Concurrent Systems

Nebenläufige Systeme

X. Non-Blocking Synchronisation

Wolfgang Schröder-Preikschat

January 22, 2020

Outline

Preface

Constructional Axis
General
Exemplification
Transition

Transactional Axis
General
Onefold Update
Twofold Update

Summary

Subject Matter

- 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

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

Outline

Preface

Constructional Axis
  General
  Exemplification
  Transition

Transactional Axis
  General
  Onefold Update
  Twofold Update

Summary

Reentrancy

(Ger.) Eintrittsinvarianz

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

Semaphore Revisited

cf. [15, p.22]

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

```c
typedef struct semaphore {
  int gate; /* value: binary or general */
  event_t wait; /* list of sleeping processes */
} 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
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.)

---

Atomic Machine Instructions
differences to [15, p. 24/25]

- 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 inline bool unban(semaphore_t *sema) {
  5     return FAA(&sema->gate, +1) < 0;
  6 }
  - avail remains unchanged
  - note that both variants are insensitive to simultaneous processes
  - due to indivisible operations for manipulation of the semaphore value

---

Wait-Action Unfolding

- 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

---

Dualism

- 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

- prepares the “multiple personality” process to be treated in time
- makes the process amenable to “go ahead” notification (p. 10, l. 11)
- 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
Propagate “go ahead” Notifications cf. p.38

- 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)**

```c
inline int coast() {
    stand();    /* latch event */
    return being(ONESELF)->merit;    /* signaller pid */
}
```

```c
int cause(event_t *this) {
    process_t *next;
    int done = 0;

    for (next = being(0); next < being(NPROC); next++)
        if (CAS(&next->event, this, 0))
            done += hoist(next, being(ONESELF)->name);

    return done;
}
```

1. recognise willingness to catch a signal and continue execution
2. 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
  - a constructional axis, as was shown with the semaphore example, and
  - 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

Optimistic Concurrency Control cf. [11, p.15]

**Definition (acc. [6])**

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

```c
do
    read phase:
        save a private copy of the shared data to be updated;
        compute a new private data value based on that copy;
        validation and, possibly, write phase:
            try to commit the computed value as new shared data;
    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
Transactional Computation

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

```c
word_t any; /* shared data */
{
    word_t old, new; /* own data */
    do new = compute(old = any); /* read */
    while (!CAS(& any , old , new)); /* validate / write */
}
```

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

```c
word_t any; /* shared data */
{
    word_t new; /* own data */
    do new = compute(LL(& any )); /* read */
    while (!SC(& any , new )); /* validate / write */
}
```

Data Type I

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

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

whereby a single list element is of the following structure:

```c
typedef struct chain {
    struct chain * link; /* next list element */
} 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

Unsynchronised Operations

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

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

```c
inline chain_t * pull_dos(stack_t * this ) {
    chain_t * node;
    if ((node = this->head.link))
        this->head.link = node->link;
    return node;
}
```

Lock-Free Synchronised Operations

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

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

```c
inline chain_t * pull_lfs(stack_t * this ) {
    chain_t * node;
    do if (!((node = this->head.link) == 0) break;
    while (!CAS(&this->head.link , node , node->link));
    return node;
}
```
Shallowness: ABA Problem

cf. [12, p. 14 & 36–37]

- workaround using a change-number tag as pointer label:
  ```c
  inline void *raw(void *item, long mask) {
    return (void*)((long)item & -mask);
  }
  inline void *tag(void *item, long mask) {
    return (void*)((long)raw(item, mask) | ((long)item + 1) & mask);
  }
  
  - 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: \(2^N \geq \text{sizeof(datastructure)}\)

- zeros the \(N\) low-order bits of the pointer—and discloses the tag field

- a kludge (Ger. Behelfslösung) than a clearcut solution
  - 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
  [3] This also holds for DCAS when using a “whole word” change-number tag.

ABA Problem Tackled I

As Ugly as Sin...

```c
typedef chain_t * chain_l;  /* labelled pointer */
#define BOX (sizeof(chain_t) - 1)  /* tag-field mask */
inline void *push_lfs(stack_t *this, chain_t 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)));
}
inline chain_t *pull_lfs(stack_t *this) {
  chain_l node;
  do if (raw((node = this->head.link), BOX) == 0) break;
  while (!CAS(&this->head.link, node->link, ((chain_t *)raw(node, BOX))->link));
  return node;
}
```

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 \(\rightarrow\) 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!

ABA Problem Tackled II

As provided by the processor plays along

```c
inline chain_t *pull_lfs(stack_t *this) {
  chain_l node;
  do if ((node = LL(&this->head.link)) == 0) break;
  while (!SC(&this->head.link, node->link, ((chain_t *)raw(node, BOX))->link));
  return node;
}
```

...provided the processor plays along

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

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

- copy the head pointer and make a reservation to his address
- update the head pointer if the reservation still exists

```c
inline chain_t *pull_lfs(stack_t *this) {
  chain_l node;
  do if ((node = LL(&this->head.link)) == 0) break;
  while (!SC(&this->head.link, node->link));
  return node;
}
```

- memorise the head pointer and make a reservation to his address
- update the head pointer if the reservation still exists

Data Type II

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

```c
typedef struct queue {
  chain_t head;  /* first item */
  chain_t tail;  /* insertion point */
} 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:

```c
inline chain_t *deplete(queue_t *this) {
  chain_t *head = this->head.link;
  this->head.link = 0;  /* null item */
  this->tail = &this->head;  /* linkage item */
  return head;
}
```

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

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

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

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

```c
inline chain_t* dequeue_dos(queue_t *this) {
    chain_t *node;
    if ((node = this->head.link) == 0)
        return node;
    if ((node = this->head.link = node->link))
        this->tail = &this->head; /* reset */
    return node;
}
```

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

---

Synchronisation, Take Two: **deq||deq**

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

```c
chain_t* dequeue_lfs(queue_t *this, chain_t *item) {
    chain_t *last;
    if ((node = this->head.link) == 0) return 0;
    while (!CAS(&this->head.link, node, node->link));
    if (node->link == 0)
        this->tail = &this->head;
    return node;
}
```

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

---

Synchronisation, Take Three: **enq||enq**

- **only single shared variable needs to be updated in this scenario.**

```c
void enqueue_lfs(queue_t *this, chain_t *item) {
    chain_t *last;
    item->link = 0;
    do last = this->tail;
       while (!CAS(&this->tail, last, item));
    last->link = item;
}
```

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

---

Synchronisation, Take One: **enq||enq**

- **hint (onefold update):**

```c
inline chain_t* enqueue_lfs(queue_t *this, chain_t *item) {
    chain_t *last;
    item->link = 0;
    do last = this->tail;
       while (!CAS(&this->tail, last, item));
    last->link = item;
    return node;
}
```

- **only single shared variable needs to be updated in this scenario.**

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

---

Neuralgic Points

- 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:
  - 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
  - dequeue assumes “vacancy” and resets the tail pointer

- enqueue and dequeue must assist each other to solve the problem:
  1. identify the conditions under which lost-enqueue may happen
  2. identify a way of interaction between enqueue and dequeue
- assist without special auxiliary nodes but preferably with simultaneous consideration of **conservative data-structure handling**
Synchronisation, Take Four

- 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 \( \text{link}_{last} \) be the chain link pointed to by \( last \), then:

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

- \( \text{link}_{last} \) 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 \( \text{link}_{last} \)
  - contrariwise, enqueue appends to \( last \) only if \( \text{link}_{last} \) 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]

Outline

- Preface
- Constructional Axis
- General
- Exemplification
- Transition
- Transactional Axis
- General
- Onefold Update
- Twofold Update
- Summary

© wosch, theoenig CS (WS 2019/20, LEC 10)
Résumé

- non-blocking synchronisation $\rightarrow$ 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.

Reference List I


In: ACM Transactions on Programming Languages and Systems 11 (1991), Jan., Nr. 1, S. 124–149

Reference List II

[4] IBM CORPORATION (Hrsg.):

Aspect-Oriented Programming.

On Optimistic Methods for Concurrency Control.

[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.):
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)

Critical Sections.
In: [10], Kapitel 4

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

```c
int cause(event_t *this) {
    chain_t *item;
    int done = 0;
    if ( (item = detach(&this->wait)) )
        do done += hoist((process_t *)
            coerce(item, (int)&((process_t *)0)->event),
            being(ONESELF)->name);
    while ((item = item->link));
    return done;
}
```

Receive-Side “Sticky Bit” Operations

- A simple mechanism that allows a process to “latch onto” an event:
  ```c
  inline void shade(process_t *this) {
      this->latch.flag = false; /* clear latch */
  }
  ```

- For non-blocking measure to signal a single process, one-time, and keep signalling effective, i.e., “sticky” (Ger. klebrig) until perceived
  ```c
  inline void punch(process_t *this) {
      if (!this->latch.flag) { /* inactive latch */
          this->latch.flag = true; /* activate it */
          if (this->state & PENDING) /* is latching */
              yield(this); /* set ready */
      }
  }
  ```

Send-Side “Sticky Bit” Operations

- Non-blocking measure to signal a single process, one-time, and keep signalling effective, i.e., “sticky” (Ger. klebrig) until perceived
  ```c
  inline int hoist(process_t *next, int code) {
      next->merit = code; /* pass result */
      punch(next); /* send signal */
      return 1;
  }
  ```

1. **Reference List IV**
   - **Schröder-Preikschat, W.**: "Guarded Sections". In: [10], Kapitel 10
   - **Schröder-Preikschat, W.**: Monitor. In: [10], Kapitel 8
   - **Schröder-Preikschat, W.**: Semaphore. In: [10], Kapitel 7


   - A simple mechanism that allows a process to “latch onto” an event:
     ```c
     inline void shade(process_t *this) {
         this->latch.flag = false; /* clear latch */
     }
     ```

3. **Send-Side “Sticky Bit” Operations** cf. p. 13

   - Non-blocking measure to signal a single process, one-time, and keep signalling effective, i.e., “sticky” (Ger. klebrig) until perceived
     ```c
     inline void punch(process_t *this) {
         if (!this->latch.flag) { /* inactive latch */
             this->latch.flag = true; /* activate it */
             if (this->state & PENDING) /* is latching */
                 yield(this); /* set ready */
         }
     }
     ```

4. **Receive-Side “Sticky Bit” Operations**

   - A simple mechanism that allows a process to “latch onto” an event:
     ```c
     inline void shade(process_t *this) {
         this->latch.flag = false; /* clear latch */
     }
     ```

5. **Send-Side “Sticky Bit” Operations**

   - Non-blocking measure to signal a single process, one-time, and keep signalling effective, i.e., “sticky” (Ger. klebrig) until perceived
     ```c
     inline int hoist(process_t *next, int code) {
         next->merit = code; /* pass result */
         punch(next); /* send signal */
         return 1;
     }
     ```

6. **Receive-Side “Sticky Bit” Operations**

   - A simple mechanism that allows a process to “latch onto” an event:
     ```c
     inline void shade(process_t *this) {
         this->latch.flag = false; /* clear latch */
     }
     ```

7. **Send-Side “Sticky Bit” Operations**

   - Non-blocking measure to signal a single process, one-time, and keep signalling effective, i.e., “sticky” (Ger. klebrig) until perceived
     ```c
     inline void punch(process_t *this) {
         if (!this->latch.flag) { /* inactive latch */
             this->latch.flag = true; /* activate it */
             if (this->state & PENDING) /* is latching */
                 yield(this); /* set ready */
         }
     }
     ```

8. **Receive-Side “Sticky Bit” Operations**

   - A simple mechanism that allows a process to “latch onto” an event:
     ```c
     inline void shade(process_t *this) {
         this->latch.flag = false; /* clear latch */
     }
     ```

9. **Send-Side “Sticky Bit” Operations**

   - Non-blocking measure to signal a single process, one-time, and keep signalling effective, i.e., “sticky” (Ger. klebrig) until perceived
     ```c
     inline int hoist(process_t *next, int code) {
         next->merit = code; /* pass result */
         punch(next); /* send signal */
         return 1;
     }
     ```

10. **Receive-Side “Sticky Bit” Operations**

   - A simple mechanism that allows a process to “latch onto” an event:
     ```c
     inline void shade(process_t *this) {
         this->latch.flag = false; /* clear latch */
     }
     ```

11. **Send-Side “Sticky Bit” Operations**

   - Non-blocking measure to signal a single process, one-time, and keep signalling effective, i.e., “sticky” (Ger. klebrig) until perceived
     ```c
     inline void punch(process_t *this) {
         if (!this->latch.flag) { /* inactive latch */
             this->latch.flag = true; /* activate it */
             if (this->state & PENDING) /* is latching */
                 yield(this); /* set ready */
         }
     }
     ```

12. **Receive-Side “Sticky Bit” Operations**

   - A simple mechanism that allows a process to “latch onto” an event:
     ```c
     inline void shade(process_t *this) {
         this->latch.flag = false; /* clear latch */
     }
     ```

13. **Send-Side “Sticky Bit” Operations**

   - Non-blocking measure to signal a single process, one-time, and keep signalling effective, i.e., “sticky” (Ger. klebrig) until perceived
     ```c
     inline int hoist(process_t *next, int code) {
         next->merit = code; /* pass result */
         punch(next); /* send signal */
         return 1;
     }
     ```

14. **Receive-Side “Sticky Bit” Operations**

   - A simple mechanism that allows a process to “latch onto” an event:
     ```c
     inline void shade(process_t *this) {
         this->latch.flag = false; /* clear latch */
     }
     ```

15. **Send-Side “Sticky Bit” Operations**

   - Non-blocking measure to signal a single process, one-time, and keep signalling effective, i.e., “sticky” (Ger. klebrig) until perceived
     ```c
     inline void punch(process_t *this) {
         if (!this->latch.flag) { /* inactive latch */
             this->latch.flag = true; /* activate it */
             if (this->state & PENDING) /* is latching */
                 yield(this); /* set ready */
         }
     }
     ```

16. **Receive-Side “Sticky Bit” Operations**

   - A simple mechanism that allows a process to “latch onto” an event:
     ```c
     inline void shade(process_t *this) {
         this->latch.flag = false; /* clear latch */
     }
     ```

17. **Send-Side “Sticky Bit” Operations**

   - Non-blocking measure to signal a single process, one-time, and keep signalling effective, i.e., “sticky” (Ger. klebrig) until perceived
     ```c
     inline int hoist(process_t *next, int code) {
         next->merit = code; /* pass result */
         punch(next); /* send signal */
         return 1;
     }
     ```

In contrast to the signalling semantics of monitors (cf. [14, p. 8]).
void block() {
    process_t *next, *self = being(OONESELF);
    do {
        /* ...become the idle process */
        while (!(next = elect(hoard(READY))))
            relax(); /* enter processor sleep mode */
    } while (((next->state & PENDING) /* clean-up? */
        && (next->scope != self->scope));
    if (next != self) { /* it's me who was set ready? */
        self->state = (BLOCKED | (self->state & PENDING));
        seize(next); /* keep pending until switch */
    }
    self->state = RUNNING; /* continue cleaned... */
}

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)

---

inline void apply(process_t *this, event_t *list) {
    #ifdef __FAME_EVENT_WAITLIST__
        insert(&list->wait, &this->event);
    #else
        this->event = list;
    #endif
}

inline void elide(process_t *this, event_t *list) {
    #ifdef __FAME_EVENT_WAITLIST__
        winnow(&list->wait, &this->event);
    #else
        this->event = 0;
    #endif
}

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

3/11 ■ dynamic data structure, bounded above, lock-free, lesser list walk
5/13 ■ elementary data type, constant overhead, atomic, larger table walk

---

© wosch, thoenig  CS (WS 2019/20, LEC 10)  Addendum – Re-Entrant Operations