Betriebssystemtechnik

Programmunterbrechungen: Weitergabe von *Interrupts*

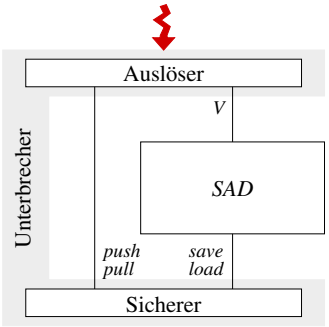
Wolfgang Schröder-Preikschat

Lehrstuhl Informatik 4

15. Mai 2012

Gliederung

Hardwaresignal ↦ Softwaresignal



Aufgabe des Auslösers:

1. Annahme und Quittierung einer Unterbrechungsanforderung
2. Generierung eines Gerätesignals: `V`

Beachte

1. läuft auf $IPL \geq 0$
2. sollte/muss auf $IPL = 0$ laufen

Anforderungen

zu 1. Laufzeitminimierung der Unterbrechungssperre ($IPL > 0$)
- konstruktiv, durch Zweiteilung der Behandlungsroutine

zu 2. sichere Unterbrechungsentsperrung: $IPL > 0 \mapsto IPL = 0$
- Emulation eines asynchronen Systemsprungs

Gliederung

Gewaltenteilung

Vorspann $\hookrightarrow$ FLIH (Abk. für engl. *first-level interrupt handler*)
- ist bedingt unterbrechbar:
  Flankensteuerung $\iff$ $IPL_{irq} > IPL_{off} = 0$
  Pegelsteuerung $\iff$ $IPL_{irq} > IPL_{act}$
- erledigt nur die dringend nötigen Geräteaufgaben:
  Flankensteuerung $\implies$ ggf. nichts
  Pegelsteuerung $\implies$ Abnahmequittung zustellen
- löst bei Bedarf einen Nachspann aus

Nachspann $\hookrightarrow$ SLIH (Abk. für engl. *second-level interrupt handler*)
- ist grundsätzlich unterbrechbar: $IPL = 0$
- erledigt ausstehende/angefallene Geräteaufgaben

Beachte: Nachspann $\neq$ Epilog [3, 1]
- läuft zwar mit $IPL = 0$, jedoch asynchron im Betriebssystemkontext
- darf weiterhin nur wiedereintrittsinvariante Funktionen verwenden!

Steuerung der Unterbrechungsbehandlung

TIP (Abk. engl. *trap/interrupt propagation*)

```
#include "luxe/queue.h"
#include "luxe/remit.h"

typedef remit_t tip_t;              /* deferrable SLIH abstraction */

extern void tip_entry ();           /* come here upon trap/interrupt */
extern void tip_start ();           /* start of FLIH, also: prototye */
extern void tip_defer (tip_t *);    /* provision of SLIH */
extern void tip_unban (tip_t *);    /* release of deferred SLIH */
extern void tip_check ();           /* propagate SLIH, if allowed */
extern void tip_clear ();           /* propagate SLIH, if any */

extern queue_t *tip_store ();       /* return pointer to SLIH queue */
```


Einstieg: Vorspann aktivieren

Pegelsteuerung: ASM (x86)

```
__GLOBAL(tip_entry):                /* come here interrupts disabled... */
    TIP_PUSH                        /* save relevant CPU state */
    call __GLOBAL(tip_start)        /* invoke associated FLIH */
                                    /* fall into SLIH check... */
```

Flankensteuerung: ASM (x86)

```
__GLOBAL(tip_entry):                /* come here interrupts disabled... */
    incl tip_level                  /* register this interrupt */
    sti                             /* approve interruptibility */
    TIP_PUSH                        /* save relevant CPU state */
    call __GLOBAL(tip_start)        /* invoke associated FLIH */
                                    /* fall into SLIH check... */
```

Beachte: Einstieg in den SLIH (S. 9/10)

Verschiedene Ausnahmevektoren verweisen auf verschiedene Einstiegsroutinen, die dann mit `jmp __GLOBAL(tip_check)` enden.

Hilfsmittel der Assembliersprachenebene

Externe Symbolnamen

```
#ifdef __luxe_leading_underscore__
#define __GLOBAL(name) _ ## name
#else
#define __GLOBAL(name) name
#endif
```

Beachte die traditionelle Weise

Einige Kompilierer für C erzeugen Symbolnamen mit einem führenden Unterstrich (engl. *underscore*). So etwa gcc für früheres Linux sowie für Win32 und die BSD-Familie.

Sicherer: Flüchtige Register

```
#ifndef TIP_PUSH
#define TIP_PUSH \
    pushl %eax; \
    pushl %ecx; \
    pushl %edx
#endif

#ifndef TIP_PULL
#define TIP_PULL \
    popl %edx; \
    popl %ecx; \
    popl %eax
#endif
```

Übergang: Nachspann bedingt aktivieren (1)

Pegelsteuerung

```
__GLOBAL(tip_check):
    cmpl $0, __GLOBAL(tip_queue)  /* any pending SLIH? */
    je 1f                         /* no, done */
    cmpb $0, tip_level            /* TIP critical section active? */
    jne 1f                        /* yes, do not reenter! */
    movb $1, tip_level            /* enter TIP critical section */
2:  sti                           /* approve interruptibility */
    call __GLOBAL(tip_clear)      /* propagate pending SLIH */
    cli                           /* prevent interruptibility */
    cmpl $0, __GLOBAL(tip_queue)  /* any SLIH pending again? */
    jne 2b                        /* yes, clear */
    movb $0, tip_level            /* leave TIP critical section */
1:  TIP_PULL                      /* restore relevant CPU state */
    iret
```

Beachte: Wettlaufsituation

- es muss sichergestellt sein, dass kein Systemsprung vergessen wurde

Übergang: Nachspann bedingt aktivieren (2)

Flankensteuerung

```
__GLOBAL(tip_check):
    cmpl $1, tip_level            /* at bottom of interrupt stack? */
    jne 1f                        /* no, do not propagate! */
2:  cli                           /* prevent interruptibility */
    cmpl $0, __GLOBAL(tip_queue)  /* any pending SLIH? */
    je 1f                         /* no, done */
    sti                           /* re-approve interruptibility */
    call __GLOBAL(tip_clear)      /* propagate pending SLIH */
    jmp 2b                        /* re-check for sporadic SLIH  */
1:  TIP_PULL                      /* restore relevant CPU state */
    cli                           /* prevent interruptibility */
    decl tip_level                /* deregister this interrupt */
    iret
```

Beachte: Wettlaufsituation

- wie zuvor (S. 9) darf auch hier kein Systemsprung verloren gehen
  - kritisch: Abfrage der Warteschlange und Auswertung der Abfrage

Richtiger Zeitpunkt der Nachspannaktivierung

Durchsetzung eines (emulierten) asynchronen Systemsprungs erfordert, dass keine Inkarnation einer Unterbrechungsbehandlung mehr aktiv ist

*de*LUXE

```
.lcomm tip_level,1,0
```

- je nach Steuerungsart ein Bitschalter oder Inkarnationszähler
- richtiger Zeitpunkt $\Longleftrightarrow$ $level = \begin{cases} 0 & \text{bei Pegelsteuerung} \\ 1 & \text{bei Flankensteuerung} \end{cases}$

Beachte

- in der Situation ist kein Vorspann mehr aktiv, auch kein Nachspann
  - nur der gesicherte Zustand einer Inkarnation liegt auf dem Stapel
  - dieser würde auch bei einem echten AST wieder oben rauf kommen
- die Stapelausdehnung bleibt trotz Unterbrechungsfreigabe begrenzt
  - festgelegt durch die Anzahl unterstützter Unterbrechungsebenen
  - hinzu kommt noch Stapelbedarf des „tiefsten“ Nachspanns

Richtiger Zeitpunkt der Nachspannaktivierung (Forts.)

Wettlaufsituation

```
__GLOBAL(tip_entry):
    incl tip_level
    sti
```

. . . im Falle kaskadierbarer Unterbrechungen!!!

- bei Unterbrechung vor `incl tip_level`
- auch hier (x86 und ggf. PIC): NMI

Lösungsansätze:

- NMI ausnehmen, d.h. nicht auf asynchronen Systemsprung abbilden
- anstatt Zählen, spezielle Fähigkeiten der CPU nutzen, z.B.:
  - wenn der 1. Befehl noch garantiert ausgeführt wird (TMS 9900)
  - wenn das Statusregister Informationen zum Arbeitsmodus enthält
    - im Unterbrechungsfall wurde der Inhalt auf dem Stapel gesichert
    - mit Informationen aus dem Zeitraum direkt *vor* der Unterbrechung
    - richtiger Zeitpunkt zum Systemsprung: voriger Modus war „Benutzer“
    - Überprüfung: indirekt indizierte Adressierung mittels Stapelzeiger

Beachte

Wenn der NMI nicht ausgenommen werden soll, dann muss eine Lösung allein auf Basis der CPU gefunden werden!

Unterbrechungsfreigabe

Verlassen des Unterbrechers: Rückkehr zur unterbrochenen Programmausführung

Aufhebung der Unterbrechungssperre, wenn dem Konzept entsprechend eine unbestimmte Ausdehnung des Stapels nicht möglich ist[1]

- *zuerst* Unterbrechungen zulassen, um sie *danach* wieder abzuwehren
- ist nicht mit einer Unterbrechungssperre zu verwechseln
  - die umgekehrt funktioniert, zum Schutz kritischer Abschnitte

Beachte

- Unterbrechungsabwehr bringt das System nur in den Normalzustand zurück, der nämlich dafür sorgt, dass
  1. kein zurückgestellter Systemsprung vergessen wird und
  2. der Stapel nicht unbestimmt wachsen kann
     - bei Unterbrechung zwischen `decl tip_level` und `iret` (S. 10)

[1] Nicht dem Konzept entsprechend ist die Verwendung von Einstieg und Übergang zur Unterbrechungsfreigabe bei Flankensteuerung für den pegelgesteuerten Betrieb. Umgekehrt genauso, unbestimmtes Anwachsen des Stapels geht dann jedoch nicht.

Freigabe zurückgestellter Systemsprünge

Nachspann zurückstellen: Ausgelöst durch den FLIH (`tip_start()`)

```
INLINE void tip_defer (tip_t *item) {
    assert(item);
    its_aback(tip_store(), (chain_t*)&item->next);
}
```

Nachspann freistellen: Ausgelöst im Übergang (`tip_check()`)

```
INLINE void tip_unban (tip_t *item) {
    (item->work)();
}

void tip_clear () {
    tip_t *next;
    do if ((next = (tip_t*)its_fetch(tip_store()))) tip_unban(next);
    while (next != 0);
}
```

- ITS steht für (Abk. engl.) *interrupt transparent synchronization*

Abstraktion zurückgestellter Systemsprünge

Nachspann $\mapsto$ Prozeduradresse in einer Kette

```
#include "luxe/chain.h"

struct remit {
    chain_t next;       /* next remit in sequence, if any */
    void (*work)();     /* deferred procedure */
};

typedef struct remit remit_t;
```

- ein Auftrag (engl. *remit*), um einen Systemsprung zu leisten
- zur Buchführung/Abrechnung bieten sich Spezialisierungen an
  - z.B. Verbuchung der pro Nachspann verbrauchten Systemzeit

Beachte

- Zurückstellung eines Nachspanns trifft eine Planungsentscheidung
- diese muss nicht zwingend auf FCFS hinauslaufen...

Wiederbehauptung asynchroner Systemsprünge

Wettlaufsituation im Falle der Zurückstellung eines Nachspanns, der sich noch auf der Nachspannwarteschlange befindet

- erneute Einreihung in die Warteschlange ist zu unterbinden

  Vorbeugung heißt, die Unterbrechungsfrequenz kann die WCET[2] *aller* möglichen Nachspänne nicht unterschreiten ?

  Vermeidung zieht algorithmische Lösungen in Erwägung ✗
- wobei das Nachspannereignis ggf. nicht verloren gehen darf

Lösungsansätze

- eine Freiliste (engl. *free list*) von Auftragsdeskriptoren verwalten
  - *defer* leert und *clear* füllt diese Liste in Kooperation
  - ist die Liste leer, gehen Nachspannereignisse verloren
- im Auftragsdeskriptor einen Ereigniszähler führen/auswerten ✗
  - *defer* erhöht und *clear* erniedrigt den Zähler in Kooperation
  - bei Zählerüberlauf, gehen Nachspannereignisse verloren

[2] Abk. engl. *worst case execution time*

Wiederbehauptung asynchroner Systemsprünge (Forts.)

Ernüchterung

- Propagierung schließt Ereignisverlust nicht ganz aus
  - kann bestenfalls unwahrscheinlicher gemacht werden
- betrifft Gerätetreiber mehr oder weniger stark
  - lässt sich nur End-zu-End [2] entscheiden
- ggf. werden zus. Vor-/Nachspannprotokolle benötigt

TIP Spezialisierung: *safe interrupt propagation*, SIP *de*LUXE

```
struct apart {          /* NOTE: this is a tip_t variant */
    remit_t item;       /* remit abstraction controlled */
    char    busy;       /* processing state */
};
```

```
void sip_defer (tip_t *item) {
    assert(item);
    if (!ami_apply(&item->busy))
        tip_defer(item);
}
```

```
void sip_unban (tip_t *item) {
    item->busy = 0;
    tip_unban(item);
}
```

apply $\models$ atomares *test and set*, TAS

Gliederung

Lastverteilung bei der Unterbrechungsbehandlung

Mehrprozessortechnik gibt dem Betriebssystem Möglichkeiten zu einer Entlastung der Recheneinheiten von der Unterbrechungsbehandlung

Idee *de*LUXE

```
__GLOBAL(tip_entry):
    TIP_PUSH
    call __GLOBAL(tip_start)
    TIP_PULL
    iret
```

- jeder Prozessor verarbeitet nur noch den für ihn bestimmten Vorspann
- die Verarbeitung des Nachspanns übernimmt ein anderer Prozessor

- neben *entry* zeichnen sich Änderungen in *defer* und *clear* ab: sie sind
  1. multiprozessorsicher auszulegen und müssen
  2. prozessorübergreifenden Abfrage- oder Anzeigebetrieb unterstützen
- als Modelle bieten sich an: Fließband, Zusteller und Zustellergruppe

Beachte

- Mehrprozessortechnik umfasst (natürlich) mehrkernige Prozessoren

Lastverteilung bei der Unterbrechungsbehandlung (Forts.)

Abfragebetrieb

- aktives Warten (engl. *busy waiting*)
- *clear* stellt den nächsten verfügbaren Nachspann frei, überprüft dabei anhaltend die Warteschlange
- *defer* reiht einen Nachspann in die Warteschlange ein

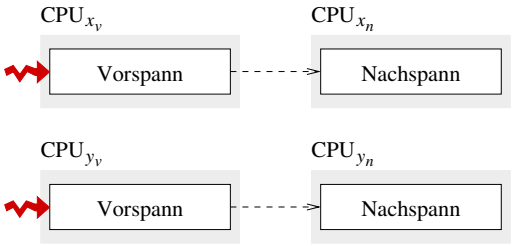
Anzeigebetrieb

- passives Warten
- *clear* stellt den nächsten verfügbaren Nachspann frei und suspendiert seine CPU bei leerer Warteschlange
- *defer* reiht einen Nachspann in die Warteschlange ein und signalisiert ggf. die beauftragte CPU

Beachte

- welche CPU im Anzeigebetrieb zu signalisieren ist, hängt ganz vom Modell der Lastverteilung ab:
  - statisch im Falle von Fließband und Zusteller, für gewöhnlich
  - dynamisch im Falle von Zustellergruppe

Fließband

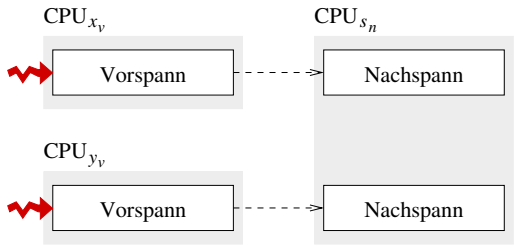


Prinzip

- CPU_{x_v} & CPU_{y_v} sind Vorspannstufen
- CPU_{x_n} & CPU_{y_n} sind feste Nachspannstufen

- jede Nachspannstufe führt die Nachspänne ihrer Vorspannstufe aus
  - zeitversetzt, parallel zu nachfolgenden Unterbrechungen/Vorspännen
- 1:1-Wettlaufsituation der gemeinsamen Nachspannwarteschlange
  - *defer* ausgeführt auf CPU_{x_v} konkurriert mit *clear* auf CPU_{x_n}
  - entsprechendes Muster der Kooperation gilt für CPU_{y_v} und CPU_{y_n}
- Nachspannverarbeitung geringer Durchlaufzeit steht im Vordergrund
  - nicht aber Maximierung der Auslastung von CPU_{x_n} & CPU_{y_n}

Zusteller

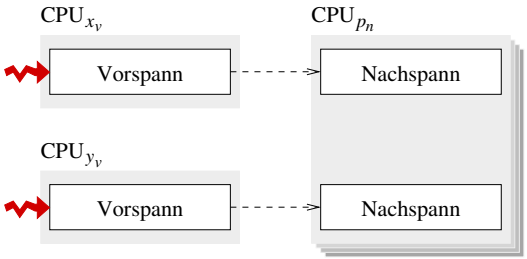


Prinzip

- CPU_{x_v} & CPU_{y_v} sind Vorspannstufen
- CPU_{s_n} ist zentrale, feste Nachspannstufe

- eine Nachspannstufe führt die Nachspänne aller Vorspannstufen aus
  - CPU_{x_v} & CPU_{y_v} sind Klienten, CPU_{s_n} ist Zusteller (engl. *server*)
- N:1-Wettlaufsituation der gemeinsamen Nachspannwarteschlange
  - *defer* auf CPU_{x_v} & CPU_{y_v} konkurrieren miteinander
  - *clear* auf CPU_{s_n} konkurriert mit *defer* auf CPU_{x_v} & CPU_{y_v}
- Auslastungsmaximierung von CPU_{s_n} steht im Vordergrund
  - weniger eine geringe Durchlaufzeit der Nachspannverarbeitung

Zustellergruppe



Prinzip

- CPU_{x_v} & CPU_{y_v} sind Vorspannstufen
- CPU_{p_n} ist ein Satz von Nachspannstufen

- M $>$ 1 Nachspannstufen führen Nachspänne der Vorspannstufen aus
  - CPU_{x_v} & CPU_{y_v} Klienten, CPU_{p_n} Zustellergruppe (engl. *server pool*)
- N:M-Wettlaufsituation der gemeinsamen Nachspannwarteschlange
  - *defer* auf CPU_{x_v} & CPU_{y_v} konkurrieren miteinander
  - *clear* auf CPU_{p_n} konkurrieren miteinander
  - *clear* auf CPU_{p_n} konkurriert mit *defer* auf CPU_{x_v} & CPU_{y_v}
- Auslastungsmaximierung & Durchlaufzeitminimierung als Ziel

Asymmetrisches Mehrprozessorsystem

Koprozessoren entlasten Prozessoren von der Unterbrechungsbehandlung

- CPU_{x_n}, CPU_{y_n}, CPU_{s_n} und CPU_{p_n} wären solche Koprozessoren
- nur diese sind für die Nachspannverarbeitung verantwortlich
  - sie erledigen die „schwere“ Arbeit zur Unterbrechungsbehandlung
  - sie werden auch evtl. Fortsetzungen ins Betriebssystem vorantreiben
- die anderen, CPU_{x_v} & CPU_{y_v}, machen nur Vorspannverarbeitung
  - sie erledigen die „leichte“ Arbeit zur Unterbrechungsbehandlung
- das BS ist *funktional dediziert verteilt* über die Prozessoren

Beachte

Vollständige Entlastung eines Prozessors sogar von der Verarbeitung eines Vorspanns der Unterbrechungsbehandlung setzt spezielle Hardware voraus, die Unterbrechungsanforderungen (statisch oder dynamisch) an andere Prozessoren weiter- bzw. umleitet.

- APIC, Abk. für engl. *advanced programmable interrupt controller*

Verwendung von Prozessor(kern)en im Betriebssystem

- Mehrkernprozessor
  - Universalkerne — *Balancierung allgemeiner Systemlast*
  - Spezialkerne — *Kaschierung systembedingter Latenzen*
    - Wandelkerne — *Behandlung von Systemaufufen als Ganzes*
    - Fixkerne — *Behandlung von Systemaufufen in Stücken*
      - Treiberkerne — *abgesetzte Geräteprogrammierung*
        - Gerätekerne — *Steuerung der Peripherie*
        - Unterbrecherkerne — *Annahme von (asynchronen) Programmunterbrechungen*
      - Dienstkerne — *abgesetzte Ausführung von Systemfunktionen*
        - Einplanungskern — *Zustellung von Arbeitsaufträgen*
        - Ein–/Ausgabekern — *Operationen auf Dateien*
        - Kommunikationskern — *Abwicklung des Protokollstapels*
        - Seitenumlagerungskern — *Durchführung der Ersetzungsstrategie (external pager)*
        - ⋮

Beachte

- Spezialkerne effizient („minimal invasiv“) zu bedienen, ist knifflig. . .

Problem: Zwischenspeicher (engl. *cache*)

Fließband- wie auch Zustellerkonzepte wechseln den Prozessor(kern) im weiteren Verlauf der Ereigniszustellung

- die Ausführung des Vorspanns[3] hat den Zwischenspeicher (L1) mit den demnächst wahrscheinlich benötigten Adressen „aufgewärmt“
  - jeder Prozessor(kern) hat seinen eigenen Zwischenspeicher (L1)
  - nur der des Prozessor(kern)s des Vorspanns ist aber „heiß“
  - die der anderen sind jedoch „kalt“ hinsichtlich des Vorspannkontextes
- nach Wechsel des Prozessor(kerns) zur Aufnahme der Ausführung des Nachspanns muss der Zwischenspeicher erst wieder aufgebaut werden
  - vermehrte Zugriffsfehler (engl. *cache miss*) werden die Folge sein
  - der Mehraufwand macht den Gewinn durch Parallelität schnell wett

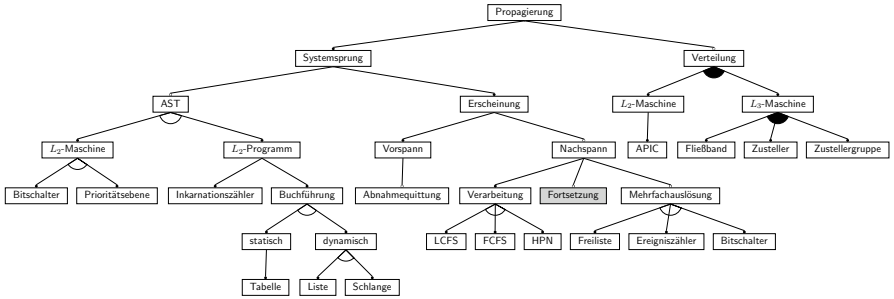
Herausforderungen

- passend „gewichtige“ Nachspänne, für die sich der Mehraufwand lohnt
- bedingte Propagierung, abhängig vom Füllstand der Zwischenspeicher
- Hardware, die den jeweils „besten“ Prozessor(kern) von selbst findet

[3] inkl. der zur Emulation eines AST benötigten Programme.

Gliederung

Artefakte der Unterbrechungsweitergabe



Fortsetzung folgt. . .

- Synchronisierung mit den Abläufen im Betriebssystem
  - ↪ Epilog [1], wobei ein Vorspann den Prolog repräsentiert
    - der Epilog selbst aber die Fortsetzung eines Nachspanns ist
    - der BST-Ansatz sich damit vom BS-Ansatz unterscheidet

Resümee

Rekapitulation $\hookleftarrow$ Ereigniszustellung

- Umwandlung eines Hardwaresignals in ein Softwaresignal
- Unterbrecher: Auslöser und Sicherer

Unterbrechungsbehandlung $\hookleftarrow$ Vor-/Nachspann

- Weiter-/Freigabe von Unterbrechungen
- Freigabe und Wiederbehauptung von Systemsprüngen

Parallelverarbeitung $\hookleftarrow$ Mehrprozessortechnik zur Lastverteilung

- Abfrage- oder Anzeigebetrieb
- Fließband, einzelner Zusteller oder Zustellergruppe
- Lokalitäten und Zwischenspeicherstände beachten

Literaturverzeichnis

[1] Lohmann, D. ; Kleinöder, J. :
Betriebssysteme.
http://www4.informatik.uni-erlangen.de/Lehre/WS07/V_BS, 2007

[2] Saltzer, J. H. ; Reed, D. P. ; Clark, D. D.:
End-To-End Arguments in System Design.
In: *ACM Transactions on Computer Systems* 2 (1984), Nov., Nr. 4, S. 277–288

[3] Schröder-Preikschat, W. :
The Logical Design of Parallel Operating Systems.
Upper Saddle River, NJ, USA : Prentice Hall International, 1994. –
ISBN 0–13–183369–3