Koordination nebenläufiger Prozesse

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.

Quelle: Herrtwich/Hommel (1989), *Kooperation und Konkurrenz*, S. 26

Koordinierung ≡ „Reihenschaltung"

ko·or·di'nie·ren beiordnen; in ein Gefüge einbauen; aufeinander abstimmen; nebeneinanderstellen; Termine ∼.

- als kritisch erachtete nebenläufige Aktivitäten *der Reihe nach* ausführen
  ☞ überlappendes Zählen (✗ Kap. 5)
  ☞ verdrängende Prozesseinplanung (✗ Kap. 6)
- „der Reihe nach" bedeutet, gewisse Prozesse bewusst zeitlich zu verzögern
  – je nach Verfahren trifft es den $\left\{ \begin{array}{c} \text{überlappenden} \\ \text{überlappten} \end{array} \right\}$ Prozess
- die Verfahren arbeiten {,nicht-}wartend bzw. {,nicht-}blockierend

Arten der Synchronisation

- „Koordination der Kooperation und Konkurrenz zwischen Prozessen" . . .
  1. *derselben Inkarnation* ☞ Intraprozess-Synchronisation
     – nicht-blockierende Synchronisation ist zwingend
     – Beispiel: asynchrone Programmunterbrechung (*interrupt*)
  2. *verschiedener Inkarnationen* ☞ Interprozess-Synchronisation
     – {,nicht-}blockierende Synchronisation ist verwendbar

☞ Differenzierung: *ein- und mehrseitige Synchronisation* nebenläufiger Prozesse

Einseitige Synchronisation — Unilateral

- die Verfahren wirken 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[47]

[47] Die Veränderung einer Bedingung, auf die ein Prozess wartet, ist z. B. von einem anderen Prozess herbeizuführen.

Mehrseitige Synchronisation — Multilateral

- die Verfahren wirken sich auf (ggf.) alle beteiligten Prozesse aus
  - welche Prozesse im weiteren Ablauf verzögert werden, ist i. A. unvorhersehbar
  - allgemein gilt: „wer zuerst kommt, mahlt zuerst“
- die betroffenen Aktivitäten stehen miteinander im **gegenseitigen Ausschluss**
  - erzwungen wird die *atomare Ausführung* von Anweisungsfolgen
  - „abschnittweise“ werden diese niemals nebenläufig/parallel durchgeführt
- die atomar ausgeführten Anweisungsfolgen bilden eine *Elementaroperation*

Asynchrone Programmunterbrechungen (✗ Kap. 5)

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

  Verzögerung des unterbrechenden Prozesses
  - weiche/harte Synchronisation der „Hardware/Software-Interrupts“
  - logischer Ansatz („Schleusen“ [28]) bzw. Ebene $_2$-Befehle: `cli`, `sti` (x86)

  Verzögerung des unterbrochenen Pozesses
  - Ebene $_2$-Befehle:
    - ☞ `cas` (IBM 370, m68020+), `cmpxchg` (i486+) — CISC
    - ☞ `ll`/`sc` (DEC Alpha, MIPS, PowerPC) — RISC
  - nicht-blockierende Synchronisation nebenläufiger Aktivitäten
- die Verfahren synchronisieren *einseitig*, d. h. sie arbeiten *unilateral*

Verzögerung des unterbrechenden Prozesses

```
int wheel = 0;

void __attribute__ ((interrupt)) tip () {
    wheel += 1;
}

int main () {
    for (;;)
        printf("%d\n", incr(&wheel));
}
```

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

blockierende Synchronisation Interrupts werden zeitweilig unterbunden

Verzögerung des unterbrochenen Prozesses

```
int wheel = 0;

void __attribute__ ((interrupt)) tip () {
    incr(&wheel);
}

int main () {
    for (;;)
        printf("%d\n", incr(&wheel));
}
```

```
int incr (int *ref) {
    int x;
    do x = *p;
    while (!cas(p, x, x + 1));
    return x + 1;
}
```

nicht-blockierende Synchronisation Wiederholung der Berechnung findet statt, wenn nebenläufig eine andere Aktivität erfolgreich beendet wurde

Spezialbefehl

```
bool cas (word *ref, word old, word new) {
    bool srZ;
    atomic();
    if (srZ = (*ref == old)) *ref = new;
    cimota();
    return srZ;
}
```

CAS (*compare and swap*)

- unteilbare Operation
  - *read-modify-write*
- Komplexbefehl ☞ CISC

Ist blockierungsfrei nur in Bezug auf Prozesse!

Multiprozessor-Synchronisation (gemeinsamer Speicher, *shared memory*)

`atomic()` verhindert (Speicher-) Buszugriffe durch andere Prozessoren
`cimota()` lässt (Speicher-) Buszugriffe anderer Prozessoren wieder zu

☞ Auf Interrupts wird, wie sonst auch, erst am Befehlsende reagiert!

Gegenseitiger Ausschluss — *mutual exclusion*

- ein Ansatz, der kennzeichnend ist für die *mehrseitige Synchronisation*:

  Sich gegenseitig ausschließende Aktivitäten werden nie parallel ausgeführt und verhalten sich zueinander, als seien sie unteilbar, weil keine Aktivität die andere unterbricht.

  Anweisungen, deren Ausführung einen gegenseitigen Ausschluß erfordert, heißen **kritische Abschnitte** (*critical sections*, *critical regions*).

  Quelle: Herrtwich/Hommel (1989), *Kooperation und Konkurrenz*, S. 137

- die kritischen Abschnitte sind durch „Synchronisationsklammern“ zu schützen

Kritischer Abschnitt

Beim Betreten (*enter*) und Verlassen (*leave*) gelten bestimmte Vorgehensweisen:

Eintrittsprotokoll (*entry protocol*)

- regelt die Belegung eines kritischen Abschnitts durch einen Prozess
  - erteilt einem Prozess die *Zugangsberechtigung*
- bei bereits belegtem kritischen Abschnitt wird der Prozess verzögert

Austrittsprotokoll (*exit protocol*)

- regelt die Freigabe des kritischen Abschnitts durch einen Prozess
- andere erhalten die Möglichkeit zum Betreten des kritischen Abschnitts

Die Vorgehensweisen variieren mit dem realisierten Synchronisationsverfahren.

Schlossvariable — *lock variable*

Ein *abstrakter Datentyp*, auf dem zwei Operationen definiert sind:

acquire (*lock*) ☞ Eintrittsprotokoll

- verzögert einen Prozess, bis das zugehörige Schloss offen ist
  - bei bereits geöffnetem Schloss fährt der Prozess unverzögert fort
- verschließt das Schloss („von innen“), wenn es offen ist

release (*unlock*) ☞ Austrittsprotokoll

- öffnet das zugehörige Schloss, ohne den öffnenden Prozess zu verzögern

Implementierungen werden als **Schlossalgorithmen** (*lock algorithms*) bezeichnet.

Schlossalgorithmus (1) — Prinzip mit Problem(en)

```
typedef char bool;

void acquire (bool *lock) {
    while (*lock);
    *lock = 1;
}

void release (bool *lock) {
    *lock = 0;
}
```

`acquire()` soll einen kritischen Abschnitt schützen, ist dabei aber selbst kritisch:

- Problem macht die Phase vom Verlassen der Kopfschleife (`while`) bis zum Setzen der Schlossvariablen
- Verdrängung des laufenden Prozesses kann einem anderen Prozess ebenfalls das Schloss geöffnet vorfinden lassen

- im weiteren Verlauf könnten (mindestens) zwei Prozesse den eigentlichen, durch `acquire()` zu schützenden kritischen Abschnitt überlappt ausführen

Schlossalgorithmus (2) — Unterbrechungssteuerung

```
void acquire (bool *lock) {
    avertIRQ();
    while (*lock) {
        admitIRQ();
        avertIRQ();
    }
    *lock = 1;
    admitIRQ();
}
```

```
void avertIRQ () { asm("cli"); }
void admitIRQ () { asm("sti"); }
```

- Überprüfen und Schließen des Schlosses bilden eine ununterbrechbare Anweisungsfolge
- die Schleife muss unterbrechbar sein, damit das Schloss aufgeschlossen werden kann

- asynchrone Programmunterbrechungen werden abgewendet, obwohl diese nie den durch `acquire()` geschützten kritischen Bereich betreten dürfen

Schlossalgorithmus (3) — Verdrängungssteuerung

```
void acquire (bool *lock) {
    avert();
    while (*lock) {
        admit();
        avert();
    }
    *lock = 1;
    admit();
}
```

```
void avert () { preemption = 0; }
void admit () { preemption = 1; }
```

- Überprüfen und Schließen des Schlosses bilden eine überlappungsfreie Anweisungsfolge
- die Schleife muss überlappbar sein, damit das Schloss aufgeschlossen werden kann

- Verdrängung des Prozesses wird abgewendet, obwohl ggf. nur einer von vielen lauffähigen Prozessen das Schloss öffnen wird

Schlossalgorithmus (4) — Komplexbefehl

```
void acquire (bool *lock) {
    while (tas(lock));
}
```

```
bool tas (bool *flag) {
    bool old;
    atomic();
    old = *flag;
    *flag = 1;
    cimota();
    return old;
}
```

TAS (*test and set*)

- atomarer Lese-Modifikations-Schreibzyklus
  - *read-modify-write*
- geeignet für Ein- *und* Mehrprozessorsysteme

☞ bei der Befehlsausführung wird kein anderer $\left\{\begin{array}{l}\text{Befehl abgearbeitet}\\ \text{Speicherzugriff durchgeführt}\end{array}\right.$

Aktives Warten — *busy waiting*

- Unzulänglichkeit der Schlossalgorithmen: der aktiv wartende Prozess . . .
  - kann selbst keine Änderung der Bedingung herbeiführen, auf die er wartet
  - behindert daher unnütz andere Prozesse, die sinnvolle Arbeit leisten könnten
  - schadet damit letztlich auch sich selbst:

    Je länger der Prozess den Prozessor für sich behält, umso länger muss er darauf warten, dass andere Prozesse die Bedingung erfüllen, auf die er selbst wartet.

- die dadurch entstehenden *Effizienzeinußen* sind nur dann unproblematisch, wenn jedem Prozess ein eigener realer Prozessor zur Verfügung steht

Passives Warten

- Prozesse geben die Kontrolle über die CPU ab während sie Ereignisse erwarten
  - im Synchronisationsfall blockiert sich ein Prozess auf ein Ereignis
    ☞ ggf. wird der PD des Prozesses in eine Warteschlange eingereiht
  - tritt das Ereignis ein, wird ein darauf wartender Prozess deblockiert
- die *Wartephase* eines Prozesses ist als *Blockadephase* („E/A-Stoß") ausgelegt
  - ggf. wird der Ablaufplan für die Prozesse aktualisiert (*scheduling*)
  - ein anderer, lauffähiger Prozess wird plangemäß abgefertigt (*dispatching*)
  - ist kein Prozess mehr lauffähig, läuft die CPU „leer" (*idle phase*)
- mit Beginn der Blockadephase eines Prozesses endet auch sein CPU-Stoß

Schlossalgorithmus (5) — Blockadephase

```
void acquire (bool *lock) {
    while (tas(lock))
        sleep(lock);
}

void release (bool *lock) {
    *lock = 0;
    awake(lock);
}
```

```
void sleep (void *flag) {
    racer()->wait = flag;
    block();
}

void awake (void *flag) {
    unsigned next;
    for (next = 0; next < NTASK; next++)
        if (task[next].wait == flag) {
            task[next].wait = 0;
            ready(&task[next]);
        }
}
```

☞ ***race condition*!**

Schlossalgorithmus (6) — Bedingungsvariable

```
void acquire (bool *lock) {
    avert();
    while (tas(lock))
        sleep(lock);
    admit();
}
```

```
void sleep (void *flag) {
    racer()->wait = flag;
    admit();
    block();
    avert();
}
```

- Verdrängung des laufenden Prozesses innerhalb der Kopfschleife (`acquire()`) bis zum Setzen der Wartebedingung (`sleep()`) wird abgewendet
- einer möglichen Überlappung von `acquire()` mit `release()` und der ggf. anhaltenden Blockade eines sich schlafenlegenden Prozesses wird vorgebeugt

☞ warten innherhalb kritischer Abschnitte bei gleichzeitiger Abschnittsfreigabe