Concurrent Systems

Nebenläufige Systeme

X. Basics of Non-Blocking Synchronisation

Wolfgang Schröder-Preikschat

January 21, 2021

Outline

Preface

Constructional Axis
  General
  Exemplification
  Transition

Transactional Axis
  General
  Case Study

Summary

Subject Matter

- discussion on abstract concepts of synchronisation without lockout of critical action sequences of interacting processes (cf. [5])
  - 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

Case Study

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 unreasoning
- 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.

Semaphore Revisited

cf. [10, p.22]

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
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 [2, 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 [2, 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 [10, p. 24/25]

- load/store-based implementation for a **binary semaphore**:
  ```c
  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**:
  ```c
  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** cf. [10, p. 23]

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>void prolaag(semaphore_t *sema)</code> {</td>
</tr>
<tr>
<td>2</td>
<td><code>catch(&amp;sema-&gt;wait );</code> /* expect notification */</td>
</tr>
<tr>
<td>3</td>
<td><code>lodge(sema);</code> /* raise claim to proceed */</td>
</tr>
<tr>
<td>4</td>
<td><code>when(!avail(sema))</code> /* check for process delay */</td>
</tr>
<tr>
<td>5</td>
<td><code>coast();</code> /* accept wake up signal */</td>
</tr>
<tr>
<td>6</td>
<td><code>clean(&amp;sema-&gt;wait);</code> /* forget notification */</td>
</tr>
<tr>
<td>7</td>
<td>}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td><code>void verhoog(semaphore_t *sema)</code> {</td>
</tr>
<tr>
<td>9</td>
<td><code>if (unban(sema))</code> /* release semaphore */</td>
</tr>
<tr>
<td>10</td>
<td><code>cause(&amp;sema-&gt;wait);</code> /* notify wake up signal */</td>
</tr>
<tr>
<td>11</td>
<td>}</td>
</tr>
</tbody>
</table>

---

**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
  ```c
  inline void catch(event_t *this) {
    process_t *self = being(ONESELF);
    self->state |= PENDING; /* watch for event */
    apply(self, this); /* enter wait list */
  }
  ```

- 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. 34)
  - “ready” waitlist, the respective idle process of a processor (p. 33)
  - “blocked” waitlist, the semaphore increasing or decreasing process
Propagate “go ahead” Notifications cf. p. 30

- 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 */
}
```

```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(OONESELF)->name);
    return done;
}
```

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

Optimistic Concurrency Control cf. [7, p. 15]

**Definition (acc. [4])**
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

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, [3])**
- 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
  - Case Study
- Summary
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. [8, p.16])

Reading phase carried out simultaneously remains undetected...

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 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;
}
```
Shallowness: ABA Problem

cf. [8, p. 14 & 36–37]

given a LIFO list (i.e., stack) of following structure: head $\rightarrow$ A $\rightarrow$ B $\rightarrow$ C

- with head stored at location $L_i$, shared by processes $P_1$ and $P_2$
- furthermore assume actual parameter this is pointing to $L_i$

\[
\text{inline chain_t *pull_lfs(stack_t *this) \{}
\text{chain_t *node;}
\text{do if } ((\text{node} = \text{this} \rightarrow \text{head}.\text{link}) == 0) \text{ break;}
\text{while } ((\text{CAS}(&\text{this} \rightarrow \text{head}.\text{link}, \text{node}, \text{node} \rightarrow \text{link}));
\text{return node;}
\}}
\]

assuming that the following sequence of actions will take place:

$P_1$
- reads head item A followed by B on the list, gets delayed at line 4
- remembers node = A, but has not yet done CAS: head $\rightarrow$ A $\rightarrow$ B $\rightarrow$ C

$P_2$
- pulls head item A from the list:
  head $\rightarrow$ B $\rightarrow$ C
- pulls head item B from the list:
  head $\rightarrow$ C
- pushes item A back to the list, now followed by C: head $\rightarrow$ A $\rightarrow$ C

$P_1$
- resumes, CAS realises head = A (followed by B):
  head $\rightarrow$ B $\forall$
- list state head $\rightarrow$ A $\rightarrow$ C as left behind by $P_2$ is lost...

ABA Problem Tackled I

...as ugly as sin

```c
#define BOX (sizeof(chain_t) - 1) /* labelled pointer! */
inline void *raw(void *item, long mask) {
  return (void *)((long)item & ~mask);
}
```

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

...provided the processor plays along

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

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

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

1. copy the head pointer and make a reservation to his address
2. update the head pointer if the reservation still exists

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

8. memorise the head pointer and make a reservation to his address
9. update the head pointer if the reservation still exists
Résumé

- 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.*

---

### Reference List I

1. **Dijkstra, E. W.**:

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

3. **Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopes, C. V.; Loingtier, J.-M.; Irwin, J.** :
   Aspect-Oriented Programming.

4. **Kung, H.-T.; Robinson, J. T.** :
   On Optimistic Methods for Concurrency Control.

---

### Reference List II

5. **Moir, M.; Shavit, N.** :
   “Concurrent Data Structures”.

6. **Schröder-Preikschat, W.; Lehrstuhl Informatik 4 (Hrsg.):**
   *Concurrent Systems*.
   FAU Erlangen-Nürnberg, 2014 (Lecture Slides)

7. **Schröder-Preikschat, W.** :
   Critical Sections.
   In: [6], Kapitel 4

8. **Schröder-Preikschat, W.** :
   Elementary Operations.
   In: [6], Kapitel 5

9. **Schröder-Preikschat, W.** :
   Monitor.
   In: [6], Kapitel 8
Reference List III

   In: [6], 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(ONESELF)->name);
  while ((item = item->link));

  return done;
}
```

- 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

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

```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:

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

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

- either suspend or continue the current process (cf. p.33)
- was marked “pending” to catch a “go ahead” notification (cf. p.12)

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 */
  }
}
```

- 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;
}
```

- 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

4In contrast to the signalling semantics of monitors (cf. [9, p.8]).
Resolving Multiple Personality

cf. [10, 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) && (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)

Waitlist Association
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