

# Concurrent Systems

*Nebenläufige Systeme*

## V. Elementary Operations

Wolfgang Schröder-Preikschat

November 27, 2019



# Agenda

---

Preface

Primitive Instructions  
Atomic Operations  
Equivalence

Memory Models  
Properties

Summary



# Outline

---

Preface

Primitive Instructions  
Atomic Operations  
Equivalence

Memory Models  
Properties

Summary



- discussion on **abstract concepts** as to elementary operations at instruction structure set architecture level
  - atomic load/store of a naturally aligned machine word
  - atomic read-modify-write of complex machine instructions



- discussion on **abstract concepts** as to elementary operations at instruction structure set architecture level
  
- impartation of knowledge on memory models that are relevant to multi-threading on multi/many-core (multi-) processors
  - atomicity, visibility, and ordering of memory operations against the background of UMA, NUMA, and (partly) COMA architectures
  - ordering enforcing hardware such as memory barriers or fences, resp., allowing one to pattern sequential, relaxed, and weak **data consistency**





# Outline

---

Preface

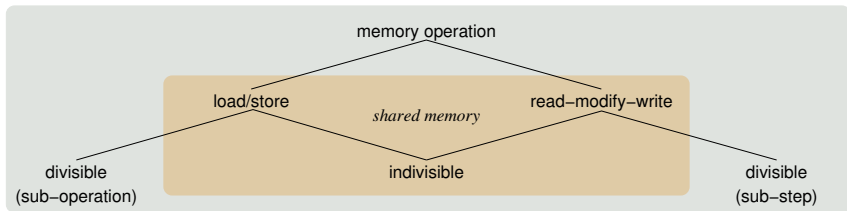
Primitive Instructions  
Atomic Operations  
Equivalence

Memory Models  
Properties

Summary



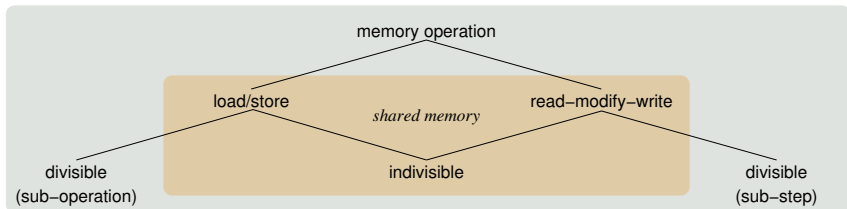
# Memory-Operation Semantics



- of particular interest (at this point) are **shared-memory operations**
  - commonality is the opportunity, at least, for **indivisible execution**



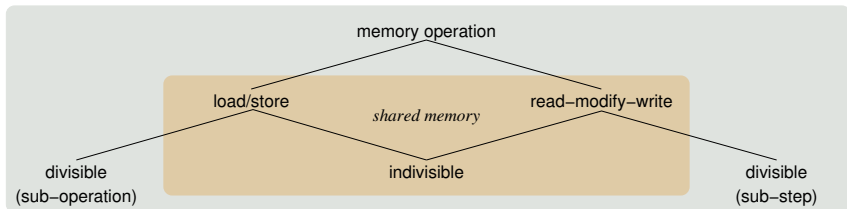
# Memory-Operation Semantics



- of particular interest (at this point) are **shared-memory operations**
- note, all memory operations are also divisible in the following respect:  
**sub-operation**
  - processors are word-oriented, but memory is byte-oriented
  - with *word size* as a multiple of *byte size*, e.g.  $4 \times 8$  bits
  - thus, loads/stores will operate on a **sequence of bytes**



# Memory-Operation Semantics



- of particular interest (at this point) are **shared-memory operations**
- note, all memory operations are also divisible in the following respect:
  - sub-operation** ■ processors are word-oriented, but memory is byte-oriented
  - sub-step** ■ processors perform a *fetch-execute-cycle* to run programs
  - $n$ -address machines mean  $n$ -operand instructions,  $n \geq 2$ <sup>1</sup>
  - thus, execution requires a **sequence of loads/stores**

<sup>1</sup>In general  $n \geq 0$ , but only for  $n \geq 2$  becomes the problem apparent.

```
1 #include <stdint.h>
2
3 static int64_t label;
4
5 int64_t get_label() {
6     return label;
7 }
8
9 void set_label(int64_t value) {
10     label = value;
11 }
```



```
1 #include <stdint.h>
2
3 static int64_t label;
4
5 int64_t get_label() {
6     return label;
7 }
8
9 void set_label(int64_t value) {
10     label = value;
11 }
```

- in logical respect any of these single statements is indivisible, atomic
  - lines 6 conceals a load and line 10 conceals a store operation
  - each case forms an ELOP of the **abstract processor** “C”



```
1 #include <stdint.h>
2
3 static int64_t label;
4
5 int64_t get_label() {
6     return label;
7 }
8
9 void set_label(int64_t value) {
10     label = value;
11 }
```

- in logical respect any of these single statements is indivisible, atomic
- in physical respect these statements are **conditionally atomic**, only
  - a matter of optimisation options, the CPU, and alignment restrictions



```
1 #include <stdint.h>
2
3 static int64_t label;
4
5 int64_t get_label() {
6     return label;
7 }
8
9 void set_label(int64_t value) {
10     label = value;
11 }
```

*Beware of  
devirtualisation!*

- in logical respect any of these single statements is indivisible, atomic
- in physical respect these statements are **conditionally atomic**, only



```
gcc -m32...
```

```
1  get_label:  
2      movl label, %eax  
3      movl label+4, %edx  
4      ret  
5  
6  set_label:  
7      movl 4(%esp), %eax  
8      movl 8(%esp), %ecx  
9      movl %ecx, label+4  
10     movl %eax, label  
11     ret
```



```
gcc -m64...
```

```
12  get_label:  
13      movq label(%rip), %rax  
14      ret  
15  
16  set_label:  
17      movq %rdi, label(%rip)  
18      ret
```



```
gcc -m32...
```

```
1  get_label:
2      movl label, %eax
3      movl label+4, %edx
4      ret
5
6  set_label:
7      movl 4(%esp), %eax
8      movl 8(%esp), %ecx
9      movl %ecx, label+4
10     movl %eax, label
11     ret
```

```
gcc -m64...
```

```
12  get_label:
13     movq label(%rip), %rax
14     ret
15
16  set_label:
17     movq %rdi, label(%rip)
18     ret
```

- actions 2-3 and 9-10 are divisible
- any of these 8 mov instructions is **conditionally indivisible**



```
gcc -m32...
```

```
1  get_label:
2      movl label, %eax
3      movl label+4, %edx
4      ret
5
6  set_label:
7      movl 4(%esp), %eax
8      movl 8(%esp), %ecx
9      movl %ecx, label+4
10     movl %eax, label
11     ret
```

```
gcc -m64...
```

```
12  get_label:
13     movq label(%rip), %rax
14     ret
15
16  set_label:
17     movq %rdi, label(%rip)
18     ret
```

- beware of the processor architecture or the data alignment, resp.
  - usually, memory-word loads/stores are indivisible if “word” corresponds to the smallest addressable unit of main memory: **byte**, nowadays
  - on some architectures (e.g., x86) they are indivisible too if the address of the memory operand is *naturally aligned*



- **execution cycle** of a machine instruction that involves the ALU<sup>2</sup>

---

<sup>2</sup>*arithmetic-logic unit*, the operation unit of the CPU.

- **execution cycle** of a machine instruction that involves the ALU<sup>2</sup>
  - consists of the following individual operation steps:
    - i load input operands (acc. operation code or addressing mode, resp.)
    - ii compute result (acc. operation code)
    - iii store output operand (acc. operation code or addressing mode, resp.)

---

<sup>2</sup>*arithmetic-logic unit*, the operation unit of the CPU.

- **execution cycle** of a machine instruction that involves the ALU<sup>2</sup>
  - consists of the following individual operation steps:
    - i load input operands (acc. operation code or addressing mode, resp.)
    - ii compute result (acc. operation code)
    - iii store output operand (acc. operation code or addressing mode, resp.)
  - steps (i) and (iii) require the **bus** in case of memory-sensitive operations
    - reusable hardware resource, shareable, allocated per (load/store) step
  - typical **compound action** at instruction set architecture (ISA) level
    - is memory-sensitive only for a *complex instruction set computer* (CISC)

---

<sup>2</sup>*arithmetic-logic unit*, the operation unit of the CPU.

- **execution cycle** of a machine instruction that involves the ALU<sup>2</sup>
  - consists of the following individual operation steps:
    - i load input operands (acc. operation code or addressing mode, resp.)
    - ii compute result (acc. operation code)
    - iii store output operand (acc. operation code or addressing mode, resp.)
  - steps (i) and (iii) require the **bus** in case of memory-sensitive operations
  
- in a **multiprocessor case**, the whole cycle is divisible (non-atomic)
  - merely the individual sub-steps may form indivisible actions (cf. p. 8)
  - while the loads/stores may be in sync, the compound action is not

---

<sup>2</sup> *arithmetic-logic unit*, the operation unit of the CPU.

- **execution cycle** of a machine instruction that involves the ALU<sup>2</sup>
  - consists of the following individual operation steps:
    - i load input operands (acc. operation code or addressing mode, resp.)
    - ii compute result (acc. operation code)
    - iii store output operand (acc. operation code or addressing mode, resp.)
  - steps (i) and (iii) require the **bus** in case of memory-sensitive operations
  
- in a **multiprocessor case**, the whole cycle is divisible (non-atomic)
  
- indivisibility requires a **bus lock** for the duration of the whole cycle:
  - i an **atomic RMW instruction** that implicitly performs the lock *or*
  - ii a **lock prefix** that makes the adjacent normal RMW instruction atomic

---

<sup>2</sup>*arithmetic-logic unit*, the operation unit of the CPU.

### Definition (TS, acc. IBM System/370)

*The leftmost bit (bit position 0) of the byte located at the second-operand address is used to set the condition code, and then the entire addressed byte is set to all ones. [8, p. 144]*



### Definition (TS, acc. IBM System/370)

*The leftmost bit (bit position 0) of the byte located at the second-operand address is used to set the condition code, and then the entire addressed byte is set to all ones. [8, p. 144]*

- the operation effectly does an **unconditional store** in main memory
  - *The byte in storage is set to all ones as it is fetched for the testing of bit position 0. [8, p. 144]*<sup>3</sup>

<sup>3</sup>A similar effect has ldstub of SPARC V9.



## Definition (TS, acc. IBM System/370)

*The leftmost bit (bit position 0) of the byte located at the second-operand address is used to set the condition code, and then the entire addressed byte is set to all ones. [8, p. 144]*

- the operation effectly does an **unconditional store** in main memory
  - in terms of **main memory significance**, this translates into the following:
    - with first and second operand being used to form effective address *ref*

```
1 bool tas(byte *ref) {
2     atomic { bool cc = *ref & 0x1; *ref = 0x11111111; }
3     return cc;
4 }
```



## Definition (TS, acc. IBM System/370)

*The leftmost bit (bit position 0) of the byte located at the second-operand address is used to set the condition code, and then the entire addressed byte is set to all ones. [8, p. 144]*

- the operation effectly does an **unconditional store** in main memory

```
1 bool tas(byte *ref) {  
2     atomic { bool cc = *ref & 0x1; *ref = 0x11111111; }  
3     return cc;  
4 }
```

- note that TS interlocks against simultaneous main memory accesses



- the original copy (IBM System/370) has **swapping characteristic**
  - $swap(x, y)$ , with  $x = *ref_{[0]}$  and  $y = 11111111_2_{[0]}$



- the original copy (IBM System/370) has **swapping characteristic**
  - for a contemporary processor (x86), this translates into the following:

```
1  int tas(any_t *ref) {      4  tas:
2      return TAS(ref);      5      movl  4(%esp), %ecx
3  }                          6      movl  $1, %eax
                              7      xchgl %eax, (%ecx)
                              8      ret
```



- the original copy (IBM System/370) has **swapping characteristic**
  - for a contemporary processor (x86), this translates into the following:

```
1 int tas(any_t *ref) {      4 tas:
2     return TAS(ref);      5     movl 4(%esp), %ecx
3 }                          6     movl $1, %eax
                              7     xchgl %eax, (%ecx)
                              8     ret
```

- whereby (using GCC atomic built-in functions):

```
9 #define TAS(ref) __sync_lock_test_and_set(ref, 1)
```



- the original copy (IBM System/370) has **swapping characteristic**
  - for a contemporary processor (x86), this translates into the following:

```
1 int tas(any_t *ref) {      4 tas:
2     return TAS(ref);      5     movl 4(%esp), %ecx
3 }                          6     movl $1, %eax
                              7     xchgl %eax, (%ecx)
                              8     ret
```

- whereby (using GCC atomic built-in functions):
- ```
9 #define TAS(ref) __sync_lock_test_and_set(ref, 1)
```
- note that xchg interlocks against simultaneous main memory accesses



- the original copy (IBM System/370) has **swapping characteristic**

- for a contemporary processor (x86), this translates into the following:

```

1  int tas(any_t *ref) {      4  tas:
2      return TAS(ref);      5      movl  4(%esp), %ecx
3  }                          6      movl  $1, %eax
                              7      xchgl %eax, (%ecx)
                              8      ret

```

- whereby (using GCC atomic built-in functions):

```

9  #define TAS(ref) __sync_lock_test_and_set(ref, 1)

```

- beware of the unconditional store carried out by both TS and xchg<sup>4</sup>
    - this semantic has a **deleterious effect** for cache-coherent processors
    - the cache line holding the main memory operand is always invalidated
- ↪ dedicated hardware implementation (p. 12) or mapping to CAS (p. 13)

<sup>4</sup>Same holds for TAS of the M68000 family and ldstub of the SPARC family.

## Definition (Dual-Ported RAM)

A kind of random access memory (RAM) that supports simultaneous load and store operations from two directions.



## Definition (Dual-Ported RAM)

A kind of random access memory (RAM) that supports simultaneous load and store operations from two directions.

- the **interlock** is conducted by a “DPRAM monitor” that, e.g. [18]:
  - records the processor that issued the TAS and acquired access
  - notifies processors that, at a time, issue a TAS simultaneously
    - signalling *BUSY* interrupt, forcing the receiving processor into **busy waiting**
  - performs the test and then, if and only if the test succeeds:
    - i sets the memory location to the value given by the owning processor *and*
    - ii releases access to that memory location



## Definition (Dual-Ported RAM)

A kind of random access memory (RAM) that supports simultaneous load and store operations from two directions.

- the **interlock** is conducted by a “DPRAM monitor”
  
- this scheme translates into a **conditional store** as follows:

```
1 word tas(word *ref) {  
2     word aux;  
3     atomic { if ((aux = *ref) == 0) *ref = 1; }  
4     return aux;  
5 }
```



## Definition (CS, acc. IBM System/370)

*The first and second operands are compared. If they are equal, the third operand is stored in the second-operand location. If they are unequal, the second operand is loaded into the first-operand location. [8, p. 123]*



## Definition (CS, acc. IBM System/370)

*The first and second operands are compared. If they are equal, the third operand is stored in the second-operand location. If they are unequal, the second operand is loaded into the first-operand location. [8, p. 123]*

- the operation effectly performs a **conditional store** in main memory
  - *The first and third operands [each are] occupying a general register. The second operand is a word in main storage. [8, p. 123]*



## Definition (CS, acc. IBM System/370)

*The first and second operands are compared. If they are equal, the third operand is stored in the second-operand location. If they are unequal, the second operand is loaded into the first-operand location. [8, p. 123]*

- the operation effectly performs a **conditional store** in main memory
  - in terms of **main memory significance**, this translates into the following:

```
1 atomic word cas(register old, word *ref, register new) {  
2     bool cc;  
3     return cc = (*ref == old) ? (*ref = new, 0) : (old = *ref, 1);  
4 }
```



## Definition (CS, acc. IBM System/370)

*The first and second operands are compared. If they are equal, the third operand is stored in the second-operand location. If they are unequal, the second operand is loaded into the first-operand location. [8, p. 123]*

- the operation effectly performs a **conditional store** in main memory

```
1 atomic word cas(register old, word *ref, register new) {  
2     bool cc;  
3     return cc = (*ref == old) ? (*ref = new, 0) : (old = *ref, 1);  
4 }
```

- note that CS interlocks against simultaneous main memory accesses



## Definition (ABA, also A-B-A)

*The ABA problem is a **false positive** execution of a CAS-based speculation on a shared location  $L_i$ . [2, p. 186]*



## Definition (ABA, also A-B-A)

*The ABA problem is a **false positive** execution of a CAS-based speculation on a shared location  $L_i$ . [2, p. 186]*

- when the successful execution of a CAS instruction indicates:
  - i that the two operands subject to comparison are equal and, thus, purport the presence of a certain condition (*positive*),
  - ii but the condition is not in fact present (*false*)



## Definition (ABA, also A-B-A)

*The ABA problem is a **false positive** execution of a CAS-based speculation on a shared location  $L_i$ . [2, p. 186]*

- assuming that processes  $P_1$  and  $P_2$  simultaneously access location  $L_i$ 
  - value  $A$  read by  $P_1$  from  $L_i$  be a sign of a dedicated global state  $S_1$ , but  $P_1$  will be delayed before being able to commit a new value to  $L_i$
  - meanwhile  $P_2$  changes the value of  $L_i$  to  $B$  and then back to  $A$ , defining a new global state  $S_2 \neq S_1$
  - $P_1$  resumes, observes that the value of  $L_i$  equals  $A$  and, thus, acts on the assumption that the global state must be  $S_1$ —which is no longer true



## Definition (ABA, also A-B-A)

*The ABA problem is a **false positive** execution of a CAS-based speculation on a shared location  $L_i$ . [2, p. 186]*

- assuming that processes  $P_1$  and  $P_2$  simultaneously access location  $L_i$
  
  
  
  
  
  
  
  
  
  
- severity of false positive execution depends on the problem (cf. p. 36)



## Definition

Paired instructions to form a flow of actions without any guarantee of indivisibility but that it succeeds only in case of indivisible operation.

- originated in the MIPS II or R6000, resp., RISC architecture [9]



## Definition

Paired instructions to form a flow of actions without any guarantee of indivisibility but that it succeeds only in case of indivisible operation.

- originated in the MIPS II or R6000, resp., RISC architecture [9]:
  - LL ■ loads a word from the specified effective memory address
    - makes a **reservation** on that very address (range)<sup>5</sup>
  - SC ■ checks for a reservation on the specified effective memory address<sup>5</sup>
    - if the reservation persists, stores the specified word at that address
    - delivers the result of the reservation check

---

<sup>5</sup>The dimension of the reservation depends on the hardware implementation. It may be exact the effective address or a larger address range around.



## Definition

Paired instructions to form a flow of actions without any guarantee of indivisibility but that it succeeds only in case of indivisible operation.

- originated in the MIPS II or R6000, resp., RISC architecture [9]:
  
  
  
  
  
  
  
  
  
  
  - reasons for **cancellation** of a persisting address (range) reservation:
    - i successful execution of SC—hoped for, normally
    - ii execution of LL by another processor applying the same address (range)
    - iii an exception (trap/interrupt) on the processor holding the reservation
- 





## Load-Linked/Store-Conditional II

- use of LL/SC to recreate TAS and CAS:
  - in case of TAS, a boolean variable is conditionally set true

```
1 int tas(long *ref) {  
2     return (LL(ref) == 0) && SC(ref, 1);  
3 }
```

- in case of CAS, a memory word is conditionally overwritten

```
4 int cas(long *ref, long old, long new) {  
5     return (LL(ref) == old) && SC(ref, new);  
6 }
```



## Load-Linked/Store-Conditional II

- use of LL/SC to recreate TAS and CAS:

- in case of TAS, a boolean variable is conditionally set true

```
1 int tas(long *ref) {
2     return (LL(ref) == 0) && SC(ref, 1);
3 }
```

- in case of CAS, a memory word is conditionally overwritten

```
4 int cas(long *ref, long old, long new) {
5     return (LL(ref) == old) && SC(ref, new);
6 }
```

- note that this implementation of CAS is free from the ABA problem:

$P_1$  ■ shares location  $ref$  with  $P_2$ , established reservation  $ref_{P_1}$  by LL

- gets delayed for some reason, thus has not yet executed SC

$P_2$  ■ overlaps  $P_1$ , establishes reservation  $ref_{P_2}$  and, thus, cancels  $ref_{P_1}$

- successfully executes SC  $\Rightarrow$  CAS succeeds

$P_1$  ■ resumes  $\Rightarrow$  SC will fail because reservation  $ref_{P_1}$  is invalid

- returns failure of CAS  $\Rightarrow$  rolls back, backs up, and retries CAS...



## Definition (acc. [6, p. 17])

A value-returning instruction that operates on a global (i.e., shared) variable  $G$  and a local variable  $L$ .



## Definition (acc. [6, p. 17])

A value-returning instruction that operates on a global (i.e., shared) variable  $G$  and a local variable  $L$ .

- an atomic RMW instruction, inspired by “Replace Add” [3, p. 6]
  - prefix (FAA) or postfix (AAF) form, as to when fetch becomes effective
    - prefix – save the old value of  $G$  for return, then add  $L$  to  $G$
    - postfix – add  $L$  to  $G$ , then return the new value of  $G$



## Definition (acc. [6, p. 17])

A value-returning instruction that operates on a global (i.e., shared) variable  $G$  and a local variable  $L$ .

- an atomic RMW instruction, inspired by “Replace Add” [3, p. 6]
  - prefix (FAA) or postfix (AAF) form, as to when fetch becomes effective
  - whereby (cf. p. 39):

$$FAA(G, L) \equiv AAF(G, L) - L \quad \text{and}$$

$$AAF(G, L) \equiv FAA(G, L) + L$$



## Definition (acc. [6, p. 17])

A value-returning instruction that operates on a global (i.e., shared) variable  $G$  and a local variable  $L$ .

- an atomic RMW instruction, inspired by “Replace Add” [3, p. 6]
  - prefix (FAA) or postfix (AAF) form, as to when fetch becomes effective
  - whereby (cf. p. 39):

$$FAA(G, L) \equiv AAF(G, L) - L \quad \text{and}$$

$$AAF(G, L) \equiv FAA(G, L) + L$$

- transferable to any associative binary operation *fetch-and- $\Phi$* 
  - but for noninvertible operations the prefix form is considered more general
  - be  $\Phi = \max$  (i.e.,  $X$ ): only  $XAF(G, L) \equiv \max(FA(X)(G, L), L)$  (cf. p. 40)



- operations that need consensus number  $n$  cannot have a **semantically equivalent implementation** by operations of consensus number  $m < n$

## Definition (Consensus Number)

*The consensus number for  $X$  is the largest  $n$  for which  $X$  solves  $n$ -process consensus. If no largest  $n$  exists, the consensus number is said to be infinite. [7, p. 130]*

- $n$  processes need to interact to achieve agreement on a single data value
- note that only 1-process consensus requires no interaction



- operations that need consensus number  $n$  cannot have a **semantically equivalent implementation** by operations of consensus number  $m < n$

## Definition (Consensus Number)

*The consensus number for  $X$  is the largest  $n$  for which  $X$  solves  $n$ -process consensus. If no largest  $n$  exists, the consensus number is said to be infinite. [7, p. 130]*

- consensus numbers of the elementary operations considered:
  - $\infty$  ■ compare-and-swap, load-linked/store-conditional
  - 2 ■ test-and-set, swap, fetch-and-add
  - 1 ■ atomic read, atomic write



- operations that need consensus number  $n$  cannot have a **semantically equivalent implementation** by operations of consensus number  $m < n$

## Definition (Consensus Number)

*The consensus number for  $X$  is the largest  $n$  for which  $X$  solves  $n$ -process consensus. If no largest  $n$  exists, the consensus number is said to be infinite. [7, p. 130]*

- key point is the **progress guarantee** a certain operation has to give
  - for wait-freedom [7], the operation must have consensus number  $n = \infty$
  - in that case, every action has guarantee to complete in finite steps/time



# Outline

---

Preface

Primitive Instructions  
Atomic Operations  
Equivalence

Memory Models  
Properties

Summary



# Properties Relevant to Multi-Threading

---

- fundamental characteristics that are of particular importance for the implementation of any synchronisation algorithm



# Properties Relevant to Multi-Threading

---

- fundamental characteristics that are of particular importance for the implementation of any synchronisation algorithm:
  - atomicity
    - as to *how* certain machine instructions are executed
    - differentiates in RISC and CISC machines
    - specific to each ELOP that was discussed before (pp. 7–17)



# Properties Relevant to Multi-Threading

---

- fundamental characteristics that are of particular importance for the implementation of any synchronisation algorithm:
  - atomicity** ■ as to *how* certain machine instructions are executed
  
  - visibility** ■ as to *when* memory-cell changes are observable
    - concerns delays in sensing the most recent memory-word write
    - introduces time factors on the availability of written data



# Properties Relevant to Multi-Threading

- fundamental characteristics that are of particular importance for the implementation of any synchronisation algorithm:
  - atomicity** ■ as to *how* certain machine instructions are executed
  - visibility** ■ as to *when* memory-cell changes are observable
  - ordering** ■ as to *how* memory operations appear to be performed
    - stands for a variant of *out-of-order* execution
    - reflects on (sequential, relaxed, or weak) consistency models



# Properties Relevant to Multi-Threading

- fundamental characteristics that are of particular importance for the implementation of any synchronisation algorithm:
  - atomicity** ■ as to *how* certain machine instructions are executed
  - visibility** ■ as to *when* memory-cell changes are observable
  - ordering** ■ as to *how* memory operations appear to be performed
- these properties are linked with each other, are mutual prerequisites
  - atomicity applies to all other—and to a single machine instruction, only
  - visibility depends on the memory architecture, may cause “jitter”
  - ordering comprises multiple machine instructions, may cause “fencing”



# Properties Relevant to Multi-Threading

- fundamental characteristics that are of particular importance for the implementation of any synchronisation algorithm:
  - atomicity** ■ as to *how* certain machine instructions are executed
  - visibility** ■ as to *when* memory-cell changes are observable
  - ordering** ■ as to *how* memory operations appear to be performed
- these properties are linked with each other, are mutual prerequisites
- as to the level of abstraction, they must all be considered together
  - this is especially true for the operating-system machine level (i.e., level 3)



- common are two classes of memory-sensitive operations (cf. p. 25):
  - L/S
    - atomic load (L) or store (S), resp., as single action
    - granularity is the **machine word**, i.e., a multiple of a byte
    - with **word-alignment** constraint on the operand address, usually
      - only word-aligned accesses will be carried out indivisibly
  - RMW
    - atomic read (R), modify (M), and write (W) as single action
    - common for CISC and, there, for **two-address machines**
      - uncommon for RISC, which is characteristic of load/store principle
    - single- or double-word cycles for 32- or 64-bit architectures, resp.
      - “double” means “physically consecutive” or “logically interrelated”
      - i.e.: CDS or `cmpxchg8b/cmpxchg16b` compared to DCAS or CAS2



# Atomicity

- common are two classes of memory-sensitive operations (cf. p. 25):
  - L/S ■ atomic load (L) or store (S), resp., as single action
  
  - RMW ■ atomic read (R), modify (M), and write (W) as single action
  
- processes cannot observe any intermediate steps and partial effects
  - here, only in matters of a single (L/S or RMW) machine instruction
  - that is to say, the ISA-level action appears *indivisible* and *irreducible*
  - as a consequence, the instruction will be performed entirely or not all
    - with the latter meaning *failure indication* (TAS, CAS, SC)



When other interacting processes will notice the changes made by the current process, and whether they will notice them at all.

- depends on the **memory architecture** and behaviour of read or write operations to the same memory location



When other interacting processes will notice the changes made by the current process, and whether they will notice them at all.

- depends on the **memory architecture** and behaviour of read or write operations to the same memory location
  - UMA ■ *uniform memory architecture*  $\leadsto$  the same access time
    - each address is assigned a fixed home in the global address space
    - no processor uses private (local) memory besides shared memory



When other interacting processes will notice the changes made by the current process, and whether they will notice them at all.

- depends on the **memory architecture** and behaviour of read or write operations to the same memory location

**NUMA** ■ *non-uniform memory architecture*  $\leadsto$  different access times

- each address is assigned a fixed home in the global address space
- each processor (“NUMA node”) uses private (local) memory, too



When other interacting processes will notice the changes made by the current process, and whether they will notice them at all.

- depends on the **memory architecture** and behaviour of read or write operations to the same memory location

- COMA
- *cache-only memory architecture*  $\leadsto$  different access times
  - no address is assigned a fixed home in the global address space
  - each processor uses private (local) memory, only

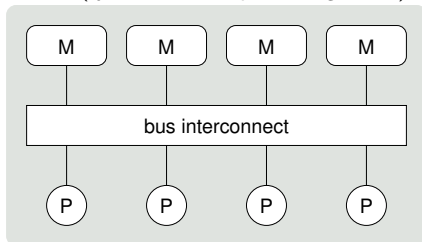


When other interacting processes will notice the changes made by the current process, and whether they will notice them at all.

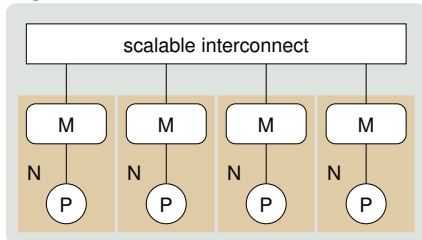
- depends on the **memory architecture** and behaviour of read or write operations to the same memory location
  - UMA ■ *uniform memory architecture*
  - NUMA ■ *non-uniform memory architecture*
  - COMA ■ *cache-only memory architecture*
- orthogonal with it is the **consistency** aspect as to shared information stored in multiple local **caches**
  - *cache-coherent (cc)* v. *non-cache-coherent (ncc)* memory architecture



## UMA (*symmetric multiprocessing, SMP*)



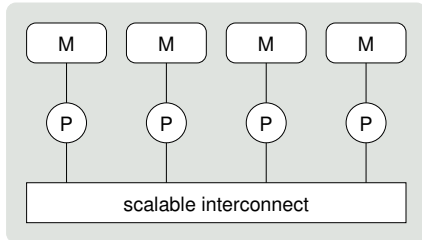
## NUMA



- NUMA node (N)
  - zone of uniform memory characteristic
- NUMA distance
  - number of (network) hops to distant memory



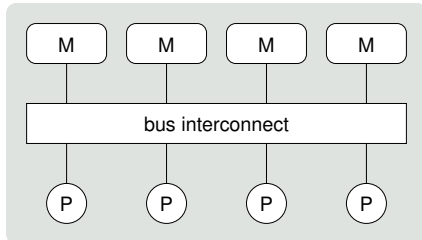
## COMA



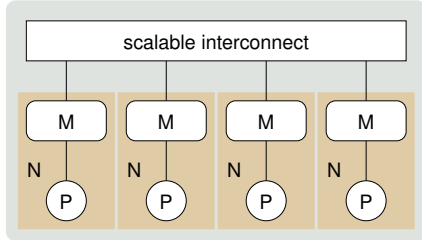
- COMA distance
  - number of (network) hops to distant memory



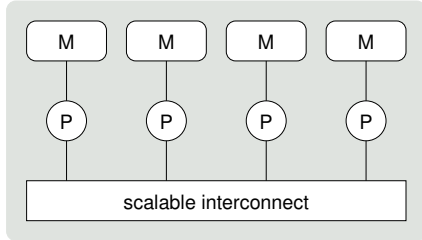
## UMA (symmetric multiprocessing, SMP)



## NUMA



## COMA



- NUMA node (N)
  - zone of uniform memory characteristic
- NUMA/COMA distance
  - number of (network) hops to distant memory
- UMA/NUMA combination



# Ordering

---

What memory re-orderings are possible for a process, relatively to the order as specified by its program.



What memory re-orderings are possible for a process, relatively to the order as specified by its program.

- to improve performance, memory-sensitive machine instructions are not executed in the order originally specified by the program
  - on the one hand, the compiler reorders (L3) instructions<sup>6</sup> before run-time
  - on the other hand, the CPU reorders (L2) instructions<sup>6</sup> at run-time
    - it is this aspect of **dynamic ordering** that is of relevance in the following

---

<sup>6</sup>According to the actual level of abstraction: operating-system machine (L3) or instruction set architecture (L2) level. See also [10] or [17, p. 34].



# Ordering

---

What memory re-orderings are possible for a process, relatively to the order as specified by its program.

- to improve performance, memory-sensitive machine instructions are not executed in the order originally specified by the program
  
  - mainly, dynamic ordering is an issue of non-blocking synchronisation
    - as blocking synchronisation implicitly can take care of “fencing” proper
      - depending on the kind of critical section and type of data dependency
    - but, critical section *per se* is no guarantee for memory ordering (cf. p. 25)
- 



# Ordering

---

What memory re-orderings are possible for a process, relatively to the order as specified by its program.

- to improve performance, memory-sensitive machine instructions are not executed in the order originally specified by the program
  - mainly, dynamic ordering is an issue of non-blocking synchronisation
  - ordering ensuring needs special instructions: **memory barrier/fence**
- 



- assuming that the following function is executed by a single processor, but the global variables are then read by at least one more processor:

```
1 int a = 1, b = 2;   ■ what values of a and b do other processors see  
2                   once line 6 has been reached by one processor?  
3 void ab_set() {  
4     a = 3;  
5     b = 4;  
6 }
```



- assuming that the following function is executed by a single processor, but the global variables are then read by at least one more processor:

```
1 int a = 1, b = 2;
2
3 void ab_set() {
4     a = 3;
5     b = 4;
6 }
```

- what values of  $a$  and  $b$  do other processors see once line 6 has been reached by one processor?
  - (1, 2), (1, 4), (3, 2), (3, 4)
  - depending on processor and memory architecture



- assuming that the following function is executed by a single processor, but the global variables are then read by at least one more processor:

```
1 int a = 1, b = 2;
2
3 void ab_set() {
4     a = 3;
5     b = 4;
6 }
```

- what values of  $a$  and  $b$  do other processors see once line 6 has been reached by one processor?
  - (1, 2), (1, 4), (3, 2), (3, 4)
  - depending on processor and memory architecture
- writes are not necessarily seen by other processors in the order as specified by the program!



- assuming that the following function is executed by a single processor, but the global variables are then read by at least one more processor:

```
1 int a = 1, b = 2;
2
3 void ab_set() {
4     a = 3;
5     b = 4;
6 }
```

- assuming that the next function is executed directly afterwards to the former one just discussed, but by a different processor:

```
7 void ab_get(int ab[2]) {
8     ab[0] = b;
9     ab[1] = a;
10 }
```

■ what values of  $a$  and  $b$  are delivered?



- assuming that the following function is executed by a single processor, but the global variables are then read by at least one more processor:

```
1 int a = 1, b = 2;
2
3 void ab_set() {
4     a = 3;
5     b = 4;
6 }
```

- assuming that the next function is executed directly afterwards to the former one just discussed, but by a different processor:

```
7 void ab_get(int ab[2]) {
8     ab[0] = b;
9     ab[1] = a;
10 }
```

■ what values of *a* and *b* are delivered?

- line 8 may read the *new* value of *b* while
- line 9 may read the *old* value of *a*

- although the assignment to *a* (line 4) was instructed previous to the one of *b*



*Memory barrier instructions directly control only the interaction of a CPU with its cache, with its write-buffer that holds stores waiting to be flushed to memory, and/or its buffer of waiting loads or speculatively executed instructions. [12]*



*Memory barrier instructions directly control only the interaction of a CPU with its cache, with its write-buffer that holds stores waiting to be flushed to memory, and/or its buffer of waiting loads or speculatively executed instructions. [12]*

- $ld_a$  LoadLoad  $ld_b$ 
  - ensures that  $a$  is read before  $b$  is accessed<sup>7</sup>
  - speculative loads, out-of-order processing
- $st_a$  StoreStore  $st_b$ 
  - ensures that  $a$  is visible before  $b$  is flushed<sup>7</sup>
  - disordered flushes from write buffers
- $ld_a$  LoadStore  $st_b$ 
  - ensures that  $a$  is read before  $b$  is flushed<sup>7</sup>
  - out-of-order processors that can bypass loads
- $st_a$  StoreLoad  $ld_b$ 
  - ensures that  $a$  is visible before  $b$  is accessed<sup>7</sup>
  - write to *same* location by another processor

<sup>7</sup>Including the execution of all subsequent loads or stores, resp.

*Memory barrier instructions directly control only the interaction of a CPU with its cache, with its write-buffer that holds stores waiting to be flushed to memory, and/or its buffer of waiting loads or speculatively executed instructions. [12]*

- $ld_a$  LoadLoad  $ld_b$  ■ ensures that  $a$  is read before  $b$  is accessed<sup>7</sup>
- $st_a$  StoreStore  $st_b$  ■ ensures that  $a$  is visible before  $b$  is flushed<sup>7</sup>
- $ld_a$  LoadStore  $st_b$  ■ ensures that  $a$  is read before  $b$  is flushed<sup>7</sup>
- $st_a$  StoreLoad  $ld_b$  ■ ensures that  $a$  is visible before  $b$  is accessed<sup>7</sup>
- CAS and LL/SC typically include a StoreLoad barrier on the target
  - i.e., not only a general-purpose but also the most expensive fence

<sup>7</sup>Including the execution of all subsequent loads or stores, resp.

- **data consistency** as close as possible to sequential processes or with optimisation margins for high-latency memory



- **data consistency** as close as possible to sequential processes or with optimisation margins for high-latency memory
  - **sequential**
    - processors see writes on the same target in the same order
    - but the order may appear different for an “external observer”<sup>8</sup>
    - two requirements: **program order** and **write atomicity** [11]

---

<sup>8</sup>Weaker than “strict consistency” that requires a read from a memory location to return the value of the most recent write.



- **data consistency** as close as possible to sequential processes or with optimisation margins for high-latency memory

- relaxed
- in terms of the constraints defined by sequential consistency
  - as to (i) program order, (ii) write atomicity, or (iii) both:
    - i write to read, write to write, read to read and read to write
    - ii read other's, write early
    - iii read own, write early
  - pertaining to (i) different or (ii) same memory locations



- **data consistency** as close as possible to sequential processes or with optimisation margins for high-latency memory

- weak**
- “limited to hardware-recognized synchronizing variables” [4]
  - yet weaker tendencies: release [5] and entry [1] consistency
    - implemented by operating system machine level programs
    - usually not provided by the instruction set architecture level



- **data consistency** as close as possible to sequential processes or with optimisation margins for high-latency memory

**relaxed** ■ in terms of the constraints defined by sequential consistency

**weak** ■ “limited to hardware-recognized synchronizing variables” [4]

- state of the art processors provide relaxed or weak consistency models

<sup>8</sup>Weaker than “strict consistency” that requires a read from a memory location to return the value of the most recent write.



# Outline

---

Preface

Primitive Instructions  
Atomic Operations  
Equivalence

Memory Models  
Properties

Summary



- elementary operations at instruction set architecture level
  - atomic load/store of a naturally aligned machine (double-) word
  - atomic read-modify-write of complex machine instructions
    - TAS, CAS and FAA or FAΦ, resp., for CISC and LL/SC for RISC
  - equality of atomic operations as to their **consensus number**
- memory-access properties that are relevant to multi-threading
  - atomicity, visibility, and ordering of memory operations
  - memory architectures of type UMA, NUMA, and COMA
  - **dynamic ordering** at instruction set architecture level
  - memory barriers or fences, resp., to enforce ordering proper
  - sequential, relaxed, and weak **data consistency**
- **hardware features** that are of importance for the implementation of any synchronisation algorithm
  - including but not limited to non-blocking synchronisation, especially



- elementary operations at instruction structure set architecture level
  - atomic load/store of a naturally aligned machine (double-) word
  - atomic read-modify-write of complex machine instructions
    - TAS, CAS and FAA or FAΦ, resp., for CISC and LL/SC for RISC
  - equality of atomic operations as to their **consensus number**
- memory-access properties that are relevant to multi-threading
  - atomicity, visibility, and ordering of memory operations
  - memory architectures of type UMA, NUMA, and COMA
  - **dynamic ordering** at instruction set architecture level
  - memory barriers or fences, resp., to enforce ordering proper
  - sequential, relaxed, and weak **data consistency**
- **hardware features** that are of importance for the implementation of any synchronisation algorithm
  - including but not limited to non-blocking synchronisation, especially



- elementary operations at instruction set architecture level
  - atomic load/store of a naturally aligned machine (double-) word
  - atomic read-modify-write of complex machine instructions
    - TAS, CAS and FAA or FAΦ, resp., for CISC and LL/SC for RISC
  - equality of atomic operations as to their **consensus number**
- memory-access properties that are relevant to multi-threading
  - atomicity, visibility, and ordering of memory operations
  - memory architectures of type UMA, NUMA, and COMA
  - **dynamic ordering** at instruction set architecture level
  - memory barriers or fences, resp., to enforce ordering proper
  - sequential, relaxed, and weak **data consistency**
- hardware features that are of importance for the implementation of any synchronisation algorithm
  - including but not limited to non-blocking synchronisation, especially



- elementary operations at instruction structure set architecture level
  - atomic load/store of a naturally aligned machine (double-) word
  - atomic read-modify-write of complex machine instructions
    - TAS, CAS and FAA or FAΦ, resp., for CISC and LL/SC for RISC
  - equality of atomic operations as to their **consensus number**
- memory-access properties that are relevant to multi-threading
  - atomicity, visibility, and ordering of memory operations
  - memory architectures of type UMA, NUMA, and COMA
  - **dynamic ordering** at instruction set architecture level
  - memory barriers or fences, resp., to enforce ordering proper
  - sequential, relaxed, and weak **data consistency**
- **hardware features** that are of importance for the implementation of any synchronisation algorithm
  - including but not limited to non-blocking synchronisation, especially



# Reference List I

---

- [1] BERSHAD, B. N. ; ZEKAUSKAS, M. J.:  
Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors / School of Computer Science, Carnegie Mellon University.  
Pittsburgh, PA, USA, 1991 (CMU-CS-91-170). –  
Forschungsbericht
  
- [2] DECHEV, D. ; PIRKELBAUER, P. ; STROUSTRUP, B. :  
Understanding and Effectively Preventing the ABA Problem in Descriptor-based Lock-free Designs.  
In: *Proceedings of the 13th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2010)*, IEEE Computer Society, 2010. – ISBN 978-1-4244-7083-9, S. 185-192
  
- [3] DRAUGHON, W. E. ; GRISHMAN, R. ; SCHWARTZ, J. ; STEIN, A. :  
Programming Considerations for Parallel Computers / Courant Institute of Mathematical Sciences.  
New York University, Nov. 1967 (IMM 362). –  
Forschungsbericht



## Reference List II

---

- [4] DUBOIS, M. ; SCHEURICH, C. ; BRIGGS, F. A.:  
Memory Access Buffering in Multiprocessors.  
In: AISO, H. (Hrsg.): *Proceedings of the 13th International Symposium on Computer Architecture (ISCA 1986)*, IEEE Computer Society, 1986. – ISBN 0-8186-0719-X, S. 434-442
- [5] GHARACHORLOO, K. ; LENOSKI, D. ; LAUDON, J. ; GIBBONS, P. ; GUPTA, A. ; HENNESSY, J. :  
Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors.  
In: *Proceedings of the 17th International Symposium on Computer Architecture (ISCA 1990)*, ACM, 1990. – ISBN 0-89791-366-3, S. 15-26
- [6] GOTTLIEB, A. ; KRUSKAL, C. P.:  
Coordinating Parallel Processors: A Partial Unification.  
In: *ACM SIGARCH Computer Architecture News* 9 (1981), Okt., Nr. 6, S. 16-24
- [7] HERLIHY, M. :  
Wait-Free Synchronization.  
In: *ACM Transactions on Programming Languages and Systems* 11 (1991), Jan., Nr. 1, S. 124-149



- [8] IBM CORPORATION (Hrsg.):  
*IBM System/370 Principles of Operation.*  
Fourth.  
Poughkeepsie, New York, USA: IBM Corporation, Sept. 1 1974.  
(GA22-7000-4, File No. S/370-01)
- [9] KANE, G. ; HEINRICH, J. :  
*MIPS RISC Architecture.*  
Second.  
Prentice Hall, 1991. –  
ISBN 978-0135904725
- [10] KLEINÖDER, J. ; SCHRÖDER-PREIKSCHAT, W. :  
*Rechnerorganisation.*  
In: LEHRSTUHL INFORMATIK 4 (Hrsg.): *Systemprogrammierung.*  
FAU Erlangen-Nürnberg, 2014 (Vorlesungsfolien), Kapitel 5
- [11] LAMPORT, L. :  
How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs.  
In: *IEEE Transactions on Computers* C-28 (1979), Sept., Nr. 9, S. 690–691



- [12] LEA, D. :  
*The JSR-133 Cookbook for Compiler Writers.*  
<http://gee.cs.oswego.edu/dl/jmm/cookbook.html>, März 2011
- [13] MOSBERGER, D. :  
Memory Consistency Models / Department of Computer Science, University of Arizona.  
Tucson, AZ, USA, 1991 (TR 93/11). –  
Forschungsbericht
- [14] MOTOROLA INC. (Hrsg.):  
*Motorola M68000 Family Programmer's Reference Manual.*  
Rev. 1.  
Schaumburg, IL, USA: Motorola Inc., 1992.  
(M68000PM/AD)
- [15] SCHRÖDER-PREIKSCHAT, W. ; LEHRSTUHL INFORMATIK 4 (Hrsg.):  
*Concurrent Systems.*  
FAU Erlangen-Nürnberg, 2014 (Lecture Slides)
- [16] SCHRÖDER-PREIKSCHAT, W. :  
Critical Sections.  
In: [15], Kapitel 4



[17] SCHRÖDER-PREIKSCHAT, W. :  
Processes.  
In: [15], Kapitel 3

[18] WIKIPEDIA:  
*Test-and-Set*.  
<http://en.wikipedia.org/wiki/Test-and-set>, Mai 2014



## Unconditional Store: Workaround

- “textbook semantics” of TAS has a **deleterious effect** for the cache:

```
1 word tas(word *ref) {  
2     atomic { word aux = *ref; *ref = 1; }  
3     return aux;  
4 }
```

- same is true when using the GCC atomic built-in function (x86, cf. p 11):

```
5 #define TAS(ref) __sync_lock_test_and_set(ref, 1)
```



## Unconditional Store: Workaround

- “textbook semantics” of TAS has a **deleterious effect** for the cache:

```
1 word tas(word *ref) {
2     atomic { word aux = *ref; *ref = 1; }
3     return aux;
4 }
```

- same is true when using the GCC atomic built-in function (x86, cf. p11):

```
5 #define TAS(ref) __sync_lock_test_and_set(ref, 1)
```

- use of CAS, with `#define CAS __sync_bool_compare_and_swap`

```
6 int tas(long *ref) {
7     return CAS(ref, 0, 1);
8 }
9 tas:
10     xorl    %eax, %eax
11     movl   $1, %ecx
12     movl   4(%esp), %edx
13     lock
14     cmpxchgl %ecx, (%edx)
15     testl  %eax, %eax
16     sete  %al
17     movzbl %al, %eax
18     ret
```

- worst-case overhead of five instructions (cf. p11)
- pays off, depending on processor and cache architecture



- given a LIFO list (i.e., stack) of following structure:  $head \leftrightarrow A \leftrightarrow B \leftrightarrow C$ 
  - with  $head$  stored at location  $L_i$ ; shared by processes  $P_1$  and  $P_2$
  - *push* (cf. [16, p.11]) and *pull* adding or removing, resp., list items:

```
1 chain_t *cas_pull(stack_t *this) {
2     chain_t *node;
3     do if ((node = this->head.link) == 0) break;
4     while (!CAS(&this->head.link, node, node->link));
5     return node;
6 }
```



- given a LIFO list (i.e., stack) of following structure:  $head \leftrightarrow A \leftrightarrow B \leftrightarrow C$ 
  - with  $head$  stored at location  $L_i$ ; shared by processes  $P_1$  and  $P_2$
  - *push* (cf. [16, p.11]) and *pull* adding or removing, resp., list items:

```

1 chain_t *cas_pull(stack_t *this) {
2     chain_t *node;
3     do if ((node = this->head.link) == 0) break;
4     while (!CAS(&this->head.link, node, node->link));
5     return node;
6 }

```

- assuming that the following sequence of actions will take place:
  - $P_1$ 
    - reads head item  $A$  followed by  $B$  on the list, gets delayed at line 4
    - remembers  $node = A$ , but has not yet done CAS:  $head \leftrightarrow A \leftrightarrow B \leftrightarrow C$
  - $P_2$ 
    - pulls head item  $A$  from the list:  $head \leftrightarrow B \leftrightarrow C$
    - pulls head item  $B$  from the list:  $head \leftrightarrow C$
    - pushes item  $A$  back to the list, now followed by  $C$ :  $head \leftrightarrow A \leftrightarrow C$
  - $P_1$ 
    - resumes, CAS realises  $head = A$  (followed by  $B$ ):  $head \leftrightarrow B \leftrightarrow \ominus$
    - list state  $head \leftrightarrow A \leftrightarrow C$  as left behind by  $P_2$  is lost...



- prevalent approach is to add a **change number** to the “control word” [8, p. 125], i.e., to practice some kind of **versioning**
  - this number increments at each CAS attempt on the control word
- appropriate techniques depend on the change-number parameters



- prevalent approach is to add a **change number** to the “control word” [8, p. 125], i.e., to practice some kind of **versioning**
  - this number increments at each CAS attempt on the control word
- appropriate techniques depend on the change-number parameters
  - a. the values margin has a whole word size available
    - both the control and change-number word must be updated, indivisibly
    - *compare double and swap* (CDS, [8, p. 124]) of two consecutive words<sup>9</sup>
    - *double compare and swap* (DCAS, also CAS2 [14, p. 4-66]) of any two words

<sup>9</sup>See also `cmpxchg8b` or `cmpxchg16b`, in case of x86.



- prevalent approach is to add a **change number** to the “control word” [8, p. 125], i.e., to practice some kind of **versioning**
  - this number increments at each CAS attempt on the control word
- appropriate techniques depend on the change-number parameters
  - a. the values margin has a whole word size available
    - both the control and change-number word must be updated, indivisibly
    - *compare double and swap* (CDS, [8, p. 124]) of two consecutive words<sup>9</sup>
    - *double compare and swap* (DCAS, also CAS2 [14, p. 4-66]) of any two words
  - b. the values margin utilizes fully unused bits in the control word itself
    - CAS facilitates indivisible updates of control word including change number
    - workaround, especially suitable for handling aligned data-structure **pointers**
    - gimmick is in data-structure padding for an object size of a power of two

<sup>9</sup>See also `cmpxchg8b` or `cmpxchg16b`, in case of x86.



- prevalent approach is to add a **change number** to the “control word” [8, p. 125], i.e., to practice some kind of **versioning**
    - this number increments at each CAS attempt on the control word
  - appropriate techniques depend on the change-number parameters
    - a. the values margin has a whole word size available
      - both the control and change-number word must be updated, indivisibly
      - *compare double and swap* (CDS, [8, p. 124]) of two consecutive words<sup>9</sup>
      - *double compare and swap* (DCAS, also CAS2 [14, p. 4-66]) of any two words
    - b. the values margin utilizes fully unused bits in the control word itself
      - CAS facilitates indivisible updates of control word including change number
      - workaround, especially suitable for handling aligned data-structure **pointers**
      - gimmick is in data-structure padding for an object size of a power of two
- ↪ an object size of  $2^n$  bytes then gives  $n - 1$  low-order bits always 0
  - ↪ these  $n - 1$  low-order bits then will be used as a **change-number tag**
  - ↪ for **pointer operations**, the change-number tag is temporary neutralised

<sup>9</sup>See also `cmpxchg8b` or `cmpxchg16b`, in case of x86.

- prevalent approach is to add a **change number** to the “control word” [8, p. 125], i.e., to practice some kind of **versioning**
    - this number increments at each CAS attempt on the control word
  - appropriate techniques depend on the change-number parameters
    - a. the values margin has a whole word size available
      - both the control and change-number word must be updated, indivisibly
      - *compare double and swap* (CDS, [8, p. 124]) of two consecutive words<sup>9</sup>
      - *double compare and swap* (DCAS, also CAS2 [14, p. 4-66]) of any two words
    - b. the values margin utilizes fully unused bits in the control word itself
      - CAS facilitates indivisible updates of control word including change number
      - workaround, especially suitable for handling aligned data-structure **pointers**
      - gimmick is in data-structure padding for an object size of a power of two
- ↪ an object size of  $2^n$  bytes then gives  $n - 1$  low-order bits always 0
  - ↪ these  $n - 1$  low-order bits then will be used as a **change-number tag**
  - ↪ for **pointer operations**, the change-number tag is temporary neutralised
- but the ABA problem never disappears, it only gets more improbable

<sup>9</sup>See also `cmpxchg8b` or `cmpxchg16b`, in case of x86.

- as GCC does not provide atomic built-in functions for this case:

```

1  long LL(long *ref) {
2      long aux;
3
4      asm volatile(
5          "lwarx %0, 0, %1"
6          : "=r" (aux)
7          : "r" (ref));
8
9      return aux;
10 }

11 int SC(long *ref, long val) {
12     long ccr;
13
14     asm volatile(
15         "stwcx. %2, 0, %1\n\t"
16         "mfcrr %0"
17         : "=r" (ccr)
18         : "r" (ref), "r" (val)
19         : "cc", "memory");
20
21     return ccr & 0x2;
22 }

```

- with `#define INLINE extern inline` for GCC to ensure that stand-alone object code is never emitted for in-line functions<sup>10</sup>

<sup>10</sup>Use `#define INLINE inline` for C99, for the same reason.

## ■ #define FAA \_\_sync\_fetch\_and\_add

```
1 int faa(int *p, int v) {           4 faa:
2     return FAA(p, v);             5     movl  4(%esp), %ecx
3 }                                  6     movl  8(%esp), %eax
                                    7     lock
                                    8     xaddl %eax, (%ecx)
                                    9     ret
```



## ■ #define FAA \_\_sync\_fetch\_and\_add

```
1 int faa(int *p, int v) {           4 faa:
2     return FAA(p, v);             5     movl  4(%esp), %ecx
3 }                                  6     movl  8(%esp), %eax
                                   7     lock
                                   8     xaddl %eax, (%ecx)
                                   9     ret
```

## ■ #define AAF \_\_sync\_add\_and\_fetch

```
10 int aaf(int *p, int v) {          13 aaf:
11     return AAF(p, v);             14     movl  4(%esp), %ecx
12 }                                  15     movl  8(%esp), %edx
                                   16     movl  %edx, %eax
                                   17     lock
                                   18     xaddl %eax, (%ecx)
                                   19     addl  %edx, %eax
                                   20     ret
```



- safe-load of global variable  $G$  and conditional-store of  $\max(G, L)$  at  $G$

```
1 word fax(word *ref, word val) {  
2     word aux;  
3     atomic { if ((aux = *ref) < val) *ref = val; }  
4     return aux;  
5 }
```



- safe-load of global variable  $G$  and conditional-store of  $\max(G, L)$  at  $G$

```
1 word fax(word *ref, word val) {
2     word aux;
3     atomic { if ((aux = *ref) < val) *ref = val; }
4     return aux;
5 }
```

- conditional-store of  $\max(G, L)$  at  $G$  and return of  $\max(G, L)$

```
6 word xaf(word *ref, word val) {
7     atomic { word aux = (*ref > val) ? *ref : *ref = val; }
8     return aux;
9 }
```



- safe-load of global variable  $G$  and conditional-store of  $\max(G, L)$  at  $G$

```
1 word fax(word *ref, word val) {
2     word aux;
3     atomic { if ((aux = *ref) < val) *ref = val; }
4     return aux;
5 }
```

- conditional-store of  $\max(G, L)$  at  $G$  and return of  $\max(G, L)$

```
6 word xaf(word *ref, word val) {
7     atomic { word aux = (*ref > val) ? *ref : *ref = val; }
8     return aux;
9 }
```

- assuming that  $G = 42$  and  $L = 4711$ :

- $XAF(G, L) \equiv \max(FAX(G, L), L)$ : both terms result in 4711
- $FAX(G, L) \not\equiv \max(XAF(G, L), L)$ :  $FAX$  may result in  $42 < 4711$

