Echtzeitsysteme

Grundlagen

27. Oktober/3. November 2008

Überblick

Fragestellungen

- Echtzeitrechensysteme nehmen Arbeitsaufträge entgegen und bearbeiten die an sie gestellten Aufgaben fristgemäß
  - was ist ein Arbeitsauftrag?
  - was ist eine Aufgabe?
- bei Ausführung von Arbeitsaufträgen darf die Einhaltung von Fristen trotz evtl. Programmunterbrechungen nicht verletzt werden
  - was ist eine Programmunterbrechung?
  - was sind in dem Zusammenhang die Verwaltungsgemeinkosten?
- die Erkennung und Behandlung von Programmunterbrechungen hat großen Einfluss auf die Robustheit von Rechensystemen
  - 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 Programmfaden (engl. *thread*)
- der Kontext eines Fadens manifestiert sich im Prozessorstatus
  - physisch die Inhalte der Arbeits- und Statusregister der CPU
  - ggf. erweitert um Segment-/Seitendeskriptoren von MMU bzw. TLB
- Einlastung (engl. *dispatching*) eines Fadens bedeutet Kontextwechsel

Prosessinstanz

Ein- oder mehrfädiges Programm, mit oder ohne eigenem Adressraum — oder aber bloß „Prozeduraktivierung“ eines übergeordneten Programms, 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]

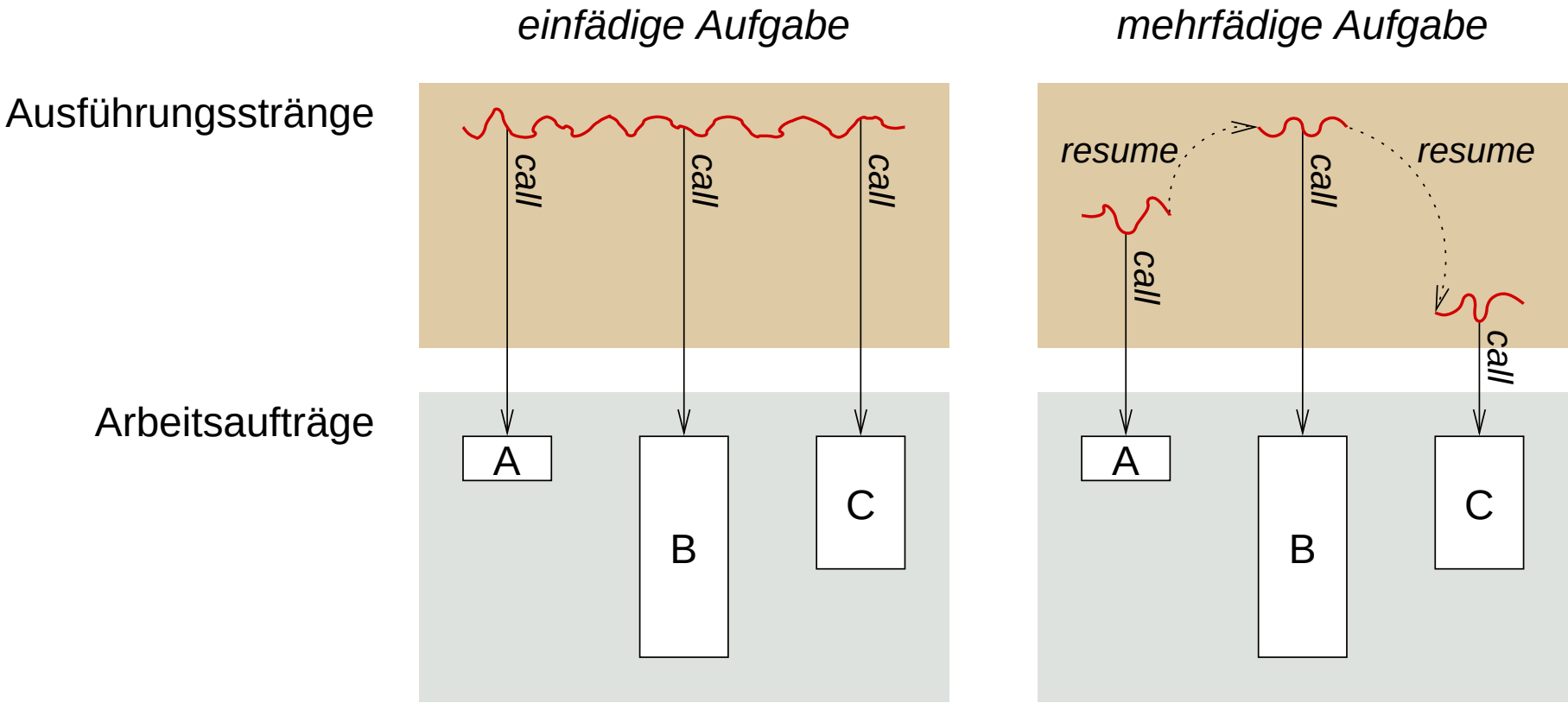
[a] Nach [1, S. 75] müssten die Arbeitsaufträge einer Aufgabe strikt nacheinander ablaufen, damit der sequentielle Charakter der Aufgabe gewahrt ist. In [2] entsteht aber ein anderes Bild: Arbeitsaufträge werden ggf. nebenläufig ausgeführt.

Einplanungseinheit 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



- 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 Ablauftabelle (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*)

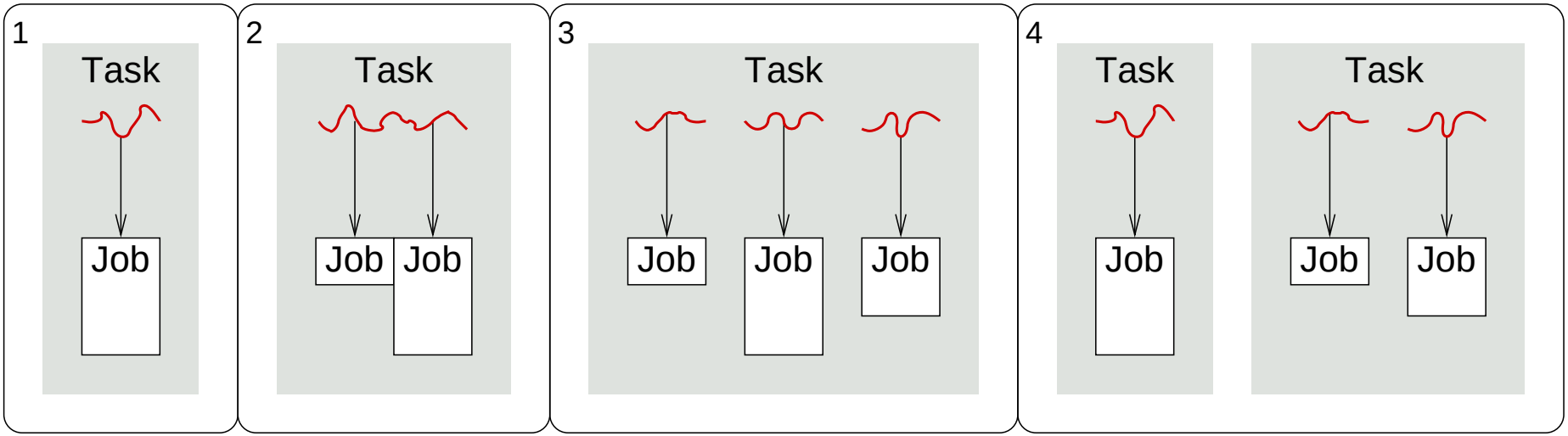
- die Architektur des Systems
  - prozedur-, objekt-, prozessorientiert
- die Performanz des Systems
  - Geschwindigkeit, Energie, Speicher
- ...

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 Echtzeitbedinungen !!!

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
   - $O(\mathit{Prozeduraufruf})$
   - Auf- und Abbau vom Aktivierungsblock (engl. *activation record*)
2. mehrfädige Aufgabe
   - 2.1 gemeinsamer Adressraum $\rightsquigarrow$ federgewichtig
     - $O(\mathit{Fadenwechsel}) + O(1.)$
     - Austausch der Inhalte von Arbeits-/Statusregister der CPU
   - 2.2 separierter Betriebssystemkern $\rightsquigarrow$ leichtgewichtig
     - $O(\mathit{Systemaufruf}) + O(2.1.)$
     - Behandlung der synchronen Programmunterbrechung (engl. *trap*)
   - 2.3 getrennte Adressräume $\rightsquigarrow$ schwergewichtig
     - $O(\mathit{Adressraumwechsel}) + O(2.2.)$
     - 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 ablaufen, können Verdrängung (engl. *preemption*) erleiden

- dem Faden des laufenden Arbeitsauftrags wird die CPU entzogen:
  1. eine asynchrone Programmunterbrechung (engl. *interrupt*) tritt auf
  2. der unterbrochene Faden wird als laufbereit (erneut) eingeplant
  3. ein anderer laufbereiter 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 laufbereiter 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
   - fordert 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
- Reproduzierbarbeit

☞ 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

- tritt unabhängig vom Arbeitszustand des laufenden Programms auf:
  - 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:[1]

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

[1] Dies gilt 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 { unterbrochenen / behandelnden } zum { behandelnden / unterbrochenen } 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 Größe[2]

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

Software (Betriebssystem/Kompilierer) sichert den restlichen Zustand

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

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

[2] Möglicherweise aber auch den kompletten Registersatz.

Prozessorstatus sichern und wiederherstellen

Unabhängigkeit von der Sprachebene der Behandlungsprozedur

. . . alle dann noch ungesicherten CPU-Register:

Zeile	x86	m68k
1:	`train:`	`train:`
2:	`  pushal`	`  moveml d0-d7/a0-a6,a7@-`
3:	`  call handler`	`  jsr handler`
4:	`  popal`	`  moveml a7@+,d0-d7/a0-a6`
5:	`  iret`	`  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[3] (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
```

[3] Register, 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 Kompilierers

. . . 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);
    }
}
```

Aufwand je nach CPU, Prozedur, 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 Prozedur aufrufende Ebene inhaltsinvariante Register der CPU, deren Inhalte jedoch innerhalb einer aufgerufenen Prozedur verändert werden:

gcc/x86 $\sim$ `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

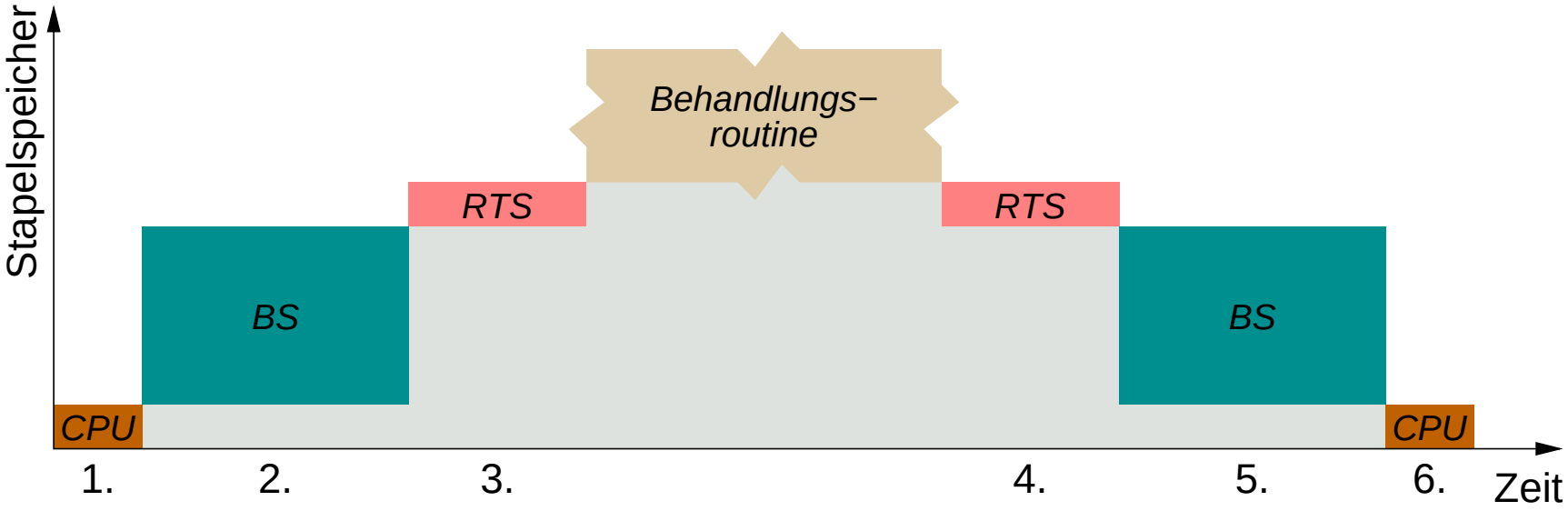
. . . bis zur Fortführung des unterbrochenen Programms:

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 μs und muss nach erfolgter Einlastung spätestens in 100 μs erledigt sein:
  - nach 17 μs wird er von einem *Interrupt* unterbrochen
  - die Unterbrechungsbehandlung dauert 42 μs
  - bei Fortsetzung hat der Job noch $90 - 17 = 73\,\mu$s Arbeit vor sich
  - es bleiben jedoch nur noch $100 - 42 - 17 = 41\,\mu$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 [3, 4]

- das Rendezvousradar[4] 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 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

[4] Messung von Zeitintervallen zwischen bekannten Landmarken und Überprüfung von Position und Geschwindigkeit des Landemoduls relativ zum Kommandomodul.

Maxime von Echtzeitrechensystemen: Unterlast

Kapazitäten gezielt frei lassen . . .

- Echtzeitrechensysteme dürfen in kritischen Situation nur bis zu einem vorgegebenen Maximum belastet werden
  - das deutlich unter 100 % liegt
- solche Situationen zu identifizieren, zu bewerten und freizuhaltende Kapazitäten zu bestimmen, ist eine große Herausforderung
  - die durchgehende Anforderungsanalyse[5] zwingend macht
- unterbrechungsbedingte Verzögerungen und Last im Voraus einzuplanen, benötigt eine ordentliche Portion von Expertenwissen
  - das auch Scheinunterbrechungen kaum vorhersagen kann

[5] engl. *requirements engineering*, 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. Pegelwechsel[6] 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

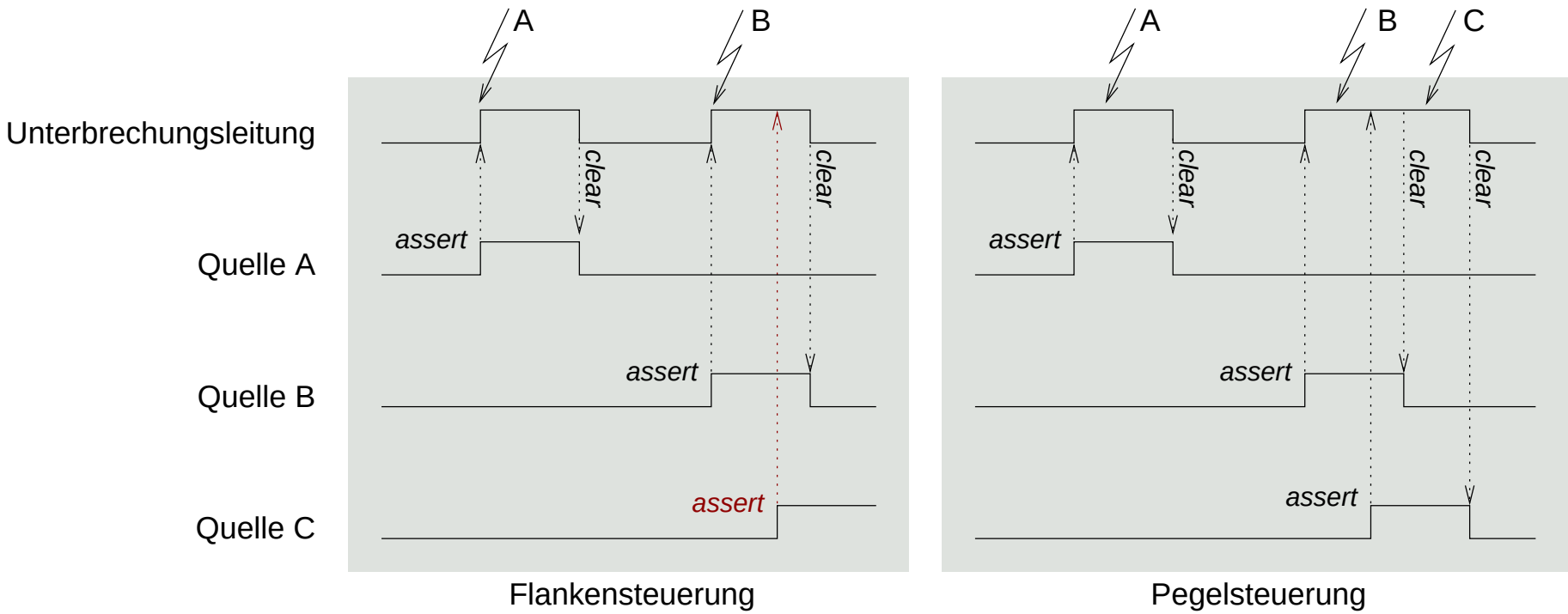
pegelgesteuert (engl. *level triggered*) Logikpegel zwischen den Flanken

- 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

[6] Von 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 unerkannt
- unerkannte 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 $\leadsto$ Unterbrechungsanforderungen können „verpuffen“

- wenn zum IRQ-Zeitpunkt die CPU hart synchronisiert arbeitet[7]

[7] Die CPU müsste jeden IRQ zählen, der verpasst wurde, um bei Wiederzulassung nachträglich 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 können[8]

- 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: Frequenz und Zeitabstand der Unterbrechungen
  - Parameter der Umgebung des Echtzeitrechensystems ?

[8] Programmierfehler nicht betrachtet.

Resümee

Einplanungseinheit $\mapsto$ Prozedur, Faden und/oder Fadengruppe

- Aufgaben (*Tasks*) von Arbeitsaufträgen (*Jobs*)
- Verwaltungsgemeinkosten ein- und mehrfädiger Ausfgaben
- 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

Literaturverzeichnis

[1] Hermann Kopetz.
Real-Time Systems: Design Principles for Distributed Embedded Applications.
Kluwer Academic Publishers, 1997.

[2] Jane W. S. Liu.
Real-Time Systems.
Prentice-Hall, Inc., 2000.

[3] Stanley R. Mohler Jr.
My fascinating interview with Allan Klumpp.
http://www.unt.edu/UNT/departments/CC/Benchmarks/benchmark
1995.

Literaturverzeichnis (Forts.)

[4] Roger Zühlsdorf.
Protokoll des Funkverkehrs bei der ersten Landung auf dem Mond.
http://members.fortunecity.de/rogerzuehlsdorf/Ap11d.htm, 1999.