Concurrent Systems

Nebenläufige Systeme

VI. Locks

Wolfgang Schröder-Preikschat

November 27, 2018

Agenda

Outline

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. *Umlaufsperre*)

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

Outline

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:
  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

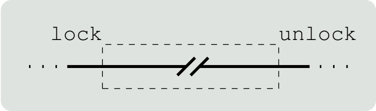
- in order that the mechanism is suited to pattern a **hardware ELOP**:[1]
  - lock
    - disables interrupts and acquires a (memory) bus lock
    - turns time monitoring on, i.e., arms some timeout mechanism
      - predefined worst-case execution time (WCET) or
      - upper limit of the number of 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 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

[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

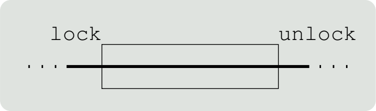
- critical section considered as logical or physical ELOP, referred to [3]

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

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. *Schlafsperre*), 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.[3]
  - a conventional lock manages on instruction set architecture level, only

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

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.

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

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 (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

Outline

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


```
#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.

Dekker's Algorithm for $N = 2$

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

- **altruistic** ("self-forgetting") entry protocol with **passsing zone**:[5]

```
void lock(lock_t *bolt) {
    unsigned self = earmark();      /* my process index */

    bolt->want[self] = true;        /* I am interested */
    while (bolt->want[self^1])      /* 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 */
        }
}

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. 38.

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

- **egoistic** ("self-serving") entry protocol with **no-passsing zone**:[6]

```
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

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

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

```
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

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**[7]

[7] 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. 40*ff.*)
- 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
  - unept 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!

Lock Type II

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

```c
#include <stdbool.h>

typedef volatile struct lock {
    bool busy;      /* initial: false */
} 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 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. 22 *ff.*). . .

Spin-Lock

(Ger.) *Umlaufsperre*

```
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

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

- 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

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

- overcomes the problem of an "unconditional store"-prone TAS

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

  - 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...

Spin on Read

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

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.

- 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[8] 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...

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

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[9] 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.

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

- 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 <u>known</u> “wranglers” for the next trial

```
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

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

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)

```
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

Backoff Procedure

```
#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

```
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**
  - that is to say, a timer interrupt is used to force the processor out of *halt*

Spin with Ticket


```
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);
    }
}

void unlock(lock_t *bolt) {
    bolt->this += 1;    /* register next one's turn */
}
```

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"

Outline

Résumé

- 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 (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

[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™ Mircroprocessor 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

Original Dekker's Algorithm for $N = 2$

cf. [4, p. 17–18]

```
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[10]
  - 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)

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

Original Dekker's Algorithm for $N = 2$

- 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[11] of the processes, *self* waits on *peer* for being appointed to enter *CS*
  - 9 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:

$$
\begin{aligned}
\textit{wait until condition} &= \textit{repeat nothing until condition}\\
&= \textit{do nothing while } \neg\textit{condition}\\
\text{applied to C} &= \texttt{while } (\neg\textit{condition})\,;\\
\text{with } \textit{condition} &= \neg Q_i \textit{ or } turn = i\\
\text{inserted and factored out} &= \texttt{while } (\neg(\neg Q_i \textit{ or } turn = i))\,;\\
&= \texttt{while } (Q_i \textit{ and } turn \neq i)\,;\\
&= \texttt{while } (Q_i \textit{ and } turn = j)\,;\\
&\ \text{with } j \neq i
\end{aligned}
$$

- 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;
}
```

Memory Barriers/Fences

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

Peterson's Solution for $N > 2$

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**

```c
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 */
}
```

Lamport's Bakery Algorithm II


```
void lock(lock_t *bolt) {
    unsigned next, self = earmark();

    ticketing(bolt, self);          /* take a number */

    for (next = 0; next < NPROC; next++) {
        while (bolt->want[next]);   /* next chooses.. */
        while ((bolt->turn[next] != 0)
           && ((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;
}
```

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:
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
```

```
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)

- 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