Concurrent Systems

Nebenläufige Systeme

XIII. Progress Guarantee

Wolfgang Schröder-Preikschat

— for further study —
Agenda

Process Liveliness
Outline

Process Liveliness
necessary characteristics of a synchronisation protocol for a reference system of a specific number of concurrent processes:

i. the solution must be *symmetrical* between the processes; as a result we are not allowed to introduce a static priority

ii. nothing may be assumed about the *relative speeds* of the processes; we may not even assume their speeds to be constant in time

iii. if any of the processes is stopped well outside its critical section, this is not allowed to potential *blocking* of the others

iv. if more than one process is about to enter its critical section, it must be impossible to derive for them such finite speeds, that the decision to determine which one of them will enter its critical section first is postponed until *eternity*
necessary characteristics of a synchronisation protocol for a reference system of a specific number of concurrent processes:

i. the solution must be symmetrical between the processes; as a result we are not allowed to introduce a static priority

ii. nothing may be assumed about the relative speeds of the processes; we may not even assume their speeds to be constant in time

iii. if any of the processes is stopped well outside its critical section, this is not allowed to potential blocking of the others

iv. if more than one process is about to enter its critical section, it must be impossible to derive for them such finite speeds, that the decision to determine which one of them will enter its critical section first is postponed until eternity

although originally based on mutual exclusion of those processes, the characteristics hold for non-blocking synchronisation as well
characteristics of algorithms for non-blocking synchronisation
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 [4]
  - 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” [3, p.142]

- **wait-free**
  - if “any process can complete any operation in a finite number of steps, regardless of the relative speeds of the other processes” [3, p.7]
  - free of starvation of any conflicting process

Original, also referred to as “non-blocking.”
Liveness Properties

characteristics of algorithms for non-blocking synchronisation:

- **lock-free** if “some process will complete an operation in a finite number of steps, regardless of the relative speeds of the processes” [3, p. 142]
- free of starvation of at least one conflicting process

¹Originally, also referred to as “non-blocking”.
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 [4]

- **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” [3, p. 142]

- **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” [3, p. 7]
  - Free of starvation of any conflicting process

Originally, also referred to as “non-blocking”.

© wosch, thoenig CS (— for further study —) Process Liveliness
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 \[4\]
    - 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” \[3, p. 142\]^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” \[3, p. 7\]
    - 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**

---

^1Originally, also referred to as “non-blocking”.
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 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 A desirable property given priority-based process scheduling.
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

\(^2\)A desirable property given priority-based process scheduling.
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

\(^2\)A desirable property given priority-based process scheduling.
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 \textit{compute}
  - stands – lock-free if \textit{compute} is either lock- or wait-free
  - falls – off to obstruction-free if \textit{compute} is obstruction-free

- thus, the likewise weaker property of \textit{compute} comes out on top

- strengthening of the progress guarantee at a higher level is illusionary

\(^2\)A desirable property given priority-based process scheduling.
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. 6):
- SC ensures that one out of possibly many conflicting processes succeeds
- also, the likewise weaker property of *compute* comes out on top
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. 6):
- 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
at a first glance, the progress guarantee is the same as before (p. 6):
■ 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:
   • make a reservation for a hardware-dependent address range
   • 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. 13)
■ exceptions (i.e., traps/interrupts) may or may not cancel reservations
■ similar may hold for specific (“manually ejected”) memory operations
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)
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:

```c
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;
}
```
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
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 [5, p. 30]  
if succeeded, coerce the pointer to the purged chain item into the pointer to the enclosing process descriptor (cf. p. 14)
## Table-Based Scheduling II

```c
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 [5, p. 30]
  - if succeeded, coerce the pointer to the purged chain item into the pointer to the enclosing process descriptor (cf. p. 14)
  - even in case of a wait-free purge, the schedule loop is **unbounded**
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 [5, p. 30]
■ if succeeded, coerce the pointer to the purged chain item into the pointer to the enclosing process descriptor (cf. p. 14)
■ 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 the process blocks on the same event while signalling (cause)
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\textsuperscript{3} of the schedule loop is given only with a wait-free purge
in that case: \textit{WCET}(loop) \leq N \times \textit{WCET}(\text{purge}), thus bounded
Table-Based Scheduling III

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

    while ((done < N) && (next = uncage(&this->wait))) {
        next->merit = being(OSELF)->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: \( WCET(\text{loop}) \leq N \times WCET(\text{purge}) \), 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

\(^3\) worst-case execution time
Table-Based Scheduling III

Obstruction/Lock/Wait Freedom

```
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

\(^3\text{worst-case execution time}\)
each signaller takes the scheduling decision in a finite number of steps

```c
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 1


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

```c
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;
}
```