protected:
    void action () {
        for (;;) {
            (*(Thingy*)this)();    // perform the job
            resume(*master());     // pause, reactivate dispatcher
        }
    }
    ...
};
```

- ▶ Ausführung einer Arbeitsauftragsprozedur im eigenen Faden
 - ▶ Prozedur `Thingy::operator()()` wird vom Faden wiederverwendet
 - ▶ die **Prozedur implementiert**, der **Faden exekutiert** die Kontrollfunktion
- ▶ Faustregel: Arbeitsaufträge als **funktionale Abstraktion** auslegen

Generische Systemsoftware (Forts.)

Gemeinsamkeiten und Unterschiede

funktionale Eigenschaft (engl. *functional property*)

- ▶ die auszuführende Kontrollfunktion
 - ▶ abtasten, regeln, steuern

nicht-funktionale Eigenschaft (engl. *non-functional property*)

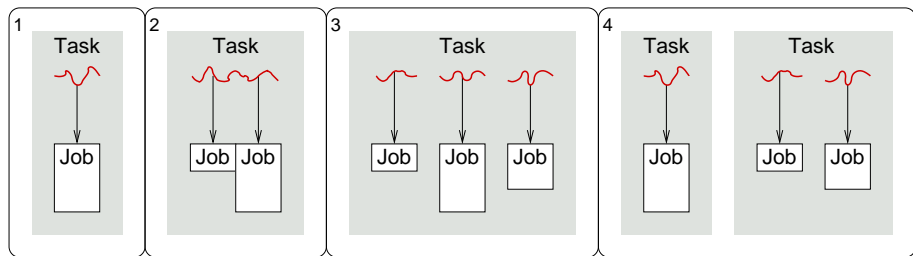
- ▶ die **Architektur** des Systems
 - ▶ prozedur-, objekt-, prozessorientiert
- ▶ die **Performanz** des Systems
 - ▶ Geschwindigkeit, Energie, Speicher
- ▶ ...

Trennung von Belangen (engl. *separation of concerns*)

- ▶ Implementierungen beider Eigenschaftsarten voneinander trennen
 - ▶ in erster Linie eine strukturelle Maßnahme \rightsquigarrow „Bleistift und Papier“
- ▶ unter Berücksichtigung der vorgegebenen **Echtzeitbedingungen** !!!

Aufgabe vs. Arbeitsauftrag vs. Faden

Konfigurationsbeispiele für jeweils ein Rechensystem



1. eine einfädige Aufgabe, ein Arbeitsauftrag
2. eine einfädige Aufgabe, zwei Arbeitsaufträge
3. eine mehrfädige Aufgabe, drei Arbeitsaufträge
 - ▶ ggf. jeder Arbeitsauftrag/Faden mit eigenem Adressraum
4. zwei Aufgaben (ein- und mehrfädig), drei Arbeitsaufträge
 - ▶ ggf. jede Aufgabe mit eigenem Adressraum

Verwaltungsgemeinkosten (engl. *overhead*)

Eine Frage der Repräsentation von Aufgabe und Arbeitsauftrag

1. einfädige Aufgabe \rightsquigarrow fliegengewichtig

- ▶ $O(\text{Prozeduraufruf})$
- ▶ Auf- und Abbau vom Aktivierungsblock (engl. *activation record*)

2. mehrfädige Aufgabe

2.1 gemeinsamer Adressraum \rightsquigarrow federgewichtig

- ▶ $O(\text{Fadenwechsel}) + O(1.)$
- ▶ Austausch der Inhalte von Arbeits-/Statusregister der CPU

2.2 separierter Betriebssystemkern \rightsquigarrow leichtgewichtig

- ▶ $O(\text{Systemaufruf}) + O(2.1.)$
- ▶ Behandlung der synchronen Programmunterbrechung (engl. *trap*)

2.3 getrennte Adressräume \rightsquigarrow schwergewichtig

- ▶ $O(\text{Adressraumwechsel}) + O(2.2.)$
- ▶ Löschen/**Laden** vom Zwischenspeicher (engl. *cache*) der MMU

▶ bis auf $O(2.3)$ haben alle anderen Fälle konstanten Aufwand

Verdrängbare Aufgabe

Unterbrechung und Wiederaufnahme der Bearbeitung eines Arbeitsauftrags

Aufgaben, die sich in Ausführung befinden, können **Verdrängung** erfahren

- ▶ dem Faden des laufenden Arbeitsauftrags wird die CPU entzogen:
 1. eine asynchrone Programmunterbrechung (engl. *interrupt*) tritt auf
 2. der unterbrochene Faden wird als lafbereit (erneut) eingeplant
 3. ein anderer lafbereiter Faden wird ausgewählt und eingelastet
- ▶ eine Systemfunktion, die Bedingungen zum korrekten Ablauf stellt:
 - ▶ asynchrone Programmunterbrechungen müssen möglich sein
 - ▶ die Behandlungsroutine muss den Planer (engl. *scheduler*) aktivieren
 - ▶ der Planer muss verdrängend arbeiten (engl. *preemptive scheduling*)
 - ▶ mindestens ein anderer lafbereiter Faden muss zur Verfügung stehen

Verdrängung ist als **nicht-funktionale Systemeigenschaft** anzusehen

- ▶ die an anderen Programmstellen Synchronisationsbedarf impliziert
- ▶ die transparent für die betroffene Aufgabe sein muss

Verdrängbare Aufgabe (Forts.)

Verarbeitung der Ausführungsstränge frei von Seiteneffekten

Transparenz (engl. *transparency*) von Verdrängung meint zweierlei:

1. der Zustand eines unterbrochenen Fadens ist invariant
 - ▶ erfordert Sicherung und Wiederherstellung des Prozessorstatus
 - ▶ Maßnahmen zur **Einlastung** (engl. *dispatching*) von Fäden
 2. die verzögerte Ausführung des Fadens verletzt keine Fristen
 - ▶ erfordert Vergabe, Überwachung und Einhaltung von Dringlichkeiten
 - ▶ jeder Faden ist von statischer/dynamischer Priorität (engl. *priority*)
 - ▶ Maßnahmen zur **Einplanung** (engl. *scheduling*) von Fäden
- ▶ einfache nicht-verdrängbare Aufgaben können auf 1. verzichten
- ▶ einfache verdrängbare oder komplexe jedoch nicht
- ▶ für Echtzeitrechensysteme ist 2. ggf. sogar verzichtbar
- ▶ sofern die Umgebung keine harten Echtzeitbedingungen vorgibt

Unterbrechungsarten

Zwei Arten von Programmunterbrechungen werden unterschieden:

synchron die „Falle“ (engl. *trap*)

asynchron die „Unterbrechung“ (engl. *interrupt*)

Unterschiede ergeben sich hinsichtlich...

- ▶ Quelle
- ▶ Synchronität
- ▶ Vorhersagbarkeit
- ▶ Reproduzierbarkeit

☞ Behandlung ist zwingend und grundsätzlich prozessorabhängig

Synchrone Programmunterbrechung

- ▶ unbekannter Befehl, falsche Adressierungsart oder Rechenoperation
- ▶ Systemaufruf, Adressraumverletzung, unbekanntes Gerät
- ▶ Seitenfehler im Falle lokaler Ersetzungsstrategien

Trap \mapsto synchron, vorhersagbar, reproduzierbar

- ▶ geschieht abhängig vom Arbeitszustand des laufenden Programms:
 - ▶ unverändertes Programm, mit den selben Eingabedaten versorgt
 - ▶ auf ein und dem selben Prozessor zur Ausführung gebracht
- ▶ die Unterbrechungsstelle im Programm ist vorhersehbar

☞ die Programmunterbrechung/-verzögerung ist determiniert

Asynchrone Programmunterbrechung

- ▶ Signalisierung „externer“ Ereignisse
- ▶ Beendigung einer DMA- bzw. E/A-Operation
- ▶ Seitenfehler im Falle globaler Ersetzungsstrategien

Interrupt \mapsto asynchron, unvorhersagbar, nicht reproduzierbar

- ▶ geschieht unabhängig vom Arbeitszustand des laufenden Programms:
 - ▶ hervorgerufen durch einen „externen Prozess“ (z.B. ein Gerät)
 - ▶ ein Ereignis signalisierend
- ▶ die Unterbrechungsstelle im Programm ist nicht vorhersehbar

☞ die Programmunterbrechung/-verzögerung ist **nicht determiniert**

Asynchrone Programmunterbrechung (Forts.)

Interrupts machen determinierte Programme nicht-deterministisch:³

determinierte Programme

- ▶ lassen bei ein und derselben Eingabe verschiedene Abläufe zu
- ▶ alle Abläufe liefern jedoch stets das gleiche Resultat
 - ▶ nebenläufige Programme können also sehr wohl determiniert sein

nicht-deterministisch

- ▶ nicht zu jedem Zeitpunkt ist bestimmt, wie weitergefahren wird
 - ▶ wann *Interrupts* (exakt) auftreten werden, ist unvorhersagbar

³Dies gilt genau genommen auch im Falle eines strikt zeitgesteuerten Systems, da physikalische und elektrotechnische Effekte aber auch atmosphärische Störungen zu Ungenauigkeiten des Zeitgebers führen können. Das Problem wird lediglich auf ein Minimum reduziert.

Trap/Interrupt \rightsquigarrow Ausnahmesituationen

Ereignisse, oftmals unerwünscht aber nicht immer eintretend:

- ▶ Signale von der Peripherie (z.B. E/A, Zeitgeber oder „Wachhund“)
- ▶ Wechsel der Schutzdomäne (z.B. Systemaufruf)
- ▶ Programmierfehler (z.B. ungültige Adresse)
- ▶ unerfüllbare Speicheranforderung (z.B. bei Rekursion)
- ▶ Einlagerung auf Anforderung (z.B. beim Seitenfehler)
- ▶ Warnsignale von der Hardware (z.B. Energiemangel)

Ereignisbehandlung, die problemspezifisch zu gewährleisten ist:

- ▶ als Ausnahme während der „normalen“ Programmausführung

Ausnahmebehandlung

Abrupter Zustandswechsel

Programmunterbrechungen implizieren **nicht-lokale Sprünge**:

- ▶ vom $\left\{ \begin{array}{l} \text{unterbrochenen} \\ \text{behandelnden} \end{array} \right\}$ zum $\left\{ \begin{array}{l} \text{behandelnden} \\ \text{unterbrochenen} \end{array} \right\}$ Programm

Sprünge (und Rückkehr davon), die **Kontextwechsel** nach sich ziehen:

- ▶ erfordert Maßnahmen zur Zustandssicherung/-wiederherstellung
- ▶ Mechanismen dazu liefert das behandelnde Programm selbst
 - ▶ bzw. eine tiefer liegende Systemebene (Betriebssystem, CPU)

☞ der **Prozessorstatus** unterbrochener Programme muss invariant sein

Zustandssicherung

Prozessorstatus invariant halten

Hardware (CPU) sichert einen Zustand minimaler Größe⁴

- ▶ Statusregister (SR)
- ▶ Befehlszeiger (engl. *program counter*, PC)

Software (Betriebssystem/Kompilierer) sichert den restlichen Zustand

- ▶ alle $\left\{ \begin{array}{l} \text{dann noch ungesicherten} \\ \text{flüchtigen} \\ \text{im weiteren Verlauf verwendeten} \end{array} \right\}$ CPU-Register

☞ je nach CPU werden dabei wenige bis sehr viele Daten(bytes) bewegt

⁴Möglicherweise aber auch den kompletten Registersatz.

Prozessorstatus sichern und wiederherstellen

Unabhängigkeit von der Sprachebene der Behandlungsprozedur

... alle dann noch ungesicherten CPU-Register:

Zeile

1:

2:

3:

4:

5:

x86

train:

pushal

call handler

popal

iret

m68k

train:

moveml d0-d7/a0-a6,a7@-

jsr handler

moveml a7@+,d0-d7/a0-a6

rte

train (trap/interrupt):

- ▶ Arbeitsregisterinhalte im RAM sichern (2) und wiederherstellen (4)
- ▶ Unterbrechungsbehandlung durchführen (3)
- ▶ Ausführung des unterbrochenen Programms wieder aufnehmen (5)

Prozessorstatus sichern und wiederherstellen (Forts.)

Abhängigkeit von der Sprachebene der Behandlungsprozedur

... alle **flüchtigen Register**⁵ (engl. *volatile register*) der CPU:

x86

```
train:
    pushl %edx
    pushl %ecx
    pushl %eax
    call  handler
    popl  %eax
    popl  %ecx
    popl  %edx
    iret
```

m68k

```
train:
    moveml d0-d1/a0-a1,a7@-
    jsr handler
    moveml a7@+,d0-d1/a0-a1
    rte
```

⁵Register, deren Inhalte nach Rückkehr von einem Prozeduraufruf verändert worden sein dürfen: festgelegt in den **Prozedurkonventionen** des Kompilierers.

Prozessorstatus sichern und wiederherstellen (Forts.)

Abhängigkeit von den Eigenschaften des Kompilers

... alle im weiteren Verlauf verwendeten CPU-Register:

```
gcc
void __attribute__((interrupt)) train () {
    handler();
}
```

`__attribute__((interrupt))`

- ▶ Generierung der speziellen Maschinenbefehle durch den **Kompilierer**
 - ▶ zur Sicherung/Wiederherstellung der Arbeitsregisterinhalte
 - ▶ zur Wiederaufnahme der Programmausführung
- ▶ nicht jeder „Prozessor“ (für C/C++) implementiert dieses Attribut

Aktivierungsblock (engl. *activation record*)

Sicherung/Wiederherstellung nicht-flüchtiger Register (engl. *non-volatile register*)

Türme von Hanoi

```
void hanoi (int n, char from, char to, char via) {
    if (n > 0) {
        hanoi(n - 1, from, via, to);
        printf("schleppe Scheibe %u von %c nach %c\n", n, from, to);
        hanoi(n - 1, via, to, from);
    }
}
```

Overhead je nach CPU, Prozedur und Kompilierer: `gcc -O6 -S hanoi.c`

hanoi()-Eintritt

```
pushl %ebp
movl  %esp,%ebp
pushl %edi
pushl %esi
pushl %ebx
subl  $12,%esp
```

hanoi()-Austritt

```
leal  -12(%ebp),%esp
popl  %ebx
popl  %esi
popl  %edi
popl  %ebp
ret
```

Für eine prozeduraufrufende Ebene **inhaltsinvariante Register** der CPU, deren Inhalte jedoch innerhalb einer aufgerufenen Prozedur verändert werden:

`gcc/x86` \mapsto `ebp, edi, esi, ebx`.

Verwaltungsgemeinkosten des schlimmsten Falls

(engl. *worst-case administrative overhead*, WCAO)

Latenz ... bis zum Start der Unterbrechungsbehandlung:

1. Annahme der Unterbrechung durch die Hardware
2. Sicherung der Inhalte der (flüchtigen) CPU-Register
3. Aufbau des Aktivierungsblocks der Behandlungsprozedur

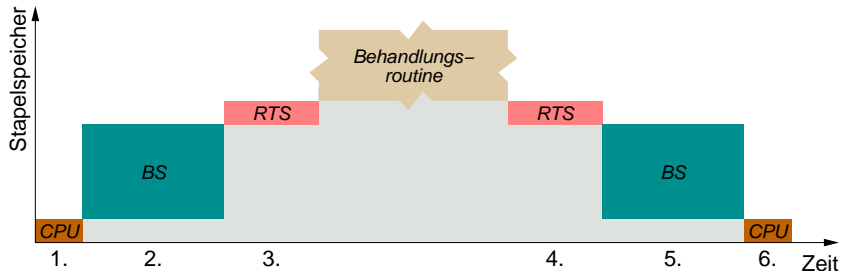
... bis zur Fortführung des unterbrochenen Programms:

4. Abbau des Aktivierungsblocks der Behandlungsprozedur
5. Wiederherstellung der Inhalte der (flüchtigen) CPU-Register
6. Beendigung der Unterbrechung

☞ **Zeitpunkte** und **Häufigkeit** der Gemeinkosten sind i.a. unbestimmbar

Verwaltungsgemeinkosten des schlimmsten Falls (Forts.)

Speicherplatz vs. Laufzeit



Systemkonstanten bzw. **Werte mit fester oberer Schranke** sind gefordert:

- ▶ CPU resp. Hardware
- ▶ Betriebssystem (BS), Laufzeitsystem (engl. *run-time system*, RTS)
- ▶ „Anwendung“ (Behandlungsroutine)

Unvorhersagbarkeit asynchroner Ereignisse

Gefahr von Fristverletzungen

Interrupts verursachen Last und erzeugen Nebenläufigkeit ...

- ▶ zu unvorhersagbaren Zeitpunkten
- ▶ in unvorhersagbaren Abständen

Synchronisation ist „einfach“, Fristen garantiert einhalten eher nicht:

- ▶ angenommen, ein Arbeitsauftrag dauert $90 \mu\text{s}$ und muss nach erfolgter Einlastung spätestens in $100 \mu\text{s}$ erledigt sein:
 - ▶ nach $17 \mu\text{s}$ wird er von einem *Interrupt* unterbrochen
 - ▶ die Unterbrechungsbehandlung dauert $42 \mu\text{s}$
 - ▶ bei Fortsetzung hat der Job noch $90 - 17 = 73 \mu\text{s}$ Arbeit vor sich
 - ▶ es bleiben jedoch nur noch $100 - 42 - 17 = 41 \mu\text{s}$ bis zum Fristablauf
- ▶ der Arbeitsauftrag hält seine Frist ein oder nicht, je nachdem, ob und wie lange seine Ausführung unterbrechungsbedingt verzögert wird

☞ (zer)störend: die **Scheinunterbrechung** (engl. *spurious interrupt*)

Apollo 11, Mondlandung

Folklore zum Bordcomputer der Landefähre [12, 13]

- ▶ das Rendezvousradar⁶ wurde vor Beginn der Landung eingeschaltet
 - ▶ falsche Vorgabe der Checkliste an die Astronauten †
- ▶ dadurch beanspruchte das Radarsteuerprogramm zuviel Rechenzeit
 - ▶ Netzteile von Radar und Landeeinheit waren nicht synchronisiert
 - ▶ das Rendezvousradar erzeugte so eine Flut von **Scheinunterbrechungen**
 - ▶ unerwarteterweise wurden dadurch etwa 15 % an Rechenlast erzeugt
 - ▶ Folge: Verzögerung/Ausfall von Berechnungen zur Landungskontrolle
- ▶ die Landungskontrolle hatte minimalen Treibstoffverbrauch als Ziel
 - ▶ das Kontrollprogramm setzte alle zwei Sekunden ein Kommando ab
 - ▶ zur Stabilisierung wurde der Autopilot jede 1/10 Sekunde aktiv
- ▶ die Landephase war mit einer Dauer von 11 Minuten geplant
 - ▶ fehlerbedingt fielen gut eine Minute lang alle Kontrollkommandos aus
 - ▶ dennoch klappte die Landung: Umschaltung auf manuelle Kontrolle

⁶Zur Messung von Zeitintervallen zwischen bekannten Landmarken und Überprüfung von Position und Geschwindigkeit des Landemoduls relativ zum Kommandomodul.

Maxime von Echtzeitsystemen: Unterlast

Kapazitäten gezielt frei lassen ...

- ▶ Echtzeitsysteme dürfen in kritischen Situation nur bis zu einem vorgegebenen Maximum belastet werden
 - ▶ das deutlich unter 100 % liegt
- ▶ solche Situationen zu identifizieren, zu bewerten und freizuhalten Kapazitäten zu bestimmen, ist eine große Herausforderung
 - ▶ die durchgehende *Anforderungsanalyse*⁷ zwingend macht
- ▶ unterbrechungsbedingte Verzögerungen und Last im Voraus einzuplanen, benötigt eine ordentliche Portion von Expertenwissen
 - ▶ das auch Scheinunterbrechungen kaum vorhersagen kann

⁷(engl. *requirements engineering*), zugleich Fundament und Teilaktivität systematischer Softwareentwicklung — hier aber nicht nur Software.

Erkennung asynchroner Programmunterbrechungen

Flanke kontra Pegel

flankengesteuert (engl. *edge triggered*) **Taktflanke**

- ▶ zweiphasige Erkennung der *Interrupts* durch die CPU:
 1. Pegelwechsel⁸ im **Auffangregister** (engl. *latch*) zwischenspeichern
 2. auf Zustandsänderung am Ende des laufenden Befehls prüfen
- ▶ ggf. implizites Löschen der Quelle nach erkannter Unterbrechung
 - ▶ eine Funktion der Hardware \mapsto PIC bzw. Gerät

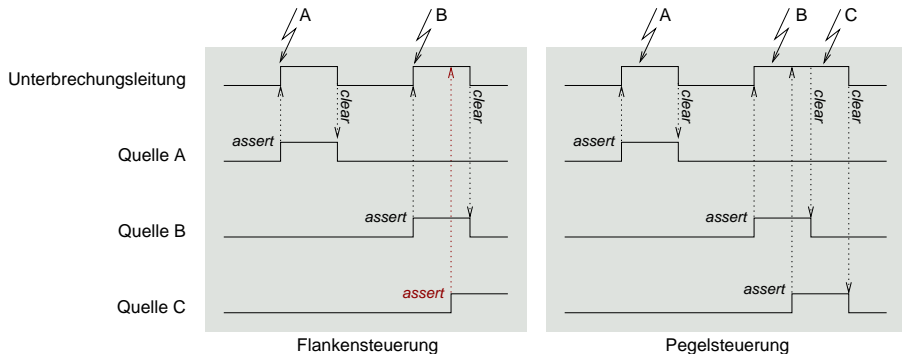
pegelgesteuert (engl. *level triggered*) **Logikpegel** zwischen den Flanken

- ▶ zyklische Zustandsabfrage der Unterbrechungsleitung (engl. *pin*)
 - ▶ taktweise oder am Ende des laufenden Befehls
 - ▶ Unterbrechung erfolgt je nach definierter Logikebene (0 oder 1)
- ▶ explizites Löschen der Quelle bei der Unterbrechungsbehandlung
 - ▶ eine Funktion der Software \mapsto Behandlungsroutine

⁸Von logisch „0“ (engl. *low*) zu logisch „1“ (engl. *high*) oder umgekehrt.

Flanken- vs. Pegelsteuerung

Gemeinsame Benutzung derselben Unterbrechungsleitung



Wiederbehauptung (engl. *reassertion*) einer Unterbrechung über eine mehreren Quellen gemeinsame Unterbrechungsleitung

- ▶ ggf. bleiben flankengesteuerte Unterbrechungen unerkant
- ▶ unerkantete Quellen werden keine neuen Unterbrechungen melden
 - ▶ ein **Aufhänger** (engl. *hangup*) im System kann die Folge sein

Flanken- vs. Pegelsteuerung (Forts.)

Querschneidender Belang der Unterbrechungsbehandlung

Quelle zur Zurücknahme des Unterbrechungssignals veranlassen

Flankensteuerung \mapsto nachfolgende Flanken ermöglichen

- ▶ implizit bei oder direkt nach der Erkennung
- ▶ explizit bei der Unterbrechungsbehandlung
 - ▶ EOI (engl. *end of interrupt*) als erste Anweisung

Pegelsteuerung \mapsto permanente Unterbrechung vorbeugen

- ▶ explizit durch Geräteprogrammierung
 - ▶ auf Anweisung des jeweiligen Gerätetreibers

Anweisungsfolgen, die in Abhängigkeit vom Unterbrechungsansatz ...

- ▶ nicht in Software implementiert werden müssen
- ▶ noch vor Start eines Gerätetreibers auszuführen sind
- ▶ quer über die Gerätetreiber verstreut vorliegen

Arten asynchroner Programmunterbrechungen

Von einer CPU bedingt oder unbedingt anzunehmende Unterbrechungen

maskierbare Unterbrechung (engl. *maskable interrupt*)

- ▶ auch: **Unterbrechungsanforderung** (engl. *interrupt request*, IRQ)
- ▶ Ab-/Anschalten möglich durch Spezialbefehle der CPU
 - ▶ privilegierte Befehle, je nach CPU
- ▶ die CPU dadurch anweisen, einen IRQ zu ignorieren/beachten
 - ▶ den Unterbrecher in der CPU steuern

nicht-maskierbare Unterbrechung (engl. *non-maskable interrupt*, NMI)

- ▶ Ab-/Anschalten ggf. möglich durch Funktionen des Gerätetreibers
 - ▶ sofern das Gerät die Programmierung der Unterbrechung zulässt
- ▶ so ggf. die Signalisierung der Unterbrechung unterbinden können
 - ▶ ein an der CPU angeschlossenes Gerät steuern

Abwehren/Zulassen von Unterbrechungsanforderungen

Einseitige („harte“) Synchronisation

x86

```
inline void avertIRQ () {  
    asm("cli");  
}
```

```
inline void admitIRQ () {  
    asm("sti");  
}
```

m68k

```
inline void avertIRQ () {  
    asm("or #$0700,sr");  
}
```

```
inline void admitIRQ () {  
    asm("and #$f8ff,sr");  
}
```

Flankensteuerung \rightsquigarrow Unterbrechungsanforderungen können „verpuffen“

- ▶ wenn zum IRQ-Zeitpunkt die CPU hart synchronisiert arbeitet⁹

⁹Die CPU müsste jeden verpassten IRQ zählen, um bei Wiedenzulassung nachträglich die Behandlung(en) starten zu können. Aber welche CPU kann das schon ...

Kaskadierte Unterbrechung

Unterbrechungen von Programmen zur Unterbrechungsbehandlung

Programme zur Unterbrechungsbehandlung sind grundsätzlich mit der Tatsache konfrontiert, selbst unterbrochen werden zu können¹⁰

- ▶ wenn Behandlungsroutinen virtuelle Adressen verwenden \mapsto *Trap*
 - ▶ ggf. bei E/A oder der Behandlung von Segment-/Seitenfehlern
- ▶ wenn **höher priorisierte Ereignisse** auftreten \mapsto *Interrupt*
 - ▶ ein NMI unterbricht einen NMI oder einen IRQ
 - ▶ ein IRQ höherer Priorität unterbricht einen IRQ niedrigerer Priorität

Unterbrechungslatenzen erhöhen sich ggf. **nicht deterministisch**

- ▶ der WCAO multipliziert sich mit der Anzahl der Prioritätsebenen
 - ▶ Parameter der Hardware/Software des Echtzeitrechensystems ✓
- ▶ weitere Faktoren sind Frequenz und Zeitabstand der Unterbrechungen
 - ▶ Parameter der **Umgebung** des Echtzeitrechensystems ?

¹⁰ Programmierfehler nicht betrachtet.

Resümee

Einplanungseinheit \mapsto Prozedur, Faden und/oder Fadengruppe

- ▶ Aufgaben (*Tasks*) von Arbeitsaufträgen (*Jobs*)
- ▶ Verwaltungsgemeinkosten ein- und mehrfädiger Ausgaben
- ▶ verdrängbare und/oder nicht-verdrängbare Prozessinstanzen

Programmunterbrechung in synchroner oder asynchroner Ausprägung

- ▶ Zustandssicherung, Verwaltungsgemeinkosten des schlimmsten Falls
- ▶ *Interrupts* machen determinierte Programme nicht-deterministisch
 - ▶ nicht zu jedem Zeitpunkt ist bestimmt, wie weitergefahren wird
- ▶ Unvorhersagbarkeit, Überlast, Verzögerung, . . . , Nebenläufigkeit

Unterbrechungstechnik \mapsto Pegelsteuerung & Flankensteuerung

- ▶ Problem der Wiederbehauptung flankengesteuerter *Interrupts*
- ▶ maskierbare und nicht-maskierbare Unterbrechungen
- ▶ kaskadierbare bzw. kaskadierte Unterbrechungen