

Systemprogrammierung

Synchronisation

14./20./21. Januar 2010

Überblick

Synchronisation

Konkurrenz und Koordination

Verfahrensweisen

Ereignissteuerung

Blockierende Synchronisation

Schlossvariable

Bedingungsvariable

Semaphor

Monitor

Nicht-blockierende Synchronisation

Zusammenfassung

Bibliographie

Nebenläufige Aktivitäten

Nichtsequentielles Programm

Nebenläufigkeit (engl. *concurrency*) bezeichnet das Verhältnis von nicht kausal abhängigen Ereignissen, die sich also nicht beeinflussen

- ▶ Aktionen können nebenläufig ausgeführt werden, wenn keine das Resultat des anderen benötigt

```

1:   foo = 4711;
2:   bar = 42;
3:  foobar = foo + bar;
4:  barfoo = bar + foo;
5:   hal = foobar + barfoo;

```

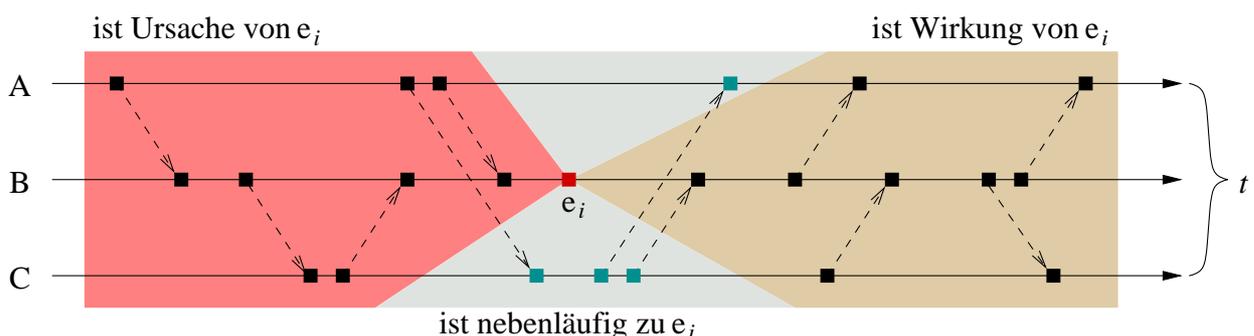
- ▶ Zeile 1 kann nebenläufig zu Zeile 2 ausgeführt werden
- ▶ Zeile 3 kann nebenläufig zu Zeile 4 ausgeführt werden

Kausalität (lat. *causa*: Ursache) ist die Beziehung zwischen **Ursache** und **Wirkung**, d.h., die ursächliche Verbindung zweier Ereignisse

- ▶ Ereignisse sind nebenläufig, wenn keines Ursache des anderen ist

Kausalordnung

Nebenläufigkeit als relativistischer Begriff von Gleichzeitigkeit



- ▶ ein Ereignis **ist nebenläufig zu** einem anderen (e_i), wenn es im **Anderswo** des anderen Ereignisses (e_i) liegt
 - ▶ d.h., weder in der Zukunft noch in der Vergangenheit des anderen
- ▶ das Ereignis ist nicht Ursache/Wirkung des anderen Ereignisses (e_i)
 - ▶ ggf. aber Ursache/Wirkung anderer (von e_i verschiedener) Ereignisse

Kausalordnung (Forts.)

Rangfolge aus Gründen von Daten- und Zeitabhängigkeit

Um selbst ein korrektes Ergebnis zu produzieren, können Aktionen nebenläufig stattfinden, sofern. . .

allgemein keine das Resultat der anderen benötigt (S. 11-2)

- ▶ **Datenabhängigkeiten** gleichzeitiger Prozesse beachten

speziell (zusätzlich im Echtzeitbetrieb) keine die **Zeitbedingungen** der anderen verletzt \rightsquigarrow EZS (S. 1-3)

- ▶ Zeitpunkte dürfen nicht/nur selten verpasst werden
- ▶ Zeitintervalle dürfen nicht/nur begrenzt gedehnt werden

Umgang mit Ereignissen bzw. Aktionen gleichzeitiger Prozesse

„ist Ursache von“
 „ist Wirkung von“
 „ist nebenläufig zu“

$\left. \begin{array}{l} \text{„ist Ursache von“} \\ \text{„ist Wirkung von“} \end{array} \right\} \rightsquigarrow \text{Koordinierung (vor/zur Laufzeit)}$
 $\models \text{Parallelität (implizit)}$

Koordinierung

Reihenschaltung gleichzeitiger Aktivitäten

Maßnahme zum korrekten Zugriff gleichzeitiger Prozesse auf gemeinsame aber unteilbare Betriebsmittel \rightsquigarrow Synchronisation

- ▶ blockierend, wenn das Betriebsmittel nicht die CPU ist
- ▶ möglicherweise nicht-blockierend, sonst. . .
 - ▶ kritische Abschnitte

Synchronisation (gr. *sýn*: zusammen, *chrónos*: Zeit) bezeichnet das „Herstellen von Gleichzeitigkeit“

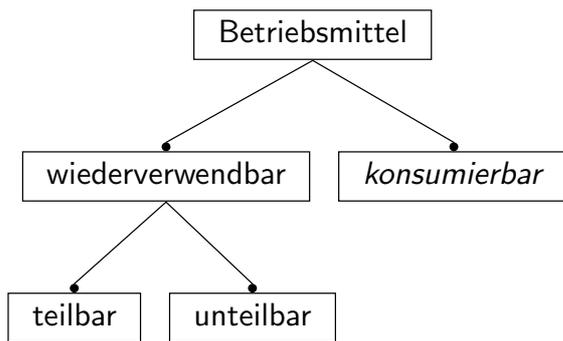
- ▶ Koordination der Kooperation und Konkurrenz zwischen Prozessen
- ▶ Abgleich von Echtzeituhren (oder Daten) in verteilten Systemen
- ▶ Sequentialisierung von Ereignissen entlang einer Kausalordnung
 - ▶ z.B. logische Uhren

☞ analytisch (Einplanung) oder **konstruktiv** (Implementierung)

Konkurrenz

Betriebsmittel und Betriebsmittelarten

(vgl. S. 11-43)



Hardware

CPU, Speicher, Geräte, *Signale*

Software

Dateien, Prozesse, Seiten,
Puffer, *Signale, Nachrichten*

Wettbewerb um Betriebsmittel (engl. *resource contention*) bezieht sich auf Anzahl und Art eines Betriebsmittels

einseitige Synchronisation \mapsto konsumierbare Betriebsmittel

mehrseitige Synchronisation \mapsto wiederverwendbare Betriebsmittel

- ▶ Begrenzung, gegenseitiger Ausschluss

Konkurrenz (Forts.)

Reihenschaltung von Prozessen mit begrenzten/unteilbaren Betriebsmitteln

Betriebsmittel, die unteilbar sind, können von gleichzeitigen Prozessen nur nacheinander belegt werden

Vergabe \mapsto das Betriebsmittel sperren und dem Prozess zuteilen

- ▶ beim Versuch, ein gesperrtes Betriebsmittel erneut zu belegen, wird der anfordernde Prozess blockiert
- ▶ der blockierende Prozess erwartet das Ereignis/Signal zur Freigabe des gesperrten Betriebsmittels, ihm wird die CPU entzogen

Freigabe \mapsto das Betriebsmittel dem Prozess wieder entziehen

- ▶ sollten Prozesse die Freigabe dieses Betriebsmittels erwarten, wird es sofort der **Wiedervergabe** zugeführt; das bedeutet:
 - das Betriebsmittel entsperren und alle Prozesse deblockieren, die sich dann wiederholt um die Vergabe zu bemühen haben *oder*
 - einen Prozess auswählen und ihm das Betriebsmittel zuteilen
- ▶ nur der das Betriebsmittel „besitzende“ Prozess kann es freigeben

Konfliktsituation

Blockierung gleichzeitiger Prozesse

Prozesse befinden sich untereinander im **Konflikt**, wenn...

- ▶ nur eine begrenzte Anzahl gemeinsamer Betriebsmitteln vorrätig ist
- ▶ sie unteilbare Betriebsmittel desselben Typs gemeinsam verwenden

Prozesse sind im **Streit** (engl. *contention*) um ein Betriebsmittel, wenn einer das Betriebsmittel anfordert, das ein anderer bereits besitzt

- ▶ der anfordernde Prozess blockiert und wartet auf die Freigabe des Betriebsmittels durch den Prozess, der das Betriebsmittel belegt
- ▶ der das Betriebsmittel belegende Prozess löst den auf die Freigabe des Betriebsmittels wartenden Prozess aus, deblockiert in wieder

☞ Nutzung begrenzter/unteilbarer Betriebsmittel impliziert **Kooperation**

Koordinierung \equiv „Reihenschaltung“

Koordination der Kooperation und Konkurrenz zwischen Prozessen \rightsquigarrow Synchronisation

ko-or-di'nie-ren bei**ordnen**; in ein Gefüge einbauen; aufeinander abstimmen; nebeneinanderstellen; Termine \sim .

- ▶ sich überlappen könnende Aktivitäten (gleichzeitige Prozesse) **der Reihe nach** ausführen; vgl. S. 7-77: **kritischer Abschnitt**
 - ▶ überlapptes Zählen (asynchrone Programmunterbrechung)
 - ▶ Verwaltung der Bereitliste (verdrängende Prozesseinplanung)
- ▶ „der Reihe nach“ \rightsquigarrow die Verzögerung von Prozessen erzwingen
 - ▶ die überlappende oder die überlappte Aktivität, je nach Verfahren

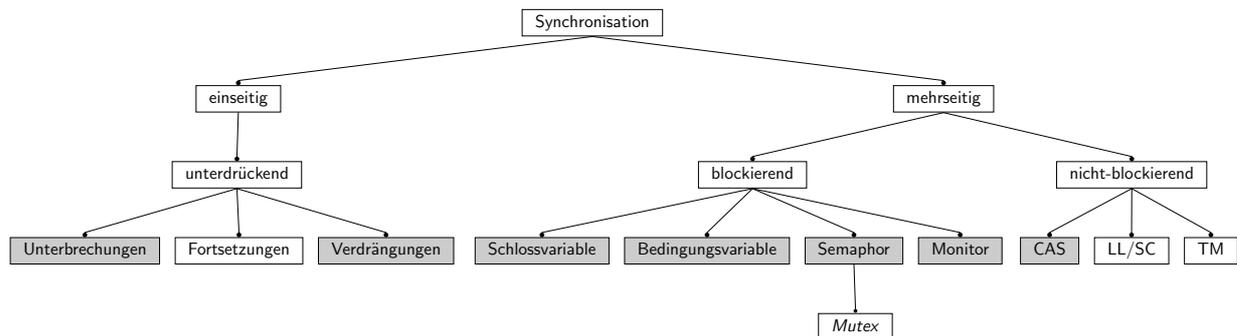
Synchronisationsverfahren...

- ▶ wirken einseitig oder mehrseitig
 - ▶ unterdrückend, blockierend, nicht-blockierend
 - ▶ behinderungs-, sperr-, wartefrei



Arten der Synchronisation

Klassifikation



Synchronisationsverfahren, die in SP betrachtet werden, arbeiten...

einseitig unterdrückend: Ereignisse (Unterbrechungen, Verdrängungen)
mehrseitig blockierend bzw. nicht-blockierend: Prozessinstanzen

Einseitige Synchronisation

Unilateral

Auswirkung haben die Verfahren nur auf einen der beteiligten Prozesse:

Bedingungsynchronisation

- ▶ der Ablauf 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 „Rollenspiel“ der beteiligten Prozesse

- ▶ andere Prozesse sind jedoch nicht gänzlich unbeteiligt:
 - ▶ die Veränderung einer Bedingung, auf die ein Prozess wartet, ist z.B. von einem anderen Prozess herbeizuführen

Mehrseitige Synchronisation

Multilateral

Auswirkung haben die Verfahren ggf. auf alle beteiligten Prozesse:

- ▶ welche Prozesse verzögert werden, ist i.A. unvorhersehbar
- ▶ allgemein gilt: „wer zuerst kommt, mahlt zuerst“
 - ▶ d.h., schreitet (relativ) unverzögert in seiner weiteren Ausführung fort

Beispiel: gegenseitiger Ausschluss (engl. *mutual exclusion*)

- ▶ erzwungene **sequentielle Ausführung** von Anweisungsfolgen
- ▶ im Regelfall zeitlich begrenzte, exklusive Betriebsmittelvergabe

Beispiel: kritischer Abschnitt (engl. *critical section/region*)

- wettlaufintolerant ▶ gegenseitiger Ausschluss
- wettlauftolerant ▶ nichtsequentielle Ausführung
- ▶ Kollisionserkennung/-behandlung

☞ Modularisierung der Anweisungsfolgen schafft **Elementaroperationen**

Betriebsmittelart und Synchronisationstechnik

Artenspezifische Verfahren

Notwendigkeit des Ausschlusses gleichzeitiger Prozesse ist immer nur in folgenden Fällen gegeben \leadsto **blockierende Synchronisation**:

- ▶ beim Zugriff auf unteilbare wiederverwendbare Betriebsmittel
- ▶ bei Inanspruchnahme eines konsumierbaren Betriebsmittels¹

Schutz kritischer Abschnitte bedingt sich in anderer Weise und ist auf zwei verschiedenen Wegen möglich:

- pessimistisch** \leftrightarrow **gegenseitiger Ausschluss** (block. Synchronisation) —
- ▶ KA implementiert als sequentielles Programm
 - ▶ entsprechend der „klassischen Definition“ (S. 11-14)
- optimistisch** \leftrightarrow Kollisionserkennung und -behandlung +
- ▶ KA implementiert als *nichtsequentielles Programm*
 - ▶ Prinzip: **nicht-blockierende Synchronisation**

¹Nämlich wenn der konsumierende Prozess auf die Betriebsmittelbereitstellung durch den produzierenden Prozess warten muss.

Kritischer Abschnitt (KA)

Kennzeichnend für mehrseitige Synchronisation

... herkömmliche Sichtweise, z.B. nach [1, S. 137]

- ▶ sich gegenseitig ausschließende Aktivitäten
 - ▶ werden nie parallel ausgeführt
 - ▶ verhalten sich zueinander, als seien sie unteilbar, weil keine Aktivität die andere unterbricht
- ▶ Anweisungen, deren Ausführung gegenseitigen Ausschluss erfordert

Beachte \leftrightarrow Abstraktion eines KA als „Elementaroperation“ (ELOP)

- ▶ ein gegenseitiger Ausschluss ist nicht zwingend, um einen kritischen Abschnitt zu schützen
- ▶ vielmehr gilt es sicherzustellen, dass die Ausführung eines solchen Abschnitts jederzeit ein **konsistentes Ergebnis** liefert

Kritischer Abschnitt (Forts.)

Protokolle regeln den Ein- und Austritt

Betreten (engl. *enter*) und Verlassen (engl. *leave*) kritischer Abschnitte unterliegen bestimmten **Verhaltensregeln**:

Eintrittsprotokoll (engl. *entry protocol*)

- ▶ regelt die Belegung des kritischen Abschnitts durch einen Prozess
 - ▶ erteilt einem Prozess die **Zugangsberechtigung**
- ▶ bei bereits belegtem Abschnitt: **Wettstreitigkeit** (engl. *contention*)
 - ▶ Eintritt begehrenden Prozess verzögern („pessimistisch sein“) oder
 - ▶ Abschnitt belegenden Prozess zurücksetzen („optimistisch sein“)

Austrittsprotokoll (engl. *exit protocol*)

- ▶ regelt die Freigabe des kritischen Abschnitts durch einen Prozess
- ▶ Prozesse können den kritischen Abschnitt (wieder) belegen

 die Vorgehensweisen variieren mit dem jew. Synchronisationsverfahren

Fortschrittsgarantien

Aussagen zur **Lebendigkeit** (engl. *liveliness*) nichtsequentieller Programme

behinderungsfrei (engl. *obstruction-free*)

- ▶ ein einzelner, in Isolation ablaufender Faden wird seine Operation in begrenzter Anzahl von Schritten beenden
- ▶ ein Faden läuft in Isolation ab, wenn alle ihn behindern könnenden anderen Fäden in der Ausführung zurückgestellt sind

sperrfrei (engl. *lock-free*), umfasst Behinderungsfreiheit

- ▶ jeder bei Ablauf eines Fadens auszuführende Schritt trägt dazu bei, dass das nichtsequentielle Programm insgesamt voranschreitet
- ▶ garantiert systemweiten Durchsatz, erlaubt jedoch **Aushungerung** (engl. *starvation*) eines einzelnen Fadens

wartefrei (engl. *wait-free*), umfasst Sperrfreiheit

- ▶ die Anzahl der zur Beendigung einer Operation bei Fadenabläufen auszuführenden Schritte ist begrenzt
- ▶ garantiert systemweiten Durchsatz und ist frei von Aushungerung

Unterbrechungssteuerung

Kontrollierter Ablauf asynchroner Programmunterbrechungen

Ansatz: asynchrone Programmunterbrechungen entweder verhindern oder tolerieren, und zwar durch **Verzögerung der...**

überlappenden Aktivität \mapsto pessimistisches Verfahren

- ▶ Spezialbefehle der $\left\{ \begin{array}{l} \text{Ebene}_2: \text{ cli, sti (x86)} \\ \text{Ebene}_3: \text{ sigprocmask (POSIX)} \end{array} \right\}$ **hart**

überlappten Aktivität \mapsto optimistisches Verfahren

- ▶ durch Spezialbefehle und Programme der Ebene₂:
CISC \leftrightarrow CAS (IBM 370, m68020+), CMPXCHG (i486+)
RISC \leftrightarrow LL/SC (DEC Alpha, MIPS, PowerPC)
- ▶ ohne Spezialbefehle \rightsquigarrow BS/BST (S. 1-3) **weich**

Unterbrechungen sperren: einfach — aber nicht immer zweckmäßig

- ▶ Faustregel: harte Synchronisation ist möglichst zu vermeiden

Unterbrechungssteuerung (Forts.)

Wiedersehen mit einem alten Problem (S. 5-53): überlapptes Zählen

```
unsigned int wheel = 0;
```

Plötzlich...

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

Schlecht...

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

Wettlaufsituation

- ▶ kritischer Abschnitt ++
- ▶ Laufgefahr vorbeugen
- ▶ ELOP `int incr (int*)`
- ▶ unteilbares Zählen konstruktiv und problemadäquat sicherstellen

Besser...

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

Unterbrechungssteuerung (Forts.)

Verhinderung vs. Tolerierung von asynchronen Programmunterbrechungen

Verhinderung

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

- ▶ Komplexbefehl der Ebene 3
- ▶ **Unterbrechungssperre**
- ▶ ggf. droht Ereignisverlust

Tolerierung

```
int incr (int *ip) {
    asm ("incl %0"
        : "=m" (*ip)
        : "m" (*ip)
        : "memory");
    return *ip;
}
```

- ▶ Komplexbefehl der Ebene 2
- ▶ **Elementaroperation** der CPU
- ▶ „unterbrechungstransparent“

Achtung ~ Multiprozessorbetrieb

- ▶ die Befehle haben nur lokale Signifikanz für „ihren“ Prozessor(kern)

Verdrängungssteuerung

Verdrängungsfreie kritische Abschnitte

NPCS (Abk. für engl. *non-preemptive critical section*)

- ▶ Einplanung von Fäden läuft (nahezu) wie gewöhnlich durch
 - ▶ durch externe Prozesse ausgelöste Fäden kommen auf die Bereitliste
 - ▶ der laufende Faden bekommt jedoch nicht den Prozessor entzogen
- ▶ nur die Einlastung von Fäden wird zeitweilig ausgesetzt
 - ▶ Verdrängungsereignisse werden zeitversetzt behandelt
 - ▶ d.h., fremd- wie auch selbstverursachte Verdrängungen²
- ▶ zeitweilige Monopolisierung des Prozessors durch einen Faden

```
int npcs = 0;
```

```
void npcs_enter () { npcs = 1; }
void npcs_leave () { npcs = 0; }
```

```
/* Timeslice handler... */
void npcs_check () {
    if (npcs == 0) preempt();
}
```

²Eine Verdrängung ist selbstverursacht, wenn der laufende Faden einen Faden höherer Priorität als er selbst bereitstellt. Sie ist fremdverursacht, wenn ein anderer (ggf. externer) Prozess den laufenden Faden dazu zwingt, den Prozessor abzugeben.

Schlossvariable

(engl. *lock variable*)

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

acquire (auch: *lock*) \models Eintrittsprotokoll

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

release (auch: *unlock*) \models Austrittsprotokoll

- ▶ öffnet ein Schloss, ohne den öffnenden Prozess zu verzögern

Implementierungen dieses Konzepts werden auch als **Schlossalgorithmen** (engl. *lock algorithms*) bezeichnet

Schlossalgorithmus

Prinzip — mit Problem(en)

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

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

- ▶ kritisch ist die Phase vom Verlassen der Kopfschleife bis zum Setzen der Schlossvariablen
- ▶ Verdrängung des laufenden Prozesses kann einem anderen Prozess ebenfalls das Schloss geöffnet vorfinden lassen

- ▶ im weiteren Verlauf könnten (min.) zwei Prozesse den eigentlichen zu schützenden kritischen Abschnitt überlappt ausführen

Schutz eines kritischen Abschnitts bildet selbst einen kritischen Abschnitt:

acquire muss als Elementaroperation implementiert sein

- ▶ das Eintrittsprotokoll muss unteilbar ausgeführt werden

Schlossalgorithmus (Forts.)

Totale Unterbrechungssteuerung: Steuerung der Ebene₂ durch die Ebene₃

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

(IRQ Abk. engl. *interrupt request*)

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

- ▶ Überprüfen und Schließen des Schlosses verläuft unteilbar
 - ▶ nur bei Monoprozessorsystemen

- ▶ der Schleifenrumpf muss jedoch teilbar sein, damit der Planer aufgerufen werden und ggf. eine Umplanung vornehmen kann

Interrupts werden abgewendet, obwohl im Zuge ihrer Behandlung der überlapppte Prozess nie einen Schlossalgorithmus durchlaufen dürfte

- ▶ darüberhinaus können **flankengesteuerte *Interrupts*** verloren gehen

Schlossalgorithmus (Forts.)

Totale Verdrängungssteuerung: Steuerung der Ebene₃ durch die Ebene₃

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

(PRQ Abk. engl. *preemption request*)

```
void avertPRQ () { npcs_enter(); }
void admitPRQ () { npcs_leave(); }
```

- ▶ Überprüfen und Schließen des Schlosses verläuft unteilbar
 - ▶ nur bei Monoprozessorsystemen

- ▶ der Schleifenrumpf muss jedoch teilbar sein, damit der laufende Prozess verdrängt werden kann

Verdrängungsereignisse werden unnötigerweise abgewendet auch für die Prozesse, die den KA nie durchlaufen

- ▶ darüberhinaus arbeitet das System **nicht voll verdrängend**

Schlossalgorithmus (Forts.)

Spezialbefehl des Prozessors

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

- ▶ Überprüfen und Schließen des Schlosses verläuft unteilbar

tas (*test and set*) testet den Inhalt der adressierten Speicherzelle und setzt ihren Wert auf 1, wenn der Wert 0 ist:

- ▶ `return *lock ? 1 : !(*lock = 1);`
- ▶ atomarer Maschinenbefehl für Ein- oder Mehrprozessorsysteme

„Dreh Schloss“, Umlaufsperr (engl. *spin lock*): **Vorsicht...**

- ▶ pausenloses Schleifen hindert andere Prozessoren am Buszugang
 - ▶ im Schleifenrumpf ist eine Pause einzulegen — nur wie lange?
- ▶ starke Leistungseinbußen können die Folge sein [2]

Bedingtes Flaggesetzen (engl. *test and set*, TAS)

Spezialbefehl zur Multiprozessorsynchronisation

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

Komplexbefehl (einer CPU), der den aktuellen Wert einer Schlossvariablen liefert und diese (auf 1) setzt:

true \models das Schloss ist bereits verschlossen

▶ die Schlossvariable ist unverändert

false \models das Schloss wurde verschlossen

▶ die Schlossvariable wurde verändert

Elementaroperation³ eines CISC, **atomarer „read-modify-write“-Zyklus**: `atomic()` verhindert (Speicher-) Buszugriffe durch andere Prozessoren `cimota()` lässt (Speicher-) Buszugriffe anderer Prozessoren wieder zu. Lese-/Schreibzyklen des Prozessors werden unteilbar ausgeführt

▶ auf *Interrupts* wird, wie sonst auch, erst am Befehlsende reagiert

³Typischerweise ist diese Operation ein Befehl der Ebene₂ und nicht, wie hier zum besseren Verständnis gezeigt, als Programm der Ebene₅ implementiert.

Aktives Warten

(engl. *busy waiting*)

Unzulänglichkeit der Schlossalgorithmen: der aktiv wartende Prozess...

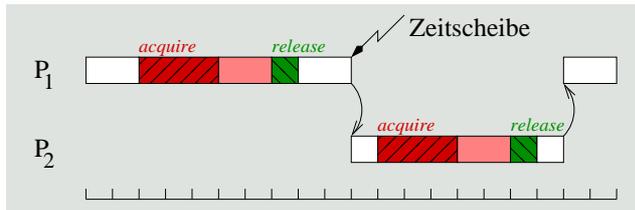
- ▶ kann keine Änderung der Bedingung herbeiführen, auf die er wartet
- ▶ behindert andere Prozesse, die sinnvolle Arbeit leisten könnten
- ▶ schadet damit letztlich auch sich selbst

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.

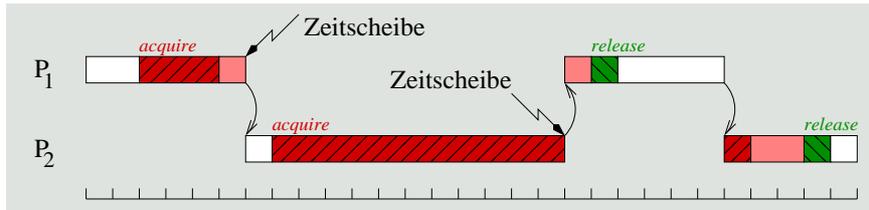
- ▶ in den meisten Fällen sind Effizienzeinbußen in Kauf zu nehmen
 - ▶ es sei denn, jeder Prozess hat seinen eigenen realen Prozessor
 - ▶ nicht unrealistisch: „gang scheduling“ und „barrier synchronization“

Aktives Warten ohne Prozessorabgabe

„Spin locking considered harmful“



	T_s	T_q	T_q/T_s
P_1	12	12	1.0
P_2	8	8	1.0



	T_s	T_q	T_q/T_s
P_1	12	24	2.0
P_2	17	23	1.35

Faustregel: in der Warteschleife die Kontrolle über den Prozessor abgeben
 laufend \mapsto bereit in Laufbereitschaft bleiben (S. 9-16)
 laufend \mapsto blockiert **selektive Verdrängungssteuerung**

Aktives Warten mit Prozessorabgabe

Kooperative Ausführung der Warteschleife

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

► der laufende Prozess gibt freiwillig die CPU ab, wenn die Schlossvariable nicht gesetzt werden konnte

- die Effektivität des Ansatzes hängt ab von der Umplanungsstrategie:
 - RR der aufgebende Prozess kommt ans Ende der Bereitliste ✓
 - sonst seine (stat./dyn.) Priorität bestimmt seine Listenposition ?
 - damit erhält er mehr oder weniger schnell wieder den Prozessor
 - hat er die höchste Priorität, gibt er den Prozessor nicht ab
 - seine **Wartepriorität** muss niedriger sein als seine **Laufpriorität**
- suboptimal: wartende Prozesse belasten nur seltener den Prozessor
 - besser wäre, wenn wartende Prozesse „schlafen“, d.h., blockieren

Aktives Warten mit Prozessorabgabe (Forts.)

Selektive Verdrängungssteuerung, z.B. durch NPCS

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

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

Zustandsmaschine der Prozesseinplanung:

`avert()` wehrt mögliche Verdrängung des laufenden Prozesses zeitweilig ab

`sleep()` lässt laufenden Prozess auf ein Ereignis (`lock`) passiv warten

▶ löst Verdrängungssperre

`admit()` lässt Verdrängung des laufenden Prozesses zu

- ▶ der laufende Prozess „legt sich selbst schlafen“, wenn das Setzen der Schlossvariablen scheitert
- ▶ bei Freigabe des KA werden all die Prozesse aufgeweckt, die auf das **Freigabeereignis** blockiert sind: `rouse(lock)`

Bedingungsvariable

(engl. *condition variable*)

Ein mit einer Schlossvariablen verknüpfter **abstrakter Datentyp** auf dem zwei Operationen definiert sind [3]:

await (auch: *wait*) \models Unterbrechungsprotokoll

- ▶ gibt den durch die Schlossvariable gesperrten kritischen Abschnitt automatisch frei
- ▶ blockiert den laufenden Prozess auf eine Bedingungsvariable
- ▶ bewirbt einen durch Ereignisanzeige deblockierten Prozess erneut um den Eintritt in den kritischen Abschnitt

cause (auch: *signal*) \models Signalisierungsprotokoll

- ▶ zeigt das mit der Bedingungsvariable assoziierte Ereignis an
- ▶ deblockiert die ggf. auf das Ereignis wartenden Prozesse

Ermöglicht einem Prozess, innerhalb eines kritischen Abschnitts zu warten, ohne diesen während der Wartephase belegt zu halten.

Beispiel: Datenpuffer ohne Pufferbegrenzung

Virtuell unendlich großer Puffer: Ringpuffer

```
typedef struct ringbuffer {
    char    data[NDATA];
    unsigned nput;
    unsigned nget;
} ringbuffer;

void rb_reset (ringbuffer *bufp) {
    bufp->nput = bufp->nget = 0;
}

char rb_fetch (ringbuffer *bufp) {
    return bufp->data[bufp->nget++ % NDATA];
}

void rb_store (ringbuffer *bufp, char item) {
    bufp->data[bufp->nput++ % NDATA] = item;
}
```

Problemstellen:

Füllstand log. Ablauf

- ▶ voll?
- ▶ leer?

füllen Zählen

- ▶ nput++

leeren Zählen

- ▶ nget++

 Synchronisation

Beispiel: Datenpuffer mit Pufferbegrenzung

(engl. *bounded buffer*)

Datenpuffer begrenzter Speicherkapazität als Ringpufferspezialisierung:

```
typedef struct buffer {
    ringbuffer    ring;
    unsigned char free;
    bool          lock;
} buffer;

void bb_reset (buffer *bufp) {
    rb_reset(&bufp->ring);
    bufp->free = NDATA;
    bufp->lock = 0;
}
```

free Bedingungsvariable

- ▶ Füllstandkontrolle

voll free = 0

leer free = NDATA

frei $0 < free \leq NDATA$

- ▶ Puffer ist initial leer

lock Schlossvariable

- ▶ Absicherung

- ▶ KA ist initial offen

Beispiel: Datenpuffer mit Pufferbegrenzung (Forts.)

Koordiniertes Leeren

```
char bb_fetch (buffer *bufp) {
    char item;
    acquire(&bufp->lock);
    while (bufp->free == NDATA)
        await(&bufp->ring, &bufp->lock);
    item = rb_fetch(&bufp->ring);
    bufp->free++;
    cause(&bufp->free);
    release(&bufp->lock);
    return item;
}
```

Puffer leeren ist ein KA:

- ▶ darf sich weder selbst noch mit dem Füllen überlappen
- ▶ **gegenseitiger Ausschluss**

Wartebedingung:

- ▶ Puffer ist leer

Entnahme eines Datums gibt ein **wiederverwendbares Betriebsmittel** frei

- ▶ die Anzahl der freien Puffereinträge erhöht sich um 1
- ▶ die Wartebedingung zum Füllen kann signalisiert werden

☞ das Datum selbst ist ein **konsumierbares Betriebsmittel**

Beispiel: Datenpuffer mit Pufferbegrenzung (Forts.)

Koordiniertes Füllen

```
void bb_store (buffer *bufp, char item) {
    acquire(&bufp->lock);
    while (!bufp->free)
        await(&bufp->free, &bufp->lock);
    rb_store(&bufp->ring, item);
    bufp->free--;
    cause(&bufp->ring);
    release(&bufp->lock);
}
```

Puffer füllen ist ein KA:

- ▶ darf sich weder selbst noch mit dem Leeren überlappen
- ▶ **gegenseitiger Ausschluss**

Wartebedingung:

- ▶ Puffer ist voll

Pufferung des Datums macht **konsumierbares Betriebsmittel** verfügbar

- ▶ die Anzahl der freien Puffereinträge erniedrigt sich um 1
- ▶ die Wartebedingung zum Leeren kann signalisiert werden

Bedingter kritischer Abschnitt [4]

(engl. *conditional critical section*, resp. *region*)

Betreten des kritischen Abschnitts ist von einer Wartebedingung abhängig, die nicht erfüllt sein darf, um den Prozess fortzusetzen

- ▶ die Bedingung ist als **Prädikat** über die im kritischen Abschnitt enthaltenen bzw. verwendeten Daten definiert
- ▶ vgl. Abbruchbedingungen (`while`) auf S. 11-34/11-35

Auswertung der Wartebedingung muss im kritischen Abschnitt erfolgen

- ▶ bei Nichterfüllung der Bedingung wird der Prozess auf Eintritt eines zur Wartebedingung korrespondierenden Ereignisses blockiert
 - ▶ damit das Ereignis später signalisiert werden kann, muss der kritische Abschnitt beim Schlafenlegen jedoch freigegeben werden
- ▶ bei (genauer: nach) Erfüllung/Signalisierung der Bedingung versucht der Prozess den kritischen Abschnitt wieder zu belegen
 - ▶ ggf. muss ein deblockierter Prozess die Bedingung neu auswerten

Unterbrechungsprotokoll

Prinzip — mit Problem(en)

```
void await (void *beep, bool *lock) {  
    release(lock);  
    block(beep);  
    acquire(lock);  
}
```

1. Freigabe des KA
2. Blockierung des Prozesses
3. Belegung des KA

Laufgefahr

Angenommen, der laufende Prozess hat den KA freigegeben und wird danach verdrängt \rightsquigarrow Zustand „laufbereit“:

- ▶ Während seiner Laufbereitschaft wird das Ereignis (`beep`), auf dessen Eintritt er (mit `block()`) passiv warten wollte, signalisiert.
- ▶ Da der Signalzustellung das „Vorhaben“ dieses Prozesses unbekannt ist, geht diesem das Ereignis (`beep`) verloren.
- ▶ Nach Wiedertzuteilung der CPU wird sich der Prozess blockieren und wartet sodann ggf. vergebens auf den Ereignisseintritt.

Unterbrechungs- und Signalisierungsprotokoll

Ereigniserwartung und -anzeige \rightsquigarrow Zustandsmaschine der Prozesseinplanung

```
void await (void *beep, bool *lock) {
    abide(beep);
    release(lock);
    block();
    acquire(lock);
}

void cause (void *beep) {
    rouse(beep);
}
```

`abide()` zeigt dem Planer an, dass der laufende Prozess ein Ereignis (`beep`) erwartet

`block()` blockiert den Prozess *logisch* auf das Ereignis: er wartet nur, falls das Ereignis noch nicht signalisiert wurde

`rouse()` weckt Prozesse, ggf.

- ▶ trat das Ereignis ein und wurde der Prozess aufgeweckt, versucht er erneut den kritischen Abschnitt zu betreten

☞ `abide()`, `block()`, `rouse()` \models **Wettlaufsituation** (vgl. S. 11-37)

Semaphor

„Signalmast“ (Eisenbahnwesen, engl. *semaphore*)

Eine „nicht-negative ganze Zahl“, für die zwei **unteilbare Operationen** definiert sind [5]:

P (hol. *prolaag*, „erniedrige“; auch *down*, *wait*)

- ▶ hat der Semaphor den Wert 0, wird der laufende Prozess blockiert
- ▶ ansonsten wird der Semaphor um 1 dekrementiert

V (hol. *verhoog*, erhöhe; auch *up*, *signal*)

- ▶ inkrementiert den Semaphor um 1
- ▶ auf den Semaphor ggf. blockierte Prozesse werden deblockiert

Ein **abstrakter Datentyp** zur **Signalisierung von Ereignissen** zwischen gleichzeitigen Prozessen (deren Ausführung sich zeitlich überschneidet).

☞ vgl. S. 7-81

Kompositer Datentyp

Zusammengesetzt aus Bedingungsvariable und Schlossvariable

```
typedef struct semaphore {
    unsigned int load;
    bool        lock;
} semaphore;

void initial (semaphore *sema, unsigned int load) {
    sema->load = load;
    sema->lock = 0;
}
```

Vorbelegung...

- ▶ definiert die Anzahl der vom Semaphor zu verwaltenden Betriebsmittel
- ▶ entriegelt das Schloss

load Bedingungsvariable

- ▶ implementiert das Protokoll zwischen P und V

lock Schlossvariable

- ▶ zur Unteilbarkeit der Operationen: **kritischer Abschnitt**

Kritische Abschnitte P und V

Verhungersgefahr — auch kurze nichtsequentielle Programme sind nicht einfach...

```
void prolaag (semaphore *sema) {
    acquire(&sema->lock);
    while (sema->load == 0)
        await(&sema->load, &sema->lock);
    sema->load--;
    release(&sema->lock);
}

void verhoog (semaphore *sema) {
    acquire(&sema->lock);
    if (sema->load++ == 0)
        cause(&sema->load);
    release(&sema->lock);
}
```

Wartebedingung

- ▶ Semaphorwert ist 0
- ▶ muss wiederholt ausgewertet werden

Signalisierung

- ▶ Semaphorwert war 0
- ▶ läuft ggf. ins Leere
 - ▶ **Warteschlange** würde dem vorbeugen *oder*
 - ▶ Semaphorwert < 0 als Warteähler

Wiedereintritt nach erfolgter Signalisierung ist damit konfrontiert, dass andere Prozesse ggf. vorbeigezogen sind

Instrumente zur Betriebsmittelvergabe

Differenziert nach dem Wertebereich eines Semaphors

binärer Semaphor (engl. *binary semaphore*)

- ▶ verwaltet zu einem Zeitpunkt immer nur genau ein Betriebsmittel
 - ▶ gegenseitiger Ausschluss (engl. *mutual exclusion*, *mutex*)⁴
- ▶ vergibt **unteilbare Betriebsmittel** an Prozesse
- ▶ besitzt den Wertebereich $[0, 1]$

zählender Semaphor (engl. *counting semaphore*, *general semaphore*)

- ▶ verwaltet zu einem Zeitpunkt mehr als ein Betriebsmittel
 - ▶ d.h., mehrere Betriebsmittelinstanzen desselben Typs
- ▶ vergibt **konsumier-** bzw. **teilbare Betriebsmittel** an Prozesse
- ▶ besitzt den Wertebereich $[0, N]$, für N Betriebsmittel

⁴Der Begriff „Mutex“ steht i.A. für einen binären Semaphor, mit dem zeitweilige Eigentümerschaft eines Fadens verknüpft ist: Nur für den Faden, der Mutex M besitzt, d.h., dem also $P(M)$ gelungen ist, wird $V(M)$ gelingen.

Arten von Betriebsmitteln

(vgl. S. 11-6)

Semaphore und Betriebsmittelverwaltung

wiederverwendbare Betriebsmittel werden angefordert und freigegeben

- ▶ ihre Anzahl ist begrenzt: Prozessoren, Geräte, Speicher (Puffer)
 - teilbar** zu einer Zeit von mehreren Prozessen belegbar
 - unteilbar** zu einer Zeit von einem Prozess belegbar
- ▶ auch ein kritischer Abschnitt ist solch ein Betriebsmittel
 - ▶ von jedem Typ gibt es jedoch nur eine einzige Instanz

konsumierbare Betriebsmittel werden erzeugt und zerstört

- ▶ ihre Anzahl ist (log.) unbegrenzt: Signale, Nachrichten, Interrupts
 - Produzent** kann beliebig viele davon erzeugen
 - Konsument** zerstört sie wieder bei Inanspruchnahme
- ▶ Produzent und Konsument sind voneinander abhängig (S. 7-88)

Ausschließender Semaphor

Vergabe unteilbarer Betriebsmittel

```
semaphore mutex = {1, 0};

void chain (chainlink **next, chainlink *item) {
    prolaag(&mutex);
    *next = (*next)->link = item;
    verhoog(&mutex);
}
```

Beispiel von S. 7-82:

P() prolaag(&mutex)

V() verhoog(&mutex)

unteilbares Betriebsmittel von dem es nur eine Instanz gibt

- ▶ der Initialwert des Semaphors ist 1

mehrseitige Synchronisation in welcher Reihenfolge die gleichzeitigen Prozesse den kritischen Bereich betreten werden, ist unbestimmt

- ▶ gleichzeitig können jedoch nicht mehrere Prozesse drin sein

Signalisierender Semaphor

Vergabe konsumierbarer Betriebsmittel

```
semaphore mite = {0, 0};
char      data;

char consumer () {
    prolaag(&mite);
    return data;
}

void producer (char item) {
    data = item;
    verhoog(&mite);
}
```

konsumierbares Betriebsmittel muss vor dem Verbrauch erst erzeugt werden

- ▶ der Initialwert des Semaphors ist 0

einseitige Synchronisation nur einer von beiden Prozessen wird ggf. blockieren

- ▶ der Konsument, wenn noch kein Datum verfügbar ist

Der Datenpuffer ist begrenzt, jedoch wird die Pufferbegrenzung ignoriert:

- ▶ Daten gehen verloren, wenn die Prozesse nicht im gleichen Takt arbeiten: $Konsument^* \rightarrow (Produzent \rightarrow Konsument)^+$

Beispiel: Datenpuffer mit Pufferbegrenzung

Bounded buffer revisited. . .

Ringpufferspezialisierung: „Dreiergespann“ von Semaphore. . .

```
typedef struct buffer {
    ringbuffer ring;
    semaphore lock;
    semaphore free;
    semaphore full;
} buffer;

void bb_reset (buffer *bufp) {
    rb_reset(&bufp->ring);
    initial(&bufp->lock, 1);
    initial(&bufp->free, NDATA);
    initial(&bufp->full, 0);
}
```

lock sichert die Pufferoperationen

- ▶ **gegenseitiger Ausschluss** von lesen/schreiben

free verhindert **Pufferüberlauf**

- ▶ stoppt den Schreiber beim vollen Puffer

full verhindert **Pufferunterlauf**

- ▶ stoppt den Leser beim leeren Puffer

Beispiel: Datenpuffer mit Pufferbegrenzung (Forts.)

Koordiniertes Leeren

```
char bb_fetch (buffer *bufp) {
    char item;
    prolaag(&bufp->full);
    prolaag(&bufp->lock);
    item = rb_fetch(&bufp->ring);
    verhoog(&bufp->lock);
    verhoog(&bufp->free);
    return item;
}
```

Szenario beim Leeren:

- ▶ einem leeren Puffer kann nichts entnommen werden
- ▶ freigewordener Pufferplatz soll wiederverwendbar sein
- ▶ den Puffer zu leeren, ist ein kritischer Abschnitt

einseitige Synchronisation \mapsto zwei signalisierende Semaphore

- ▶ durch **full** ein konsumierbares Betriebsmittel anfordern
- ▶ durch **free** ein wiederverwendbares Betriebsmittel bereitstellen

mehrseitige Synchronisation \mapsto ausschließender Semaphor **lock**

- ▶ sich selbst überlappendes Leeren und Leeren überlappendes Füllen

Beispiel: Datenpuffer mit Pufferbegrenzung (Forts.)

Koordiniertes Füllen

```
void bb_store (buffer *bufp, char item) {
    prolaag(&bufp->free);
    prolaag(&bufp->lock);
    rb_store(&bufp->ring, item);
    verhoog(&bufp->lock);
    verhoog(&bufp->full);
}
```

Szenario beim Füllen:

- ▶ voll ist voll...
- ▶ gepufferte Daten sollen konsumierbar sein
- ▶ Puffer füllen ist kritisch

einseitige Synchronisation \mapsto zwei signalisierende Semaphore

- ▶ durch **free** ein wiederverwendbares Betriebsmittel anfordern
- ▶ durch **full** ein konsumierbares Betriebsmittel bereitstellen

mehrseitige Synchronisation \mapsto ausschließender Semaphor **lock**

- ▶ sich selbst überlappendes Füllen und Füllen überlappendes Leeren

Beispiel: Datenpuffer mit Pufferbegrenzung (Forts.)

Beliebter Fehler — nicht nur bei Testaten/Klausuren...

```
char bb_fetch (buffer *bufp) {
    char item;
    prolaag(&bufp->lock);
    prolaag(&bufp->full);
    item = rb_fetch(&bufp->ring);
    verhoog(&bufp->free);
    verhoog(&bufp->lock);
    return item;
}
```

Was kann hier die Folge sein?

```
void bb_store (buffer *bufp, char item) {
    prolaag(&bufp->lock);
    prolaag(&bufp->free);
    rb_store(&bufp->ring, item);
    verhoog(&bufp->full);
    verhoog(&bufp->lock);
}
```

Verklemmungsgefahr

Angenommen, ein Prozess findet (a) beim Leeren, dass kein Datum *oder* (b) beim Füllen, dass kein freier Platz im Puffer verfügbar ist:

- ▶ Der Prozess wird dann im KA auf **full** oder **free** blockieren, den KA (**lock**) dabei aber nicht freigeben.
- ▶ Jeder andere Prozess, der ein Datum oder den freien Platz verfügbar machen könnte, würde dann beim Eintritt in diesen KA blockieren.

Semaphore „*considered harmful*“

Nicht alles „Gold“ glänzt. . .

- ▶ auf Semaphore basierende Lösungen sind komplex und fehleranfällig
 - ▶ Synchronisation: **Querschnittsbelang** nichtsequentieller Programme
 - ▶ kritische Abschnitte neigen dazu, mit ihren P/V-Operationen quer über die Software verstreut vorzuliegen
 - ▶ das Schützen gemeinsamer Variablen bzw. Freigeben kritischer Abschnitte kann dabei leicht vergessen werden
- ▶ hohe Gefahr der **Verklemmung** (engl. *deadlock*) von Prozessen
 - ▶ umso zwingender sind Verfahren zur Vorbeugung, Vermeidung und/oder Erkennung solcher Verklemmungen
 - ▶ nicht-blockierende Synchronisation ist mit diesem Problem nicht behaftet, dafür jedoch nicht immer durchgängig praktikierbar
- ▶ linguistische Unterstützung reduziert Fehlermöglichkeiten gravierend

Monitor

(engl. *monitor*)

Ein **synchronisierter abstrakter Datentyp**, d.h., ein ADT mit impliziten Synchronisationseigenschaften [6, 7]:

mehrseitige Synchronisation an der Monitorschnittstelle (**Semaphor**)

- ▶ gegenseitiger Ausschluss der Ausführung exportierter Prozeduren

einseitige Synchronisation innerhalb des Monitors (**Bedingungsvariable**)

wait blockiert einen Prozess auf das Eintreten eines Signals/einer Bedingung und gibt den Monitor implizit wieder frei

signal zeigt das Eintreten eines Signals/einer Bedingung an und deblockiert (genau einen oder alle) darauf blockierte Prozesse

☞ sprachgestützter Ansatz: Concurrent Pascal, PL/I, Mesa, . . . , Java.

Monitor \equiv Modul

Monitor vs. Semaphore

Kapselung (engl. *encapsulation*)

- ▶ von mehreren Prozessen gemeinsam bearbeitete Daten müssen in Monitoren organisiert vorliegen
- ▶ als Konsequenz muss die Programmstruktur kritische Abschnitte explizit sichtbar machen
 - ▶ inkl. zulässige (an zentraler Stelle definierte) Zugriffsfunktionen

Datenabstraktion (engl. *information hiding*)

- ▶ wie ein Modul, so kapselt auch ein Monitor für mehrere Funktionen Wissen über gemeinsame Daten
- ▶ Auswirkungen lokaler Programmänderungen bleiben begrenzt

☞ ein Monitor ist Konzept der Ebene ₅, ein Semaphore das der Ebene ₃

Modulkonzept erweitert um Synchronisationssemantik

Monitor \equiv implizit synchronisierte Klasse

Monitorprozeduren (engl. *monitor procedures*)

- ▶ schließen sich bei konkurrierenden Zugriffen durch mehrere Prozesse in ihrer Ausführung gegenseitig aus
 - ▶ der erfolgreiche Prozeduraufruf sperrt den Monitor
 - ▶ bei Prozedurrückkehr wird der Monitor wieder entsperrt
- ▶ repräsentieren per Definition kritische Abschnitte, deren Integrität vom Kompilierer garantiert wird
 - ▶ die „Klammerung“ kritischer Abschnitte erfolgt automatisch
 - ▶ der Kompilierer setzt die dafür notwendigen Steueranweisungen ab

Synchronisationsanweisungen: Semaphore, Schloss-/Bedingungsvariablen

- ▶ sind Querschnittsbelang eines Monitors und nicht des gesamten nichtsequentiellen Programms
 - ▶ sie liegen nicht quer über die ganze Software verstreut vor

Signalisierung einer Wartebedingung erwarten

wait

Monitorfreigabe als notwendiger Seiteneffekt beim Warten:

- ▶ andere Prozesse wären sonst an den Monitoreintritt gehindert
- ▶ als Folge könnte die zu erfüllende Bedingung nie erfüllt werden
- ▶ schlafende Prozesse würden nie mehr erwachen \leadsto **Verklemmung**

Monitordaten sind in einem konsistenten Zustand zu hinterlassen

- ▶ andere Prozesse aktivieren den Monitor während der Blockadephase
- ▶ als Folge sind (je nach Funktion) Zustandsänderungen zu erwarten
- ▶ vor Eintritt in die Wartephase muss der Datenzustand konsistent sein

 aktives Warten im Monitor ist logisch komplex und leistungsmindernd

Signalisierung einer Wartebedingung

signal

Prozessblockaden in Bezug auf eine Wartebedingung werden aufgehoben

- ▶ im Falle wartender Prozesse sind als Anforderungen zwingend:
 - ▶ wenigstens ein Prozess deblockiert an der Bedingungsvariablen
 - ▶ höchstens ein Prozess rechnet nach der Operation im Monitor weiter
- ▶ erwartet kein Prozess ein Signal, ist die Operation wirkungslos
 - ▶ d.h., Signale dürfen in Bedingungsvariablen nicht gespeichert werden

Lösungsansätze hierzu sind z.T. von sehr unterschiedlicher Semantik

- ▶ das betrifft etwa die Anzahl der befreiten Prozesse:
 - ▶ alle auf die Bedingung wartenden oder genau nur einer
- ▶ gr. Unterschiede liegen auch in **Besitzwechsel** bzw. **Besitzwahrung**
 - ▶ „falsche Signalisierungen“ werden toleriert oder nicht

Signalisierungssemantiken

Besitzwahrung

genau einen wartenden Prozess befreien ... nur welchen?

- ▶ bei mehr als einen wartenden Prozess ist eine Auswahl zu treffen
- ▶ die Auswahlentscheidung muss konform zur Fadeneinplanung sein
- ▶ ggf. ist bereits bei Blockierung möglichen Konflikten vorzubeugen

alle wartenden Prozesse befreien \mapsto Hansen [3]

- ▶ die Auswahlentscheidung unterliegt allein dem Planer
- ▶ Fadeneinplanung entgegenwirkende Effekte werden ausgeschlossen
- ▶ verschiedene Belange sind sauber voneinander getrennt

- ▶ in beiden Fällen erfolgt die **Neuauswertung der Wartebedingung**
 - ▶ dadurch werden jedoch auch falsche Signalisierungen toleriert
- ▶ signalisierte Prozesse bewerben sich erneut um den Monitorzutritt

Signalisierungssemantiken (Forts.)

Besitzwechsel

Wechsel vom signalisierenden zum signalisierten Prozess \mapsto Hoare [7]

- ▶ nur einer von ggf. mehreren wartenden Prozessen wird signalisiert
 - ▶ der signalisierte Prozess setzt seine Berechnung im Monitor fort
 - ▶ als Folge muss der signalisierende Prozess den Monitor abgeben
- ▶ **Fortführungsbedingung** des signalisierten Prozesses ist garantiert
 - ▶ seit Signalisierung war kein anderer Prozess im Monitor drin
 - ▶ kein anderer Prozess konnte die Fortführungsbedingung entkräften

- ▶ es erfolgt **keine Neuauswertung der Wartebedingung**
 - ▶ als Konsequenz werden falsche Signalisierungen nicht toleriert
- ▶ eine erhöhte Anzahl von Fadenwechseln ist in Kauf zu nehmen
- ▶ der signalisierende Prozess bewirbt sich erneut um Monitorzutritt

Beispiel: Datenmonitor mit Pufferbegrenzung

Bounded buffer revisited. . .

„Concurrent C++“

```
class Ringbuffer {
    char    data[NDATA];
    unsigned nput, nget;
public:
    Ringbuffer ()    { nput = nget = 0; }
    char fetch ()    { return data[nget++ % NDATA]; }
    void store (char) { data[nput++ % NDATA] = item; }
};
```

```
monitor Buffer : private Ringbuffer {
    unsigned free;
    condition null, full;
public:
    Buffer ()        { free = NDATA; }
    char fetch ();
    void store (char);
};
```

Beispiel: Datenmonitor mit Pufferbegrenzung (Forts.)

Koordiniertes Leeren

```
char Buffer::fetch () {
    char item;
    while (free == NDATA) full.await();
    item = Ringbuffer::fetch();
    free++;
    null.signal();
    return item;
}
```

Bedingungsvariablen:

full erwartet einen Eintrag

null signalisiert freien Platz

Instanzvariable:

free aktueller „Pegelstand“

Hansen'scher Monitor Wartebedingung ist wiederholt zu überprüfen

- ▶ bewirbt signalisierte Prozesse erneut um den Monitorzutritt
 - ▶ die Phase ab der Signalisierung von **full** bis zum Wiedereintritt in den Monitor des signalisierten (leerenden) Prozesses ist teilbar
 - ▶ der Puffer könnte zwischenzeitig geleert worden sein \rightsquigarrow blockieren
- ▶ toleriert (fehlerbedingte) **falsche Signalisierungen** von **full**

Beispiel: Datenmonitor mit Pufferbegrenzung (Forts.)

Koordiniertes Füllen

```
void Buffer::store (char item) {
    while (!free) null.await();
    Ringbuffer::store(item);
    free--;
    full.signal();
}
```

Bedingungsvariablen:

`null` erwartet freien Platz

`full` signalisiert einen Eintrag

Instanzvariable: `free` führt Buch über den aktuellen „Pegelstand“

Hansen'scher Monitor Wartebedingung ist wiederholt zu überprüfen

- ▶ bewirbt signalisierte Prozesse erneut um den Monitorzutritt
 - ▶ die Phase ab der Signalisierung von `null` bis zum Wiedereintritt in den Monitor des signalisierten (füllenden) Prozesses ist teilbar
 - ▶ der Puffer könnte zwischenzeitig gefüllt worden sein \leadsto blockieren
- ▶ toleriert (fehlerbedingte) **falsche Signalisierungen** von `null`

Monitorkonzepte im Vergleich

Hansen vs. Hoare

Hansen'scher Monitor

```
while (free == NDATA) full.await();
while (!free) null.await();
```

Prozessen **wird nicht garantiert**, dass nach ihrer Signalisierung die Fortführungsbedingung gilt

- ▶ andere Prozesse können den Monitor betreten haben
- ▶ Wartebedingung erneut prüfen
- ▶ evtl. falsche Signalisierungen **werden toleriert**

Hoare'scher Monitor

```
if (free == NDATA) full.await();
if (!free) null.await();
```

Prozessen **wird garantiert**, dass nach ihrer Signalisierung die Fortführungsbedingung gilt

- ▶ kein anderer Prozess konnte den Monitor betreten haben
- ▶ Wartebedingung einmal prüfen
- ▶ evtl. falsche Signalisierungen **werden nicht toleriert**

Blockierende Synchronisation „*considered harmful*“

Probleme von Schlossvariablen, Semaphore und Monitore

Leistung (engl. *performance*) insbesondere in SMP-Systemen [2]

- ▶ „*spin locking*“ reduziert ggf. massiv Busbandbreite

Robustheit (engl. *robustness*) „*single point of failure*“

- ▶ ein im kritischen Abschnitt scheiternder Prozess kann schlimmstenfalls das ganze System lahm legen

Einplanung (engl. *scheduling*) wird behindert bzw. nicht durchgesetzt

- ▶ un- bzw. weniger wichtige Prozesse können wichtige Prozesse „ausbremsen“ bzw. scheitern lassen
- ▶ **Prioritätsverletzung, Prioritätsumkehr** [8]
 - ▶ Mars Pathfinder [9, 10]

Verklemmung (engl. *deadlock*) einiger oder sogar aller Prozesse

Optimistisches Verfahren der Synchronisation

Nebenläufigkeit unterstützen, nicht einschränken wie beim gegenseitigen Ausschluss

Koordinierung sich einander ggf. überlappender Aktivitäten, ohne dabei **gleichzeitige Prozesse** auszuschließen

- ▶ toleriert (pseudo-) parallele Programmausführungen
 - parallel** ▶ Multiprozessor, wirkliche Parallelität
 - pseudoparallel** ▶ Uniprozessor, Parallelität durch Unterbrechungen
- ▶ die Verfahren greifen auf nichtprivilegierte Befehle der ISA zurück
 - CISC** ▶ TAS, FAA, **CAS** bzw. CMPXCHG
 - RISC** ▶ LL/SC
- ▶ d.h., sie funktionieren im Benutzer- wie auch im Systemmodus

Beachte

- ▶ kein gegenseitiger Ausschluss ⇒ **Verklemmungsvorbeugung**
- ▶ die *benutzten* Befehle sind „echte“ Elementaroperationen der ISA

Bedingte Wertzuweisung (engl. *compare and swap*, CAS)

Spezialbefehl zur Multiprozessorsynchronisation

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

Komplexbefehl (einer CPU),
der scheitern kann:

true Operation ist gelungen,
das Speicherwort wurde
überschrieben

false Operation ist gescheitert

Analogie zu TAS (S. 11-26): **atomarer „read-modify-write“-Zyklus**

- ▶ Lese-/Schreibzyklen des Prozessors werden unteilbar ausgeführt
 - ▶ auf *Interrupts* wird, wie sonst auch, erst am Befehlsende reagiert

Unteilbarkeit \leftrightarrow Unterbrechungs- und Zugriffssperre (Bus)

- ▶ ELOP implementiert eine „Transaktion“ in Speicherwortgranularität
 - ▶ für Uni- und Multi(kern)prozessoren

CAS: Implementierung

Abbildung auf CMPXCHG (i486 und später)

```
int cas (word *ref, word exp, word val) {
    unsigned char aux;

    __asm__ __volatile__(
        "lock\n\t"                /* prefix next instruction */
        "cmpxchgl %2,%1\n\t"      /* (ref) == exp ? (ref) = val */
        "sete %0"                 /* extend ZF into aux */
        : "=q" (aux), "=m" (*ref)
        : "r" (val), "m" (*ref), "a" (exp) /* %eax loaded with exp */
        : "memory");

    return aux;
}
```

Beachte \leftrightarrow *lock*

optional für Uniprozessorsysteme: Befehle der ISA sind ununterbrechbar
zwingend für Multiprozessorsysteme: setzt Zugriffssperre (Datenbus)

CAS: Prinzip der optimistischen Synchronisation

Kollisionserkennung und -behandlung

erledige NBS mit CAS;

wiederhole

ziehe *lokale Kopie* des Inhalts der *Adresse* einer globalen Variablen;
verwende die Kopie, um einen neuen *lokalen Wert* zu berechnen;
versuche CAS: an *Adresse*, die *lokale Kopie* mit dem *lokalen Wert*;

solange CAS scheitert;

basta.

- pros**
- ▶ Tolerierung beliebiger Überlappungsmuster
 - ▶ transparent für die Einplanung: keine Prioritätsumkehr
 - ▶ **Verklemmungsvorbeugung**: kein gegenseitiger Ausschluss
 - ▶ Robustheit: keine hängenden Sperren bei Programmabbrüchen
- cons**
- ▶ Gefahr von **Verhungering** (engl. *starvation*)
 - ▶ Wiederverwendung sequentieller Altsoftware unmöglich
 - ▶ **Entwicklung** nebenläufiger Varianten im Regelfall **nicht trivial**

Fallstudie: Wartestapelmanipulation

Einplanung nach LCFS (Abk. für engl. *last come, first served*)

Aufnahme in die Liste...

```
void nbs_ahead (chainlink *this, chainlink *item) {
    do item->link = this->link;                /* is elected head */
    while (!CAS(&this->link, item->link, item)); /* try push item */
}
```

Entnahme aus der Liste...

```
chainlink *nbs_strip (chainlink *this) {
    chainlink *node;

    do if ((node = this->link) == 0) break;    /* access head */
    while (!CAS(&this->link, node, node->link)); /* try pull node */

    return node;
}
```

```
#define CAS(r,e,v)    cas((word*)r, (word)e, (word)v)
```

Problem ABA

Transaktion gelingt trotz zwischenzeitlicher Änderungen

Phänomen der nichtblockierenden Synchronisation auf Basis eines CAS, d.h., einer ELOP, die inhaltsbasiert arbeitet⁵

- ▶ angenommen zwei Fäden, F_1 und F_2 , stehen im Wettstreit um eine gemeinsame Variable V
 - F_1 ▶ liest den Wert A von V , speichert diesen als Kopie, wird dann allerdings vor dem CAS_V für unbestimmte Zeit verzögert
 - F_2 ▶ durchläuft dieselbe Sequenz, schafft jedoch mittels CAS_V den Wert B an V zuzuweisen
 - ▶ anschließend wird (in einem weiteren Durchlauf dieser Sequenz) wieder der ursprüngliche Wert A an V zugewiesen
 - F_1 ▶ setzt seine Ausführung mit CAS_V fort, erkennt, dass V den Wert A seiner Kopie speichert und überschreibt V
- ▶ im Ergebnis kann dieses Überlappungsmuster dazu führen, dass F_1 mittels CAS_V einen falschen Wert nach V transferiert

⁵Bei *Adressreservierung* wie z.B. mit LL/SC besteht dieses Problem nicht.

Problem ABA (Forts.)

Wartestapelzustand: $LINK(this) \rightarrow A \rightarrow B \rightarrow C$

	F_1	F_2	F_2	F_2	F_1
	<i>st. . .</i>	<i>strip</i>	<i>strip</i>	<i>ahead</i>	<i>. . .rip</i>
$LINK(this)$	A	A	B	C	A
$node_{strip}$	A	A	B	—	A
$item_{ahead}$	—	—	—	A	—
$LINK(node)$	B	B	C	—	*
$CAS_{LINK(this)}$. . .	$A \rightsquigarrow B$	$B \rightsquigarrow C$	$C \rightsquigarrow A$	$A \rightsquigarrow ?$

* Unterbrechung vor/nach Lesen von $LINK(node)$ (vgl. S. 11-67)

vor $\mapsto LINK(node)$ undefiniert $\Rightarrow LINK(this)$ undefiniert

nach $\mapsto LINK(node) = B \Rightarrow LINK(this) = B$

Beides resultiert in einen falschen Wert für $LINK(this)$. . .

!!!

Problem ABA (Forts.)

Kritische Variable durch einen „Zeitstempel“ (logische Uhr, Zähler) absichern

Abhilfe besteht darin, den umstrittenen Zeiger (nämlich *item* bzw. *node*) um einen problemspezifischen **Generationszähler** zu erweitern

- Etikettieren**
- ▶ Zeiger mit einem Anhänger (engl. *tag*) versehen
 - ▶ Ausrichtung (engl. *alignment*) ausnutzen, z.B.:

$$\text{sizeof}(\text{chain_t}) \rightsquigarrow 4 = 2^2 \Rightarrow n = 2$$

⇒ $\text{chain_t} *$ ist Vielfaches von 4

⇒ $\text{chain_t} * \text{Bits}[0:1]$ immer 0

- ▶ Platzhalter für n -Bit Marke/Zähler in jedem Zeiger
- DCAS**
- ▶ Abk. für (engl.) *double compare and swap*
 - ▶ Marke/Zähler als elementaren Datentyp auslegen
 - ▶ *unsigned int* hat Wertebereich von z.B. $[0, 2^{32} - 1]$
 - ▶ zwei Maschinenworte (Zeiger, Marke/Zähler) ändern

Problem ABA (Forts.)

Abhilfe (engl. *workaround*)

Beachte ↔ Generationszähler

- ▶ eine Lösung für CAS-artige Verfahren ist nicht wirklich gegeben
- ▶ die Wahrscheinlichkeit, dass das Problem auftritt, wird verringert
- ▶ Überlappungsmuster haben Einfluss auf den Wertebereich
 - ▶ bestimmt durch Zusammenspiel *und* Anzahl der wettstreitigen Fäden
 - ▶ ein Bit kann reichen, ebenso, wie ein *unsigned int* zu klein sein kann

Vorbeugung (engl. *prevention*) ist der richtige Ansatz — sofern möglich

- ▶ beliebige Überlappungsmuster konstruktiv (Entwurf) ausschließen
- ▶ auf **Adressreservierungsverfahren** der Hardware zurückgreifen
 - ▶ unterstützt nicht jede Hardware, ist nur typisch für RISC
 - ▶ z.B. ELOP-Paar *load linked, store conditional* (LL/SC) verwenden

Dualität von Koordinierungstechniken

Theorie vs. Praxis

Problem	Methode
uneingeschränkte Nebenläufigkeit	nicht-blockierende Synchronisation
gegenseitiger Ausschluss	Schlossvariable
explizite Prozesssteuerung	Bedingungsvariable
bedingte Verzögerung	bedingter kritischer Abschnitt
Austausch von Zeitsignalen	Semaphor
Austausch von Daten	Nachrichtenpuffer (<i>bounded buffer</i>)

logisch betrachtet sind alle Methoden äquivalent, da jede von ihnen hilft, ein beliebiges Steuerungsproblem zu lösen

praktisch betrachtet sind die Methoden nicht äquivalent, da einige von ihnen für ein gegebenes Problem zu komplexen und ineffizienten Lösungen führen

Synchronisation

Koordination von Kooperation und Konkurrenz

- ▶ die Verfahren sind problemspezifisch und teils radikal unterschiedlich
 - ▶ einseitig oder mehrseitig
 - ▶ unterdrückend, blockierend oder nicht-blockierend
 - ▶ behinderungs-, sperr- oder wartefrei
- ▶ blockierende Verfahren erlauben die Wiederverwendung sequentieller Programme für nichtsequentielle Ausführungsumgebungen
 - ▶ Schlossvariable, Bedingungsvariable, Semaphor, Monitor
 - ▶ die Gefahr von Verklemmungen ist stellenweise sehr hoch
- ▶ nicht-blockierende Verfahren sind frei von Verklemmungen, jedoch nicht unbedingt frei von Verhungerung
 - ▶ die Ansätze profitieren von Spezialbefehlen der CPU:
 - CISC \leftrightarrow cas, cas2 (dcas), cmpxchg
 - RISC \leftrightarrow ll/sc
 - ▶ wartefreie Varianten beugen dem Verhungern von Prozessen vor
- ▶ nichtsequentielle Programmierung ist nicht nur Betriebssystemfall

Literaturverzeichnis

- [1] Ralf Guido Herrtwich and Günter Hommel.
Kooperation und Konkurrenz — Nebenläufige, verteilte und echtzeitabhängige Programmsysteme.
Springer-Verlag, 1989.
- [2] Ray Bryant, Hung-Yang Chang, and Bryan S. Rosenburg.
Experience developing the RP3 operating system.
Computing Systems, 4(3):183–216, 1991.
- [3] Peer Brinch Hansen.
Structured multiprogramming.
Communications of the ACM, 15(7):574–578, July 1972.
- [4] Charles Antony Richard Hoare.
Towards a theory of parallel programming.
In C. A. R. Hoare and R. H. Perrot, editors, *Operating System Techniques*. Academic Press, London, New York, 1971.

Literaturverzeichnis (Forts.)

- [5] Edsger Wybe Dijkstra.
Cooperating sequential processes.
Technical report, Technische Universiteit Eindhoven, Eindhoven, The Netherlands, 1965.
(Reprinted in *Great Papers in Computer Science*, P. Laplante, ed., IEEE Press, New York, NY, 1996).
- [6] Peer Brinch Hansen.
Operating System Principles.
Prentice Hall International, 1973.
- [7] Charles Antony Richard Hoare.
Monitors: An operating system structuring concept.
Communications of the ACM, 17(10):549–557, October 1974.

Literaturverzeichnis (Forts.)

- [8] Butler W. Lampson and David D. Redell.
Experiences with processes and monitors in mesa.
Communications of the ACM, 23(2):105–117, 1980.
- [9] David Wilner.
Vx-files: What really happened on mars?
Keynote at the 18th IEEE Real-Time Systems Symposium (RTSS '97), December 1997.
- [10] Michael B. Jones.
What really happened on mars?
http://www.research.microsoft.com/~mbj/Mars_Path-finder/M
1997.