Concurrent Systems

Nebenläufige Systeme

XIII. Progress Guarantee

Wolfgang Schröder-Preikschat

January 27, 2016

Agenda

In a Nutshell

Outline

In a Nutshell

Liveness Properties

- characteristics of algorithms for **non-blocking synchronisation**:
  - obstruction-free
    - if any process eventually in isolation (i.e., absence of simultaneously interacting processes) can complete any operation in a finite number of steps [2]
    - prone to **starvation** of conflicting processes
  - lock-free
    - if "some process will complete an operation in a finite number of steps, regardless of the relative speeds of the processes" [1, p. 142][1]
    - free of **starvation** of at least one conflicting process
  - wait-free
    - if "any process can complete any operation in a finite number of steps, regardless of the relative speeds of the other processes" [1, p. 7]
    - free of starvation of any conflicting process
- all, no process can be blocked by delays or failures of other processes
  - that is to say, any of these procedures ensures **deadlock freedom**

---

[1] Originally, also referred to as "nonblocking".

Transactional Computation I

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

- at a first glance, this looks like lock-free progress guarantee:
  - CAS ensures that one out of possibly many conflicting processes succeeds
  - yet, which one is not determined and process overhauling[2] is facilitated
- however, a closer look may even reveal only obstruction freedom:
  - progress guarantee stands and falls with the properties of *compute*
    - stands – lock-free if *compute* is either lock- or wait-free
    - falls – off to obstruction-free if *compute* is obstruction-free
  - thus, the likewise weaker property of *compute* comes out on top
- strengthening of the progress guarantee at a higher level is illusionary
  - without willingness to break abstraction or black-box reuse, resp., down

---

[2] A desirable property given priority-based process scheduling.

Transactional Computation II

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

- at a first glance, the progress guarantee is the same as before (p. 5):
  - SC ensures that one out of possibly many conflicting processes succeeds
  - also, the likewise weaker property of *compute* comes out on top
- however, a closer look may even reveal a further dependency:
  - LL
    - besides reading from memory, typically performs two actions:
      - i make a reservation for a hardware-dependent address range
      - ii as the case may be, remember the effective address of the location
  - SC
    - if a reservation exists, overwrite the addressed location
      - – if applicable, only if the applied address matches the remembered one
    - in any case, cancel a possibly existing reservation
- plain sequences of LL/SC may be obstruction-free, only (cf. p. 12)
  - **exceptions** (i.e., traps/interrupts) may or may not cancel reservations
  - similar may hold for specific ("manually ejected") memory operations

Table-Based Scheduling I

- bear in mind that each process also acts as a "***feeder***" of a CPU core
  - normally, a process releases its processor either voluntarily or involuntarily
  - it either blocks or yields, or it gets the processor revoked (preemption)
- each feeder takes the scheduling decision in a finite number of steps:

```
process_t *elect(hoard_t *vain) {
    process_t *next;

    for (next = being(0); next < being(NPROC); next++) {
        if (next->state != READY)
            continue;

        if (CAS(&next->state, READY, READY | PENDING))
            return next;
    }

    return 0;
}
```

- the scheduling loop is **bounded** by the number of process descriptors

Table-Based Scheduling II

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

    while ((next = uncage(&this->wait))) {
        next->merit = being(ONESELF)->name;
        next->state = READY;
        done += 1;
    }

    return done;
}
```

uncage
- attempt to *purge* a chain item from the queue-based per-event waitlist ⇝ lock-free semantics in case of [3, p. 30]
- if succeeded, coerce the pointer to the purged chain item into the pointer to the enclosing process descriptor (cf. p. 13)

- even in case of a wait-free *purge*, the schedule loop is **unbounded**
  - a former uncaged process may have been dispatched in the meantime
  - ↪ assume that the process blocks on the same event while signalling (*cause*)

Table-Based Scheduling III

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

    while ((done < N) && (next = uncage(&this->wait))) {
        next->merit = being(ONESELF)->name;
        next->state = READY;
        done += 1;
    }

    return done;
}
```

- a WCET[3] of the schedule loop is given only with a wait-free *purge*
    - in that case: $WCET(loop) \leq N \times WCET(purge)$, thus bounded
- but in case of a lock-free *purge*, the schedule loop is lock-free also
    - a lock-free *purge* implies a **possibly unbounded** latency until returning
- last but not least, similar holds in case of an obstruction-free *purge*. . .

---

[3] *worst-case execution time*

Table-Based Scheduling IV

- each signaller takes the scheduling decision in a finite number of steps

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

    for (next = being(0); next < being(NPROC); next++) {
        if (next->event != this)
            continue;

        if (CAS(&next->event, this, 0)) {
            next->merit = being(ONESELF)->name;
            next->state = READY;
            done += 1;
        }
    }

    return done;
}
```

- the scheduling loop is **bounded** by the number of process descriptors

Reference List I

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

[2] Herlihy, M. ; Luchangco, V. ; Moir, M. :
Obstruction-Free Synchronization: Double-Ended Queues as an Example.
In: *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS 2003), May 19–22, 2003, Providence, Rhode Island, USA*, IEEE Computer Society, 2003, S. 522–529

[3] Schröder-Preikschat, W. :
Non-Blocking Synchronisation.
In: Lehrstuhl Informatik 4 (Hrsg.): *Concurrent Systems*.
FAU Erlangen-Nürnberg, 2014 (Lecture Slides), Kapitel 11

LL/SC Shallows

- Alpha
    - if the effective addresses are not within the same naturally aligned 16-byte section, the sequence may fail or succeed
    - a reservation on a particular processor can be arbitrarily canceled by unspecified events on another processor
    - if any other memory access is executed on the given processor in between, the sequence may always fail on some implementations
    - a timer interrupt always cancels an existing reservation
- MIPS
    - a load, store, or prefetch event executed on the issuing processor may cause the sequence to fail or succeed
    - the largest cache line in use determines the reservation granularity
- PPC6
    - the reservation granularity is implementation-dependent
    - exceptions (traps/interrupts) hold up an existing reservation
    - a conditional store to a “scratch” address cancels the reservation
- PPC8
    - ditto, but the processor grants stores only to the reserved address

If a reservation endures exceptional situations and the processor does not compare with the reservation address, the operating system must cancel the reservation in those cases.

Process-Type Coercion

```
inline void *coerce(void *ptr, int val) {
  return (void *)((unsigned)ptr - val);
}

inline process_t *uncage(waitlist_h *list) {
  chain_t *item = purge_lfs((queue_t *)list);

  if (item)
    item = coerce(item, (int)&((process_t *)0)->event);

  return (process_t *)item;
}
```