Concurrent Systems

Nebenläufige Systeme

X. Non-Blocking Synchronisation

Wolfgang Schröder-Preikschat

January 15, 2019

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?
- following 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

(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

Semaphore Revisited

cf. [15, p. 22]

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

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

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 latter

originally, this property was typical for an interrupt handler, merely, that allows for nested execution—recursion not unresembling
- 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.
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 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.)

---

**Building Blocks for Barrier-Free Operation**

- **Atomic Machine Instructions**
  - load/store-based implementation for a **binary semaphore**:
    ```
    inline bool avail(semaphore_t *sema) {
      return CAS(&sema->gate, 1, 0);
    }
    ```
    - both *lodge* and *unban* remain unchanged
  - enumerator-based implementation for a **general semaphore**:
    ```
    inline int lodge(semaphore_t *sema) {
      return FAA(&sema->gate, -1);
    }
    ```
    ```
    inline bool unban(semaphore_t *sema) {
      return FAA(&sema->gate, +1) < 0;
    }
    ```
    - *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**

```null
void prolaag(semaphore_t *sema) {
  catch(&sema->wait); /* expect notification */
  lodge(sema); /* raise claim to proceed */
  when(!avail(sema)) /* check for process delay */
    coast(); /* accept wakeup signal */
  clean(&sema->wait); /* forget notification */
}
```
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(OONESELF)->merit;  /* signaller pid */
}
```

```
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(OONESELF)->name);

    return done;
}
```

recognise willingness to catch a signal and continue execution
notify “go ahead”, pass own identification, and ready signallee

---

**Outline**

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

---

**Optimistic Concurrency Control**

*cf. [11, p.15]*

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;
}
```

- copy the contents of the stack pointer to the item to be stacked
- update the stack pointer with the address of that 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;
}
```

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

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));
}
```

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

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

- memorise the item located at the stack top, if any
- attempt to update the stack pointer with the address of the next item
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);
}
```

```c
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
- rather a kludge (Ger. *Behelfslösung*) than a clearcut solution\(^3\)
- 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

As Ugly as Sin...

```c
typedef chain_t* chain_l; /* labelled pointer */
#define BOX (sizeof(chain_t) - 1) /* tag-field mask */
```

```c
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)));
}
```

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

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 *drain(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

En-/Dequeue

- 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 chart_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* fetch_dos(queue_t *this) {
    chain_t *node;
    if ((node = this->head.link) & ~mask) {
        this->tail = &this->head; /* reset */
        return node;
    }
```

- the tail pointer must always be valid, even in case of an empty queue
Synchronisation, Take One: chart||chart Lock-Free

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

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

**Hint (Onefold Update)**

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

Synchronisation, Take Two: fetch||fetch Lock-Free

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

```c
chain_t* fetch_lfs(queue_t *this) {
  chain_t *node;
  do 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;
}
```

**Hint (Onefold Update)**

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

Synchronisation, Take Three 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:
  - each overlapping enqueue enqueues its own copy of the tail pointer
  - validate and, if applicable, write to update the tail pointer
  - the item becomes new fastener for subsequent enqueue operations
  - eventually, the item gets inserted and becomes queue member
  - the assignment operator works on local operands, only

Synchronisation, Take Four Forgo CDS or DCAS, resp.

- 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
- let linklast be the chain link pointed to by last, then:

```c
linklast = \{
  last, chain link is valid, was not deleted
  0, chain link is invalid, was deleted
  else, chain link points to successor element
```

- linklast 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 linklast
- contrariwise, enqueue appends to last only if linklast 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]...
void chart_lfs(queue_t *this, chain_t *item) {
    chain_t *last, *hook;

    item->link = item;  /* self-reference: hook */
    do hook = (last = this->tail)->link;  /* tail end */
        while (!CAS(&this->tail, last, item));

    if (!CAS(&last->link, hook, item))  /* endpiece? */
        this->head.link = item;  /* no longer! */
}

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)

chain_t* fetch_lfs(queue_t *this) {
    chain_t *node, *next;

    do if ((node = this->head.link) == 0) return 0;
        while (!CAS(&this->head.link, node,
                (next = node->link) == node ? 0 : next));

    if (next == node) {  /* self-reference, is last */
        if (!CAS(&node->link, next, 0)) /* try to help */
            this->head.link = node->link;  /* filled */
        else CAS(&this->tail, node, &this->head);
    } else

    return node;
}

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

Résumé

non-blocking synchronisation ↦ 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**

[1] **Dijkstra, E. W.**

Cooperating Sequential Processes / Technische Universiteit Eindhoven.
Forschungsbericht. –

[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). –

[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.).**

(GA22-7000-4, File No. S/370-01)


Aspect-Oriented Programming.


On Optimal 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.
(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

**Reference List IV**

[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

```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(OONESELF)->name);
    while ((item = item->link));

    return done;
}
```

Receive-Side “Sticky Bit” Operations cf. p. 13

```c
inline void shade(process_t *this) {
    this->latch.flag = false; /* clear latch */
}

inline void stand() {
    process_t *self = being(OONESELF);
    if (!self->latch.flag) /* inactive latch */
        block(); /* relinquish... */
    shade(self); /* reset latch */
}

inline void latch() {
    being(OONESELF)->state |= PENDING; /* watch for */
    stand(); /* & latch */
}
```

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

```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 */
}

inline int hoist(process_t *next, int code) {
    next->merit = code; /* pass result */
    punch(next); /* send signal */
    return 1;
}
```

Resolving Multiple Personality cf. [15, p. 37]

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

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

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

- assuming that the PCB is not shared by simultaneous processes
- otherwise, replace by TAS(&this->latch.flag) or similar
- makes the process become a “multiple personality”, possibly queued

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)

either suspend or continue the current process (cf. p. 40)
- was marked “pending” to catch a “go ahead” notification (cf. p.12)

was marked “pending” to catch a “go ahead” notification (cf. p. 12)
- makes the process become a “multiple personality”, possibly queued
depending on the **waitlist interpretation**, operations to a greater or lesser extent in terms of non-functional properties:

```c
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
}
```

- dynamic data structure, bounded above, lock-free, lesser list walk
- elementary data type, constant overhead, atomic, larger table walk