Concurrent Systems
Zusammenfassung des Wichtigsten

Wintersemester 2015/2016

Prof. Dr.-Ing Schröder-Preikschat (Wosch)

Autor:

- Christian Strate

Inhaltsverzeichnis

0 Organisatorisches 3
1 Introduction 3
2 Concurrency 4
3 Processes 6
4 Critical Sections 7
5 Elementary Operations 8
6 Locks 19
7 Semaphore 29
8 Monitor 30
9 Deadlock and Livelock 37
10 Guarded Sections 39
11 NBS - Non-Blocking Synchronisation 45
12 Transactional Memory - TM 55
13 Progress Guarantees 57

0 Organisatorisches

- Mo Übung., ab nächster Woche
- VL wird evtl. verschoben
- auch für Anwendungssäule verwendbar
- Für (mündlichen) Prüfungstermin mail an wosch
- Pro-Tipp: Wenn es im eigenen Zimmer mal etwas zu kalt sein sollte empfehle ich ein paar CS-Algorithmen nachzuvollziehen. Der rauchende Kopf sorgt für genügend Abwärme. ;)
- Einige der Algorithmen befanden sich im Anhang und sind hochinteressant. Da ich nicht wusste inwiefern diese trotzdem Prüfungsrelevant sind, habe ich diese nicht als "Anhang" markiert.

Note:

Bei dieser Zusammenfassung handelt es sich um eine inoffizielle Mitschrift von Studenten. Entsprechend sind weder Garantie auf Vollständigkeit noch auf Korrektheit gewährleistet.

1 Introduction

Konkurrenz im Programmfluss durch multiplication – echte Parallelität – und multiplexing – Pseudo-Parallelität.

Note: Viele Code-Beispiele sind so gestaltet, dass zu Gunsten der Lesbarkeit an einigen Stellen Fences und ABA-Behandlungen weggelassen wurden.

2 Concurrency

concurrency - Nebenläufigkeit:
(Kausal) Unabhängige Ereignisse ohne gegenseitige Einflüsse.

causality - Kausalität:
Zusammenhang zwischen Ursache und Wirkung. Kausale Verkettung von Ereignissen.

Definition concurrent:
Ereignisse geschehen oder sind nebenläufig, wenn keines die Wirkung des anderen ist. Also sind die Anforderungen:

- Daten- und Kontrollflussunabhängig
- Berechnungen müssen (quasi)-simultaneous hardware multiplication/(multiplexing) sein.

Amdahlsches Gesetz:
$su = \frac{r_s + r_p}{r_s + \frac{r_p}{n}} = \frac{1}{r_s + \frac{r_p}{n}}$
wobei r_s Rate sequential Code, r_p Rate parallel Code, n Number of processors

Gustafson's Law:
$ssu = \frac{r_s + r_p \cdot n}{r_s + r_p} = r_s + r_p \cdot n = n + (1 - n) \cdot r_s$
wobei r_p Rate des parallelen Codes, der mit n skaliert.

Beachtung der Abstraktionslevel

- Problematisch ist unvermeidbare Sequentialisierung auf niedriger Abstraktionsebene (Speicherzugriff, Bus).
  implies Performanz-Verlust
- Andersrum können logisch sequentielle Operationen auf der HW nebenläufig abgearbeitet werden oder logisch atomare Operationen in mehrere aufgeteilt werden (Wertgröße > Registergröße).
  implies Race Condition und damit Fehler
- Hardware Ressourcen (wiederverwendbare + konsumierbare) - Verwaltung durch OS, Beispiele konsumierbar: Trap, IRQ
- Software Ressourcen (wiederverwendbare + konsumierbare) - Verwaltung durch Programme

- Beachte: Wiederverwendbare Ressourcen werden auch für konsumierbare benötigt (Speicher)
- Wiederverwendbare Ressourcen erfordern Mehrseitige Synchronisation falls Gleichzeitiger Ressourcen-Zugriff (auf geteilte Ressourcen) stattfindet, der die Ressource in einen inkonsistenten Zustand bringen könnte
- Konsumierbare Ressourcen erfordern einseitige Synchronisation
  Verwendung temporärer Ressourcen folgt einer kausalen Ereigniskette. Betrifft den Erzeuger nicht-blockierend und den Konsumenten blockierend (vorausgesetzt der Erzeuger kann noch weitere Ressourcen erzeugen – Speicherplatz).

Serialisierung von Operationen mit Konflikt-Potential:

- off-line: analytischer Ansatz, statische Scheduling-Entscheidungen, implizite Synchronisation
- on-line: konstruktiver Ansatz in Form eines nebenläufigen Programms, explizite Synchronisation

Synchronisations-Ansätze:

- Blockierend: sequentialisierte Synchronisation zu Beginn (Locking Protocol), erlaubt meist Code-Wiederverwendbarkeit
- Nicht-Blockierend: nebenläufige Synchronisation zum Ende (Transaktion), erlaubt selten Code-Wiederverwendbarkeit

Nicht blockierende Verfahren sind portabler durch die Verfügbarkeit entsprechender Hardware-Befehlssätze, die überall in irgendeiner Art existieren.

Gleichzeitigkeit:

- concurrency: zeitgleich und im selben Platz, kausal unabhängige Ereignisse
- simultaneity: zeitgleiche Existenz, erzeugt kritisch überlappende Berechnungen, schließt concurrency mit ein.
- synchronism: mehrere Unterschritte einer komplexen Operation - erreicht durch Umwandlung in Elementaroperationen (elementary operation - ELOP) kohärenter Instruktionen

3 Processes

Bestimmtheit eines sequentialen Prozesses über die gesamte Aktions-Reihenfolge

Interagierenden Prozessen:
(erfolgt durch Variable oder geteilte Ressource). Kann zu einem Konflikt führen, wenn mindestens einer der Prozesse

- den Wert der geteilten Variable verändert (access pattern)
- oder bereits nicht unterbrechbare Ressourcen belegt (resource type)

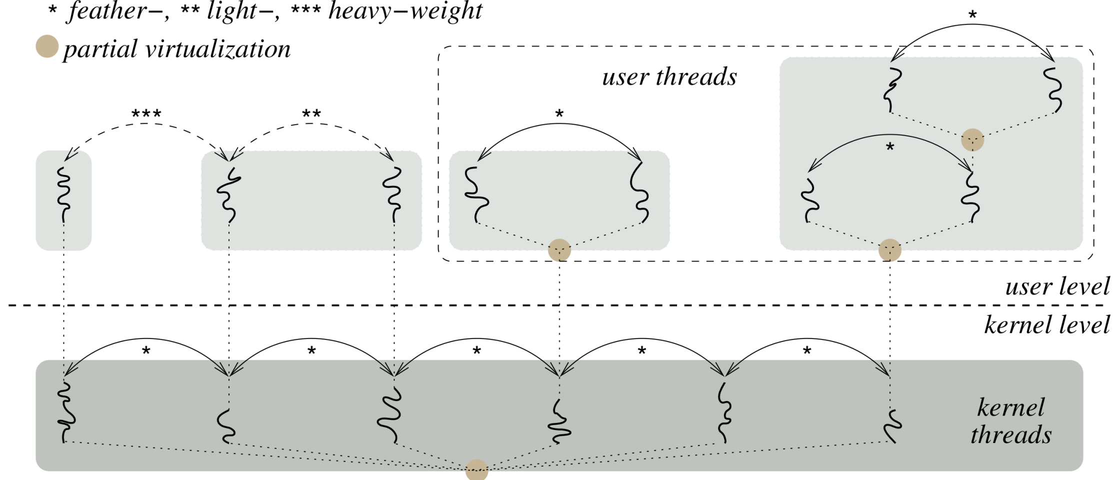


Abbildung 1: Gewichts Kategorien

- * in dem selben Adressraum
- ** Im selben Kernel-Adressraum, geteilter User-Adressraum
- *** selber Kernel-Adressraum, verschiedene User-Adressräume
- Kontextwechsel in den Kernel erfordert beim zurückgehen in den User-Mode einen Check, ob der vorherige User-Thread im selben Adressraum lief. Wenn ja, dann muss gewechselt werden

4 Critical Sections

race management:
Schützen von kritischer Abschnitte

(non-)critical race:
Fehler der in dem System zum Failure führt, vs. unerwartetes Verhalten.

Gelegentlich kann man mit Heisenbug leben, wenn man die behebenden Dummy-Instruktionen beibelässt.

Sequentialisierung mithilfe von ELOPs (atomic execution) durch den Prozessor oder die Unteilbarkeit gewisser Ressourcen

ACID - atomicity consistency isolation durability:
Soll die Verlässlichkeit von Systemen gewährleisten
Atomar - eine Transaktion ist entweder vollständig oder gar nicht passiert
Konsistenz - Ausschließlich gültige Ergebnisse werden übermittelt
Isolation - Ereignisse sind für andere zeitgleich laufende Transaktionen unsichtbar
Dauerhaftigkeit - Eine einmal getätigte Transaktion lässt sich nicht zurückspulen.

Isolation der Transaktion:
Mögliche Formen der Inkonsistenz

- nicht wiederholbarer Lesezugriff - verschiedene Lese-Operationen führen zu unterschiedlichen Ergebnissen
  read → write Abhängigkeit
- dirty read - un*comit*ete Daten wurden bereits gelesen
  write → read dependency
- lost update - bereits *comit*ete Daten wurden überschrieben
  write → write Abhängigkeit

```
1 bool tas(byte *ref){
2     bool aux;
3     atomic {aux = *ref & 0x1; *ref = 0x11111111;}
4     return aux;
5 }
```

Listing 1: TAS Unconditional Store, zerstört den Cache aufgrund des unbedingten Schreibens

Die bitweise Verundung könnte man sicherlich noch zum return runterziehen und somit den atomaren Block verkleinern

5 Elementary Operations

- Atomarer Zugriff auf Adressen sofern diese Aligned sind.
- Um Atomarität von load-compute-store (Read-Modify-Write - RMW) zu erzwingen wird häufig der Bus gesperrt

Test and Set (TS, TAS) listing 1:

Das linkeste Bit wird verwendet um den Zustand zu überprüfen
Der Befehl: *xchgl*
Der Bus wird gelockt und die Cache-Line, die den Main-Speicher-Operanden enthält wird dabei invalidiert. Dies resultiert in Performanz-Einbruch.

Prinzipiell kann man argumentieren, dass bei einem TAS die ABA-Problematik(s.u.) nicht sinnvoll auftreten kann. Das könnte es zwar, aber dann würde man mit TAS etwas synchronisieren wollen, was für ein TAS ohnehin zu komplex ist. Die Beschränkung der Relevanz auf ein einzelnen Bit (bzw. zwei mögliche Werte) ermöglicht erst das unbedingte Schreiben. Sind darüber hinaus weitere Zustände für die Überprüfung relevant, dann ist TAS ohnehin das falsche Werkzeug.

Bedingtes Schreiben ist allerdings bei Kombination mit anderen Techniken denkbar. Beispielsweise DPRAM, LL/SC (s.u.).

```
1 word tas(word *ref){
2    word aux;
3    atomic {if ((aux = *ref) == 0) *ref = 1;}
4    return aux;
5 }
```

Listing 2: TAS conditional store mit DPRAM

```
1 atomic bool cas(word old, word *ref, word new){
2    return (*ref == old) ? (*ref = new, true) : false;
3 }
```

Listing 3: CAS conditional store. Man beachte das atomic.

Dual-Ported RAM (DPRAM) listing 2:

Ein Random-Access-Memory (RAM), der nebenläufigen load/store Zugriff erlaubt und die Caches nicht pausenlos kaputt schreibt.

1. Logt den Prozessor, der das TAS durchführt
2. Signalisiert den restlichen Prozessoren die nebenläufige TAS-Durchführung mittels "Busy"-Interrupt und zwingt Prozessoren in busy waiting
3. Testet und nur dann wenn Test erfolgreich:
   a) Setzt den Speicherinhalt
   b) Gibt die Freigabe für diesen Speicherbereich

Compare and Swap (CS, CAS) listing 3:

Vergleich des ersten und zweiten Operanden. Gleich ? dritter Operand wird im zweiten gespeichert : zweiter Operand wird in den ersten geladen.

ABA, A-B-A:

False positive Ausführung einer CAS-basierten Annahme auf geteiltem Speicher. Die Gleichheit zweier Operanden soll die Erfülltheit von Bedingungen garantieren, die dennoch nicht erfüllt sein müssen.

```
1 int tas(long *ref){
2    return (LL(ref) == 0) && SC(ref,1);
3 }
4 int cas(long *ref, long old, long new){
5    return (LL(ref) == old) && SC(ref,new);
6 }
```

Listing 4: TAS und CAS mithilfe von LL und SC

Load-Linked/Store-Conditional listing 4:
Gepaarte Instruktionen, um eine Befehls-Abfolge auszuführen, ohne dabei Unteilbarkeit zu garantieren. Deren Abschluss allerdings lediglich bei ungeteiltem Ablauf erfolgreich ist.

- LL
  Lädt ein Wort von einer Speicheradresse und reserviert diese
- SC
  Überprüft die Reservierung einer Speicheradresse. Besteht diese weiterhin wird dort ein Wort gespeichert. Rückgabe ist das Ergebnis der Reservierungsüberprüfung. Gründe für die Stornierung einer bestehenden Reservierung:
  - SC wurde erfolgreich durchgeführt
  - Ausführung einer LL auf der selben Adresse durch einen anderen Prozessor
  - Exception auf dem Prozessor, der die Reservierung hält

Entsprechende Absicherung von LL und SC-Sperre gegen nebenläufige Speicherzugriffe. Folgende Implementierung (listing 4) ist frei vom ABA-Problem, weil der Verzögerte ein Roll-Back vornimmt.

Fetch and Add:

Instruktion, die einen Wert zurückgibt, der auf einer geteilten Variable G und einer lokalen L arbeitet. Spiegelt eine atomare RMW-Instruktion dar. Siehe auch Legacy __sync Built-in Functions for Atomic Memory Access

- prefix (FAA)
  Speichert alten Wert von G für die Rückgabe, addiert L auf G
- postfix (AAF)
  Addiert L auf G, gibt den neuen Wert von G zurück

$$\begin{aligned} FAA(G, L) &\equiv AAF(G, L) - L \\ AAF(G, L) &\equiv FAA(G, L) + L \end{aligned}$$

Kann in entsprechenden fetch-and-ϕ umgewandelt werden, wobei für nicht-invertierbare Operationen die Prefix Form verbreiteter ist.

$$\phi = \max, \qquad XAF(G, L) \equiv \max(FAX(G, L), L)$$

Consensus Number n:

Ist die maximale parallel mit sich selbst interagierende Ausführung eines Algorithmuses, für die der Algorithmus funktioniert, ohne an Korrektheit zu verlieren. Petterson's Algorithmus für $N = 2$ bricht für $N > 2$, weist also eine Consensus Number von 2 auf.

So ganz scheint das aber nicht zu stimmen, es geht wohl mehr darum sich auf einen Wert zu einigen, da die Beispiele nicht ganz mit der Erklärung matchen. TAS und FAA sollten ja für beliebig viele parallele Prozesse funktionieren.

Beispiele:

- ∞: CAS, LL/SC
- 2: TAS, swap, FAA
- 1: atomic read, atomic write

Memory Barriers:

Memory Barrier kontrollieren ausschließlich die Kommunikation mit CPUs, Cache und write-buffer, der den Wert beinhaltet, der in den Speicher geflusht werden soll (bzw. auf den wartenden). LINUX KERNEL MEMORY BARRIERS

- ld_a LoadLoad ld_b
  Sichert das Laden von a vor dem Zugriff auf b zu.

- st_a StoreStore st_b
  Sichert die Sichtbarkeit von a vor dem flushen von b zu. (Häufig in Kombination mit loadload-Barriers zu verwenden)
- ld_a LoadStore st_b Sichert die Abgeschlossenheit des Lesevorgangs von a vor dem flush von b zu.
- st_a StoreLoad ld_b
  Sichert die Sichtbarkeit von a vor dem Zugriff auf b zu.

Typischerweise verwenden CAS, LL/SC eine StoreLoad-Barriere. Also die teuerste Barriere.

Ganz allgemein gilt: Sollen die Commits auch auf anderen CPUs in einer spezifischen Reihenfolge gesehen werden, dann müssen dort ebenfalls Memory-Barriers verwendet werden (häufig die Gegenstücke). Wichtig ist auch, dass Verschiebungen über Condition-Grenzen hinweg stattfinden können. Wenn in den Conditions nun aber Funktionen stehen, die andere Variablen verändern, dann braucht es Fences.

Man findet häufig acquire und release, wobei acquire dafür sorgt, dass keine Memory-Operationen von unten über die Barrier nach oben gezogen werden dürfen. Wohingegen release dafür sorgt, dass keine Memory-Operationen oberhalb der Barrier über diese hinweg nach unten geschoben werden dürfen. Es gilt also zwischen Sichtbarkeit für andere CPUs und Instruktionsumordnungen des Compilers zu differenzieren.

Sichtbarkeit auf einer anderen CPU kann mit einer Kombination aus Write und Read-Barriers erreicht werden. Sichtbarkeits-Transititvität lässt sich im Kernel lediglich mit general-barriers erreichen. Die im Folgenden verwendeten READ_ONCE, WRITE_ONCE sind wohl im Grunde volatile-Casts. Ich glaube es erfolgt bei jedem Dispatch-Vorgang die Verwendung einer Memory-Barrier. Das kann unter Umständen zu unerwartetem Verhalten führen. Wichtig für das Ausbleiben von fehlerhaften Optimierungen bei I/O-Geräten (Memory Mapped) ist deren geeignete Zugriffsroutinen zu verwenden (*inX(), outX(), readX(), writeX()*).

```
CPU 1                  CPU 2
k===============        ===============
{ A == 1, B == 2, C = 3, P == &A, Q == &C }
B = 4;
<write barrier>
WRITE_ONCE(P, &B)
                        Q = READ_ONCE(P);
                        D = *Q;
```

```
There's a clear data dependency here, and it would seem that by the end of the
sequence, Q must be either &A or &B, and that:

            (Q == &A) implies (D == 1)
            (Q == &B) implies (D == 4)
```

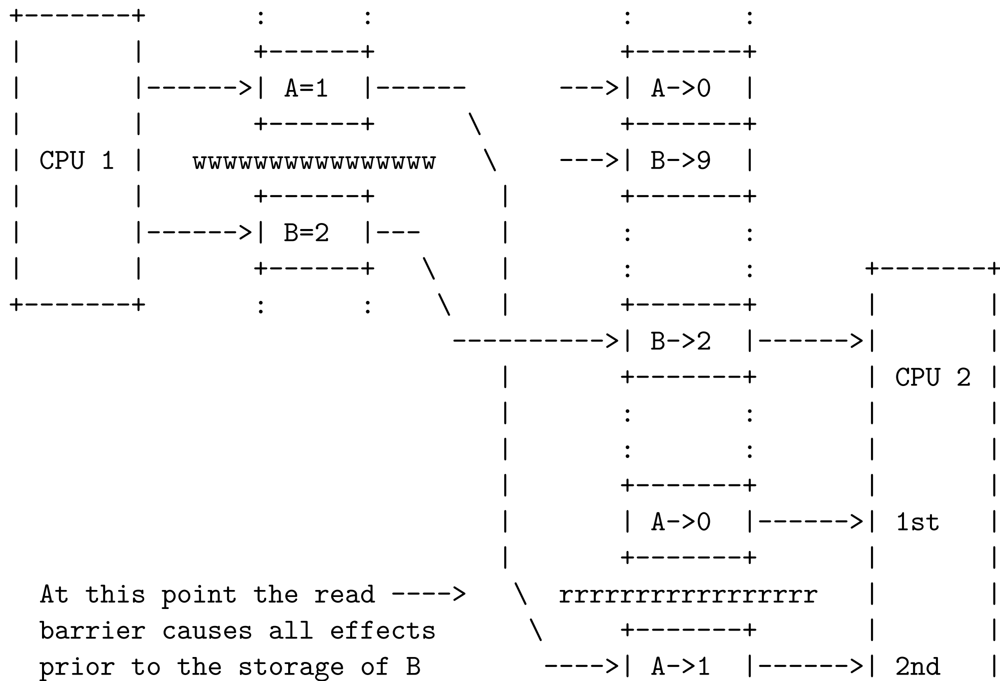
But! CPU 2's perception of P may be updated _before_ its perception of B, thus leading to the following situation:

```
(Q == &B) and (D == 2) ????
```

Behoben wird das unerwartete Verhalten durch eine <data dependency barrier> zwischen READ_ONCE und der zweiten Zuweisung der CPU2

```
CPU 1                     CPU 2
====================    =======================
             { A = 0, B = 9 }
STORE A=1
<write barrier>
STORE B=2
                        LOAD B
                        LOAD A [first load of A]
                        <read barrier>
                        LOAD A [second load of A]
```

Even though the two loads of A both occur after the load of B, they may both come up with different values:



```
  to be perceptible to CPU 2           +-------+      |       |
                                       :       :      +-------+
```

Sichtbarkeit wird mit der Kombination aus Write und Read-Barriers erreicht.

```
The compiler is within its rights to merge successive loads from
the same variable.  Such merging can cause the compiler to "optimize"
the following code:

            while (tmp = a)
               do_something_with(tmp);

into the following code, which, although in some sense legitimate
for single-threaded code, is almost certainly not what the developer
intended:

            if (tmp = a)
               for (;;)
                  do_something_with(tmp);

Use READ_ONCE() to prevent the compiler from doing this to you:

            while (tmp = READ_ONCE(a))
               do_something_with(tmp);
```

```
The compiler is within its rights to reload a variable, for example,
in cases where high register pressure prevents the compiler from
keeping all data of interest in registers.  The compiler might
therefore optimize the variable 'tmp' out of our previous example:

            while (tmp = a)
               do_something_with(tmp);

This could result in the following code, which is perfectly safe in
single-threaded code, but can be fatal in concurrent code:

            while (a)
               do_something_with(a);

For example, the optimized version of this code could result in
```

```
passing a zero to do_something_with() in the case where the variable
a was modified by some other CPU between the "while" statement and
the call to do_something_with().

Again, use READ_ONCE() to prevent the compiler from doing this:

            while (tmp = READ_ONCE(a))
               do_something_with(tmp);



The compiler is within its rights to omit a load entirely if it knows
what the value will be.  For example, if the compiler can prove that
the value of variable 'a' is always zero, it can optimize this code:

            while (tmp = a)
               do_something_with(tmp);

Into this:

            do { } while (0);

This transformation is a win for single-threaded code because it
gets rid of a load and a branch.  The problem is that the compiler
will carry out its proof assuming that the current CPU is the only
one updating variable 'a'.  If variable 'a' is shared, then the
compiler's proof will be erroneous.  Use READ_ONCE() to tell the
compiler that it doesn't know as much as it thinks it does:

            while (tmp = READ_ONCE(a))
               do_something_with(tmp);



Similarly, the compiler is within its rights to omit a store entirely
if it knows that the variable already has the value being stored.
Again, the compiler assumes that the current CPU is the only one
storing into the variable, which can cause the compiler to do the
wrong thing for shared variables.  For example, suppose you have
the following:

            a = 0;
            /* Code that does not store to variable a. */
            a = 0;
```

```
The compiler sees that the value of variable 'a' is already zero, so
it might well omit the second store.  This would come as a fatal
surprise if some other CPU might have stored to variable 'a' in the
meantime.

Use WRITE_ONCE() to prevent the compiler from making this sort of
wrong guess:

            WRITE_ONCE(a, 0);
            /* Code that does not store to variable a. */
            WRITE_ONCE(a, 0);



Use of packed structures can also result in load and store tearing,
as in this example:

            struct __attribute__((__packed__)) foo {
               short a;
               int b;
               short c;
            };
            struct foo foo1, foo2;
            ...

            foo2.a = foo1.a;
            foo2.b = foo1.b;
            foo2.c = foo1.c;

Because there are no READ_ONCE() or WRITE_ONCE() wrappers and no
volatile markings, the compiler would be well within its rights to
implement these three assignment statements as a pair of 32-bit
loads followed by a pair of 32-bit stores.



Cache-Kohärenz: Erzwingen, dass CPU2 die v-Aktualisierung vor der
p-Aktualisierung sieht. Caches der CPU 1: A, B, der CPU 2: C,D

    CPU 1               CPU 2               COMMENT
    ===============     ===============     ========================================
                                            u == 0, v == 1 and p == &u, q == &u
    v = 2;
```

```
smp_wmb();
<A:modify v=2>     <C:busy>
                   <C:queue v=2>
p = &v;            q = p;
                   <D:request p>
<B:modify p=&v>    <D:commit p=&v>
                   <D:read p>
                   smp_read_barrier_depends()
                   <C:unbusy>
                   <C:commit v=2>
                   x = *q;
                   <C:read *q>       Reads from v after v updated in cache
```

Consistency Models:

- sequential
  Prozessoren sehen Schreibzugriffe auf das selbe Ziel in der korrekten Reihenfolge, externe Beobachter womöglich nicht
  Bedingungen: Programm-Reihenfolge, Schreib-Atomarität
- relaxed
  - Programm-Reihenfolge - write to read, write to write, read to read, read to write
    Bezogen auf unterschiedliche Speicherbereiche
  - Schreib-Atomarität - Lese das andere, schreibe frühzeitig
    Bezogen auf den selben Speicherbereich
  - oder Beides - Lese eigenes, schreibe früh
- weak
  Konsistente Freigabe und Eintritte. Vom OS in System Machine Level Programs implementiert. Typischerweise nicht vom Instruction Set Architecture Level bereitgestellt

Heutzutage werden weak- und relaxed-Konsistenz-Modelle von Prozessoren bereitgestellt

ABA Design Risc Reduction:

- Eine Art Versionskontrolle bei jeder CAS-Änderung
- compare double and swap (CDS)
- double compare and swap (DCAS, CAS2)

```
#define INLINE extern inline
INLINE
long LL(long *ref){
    long aux;
    asm volatil(
        "lwarx %0, 0, %1"
        : "=r" (aux)
        : "r" (ref));
    return aux;
}
```

Listing 5: LL Implementierung

```
#define INLINE extern inline
INLINE
int SC(long *ref, long val){
    long ccr;
    asm volatile(
        "stwcx. %2, 0, %1\n\t"
        "mfcr %0"
        : "=r" (ccr)
        : "r" (ref), "r" (val)
        : "cc", "memory");
    return ccr & 0x2
}
```

Listing 6: SC Implementierung

```
#define FAA __sync_fetch_and_add
int faa(int *p, int v){
    return FAA(p,v);
}
```

Listing 7: FAA Implementierung

```
#define AAF __sync_add_and_fetch
int aaf(int *p, int v){
    return AAF(p,v);
}
```

Listing 8: AAF Implementierung

```
//sicheres Laden der globalen
    Variable G und bedingtes
    Speichern von max(G,L) in G
word fax(word *ref, word val){
    word aux;
    atomic {if ((aux = *ref) < val)
        *ref = val;}
    return aux;
}
```

Listing 9: fax - conditional store

```
//bedingtes Speichern von max(G,L)
    in G und Rückgabe von max(G,L)
word xaf(word *ref, word val){
    word aux;
    atomic {aux = (*ref > val) ?
        *ref : *ref = val;}
    return aux;
}
```

Listing 10: xaf - conditional store

6 Locks

Lock:

Unterbrechbarkeit trotz Lock muss gewahrt bleiben. Entsprechend entweder Interrupts nicht deaktivierbar gestalten oder Unterbrechung via Traps vornehmen nach einer bestimmten Anzahl von Ticks. Dh. Instruction-Trap-Handler muss verfügbar sein.

- disabled Interrupts und lockt den Speicher/Bus
- Startet Timout Mechanismen (vermeiden von Deadlock, einhalten von WCET)

Unteilbarkeit:

- logische Ebene
  - Der Abschnitt kann verzögert werden
  - erschwert Vorhersagbarkeit
  - Erlaubt nebenläufige Prozesse
- physikalische Ebene
  - Interrupts und Bus gelockt, entsprechend Verzögerungsfrei
  - erleichtert Vorhersagbarkeit
  - Deaktiviert nebenläufige Prozesse
- Heutige Prozessoren stellen kaum noch hardware ELOPs bereit. Entsprechendes muss mittels Algorithmischer Lösungen erfolgen
- Bei der Wahl welcher Prozess den kritischen Abschnitt betritt sollte evtl. der Scheduler befragt werden, um dessen Entscheidungen nicht zu behindern.

Betriebsart, Prozessregelung für Ausschluss:

- Advisory - beratend
  - Explizites Lock durch den kooperierenden Prozess
  - Lock erfordert explizites unlock
  - Niedrige Abstraktionsebene
- Mandatory - zwingend, vorgeschrieben
  - Lock erfolgt implizit durch Seiteneffekte einer komplexen Operation
  - Werden intern (meist) mittels Advisory-Locks realisiert
  - Höhere Abstraktionsebene

Overhead vs. Contention bei Locks:

Overhead:
Aufwand zum Locken und Freigeben eines kritischen Abschnitts für einen Prozess. Erhöht sich mit der Anzahl notwendiger Locks.

Contention (Wettstreit):
Wettbewerb mit den weiteren Prozessen, die den kritischen Abschnitt betreten möchten. Erhöht sich mit der Anzahl interagierender Prozesse.

Ein Grober kritischer Abschnitt sorgt für geringeren Overhead und höheren Wettstreit. Feinere kritische Abschnitte sorgen für größeren Overhead und geringeren Wettstreit.

Man möchte also einen Kompromiss finden zwischen kleiner Overhead für kurze kritische Abschnitte vs Untergrundrauschen der Lockingverfahren für größere, stärker sequentialisierte Abschnitte.

Die Unteilbarkeit der folgenden Lock-Algorithmen gilt nicht auf Prozessorsicht! Es werden entsprechend Fences/Barriers benötigt. Möchte man die Algorithmen für die Fälle $N > 2$ anpassen, so ist Unterstützung spezieller CPU-Instruktionen notwendig.

- atomares lesen-modifizieren-schreiben: TAS, CAS, FAA
- load/store: LL/SC
- Cache-Verhalten und Speichermodell beeinflussen mögliche Lösungen

```
1 #define NPROC 2
2 #ifdef __FAME_LOCK_KESSEL__ //s.u. Kessel-Algorithmus
3 #define NTURN NPROC
4 #else
5 #define NTURN (NPROC -1)
6 #endif
7 typedef volatile struct lock{
8    bool want[NPROC];    //Interesse signalisieren, initial false
9    char turn[NTURN];    //Lock-Besitzer (heißt nicht er ist drin), initial 0
10 } lock_t;
11 inline unsigned earmark(){
12    return hash_of_process_ID; //for [0, NPROC-1]
13 }
14
15 void lock(lock_t *bolt){
16    unsigned self = earmark();        //eigener Prozess-Index
17
18    bolt->want[self] = true;          //signalisiere eigene Interesse
19    while(bolt->want[self^1])         //der Andere ist interessiert
20       if(bolt->turn[0] != self){     //der Andere hat das Lock und damit das
          Recht den kritische Abschnitt zu betreten, sofern er sich nicht
          ohnehin darin befindet
21          bolt->want[self] = false;  //anuliere eigene Interesse
22          while(bolt->turn[0] != self); //Ich warte
23          bolt->want[self] = true;   //nochmals versuchen
24       }
25 }
26
27 void unlock(lock_t *bolt){
28    unsigned self = earmark();     //eigener Prozess-Index
29    bolt->turn[0] = self^1;        //dem Anderen den Zutritt gewähren
30    bolt->want[self] = false;      //selbst nicht mehr interessiert
31 }
```

Listing 11: Dekker's Algorithmus für $N = 2$, altruistisch, Überholung kann stattfinden. Sehr interessant ist, dass das turn[0] lediglich dann von Interesse ist, wenn beide den kritischen Abschnitt zeitgleich betreten wollen. Andernfalls wird turn überhaupt nicht zum Betreten benötigt.
- want[0] := false, want[1] := false, turn[0] := 0
Beispiel:
- 1 möchte betreten: want[1] := true, da 0 nicht interessiert, betritt 1 den kritischen Abschnitt
- 0 möchte jetzt auch den kritischen Abschnitt betreten (und verfügt über turn). 0 sieht: 1 ist interessiert und idled in Zeile 19 herum, da die Bedingung in Zeile 20 nicht erfüllt ist.

Beispiel zeitgleich:
- want[0] := true, want[1] := true, turn[0] := 1
- 1 sieht: 0 ist interessiert, die Bedingung in Zeile 20 ist jedoch nicht erfüllt
0 sieht: 1 ist interessiert und 0 verfügt nicht über das Lock. 0 anuliert das eigene Interesse und wartet.
- 1 sieht: ich bin der einzige Interessent und betritt den kritischen Abschnitt

```
1  void lock(lock_t *bolt){
2     unsigned self = earmark();          //eigener Prozess-Index
3
4     bolt->want[self] = true;            //signalisiere eigene Interesse
5     bolt->turn[0] = self;               //Ich möchte gern der nächste sein
6     //Hier wird das Warten beendet sobald der Andere nicht mehr interessiert
          ist. Das Lock wird nie direkt freigegeben, sondern dient hier
          lediglich dazu den Konflikt aufzulösen. Durch die Reihenfolge want ->
          turn ist das korrekt.
7     while(bolt->want[self^1]            //der Andere ist interessiert
8       && (bolt->turn[0] == self));      //der Andere hat das Lock
9  }
10
11 void unlock(lock_t *bolt){
12    unsigned self = earmark();          //eigener Prozess-Index
13    bolt->want[self] = false;           //selbst nicht mehr interessiert
14 }
```

Listing 12: Peterson's Algorithmus für $N = 2$, egoistisch, Überholung ist ausgeschlossen. Wer schneller erstes turn[0] = self setzt erhält als erstes Zugang zum freien kritischen Abschnitt.

```
1  #define __FAME_LOCK_KESSEL__
2  void lock(lock_t *bolt){
3     unsigned self = earmark();      //eigener Prozess-Index
4
5     bolt->want[self] = true;        //signalisiere eigene Interesse
6     //Vergabe des Locks erfolgt anhand des Vorherigen Besitzers
7     bolt->turn[self] = ((bolt->turn[self^1] + self) % 2);
8     while (bolt->want[self^1] &&
9           (bolt->turn[self] == ((bolt->turn[self^1]+self)%2)));
10    //)))) <-fix vim highlighting, lol
11 }
```

Listing 13: Kessel's Algorithmus für $N = 2$, schreibt nicht (nur lokale Daten), sondern ließt und zerstört entsprechend keine Cache-Lines und ermöglicht read-only sharing

Verhungerungsfrei:

- Durch aufgerufene Prozesse: bottom up, logische ELOP
- Durch aufrufende Prozesse: top down, physikalische ELOP

```
1 void lock(lock_t *bolt){
2     bool busy;
3     do atomic{          //Hardware ELOP
4         if(!(busy = bolt->busy))
5             bolt->busy = true;
6     }while(busy);
7 }
```

Listing 14: Spin-Lock

```
1 void lock(lock_t *bolt){
2     while(!TAS(&bolt->busy));
3 }
```

Listing 15: Spin-Lock mit TAS

Probleme bei der TAS-Implementierung listing 15:

- unbedingtes Schreiben löscht den Cache *deleterious effect*
- Hoher Bus-Traffic wird verursacht durch invalidieren, updaten, ...
- Durchgehende TAS-Instruktion in der Schleife blockiert andere Prozessoren dabei den Bus zu nutzen um auf Speicher oder andere Devices zuzugreifen. *(Access Contention)*

```
1 void lock(lock_t *bolt){
2     while(!CAS(&bolt->busy, false, true));
3 }
```

Listing 16: Spin-Lock mit CAS

Probleme bei der CAS-Implementierung listing 16:

- Behebt das Problem des unbedingten Speicherns
- *Access Contention* besteht aber weiterhin, der Bus bleibt gelockt.

```
1 void lock(lock_t *bolt){
2     do{
3         while(bolt->busy);                //Bus nicht gelockt
4     }while(!CAS(&bolt->busy, false, true));
5 }
```

Listing 17: Spin on Read

Probleme bei der Spin on Read-Implementierung listing 17:

- Vermindert unter Umständen massiv das Access Contention, da der Bus nicht mehr pausenlos gelockt wird.
- Kann dennoch zu einem Spin auf dem CAS führen, wenn die Ausführungszeit des kritischen Abschnitts relativ kurz ist.
- Prozesse mit vergleichbaren Aufgaben verhalten sich häufig ähnlich und führen folglich zu *Bus Lock Bursts*, da alle auf das Lock wartende auf einmal versuchen dieses zu erhalten.
- Macht aber primär nur dann Sinn, wenn die Kritischen Abschnitte länger sind. Weil nur dann Gewinn erzielt wird, wenn längere Wartezeiten vorliegen.

Backoff als Lösung:
Statische (ALOHA) oder dynamisch (Ethernet) justierte Wartezeiten bei Misserfolg. Sollte entsprechend auf das Scheduling-Planungen matchen.

verifiziere Korrektheit dieses Peterson-Codes

Peterson's Algorithmus für $N > 2$, listing 21:
Jeder Prozess muss sich selbst für $n-1$ Ränge beweisen bevor er die Berechtigung zum Betreten erhält. Dabei ist die Grundidee, dass die zwei Prozess-Lösung auf jedem Rang wiederholt wird. Dabei wird schrittweise jeweils mindestens ein Prozess eliminiert, bis lediglich einer verbleibt. Entsprechend wird das Interesse signalisiert den nächsten Rang zu erreichen und überprüft ob es noch höher rangige Prozesse gibt als der eigene.

verifiziere Korrektheit dieses Lamport Bakery-Codes

```
1 typedef volatile struct lock{
2    bool busy;
3    long (*rest)[];
4 } lock_t;
5
6 void backoff(lock_t *bolt, int hold){
7    if(bolt->rest)
8       rest((*bolt->rest)[earmark()] * hold);
9 }
10 //im Priveligierten Modus zieht man dem womöglich eher ein halt
     vor.
11 long rest(volatile long term){
12    while(term--); //busy waiting
13    return term;
14 }
15
16
17 void lock(lock_t *bolt){
18    while(!CAS(&bolt->busy, false, true))
19       backoff(bolt, 1);
20 }
```

Listing 18: Spin-Lock with Backoff

```
1 void lock(lock_t *bolt){
2    int hold = 1;
3    do{
4       while(bolt->busy);
5       if(CAS(&bolt->busy, false, true)) break;
6       backoff(bolt,hold);
7       if((hold << 1) != 0) hold <<=1;
8    }while(true);
9 }
```

Listing 19: Spin with Backoff and spin on read, Verwendet zusätzlich Feedback (shift), um die Anzahl zeitgleich Wettstreitenden zu reduzieren.

```
typedef volatile struct lock{
    long next;   //Die letzte gezogene Nummer
    long this;   //Gerade behandelte Nummer
} lock_t;

void lock(lock_t *bolt, long cset){
    long self = FAA(&bolt->next, 1);
    if(self != bolt->this){
        rest((self - bolt->this) * cset);
        while(self < bolt->this);
    }
}
void unlock(lock_t *bolt){
    bolt->this++;
}
```

Listing 20: Ticket Lock. Hier ist es sinnvoll die Critical section execution time (CSET) zu kennen, um gute BCET,ACET,WCET zu erreichen. Dies ist möglich, indem der Programmierer dies in irgend einer Form notiert.
Ticket Locks garantieren Verhungerungsfreiheit beim Warten auf das Lock.

```
1  typedef volatile struct lock{
2     unsigned want[NPROC];
3     unsigned turn[NTURN];
4  } lock_t;
5
6  void lock(lock_t *bolt){
7     unsigned rank, next, self = earmark();
8     for(rank = 0; rank < NPROC- 1; rank++){
9        //signalisiere Interesse am nächsten Rang
10       lock->want[self] = rank;
11       lock->turn[rank] = self;
12       //überprüfe den Rang aller Prozesse, die an dem nächsten
            Rang interessiert sind
13       for(next = 0; next < NPROC; next++)
14          if(next != self)
15             //es liegen noch höher rangige Prozesse vor
16             while((lock->want[next] >= rank)
17                && (lock->turn[rank] == self));
18    }
19 }
20
21 void unlock(lock_t *bolt){
22    unsigned self = earmark(); //eigener Prozess-Index
23    bolt->want[self] = -1;  //selbst nicht mehr interessiert
24 }
```

Listing 21: Peterson's Algorithmus für $N > 2$

```
typedef volatile struct lock{
    bool want[NPROC];
    long turn[NPROC];
};

inline void ticketing(lock_t *bolt, unsigned slot){
    unsigned next, high = 0;
    bolt->want[slot] = true;          //Auswahl betreten
    for(next = 0; next < NPROC; next++)
        if(bolt->turn[next] > high)
            high = bolt->turn[next];
    bolt->turn[slot] = high+1;        //Slave Nummer
    bolt->want[slot] = false          //verlasse Auswahl
}

void lock(lock_t *bolt){
    unsigned next, self = earmark();
    ticketing(bolt, self);            //ziehe eine Nummer
    for(next = 0; next < NPROC; next++){
        while(bolt->want[next]);      //nächste Auswahl
        //zuerst den nächsten wählen
        while((bolt->turn[next] != 0)
            &&((bolt->turn[next] < bolt->turn[self])
                || ((bolt->turn[next] == bolt->turn[self])
                  &&(next < self))));
    }
}

void unlock(lock_t *bolt){
    unsigned self = earmark();
    bolt->turn[self] = 0;
}
```

Listing 22: Lamport's Bakery Algorithmus, verwendet Tickets

7 Semaphore

- Mutex: Binäre Semaphore mit expliziter Überprüfung der Berechtigung bei der Freigabe eines kritischen Abschnitts. Sollte eine Verletzung beobachtet werden sollte der entsprechende Prozess abgeschossen werden und nicht, wie in den meisten Implementierungen eine Fehlermeldung ausgegeben werden
- Auch hier sollte weder bei Mutex noch bei Semaphore die Schedule-Strategie sabotiert werden
- Block kann erfolgen:
  - logisch
    Magische Adresse im Prozess-Deskriptor, P: konstant, V: beschränkt
  - physisch
    Prozess-Deskriptor in eine Datenstruktur, P: beschränkt, V: konstant
- und
  - Einseitig: Processor Multiplexing - Interrupt Kontrolle
  - Gegenseitig: Processor Multiplication - Prozess-Locks
  - Kombination aus beidem

8 Monitor

verschiedene Signalisierungsformen:

- Hansen (Original) $\mapsto$ signal and wait listing 28. Alle werden signalisiert, entsprechend tolerant für fehlerhafte Signalisierung. Verlässt Monitor vor Signalisierung und betritt diesen im Anschluss erneut.
- Hoare $\mapsto$ signal and urgent wait listing 29. Fehlerhafte Signalisierung nicht tolerierbar, da explizit einer signalisiert wird. Toleriert werden kann dies ebenfalls durch die Verwendung von while, anstelle von if bei der Condition-Überprüfung. Verlässt Monitor vor Signalisierung und betritt diesen im Anschluss erneut.
- Mesa $\mapsto$ signal and continue listing 26. Benötigt weniger Kontext-Wechsel. Bleibt im Monitor. Hat den Vorteil, dass weniger Kontext-Wechsel notwendig sind. Dafür sind die Signal-Allokationskosten höher.
- Hansen (Concurrent Pascal) $\mapsto$ signal and return listing 27. Verlässt den Monitor und signalisiert einen Prozess.
- automatische Signalisierung als Signalisierungsprimitive vermeidet Aufwecken von Prozessen, die gar nicht dran kommen. Ist aber extrem aufwendig und daher unpraktikabel, auch wenn Programmierfehler vermieden werden.
- Monitor sollte nicht von einem reinen Mutex gesichert werden, da Besitzerwechsel erfolgen können.
- In der Realität wird das vielleicht eher in Assemblee programmiert (oder aber in C++)

Selbstverständlich können bei Monitoren andere Prozesse das Lock freigeben, als diejenigen, die es zuvor sperrten. Dies ist erforderlich, da zwischendrin Kontext-Wechsel stattfinden können

```
typedef struct monitor{
    semaphore_t mutex;
#ifdef __FAME_MONITOR_SIGNAL_URGENT_WAIT__
    lineup_t urgent;  //zwingend wartende Signalgeber
#endif
}monitor_t;

typedef struct condition{
    monitor_t *guard; //Monitor
    lineup_t event;   //Signal-erwartende Prozesse
}condition_t;

typedef struct lineup{
    int count;        //Anzahl wartender Prozesse
    event_t crowd;    //Das zu erwartende Ereignis
}lineup_t;

extern void lockout(monitor_t*); //enter monitor
extern void proceed(monitor_t*); //leave monitor

extern void watch(condition_t*); //Warte auf Signal
extern void spark(condition_t*); //Signal Bedingung
```

Listing 23: Monitor-Datentypen

```c
#define BUF_SIZE 80
typedef struct buffer{
    condition_t space;    //verwalte Wiederverwendbares
    condition_t data;     //verwalte Konsumierendes
    char store[BUF_SIZE];//Wiederverwendbare Ressource
    unsigned in,out;      //Puffer-Verwaltung, Produzent/Konsument
    unsigned count;       //Warte-Signal-Bedingung, Füllstand
}buffer_t;

static monitor_t storehouse = {1};
static buffer_t buffer={
    {&storehouse},{&storehouse}
};

void put(char item){
    lockout(&storehouse);    //Prologue
    //Betreten und Verlassen muss vom Compiler generiert werden
    {
        while(buffer.count == BUF_SIZE)
            watch(&buffer.space);
        buffer.store[buffer.in] = item;
        buffer.in = (buffer.in +1) % BUF_SIZE;
        buffer.count++;

        spark(&buffer.data);
    }
    proceed(&storehouse); //Epilogue
}

char get(){
    char item;
    lockout(&storehouse);    //Prologue
    {
        while(!buffer.count)
            watch (&buffer.data);
        item = buffer.store[buffer.out];
        buffer.out = (buffer.out +1)%BUF_SIZE;
        buffer.count--;

        spark(&buffer.space);
    }
    proceed(&storehouse);    //Epilogue
    return item;
}
```

Listing 24: Monitor-Verwendung, Bounded Buffer, Typen: listing 23

Event Queue listing 25:

- *catch*
  - Weniger anfällig für lost wakeup
  - Benachrichtigt potentielle Signalgeber
  - Sichert zu, dass ein laufender Prozess von *cause* bemerkbar ist
- *coast*
  - Blockiert an dem Event, wenn der Prozess noch nicht von *cause* bemerkt wurde
  - Sonst: Bereinigt den Catch-State und bleibt laufend
- *await*
  - Blockiert den Prozess an dem Event (signalisiert von cause)
- *cause*
  - Deblockiert Prozesse, die an einem Event warten

```
1 inline void brace(lineup_t *this){
2     this->count++;
3     catch(&this->crowd);            //bereit um zu de-/blockieren
4 }
5
6 inline void shift(lineup_t *this){
7     coast();                        //bedingte Blockierung
8     this->count--;
9 }
10
11 inline void defer(lineup_t *this){
12     this->count++;
13     await(&this->crowd);            //unbedingte Blockierung
14     this->count--;
15 }
16
17 inline int level(lineup_t *this){
18     return this->count;
19 }
20
21 inline int avail(lineup_t *this){
22     if(this->count > 0)             //wartende Prozesse?
23         cause(&this->crowd);        //unblock
24     return this->count;
25 }
26
27 //Erzwingt Prozesswechsel innerhalb eines gelockten Monitors
28 inline int evoke(lineup_t *this){
29     int count = this->count;            //sichere Status
30     if(count > 0)                       //wartende Prozesse?
31         admit(elect(&this->crowd));     //belege CPU, wobei elect
           den nächsten Prozess von der Warteliste wählt und
           admit den Prozess als laufbereit bucht.
32     return count;
33 }
```

Listing 25: Warteliste

```
1 void lockout(monitor_t *this){P(&this->mutex);}
2 void proceed(monitor_t *this){V(&this->mutex);}
3
4 //Freigabe des Monitors muss vor Freigabe des Prozessors
     erfolgen, Race-Condition muss verhindert werden!
5 void watch(condition_t *this){
6    brace(&this->event);     //Bereite Freigabe vor
7    proceed(this->guard);    //Gebe Monitor frei
8    shift(&this->event);     //Gebe Prozessor frei
9 }
10
11 void spark(condition_t *this){
12    avail(&this->event);     //Versuche Prozess zu signalisieren
13 }
```

Listing 26: Signal and Continue

```
1 //lockout, proceed, watch, wie bei signal and continue
2
3 //Muss die letzte Handlung einer Monitor-Prozedur sein
4 void spark(condition_t *this){
5    if(!avail(&this->event))       //kein Watcher wartet?
6       proceed(this->guard);       //gebe Monitor frei
7 }
```

Listing 27: Signal and Return

```
1 //lockout, proceed und watch wie bei signal and continue
2
3 void spark(condition_t *this){
4    //Im Fall von anhängenden Signalisierten unterbricht der
        Signaler die Ausführung
5    //Es erfolgt ein Prozesswechsel innerhalb des gelockten
        Monitors
6    if(evoke(&this->event))     //Signalisierter bearbeitet
7       lockout(this->guard);    //Wiederbetreten des Monitors
8 }
```

Listing 28: Signal and Wait

```
1 //lockout und watch wie bei signal and continue
2
3 //Hier müssen beim Verlassen des Monitors zwei Wartelisten auf
    anhängende Prozesse überprüft werden: ‘‘re-entrance’’
    (urgent) im Falle von urgent-Prozessen und sonst
    ‘‘entrance’’ (mutex)
4 void proceed(monitor_t *this){
5     if(!avail(&this->urgent))      //Keine vorrangig Wartenden
6         V(&this->mutex);           //Gebe Monitor frei
7 }
8
9 void spark(condition_t *this){
10    if(avail(&this->event))           //Wartet ein Beobachter?
11        defer(&this->guard->urgent);  //vorrangiges Warten
12 }
```

Listing 29: Signal and Urgent Wait

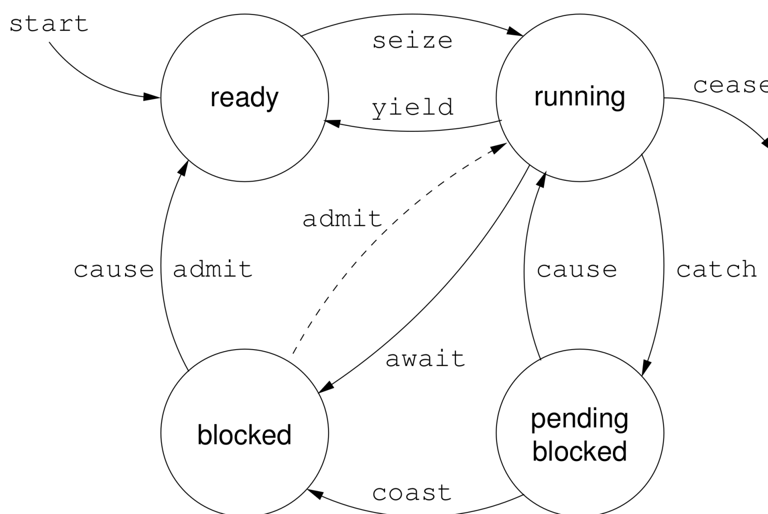


Abbildung 2: Prozess Status und Status-Transitionen

ready $\leftrightarrow$ *running*: wait ($\leftarrow$), scheduler ($\leftrightarrow$)
running $\leftrightarrow$ *blocked*: urgent wait ($\rightarrow$),full preemptive $\Rightarrow$ wait ($\leftarrow$)
blocked $\rightarrow$ *ready*: Alle, sofern wartende zu Signalisierende
running $\leftrightarrow$ *pending*: alle ($\rightarrow$),Signalisierende geben Monitor frei ($\leftarrow$)
pending $\rightarrow$ *blocked*: Alle

```
static semaphore_t bolt = {1};
P(&bolt);
P(&from->lock);
P(&to->lock);
V(&bolt);

doCriticalStuff(from->data,to->data);

V(&to->lock);
V(&from->lock);
```

Listing 30: Deadlockvermeidung Beispiel 1, Großes Lock

```
P(&from->lock)
doStuff(from->data);
V(&from->lock);
P(&to->lock);
doStuff(to->data);
V(&to->lock);
```

Listing 31: Deadlockvermeidung Beispiel 2, Entfernung der Unteilbarkeit

9 Deadlock and Livelock

- Livelock schwerer erkennbar als Deadlock (naja, je nach System-Auslastung)
- wiederverwendbare Ressource $\mapsto$ mehrseitige Synchronisation $\leftrightarrow$ konsumierbare Ressource $\mapsto$ einseitige Synchronisation
- Deadlock Prevention (Vorbeugung) $\mapsto$ statisch, Design $\leftrightarrow$ Deadlock Avoidance (Vermeidung) $\mapsto$ dynamische Reaktionen zur Laufzeit (Lock verweigern, freigeben)
- Bei erkanntem (was schwierig/teuer ist) Deadlock kann dieser gelöst werden durch
  - Töten eines betroffenen Prozesses
  - Rollback eines betroffenen Prozesses

```
1  monitor Account{
2     double inner_data;
3  public:
4     inline void change(double data){
5        doStuff(data);
6     }
7  };
8
9  void transfer(Account& from, Account& to, double data){
10    from.change(-data);
11    to.change(data);
12 }
```

Listing 32: Deadlockvermeidung Beispiel 3, Monitor ⇒ Compiler macht den Rest

10 Guarded Sections

NOTE: Im Folgenden wird für die Synchronisation stets multiple-enqueue/single-dequeue angenommen.

- Ansätze der nichtblockierenden Synchronisation
  - guardian: interrupt-handler dispatcher auf CPU-Priorität - Synchronisation von Prozess-erzeugten Events
    - Edge Triggered ⇒ sti kann die erste Instruktion sein
    - Level Triggered ⇒ sti erfolgt am Ende der Behandlung
    - Läuft in der selben Prozessinstanz wie der Aufrufer (kein separater Kontext)
    - Kann gleichzeitig mehrere Elemente einreihen, aber höchstens eines zeitgleich entnehmen.
  - Pre-/Postlude: Synchronisation von Hardware-erzeugten Events
  - Prelude: First level interrupt handler (FLIH), CPU/OS-Priorität (Prolog)
    - Ergibt nur im Zusammenhang mit Level-Triggered Sinn (startet sofort)
  - Postlude: second level interrupt handler (SLIH), OS-Priorität (Epilog)

Pre-Postlude ist nicht ganz das Selbe wie Pro-Epilog, da Seite 15 (Gleichzeitige Prozesse mit dem Kernelmode werden nicht gesichert). Epilogue-Ausführung ist synchron zum unterbrochenen Kernel-Level-Prozess, bei postlude gilt dies nicht.

Asynchronous System Trap (AST):

Beim Verlassen wird automatisch ein Wiedereintritt ausgelöst, da asynchrone Zustellung. Sind im Grunde Signale, wobei es kein Signal-Code gibt, sondern die Behandlungsfunktion mit der Adresse des AST spezifiziert ist.

Am Ende werden Interrupts aktiviert, sofern man der Letzte ist der kritische Dinge tun muss.

```
1  __attribute__((fastcall)) void guardian(long irq){
2     static usher_t tube = {0, {0, &tube.load.head}};
3     extern remit_t *(*flih[])(usher_t *);
4     remit_t *task;
5  #ifdef __FAME_INTERRUPT_EDGE_TRIGGERED__
6     pivot(&tube.busy, +1); admit(IRQ);      //nehme OS-Priorität
7  #endif
8     task = (*flih[irq])(&tube);             //Prelude-Aktivierung
9
10 #ifdef __FAME_INTERRUPT_LEVEL_TRIGGERED__
11    pivot(&tube.busy, +1); admit(IRQ);      //nehme OS-Priorität
12 #endif
13
14    if(tube.busy > 1){                      //bereits in Verwendung
15       if(task) deter(&tube, task);         //enqueue postlude
16       avert(IRQ);                          //verlasse mit CPU-Priorität
17    }else{
18       if(task && !(tube.load.head.link))
19          remit(task);
20       avert(IRQ);                          //vermeide lost unload
21       while(tube.load.head.link){
22          admit(IRQ);                       //Wiederaufnahme der OS-Priorität
23          flush(&tube);                     //forward pending postludes
24          avert(IRQ);                       //verlasse mit CPU-Priorität
25       }
26    }
27    pivot(&tube.busy, -1);                  //verlasse kritischen Abschnitt
28 }
```

Listing 33: Guardian

Das Increment bei Level-Triggered hat atomar zu erfolgen, was nicht sein muss, da höhere Prioritäten dazwischen funken können. (Mehrere Prioritätsebenen macht bei Level-Triggered durchaus Sinn).

Avert,Admit: Zustandswiederherstellung (Interrupt disable), -Sicherung(Interrupt enable). Das letzte avert ist jeweils wichtig, um Lost-Wakeup zu vermeiden. Es darf nichts eingequeued werden, wenn man gerade dabei ist die Abarbeitung zu verlassen!

```
1 void chart_ms_lfs(queue_t *this, chain_t *item){
2    chain_t *last;
3    item->link = 0;       //FIFO
4    last = this->tail;    //Einfüge-Position
5    this->tail = item;    //Erzeuge neue Teil-Liste
6    while(last->link)     //Wieder einfügen, da Überlappung
7       last = last->link //gehe ans Ende der zwischenzeitlich
           erweiterten Liste
8
9    last->link = item //Einfügen und verschmelzen der beiden
        Listen
10 }
```

Listing 34:

Hier liegt eine Race vor korrigierte Version: listing 45. Wenn zwei (oder mehr) gleichzeitig kommen und hinter dem selben Element eingehangen werden wollen und feststellen hey, cool $last- > link$ ist noch nicht belegt, beide verlassen die Schleife und setzen beide $last- > link = item$, wodurch eines verloren geht

Lock-Free synchronisiertes (lfs) Enqueue
: Unter der Annahme des Enqueue-Vorgangs ausschließlich auf Stackbasierter Ordnung ist dies Thread-Safe. Weiterhin darf maximal einer dequeuen. $this \rightarrow tail$ verweist auf sich selbst, sofern es das selbst das letzte Element ist

```
1 #define FAS(ref,val) __sync_lock_test_and_set(ref, val)
2 #define CAS __sync_bool_compare_and_swap
3 void chart_ms_wfs(queue_t *this, chain_t *item){
4     chain_t *last;
5     item->link = 0;
6     last = FAS(&this->tail, item);
7     last->link = item;
8 }
9
10 chain_t *fetch_ms_wfs(queue_t *this){
11     chain_t *item = this->head.link;
12     if(item){              //überprüfe das letzte Element
13         if(item->link)     //kritisch, kann null sein und
               nachträglich wird ein Element enqueued (tail),
               deswegen gibt es im else noch das CAS
14             this->head.link = item->link;
15         else if (CAS(&this->tail, item, &this->head))
16             CAS(&this->head.link, item, 0);
17     }
18     return item;
19 }
```

Listing 36: Warte freihe Lösung (wfs)

```
1 chain_t *fetch_ms_lfs(queue_t *this){
2     chain *item;
3     if((item = this->head.link)                     //nächstes Elem holen
4           && !(this->head.link = item->link)){      //war das letzte Elem
5         this->tail = &this->head;                   //letztes => reset
6         if(item->link){                 //Überlappung mit enqueue trat auf
7             chain_t *help, *lost = item->link;
8             do{                                     //Nachzügler Abfangen
9                 help = lost->link                   //merke den Nächsten
10                chart_ms_lfs(this,lost);            //rearrange
11            }while((lost=help));
12        }
13    }
14    return item;
15 }
```

Listing 35: Lock-Free synchronisiertes (lfs) Dequeue
Im Unterbrechungsfall werden alle neuen Elemente entnommen und wieder hinten eingehangen. Verhungerung ist dennoch weiterhin möglich.

```
1  void push(lifo_t *list, chain_t *item){
2     acquire(&list->lock);   //enter critical section
3     item->link = list->link;
4     list->link = item;
5     release(&list->lock);   //leave critical section
6  }
7
8  chain_t *pull(lifo_t *list){
9     chain_t *item;
10    acquire(&list->lock);   //enter critical section
11    if((item = list->link))
12       list->link = item->link;
13    release(&list->lock);   //leave critical section
14
15    return item;
16 }
```

Listing 37: Critical Section, Stack

Unter der Annahme ein Stack wird mittels LIFO und einer einfach verketteten Liste dargestellt. Das Push muss nicht sofort, sondern kann theoretisch auch erst im Epilog erfolgen. Beim pull hingegen wird sofort ein Ergebnis erwartet. Daher bietet es sich hierbei an auf Future-Elemente zurückzugreifen.

Pre-/postludes vs. Pro-/Epilogue:

Bei Preludes sind diese Äquivalent, nicht jedoch bei Postludes. Epilogue sind im Gegensatz zu Postludes synchron.

Run to Completion (Process):

Ein etwaig preemptiver Prozess ist frei von selbstständig eingeleiteten Warte-Zuständen.

Run to Stopover (Process):

Ein etwaig preemptiver Prozess, der unter Umständen Warte-Gegebenheiten unterliegt.

Wait Freedom:

Ein Prozess kann jede Operation in einer endlichen Anzahl Schritte erledigen, unabhängig von der Ausführungsgeschwindigkeit der anderen Prozesse.

```
1 typedef struct guard{
2    unsigned book;     //Anzahl konkurierender requests
3    queue_t load;      //anhängende passage requests
4 }guard_t;
5
6 //sequentielle Ausführung der guarded critical section
     garantieren
7 inline order_t *vouch(guard_t *this, order_t *work){
8    enqueu(&this->load, work);         //merke passage request
9    if(!FAA(&this->book, 1))           //überprüfe Belegt-Status,
        Buche passage request
10   //war frei, wurde sequencer, akzeptiere erste passage request
11      return dequeue(&this->load);
12   return 0;
13 }
14
15 //nächste passage request bereinigen, sofern irgendwelche zur
     Abarbeitung anhängend sind.
16 inline order_t *clear(guard_t *this){
17   if(FAA(&this->book, -1) > 1)       //zähle abgeschlossene und ü
        berprüfe weiter anhängende requests
18      return dequeue(&this->load);  //entferne die nächste
           passage request, sofern vorliegend
19   return 0;
20 }
21
22
23 //entry:
24 extern guard_t gate;
25 if(vouch(&gate, parameter)) do
26   //stuff
27   //exit
28   while((task = clear(&gate)));
```

Listing 38: Claiming and Clearing

11 NBS - Non-Blocking Synchronisation

Re-entrant - ablaufinvariant:
: Ein Programm ist re-entrant (ablaufinvariant), wenn dessen Sequenz-Abfolgen zur Ausführungszeit selbst-überlappende Tätigkeit toleriert. Also beliebig Unterbrochen werden kann und auch beliebig häufig parallel ausführbar ist. Damit stellt dies ein elementaren Grundbaustein für Threadsicherheit dar.

Semaphore-Guarded Sections (nicht die ursprüngliche Definition) unterliegen keinem gegenseitigen Ausschluss auf OS-Machine Level, sondern vielmehr werden sie von gegenseitigem Ausschluss auf dem ISA-level im Sinne von atomaren load/store oder read-modify-write Instruktionen geschützt. Diese Semaphore-Operationen sind jetzt nicht mehr unteilbar, sondern nur noch konsistent.

wait-action unfolding:
: Beugt lost-wakeup vor. Ein Prozess wird als “pending blocked” markiert, **bevor** er versucht die Semaphore zu belegen. Diese Markierung wird erst mit der erfolgreichen Erwerbung entfernt. Siehe auch listing 39

Transaction Backup:
: Koordinations-Methode um geteilte Daten zu aktualisieren, durch Verwendung von Transaction Backup-Kontrollmechanismen.
Private Kopie anlegen, darauf Berechnungen durchführen, Validierung und optionales schreiben. Solange der commit fehlschlägt wird dieser gesamte Ablauf wiederholt.

Stop Seite 17

```
1  void prolaag(semaphore_t *sema){
2     catch(&sema->wait);         //Signalisierungs-Interesse
3     while(!claim(sema))         //fordere Semaphore an
4        coast();                 //akzeptiere Weck-Signal
5     clean(&sema->wait);         //lösche Signalisierungs-Interesse
6  }
7
8
9  void verhoog(semaphore_t *sema){
10    if(unban(sema))
11       cause(&sema->wait);      //signalisiere Weck-Signal
12 }
13
14 inline int coast(){
15    stand();                        //sperre Event
16    return being(ONESELF)->merit; //signaler PID
17 }
18
19 int cause(event_t *this){
20    process_t *next;
21    int done = 0;
22    for(next = being(0); next < being(NPROC); next++)
23       if(CAS(&next->event, this, 0))
24          done += hoist(next, being(ONESELF)->name);
             //Signalisiere ``go ahead'', übergebe eigene ID
             und bereite Signal-Empfänger vor.
25
26    return done;
27 }
```

Listing 39: Wait-Action Unfolding Semaphore

```
1 inline void catch(event_t *this){
2    process_t *self = being(ONESELF);
3    self->state |= PENDING;     //Erwarte Ereignisse
4    apply(self, this);          //betrete ‘‘go ahead’’-Warteliste
5 }
6
7 inline void clean(event_t *this){
8    elide(being(ONESELF), this);  //verlasse Warteliste
9 }
```

Listing 40: Dualism

Forcierung von Prozessen, die sich im laufenden Zustand befinden und sich zeitgleich auf der Ready-Liste befinden. Intern ist dies für den Idle-Prozess auch ganz normal, um leere Listen zu vermeiden.

```
1 typedef struct stack{
2    chain_t head;          //top of stack: list head
3 }stack_t;
4
5 typedef struct chain{
6    struct chain *link;    //next list element
7 }chain_t;
8
9 inline void push_dos(stack_t *this, chain_t *item){
10    do //kopiere den Stack Pointer-Inhalt in das einzufügende Element
11       item->link = this->head.link;
12    //Aktualisiere den Stack Pointer
13    while(!CAS(&this->head.link, item->link, item));
14 }
15
16 inline chain_t *pull_dos(stack_t *this){
17    chain_t *node;
18    do
19       if(!(node = this->head.link)) //merke das Item, das auf dem TOS liegt
20          break;
21    while(!CAS(&this->head.link, node, node->link));
22
23    return node;                  //Aktualisiere den Stack-Pointer
24 }
```

Listing 41: Lock Frei synchronisierter Stack

Annahme jedes Element kann lediglich einmalig eingefügt werden. Dadurch ist die Annahme legitim, dass Elemente nicht verändert werden und kann entsprechend als ”lokale Variable”betrachtet und verwendet werden kann.

```
inline void *raw(void *item, long mask){
    return (void*)((long)item & ~mask);
}
inline void *tag(void *item, long mask){
    return (void*)(((long)item +1) & mask);
}
```

Listing 42: Generationszähler anhand des Alignments
Gelegentlich kommt einem das Alignment zu gute und man kann ungenutzten Speicher für Generationszähler verwenden. Hierbei ist man auf die Hilfe des Compilers angewiesen.

```
typedef chain_t *chain_l;         //labelled pointer
#define BOX(sizeof(chain_t) -1)   //Maske

inline void push_lfs(stack_t *this, chain_l item){
    do
        ((chain_t*)raw(item, BOX))->link = this->head.link;
    while(!CAS(&this->head.link,
            ((chain_t*)raw(item, BOX))->link, tag(item, BOX)));
}

chain_l pull_lfs(stack_t *this){
    chain_l node;
    do
        if(!raw((node = this->head.link), BOX))
            break;
    while(!CAS(&this->head.link, node,
            ((chain_t*)raw(node, BOX))->link));

    return node;
}
```

Listing 43: CAS-stack mit Generationszähler anhand des Alignments, ABA gelöst (Problem besteht bei LL/SC erst gar nicht)

```
1 typedef struct queue{
2    chain_t head;
3    chain_t *tail; //Einfüge-Punkt, nicht zwingend letztes Element
4 }queue_t;
5
6
7 //kann zum Zurücksetzen der Queue verwendet werden, gibt alle
     Listen-Elemente zurück
8 inline chain_t *drain(queue_t *this){
9    chain_t *head = this->head.link;
10   this->head.link = 0;
11   this->tail = &this->head;  //verknüpfendes Element
12
13   return head;
14 }
15
16 inline void chart_dos(queue_t *this, chain_t *item){
17   item->link = 0;
18   this->tail->link = item;   //apend item
19   this->tail = item;         //setze Einfüge-Punkt
20 }
21
22 inline chain_t *fetch_dos(queue_t *this){
23   chain_t *node;
24   //falls befüllt und letztes Element
25   if((node = this->head.link) && !(this->head.link = node->link))
26      this->tail = &this->head;  //reset
27
28   return node;
29 }
```

Listing 44: unsynchronisierte Queue

Synchronisation fast leerer Queues mithilfe von Flags, um das Verlieren von Elementen zu vermeiden. Dadurch kann enqueue und dequeue miteinander kommunizieren. s. listing 45

```
1 void chart_lfs(queue_t *this, chain_t *item){
2    chain_t *last;
3    item->link = 0;
4    do
5       last = this->tail;   //tail Pointer als kritisch geteiltes Datum
6    //validieren und (schreiben) des Tail-Pointers
7    while(!CAS(&this->tail, last, item));
8    last->link = item;      //Einfügen
9 }
10
11 chain_t *fetch_lfs(queue_t *this){
12    chain_t *node;
13    do    //head Pointer als kritisch geteiltes Datum
14       if(!(node = this->head.link))
15          return 0;
16    //validieren und (schreiben) des Head-Pointers
17    while(!CAS(&this->head.link, node, node->link));
18
19    //Jedes entnommene Item ist einzigartig. Fall des letzten Elements
20    if(!node->link)
21       //leere Queue, Validität des Tail-Pointers wiederherstellen
22       this->tail = &this->head;
23
24    return node;
25 }
```

Listing 45: Synchronisation linked List, Take one chart ||chart and fetch ||fetch. Nicht jedoch chart||fetch oder fetch||chart, da ein enqueue verloren ginge, wenn head und tail das selbe Elemente referenzieren

Zweifach-Feld-Aktualisierung:
Sind unter Umständen selbst mit DCAS, CDS sehr schwierig.

Warteliste Implementierungselemente listings 47 to 51

↑ Code verifizieren

```
1  void chart_lfs(queue_t *this, chain_t *item){
2     chain_t *last, *hook;
3     item->link = item;   //self-reference
4     do
5        hook = (last = this->tail)->link;   //tail end
6     while(!CAS(&this->tail, last, item));
7     //Ende?, gelingt nur, wenn letztes Element immernoch
         self-reference => letztes Element wurde nicht entnommen
8     if(!CAS(&last->link, hook, item))
9        //letztes Element nicht mehr self-reference => letztes
            Element entnommen. Weitere enqueues verwenden dieses
            Element als führendes link-Element
10       this->head.link = item;
11 }
12
13 chain_t *fetch_lfs(queue_t *this){
14    chain_t *node, *next;
15    do
16       if(!(node = this->head.link))
17          return 0;
18    while(!CAS(&this->head.link, node,
19          ((next=node->link) == node ? 0 : next)));
20
21    if(next == node){ //selbst-reference, ist last
22       //versuche zu helfen, schlägt fehl, wenn keine
            selbst-Referenz mehr enthalten
23       if(!CAS(&node->link, next, 0))
24          this->head.link = node->link; //enqueue
25       else
26          //enqueue wurde assistiert und entnommener Knoten
               könnte der letzte sein
27          CAS(&this->tail, node, &this->head);
28    }
29    return node;
30 }
```

Listing 46: Lock-Free Synchronisierte verkettete Liste, take four

last verweist auf das letzte Element, $link_{last}$ dessen next-Element

$$link_{last} = \begin{cases} last, & \text{chain link gültig, wurde nicht gelöscht} \\ 0, & \text{chain link ungültig, wurde gelöscht} \\ else, & \text{chain link verweist auf Vorgänger} \end{cases}$$

Hier helfen sich chart und fetch gegenseitig.

Code Verifizieren, wirkt absolut plausibel, aber Korrektheit habe ich nicht anhand von Beispielen durchgespielt.

```
1 inline void shade(process_t *this){
2    this->latch.flag = false;  //clear latch
3 }
4 inline void stand(){
5    process_t *self = being(ONESELF);
6    if(!self->latch.flag)   //inactive latch
7       //aufgeben, entweder suspend oder den aktuelle Prozess
           fortsetzen. (War als pending markiert, um ‘‘go
           ahead’’-Signalisierung zu empfangen)
8       block();
9    shade(self);            //reset latch
10 }
11 inline void latch(){
12    being(ONESELF)->state |= PENDING;    //warte
13    stand();                             //latch
14 }
```

Listing 47: Warteliste - receive sticky bit

```
1 inline void punch(process_t *this){
2    //inactive latch (PCB wird nicht mit simultanen Prozessen
        geteilt, sonst müsste hier ein TAS verwendet werden)
3    if(!this->latch.flag){
4       this->latch.flag = true;          //activate it
5       if(this->state & PENDING)         //ist verriegelt
6          yield(this);                   //set ready
7    }
8 }
9
10 inline int hoist(process_t *next, int code){
11    next->merit = code;                 //reiche Ergebnis weiter
12    punch(next);                        //sende Signal
13    return 1;
14 }
```

Listing 48: Waitlist - send sticky bit

```
void block(){
    process_t *next, *self = being(ONESELF);
    do{   //werde zum idle-Prozess
        while(!(next = elect(hoard(READY))))
            relax();    //schlafen
    }while((next->state & PENDING) &&
            (next->scope != self->scope));   //clean up?

    if(next != self){    //wurde ich ready gesetzt?
        self->state = (BLOCKED | (self->state & PENDING));
        seize(next);      //weiterhin anhängend bleiben
    }
    self->state = RUNNING;
}
```

Listing 49: Waitlist - Multiple Persönlichkeiten auflösen

```
inline void apply(process_t *this, event_t *list){
#ifdef __FAME_EVENT_WAITLIST__
    insert(&list->wait, &this->event);  //lock-free, listen-Durchlauf
#else
    this->event = list;                 //atomarer table walk
#endif

inline void elide(process_t *this, event_t *list){
#ifdef __FAME_EVENT_WAITLIST__
    winnow(&list->wait, &this->event);  //lock-free, listen-Durchlauf
#else
    this->event = 0;                    //atomarer table walk
#endif
}
```

Listing 50: Waitlist - Waitlist Association

Stop s. 37

```
1 int cause(event_t *this){
2    chain_t *item;
3    int done = 0;
4    //unteilbar und Warte-freie Annahme der Warteliste
5    if((item = detach(&this->wait)))
6       do
7          //‘‘go ahead’’, setze signalisierten bereit
8          done += hoist((process_t*)coerce(item,
               (int)&((process_t*)0)->event),
               being(ONESELF)->name);
9       while((item = item->link));   //signalisiere stets einen
           Prozess auf einmal, maximal N-1 notwendig
10   return done;
11 }
```

Listing 51: Waitlist - Signalisierung verbreiten

12 Transactional Memory - TM

Je größer der geteilte Status ist, desto besser scheint TM zu sein. Ist im Grunde eine Abstraktion von mehrseitiger Synchronisation.

Pros:

- Erlaubt Definition vieler Befehle in einer atomaren Operation.
- Kann als Caching-Methode für die Implementierung von Daten-Strukturen unter blockierfreien Gesichtspunkten angesehen werden
- Technisch als Erweiterung von Cache-Kohärenz-Protkollen einfach realisierbar.
- Einfache Handhabung

Cons:

- Ist unter Umständen lediglich ein Ersatz für mehrseitige Synchronisation
- Behindert uniliterale Synchronisation
- Sehr Overhead-Anfällig, sofern massiv auf externe Funktionen oder Bibliotheken zugegriffen wird (STM)

Hardware Transactional Memory (HTM):

Ursprünglich gedacht um zu einem transactionalen Cache eine Cache Duplikation zu erzeugen. Daten verschiedener Transaktionen sollten in verschiedenen Cache-Zeilen untergebracht werden, da dort die Konflikt-Detektion stattfindet. Explicitly vs. implicitly transactional: Explizite Transaktions load/store vs. begin-end-Grenzen einer Transaktion, wobei die implizite Variante für mehr Overhead sorgt, da der Prozessor sämtliche Zugriffe auf Variablen innerhalb des kritischen Bereichs verfolgen muss. Entsprechend werden vom Compiler zwei Code-Pfade erzeugt.

Ist in aktuellen Prozessor-Architekturen kaum verfügbar.

Schlägt eine HTM-Transaktion fehl, so entscheidet ein Run-Time-System wie der Vorgang wiederholt werden soll:

- HTM, wenn der Wettstreit schwach ist
- STM, sonst, möglicherweise mit flexiblerer Wettstreit-Kontrolle

Software Transactional Memory (STM):
Signifikanter Metadaten-Overhead und Begrenzung durch virtuelle Speichergröße. Ist ein geteilter Speicherbereich bestimmter Größe. Transaktionen werden durch einen Generationszähler einzigartig.

1. Besitz jeder Speicherstelle von Data-Set-Membern anfordern (LL)
2. Wenn erfolgreich
   a) Lesen des Data-Set-Members Speichers
   b) Berechne neue Werte
   c) aktualisiere Data-Set-Speicher Werte
   d) Gebe Besitz ab
3. Nicht erfolgreich
   a) Gebe jeglichen Besitz frei, sofern vorhanden
   b) Hilf der Transaktion, der die fehlgeschlagenen Bereiche gehören.

Hybrid Transactional Memory (HyTM):
Hybrid. Versuche erst HTM und verwende STM in kritischen Situationen als Notfallplan.

13 Progress Guarantees

Hier ist noch was dazu gekommen?

Eigenschaften von Non-Blocking Synchronisationsalgorithmen:

- obstruction-free
  Beendigung jeder Operation in einer endlichen Anzahl Schritte, sofern Isolation vorliegt. Ist anfällig für Verhungerung bei Einflussnahme anderer Prozesse.
- lock-free
  Einige Prozesse sind in der Lage eine Operation in einer endlichen Anzahl Schritte zu beenden.
  Verhungerungsfreiheit von mindestens einem Prozess und damit auch des Systems.
- wait-free
  Jeder Prozess kann jede Operationen in endlicher Schrittanzahl beenden.
  Jeder Prozess ist verhungerungsfrei

Es ist auch dann noch non-blocking, wenn ein Prozess dem Scheduler signalisiert, dass er selbst gerade nichts sinnvolles zu erledigen hat und die Kontrolle abgibt, ohne die Laufbereitschaft zu verlieren.

CAS bestenfalls lock free
LL/SC bestenfalls obstruction free, weil zusätzlich Reservierungen gestrichen werden, wenn Interrupts auftreten. Diese Interrupts stehen im Grunde nicht im Konflikt zu dem kritischen Bereich.
Table-Based Scheduling I ist wartefrei, da obere Grenze

Listings

1 TAS unconditional store . . . 8
2 TAS conditional store mit DPRAM . . . 9
3 CAS conditional store . . . 9
4 TAS und CAS mithilfe von LL und SC . . . 10
5 LL Implementierung . . . 18
6 SC Implementierung . . . 18
7 FAA Implementierung . . . 18
8 AAF Implementierung . . . 18
9 fax - conditional store . . . 18
10 xaf - conditional store . . . 18
11 Dekker's Algorithmus für $N = 2$ . . . 21
12 Peterson's Algorithmus für $N = 2$ . . . 22
13 Kessel's Algorithmus für $N = 2$ . . . 22
14 Spin-Lock . . . 23
15 Spin-Lock mit TAS . . . 23
16 Spin-Lock mit CAS . . . 23
17 Spin on Read . . . 24
18 Spin-Lock with Backoff . . . 25
19 Spin with Backoff and spin on read . . . 25
20 Ticket Lock . . . 26
21 Peterson's Algorithmus für $N > 2$ . . . 27
22 Lamport's Bakery Algorithmus, verwendet Tickets . . . 28
23 Monitor-Datentypen . . . 31
24 Monitor-Verwendung, Bounded Buffer, Typen: listing 23 . . . 32
25 Warteliste . . . 34
26 Signal and Continue . . . 35
27 Signal and Return . . . 35
28 Signal and Wait . . . 35
29 Signal and Urgent Wait . . . 36
30 Deadlockvermeidung Beispiel 1, Großes Lock . . . 37
31 Deadlockvermeidung Beispiel 2, Entfernung der Unteilbarkeit . . . 37
32 Deadlockvermeidung Beispiel 3, Monitor ⇒ Compiler macht den Rest . . . 38
33 Guardian . . . 40
34 Lock-Free synchronisiertes Enqueue (lfs) . . . 41
36 Wartefreie Lösung (wfs) . . . 42
35 Lock-Free synchronisiertes Dequeue (lfs) . . . 42
37 Critical Section, Stack . . . 43
38 Claiming and Clearing . . . 44
39 Wait-Action Unfolding Semaphore . . . 46
40 Dualism . . . 47
41 Lock Frei synchronisierter Stack . . . 47

42 Generationszähler anhand des Alignments 48
43 CAS-stack mit Generationszähler anhand des Alignments, ABA gelöst . . 48
44 unsynchronisierte Queue 49
45 Synrchonisation linked List, Take one chart ||chart and fetch ||fetch 50
46 Lock-Free Synchronisierte verkettete Liste, take four 51
47 Warteliste - receive sticky bit 52
48 Waitlist - send sticky bit 52
49 Waitlist - Multiple Persönlichkeiten auflösen 53
50 Waitlist - Waitlist Association 53
51 Waitlist - Signalisierung verbreiten 54

Liste der noch zu erledigenden Punkte

- Note: Viele Code-Beispiele sind so gestaltet, dass zu Gunsten der Lesbarkeit an einigen Stellen Fences und ABA-Behandlungen weggelassen wurden. . . . 3
- Die bitweise Verundung könnte man sicherlich noch zum return runterziehen und somit den atomaren Block verkleinern 8
- So ganz scheint das aber nicht zu stimmen, es geht wohl mehr darum sich auf einen Wert zu einigen, da die Beispiele nicht ganz mit der Erklärung matchen. TAS und FAA sollten ja für beliebig viele parallele Prozesse funktionieren. . . 11
- verifiziere Korrektheit dieses Peterson-Codes 24
- verifiziere Korrektheit dieses Lamport Bakery-Codes 24
- NOTE: Im Folgenden wird für die Synchronisation stets multiple-enqueue/single-dequeue angenommen. 39
- Hier liegt eine Race vor korrigierte Version: listing 45. Wenn zwei (oder mehr) gleichzeitig kommen und hinter dem selben Element eingehangen werden wollen und feststellen hey, cool $last->link$ ist noch nicht belegt, beide verlassen die Schleife und setzen beide $last->link = item$, wodurch eines verloren geht 41
- Stop Seite 17 45
- ↑ Code verifizieren 50
- Code Verifizieren, wirkt absolut plausibel, aber Korrektheit habe ich nicht anhand von Beispielen durchgespielt. 51
- Stop s. 37 53
- Hier ist noch was dazu gekommen? 57