Concurrent Systems

Nebenläufige Systeme

VI. Locks

Wolfgang Schröder-Preikschat

December 10, 2020

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 \(\sim\) ISA level
  - standby position, control mode, properties, computational burden
  - relying on atomic read/write, with and without special instructions

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.\(^1\)

\(^1\)abbr. multiprogrammed computer system

- already this original reference foreshadows two levels of abstraction at which an implementation may be organisationally attached to:
  - i 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
  - ii 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 must be unmaskable at the level of lock/unlock
  - 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 has to have “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

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

\(^1\)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].

\(^2\)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. *Schlafsperrre*), 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.¹
  - a conventional lock manages on instruction set architecture level, only

¹Operating system machine level concepts are discussed in LEC 7.

Coordinating Cooperation

- enforcement of **sequential execution** of any critical section always goes according to one and the same pattern:
  - **entry protocol**: acquire exclusive right to run through the critical section
  - refuse other processes entrance to the critical section
  - as a function of the *lock* operation
  - **exit protocol**: release exclusive right to run through the critical section
  - provide a process entrance to the critical section
  - as a function of the *unlock* operation

- including the assurance of fundamental **mandatory properties**:
  - **mutual exclusion**: at any point in time, at most one process may “have a command of” (Ger. *beherrschen*) the critical section
  - **deadlock freedom**: if several processes simultaneously aim for entering the critical section, one of them will eventually succeed
  - **starvation freedom**: if a process aims for entering the critical section, it will eventually succeed
  - not least, **desirable property** is to not interfere with the scheduler

Lock Characteristics

- 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.

Working Resistance

- 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
  - 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/lower 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).

⚠️ altruistic (“self-forgetting”) entry protocol with passsing zone.\(^5\)

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

void unlock(lock_t *bolt) {
    unsigned self = earmark(); /* my process index */
    bolt->turn[0] = self^1; /* I defer to you */
    bolt->want[self] = false; /* I am uninterested */
}
```

\(^5\)For an interpretation, see also p. 39.

---

Peterson’s Algorithm for $N = 2$ cf. [12]

⚠️ egoistic (“self-serving”) entry protocol with no-passsing zone.\(^6\)

```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->want[0] != self); /* you are interested */
    & (bolt->turn[0] == self); /* & inside CS */
}

void unlock(lock_t *bolt) {
    unsigned self = earmark(); /* my process index */
    bolt->want[self] = false; /* I am uninterested */
}
```

\(^6\)Example for the C version is the original document [12]. See also p. 40.

---

Kessels’ Algorithm for $N = 2$ cf. [8]

⚠️ refinement of Peterson’s solution, but a mutable entry protocol:
- as far as the commitment on the next process is concerned

```c
#define __FAME_LOCK_KESSEL__
...
void lock(lock_t *bolt) {
    unsigned self = earmark(); /* my process index */
    bolt->want[self] = true; /* I am interested */
    bolt->turn[self] = ((bolt->turn[self^1] + self) % 2);
    while (bolt->want[self^1] &&
    (bolt->turn[self] == ((bolt->turn[self^1]+self)%2)));
}
```

4–7 □ compared to the entry protocol of Dekker’s algorithm, the interest in entering the critical section (l. 4) never disappears

---

Starvation Freedom Question of Interpretation (cf. p. 11)

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, = false)
    - all the critical section (want, = 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\(^7\)

\(^7\)As if it is implemented as a physical ELOP (cf. p. 8).
unoccupied critical section, unblocked processes retries to enter
occupied critical section, processes seeking entry will block
all these processes might find the door to the critical section open
all of those processes who found the door open will lock the door
all of those who locked the door will enter the critical section
assuming that these actions are due to
fundamental aspect common to all the solutions discussed before:
processes rely on plain—but atomic—read/write operations, only
there is no read-modify-write cycle w.r.t. the same shared variable
as a consequence, arbitration at ISA level is less overhead-prone
solutions for \( N = 2 \) are "simple", compared to \( N > 2 \) (cf. p. 41ff.)
solutions for \( N > 2 \) processes benefit from special CPU instructions
atomic read-modify-write instructions such as TAS, CAS, or FAA
but also load/store instructions that can be interlinked such as LL/SC
not only the memory model but in particular the caching behaviour
of the real processor have a big impact on the solutions
most of the special instructions are considered harmful for data caches
unet use breeds interference with all sorts of simultaneous processes
in case of high contention, this unwanted property is even more critical
mean to say: solutions for synchronisation making use of specialised processor instructions are not necessarily straightforward!

Spin–Lock (Ger.) Umlaufsperre

```c
#include <stdbool.h>
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

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

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

Spin with TAS cf. p. 45

```c
#include <atomic.h>
atomic word TAS(word *ref) { word aux = *ref; *ref = 1; return aux; }

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

atomic word TAS(word *ref) { word aux = *ref; *ref = 1; return aux; }

be aware of the conventional implementation of TAS [13, p.10 & 35]:
the unconditional store has a deleterious effect for the cache
as to the cache operation (write invalidate or update, resp.), the cache
blocks other processors from using the shared bus to access memory or other devices that are attached to access contention
thereby interfering in particular with processes that are unrelated to the spinning process, thus constraining concurrency
in non-functional terms, a solution that scales baddish...
```

fundamental aspect common to all the solutions discussed before:
atomic read-modify-write instructions such as TAS, CAS, or FAA
but also load/store instructions that can be interlinked such as LL/SC
not only the memory model but in particular the caching behaviour
of the real processor have a big impact on the solutions
most of the special instructions are considered harmful for data caches
unet use breeds interference with all sorts of simultaneous processes
in case of high contention, this unwanted property is even more critical
mean to say: solutions for synchronisation making use of specialised processor instructions are not necessarily straightforward!
Backoff
Avoidance of Bus Lock Bursts

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.

8Note that in interference-prone environments of unknown frequency, periods, and lengths of delays it is hardly feasible to prevent lock contention.

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

Critical Section Execution Time (CSET)
Risk of degeneration to spin on CAS if the CSET is too short and, thus, the cycle time of the entry/exit protocol possibly becomes shorter than the start-up time of the CPU for the next cycle within the cache (line 3): in the case of an x86, e.g., a handful (2–6) of processor instructions.

Spin with CAS
1 void lock(lock_t *bolt) {
2 while (!CAS(&bolt->busy, false, true));
3 }

Spin on Read
1 void lock(lock_t *bolt) {
2 do {
3 while (bolt->busy);
4 } while (!CAS(&bolt->busy, false, true));
5 }

Note that the spinning processes may have been passed by a process.
Spin with Backoff I

**Static Backoff**

**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

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

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

**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;
}
```

- **in privileged mode** and if applicable a **halt** instruction is preferred
- that is to say, a timer interrupt is used to force the processor out of **halt**

Spin with Backoff II

**Truncated Exponential Backoff**

- 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);
}
```

in non-functional terms, solutions that scale to some extent...
- including the solutions of static backoff as shown before

Spin with Ticket

**Proportional Backoff**

```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);
    }
}
```

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

- knowing the **critical section execution time** (CSET) would be great
  - a choice of best-, average-, or worst-case execution time (B/A/WCET)
  - depends on the structure of critical sections as well as "background noise"
Résumé

- conventional locking under prevention of context switches
- hierarchical 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
  - lock contention when processes try to acquire a lock simultaneously
  - bus lock bursts when processes run the entry protocol in common mode

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.

Reference List I


Reference List II


Original Dekker’s Algorithm for $N = 2$ Interpretation

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:

1. \( self \) shows interest in entering \( CS \), maybe simultaneously to \( peer \)'s intend to enter the same \( CS \) as well
2. if applicable, \( self \) hence waits on \( peer \) to yield \( CS \) and appoint \( self \) being candidate to run \( CS \) next
3. upon a closer look, the entry protocol takes care of the following:
   - as the case my be, \( self \) contends with \( peer \) for entrance but retries if it should be \( self \)'s turn to enter
   - in that case, while preventing potential deadlock\(^{11} \) of the processes, \( self \) waits on \( peer \) for being appointed to enter \( CS \)
   - reconsider entering of the critical section... 

\(^{11}\)Imagine, line 7 would have been considered redundant and, thus, omitted.

Peterson’s Solution for $N = 2$: Transformation

the construct of the busy wait loop in the entry protocol originally described in [12] is to be read as follows:

\[
\begin{align*}
wait \text{ until condition} & = repeat \text{ nothing until condition} \\
& = do \text{ nothing while } \neg \text{condition} \\
& \text{applied to } C = \text{ while } (\neg \text{condition}); \\
& \text{with condition} = \neg Q_i \text{ or turn } = i \\
& \text{inserted and factored out} = \text{ while } (\neg(\neg Q_i \text{ or turn } = i)); \\
& = \text{ while } (Q_i \text{ and turn } \neq i); \\
& = \text{ while } (Q_i \text{ and turn } = j); \\
& \text{with } j \neq i
\end{align*}
\]

this results in a code structure of the entry protocol that is different from the many examples as can be found in the Web

---


Peterson’s Solution for $N > 2$

`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;
} `
number of “busy wait” loop actions with bus locked and unlocked:

```assembly
1  lock:
2  movl 4(%esp), %eax
3  LBB0_1:
4  movb $1, %cl
5  xchgb %cl, (%eax)
6  testb $1, %cl
7  je  LBB0_1
8  ret
```

```assembly
9  lock:
10 movl 4(%esp), %ecx
11 movb $1, %dl
12 LBB0_1:
13 xorl %eax, %eax
14 lock
15 cmpxchgb %dl, (%ecx)
16 testb %al, %al
17 jne  LBB0_1
18 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