— BS //—

Synchronisation

Betriebssysteme, ©Wolfgang Schröder-Preikschat

Überblick

Kooperation und Konkurrenz

Die Koordination der Kooperation und Konkurrenz zwischen Prozessen wird **Synchronisation** (*synchronization*) genannt. Eine Synchronisation bringt die Aktivitäten verschiedener nebenläufiger Prozesse in eine Reihenfolge. Durch sie erreicht man also prozeßübergreifend das, wofür innerhalb eines Prozesses die Sequentialität von Aktivitäten sorgt. [1], S. 26

¿Synchronisation?

- Wie ist vor diesem Hintergrund die Interrupt-Synchronisation einzuordnen?
- Muss die Interrupt-Behandlungsroutine immer ein eigener Prozess sein?
- Ist ein interrupt-behaftetes System immer ein System aus mehreren Prozessen?
- Kann es in Einprozesssystemen überhaupt Synchronisationsbedarf geben?

⋮

- Was ist ein Prozess?

zum Verständnis → pp. 4–11

Ein Prozess . . .

. . . wird durch ein Programm kontrolliert und benötigt zur Ausführung dieses Programms einen Prozessor.

Habermann, *Introduction to Operating System Design*

. . . P ist ein Tripel (S, f, s), wobei S einen Zustandsraum, f eine Aktionsfunktion und $s \subset S$ die Anfangszustände des Prozesses P bezeichnen. Ein Prozess erzeugt Abläufe, die durch die Aktionsfunktion generiert werden können.

Horning/Randell, *Process Structuring*

. . . ist das Aktivitätszentrum innerhalb einer Folge von Elementaroperationen. Damit wird ein Prozess zu einer abstrakten Einheit, die sich durch die Instruktionen eines abstrakten Programms bewegt, wenn dieses auf einem Rechner ausgeführt wird.

Dennis/van Horn, *Programming Semantics for Multiprogrammed Computations*

. . . ist ein Programm in Ausführung.

unbekannte Referenz, „Mundart“

Programm vs. Prozess

Programm ist *statisch*, ein konkretes Gebilde

- eine zur **Übersetzungszeit** festgelegte Folge von Elementaroperationen

Prozess ist *dynamisch*, ein abstraktes Gebilde

- eine sich erst zur **Laufzeit** einstellende Folge[1] von Elementaroperationen
  - unterschiedliche Eingabewerte bewirken ggf. andere Ausführungspfade
  - unterschiedliche Ausführungspfade bedeuten andere Elop-Folgen
  - andere Elop-Folgen werden auch von Unterbrechungen hervorgerufen
  - Unterbrechungen sind nicht immer eingeplant, beispielsweise Interrupts
- die Elop-Folge ist interrupt-bedingt nicht bzw. nur schwer vorhersagbar

[1] Auch bei unverändertem Programm muss diese Folge nicht immer gleich sein.

Prozess vs. Interrupt

Prozess ist ein asynchroner Programmablauf

- Prozesse laufen $\left\{ \begin{array}{l} \text{unabhängig voneinander} \\ \text{mit nicht vorhersagbarer relativer Geschwindigkeit} \end{array} \right\}$ [2]

Interrupt ist eine asynchrone Programmunterbrechung

- Interrupts müssen von Programmen behandelt werden
  - Ausnahmebehandlung auf $\left\{ \begin{array}{ll} \text{Ebene}_3 & \text{Betriebssystemebene} \\ \text{Ebene}_i, i > 3 & \text{Anwendungsebene(n)} \end{array} \right.$
- Interrupts bewirken asynchrone Programmabläufe
  - sie „schöpfen“ einen neuen Prozess, der nicht zwingend physisch existiert
- Interrupts lösen die Ausführung anderer Prozesse aus

Prozess — logische vs. physiche Sicht

logische Sicht „Prozess“ als abstraktes Gebilde

- ein unabhängig vom Rest des Programms ablauffähiger Programmteil
- differenziert zwischen „Prozess“ und „Inkarnation eines Prozesses“

physische Sicht „Prozess“ als „Prozessinkarnation“

- ein Programmfaden (*thread*) bzw. ein aktives Objekt
  - versorgt mit eigenem Stapel (*stack*) und ggf. eigenem Adressraum
  - erzeugt, gepflegt und zerstört von der Prozessverwaltung
- ein „Interrupt-Prozess“ *kann* als Prozessinkarnation vorliegen[2]

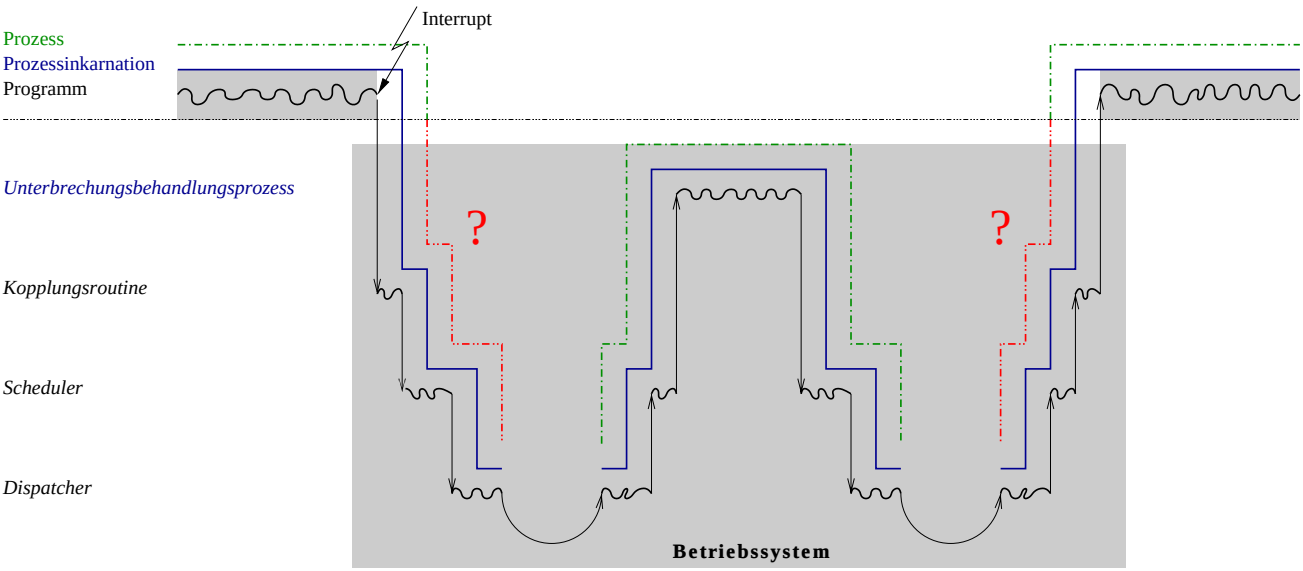
[2] Hängt ganz von den Eigenschaften der CPU (Ebene $_2$) bzw. der jeweiligen Betrachtungsebene ab.

Interrupt als Prozessinkarnation (1)

Soll die Unterbrechungsbehandlung durchgängig von einer Prozessinkarnation erledigt werden, muss, rein technisch gesehen, die CPU (Ebene $_2$) ein Prozesskonzept realisieren. Ist dies nicht der Fall — *Welche CPU implementiert ein Prozesskonzept?* —, so stellt das Betriebssystem (Ebene $_3$) keine Alternative für eine durchgängige Lösung dar. Diese Alternative würde ja bedeuten, dass vor und nach dem „Interrupt-Prozess“ Programme der Ebene $_3$ auszuführen sind, um den Prozesswechsel herbeizuführen. *Welcher Prozessinkarnation wären diese Anweisungen dann jedoch zuzuordnen?*

Unabhängig davon kann jedoch oberhalb der Ebene $_3$ eine Sicht bestehen, bei der die Unterbechungsbehandlung scheinbar durchgängig von einem „Interrupt-Prozess“ bewerkstelligt wird.

Interrupt als Prozessinkarnation (2)

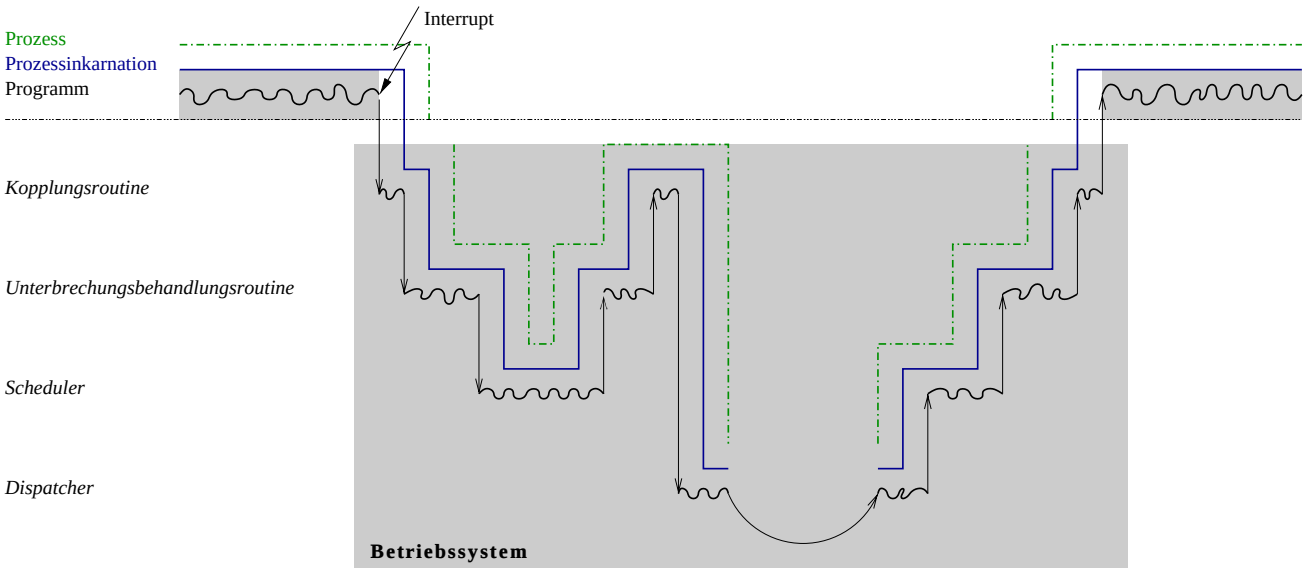


Interrupt vs. Prozessinkarnation

- die rekursive Aktivierung der Unterbrechungsbehandlung wirft Probleme auf
  - ein „Interrupt-Prozess“ könnte zwar von sich selbst unterbrochen werden
  - der Wechsel zur zugehörigen Inkarnation würde jedoch nichts bewirken[3]
- jeder Interrupt müsste von einer eigenen Prozessinkarnation behandelt werden
  - ggf. ist diese dynamisch mit jedem Interrupt zu erzeugen und zu zerstören
  - zumindest müsste ein Lager solcher Prozessinkarnationen vorhanden sein
- *Interrupts sind Prozesse . . . . . aber nicht zwingend Prozessinkarnationen*

[3] In Anhängigkeit davon, wie der Prozesswechsel technisch realisiert ist, würde der von sich selbst unterbrochene „Interrupt-Prozess“ günstigstenfalls zu sich selbst wechseln und dort weitermachen, wo er zuletzt die Kontrolle abgegeben hatte — dies ist aber ganau die Stelle, an der er sich selbst unterbrach!

Programm . . . Unterbrechung . . . Prozess . . . Prozessinkarnation



Synchronisationsarten

- die Prozessinkarnationen sind zu betrachten — und sie sind zu differenzieren:

Koordination der Kooperation und Konkurrenz zwischen Prozessen

1. ***derselben Prozessinkarnation***
   - einem Prozess und dem ihn unterbrechenden „Interrupt-Prozess“
   - Synchronisation asynchroner Programmunterbrechungen
   - darf sich niemals blockierend auswirken
2. ***verschiedener Prozessinkarnationen***

- daraus resultieren unterschiedliche Synchronisationsverfahren und -techniken
  - ein- vs. mehrseitige Synchronisation nebenläufiger Prozesse

Einseitige Synchronisation — Unilateral

- die Synchronisation wirkt sich nur auf einen der beteiligten Prozesse aus:

  Bedingungssynchronisation bzw.
  - das Weiterarbeiten des einen Prozesses ist abhängig von einer Bedingung
  - der andere Prozess erfährt keine Verzögerung in seinem Ablauf

  logische Synchronisation
  - die Maßnahme resultiert aus der logischen Abfolge der Aktivitäten
  - vorgegeben durch das Zugriffsmuster der beteiligten Prozesse

- andere Prozesse sind jedoch nicht gänzlich an der Synchronisation unbeteiligt[4]

[4] Die Veränderung einer Bedingung, auf die ein Prozess wartet, wird natürlich von einem anderen Prozess herbeigeführt.

Mehrseitige Synchronisation — Multilateral

- die Synchronisation wirkt sich auf (ggf.) alle beteiligten Prozesse aus
  - welche Prozesse im weiteren Ablauf verzögert werden, ist unvorhersehbar
- die betroffenen Aktivitäten stehen miteinander im **gegenseitigen Ausschluss**
  - sie verhalten sich zueinander, als seien sie unteilbar
    - d.h., sie werden niemals nebenläufig/parallel durchgeführt
  - keine Aktivität unterbricht bzw. überlappt die andere Aktivität
- gegenseitigen Ausschluss erfordernde Anweisungen sind **kritische Abschnitte**

Synchronisation asynchroner Programmunterbrechungen

- je nach Verfahren erfährt die eine oder andere Seite eine Verzögerung:

  Verzögerung des unterbrechenden Prozesses
  - harte/weiche Synchronisation der „Hardware/Software-Interrupts“

  Verzögerung des unterbrochenen Pozesses
  - nicht-blockierende Synchronisation nebenläufiger Aktivitäten
- je nach Verfahren sind dazu spezielle Elementaroperationen erforderlich
  - implementiert von Ebene $_2$ (CPU): {`cli`, `sti`}, `cas`, `cmpxchg`, `ll`/`sc`
- die Verfahren synchronisieren einseitig, d.h. sie arbeiten unilateral

Fallstudie — Atomarer Zähler

```
class atomicCounter {
    int item;
public:
    atomicCounter (int = 0);

    int operator ++ ();
    int operator ++ (int);
    int operator += (int);

    void raise ();
};
```


Disable/Enable Interrupts (1) cli/sti

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

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

Disable/Enable Interrupts (2)

cli/sti

```
int atomicCounter::operator ++ () {
    int aux;
    cli();
    aux = ++item;
    sti();
    return aux;
}
```

.....

```
___pp__13atomicCounter:
    movl 4(%esp),%edx
    cli
    movl (%edx),%eax
    leal 1(%eax),%ecx
    movl %ecx,(%edx)
    movl %ecx,%eax
    sti
    ret
```


Disable/Enable Interrupts (3)

cli/sti

```
int atomicCounter::operator ++ (int) {
    int aux;
    cli();
    aux = item++;
    sti();
    return aux;
}
```

.....

```
___pp__13atomicCounteri:
    movl 4(%esp),%eax
    cli
    movl (%eax),%edx
    incl (%eax)
    sti
    movl %edx,%eax
    ret
```

Disable/Enable Interrupts (4) `cli/sti`

```
int atomicCounter::operator += (int i) {
    int aux;
    cli();
    aux = item += i;
    sti();
    return aux;
}
```

.....

```
___apl__13atomicCounteri:
    movl 4(%esp),%eax
    cli
    movl (%eax),%edx
    addl 8(%esp),%edx
    movl %edx,(%eax)
    sti
    movl %edx,%eax
    ret
```


Disable/Enable Interrupts (5) `cli/sti`

Problem Die Operationen klammern kritische Bereiche. Wird die „öffnende Klammer“ (`cli`) passiert, jedoch nicht die zugehörige „schließende Klammer“ (`sti`), bleiben asynchrone Programmunterbrechungen auf ungewisse Zeit gesperrt.[5]

Weiteres Problem ist die **harte Synchronisation**, d.h., auf von der Peripherie signalisierte Ereignisse kann nicht (sofort) reagiert werden.

Lösung

- unterbrechungstransparente Synchronisation [ein anderes Kapitel]
- nicht-blockierende Synchronisation → pp. 23–28

[5] Gäbe es Sprachkonstrukte zum Sperren kritischer Bereiche, könnte ein Übersetzer zumindest eine Fehlermeldung in dem Fall anzeigen. Der Konstruktor-/Destruktormechanismus von C++ kann „zweckentfremdet“ werden, um sicherzustellen, dass beim Verlassen kritischer Abschnitte die Synchronisation automatisch wieder aufgehoben wird[5]. Auch als *scoped locking* bekanntes Muster[4].

„Elementare Komplexoperation“ CISC

```
void atomicCounter::raise () {
    asm volatile ("incl %0" : "=m" (item));
}
```


Compare and Swap (1) cas

```
int cas (int& ref, int now, int val) {
    register short srZ;
    <atomic>
    if (srZ = (ref == now)) ref = val;
    <cimota>
    return srZ;
}
```

```
int atomicCounter::operator ++ (int) {
    int aux;
    do aux = item;
    while (!cas(item, aux, aux + 1));
    return aux;
}
```

Compare and Swap (2) cas

Problem Der cas-Befehl ist zwar unteilbar, er prüft jedoch nur, ob der gegenwärtige Wert (now) noch dem alten Wert (ref) entspricht. Dass der alte Wert in der Zwischenzeit nach dem Lesen ggf. geändert und dann wieder hergestellt worden ist, wird jedoch nicht erkannt.

Annahme ein Prozess muss erfahren, dass ein Speicherbereich aktualisiert wurde und es besteht dafür keine explizite Möglichkeit der Signalisierung. Der betreffende Prozess kann nicht jede Speicheradresse überwachen, er wird eine „Stichprobe“ machen und nur eine Speicherzelle mit cas überwachen. Diese Zelle wird jedoch mit demselben Wert beschrieben. Auf cas-Basis wäre also nicht entscheidbar, ob der gesamte Speicherbereich aktualisiert worden ist.

Lösung Ein Adressenreservierungsschema wie bei ll/sc (→ p. 27)

Compare and Exchange (1) cmpxchg

```
int cmpxchg (int& ref, int val, register int eax) {
    register char srZ;
    <atomic>
    if (srZ = (eax == ref)) ref = val;
    else eax = ref;
    <cimota>
    return srZ;
}
```

Compare and Exchange (2) cmpxchg

```
void atomicCounter::raise () {
    asm volatile ("movl    %0,%%eax"       : : "m" (item) : "eax" );
    asm volatile ("0:");
    asm volatile ("leal    1(%%eax),%%edx" : : : "edx" );
    asm volatile ("cmpxchg %%edx,%0"       : "=m" (item));
    asm volatile ("jne 0b");
}
```


Load Linked and Store Conditional (1) ll/sc

```
register int* want;
register int  data;
```

```
int ll (int& ref) {
    <atomic>
    want = &ref;
    int aux = data = ref;
    <cimota>
    return aux;
}
```

```
int sc (int& ref, int val) {
    register short srZ;
    <atomic>
    if (want != &ref) srZ = 0;
    else if (data != ref) srZ = 0;
    else {
        ref = val;
        want = 0;
        srZ = 1;
    }
    <cimota>
    return srZ;
}
```

Load Linked and Store Conditional (2) ll/sc

```
int atomicCounter::operator ++ (int) {
    int aux;
    do aux = ll(item) + 1;
    while (!sc(item, aux));
    return aux;
}
```

```
int atomicCounter::operator += (int i) {
    int aux;
    do aux = ll(item) + i;
    while (!sc(item, aux));
    return aux;
}
```


Diskussion

Wer beeinflusst wen wie und welche Probleme tauchen auf?

Elop	Prozess		Problem
	verzögernde	*verzögerte*	
cli	laufende	unterbrechende	Blockade
cas cmpxchg ll/sc	unterbrechende	unterbrochene	Wiederholung(en)

Welche Alternative gibt es?

- Verfahren, die keine speziellen Elementaroperationen der Ebene$_2$ erfordern

Zusammenfassung

- Synchronisation ist die Koordination von Kooperation und Konkurrenz
  - Prozesse stehen im „Wettstreit“ um die Zuteilung von Betriebsmitteln
- Interrupts bewirken asynchrone Programmabläufe und lösen Prozesse aus
  - der „Interrupt-Prozess“ muss jedoch nicht zwingend als Inkarnation vorliegen
- die Synchronisation asynchroner Programmunterbrechungen erfolgt einseitig
  - entweder der unterbrechende oder der unterbrochene Prozess wird verzögert
- die Verzögerung kann blockierend oder nicht-blockierend realisiert sein

Referenzen

[1] R. G. Herrtwich and G. Hommel. *Kooperation und Konkurrenz — Nebenläufige, verteilte und echtzeitabhängige Programmsysteme*. Springer-Verlag, 1989. ISBN 3-540-51701-4.

[2] K.-P. Löhr. Nichtsequentielle Programmierung. http://www.inf.fu-berlin.de/inst/ag-ss, 2002. Vorlesungsskript.

[3] A. Micklei and A. Fetke. Non-Blocking Synchronization. http://www.first.gmd.de/~fs/peace-pro97/non-blocking-sync/, 1997. Seminarausarbeitung.

[4] D. C. Schmidt. Strategized Locking, Thread-safe Decorator, and Scoped Locking: Patterns and Idioms for Simplifying Multi-threaded C++ Components. *C++ Report*, 11(9), Sept. 1999.

[5] W. Schröder-Preikschat. *The Logical Design of Parallel Operating Systems*. Prentice Hall International, 1994. ISBN 0-13-183369-3.