Concurrent Systems

Nebenläufige Systeme

XI. Non-Blocking Synchronisation

Wolfgang Schröder-Preikschat

December 18, 2014

Outline

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

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

---

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:

1 typedef struct semaphore {
  2   int gate; /* value: binary or general */
  3   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
Building Blocks for Barrier-Free Operation

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

Atomic Machine Instructions cf. [15, p. 24]

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

```c
1 inline bool claim(semaphore_t *sema) {
2     return CAS(&sema->gate, 1, 0);
3 }
4
5 inline bool unban(semaphore_t *sema) {
6     return (sema->gate = 1);
7 }
```

- enumerator-based implementation for a general semaphore:

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

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

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

```c
void prolaag(semaphore_t *sema) {
    catch(&sema->wait); /* expect notification */
    coast(); /* acquire semaphore */
    /* expect notification */
}

void verhoog(semaphore_t *sema) {
    if (unban(sema)) /* release semaphore */
        cause(&sema->wait); /* notify wakeup signal */
}
```

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 waitlist */
}

inline void clean(event_t *this) {
    elide(being(ONESELF), this); /* leave waitlist */
}
```

- prepares the “multiple personality” process to be treated in time
- 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. 10)
  - “blocked” waitlist, the semaphore increasing or decreasing process
Propagate “go ahead” Notifications

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 (ONESELF)->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 (ONESELF)->name);

  return done;
}

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

non-blocking synchronisation spans two dimensions of measures in the organisation of a non-sequential program:
i a constructional axis, as was shown with the semaphore example, and
ii a transactional axis, which is coming up in the next section

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

Optimistic Concurrency Control

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).
  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

Outline

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

- CAS-oriented approach, value-based, typical for CISC:
  1. `word_t any; /* shared data */`
  2. `{ `word_t old, new; /* own data */`
  3. `do new = compute(old = any); /* read */`
  4. `while (!CAS(&any, old, new)); /* validate/write */`
  5. `}`

- LL/SC-oriented approach, reservation-based, typical for RISC:
  1. `word_t any; /* shared data */`
  2. `{ `word_t new; /* own data */`
  3. `do new = compute(LL(&any)); /* read */`
  4. `while (!SC(&any, new)); /* validate/write */`
  5. `}`

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

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

Data Type I

- let a very simple dynamic data structure be object of investigation
  - modelling a stack in terms of a single-linked list:
    1. `typedef struct stack {
       2. `chain_t head; /* top of stack: list head */`
       3. `} stack_t;`

    whereby a single list element is of the following structure:

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

- stack manipulation by pushing or pulling an item involves the update of a single variable, only: the “stack pointer”

- when simultaneous processes are allowed to interact by sharing that stack structure, the update must be an indivisible operation

Unsynchronised Operations

- basic precondition: an item to be stacked is not yet stacked/queued
  1. `inline void push_dos(stack_t *this, chain_t *item) {
     2. `item->link = this->head.link;
     3. `this->head.link = item;`
     4. `}
  5. `}`

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

- inline chain_t *pull_dos(stack_t *this) {
    1. `chain_t *node;
    2. `if ((node = this->head.link))
    3. `this->head.link = node->link;
    4. `return node;`
  5. `}

    - memorise the item located at the stack top, if any
    - 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”
  1. `inline void push_lfs(stack_t *this, chain_t *item) {
     2. `do item->link = this->head.link;
     3. `while (!CAS(&this->head.link, item->link, item));`
     4. `}
  5. `}`

    - 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

- inline chain_t *pull_lfs(stack_t *this) {
    1. `chain_t *node;
    2. `do if ((node = this->head.link) == 0) break;
    3. `while (!CAS(&this->head.link, node, node->link));`
    4. `return node;`
  5. `}

    - 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

- workaround using a change-number tag as pointer label:
  - 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 solution³

- makes ambiguities merely unlikely, but cannot prevent them

- “operation frequency” must be in line with the finite values margin

- if applicable, attempt striving for problem-specific frequency control

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

ABA Problem Tackled

As Ugly as Sin...

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) {
inline chain_t* chain_l;
  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!

 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

inline void chart_dos(queue_t*this, chain_t* item) {
  item->link = 0; /* finalize 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

inline chain_t* purge_dos(queue_t *this) {
  chain_t* node;
  if (((node = this->head.link) & 0) == 0) { /* filled */
    &this->head.link = node->link); /* last item */
    this->tail = &this->head; /* reset */
  return node;
}

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

- 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->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. each overlapping enqueue holds its own copy of the tail pointer
  5. eventually, the item gets inserted and becomes queue member
  6. the assignment operator works on local operands, only

Synchronisation, Take Three

- 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: the chain link of that element
  - dequeue: the element including the chain link
  - enqueue: the new element using an invalid chain link
  - dequeue: 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

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 link_{last} be the chain link pointed to by last, then:

```
link_{last} = |
| last, chain link is valid, was not deleted |
| 0, chain link is invalid, was deleted |
| else, chain link points to successor element |
```

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

Enqueue acc. [9]

```c
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 dequeuing 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]

```c
chain_t* purge_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

Outline

Preface

Constructional Axis
   General
   Examplification
   Transition

Transactional Axis
   General
   Onefold Update
   Twofold Update

Summary

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.


int cause(event_t *this) {
    chain_t *item;
    int done = 0;

    if ((item = detach(&this->wait)))
        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

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

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

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

Send-Side “Sticky Bit” Operations

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;
}

Receive-Side “Sticky Bit” Operations

Resolving Multiple Personality

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

void block() {
    process_t *next, *self = being(ONESELF);
    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));
        seize(next);  /* keep pending until switch */
    } else self->state = RUNNING;    /* continue cleaned... */
}

Send-Side “Sticky Bit” Operations

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

Resolving Multiple Personality

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

In contrast to the signalling semantics of monitors (cf. [14, p.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