Concurrent Systems
Nebenläufige Systeme

VI. Locks

Wolfgang Schröder-Preikschat

December 11, 2019

Outline

Preface

Fundamentals
  Bifocal Perspective
  Basic Attributes

Avenues of Approach
  Atomic Memory Read/Write
  Specialised Instructions

Summary

Subject Matter

- discussion on **abstract concepts** as to blocking synchronisation:
  - **lock** a critical section
    - shut simultaneous processes out of entrance
    - block (delay) interacting processes
  - **unlock** a critical section
    - give a simultaneous process the chance of entrance
    - unblock one or several interacting processes
- treatment of basic characteristics and common variants of locking
  - hierarchic placement of lock/unlock implementations ~ ISA level
  - standby position, control mode, properties, computational burden
  - relying on atomic read/write, with and without special instructions
- explanation of benefits, limits, shallows, drawbacks, but also myths

**Spin-Lock (Ger. Umlaufsperre)**

Blocking synchronisation under prevention of context switches and by active waiting, including processor halt, for unlocking.
Purpose and Interpretation

Lockout [3, p.147]

A provision whereby two processes may negotiate access to common data is a necessary feature of an MCS.\(^a\)

\(^a\) abbr. multiprogrammed computer system

already this original reference foreshadows two levels of abstraction at which an implementation may be organisationally attached to:

- by means of a program at instruction set architecture level (i.e., level 2)
  - busy waiting until success of a TAS-like instruction [3, p. 147, Fig. 3a]
  - the TAS-like instruction—was and still—is an unprivileged operation
- by means of a program at operating system machine level (i.e., level 3)
  [To prevent hangup,] inhibit interruption of a process between execution of a lock and execution of the following unlock. [3, p. 147]
  - inhibit interruption beyond a hardware timeout is a privileged operation

note: (ii) takes a logical view as to hierarchic placement of lockout

Hierarchic Placement

Inhibit of Interruption/Preemption

in order that the mechanism is suited to pattern a hardware ELOP:\(^1\)

- lock
  - disables interrupts and acquires a (memory) bus lock
  - turns time monitoring on, i.e., arms some timeout mechanism
    - predefined worst-case execution time (WCET) or
    - upper limit of the number of processor instructions or cycles, resp.
  - raises an exception or issues an instruction trap [7] upon timeout
- unlock
  - turns time monitoring off
  - releases the (memory) bus lock and re-enables interrupts

for integrity reasons, the processor must enforce an absolute timeout

- the instruction-trap handler must be indispensable
  - a necessary part that needs to be provided by the operating system
- the lock/unlock pair does not have to be system calls to this end

- it must have some “use” [11] an operating system and
  - it may benefit from an operating system as to problem-specific timeouts
  - in which case the lock/unlock pair does have to be system calls, yet

\(^1\) As indicated by [3, p. 147], to prevent hangup of processes interrogating the lock indicator, and once supported by the Intel i860 [7, p. 7-24].

Indivisibility Revisited

critical section considered as logical or physical ELOP, referred to [3]

logical

- process lock, only
  - passage is vulnerable to delays
  - blocking time is two-dimensional
  - WCET\(^2\) of critical section and interrupt/preemption latency
  - hinders predictability
  - irrelevant for time-sharing mode
  - enables concurrent processes

physical

- interrupt and bus lock
  - passage is without delays
  - blocking time is one-dimensional
  - WCET\(^2\) of critical section
  - eases predictability
  - relevant for real-time mode
  - disables concurrent processes

\(^2\) abbr. worst-case execution time
Process Locks

Critical Section as ELOP in Logical Terms

Hint (Lockout)

Contemporary (real) processors do no longer offer a means to pattern a hardware ELOP. Instead, locking falls back on algorithmic solutions.

- the standby position of a process may be either active or passive
  - active: a spin-lock (Ger. Umlaufsperre), busy waiting
    - lock holder interruption/preemption is crucial to performance
    - periods out of processor increase latency for competing processes
    - extends the point in time until execution of unlock
  - passive: a sleeping lock (Ger. Schlafspere), idle waiting
    - lock/unlock entail system calls, thus are crucial to granularity
    - impact of system-call overhead depends on the critical sections
      - number, frequency of execution, and best-case execution time

- “passive waiting” for unlock is untypical for conventional locking
  - a sleeping lock typically falls back on a binary semaphore or mutex, resp.
  - a conventional lock manages on instruction set architecture level, only

3Operating system machine level concepts are discussed in LEC 7.

Coordinating Cooperation

- the control mode (Ger. Betriebsart, Prozessregelung) for a lockout may be either advisory or mandatory
  - advisory locking is explicit, performed by cooperating processes
    - first-class object of the real processor, e.g. a critical section
    - assumes process-conformal protocol behaviour
    - a lock action must be followed by an unlock action
    - complies with a lower level of abstraction
  - mandatory locking is implicit, as a side effect of a complex operation
    - first-class object of an operating system, e.g. a file
    - enables recognition of exceptional conditions
    - “extrinsic” access on a locked file by a simultaneous process
    - calls for a higher level of abstraction

- mandatory locks are implemented using advisory locks internally
  - the exception proves the rule...

Hint

Advisory locks are in the foreground of this lecture, mandatory locks (in its classical meaning) will not be covered.

Lock Characteristics

- the computational burden of synchronisation in general and locking in specific is ambilateral and applies particularly to:
  - overhead
    - as to the computing resources demands of a single lock:
      - memory footprint (code, data) of a lock data type instance
      - needs to allocate, initialise, and destroy those instances
      - time and energy needed to acquire and release a lock
    - increases with the number of locks per (non-seq.) program
  - contention
    - as to the competitive situation of interacting processes
      - on the one hand, running the entry protocol
      - on the other hand, running the critical section
    - increases with the number of interacting processes

- both factors affect the granularity of the object (data structure or critical section, resp.) to be protected
  - the more coarse-grained the object, the lower overhead/higher contention
    - scarcely audible background noise v. higher probability of interference
    - the more fine-grained the object, the higher overhead/lower contention
    - easily audible background noise v. lower probability of interference
  - striking a balance between the two—if at all sensible—is challenging
Solutions Devoid of Dedicated Processor Instructions

■ sole demand is the atomic read/write of one machine word from/to main memory by the real processor
■ classical approaches are in the foreground
  - for \( N = 2 \) processes: Dekker (1965), Peterson (1981), and Kessels (1982)
  - more of Lamport (1974) and Peterson (1981) for \( N > 2 \) in the addendum
■ all of them are more than an exercise to read, but significant even today
  - some are confined to two contenting processes, ideal for dual-core processors
  - others are computationally complex, but may result only in background noise
■ they demonstrate what “coordination of cooperation” in detail means\(^4\)

■ an additional and utmost important constraint of these approaches is related to the memory model of the real processor
■ for sequential consistent memory only, less important in olden days
■ but more recent, this changed dramatically and gives one a hard time
■ mean to say: solutions for synchronisation that do not use specialised processor instructions are not necessarily portable!

\(^4\)The “state machine” approach will be picked up again later for non-blocking synchronisation (LEC 10), e.g. of a semaphore implementation (LEC 11).

Lock Type I

Algorithms of Dekker, Peterson, and Kessels

```c
#ifndef NPROC
#define NPROC 2
#endif

#ifdef __FAME_LOCK_KESSEL__
#define NTURN NPROC
#else
#define NTURN NPROC - 1
#endif

typedef volatile struct
    lock {
        bool want[ NPROC ]; /* initial: all false */
        char turn[ NTURN ]; /* initial: all 0 */
    } lock_t;

inline unsigned earmark () {
    return /* hash of process ID for [0, NPROC - 1] */
}
```

Memory Barriers/Fences

Beware of dynamic ordering of read/write operations.
Kessels' Algorithm for $N = 2$  

**egoistic** ("self-serving") entry protocol with **no-passing zone:**

```c
void lock(lock_t *bolt) {
    unsigned self = earmark(); /* my process index */
    bolt->want[self] = true; /* I am interested */
    bolt->turn[self] = false; /* I withdraw */
    bolt->want[self] = true; /* I am interested */
    bolt->turn[self] = false; /* I defer to you */
    if (bolt->turn[0] != self)
        /* & will wait */
    if (bolt->turn[0] != self)
        /* & reconsider */
    bolt->want[self] = false; /* & like to be next */
    bolt->turn[self] = (bolt->turn[self-1] + self) % 2;
    while (bolt->turn[self-1] && (bolt->turn[0] == self))
        /* & inside CS */
    void unlock(lock_t *bolt) {
        unsigned self = earmark(); /* my process index */
        bolt->want[self] = false; /* I am uninterested */
    }
}
```

For an interpretation, see also p. 39.

---

Dekker's Algorithm for $N = 2$  

**altruistic** ("self-forgetting") entry protocol with **passing zone:**

```c
void lock(lock_t *bolt) {
    unsigned self = earmark(); /* my process index */
    bolt->want[self] = true; /* I am interested */
    bolt->turn[self] = false; /* you are interested */
    if (bolt->turn[0] != self) {
        /* & inside CS */
    bolt->want[self] = false; /* I withdraw */
    while (bolt->turn[0] != self); /* & will wait */
    bolt->want[self] = true; /* & reconsider */
    bolt->want[self] = false; /* & like to be next */
    bolt->turn[0] = self; /* you are interested */
    void unlock(lock_t *bolt) {
        unsigned self = earmark(); /* my process index */
        bolt->want[self] = true; /* I am interested */
        bolt->turn[0] = self; /* & inside CS */
        bolt->want[self] = false; /* I am uninterested */
    }
}
```

---

**Peterson's Algorithm for $N = 2$**

**egocentric** ("self-serving") entry protocol with **no-passing zone:**

```c
void lock(lock_t *bolt) {
    unsigned self = earmark(); /* my process index */
    bolt->want[self] = true; /* I am interested */
    bolt->turn[0] = self; /* & like to be next */
    while (bolt->turn[self-1] && (bolt->turn[0] == self))
        /* & inside CS */
    void unlock(lock_t *bolt) {
        unsigned self = earmark(); /* my process index */
        bolt->want[self] = false; /* I am uninterested */
    }
}
```

---

**Starvation Freedom**

**Hint (Progress)**

A matter of interaction of processes by means of the entry and exit protocols, while abstracting away from potential delays caused by "external incidents" of the instruction set architecture (ISA) level.

- in terms of the lock **callee** process: "bottom up" point of view of the level of abstraction of the entry protocol
  - the entry or exit, resp., protocol is shaped up as a logical ELOP (cf. p. 8)
  - depending on the solution, process delays are "accessory symptom" of:
    - Dekker
      - noncritical parts of the entry protocol ($want_i = false$)
      - all the critical section ($want_i = true$)
  - in terms of the lock **caller** process: "top down" point of view of the level of abstraction of the critical section
    - the entry or exit, resp., protocol appears to be instantaneous

---

As if it is implemented as a physical ELOP (cf. p. 8).
Spin-Lock

(Ger.) Umlaufsperre

void lock(lock_t *bolt) {
    bool busy;
    do atomic {
        if (!( busy = bolt->busy )); /* check/try lock */
        bolt->busy = true; /* acquire lock */
    } while (busy); /* if applicable, retry sequence */
}

checking/trying and, if applicable, then acquiring the lock need to be an atomic action because:
5–6 ■ assuming that these actions are due to simultaneous processes
5 ■ all these processes might find the door to the critical section open
6 ■ all of those processes who found the door open will lock the door
7 ■ all of those who locked the door will enter the critical section
6–7 ■ multiple processes may be in the critical section, simultaneously
■ ensuring the mutual exclusion property requires a hardware ELOP that allows for to resemble the atomic construct

Lock Type II

in its simplest form, a binary variable indicating the lock status:

```c
typedef volatile struct lock {
    bool busy; /* initial: false */
} lock_t;
```

true ■ occupied critical section, processes seeking entry will block
false ■ unoccupied critical section, unblocked processes retries to enter

just as simple the exit protocol for a number of lock variants

```c
void unlock(lock_t *bolt) {
    bolt->busy = false; /* release lock */
}
```

more distinct is variant diversity of the entry protocol (p. 23 ff.)...
Backoff

Definition

Static or dynamic holding time, stepped on a per-process(or) basis, that must elapse until resumption of a formerly contentious action.

- originally from telecommunications to facilitate congestion control (Ger. Blockierungskontrolle) by avoiding channel oversubscription
- statically (ALOHA [1]) or dynamically (Ethernet [10]) assigned delays
- practised at broadcasting/sending time or to resolve contention, resp.
- adopted for parallel computing systems to reduce the probability of contention in case of conflicting accesses to shared resources
- common are dynamic approaches: exponential and proportional backoff

Interference with Scheduling: Priority Violation/Inversion etc.

Allocation of stepped holding times on a per-process basis rivals with planning decisions of the process scheduler.

Spin with CAS  
cf. p. 45

```
void lock(lock_t *bolt) {
    while (!CAS(&bolt->busy, false, true));
}
```

- overcomes the problem of an “unconditional store”-prone TAS
- the cache protocol runs write invalidate or update, resp., conditionally
- but the problem of access contention at the shared bus remains
- the processor is instructed to repeatedly run atomic “read-modify-write” cycles with only very short periods of leaving the bus unlocked
- all sorts of simultaneous processes will have to suffer for bandwidth loss
- in non-functional terms, a solution that scales bad...

Spin on Read

```
void lock(lock_t *bolt) {
    do {
        while (bolt->busy);
    } while (!CAS(&bolt->busy, false, true));
}
```

- attenuates the problem of bus access contention and interference
- the actual wait loop proceeds with a full-time unlocked bus
- unrelated simultaneous (i.e., concurrent) processes are not affected
- the lock is acquired at a time of a probably deserted critical section
- related simultaneous (i.e., interacting) processes are affected, only
- suffers from regular (constant) non-sequential programs or processes
- such as single program, multiple data (SPMD, [2]), a programming model of parallel computing with tendency to common mode (Ger. Gleichtakt)
- in such a case, “clustered” processes behave and operate almost identical and, thus, will intermittently create a storm of bus lock bursts
- in non-functional terms, a solution that scales in a lesser extent...

Footnote:

8Note that the spinning processes may have been passed by a process.

Lock Type III and IV

- for possibly lock-specific static/exponential backoff:
  - extended by a pointer to an open array of backoff values
  - typically, the array size complies with the number of processors
  ```
typedef volatile struct lock {
    bool busy; /* initial: false */
    long (*rest)[]; /* initial: null */
} lock_t;
```
  - for lock-specific proportional backoff: ticket-based
    - not dissimilar to a wait ticket dispenser (Ger. Wartemarkenspender) for a passenger paging system (Ger. Personenaufrufanlage)
    ```
typedef volatile struct lock {
    long next; /* number being served next */
    long this; /* number being currently served */
} lock_t;
```
Spin with Backoff I

- principle is to **pause** execution **after a collision** has been detected:
  - attenuate lock contention amongst known “wranglers” for the next trial

```c
void lock(lock_t *bolt) {
    while (!CAS(&bolt->busy, false, true))
        backoff(bolt, 1);
}
```

- combined with “**spin on read**” before (re-) sampling the lock flag:
  - combat lock contention for the next trial by assuming that “wranglers” could be overtaken by another simultaneous process

```c
void lock(lock_t *bolt) {
    do {
        while (bolt->busy);
        if (CAS(&bolt->busy, false, true)) break;
        backoff(bolt, 1);
    } while (true);
}
```

### Backoff Procedure

- **busy waiting** in pure form
  - **volatile** forces the compiler not to clean out the count down loop

```c
long rest(volatile long term) {
    while (term--); /* let the holding time pass */
    return term;
}
```

- **volatile** forces the compiler not to clean out the count down loop

```c
# include "lock.h"
# include "earmark.h"

void backoff(lock_t *bolt, int hold) {
    if (bolt->rest)
        rest((bolt->rest)[earmark()] * hold);
}
```

- in **privileged mode** and if applicable a **halt** instruction is preferred
  - in that case, the actual parameter of **rest** defines a **hardware timeout**
  - that is to say, a timer interrupt is used to force the processor out of **halt**

Spin with Backoff II

- rely on **feedback** to decrease the rate of simultaneous processes:
  - gradual doubling of the per-process holding time when allocation failed
  - increasing lock-retry timeout with “ceiling value” (most significant bit)

```c
void lock(lock_t *bolt) {
    int hold = 1;
    do {
        while (bolt->busy);
        if (CAS(&bolt->busy, false, true)) break;
        backoff(bolt, hold);
        if ((hold << 1) != 0) hold <<= 1;
    } while (true);
}
```

### Proportional Backoff

- note that **self – this** gives the number of waiting processes that will be served first in order to run the critical section

```c
void lock(lock_t *bolt, long cset) {
    long self = FAA(&bolt->next, 1);
    if (self != bolt->this) {
        rest((self - bolt->this) * cset);
        while (self < bolt->this);
    }
}
```

Spin with Ticket

- **volatile** forces the compiler not to clean out the count down loop

```c
long rest(volatile long term) {
    while (term--); /* let the holding time pass */
    return term;
}
```

- in privileged mode and if applicable a halt instruction is preferred
  - in that case, the actual parameter of rest defines a hardware timeout

```
© wosch, thoenig  CS (WS 2019/20, LEC 6)  Avenues of Approach–Specialised Instructions 29
```

© wosch, thoenig  CS (WS 2019/20, LEC 6)  Avenues of Approach–Specialised Instructions 30

© wosch, thoenig  CS (WS 2019/20, LEC 6)  Avenues of Approach–Specialised Instructions 31

© wosch, thoenig  CS (WS 2019/20, LEC 6)  Avenues of Approach–Specialised Instructions 32
Résumé

- conventional locking under prevention of context switches
- hierarchic placement of lock/unlock implementations → ISA level
- standby position, control mode, properties, computational burden
- approaches with atomic read/write or added specialised instructions
  - algorithms falling back on TAS, CAS, FAA, and backoff procedures
- although simple in structure, potential deleterious cache effects

**Critical Section Execution Time (CSET)**

That locks are suitable for a short CSET is computer-science folklore, but by far too flat. Much more important is to have a **bounded** and, even better, **constant** CSET. Above all, this makes high demands on the design of critical sections and non-sequential programs.
let self be the current process, peer be the counterpart, and bolt be the lock variable used to protect some critical section CS

a first glance at the entry protocol reveals:

4. self shows interest in entering CS, maybe simultaneously to peer’s intend to enter the same CS as well

5–9. if applicable, self hence waits on peer to yield CS and appoint self being candidate to run CS next

upon a closer look, the entry protocol takes care of the following:

5–6. as the case my be, self contends with peer for entrance but retries if it should be self’s turn to enter

7–8. in that case, while preventing potential deadlock of the processes, self waits on peer for being appointed to enter CS

9. reconsider entering of the critical section...

11Imagine, line 7 would have been considered redundant and, thus, omitted.
**Peterson’s Solution for $N > 2$**

cf. [6] or [12], resp.

```c
void lock(lock_t *lock) {
    unsigned rank, next, self = earmark();
    for (rank = 0; rank < NPROC - 1; rank++) {
        lock->want[self] = rank;
        lock->turn[rank] = self;
    }
    for (next = 0; next < NPROC; next++)
        if (next != self)
            while ((lock->want[next] >= rank) && (lock->turn[rank] == self));
}

void unlock(lock_t *lock) {
    unsigned self = earmark();
    lock->want[self] = -1;
}
```

**Interpretation**

**Hint**

Every process must have proved oneself for $n−1$ ranks to be eligible for entering the critical section.

- basic idea is to apply the two-process solution at each rank repeatedly
  - at least one process is eliminated, stepwise, until only one remains
  - let want[$p$] be the rank of process $p$, let turn[$r$] be the process that entered rank $r$ last, and let CS be a critical section:
    - 5-6 in attempting to enter CS, indicate interest to reach the next rank
    - 8-9 for it, check all other processes for their particular rank and
    - 10-11 busy wait if there are still higher ranked processes and the current process is still designed to be promoted

- often also labelled as filter or tournament algorithm:
  - deters one out of $N$ simultaneous processes from entering CS
  - repeated for $N−1$ times, only one process will be granted access finally

---

**Lamport’s Bakery Algorithm I**

Lock Type and Ticket Dispenser

```c
#include <stdbool.h>
typedef volatile struct lock {
    bool want[NPROC]; /* initial: all false */
    long turn[NPROC]; /* initial: all 0 */
} lock_t;

entry protocol patterns a “take a number” system: a.k.a. ticket lock

inline void ticketing(lock_t *bolt, unsigned slot) {
    unsigned next, high = 0;
    bolt->want[slot] = true; /* enter choosing */
    for (next = 0; next < NPROC; next++)
        if (bolt->turn[next] > high)
            high = bolt->turn[next];
    bolt->turn[slot] = high + 1; /* state number */
    bolt->want[slot] = false; /* leave choosing */
}

void lock(lock_t *bolt) {
    unsigned next, self = earmark();
    ticketing(bolt, self); /* take a number */
    for (next = 0; next < NPROC; next++)
        while (bolt->turn[next] != 0)
            while ((bolt->want[next]) && (bolt->turn[next] < bolt->turn[self])
                || ((bolt->turn[next] == bolt->turn[self])
                && (next < self))); /* next first */
}

void unlock(lock_t *bolt) {
    unsigned self = earmark();
    bolt->turn[self] = 0;
}
```

---

**Lamport’s Bakery Algorithm II**

cf. [9]

```c
#include <stdbool.h>
typedef volatile struct lock {
    bool want[NPROC]; /* initial: all false */
    long turn[NPROC]; /* initial: all 0 */
} lock_t;

entry protocol patterns a “take a number” system: a.k.a. ticket lock

inline void ticketing(lock_t *bolt, unsigned slot) {
    unsigned next, high = 0;
    bolt->want[slot] = true; /* enter choosing */
    for (next = 0; next < NPROC; next++)
        if (bolt->turn[next] > high)
            high = bolt->turn[next];
    bolt->turn[slot] = high + 1; /* state number */
    bolt->want[slot] = false; /* leave choosing */
}

void lock(lock_t *bolt) {
    unsigned next, self = earmark();
    ticketing(bolt, self); /* take a number */
    for (next = 0; next < NPROC; next++)
        while (bolt->turn[next] != 0)
            while ((bolt->want[next]) && (bolt->turn[next] < bolt->turn[self])
                || ((bolt->turn[next] == bolt->turn[self])
                && (next < self))); /* next first */
}

void unlock(lock_t *bolt) {
    unsigned self = earmark();
    bolt->turn[self] = 0;
}
```
number of “busy wait” loop actions with bus locked and unlocked:

```
# lock:  
movl 4(%esp), %eax
LBB0_1:
  movb $1, %cl
  xchgb %cl, (%eax)
  testb $1, %cl
  je LBB0_1
  ret

# lock:  
movl 4(%esp), %ecx
movb $1, %dl
LBB0_1:
  xorl %eax, %eax
  lock
  cmpxchgb %dl, (%ecx)
  testb %al, %al
  jne LBB0_1
  ret
```

- 1:3
- line (5) v. lines (4, 6, 7)
- lines (14, 15) v. lines (13, 16, 17)

in case of x86, there is no difference as to the number of actions
but there is still the difference as to the frequency of cache interference
the ratio depends on the code generator (compiler) and the CPU