

# Concurrent Systems

*Nebenläufige Systeme*

## VI. Locks

Wolfgang Schröder-Preikschat

November 13, 2014



## Outline

Preface

Fundamentals

Bifocal Perspective  
Basic Attributes

Avenues of Approach

Atomic Memory Read/Write  
Specialised Instructions

Summary



## Agenda

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  $\leadsto$  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. *Umlaufsperr*)

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



Preface

## Fundamentals

- Bifocal Perspective
- Basic Attributes

## Avenues of Approach

- Atomic Memory Read/Write
- Specialised Instructions

Summary



## Lockout [3, p. 147]

*A provision whereby two processes may negotiate access to common data is a necessary feature of an MCS.<sup>a</sup>*

<sup>a</sup>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**:<sup>1</sup>
  - 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 does have to “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

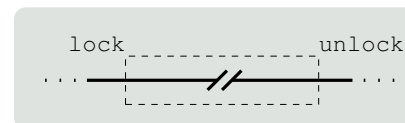
<sup>1</sup>As indicated by [3, p. 147], to prevent hangup of processes interrogating the lock indicator, and once supported by the Intel i860 [7, p. 7-24].



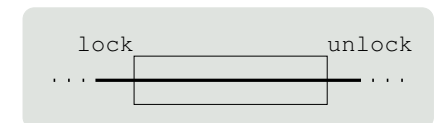
## Indivisibility Revisited

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

### logical



### physical



- |  |  |
|--|--|
| <ul style="list-style-type: none"> <li>■ process lock, only                             <ul style="list-style-type: none"> <li>- passage is vulnerable to delays</li> </ul> </li> <li>■ blocking time is two-dimensional                             <ul style="list-style-type: none"> <li>- WCET<sup>2</sup> of critical section <i>and</i></li> <li>- interrupt/preemption latency</li> </ul> </li> <li>■ hinders predictability                             <ul style="list-style-type: none"> <li>- irrelevant for time-sharing mode</li> </ul> </li> <li>■ enables concurrent processes</li> </ul> | <ul style="list-style-type: none"> <li>■ interrupt <i>and</i> bus lock                             <ul style="list-style-type: none"> <li>- passage is without delays</li> </ul> </li> <li>■ blocking time is one-dimensional                             <ul style="list-style-type: none"> <li>- WCET<sup>2</sup> of critical section</li> </ul> </li> <li>■ eases predictability                             <ul style="list-style-type: none"> <li>- relevant for real-time mode</li> </ul> </li> <li>■ disables concurrent processes</li> </ul> |
|--|--|

<sup>2</sup>abbr. *worst-case execution time*



## 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. *Umlaufsperr*), 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. *Schlafsperr*), 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.<sup>3</sup>
    - a conventional lock manages on instruction set architecture level, only

<sup>3</sup>Operating 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



- 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

(Ger.) Bürde

- 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 (nonseq.) 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



Preface

Fundamentals

Bifocal Perspective  
Basic Attributes

Avenues of Approach

Atomic Memory Read/Write  
Specialised Instructions

Summary



- 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<sup>4</sup>
- 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!

<sup>4</sup>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 Kessel

```

1 #ifndef NPROC
2 #define NPROC 2
3 #endif
4
5 #ifdef __FAME_LOCK_KESSEL__
6 #define NTURN NPROC
7 #else
8 #define NTURN NPROC - 1
9 #endif
10
11 typedef volatile struct lock {
12     bool want[NPROC];        /* initial: all false */
13     char turn[NTURN];       /* initial: all 0 */
14 } lock_t;
15
16 inline unsigned earmark() {
17     return /* hash of process ID for [0, NPROC - 1] */
18 }
```

Memory Barriers/Fences

Beware of **dynamic ordering** of read/write operations.



## Dekker's Algorithm for $N = 2$

cf. [4] or p. 37, resp.

- **altruistic** (“self-forgetting”) entry protocol with **passing zone**:<sup>5</sup>

```

1 void lock(lock_t *bolt) {
2     unsigned self = earmark(); /* my process index */
3
4     bolt->want[self] = true;    /* I am interested */
5     while (bolt->want[self^1] /* you are interested */
6           && (bolt->turn[0] != self)) { /* & inside CS */
7         bolt->want[self] = false; /* I withdraw */
8         while (bolt->turn[0] != self); /* & will wait */
9         bolt->want[self] = true;  /* & reconsider */
10    }
11 }
12
13 void unlock(lock_t *bolt) {
14     unsigned self = earmark(); /* my process index */
15     bolt->turn[0] = self^1;    /* I defer to you */
16     bolt->want[self] = false; /* I am uninterested */
17 }
```

<sup>5</sup>For an interpretation, see also p. 38.



- **egoistic** (“self-serving”) entry protocol with **no-passing zone**.<sup>6</sup>

```

1 void lock(lock_t *bolt) {
2     unsigned self = earmark(); /* my process index */
3
4     bolt->want[self] = true;    /* I am interested */
5     bolt->turn[0] = self;      /* & like to be next */
6     while (bolt->want[self^1] /* you are interested */
7           && (bolt->turn[0] == self)); /* & inside CS */
8 }
9
10 void unlock(lock_t *bolt) {
11     unsigned self = earmark(); /* my process index */
12     bolt->want[self] = false; /* I am uninterested */
13 }

```

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

<sup>6</sup>Example for the C version is the original document [12]. See also p. 39.

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

```

1 #define __FAME_LOCK_KESSEL__
2 ...
3 void lock(lock_t *bolt) {
4     unsigned self = earmark(); /* my process index */
5
6     bolt->want[self] = true;    /* I am interested */
7     bolt->turn[self] = ((bolt->turn[self^1] + self) % 2);
8     while (bolt->want[self^1] &&
9           (bolt->turn[self] == ((bolt->turn[self^1]+self)%2)));
10 }

```

- 7 ■ who's next uses feedback as to peer's view on who's turn was last  
 9 ■ in case of lock contention, gives only a single process precedence

- essential difference is the **single-writer** approach:
  - that is, the entry protocol constrains processes to **read-only sharing**
  - each process will only write to own variables, but may read all variables

## 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 ( $want_i = false$ )
    - 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**<sup>7</sup>

<sup>7</sup>As if it is implemented as a **physical ELOP** (cf. p. 8).

## Solutions Based on Dedicated Processor Instructions

- 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. 40ff.)
- 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
  - unopt 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!

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

```

1 #include <stdbool.h>
2
3 typedef volatile struct lock {
4     bool busy;      /* initial: false */
5 } lock_t;

```

- true**
  - occupied critical section, processes seeking entry will block
  - blocking is implemented solely by means of the ISA level
- false**
  - unoccupied critical section, unblocked processes will retry to enter

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

```

1 void unlock(lock_t *bolt) {
2     bolt->busy = false;    /* release lock */
3 }

```

- more distinct is variant diversity of the **entry protocol** (p. 22 ff.)...



```

1 void lock(lock_t *bolt) {
2     bool busy;
3
4     do atomic {
5         if (!(busy = bolt->busy)) /* check/try lock */
6             bolt->busy = true;    /* acquire lock */
7     } while (busy); /* if applicable, retry sequence */
8 }

```

- 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
- ↪ 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



## Spin with TAS

cf. p. 44

```

1 void lock(lock_t *bolt) {
2     while (!TAS(&bolt->busy)); /* loop if door closed */
3 }

```

- be aware of the conventional implementation of TAS [13, p. 10 & 35]:

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

- the unconditional store has a **deleterious effect** for the cache
- as to the cache operation (write invalidate or update, resp.), the cache line holding the main memory operand causes high bus traffic
- for  $N$  contending processes, either  $N - 1$  cache misses or update requests
- further problem dimension is non-stop instruction of TAS in the loop
  - blocks other processors from using the **shared bus** to access memory or other devices that are attached to  $\leadsto$  **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...



## Spin with CAS

cf. p. 44

```

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

```

- overcomes the problem of an “unconditional store”-prone TAS

$$\text{CAS} = \begin{cases} \text{true} \rightarrow \text{stored true into busy,} & \text{if busy = false} \\ \text{false,} & \text{otherwise} \end{cases}$$

- 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...



```

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

```

Risk of degeneration to *spin on CAS* if the CSEC 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.

- attenuates the problem of bus access contention and interference
  - 3 ■ the actual wait loop proceeds with a full-time unlocked bus
    - unrelated simultaneous (i.e., concurrent) processes are not affected
  - 4 ■ the lock is acquired at a time of a probably<sup>8</sup> 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...

<sup>8</sup>Note 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
- ```

1 typedef volatile struct lock {
2     bool busy;          /* initial: false */
3     long (*rest)[];    /* initial: null */
4 } 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*)

```

1 typedef volatile struct lock {
2     long next; /* number being served next */
3     long this; /* number being currently served */
4 } lock_t;

```

### 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<sup>9</sup> 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.

<sup>9</sup>Note that in interference-prone environments of unknown frequency, periods, and lengths of delays it is hardly feasible to prevent lock contention.

## 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

```

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

```

- 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

```

1 void lock(lock_t *bolt) {
2     do {
3         while (bolt->busy);
4         if (CAS(&bolt->busy, false, true)) break;
5         backoff(bolt, 1);
6     } while (true);
7 }

```

- 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)

```

1 void lock(lock_t *bolt) {
2     int hold = 1;
3
4     do {
5         while (bolt->busy);
6         if (CAS(&bolt->busy, false, true)) break;
7         backoff(bolt, hold);
8         if ((hold << 1) != 0) hold <<= 1;
9     } while (true);
10 }

```

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



```

1 #include "lock.h"
2 #include "earmark.h"
3
4 void backoff(lock_t *bolt, int hold) {
5     if (bolt->rest)
6         rest((*bolt->rest)[earmark()] * hold);
7 }

```

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

8 long rest(volatile long term) {
9     while (term--); /* let the holding time pass */
10    return term;
11 }

```
- 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*



```

1 void lock(lock_t *bolt, long cset) {
2     long self = FAA(&bolt->next, 1);
3
4     if (self != bolt->this) {
5         rest((self - bolt->this) * cset);
6         while (self < bolt->this);
7     }
8 }
9
10 void unlock(lock_t *bolt) {
11     bolt->this += 1; /* register next one's turn */
12 }

```

#### Interference by Ticket-Lock

Entry policy is first-come, first-served (FCFS), which rarely complies with the process scheduler policy.

- 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”



Preface

Fundamentals

Bifocal Perspective

Basic Attributes

Avenues of Approach

Atomic Memory Read/Write

Specialised Instructions

Summary



- conventional locking under prevention of context switches
  - hierarchic placement of lock/unlock implementations  $\leadsto$  ISA level
  - standby position, control mode, properties, computational burden
- approaches with atomic read/write or added specialised instructions
  - algorithms of Dekker (1965), Peterson (1981), and Kessels (1982)
  - 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 (CSEC)

That locks are suitable for a short CSEC is computer-science folklore, but by far too flat. Much more important is to have a **bounded** and, even better, **constant** CSEC. Above all, this makes high demands on the **design of critical sections** and non-sequential programs.



- [1] ABRAMSON, N. :  
The ALOHA System: Another Alternative for Computer Communication.  
In: *Proceedings of the Fall Joint Computer Conference (AFIPS '70)*.  
New York, NY, USA : ACM, 1970, S. 281–285
- [2] DAREMA-ROGERS, F. ; GEORGE, D. A. ; NORTON, V. A. ; PFISTER, G. F. :  
A VM Parallel Environment / IBM.  
1985 (RC 11225). –  
IBM Research Report
- [3] DENNIS, J. B. ; VAN HORN, E. C. :  
Programming Semantics for Multiprogrammed Computations.  
In: *Communications of the ACM* 9 (1966), März, Nr. 3, S. 143–155
- [4] DIJKSTRA, E. W. :  
Cooperating Sequential Processes / Technische Universiteit Eindhoven.  
Eindhoven, The Netherlands, 1965 (EWD-123). –  
Forschungsbericht. –  
(Reprinted in *Great Papers in Computer Science*, P. Laplante, ed., IEEE Press, New York, NY, 1996)



## Reference List II

- [5] DIJKSTRA, E. W. :  
Go To Statement Considered Harmful.  
In: *Communications of the ACM* 11 (1968), März, Nr. 3, S. 147–148. –  
Letters to the Editor
- [6] HOFRI, M. :  
Proof of a Mutual Exclusion Algorithm—A Classic Example.  
In: *ACM SIGOPS Operating Systems Review* 24 (1990), Jan., Nr. 1, S. 18–22
- [7] INTEL CORPORATION (Hrsg.):  
*i860™ Microprocessor Family Programmer's Reference Manual*.  
Santa Clara, CA, USA: Intel Corporation, 1991
- [8] KESSELS, J. L. W. :  
Arbitration Without Common Modifiable Variables.  
In: *Acta Informatica* 17 (1982), Nr. 2, S. 135–141
- [9] LAMPORT, L. :  
A New Solution of Dijkstra's Concurrent Programming Problem.  
In: *Communications of the ACM* 17 (1974), Aug., Nr. 8, S. 453–455
- [10] METCALFE, R. M. ; BOOGS, D. R. :  
Ethernet: Distributed Packet Switching for Local Computer Networks.  
In: *Communications of the ACM* 19 (1976), Jul., Nr. 5, S. 395–404



## Reference List III

- [11] PARNAS, D. L. :  
Some Hypothesis About the “Uses” Hierarchy for Operating Systems / TH  
Darmstadt, Fachbereich Informatik.  
1976 (BSI 76/1). –  
Forschungsbericht
- [12] PETERSON, G. L. :  
Myths About the Mutual Exclusion Problem.  
In: *Information Processing Letters* 12 (1981), Jun., Nr. 3, S. 115–116
- [13] SCHRÖDER-PREIKSCHAT, W. :  
Elementary Operations.  
In: LEHRSTUHL INFORMATIK 4 (Hrsg.): *Concurrent Systems*.  
FAU Erlangen-Nürnberg, 2014 (Lecture Slides), Kapitel 5



```

1 void lock(lock_t *bolt) {
2     unsigned self = earmark();
3
4     A: bolt->want[self] = true;
5     L: if (bolt->want[self^1]) {
6         if (bolt->turn[0] == self) goto L;
7         bolt->want[self] = false;
8         B: if (bolt->turn[0] == (self^1)) goto B;
9         goto A;
10    }
11 }

```

- note that **overtaking** of *self* by *peer* is volitional “feature” [4, p.13] and not owed to `goto`-less or structured, resp., programming<sup>10</sup>

9 ■ assuming that *self* gets delayed for undefined length

5 ■ then *peer* could find *CS* unoccupied and overtakes *self*

- unlock* remains unchanged (as to statements l. 13–18 of p.16)

<sup>10</sup>Disregarding the original reference, EWD is also renowned for a pamphlet that argues for abolishment of `goto` from high-level programming languages [5].

- 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 may 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<sup>11</sup> of the processes, *self* waits on *peer* for being appointed to enter *CS*
  - 9 ■ reconsider entering of the critical section...

<sup>11</sup>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:

*wait until condition* = *repeat nothing until condition*  
 = *do nothing while*  $\neg$ *condition*

applied to *C* = `while ( $\neg$ condition);`

with *condition* =  $\neg Q_i$  or *turn* = *i*

inserted and factored out = `while ( $\neg$ ( $\neg Q_i$  or turn = i));`

= `while (Qi and turn  $\neq$  i);`

= `while (Qi and turn = j);`

with *j*  $\neq$  *i*

- 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 $N > 2$

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

```

1 void lock(lock_t *lock) {
2     unsigned rank, next, self = earmark();
3
4     for (rank = 0; rank < NPROC - 1; rank++) {
5         lock->want[self] = rank;
6         lock->turn[rank] = self;
7
8         for (next = 0; next < NPROC; next++)
9             if (next != self)
10                while ((lock->want[next] >= rank)
11                    && (lock->turn[rank] == self));
12    }
13 }
14
15 void unlock(lock_t *lock) {
16     unsigned self = earmark();
17
18     lock->want[self] = -1;
19 }

```

Memory Barriers/Fences

Beware of **dynamic ordering** of read/write operations.

## 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



```

1 #include <stdbool.h>
2
3 typedef volatile struct lock {
4     bool want[NPROC]; /* initial: all false */
5     long turn[NPROC]; /* initial: all 0 */
6 } lock_t;

```

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

```

7 inline void ticketing(lock_t *bolt, unsigned slot) {
8     unsigned next, high = 0;
9
10    bolt->want[slot] = true; /* enter choosing */
11    for (next = 0; next < NPROC; next++)
12        if (bolt->turn[next] > high)
13            high = bolt->turn[next];
14    bolt->turn[slot] = high + 1; /* state number */
15    bolt->want[slot] = false; /* leave choosing */
16 }

```



## Lamport's Bakery Algorithm II

cf. [9]

```

1 void lock(lock_t *bolt) {
2     unsigned next, self = earmark();
3
4     ticketing(bolt, self); /* take a number */
5
6     for (next = 0; next < NPROC; next++) {
7         while (bolt->want[next]); /* next chooses.. */
8         while ((bolt->turn[next] != 0)
9             && ((bolt->turn[next] < bolt->turn[self])
10                || ((bolt->turn[next] == bolt->turn[self])
11                    && (next < self))))); /* next first */
12     }
13 }
14
15 void unlock(lock_t *bolt) {
16     unsigned self = earmark();
17
18     bolt->turn[self] = 0;
19 }

```

## Memory Barriers/Fences

Beware of **dynamic ordering** of read/write operations.



## Spin with TAS or CAS, resp.

cf. p.23 and p.24

- number of “busy wait” loop actions with bus locked and unlocked:

1 _lock:	9 _lock:
2 movl 4(%esp), %eax	10 movl 4(%esp), %ecx
3 LBB0_1:	11 movb \$1, %dl
4 movb \$1, %cl	12 LBB0_1:
5 xchgb %cl, (%eax)	13 xorl %eax, %eax
6 testb \$1, %cl	14 lock
7 je LBB0_1	15 cmpxchgb %dl, (%ecx)
8 ret	16 testb %al, %al
	17 jne LBB0_1
	18 ret

- 1 : 3
  - line (5) v. lines (4, 6, 7)
- 1 : 3
  - 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

