Agenda

Preface

Constructional Axis
  General
  Examplification
  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

Source: Commemorative plaque, Berlin, Bundesallee 79

Outline
Preface
Constructional Axis
General
Examplification
Transition
Transactional Axis
General
Onefold Update
Twofold Update
Summary

Reentrancy
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 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

Semaphore Revisited

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

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 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 cf. [15, p. 24]

■ load/store-based implementation for a binary semaphore:
1 inline bool claim(semaphore_t *sema) {
2 \quad return CAS(&sema->gate, 1, 0); 
3 }
4
5 inline bool unban(semaphore_t *sema) {
6 \quad return (sema->gate = 1);
7 }
8
9 ■ enumerator-based implementation for a general semaphore:
1 inline bool claim(semaphore_t *sema) {
2 \quad return FAA(&sema->gate, 1) > 0;
3 }
4
5 inline bool unban(semaphore_t *sema) {
6 \quad return FAA(&sema->gate, -1) < 0;
7 }
8
9 ■ note that both variants are insensitive to simultaneous processes
10 \quad due to indivisible operations for manipulation of the semaphore value

Wait-Action Unfolding cf. [15, p. 23]

1 void prolaag(semaphore_t *sema) {
2 \quad catch(&sema->wait); /* expect notification */
3 \quad while (!claim(sema)) /* acquire semaphore */
4 \quad \quad coast(); /* accept wakeup signal */
5 \quad clean(&sema->wait); /* forget notification */
6 }
7
8 void verhoog(semaphore_t *sema) {
9 \quad if (unban(sema)) /* release semaphore */
10 \quad \quad cause(&sema->wait); /* notify wakeup signal */
11 }
12
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

■ prepare the “multiple personality” process to be treated in time

■ 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

■ replaces the process from potential receive of “go ahead” notifications

■ makes the process amenable to “go ahead” notification (p. 10, l.10)

■ 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. 41)

■ “ready” waitlist, the respective idle process of a processor (p. 40)

■ “blocked” waitlist, the semaphore increasing or decreasing process
Propagate “go ahead” Notifications cf. p. 37

- 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)**

```
    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;
    }
```

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

---

**Outline**

Preface

Constructional Axis
General
Examplification
Transition

Transactional Axis
General
Onefold Update
Twofold Update

Summary

---

**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

---

**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.

```
    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).
    while 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 */
}
```

CAS recreated using LL/SC (cf. [12, p. 16])

Is prone to a race condition: e.g., act out simple counting...

Unsynchronised Operations

Devoid of Synchronisation

- 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;
}
```

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
Shallowness: ABA Problem

- workaround using a change-number tag as pointer label:
  - inline void *raw(void *item, long mask) {
  2     return (void*)((long)item & ~mask);
  3  }
  4
  inline void *tag(void *item, long mask) {
  5     return (void*)((long)item + 1) & mask;
  6  }
  7
  - 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.

### 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

ABA Problem Tackled

- 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_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) == 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...

### 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 chart_dos(queue_t *this, chain_l *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) != node) /* filled? */
        this->tail = &this->head; /* reset */
    return node;
}
```

- 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:

- 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
  \[\rightarrow last enqueue\]: linking depends on dequeue progression

- dequeue removed that element and notices “vacancy”
- enqueue appends an element to the one just removed
- dequeue assumes “vacancy” and resets the tail pointer
  \[\rightarrow last enqueue\]: resetting depends on enqueue progression

enqueue and dequeue must assist each other to solve the problem:

- identify the conditions under which lost-enqueue may happen
- identify a way of interaction between enqueue and dequeue
- assist without special auxiliary nodes but preferably with simultaneous consideration of conservative data-structure handling
lock-free synchronised operations I

enqueue acc. [9]

```
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)

lock-free synchronised operations II

dequeue acc. [9]

```
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);
    }

    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


Reference List II


Reference List III


Reference List IV

[13] SCHRÖDER-PREIKSCHAT, W.: “Guarded Sections”. In: [10], Kapitel 10


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;
}
```

- **Variant relying on a dynamic data structure** for the waitlist
  5-8 notify "go ahead", pass own identification, and ready signalleee
  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

---

Send-Side “Sticky Bit” Operations

```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 */
    }
}
```

- non-blocking measure to signal a single process, one-time, and keep signalling effective, i.e., “sticky” (Ger. klebrig) until perceived\(^4\)
- assuming that the PCB is not shared by simultaneous processes
- otherwise, replace by \(\text{TAS}(&\text{this->latch.flag})\) or similar
- makes the process become a “multiple personality”, possibly queued

| 2-3 | assuming that the PCB is not shared by simultaneous processes |
| 5   | otherwise, replace by \(\text{TAS}(&\text{this->latch.flag})\) or similar |

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

---

Receive-Side “Sticky Bit” Operations

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

- a simple mechanism that allows a process to “latch onto” an event:
- either suspend or continue the current process (cf. p.40)
- was marked “pending” to catch a "go ahead" notification (cf. p.12)

---

Resolving Multiple Personality

```c
void block() {
    process_t *next, *self = being(OONESELF);
    do {
        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));
                }
        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)
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
}
```

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