Concurrent Systems

Nebenläufige Systeme

VI. Locks

Wolfgang Schröder-Preikschat

November 27, 2018

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

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.

*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 do with “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 physical

- 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

- 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

© wosch, thoenig  CS (WS 2018/19, LEC 6)  Fundamentals–Bifocal Perspective
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. Schlaufsperre), idle waiting
    - lock/unlock entail system calls, thus are crucial to granularity
    - impact of system-call overhead depends on the critical section
    - number, frequency of execution, and best-case execution time

- "Passive waiting" for unlock is atypical 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.

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 (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 vs. higher probability of interference
  - The more fine-grained the object, the higher overhead/lower contention
    - Easily audible background noise vs. 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
    - 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 contending 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
  - 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
#include <stdbool.h>

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] */
}
```

Dekker’s Algorithm for $N = 2$

```
void lock(lock_t *bolt) {
  unsigned self = earmark();   /* my process index */
  bolt->want[self] = true;     /* I am interested */
  while (bolt->want[self-1]) { /* & inside CS */
    bolt->want[self-1] = 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;
  bolt->want[self] = false;     /* I defer to you */
}
```

5 For an interpretation, see also p. 38.
Peterson’s Algorithm for $N = 2$ cf. [12]

Egoistic (“self-serving”) entry protocol with no-passing 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[(self ^ 1)] /* 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 */
}
```

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

---

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

Solutions Based on Dedicated Processor Instructions

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
- who’s next uses feedback as to peer’s view on who’s turn was last
- 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

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

---

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

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

Egoistic (“self-serving”) entry protocol with no-passing 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[(self ^ 1)] /* 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 */
}
```

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

---

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

Solutions Based on Dedicated Processor Instructions

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
- who’s next uses feedback as to peer’s view on who’s turn was last
- 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

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

---

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

© wosch, theoenig CS (WS 2018/19, LEC 6) Avenues of Approach–Specialised Instructions 21

Spin-Lock

```c
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:
  1. assuming that these actions are due to simultaneous processes
  2. all these processes might find the door to the critical section open
  3. all of those processes who found the door open will lock the door
  4. all of those who locked the door will enter the critical section
  5. → 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

© wosch, theoenig CS (WS 2018/19, LEC 6) Avenues of Approach–Specialised Instructions 22

Spin with TAS

cf. p. 44

```c
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]:
  ```c
  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 ~ **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...

© wosch, theoenig CS (WS 2018/19, LEC 6) Avenues of Approach–Specialised Instructions 23

Spin with CAS

cf. p. 44

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

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

  ```c
  CAS = \{
    \begin{array}{ll}
      \text{true} & \rightarrow \text{store true into busy, if } busy = \text{false, otherwise}
      \end{array}
  \}
  ```

  - 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...

© wosch, theoenig CS (WS 2018/19, LEC 6) Avenues of Approach–Specialised Instructions 24
Spin on Read

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

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

### Critical Section Execution Time (CSET)

The actual wait loop proceeds with a full-time unlocked bus. The lock is acquired at a time of a possibly desert critical section.

### 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.

### Avoidance of Bus Lock Bursts

**Definition**

Static or dynamic holding time, stepped on a per-process(or) basis, to prevent lock contention.

- 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

**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);
}
```
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);
    }
}
```

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

- 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

Preface

Fundamentals

Bifocal Perspective

Basic Attributes

Avenues of Approach

Atomic Memory Read/Write

Specialised Instructions

Summary
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 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.:
In: Proceedings of the Fall Joint Computer Conference (AFIPS ’70).

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

Cooperating Sequential Processes / Technische Universiteit Eindhoven.
Forschungsbericht. –

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

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.):
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.

Ethernet: Distributed Packet Switching for Local Computer Networks.
In: Communications of the ACM 19 (1976), Jul., Nr. 5, S. 395–404

Reference List III

Some Hypothesis About the “Uses” Hierarchy for Operating Systems / TH Darmstadt, Fachbereich Informatik.
1976 (SSI 76/1). –
Forschungsbericht

[12] Peterson, G. L.:
Myths About the Mutual Exclusion Problem.

[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 $N = 2$  

```c
void lock(lock_t *bolt) {
    unsigned self = earmark();
    A: bolt->want[self] = true;
    L: if (bolt->want(self-1)) {
        if (bolt->turn[0] == self) goto L;
        bolt->want[self] = false;
    }
    B: if (bolt->turn[0] == (self-1)) goto B;
    goto A;
}
```

note that overtaking of self by peer is volitional “feature” [4, p. 13] and not owed to goto-less or structured, resp., programming\(^\text{10}\)

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 $N = 2$ 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:
  - 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\(^\text{11}\) 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:
  - `wait until condition` = repeat nothing until condition = do nothing while ¬condition
  - applied to C = while (¬condition);
  - with condition = ¬Qi or turn = i
  - inserted and factored out = while (¬(¬Qi or turn = i));
  - = while (Qi and turn ≠ i);
  - = while (Qi and turn = j);
  - with j ≠ 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.

```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;
    }
```

© wosch, thoenig  CS (WS 2018/19, LEC 6)  Addendum – Load/Store

\(\text{Memory Barriers/Fences}\)

Beware of dynamic ordering of read/write operations.
Peterson’s Solution for $N > 2$

**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:
  - in attempting to enter $CS$, indicate interest to reach the next rank
  - for it, check all other processes for their particular rank and
  - 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;

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

**cf. [9]**

```c
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;
}
```

Spin with TAS or CAS, resp.

cf. p23 and p.24

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

```asm
_line: movl 4(%esp), %eax
     _lock:
     LBB0_1:
     movb $1, %cl
     xorl %eax, %eax
     lock cmpxchgb %cl, (%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
```

---

Memory Barriers/Fences

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