— BS //—

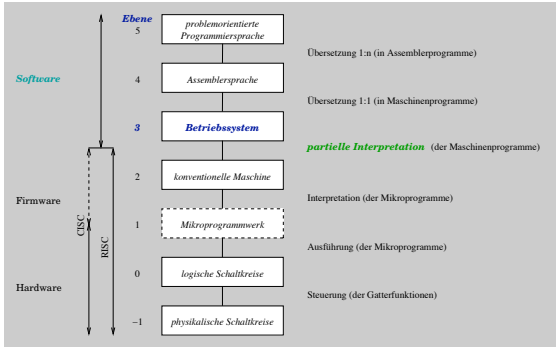
Programmunterbrechungen

Betriebssysteme, ©Wolfgang Schröder-Preikschat

Überblick

- Software/Hardware-Hierarchie, partielle Interpretation 2
- Unterbrechungsarten 6
  - synchrone Programmunterbrechung (*trap*) 9
  - asynchrone Programmunterbrechung (*interrupt*) 10
- Zustandssicherung/-wiederherstellung 15
- Unterbrechungsbehandlung und -latenz, Transparenz 24
- Benutztbeziehung 32

Hardware/Software-Hierarchie [4]



Partielle Interpretation

- die „Betriebssystemmaschine“ ist Ebene $_3$ der Hardware/Software-Hierarchie
  - Ebene $_2$ ist die konventionelle Maschinenebene (CPU)
  - Ebene $_4$ ist die Assemblerebene
- Ebene $_3$-Befehle setzen sich aus zwei Gruppen zusammen:
  1. Systemaufrufe, interpretiert vom abstrakten Prozessor Betriebssystem
     - d.h., repräsentiert als Folge von Ebene $_2$-Befehlen
  2. Ebene $_2$-Befehle, direkt ausgeführt vom realen Prozessor CPU
- ausführende Instanz im Rechner ist immer der Ebene $_2$-Prozessor (die CPU)

Schnappschuss der Programmausführung

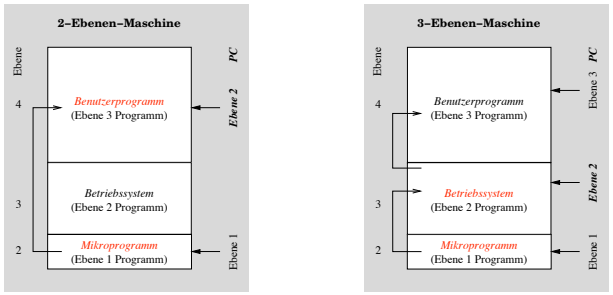
- je nach Arbeitsmodus funktioniert ein Rechner als 2- oder 3-Ebenen-Maschine:

  2-Ebenen-Maschine Die Ebene$_2$-Befehle des Benutzerprogramms werden vom Ebene$_2$-Prozessor (CPU) ausgeführt. Der Rechner arbeitet im nicht-privilegierten Benutzermodus (*user mode*).

  3-Ebenen-Maschine Die Ebene$_2$-Befehle der Betriebssystemprogramme werden vom Ebene$_2$-Prozessor (CPU) ausgeführt. Die Betriebssystemprogramme implementieren die Systemaufrufe (*system calls*) und dienen der Unterbrechungsbehandlung. Der Rechner arbeitet im privilegierten Systemmodus (*system mode*).
  Eine Programmunterbrechung (*trap*/*interrupt*) bzw. ein spezieller Ebene$_2$-Befehl (z.B. `int` oder `trap`) schaltet den Rechner in den Systemmodus um. Ebenso bewirkt ein spezieller Ebene$_2$-Befehl (z.B. `iret` oder `rte`) die Umschaltung zurück in den Benutzermodus.

- Ebene$_3$ (d.h., das Betriebssystem) ist immer nur „ausnahmsweise“ aktiv

Interpreter „Betriebssystem“



Unterbrechungsarten

- es werden zwei Kategorien von Programmunterbrechungen unterschieden:
  1. die „Falle“ — der **Trap**
  2. die „Unterbrechung“ — der **Interrupt**

- die sie auslösenden Ausnahmesituationen unterscheiden sich hinsichtlich . . .
  - Quelle
  - Synchronität
  - Vorhersagbarkeit
  - Reproduzierbarbeit

- softwaremäßige Behandlung ist zwingend und grundsätzlich prozessorabhängig[1]

[1] Egal ob abstrakter Prozessor (virtuelle Maschine) oder physikalischer Prozessor (reale Maschine).

Trap vs. *Interrupt*

Trap synchron, vorhersagbar, reproduzierbar . . . . . *kein Interrupt*

- unbekannter Befehl, falsche Adressierungsart, fehlerhafte Rechenoperation
- Adreßraumverletzung, unbekanntes Gerät (*double bus fault*)
- ⋮
- Seitenfehler im Falle lokaler Ersetzungsstrategien

Interrupt asynchron, unvorhersagbar, nicht reproduzierbar . . . . . *kein Trap*

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

Trap oder *Interrupt* ?

```
#include <stdlib.h>
...
foo = bar / random();
...
```


Synchrone Programmunterbrechung

- der **Trap** — ist vorhersagbar und reproduzierbar

  Ein in die Falle gelaufenes („ge*trap*tes“) Programm, das unverändert wiederholt und jedesmal mit den selben Eingabedaten versorgt auf ein und dem selben Prozessor zur Ausführung gebracht wird, wird auch immer wieder an der selben Stelle in die selbe Falle laufen, d.h. den selben Trap verursachen.

- ohne Behebung der Ausnahmesituation ist eine Trap-Vermeidung unmöglich

Asynchrone Programmunterbrechung

- der **Interrupt** — ist unvorhersagbar und nicht reproduzierbar

  Auch wenn manche Geräte (z.B. Zeitgeber) die Interrupts in zyklischen Abständen (re-)produzieren können, ist der exakte Zeitpunkt und demzufolge auch die genaue Stelle der Unterbrechung (d.h. der davon betroffene Maschinenbefehl) nicht vorhersagbar. Interrupts sind jedoch vorhersehbar in dem Sinne, dass mit ihrem Auftreten gerechnet werden kann.

- die Behandlung der Ausnahmesituation muss Nebeneffekt frei verlaufen

Ausnahmesituationen

sind nicht immer eintretende und erwünschte Ereignisse . . .

- 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)
- Warnsignale von der Hardware . . . . . (z.B. Energiemangel)
- Seitenfehler (*page fault*) . . . . . [warum?]

. . . die jedoch zu erwarten sind und behandelt werden müssen

- die Behandlung ist sehr problemspezifisch — und schließt Ignoranz mit ein

Ausnahme — *Exception*

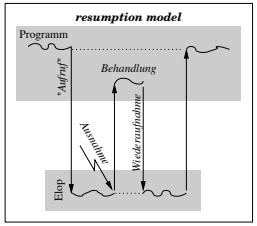
- aus Verursachersicht sind grundlegend zwei Konzepte zu unterscheiden [1]:

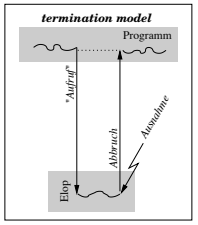
  resumption model Die erfolgreiche Behandlung der Ausnahmesituation führt zur Wiederaufnahme der Ausführung des unterbrochenen Programms. Ein Trap kann, ein Interrupt muss nach diesem Modell behandelt werden.

  termination model Konnte (oder sollte) die Ausnahmesituation nicht behandelt werden, wird ein schwerwiegender Fehler konstatiert, der zum Abbruch des unterbrochenen Programms führen muss. Ein Trap kann, ein Interrupt darf niemals nach diesem Modell behandelt werden.

- hervorrufen (*raising*) einer Ausnahme impliziert einen **Kontextwechsel**

Ausnahmebehandlung — *Exception Handling*





Elop — *El*ementar*op*eration eines (abstrakten) Prozessors/einer (virtuellen) Maschine: der Maschinenbefehl der CPU, eine Betriebssystemfunktion (aktiviert via *system call*), ein Unterprogramm (aktiviert via Prozeduraufruf).

Kontextwechsel

- die Unterbrechungen ziehen nicht-lokale Sprünge nach sich . . .

  vom { unterbrochenen / behandelnden } Programm zum { behandelnden / unterbrochenen } Programm

- der **Prozessorstatus** des unterbrochenen Programms gilt dabei als Invariante
  - dies erfordert Maßnahmen zur Sicherung/Wiederherstellung des Kontextes
  - die Mechanismen liefert das behandelnde Programm bzw. eine tiefere Ebene

- ein „tieferer“ realer und/oder abstrakter Prozessor hält den Status konsistent

Prozessorstatus

- die CPU („realer Prozessor“) sichert einen Zustand minimaler Größe
  - typischerweise nur SR und PC, evtl. aber auch den kompletten Registersatz
  - je nach CPU werden dabei sehr wenige bis sehr viele Daten(bytes) bewegt

- Systemsoftware („abstrakter Prozessor“) sichert ggf. den restlichen Zustand
  - hierzu stehen zwei grundsätzliche, alternative Ansätze zur Verfügung:
    1. Sicherung aller bisher noch nicht automatisch gesicherten CPU-Register
    2. Sicherung aller im weiteren Verlauf nicht gesicherten CPU-Register
  - die Varianten resultieren aus unterschiedlichen Betriebssystemkonzepten

- die Sicherungsmaßnahmen sind höchst abhängig vom jeweiligen Prozessor

Zustandssicherungskonzepte

totale Sicherung aller bislang nicht automatisch gesicherten Register

- der CPU-Status des unterbrochenen Programms wird vollständig eingefroren
  - auch die invarianten Anteile werden gesichert −
  - der Programmzustand ist damit leicht „zugreifbar“ +
- weit verbreitet bei Allzweckbetriebssystemen (z.B. UNIX & Co.)

partielle Sicherung der im weiteren Verlauf nicht gesicherten Register

- der CPU-Status des unterbrochenen Programms wird teilweise eingefroren
  - es wird nur der wirklich von Änderungen betroffene Anteil gesichert +
  - der Programmzustand ist damit nicht leicht „zugreifbar“ −
- weit verbreitet bei Spezialzweckbetriebssystemen

Partielle Sicherung

- gesichert werden zunächst nur die „flüchtigen CPU-Register“[2]
  - die Unterbrechungsbehandlung erfolgt in einer höheren Programmiersprache
  - die Sicherung weiterer Register übernimmt das Laufzeitsystem
  - der Programmzustand wird mehr oder weniger verstreut hinterlassen
- die Menge flüchtiger Register kann aber auch leer sein **!**
  - die Unterbrechungsbehandlung erfolgt durch „idempotente Operationen“

[2] Übersetzer teilen den CPU-Registersatz üblicherweise auf in „flüchtige“ (*volatile*) und „nicht-flüchtige“ (*non-volatile*) Register. Erstere sind jene, deren Inhalte über die Schnittstellen zu Unterprogrammen hinweg verändert werden dürfen und die Rückgabewerte enthalten können. Letztere sind jene, deren Inhalte über diese Schnittstellen hinweg nicht verändert werden dürfen und die demzufolge in den Unterprogrammen bei Bedarf automatisch gesichert werden müssen.

Aufbewahrung des CPU-Status

- manche Prozessoren unterscheiden zwischen verschiedenen **Arbeitsmodi**[3]
  - nicht privilegierter *user mode*, privilegierter *system* und *interrupt mode*
  - die Unterbrechung aktiviert (ggf.) den privilegierten Arbeitsmodus
- die CPU-Registerinhalte werden im **Stapelspeicher** (*stack*) gesichert
  - genauer: der Stack des mit der Unterbrechung aktivierten Arbeitsmodus
  - aus logischer Sicht ist dies immer der Stack des privilegierten Arbeitsmodus
- replizierte, modusbezogene Registersätze ersetzen den Stack nicht [warum?]

[3] Je nach Technologie: bei 32-Bit und drüber gilt die Differenzierung als „Stand der Kunst“, bei 16-Bit ist sie nur sehr vereinzelt (bei „Exoten“) anzutreffen, ab 8-Bit und drunter gibt es sie nicht.

Sicherung und Wiederherstellung

- ein Fall für die maschinenabhängige Programmierung in **Assembler**

totale Sicherung:

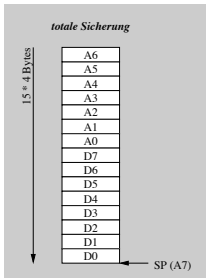
```
moveml d0-d7/a0-a6,sp@-
...
moveml sp@+,d0-d7/a0-a6
```

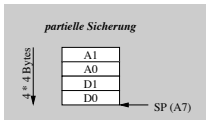
partielle Sicherung:

```
moveml d0-d1/a0-a1,sp@-
...
moveml sp@+,d0-d1/a0-a1
```

- Sicherung und Wiederherstellung geschieht durch *push*- und *pop*-Operationen
  - hier (m68k) ausgedrückt durch die Adressierungsart (*auto*-{*de*,*in*}*crement*)

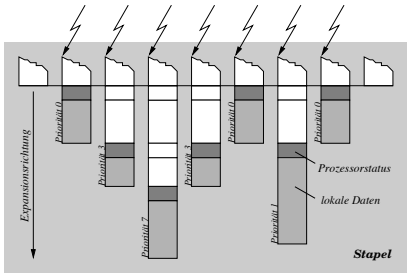
Totale vs. Partielle Sicherung





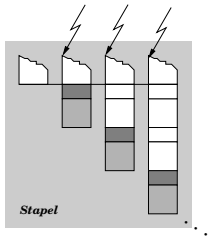
Nicht nur der Speicherbedarf ist je nach Prozessor sehr unterschiedlich, auch die Laufzeit (d.h. der Bedarf an Taktzyklen). Beim m68k erfordert die totale Sicherung/Wiederherstellung 11/11 (32-Bit) Speicherzugriffe mehr als die partielle Sicherung.

Dimensionierung des Stapelspeichers



Der Stapelspeicher (des Systems) muss entsprechend der tiefsten möglichen **Interrupt-Verschachtelung** dimensioniert sein. Die max. mögliche Verschachtelungstiefe richtet sich nach dem Interrupt-Modell der Hardware — und nach dem Modell der Interrupt-Behandlung des Betriebssystems.

Terminierung der Ausdehnung des Stapelspeichers



Die rekursive Aktivierung der Behandlungsroutinen ist mit sehr großer Vorsicht zu bewerkstelligen — und am besten überhaupt zu vermeiden! Es ist sicherzustellen, dass vor Wiederaufnahme der Ausführung des unterbrochenen Programms, (1) die Unterbrechungsursache beseitigt und (2) der Systemstapel vollständig abgebaut worden ist. Die Art der Signalisierung (*edge/level-triggered interrupt*) ist dabei von großer Bedeutung.

Wiederaufnahme der Programmausführung

- ein weiterer Fall für die maschinenabhängige Programmierung in **Assembler**
  - Unterbrechungen ändern den Arbeitsmodus der CPU (→ privilegierter Mode)
  - der vorige Arbeitsmodus ist nach erfolgter Behandlung zu reaktivieren
- eine spezielle, der Unterbrechung inverse Operation[4] ist auszuführen
  - um den „minimalen Prozessorstatus“ (→ p. 15) wieder herzustellen
    - d.h., um den „*exception stack frame*“ wieder abzubauen
  - um den zum Unterbrechungszeitpunkt gültigen Arbeitsmodus zu reaktivieren
- die partielle Interpretation privilegierten Programmtextes wird damit beendet

[4] Im Falle des m68k ist dies z.B. `rte` (*return from exception*).

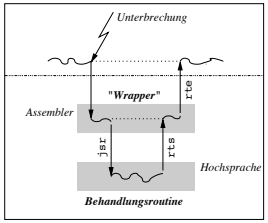
Systemprogrammiersprache

- Assemblerprogrammierung (→ p. 19 und p. 23) wäre vermeidbar, wenn . . .
  - die Unterbrechung einem Unterprogrammaufruf ähnlich verstanden wird √
  - die Unterbrechungsbehandlung als Unterprogramm formuliert wird √
  - die (höhere) Programmiersprache spezielle Ausdrucksmittel vorsieht !?

```
void handler () __attribute__ ((interrupt));
void handler () { /* ... */ }
```

  - der Übersetzer daraus die erforderlichen Maschinenbefehle erzeugt !?
- derartige (C/C++) Dialekte gelten nicht für alle Prozessoren — und Übersetzer

Unterbrechungsbehandlung



- „*separation of concerns*“:
  1. **Kopplungsroutine** . . . . . *Assembler*
     - Zustandssicherung/-wiederherstellung
     - Arbeitsmodereaktivierung
  2. **Behandlungsroutine** . . . . . *Hochsprache*
     - Behandlung der Ausnahmesituation
     - ggf. CPU-unabhängig
- die Maßnahme schlägt die „Brücke“ zwischen realem und abstraktem Prozessor

Kopplungsroutine

```
   .text tipWrapper
tipWrapper:
   movem! d0-d1/a0-a1,sp@-
   INVOKE(Handler,sp)
   movem! sp@+,d0-d1/a0-a1
   rte
```

tip — trap/interrupt propagation

- „Ummantelung“ der Behandlungsroutine
  - Prolog: Zustandssicherung
  - Epilog: Zustandswiederherstellung
  - Finale: Arbeitsmodereaktivierung
- generierbar für jede Sorte CPU
- zur Einbindung der Behandlungsroutine (Handler) bestehen mehrere Optionen
  - z.B. als Prozeduraufruf oder als Makroaufruf (*inline function*)
- sehr zweckmäßig ist zudem die Übergabe des Stapelzeigers (sp) [warum?]

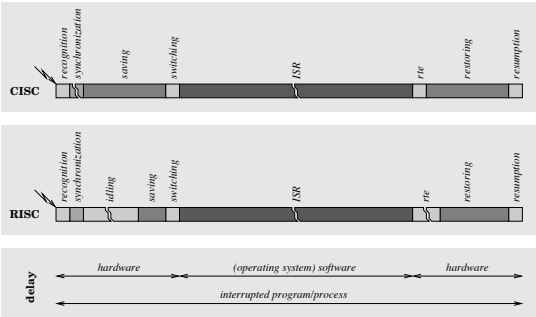
Unterbrechungslatenz — *Interrupt Latency*

- die Anzahl der Zyklen pro Maschinenbefehl variiert je nach Prozessortechnologie

  CISC (*complex instruction set computer*)
  - die Maschinenbefehle sind unterschiedlich lang in Zeit und Raum
  - die Zustandssicherung erfolgt hardware-seitig über den (System-) Stapel

  RISC (*reduced instruction set computer*)
  - die Maschinenbefehle sind gleich lang in Zeit und Raum
  - die Zustandssicherung erfolgt hardware-seitig über „Schattenregister“
    - Software entscheidet über die weitere Verwendung des (System-) Stapels
  - die *Pipeline* läuft vor der Unterbrechungsbehandlung leer !
- die CPU akzeptiert Unterbrechungen (i.d.R.) am Ende eines Maschinenbefehls

Interrupt Latency — CISC vs. RISC



Transparenz (1)

- Interrupts beeinflussen das Laufzeitverhalten der unterbrochenen Programme
  - Aussagen zur **Rechtzeitigkeit** sind unmöglich bzw. relativ unscharf

```
void ack (char* iop) {
    *iop |= 1;
}
```

Innerhalb welcher Zeit bzw. bis zu welchem Zeitpunkt kann z.B. diese Anweisung abgeschlossen werden? → p. 30

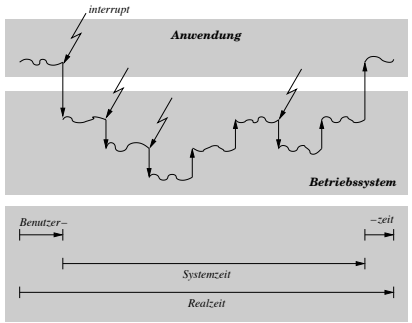
  - einfachste Befehlsfolgen können bei **Nebenläufigkeit** zum Problem werden
- Nichtdeterminismus stellt in Echtzeitszenarien eine große Hürde dar

Transparenz (2)

void ack (char* iop)			
x86		*sparc*	
gcc	*gcc -O6*	*gcc*	*gcc -O6*

movl 4(%esp),%eax
movl 4(%esp),%edx
movb (%edx),%cl
orb $1,%cl
movb %cl,(%eax)
ret
	movl 4(%esp),%eax
	orb $1,(%eax)
	ret
		save %sp,-112,%sp
		st %i0,[%sp+68]
		ld [%sp+68],%i0
		ldub [%i0],%i0
		or %i0,1,%i1
		ld [%sp+68],%i0
		stb %i1,[%i0]
		ret
		restore
			ldub [%o0],%o1
			or %o1,1,%g1
			retl
			stb %g1,[%o0]

Nichtdeterministisches Laufzeitverhalten



Last but not least: Benutztbeziehung [2]

- alle Programme eines Rechners *benutzen* die Unterbrechungsbehandlung [3]

  Die korrekte Durchführung der Unterbrechungsbehandlung ist zwingend dafür erforderlich, dass ein beliebiges Programm seine Aufgabe gemäß Spezifikation erfüllen kann. Die Korrektheit eines beliebigen Programms hängt ab von der Korrektheit der Unterbrechungsbehandlung.

- vor diesem Hintergrund ist „benutzen“ nicht zu verwechseln mit „aufrufen“

  Beispielsweise die Behandlungsroutine des Zeitgeber-Interrupts: sie wird von keinem Programm aufgerufen, jedoch von allen benutzt. Eine ggf. nicht korrekte Zustandssicherung wird direkte Auswirkungen auf die weitere Verarbeitung des unterbrochenen Programms haben.

- **Faustregel**: Unterbrechungsbehandlung so kurz und einfach wie möglich halten

Referenzen

[1] J. B. Goodenough. Exception Handling: Issues and a Proposed Notation. *Communications of the ACM*, 18(12):683–696, 1975.

[2] D. L. Parnas. Some Hypotheses About the “Uses” Hierarchy for Operating Systems. Technical Report BS I 75/2, TH Darmstadt, 1975.

[3] W. Schröder-Preikschat. Operating-System Engineering. http://www4.informatik.uni-erlangen.de, 2002.

[4] A. S. Tanenbaum. *Structured Computer Organization*. Prentice Hall, 1990.