

Concurrent Systems

Nebenläufige Systeme

X. Non-Blocking Synchronisation

Wolfgang Schröder-Preikschat

January 22, 2020



Agenda

Preface

Constructional Axis

General

Exemplification

Transition

Transactional Axis

General

Onefold Update

Twofold Update

Summary



Preface

Constructional Axis

General

Exemplification

Transition

Transactional Axis

General

Onefold Update

Twofold Update

Summary



- discussion on abstract concepts of synchronisation without lockout of critical action sequences of interacting processes (cf. [7])
 - attribute “non-blocking” here means **abdication of mutual exclusion** as the conventional approach to protect critical sections
 - note that even a “lock-free” solution may “block” a process from making progress, very well!
- develop an intuition for the dependency on **process interleaving** and **contention rate** when arguing on performance issues
 - what in case of high and what else in case of low contention?
 - what is the exception that proves the rule?
- follow suit, an explanation of the **two-dimensional** characteristic of non-blocking synchronisation is given
 - on the one hand, constructional, on the other hand, transactional
 - with different weighting, depending on the use case and problem size
- not least, engage in sort of *tolerance to races* of interacting processes while preventing faults caused by race conditions. . .



*Tolerance is the suspicion
that the other person just might be right.¹*



Source: Commemorative plaque, Berlin, Bundesallee 79

¹(Ger.) *Toleranz ist der Verdacht, dass der andere Recht hat.*

Preface

Constructional Axis

General

Exemplification

Transition

Transactional Axis

General

Onefold Update

Twofold Update

Summary



Definition

A program is **re-entrant** (Ger. *ablaufinvariant*) if, at execution time, its sequence of actions tolerates self-overlapping operation.

- those programs can be re-entered at any time by a new process, and they can also be executed by simultaneous processes
 - the latter is a logical consequence of the former: **full re-entrant**
 - but the former does not automatically imply the latter²
- originally, this property was typical for an **interrupt handler**, merely, that allows for nested execution—recursion not unressembling
 - each interrupt-driven invocation goes along with a new process
 - whereby the simultaneous processes develop **vertically** (i.e., stacked)
- generally, this property is typical for a large class of **non-sequential programs** whose executions may overlap each other
 - each invocation goes along with a new process, it must be “thread-safe”
 - whereby the simultaneous processes develop **horizontally**, in addition

²For example, if lockout becomes necessary to protect a critical section.



- devoid of an explicit protective shield all-embracing the semaphore implementation, i.e., the elementary operations P and V :

```
1 typedef struct semaphore {  
2     int gate;                /* value: binary or general */  
3     event_t wait;           /* list of sleeping processes */  
4 } semaphore_t;
```

- other than the original definition [1, p. 29], semaphore primitives are considered **divisible operations** in the following
 - merely single steps that are to be performed inside of these primitives are considered indivisible
 - these are operations changing the semaphore value (*gate*) and, as the case may be, the waitlist (*wait*)
 - but not any of these operations are secured by means of mutual exclusion at operating-system machine level
 - rather, they are safeguarded by falling back on ISA-level mutual exclusion in terms of atomic load/store or read-modify-write instructions



Building Blocks for Barrier-Free Operation

- use of **atomic** (ISA-level) **machine instructions** for changing the semaphore value consistently (p. 11)
 - a TAS or CAS, resp., for a binary and a FAA for a general semaphore
 - instruction cycle time is bounded above, solely hardware-defined
 - wait-free [3, p. 124], irrespective of the number of simultaneous processes
- abolish abstraction in places, i.e., perform **wait-action unfolding** to prevent the lost-wakeup problem (p. 10)
 - make a process “pending blocked” before trying to acquire the semaphore
 - cancel that “state of uncertainty” after semaphore acquirement succeeded
 - wait- or lock-free [3, p. 142], depending on the waitlist interpretation
- accept **dualism** as to the incidence of processing states, i.e., tolerate a “running” process being seemingly “ready to run” (p. 12)
 - delay resolving until some process is in its individual idle state
 - have also other processes in charge of clearing up multiple personality
 - wait-free, resolution produces background noise but is bounded above
- forgo dynamic data structures for any type of waitlist or synchronise them using **optimistic concurrency control** (p. 16ff.)



```
1 void prolaag(semaphore_t *sema) {
2     catch(&sema->wait);          /* expect notification */
3     lodge(sema);                  /* raise claim to proceed */
4     when (!avail(sema))           /* check for process delay */
5         coast();                  /* accept wakeup signal */
6     clean(&sema->wait);           /* forget notification */
7 }
8
9 void verhoog(semaphore_t *sema) {
10     if (unban(sema))              /* release semaphore */
11         cause(&sema->wait);      /* notify wakeup signal */
12 }
```

■ implementation in the shape of a **non-sequential program**:

- 2 ■ show interest in the receive of a notification to continue processing
- 3/4 ■ draw on walkover, bethink and, if applicable, watch for notification
- 5 ■ either suspend or continue execution, depending on notification state
- 6 ■ drop interest in receiving notifications, occupy resource
- 10 ■ deregulate “wait-and-see” position above (l. 4), check for a sleeper
- 11 ■ send notification to interested and, maybe, suspended processes



- load/store-based implementation for a **binary semaphore**:

```
1 inline bool avail(semaphore_t *sema) {  
2     return CAS(&sema->gate, 1, 0);  
3 }
```

- both *lodge* and *unban* remain unchanged

- enumerator-based implementation for a **general semaphore**:

```
1 inline int lodge(semaphore_t *sema) {  
2     return FAA(&sema->gate, -1);  
3 }  
4  
5 inline bool unban(semaphore_t *sema) {  
6     return FAA(&sema->gate, +1) < 0;  
7 }
```

- *avail* remains unchanged

- note that both variants are insensitive to simultaneous processes
 - due to **indivisible operations** for manipulation of the semaphore value



- a process being in “running” state and, as the case may be, at the same time recorded on the waitlist of “ready to run” peers

```
1 inline void catch(event_t *this) {
2     process_t *self = being(ONESELF);
3     self->state |= PENDING;           /* watch for event */
4     apply(self, this);                /* enter waitlist */
5 }
6
7 inline void clean(event_t *this) {
8     elide(being(ONESELF), this);      /* leave waitlist */
9 }
```

- 3 ■ prepares the “multiple personality” process to be treated in time
- 4 ■ makes the process amenable to “go ahead” notification (p. 10, l. 11)
- 8 ■ excludes the process from potential receive of “go ahead” notifications
- treatment of “multiple personality” processes is based on **division of labour** as to the different types of waitlist (cf. p. 42)
 - “ready” waitlist, the respective idle process of a processor (p. 41)
 - “blocked” waitlist, the semaphore increasing or decreasing process



- catch of a “go ahead” event is by means of a **per-process latch**
 - i.e., a “sticky bit” holding member of the *process control block* (PCB)

```
1 inline int coast() {
2     stand();                                /* latch event */
3     return being(ONESELF)->merit;          /* signaller pid */
4 }
5
6 int cause(event_t *this) {
7     process_t *next;
8     int done = 0;
9
10    for (next = being(0); next < being(NPROC); next++)
11        if (CAS(&next->event, this, 0))
12            done += hoist(next, being(ONESELF)->name);
13
14    return done;
15 }
```

11 ■ recognise willingness to catch a signal and continue execution

12 ■ notify “go ahead”, pass own identification, and ready signallee



A Means to an End...

- non-blocking synchronisation spans **two dimensions** of measures in the organisation of a non-sequential program
 - i a constructional axis, as was shown with the semaphore example, and
 - ii a transactional axis, which is coming up in the next section
- in many cases, particularly given complex software structures such as operating systems, the former facilitates the latter
 - the building blocks addressed and drafted so far are not just dedicated to operating systems, but are suited for any kind of “threads package”
 - although quite simple, they still disclose handicaps as to **legacy software**
- reservation towards the exploitation of non-blocking synchronisation originates much more from the **constructional axis**
 - synchronisation is a typical **cross-cutting concern** of software and, thus, use case of *aspect-oriented programming* (AOP, [5])
 - but the semaphore example shows that even AOP is not the loophole here
- but note that the **transactional axis** does not suggest effortlessness and deliver a quick fix to the synchronisation problem
 - appropriate solutions, however, benefit from a much more localised view



Outline

Preface

Constructional Axis

General

Exemplification

Transition

Transactional Axis

General

Onefold Update

Twofold Update

Summary



Definition (acc. [6])

Method of coordination for the purpose of updating shared data by mainly relying on **transaction backup** as control mechanisms.

```
1 do
2     read phase:
3         save a private copy of the shared data to be updated;
4         compute a new private data value based on that copy;
5     validation and, possibly, write phase:
6         try to commit the computed value as new shared data;
7     while commit failed (i.e., transaction has not completed).
```

- during the **read phase**, all writes take place only on *local copies* of the shared data subject to modification
- a subsequent **validation phase** checks that the changes as to those local copies will not cause loss of integrity of the shared data
- if approved, the final **write phase** makes the local copies global, i.e., commits their values to the shared data



- CAS-oriented approach, value-based, typical for CISC:

```
1 word_t any;                                /* shared data */
2 {
3     word_t old, new;                        /* own data */
4     do new = compute(old = any);           /* read */
5     while (!CAS(&any, old, new));          /* validate/write */
6 }
```

- LL/SC-oriented approach, reservation-based, typical for RISC:

```
1 word_t any;                                /* shared data */
2 {
3     word_t new;                             /* own data */
4     do new = compute(LL(&any));            /* read */
5     while (!SC(&any, new));                /* validate/write */
6 }
```



Data Type I

- let a very simple **dynamic data structure** be object of investigation
 - modelling a **stack** in terms of a single-linked list:

```
1 typedef struct stack {  
2     chain_t head;      /* top of stack: list head */  
3 } stack_t;
```

- whereby a single **list element** is of the following structure:

```
1 typedef struct chain {  
2     struct chain *link; /* next list element */  
3 } chain_t;
```

- stack manipulation by pushing or pulling an item involves the update of a single variable, only: the “stack pointer”
- when simultaneous processes are allowed to interact by sharing that stack structure, the update must be an indivisible operation



- basic **precondition**: an item to be stacked is not yet stacked/queued

```
1 inline void push_dos(stack_t *this, chain_t *item) {  
2     item->link = this->head.link;  
3     this->head.link = item;  
4 }
```

- 2 ■ copy the contents of the stack pointer to the item to be stacked

- 3 ■ update the stack pointer with the address of that item

```
5 inline chain_t *pull_dos(stack_t *this) {  
6     chain_t *node;  
7     if ((node = this->head.link))  
8         this->head.link = node->link;  
9     return node;  
10 }
```

- 7 ■ memorise the item located at the stack top, if any

- 8 ■ update the stack pointer with the address of the next item



- benefit from the precondition: an item to be stacked is “own data”

```
1 inline void push_lfs(stack_t *this, chain_t *item) {  
2     do item->link = this->head.link;  
3     while (!CAS(&this->head.link, item->link, item));  
4 }
```

- 2 ■ copy the contents of the stack pointer to the item to be stacked
- 3 ■ attempt to update the stack pointer with the address of that item

```
5 inline chain_t *pull_lfs(stack_t *this) {  
6     chain_t *node;  
7  
8     do if ((node = this->head.link) == 0) break;  
9     while (!CAS(&this->head.link, node, node->link));  
10  
11     return node;  
12 }
```

- 8 ■ memorise the item located at the stack top, if any
- 9 ■ attempt to update the stack pointer with the address of the next item



- workaround using a **change-number tag** as pointer label:

```
1 inline void *raw(void *item, long mask) {  
2     return (void *)((long)item & ~mask);  
3 }  
4  
5 inline void *tag(void *item, long mask) {  
6     return (void *)  
7         ((long)raw(item, mask) | ((long)item + 1) & mask);  
8 }
```

- **alignment** of the data structure referenced by the pointer is assumed
 - an **integer factor** in accord with the data-structure size (in bytes)
 - rounded up to the next **power of two**: $2^N \geq \text{sizeof}(\text{datastructure})$
- zeros the N low-order bits of the pointer—and discloses the **tag field**
- rather a **kludge** (Ger. *Behelfslösung*) than a clearcut solution³
 - makes ambiguities merely unlikely, but cannot prevent them
 - “operation frequency” must be in line with the **finite values margin**
- if applicable, attempt striving for problem-specific **frequency control**

³This also holds for DCAS when using a “whole word” change-number tag.



```
1 typedef chain_t* chain_l;          /* labelled pointer! */
2
3 #define BOX (sizeof(chain_t) - 1)  /* tag-field mask */
4
5 inline void push_lfs(stack_t *this, chain_l item) {
6     do ((chain_t *)raw(item, BOX))->link = this->head.link;
7     while (!CAS(&this->head.link, ((chain_t *)raw(item, BOX))->link, tag(item, BOX)));
8 }
9
10 chain_l pull_lfs(stack_t *this) {
11     chain_l node;
12
13     do if (raw((node = this->head.link), BOX) == 0) break;
14     while (!CAS(&this->head.link, node, ((chain_t *)raw(node, BOX))->link));
15
16     return node;
17 }
```

- aggravating side-effect of the solution is the **loss of transparency**
 - the pointer in question originates from the environment of the critical operation (i.e., *push* and *pull* in the example here)
 - tampered pointers must not be used as normal $\leadsto$ *derived type*
- language embedding and compiler support would be of great help...

Hint (CAS vs. LL/SC)

The ABA problem does not exist with LL/SC!



- same precondition (cf. p.20): an item to be stacked is “own data”

```
1 inline void push_lfs(stack_t *this, chain_t *item) {  
2     do item->link = LL(&this->head.link);  
3     while (!SC(&this->head.link, item));  
4 }
```

- 2 ■ copy the head pointer and make a reservation to his address

- 3 ■ update the head pointer if the reservation still exists

```
5 inline chain_t *pull_lfs(stack_t *this) {  
6     chain_t *node;  
7  
8     do if ((node = LL(&this->head.link)) == 0) break;  
9     while (!SC(&this->head.link, node->link));  
10  
11     return node;  
12 }
```

- 8 ■ memorise the head pointer and make a reservation to his address

- 9 ■ update the head pointer if the reservation still exists



Data Type II

- a much more complex object of investigation, at a second glance:

```
1 typedef struct queue {  
2     chain_t  head;           /* first item */  
3     chain_t *tail;          /* insertion point */  
4 } queue_t;
```

- the tail pointer addresses the linkage element of a next item to be queued
- it does not directly address the last element in the queue, but indirectly

- consequence is that even an empty queue shows a valid tail pointer:

```
1 inline chain_t *deplete(queue_t *this) {  
2     chain_t *head = this->head.link;  
3  
4     this->head.link = 0;      /* null item */  
5     this->tail = &this->head; /* linkage item */  
6  
7     return head;  
8 }
```

- used to reset a queue and at the same time return all its list members



- same **precondition** as before: an item to be queued is not yet queued
 - a simple **first-in, first-out method** (FIFO) is implemented

```
1 inline void enqueue_dos(queue_t *this, chain_t *item) {
2     item->link = 0;                /* finalise chain */
3     this->tail->link = item;        /* append item */
4     this->tail = item;              /* set insertion point */
5 }
```

- note that the queue head pointer gets set to the first item implicitly

```
6 inline chain_t* dequeue_dos(queue_t *this) {
7     chain_t *node;
8     if ((node = this->head.link)    /* filled? */
9     && !(this->head.link = node->link)) /* last item? */
10         this->tail = &this->head;    /* reset */
11     return node;
12 }
```

- 11 ■ the tail pointer must always be valid, even in case of an empty queue



- inspired by the lock-free solution using atomic load/store [13, p. 28]:

```
1 void enqueue_lfs(queue_t *this, chain_t *item) {  
2     chain_t *last;  
3  
4     item->link = 0;  
5  
6     do last = this->tail;  
7     while (!CAS(&this->tail, last, item));  
8  
9     last->link = item;  
10 }
```

Hint (Onefold Update)

Only a single shared variable needs to be updated in this scenario.

- a **plausibility check** shows correctness as to this overlap pattern:
 - 6 ■ critical shared data is the tail pointer, a local copy is read
 - each overlapping enqueue holds its own copy of the tail pointer
 - 7 ■ validate and, if applicable, write to update the tail pointer
 - the item becomes new fastener for subsequent enqueue operations
 - 9 ■ eventually, the item gets inserted and becomes queue member
 - the assignment operator works on local operands, only



- inspired by the lock-free solution for a stack pull operation (p. 20):

```
1 chain_t* dequeue_lfs(queue_t *this) {
2     chain_t *node;
3
4     do if ((node = this->head.link) == 0) return 0;
5     while (!CAS(&this->head.link, node, node->link));
6
7     if (node->link == 0)
8         this->tail = &this->head;
9
10    return node;
11 }
```

Hint (Onefold Update)

Only a single shared variable needs to be updated in this scenario.

- a **plausibility check** shows correctness as to this overlap pattern:
 - critical shared data is the head pointer, a local copy is read
 - each overlapping dequeue holds its own copy of the head element
 - validate and, if applicable, write to update the head pointer
 - each dequeued item is unique, only of them was last in the queue
 - the tail pointer must always be valid, even in case of an empty queue



- critical is when head *and* tail pointer refer to the same “hot spot” and enqueue and dequeue happen simultaneously
- assuming that the **shared queue** consists of only a single element:
 - enq || deq* ■ enqueue memorised the chain link of that element
 - dequeue removed that element including the chain link
 - enqueue links the new element using an invalid chain link
 - ↪ **lost enqueue**: linking depends on dequeue progression
 - deq || enq* ■ dequeue removed that element and notices “vacancy”
 - enqueue appends an element to the one just removed
 - dequeue assumes “vacancy” and resets the tail pointer
 - ↪ **lost enqueue**: resetting depends on enqueue progression
- enqueue and dequeue must assist each other to solve the problem:
 - i identify the conditions under which lost-enqueue may happen
 - ii identify a way of interaction between enqueue and dequeue
- assist without special auxiliary nodes but preferably with simultaneous consideration of **conservative data-structure handling**



- idea is to use the chain-link of a queue element as **auxiliary means** for the interaction between enqueue and dequeue [9]
 - let $last$ be the pointer to the chain link of the queue end tail and
 - let $link_{last}$ be the chain link pointed to by $last$, then:

$$link_{last} = \begin{cases} last, & \text{chain link is valid, was not deleted} \\ 0, & \text{chain link is invalid, was deleted} \\ \text{else,} & \text{chain link points to successor element} \end{cases}$$

- $link_{last}$ set to 0 models the per-element “deleted bit” as proposed in [2]
 - for a FIFO queue, only the end-tail element needs to carry that “bit”
- in contrast to [2], advanced idea is to do without a garbage-collection mechanism to dispose of the “deleted” queue end-tail element
 - purpose is to signal unavailability of the end-tail chain link to enqueue
 - thus, when dequeue is going to remove $last$ it attempts to zero $link_{last}$
 - contrariwise, enqueue appends to $last$ only if $link_{last}$ still equals $last$
- signalling as well as validation can be easily achieved using CAS
 - algorithmic construction versus CDS [4, p. 124] or DCAS [8, p. 4-66]...



```
1 void enqueue_lfs(queue_t *this, chain_t *item) {
2     chain_t *last, *hook;
3
4     item->link = item;          /* self-reference: hook */
5
6     do hook = (last = this->tail)->link;    /* tail end */
7     while (!CAS(&this->tail, last, item));
8
9     if (!CAS(&last->link, hook, item))      /* endpiece? */
10        this->head.link = item;             /* no longer! */
11 }
```

■ validate availability of the ending and potential **volatile chain link**:

- 9 ■ CAS succeeds only if the last chain link is still a self-reference
 - in that case, the embracing last element was not dequeued
- 10 ■ CAS fails if the last chain link is no more a self-reference
 - in that case, the embracing last element was dequeued
 - the item to be queued must be head element of the queue, because further enqueues use this very item as leading chain link (l.7)



```
1 chain_t* dequeue_lfs(queue_t *this) {
2     chain_t *node, *next;
3
4     do if ((node = this->head.link) == 0) return 0;
5     while (!CAS(&this->head.link, node,
6         ((next = node->link) == node ? 0 : next)));
7
8     if (next == node) {          /* self-reference, is last */
9         if (!CAS(&node->link, next, 0)) /* try to help */
10             this->head.link = node->link; /* filled */
11         else CAS(&this->tail, node, &this->head);
12     }
13
14     return node;
15 }
```

- validate **tail-end invariance** of a one-element queue ($head = tail$):
 - 9 ■ CAS fails if the node dequeued no more contains a self-reference
 - 10 ■ thus, enqueue happened and left at least one more element queued
 - 11 ■ enqueue was assisted and the dequeued node could be last, really



Outline

Preface

Constructional Axis

General

Exemplification

Transition

Transactional Axis

General

Onefold Update

Twofold Update

Summary



- non-blocking synchronisation $\mapsto$ **abdication of mutual exclusion**
- systems engineering makes a **two-dimensional approach** advisable
 - the *constructional track* brings manageable “complications” into being
 - these “complications” are then subject to a *transactional track*

The latter copes with *non-blocking synchronisation* “in the small”, while the former is a *state-machine outgrowth* using atomic instructions, sporadically, and enables barrier-free operation “in the large”.

- no bed of roses, no picnic, no walk in the park—so is non-blocking synchronisation of reasonably complex simultaneous processes
 - but it constrains sequential operation to the absolute minimum and,
 - thus, paves the way for parallel operation to the maximum possible

Hint (Manyfold Update)

*Solutions for twofold updates already are no “no-brainer”, without or with special instructions such as CDS or DCAS. Major updates are even harder and motivate techniques such as **transactional memory**.*



- [1] DIJKSTRA, E. W.:
Cooperating Sequential Processes / Technische Universiteit Eindhoven.
Eindhoven, The Netherlands, 1965 (EWD-123). –
Forschungsbericht. –
(Reprinted in *Great Papers in Computer Science*, P. Laplante, ed., IEEE Press, New York, NY, 1996)

- [2] HARRIS, T. L.:
A Pragmatic Implementation of Non-blocking Linked-Lists.
In: WELCH, J. (Hrsg.): *Proceedings of the 15th International Conference on Distributed Computing (DISC '01)* Bd. 2180, Springer-Verlag, 2001 (Lecture Notes in Computer Science). –
ISBN 3-540-42605-1, S. 300–314

- [3] HERLIHY, M. :
Wait-Free Synchronization.
In: *ACM Transactions on Programming Languages and Systems* 11 (1991), Jan., Nr. 1, S. 124–149



Reference List II

- [4] IBM CORPORATION (Hrsg.):
IBM System/370 Principles of Operation.
Fourth.
Poughkeepsie, New York, USA: IBM Corporation, Sept. 1 1974.
(GA22-7000-4, File No. S/370-01)

- [5] KICZALES, G. ; LAMPING, J. ; MENDHEKAR, A. ; MAEDA, C. ; LOPES, C. V. ;
LOINGTIER, J.-M. ; IRWIN, J. :
Aspect-Oriented Programming.
In: AKSIT, M. (Hrsg.) ; MATSUOKA, S. (Hrsg.): *Proceedings of the 11th European
Conference on Object-Oriented Programming (ECOOP'97)* Bd. 1241,
Springer-Verlag, 1997 (Lecture Notes in Computer Science). –
ISBN 3-540-63089-9, S. 220-242

- [6] KUNG, H.-T. ; ROBINSON, J. T.:
On Optimistic Methods for Concurrency Control.
In: *ACM Transactions on Database Systems* 6 (1981), Jun., Nr. 2, S. 213-226

- [7] MOIR, M. ; SHAVIT, N. :
"Concurrent Data Structures".
In: MEHTA, D. P. (Hrsg.) ; SAHNI, S. (Hrsg.): *Handbook of Data Structure and
Applications.*
CRC Press, Okt. 2004, Kapitel 47, S. 1-32



Reference List III

- [8] MOTOROLA INC. (Hrsg.):
Motorola M68000 Family Programmer's Reference Manual.
Rev. 1.
Schaumburg, IL, USA: Motorola Inc., 1992.
(M68000PM/AD)

- [9] SCHÖN, F. ; SCHRÖDER-PREIKSCHAT, W. :
*Lock-Free FIFO Queue Using CAS With Simultaneous Consideration of
Conservative Data-Structure Handling.*
Febr./März 2009. –
Discourse

- [10] SCHRÖDER-PREIKSCHAT, W. ; LEHRSTUHL INFORMATIK 4 (Hrsg.):
Concurrent Systems.
FAU Erlangen-Nürnberg, 2014 (Lecture Slides)

- [11] SCHRÖDER-PREIKSCHAT, W. :
Critical Sections.
In: [10], Kapitel 4

- [12] SCHRÖDER-PREIKSCHAT, W. :
Elementary Operations.
In: [10], Kapitel 5



- [13] SCHRÖDER-PREIKSCHAT, W. :
"Guarded Sections".
In: [10], Kapitel 10
- [14] SCHRÖDER-PREIKSCHAT, W. :
Monitor.
In: [10], Kapitel 8
- [15] SCHRÖDER-PREIKSCHAT, W. :
Semaphore.
In: [10], Kapitel 7



Propagate Notifications

```
1  int cause(event_t *this) {
2      chain_t *item;
3      int done = 0;
4
5      if ((item = detach(&this->wait)))
6          do done += hoist((process_t *)
7                          coerce(item, (int)&((process_t *)0)->event),
8                          being(ONESELF)->name);
9          while ((item = item->link));
10
11     return done;
12 }
```

- variant relying on a **dynamic data structure** for the waitlist
 - 5 ■ adopt the waitlist on the whole, indivisible, and wait-free
 - 6–8 ■ notify “go ahead”, pass own identification, and ready signallee
 - 7 ■ pattern a dynamic type-cast from the `chain_t*` member `event` to the `process_t*` of the enclosing process structure (i.e., PCB)
 - 9 ■ notify one process at a time, bounded above, $N - 1$ times at worst



- a simple mechanism that allows a process to “latch onto” an event:

```
1 inline void shade(process_t *this) {
2     this->latch.flag = false;          /* clear latch */
3 }
4
5 inline void stand() {
6     process_t *self = being(ONESELF);
7     if (!self->latch.flag)              /* inactive latch */
8         block();                        /* relinquish... */
9     shade(self);                        /* reset latch */
10 }
11
12 inline void latch() {
13     being(ONESELF)->state |= PENDING;  /* watch for */
14     stand();                            /* & latch */
15 }
```

- 8 ■ either suspend or continue the current process (cf. p.41)
- was marked “pending” to catch a “go ahead” notification (cf. p.12)



- non-blocking measure to signal a single process, one-time, and keep signalling effective, i.e., “sticky” (Ger. *klebrig*) until perceived⁴

```
1 inline void punch(process_t *this) {
2     if (!this->latch.flag) {           /* inactive latch */
3         this->latch.flag = true;       /* activate it */
4         if (this->state & PENDING)     /* is latching */
5             yield(this);              /* set ready */
6     }
7 }
8
9 inline int hoist(process_t *next, int code) {
10     next->merit = code;                /* pass result */
11     punch(next);                      /* send signal */
12     return 1;
13 }
```

- 2–3 ■ assuming that the PCB is not shared by simultaneous processes
■ otherwise, replace by `TAS(&this->latch.flag)` or similar

- 5 ■ makes the process become a “multiple personality”, possibly queued

⁴In contrast to the signalling semantics of monitors (cf. [14, p.8]).



```
1 void block() {
2     process_t *next, *self = being(ONESELF);
3
4     do {                                     /* ...become the idle process */
5         while (!(next = elect(hoard(READY))))
6             relax();                       /* enter processor sleep mode */
7     } while ((next->state & PENDING)        /* clean-up? */
8             && (next->scope != self->scope));
9
10    if (next != self) { /* it's me who was set ready? */
11        self->state = (BLOCKED | (self->state & PENDING));
12        seize(next);    /* keep pending until switch */
13    }
14    self->state = RUNNING;    /* continue cleaned... */
15 }
```

- a “pending blocked” process is still “running” but may also be “ready to run” as to its queueing state regarding the ready list
 - such a process must never be received by another processor (l. 7–8)



- depending on the **waitlist interpretation**, operations to a greater or lesser extent in terms of non-functional properties:

```
1 inline void apply(process_t *this, event_t *list) {
2   #ifdef __FAME_EVENT_WAITLIST__
3     insert(&list->wait, &this->event);
4   #else
5     this->event = list;
6   #endif
7 }
8
9 inline void elide(process_t *this, event_t *list) {
10  #ifdef __FAME_EVENT_WAITLIST__
11    winnow(&list->wait, &this->event);
12  #else
13    this->event = 0;
14  #endif
15 }
```

3/11 ■ dynamic data structure, bounded above, lock-free, lesser list walk

5/13 ■ elementary data type, constant overhead, atomic, larger table walk

