Synchrone Programmunterbrechung

Trap — synchron, vorhersagbar, reproduzierbar

Ein in die Falle gelaufenes („ge*trap*ptes“) 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 tappen:

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

☞ Trapvermeidung ist ohne Behebung der Ausnahmebedingung unmöglich

Asynchrone Programmunterbrechung

Interrupt — asynchron, unvorhersagbar, nicht reproduzierbar

Ein „externer Prozess“ (z.B. ein Gerät) signalisiert einen Interrupt unabhängig vom Arbeitszustand des gegenwärtig sich in Ausführung befindlichen Programms. Ob und ggf. an welcher Stelle das betreffende Programm unterbrochen wird, ist nicht vorhersehbar:

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

☞ Ausnahmesituationsbehandlung muss nebeneffektfrei verlaufen

Trap oder Interrupt?

```
#include <stdlib.h>

float frandom () {
    return random()/random();
}
```

Division durch 0 ...

- Programmunterbrechung (je nach CPU)
- wird zufällig geschehen

Programmierfehler, der sich jedoch nicht zwingend auswirken muss:

- die Unterbrechung verläuft synchron zum Programmablauf . . . . . . Trap
- die Unterbrechungsstelle im Programm ist vorhersagbar . . . . Trap
- der Zufall macht die Unterbrechung nicht reproduzierbar . . . . Interrupt

Trap oder Interrupt? (Forts.)

```
extern edata, end;
int main () {
    char* p = (char*)&edata;
    do *p++ = 0;
    while (p != (char*)&end);
}
```

Indirekte Adressierung (*p++) ...

- Programmunterbrechung (je nach Systemauslastung)
- trotz korrektem Text

Seitenfehler (engl. *page fault*), vorbehaltlich eines virtuellen Speichers

- die Unterbrechung verläuft synchron zum Programmablauf . . . . . . Trap

Diskussionsstoff liefert die Ersetzungsstrategie (engl. *replacement policy*)

- lokal: Stelle vorhersagbar, Unterbrechung reproduzierbar . . . . Trap
- global: unvorhersagbar und nicht reproduzierbar . . . . Interrupt

Ausnahmesituationen sind Betriebssystemnormalität

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

Modelle zur Ausnahmebehandlung [15]

(engl. *exception handling*)

Wiederaufnahmemodell (engl. *resumption model*)

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

Beendigungsmodell (engl. *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.

☞ Auslösung (engl. *raising*) einer Ausnahme bedeutet Kontextwechsel

Ausnahmebehandlung bringt Kontextwechsel mit sich

Abrupter Zustandswechsel

Programmunterbrechungen implizieren nicht-lokale Sprünge:

$$\text{vom} \left\{ \begin{array}{c} \text{unterbrochenen} \\ \text{behandelnden} \end{array} \right\} \text{Programm zum} \left\{ \begin{array}{c} \text{behandelnden} \\ \text{unterbrochenen} \end{array} \right\} \text{Programm}$$

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

- erfordert Maßnahmen zur Zustandssicherung/-wiederherstellung
- Mechanismen liefert das behandelnde Programm/die tiefere Ebene

☞ der Prozessorstatus unterbrochener Programme muss invariant sein

Prozessorstatus invariant halten

Ebene $_2$ (CPU) sichert bei Ausnahmen einen Zustand minimaler Größe

- Statusregister (SR) und Befehlszeiger (engl. *program counter*, PC)
- möglicherweise aber auch den kompletten Registersatz
- je nach CPU werden dabei wenige bis sehr viele Daten(bytes) bewegt

Ebene $_{3/5}$ (Betriebssystem/Kompilierer) sichert den restlichen Zustand

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

☞ die zu ergreifenden Maßnahmen sind höchst prozessorabhängig

Prozessorstatus sichern und wiederherstellen

Prozessor „Betriebssystem“

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` (<u>trap</u>/<u>in</u>terrupt):

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

Prozessorstatus sichern und wiederherstellen (Forts.)

Prozessor „Kompilierer“

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

☞ „prozessorabhängig“ bedeutet nicht immer gleich „CPU-abhängig“ !!!

Unterbrechungen verzögern Programmabläufe

Problem für determinierte Programme. . .

- lassen bei ein und derselben Eingabe verschiedene Abläufe zu
- alle Abläufe liefern jedoch stets das gleiche Resultat

. . . da asynchrone Unterbrechungen sie nicht-deterministisch machen

- nicht zu jedem Zeitpunkt ist bestimmt, wie weitergefahren wird

☞ ist besonders kritisch für echtzeitabhängige Programme

Echtzeitfähigkeit

Unter Einbezug aller Last- und Fehlerbedingungen, kann ein sich in Ausführung befindliches Programm alle Zeit- und Terminvorgaben seiner Umgebung (weich, fest oder hart) einhalten.

Unterbrechungen erschweren Echtzeitprogrammierung

Unvorhersagbare Laufzeitvarianzen

weich (engl. *soft*) auch „schwach“

- Das Ergebnis einer zu einem vorgegebenen Termin nicht geleisteten Arbeit ist weiterhin von Nutzen.
- Terminverletzung ist tolerierbar.

fest (engl. *firm*) auch „stark“

- Das Ergebnis einer zu einem vorgegebenen Termin nicht geleisteten Arbeit ist wertlos und wird verworfen.
- Terminverletzung ist tolerierbar, führt zum Arbeitsabbruch.

hart (engl. *hard*) auch „strikt“

- Das Versäumnis eines fest vorgegebenen Termins kann eine „Katastrophe“ hervorrufen.
- Terminverletzung ist keinesfalls tolerierbar.

Asynchronität von Programmunterbrechungen

Nicht-deterministische Programme

Welche `wheel`-Werte gibt `main()` aus?

```
unsigned int wheel = 0;

void __attribute__ ((interrupt)) train () {
    wheel++;
}

int main () {
    for (;;)
        printf("%10u", wheel++);
}
```

0 1 2 4 . . . 13 13 14 15 16 . . . 4711 4711 4714 . . . ???

Teilbarkeit von Operationen

`wheel++` ist Elementaroperation (kurz: Elop) der Ebene $_5$. . .

- ein Prozessor führt seine Elop atomar, d.h. unteilbar aus

. . . jedoch nicht notwendigerweise auch eine der Ebene $_4$ (und tiefer)

`wheel++` in `main()`

```
movl wheel,%edx
incl %edx
movl %edx,wheel
```

`wheel++` in `train()`

```
movl wheel,%eax
incl %eax
movl %eax,wheel
```

☞ `train()` überlappt `main()` im Unterbrechungsfall !!!

Unterbrechungsbedingte Überlappungseffekte

Kritischer Programmtext

Nebenläufiges Zählen

main()		train()		wheel
x86-Befehl	%edx	*x86-Befehl*	%eax	
`movl wheel,%edx`	42			42
		`movl wheel,%eax`	42	42
		`incl %eax`	43	42
		`movl %eax,wheel`	43	43
`incl %edx`	43			43
`movl %edx,wheel`	43			43

☞ zweimal durchlaufen (`main()` und `train()`), aber nur einmal gezählt

Laufgefahr asynchroner Programmunterbrechungen

engl. *race hazard* (auch: *race condition*)

Dem Begriff liegt die Vorstellung zu Grunde, dass sich zwei (oder mehr) Signale in einem Wettlauf zueinander befinden, um als erstes die Ausgabe (ein Berechnungsergebnis) zu veranlassen:

- fehlerhafte Stelle in einem System, an der eine Berechnung eine unerwartet kritische Abhängigkeit vom relativen Zeitverlauf von Ereignissen zeigt
- potentielles Problem, sobald die Ausführung von Programmen nebenläufig (d.h. überlappend oder parallel) möglich ist

☞ kritischer Abschnitt eines nebenläufig ausgeführten Programms

Kritischen Abschnitt als Elementaroperation auslegen

Lösungsansatz (für den gegebenen Fall, Monoprozessoren):

- Schutz vor einer möglichen überlappenden Programmausführung
  - temporäres Abschalten asynchroner Programmunterbrechungen
  - „Synchronisationsklammern“ um den kritischen Abschnitt setzen
- auf Elop eines tieferen Prozessors (genauer: der CPU) abbilden
  - die bessere Lösung, sofern die CPU eine passende Elop dafür anbietet
  - ggf. praktikabel bei CISC (z.B. x86), nicht aber bei RISC (z.B. ppc)

Grundsätzlicher Lösungsansatz: Abstraktion

- einen kritischen Abschnitt als Modul abkapseln
- den modularisierten Programmtext passend synchronisieren

Kritischen Abschnitt als Elementaroperation auslegen (Forts.)

Modularisieren

```
int main () {
  for (;;)
    printf("%10u", incr(&wheel));
}
```

Unterbrechungsfrequenz

Schrittweiten größer als 1 bei der Ausgabe sind weiterhin möglich — und auch kein Fehler! Interrupts müssen nur in entsprechend kurzen Abständen auftreten.

Komplexbefehl verwenden

```
inline int incr (int* ip) {
  asm ("incl %0" : : "g" (*ip));
  return *ip;
}
```

Interrupts abschalten

```
inline int incr (int* ip) {
  asm ("cli"); *ip += 1; asm ("sti");
  return *ip;
}
```


Äquivalenz von Hardware und Software

Ebene $_2$-Befehle sind Teil der ISA, ihre Implementierungen nur bedingt

- Befehlssätze, sind optional in Hardware oder Software implementiert
  - Koprozessor (z.B. *floating-point unit*, FPU)
  - rekonfigurierbare Hardware (z.B. *field-programmaable array* FPGA)
- die Festlegung der wirklichen Implementierungsebene erfolgt später

Ebene $_2$-Befehle können durch Ebene $_2$-Programme *emuliert* werden

Allgemein gilt:

- Ebene $_i$-Befehl kann durch Ebene $_j$-Programm emuliert werden
- Ebene $_i$-Programm kann durch Ebene $_j$-Befehl implementiert werden

Emulation

Spezialfall der Simulation, bei dem das Verhalten einer Maschine durch eine andere Maschine vollständig nachgebildet wird:

- die Nachahmung der Eigenschaften eines ggf. anderen Rechnersystems
- bei Selbstvirtualisierung emuliert sich ein Rechnersystem selbst

Ein Emulator interpretiert die von der realen Maschine (die CPU) nicht ausführbaren Befehle:

- die betreffenden Befehle sind der (realen) Maschine bekannt
- sie müssen jedoch nicht zwingend auch in ihr implementiert sein

Extremfall der Emulation, z.B., Virtual PC für Apple-Rechner

- x86 (Ebene $_2$) wird auf PowerPC-Basis (Ebene $_2$) nachgebildet
- Ebene $_3$-Programm (MacOS/PowerPC) simuliert einen x86

Selbstvirtualisierung durch Teilinterpretation

Voraussetzung ist, die CPU „*trappt*“ privilegierte Befehle im nicht-privilegierten Arbeitsmodus

- beim x86 z.B. Befehle
  - zur Ein-/Ausgabe (`in`, `out`)
  - zur Synchronisation (`cli`, `sti`)
  - ⋮
  - zum Moduswechsel (`int`, `iret`)

Arbeitsmodi (einer CPU — jedoch nicht jeder)

- im privilegierten Modus läuft nur das Betriebssystem
  - ggf. auch nur ein Minimalkern (*virtual machine monitor*, VMM) davon
- im nicht-privilegierten Modus laufen alle anderen Programme

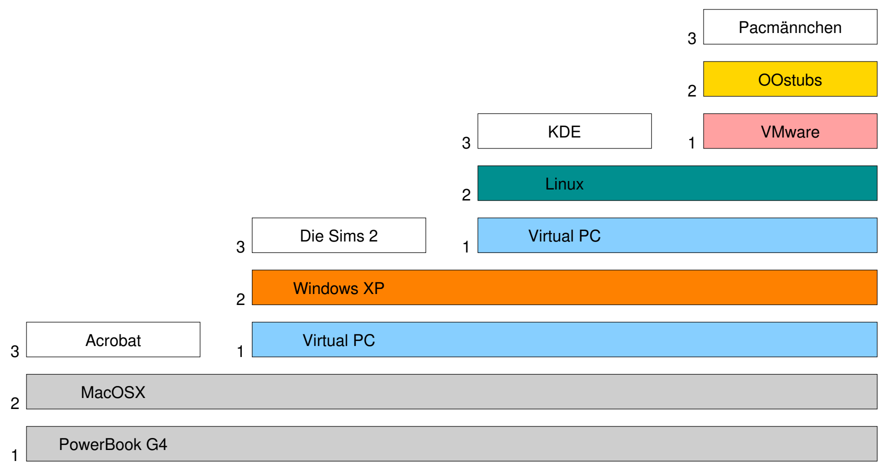
Selbstvirtualisierung durch Teilinterpretation (Forts.)

Jede Ausnahmesituation aktiviert den privilegierten Arbeitsmodus:

1. das Betriebssysteme/der VMM analysiert die Unterbrechung
   - d.h., den Systemaufruf, Trap oder Interrupt
2. für das unterbrochene Programm wird ein Emulator gestartet
   - Ausführung des die Unterbrechung hat verursachenden Befehls
   - Abbildung von Geräteaktionen auf E/A-Funktionen des Betriebssystems
   - Umsetzung von Adressraumzugriffen und privilegierten Befehlen
3. das unterbrochene Programm wird weiter fortgeführt
   - Beendigung der Emulation des Befehls bzw. Zugriffs
   - Reaktivierung des nicht-privilegierten Arbeitsmodus

☞ Abruf- und Ausführungszyklus eines „fiktiven“ Prozessors durchlaufen

Hierarchie virtueller Maschinen



Grenzen der Emulation

Funktionale vs. nicht-funktionale Eigenschaften

Nachahmung funktionaler Eigenschaften ist „leicht“ möglich

- d.h., in Funktionseinheiten gekapselter Fähigkeiten eines Prozessors:
  - Fließkommaeinheit, Vektoreinheit, Graphikbeschleuniger
  - Adressumsetzungs- und Kommunikationshardware
  - ISA eines beliebigen Prozessors
- solitäre („einzeln stehende“) Funktionen von Hardware oder Software

Schwierigkeiten bereiten nicht-funktionale Eigenschaften:

- ein emulierter Befehl wird durch ein „Unterprogramm“ ausgeführt
- er läuft dadurch langsamer ab und verbraucht auch mehr Energie

Virtual PC: MacOS $\longmapsto$ Windows XP

Ein 1.25 GHz PowerPC G4 wird zum 293 MHz Pentium 686.

Rechnerorganisation

Strukturierte Organisation von Rechensystemen

Hierarchie virtueller Maschinen (bzw. abstrakter Prozessoren)

- schrittweises Schließen der semantischen Lücke
- Mehrebenenmaschinen — Betriebssysteme implementieren Ebene $_3$
- Ebene $_i \mapsto$ Ebene $_{i-1}$ durch Programme, für $i > 1$
- Teilinterpretation (von Systemaufrufen) durch das Betriebssystem
- synchrone und asynchrone Programmunterbrechungen
- nebenläufige (bzw. überlappende) Ausführung von Programmen
- Nachahmung von Eigenschaften ggf. anderer (abstrakter) Prozessoren