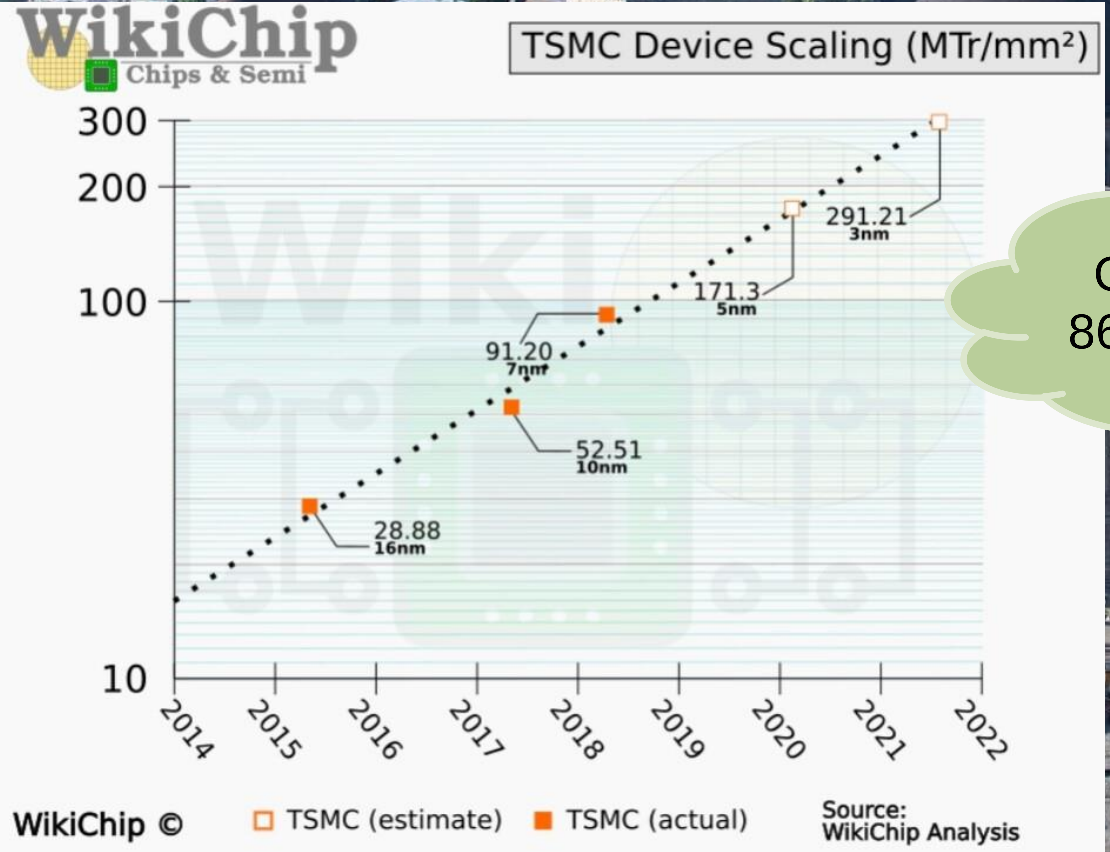
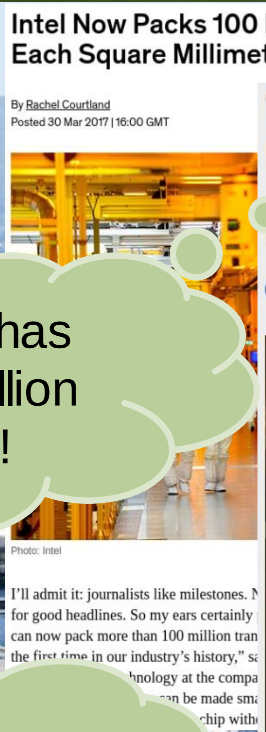
TORSTEN HOEFLER

# Parallel Programming, Spring 2021, Lecture 15: Solving Mutual Exclusion for many processes, Hardware Primitives for mutual excl



Your brain has about 86 billion neurons!

Ok, with a total of 86 trillion synapses.

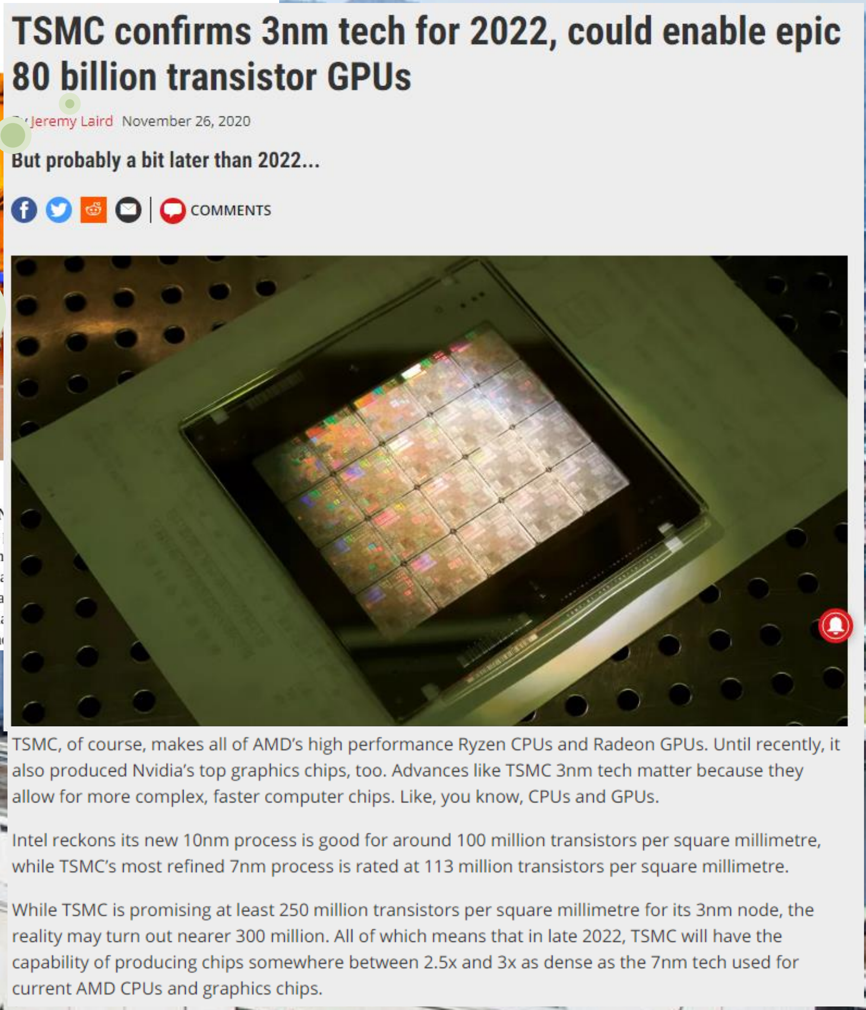


Intel Now Packs 100 Million Transistors in Each Square Millimeter

By Rachel Courtland
Posted 30 Mar 2017 | 16:00 GMT

Photo: Intel

I'll admit it: journalists like milestones. N
for good headlines. So my ears certainly
can now pack more than 100 million tran
the first time in our industry's history," sa
hnology at the compa
an be made sma
chip with



TSMC confirms 3nm tech for 2022, could enable epic 80 billion transistor GPUs

Jeremy Laird November 26, 2020

But probably a bit later than 2022...

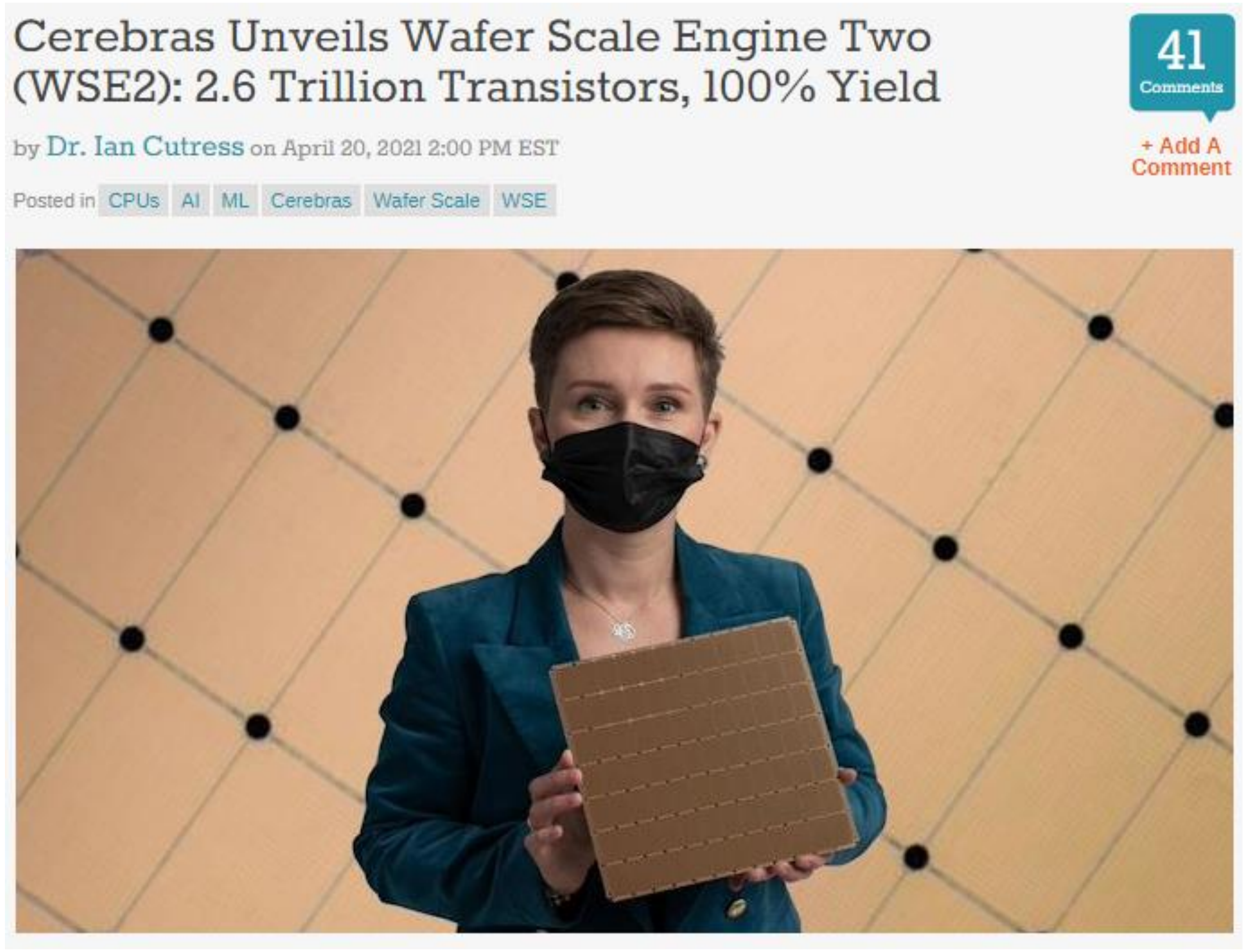
COMMENTS

TSMC, of course, makes all of AMD's high performance Ryzen CPUs and Radeon GPUs. Until recently, it also produced Nvidia's top graphics chips, too. Advances like TSMC 3nm tech matter because they allow for more complex, faster computer chips. Like, you know, CPUs and GPUs.

Intel reckons its new 10nm process is good for around 100 million transistors per square millimetre, while TSMC's most refined 7nm process is rated at 113 million transistors per square millimetre.

While TSMC is promising at least 250 million transistors per square millimetre for its 3nm node, the reality may turn out nearer 300 million. All of which means that in late 2022, TSMC will have the capability of producing chips somewhere between 2.5x and 3x as dense as the 7nm tech used for current AMD CPUs and graphics chips.

## … that was so much last year … what happened yesterday?



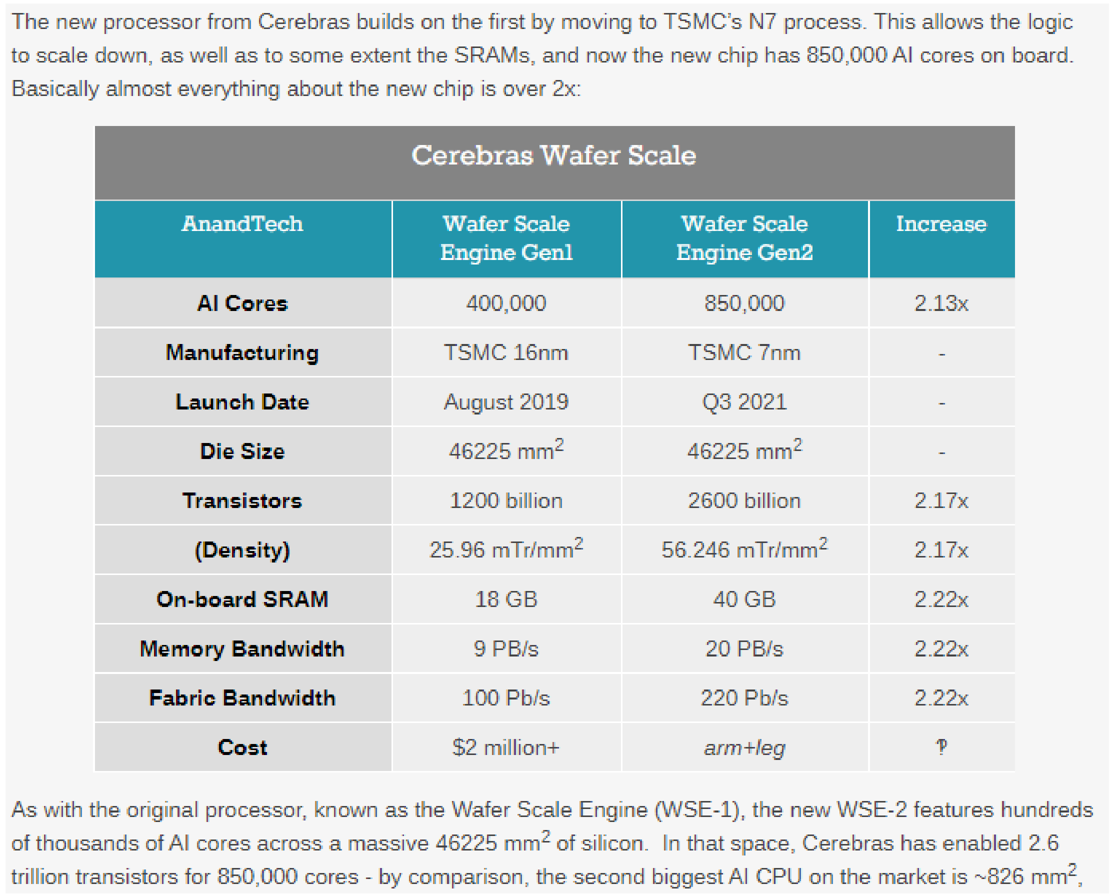
Cerebras Unveils Wafer Scale Engine Two (WSE2): 2.6 Trillion Transistors, 100% Yield

by Dr. Ian Cutress on April 20, 2021 2:00 PM EST

41 Comments

+ Add A Comment

Posted in CPUs AI ML Cerebras Wafer Scale WSE



The new processor from Cerebras builds on the first by moving to TSMC's N7 process. This allows the logic to scale down, as well as to some extent the SRAMs, and now the new chip has 850,000 AI cores on board. Basically almost everything about the new chip is over 2x:

| Cerebras Wafer Scale | | | |
|---|---|---|---|
| AnandTech | Wafer Scale Engine Gen1 | Wafer Scale Engine Gen2 | Increase |
| AI Cores | 400,000 | 850,000 | 2.13x |
| Manufacturing | TSMC 16nm | TSMC 7nm | - |
| Launch Date | August 2019 | Q3 2021 | - |
| Die Size | 46225 mm$^2$ | 46225 mm$^2$ | - |
| Transistors | 1200 billion | 2600 billion | 2.17x |
| (Density) | 25.96 mTr/mm$^2$ | 56.246 mTr/mm$^2$ | 2.17x |
| On-board SRAM | 18 GB | 40 GB | 2.22x |
| Memory Bandwidth | 9 PB/s | 20 PB/s | 2.22x |
| Fabric Bandwidth | 100 Pb/s | 220 Pb/s | 2.22x |
| Cost | $2 million+ | *arm+leg* | ? |

As with the original processor, known as the Wafer Scale Engine (WSE-1), the new WSE-2 features hundreds of thousands of AI cores across a massive 46225 mm$^2$ of silicon. In that space, Cerebras has enabled 2.6 trillion transistors for 850,000 cores - by comparison, the second biggest AI CPU on the market is ~826 mm$^2$,

## Administrivia

- **Zoom links for exercise sessions may change!**
  - Please check the webpage and/or Moodle
    http://spcl.inf.ethz.ch/Teaching/2021-pp/
- **We are streaming on youtube channel "SPCL Lab"**
  - Lectures will also appear in a playlist there
- **Head TA for the second section: Timo Schneider**
- **If anything goes wrong during an exercise: call him ☺**
  - +41764688942
- **If anything non-urgent happens, send him email**
  - timos@inf.ethz.ch
- **For questions – ask your group TA (or me in the break)**

## Learning goals for today

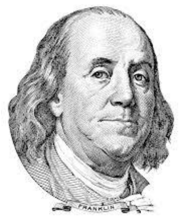
**So far:**

- **Simple proofs of correctness and unexpected problems with real computers**
- **Memory models as contract between programmer, compiler, runtime, and architecture**
- **Java's volatile and synchronized**

**Now:**

- Implementation of a two-thread locks with Atomic Registers
  *Dekker's algorithm*
  *Peterson's algorithm*
- Implementation of n-thread locks with Atomic Registers
  *Filter lock*
  *Bakery lock*
- Locks using atomic operations
  *TAS, TATAS, exponential backoff lock*

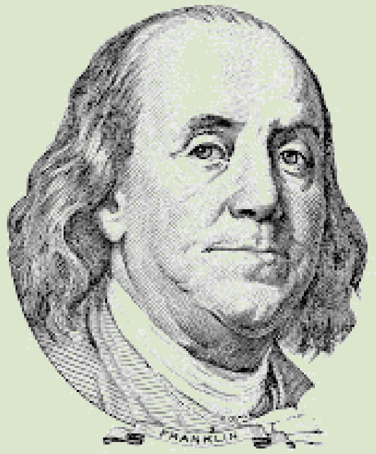
- Context: remember you will not use these locks (you will use functions provided by the programming model!)
  YET: you will learn important principles by "doing" – and watching your (my) mistakes carefully



*"Tell me and I forget, teach me and I may remember, involve me and I learn."*

## Recap last lecture by a short quiz

- **Please participate in the zoom poll!**



*"Tell me and I forget, teach me and I may remember, involve me and I learn."*

## Java Memory Model (JMM): Necessary basics

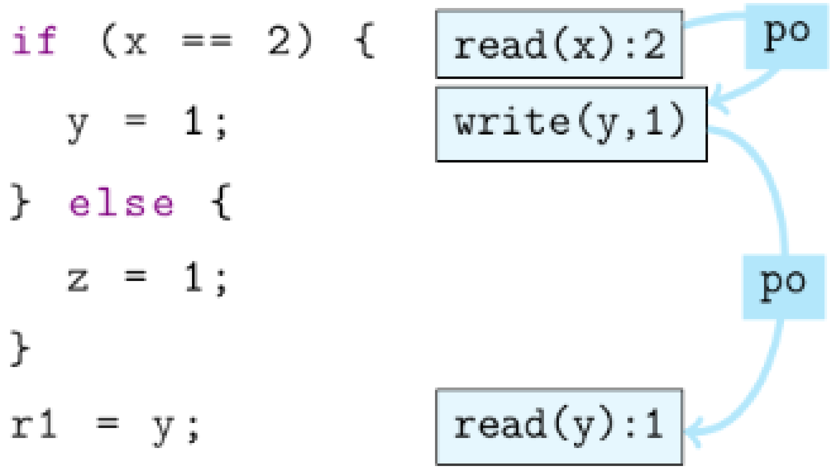
- **JMM restricts allowable outcomes of programs**
  - You saw that if we don't have these operations (volatile, synchronized etc.) – outcome can be "arbitrary" (not quite correct, say unexpected ☺)

- **JMM defines *Actions:* `read(x):1` "read variable x, the value read is 1"**

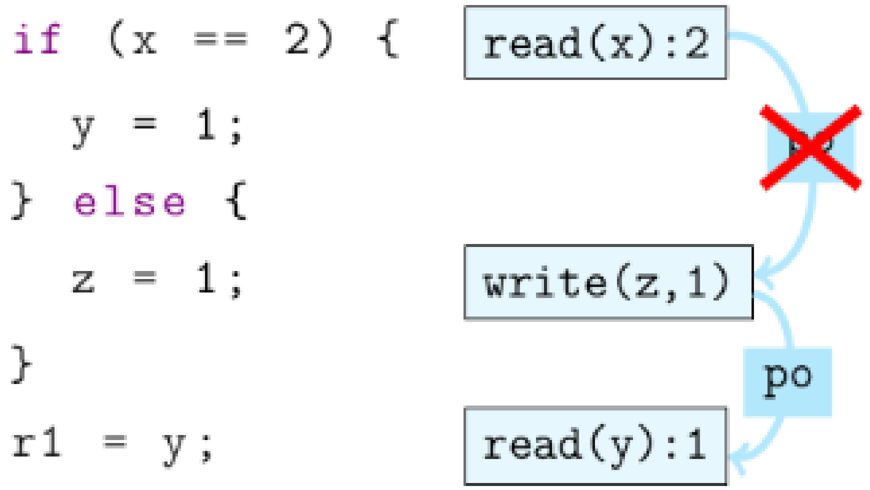
- ***Executions* combine *actions* with *ordering:***
  - *Program Order*
  - *Synchronization Order*
  - *Synchronizes-with*
  - *Happens-before*

## JMM: Program Order (PO)

- **Program order is a total order of intra-thread actions**
  - Program statements are NOT a total order across threads!
- **Program order does not provide an ordering guarantee for memory accesses!**
  - The only reason it exists is to provide the link between possible executions and the original program.
- **Intra-thread consistency: Per thread, the PO order is consistent with the threads isolated execution**



```
if (x == 2) {
  y = 1;
} else {
  z = 1;
}
r1 = y;
```



```
if (x == 2) {
  y = 1;
} else {
  z = 1;
}
r1 = y;
```

## JMM: Synchronization Actions (SA) and Synchronization Order (SO)

- **Synchronization actions are:**
  - Read/write of a volatile variable
  - Lock monitor, unlock monitor
  - First/last action of a thread (synthetic)
  - Actions which start a thread
  - Actions which determine if a thread has terminated
- **Synchronization Actions form the Synchronization Order (SO)**
  - SO is a total order
  - All threads see SA in the same order
  - SA within a thread are in PO
  - SO is consistent: all reads in SO see the last writes in SO

```
volatile int x, y;
--------------------------------
x = 1;           | y = 1;
int r1 = y;      | int r2 = x;
```

Exercise: List all outcomes (r1,r2) allowed by the JMM.

## JMM: Synchronizes-With (SW) / Happens-Before (HB) orders

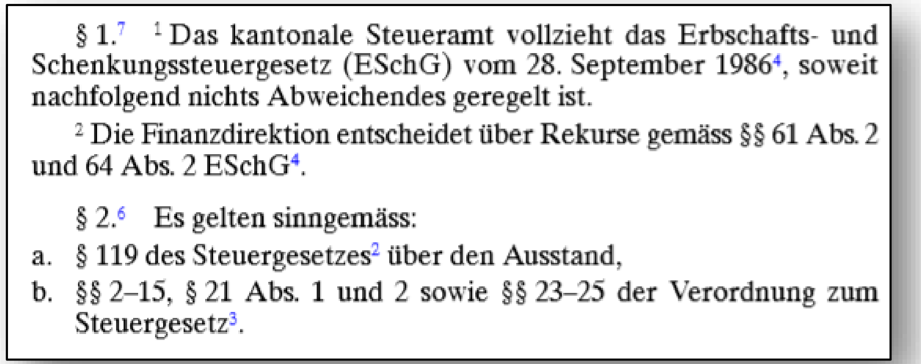
- **SW only pairs the specific actions which "see" each other**
- **A volatile write to x synchronizes with subsequent read of x (subsequent in SO)**
- **The transitive closure of PO and SW forms HB**
- **HB consistency: When reading a variable, we see either the last write (in HB) or any other unordered write.**
  - This means races are allowed!

## The Java Memory Model – a legal view

- **Memory models provide (often minimal) guarantees for visibility of memory operations**
  - Contract between programmer, compiler, architecture about semantics
  - Details are far from trivial – cf. Steuergesetz Kanton Zurich

    *Yet, if one wants to really understand an example – it's the reference!*
  - For our purposes, remember volatile and synchronized()

    *Roughly: Memory operations will not be reordered with respect to accesses to volatile variables or synchronized blocks.*



§ 1.[7] [1] Das kantonale Steueramt vollzieht das Erbschafts- und Schenkungssteuergesetz (ESchG) vom 28. September 1986[4], soweit nachfolgend nichts Abweichendes geregelt ist.

[2] Die Finanzdirektion entscheidet über Rekurse gemäss §§ 61 Abs. 2 und 64 Abs. 2 ESchG[4].

§ 2.[6] Es gelten sinngemäss:

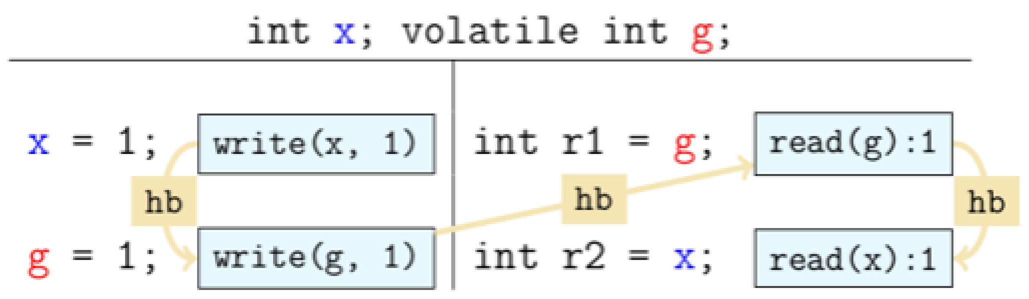
a. § 119 des Steuergesetzes[2] über den Ausstand,

b. §§ 2–15, § 21 Abs. 1 und 2 sowie §§ 23–25 der Verordnung zum Steuergesetz[3].

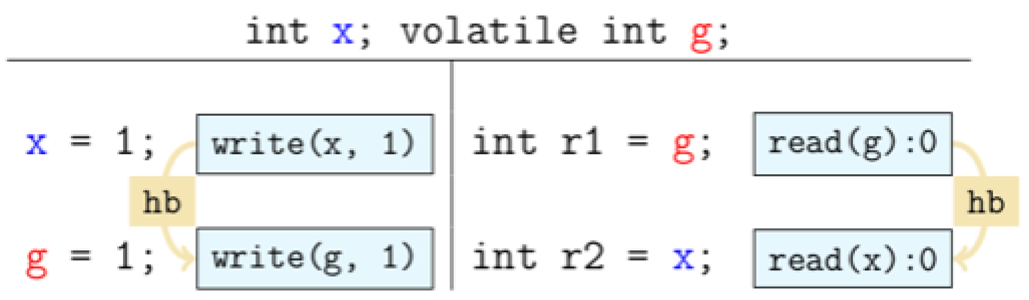
- **We should still be able to understand the laws of the memory model – thus quick repetition**
  - No worry, you will do this yourself in exercises

  - Program order – order in which statements are executed (of course, meaning the actions resulting from statements!)
  - Synchronization order – order of synchronzing memory actions (in the same thread)!
  - Synchronizes with – order of observed synchronizing memory actions across threads
  - Happens before – the union (transitive closure) of PO and SW

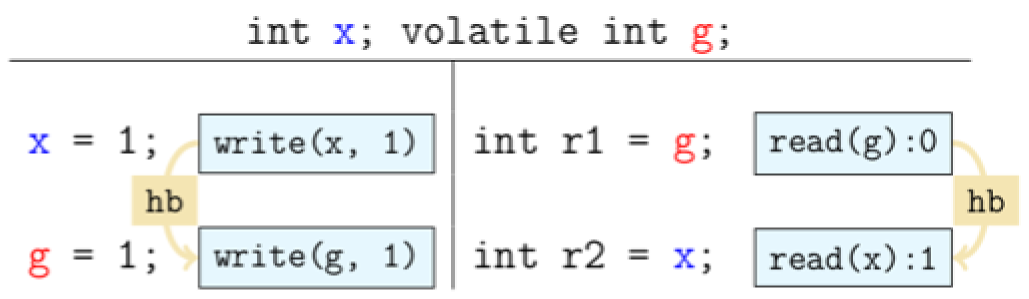
## Examples



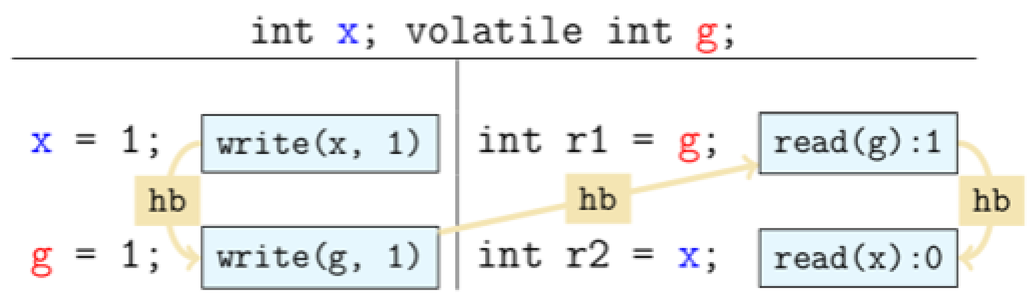
Case 1: HB consistent, observe the latest write in $\xrightarrow{hb}$
$(r1, r2) = (1, 1)$



Case 2: HB consistent, observe the default value
$(r1, r2) = (0, 0)$



Case 3: HB consistent (!), reading via race!
$(r1, r2) = (0, 1)$



Case 4: HB **in**consistent, execution can be thrown away

## Behind Locks
Implementation of Mutual Exclusion

## Assumptions

Will make «atomic» more precise today.

**In the following we assume**

1) **atomic reads and writes of variables of primitive type**
2) **no reordering of read and write sequences (! not true in practice ! here for simplicity !)**
3) **threads entering a critical section will leave it eventually**

**Otherwise we assume a multithreaded environment where processes can arbitrarily interleave.**

**We make no assumptions for progress in non-critical section (i.e., threads may deadlock outside of a CS)!**

## Critical sections

**Pieces of code with the following conditions**

1. **Mutual exclusion: statements from critical sections of two or more processes must not be interleaved**
2. **Freedom from deadlock: if some processes are trying to enter a critical section then one of them must eventually succeed**
3. **Freedom from starvation: if *any* process tries to enter its critical section, then that process must eventually succeed**

According to M. Ben Ari, Principles of Concurrent and Distributed Programming

## Critical section problem

global (shared) variables

Easy to implement on a single-core machine. How?

```
Process P
local variables
loop
        non-critical section
        preprotocol
        critical section
        postprotocol
```

```
Process Q
local variables
loop
        non-critical section
        preprotocol
        critical section
        postprotocol
```

## Easy to implement on a single core system …

**global (shared) variables**

```
Process P
local variables
loop
        non-critical section
        Switch off IRQs
        critical section
        Switch on IRQs
```

```
Process Q
local variables
loop
        non-critical section
        Switch off IRQs
        critical section
        Switch on IRQs
```

## Mutual exclusion for 2 processes -- 1st Try

```
volatile boolean wantp=false, wantq=false
```

Do you see the problem?

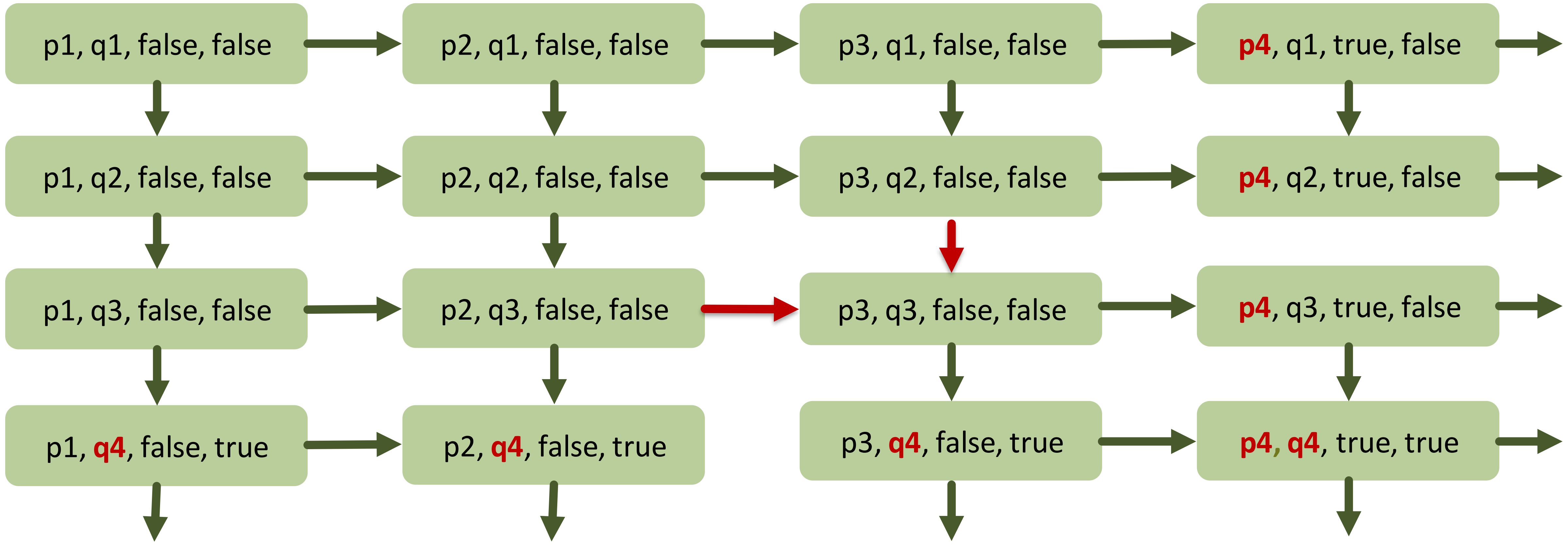
```
Process P
local variables
loop
p1    non-critical section
p2    while(wantq);
p3    wantp = true
p4    critical section
p5    wantp = false
```

```
Process Q
local variables
loop
q1    non-critical section
q2    while(wantp);
q3    wantq = true
q4    critical section
q5    wantq = false
```

| | |
|---|---|
| p1 | non-critical section |
| p2 | while (wantq); |
| p3 | wantp = true |
| p4 | critical section |
| p5 | wantp = false |

## State space diagram [p, q, wantp, wantq]

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| non-critical section | while(wantp) while(wantq) | wantp = true wantq = true | critical section | wantp = false wantq = false |



$$\begin{tikzcd}
\text{p1, q1, false, false} \arrow[r] \arrow[d] & \text{p2, q1, false, false} \arrow[r] \arrow[d] & \text{p3, q1, false, false} \arrow[r] \arrow[d] & \text{p4, q1, true, false} \arrow[r] \arrow[d] & {} \\
\text{p1, q2, false, false} \arrow[r] \arrow[d] & \text{p2, q2, false, false} \arrow[r] \arrow[d] & \text{p3, q2, false, false} \arrow[r] \arrow[d] & \text{p4, q2, true, false} \arrow[r] & {} \\
\text{p1, q3, false, false} \arrow[r] \arrow[d] & \text{p2, q3, false, false} \arrow[r] \arrow[d] & \text{p3, q3, false, false} \arrow[r] \arrow[d] & \text{p4, q3, true, false} \arrow[r] \arrow[d] & {} \\
\text{p1, q4, false, true} \arrow[r] \arrow[d] & \text{p2, q4, false, true} \arrow[d] & \text{p3, q4, false, true} \arrow[r] \arrow[d] & \text{p4, q4, true, true} \arrow[r] \arrow[d] & {} \\
{} & {} & {} & {} &
\end{tikzcd}$$

**no mutual exclusion !**

## Observation: state space diagram too large

```
volatile bool
```

Only of interest: state transitions of the protocol.
p1/q1 is identical to p2/q2 – call state 2
p4/q4 is identical to p5/q5 – call state 5
**Then forbidden: both processes in state 5**

**Process P**
**local variables**
**loop**

| | |
|---|---|
| **p1** | **non-critical section** |
| **p2** | **while(wantq);** |
| **p3** | **wantp = true** |
| **p4** | **critical section** |
| **p5** | **wantp = false** |

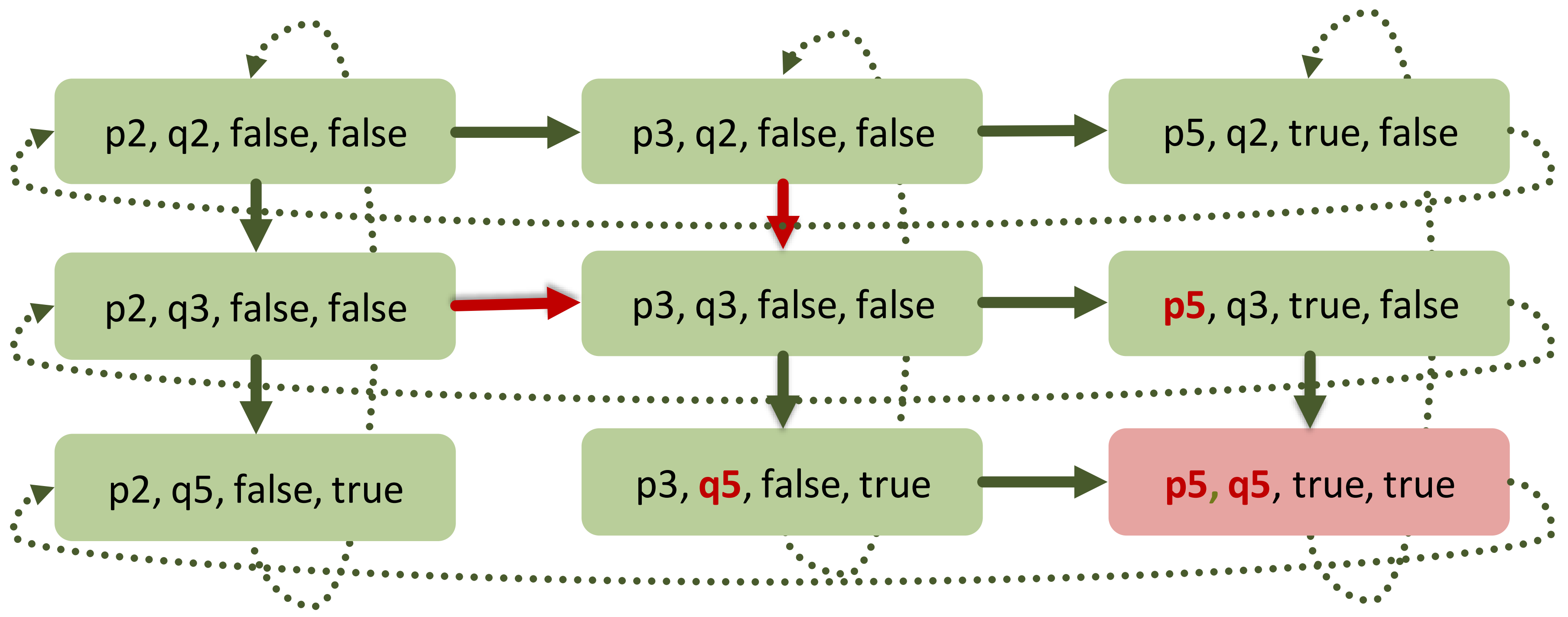
**loop**

| | |
|---|---|
| **q1** | **non-critical section** |
| **q2** | **while(wantp);** |
| **q3** | **wantq = true** |
| **q4** | **critical section** |
| **q5** | **wantq = false** |

## Reduced state space diagram [p, q, wantp, wantq] – only states 2, 3, and 5

1 non-critical section 2 await wantq == false / await wantp == false 3 wantp = true / wantq = true 4 critical section 5 wantp = false / wantq = false

**All of interest covered:**



| | |
|---|---|
| p1 | non-critical section |
| p2 | while (wantq); |
| p3 | wantp = true |
| p4 | critical section |
| p5 | wantp = false |

no mutual exclusion !

## Mutual exclusion for 2 processes -- 2nd Try

```
volatile boolean wantp=false, wantq=false
```

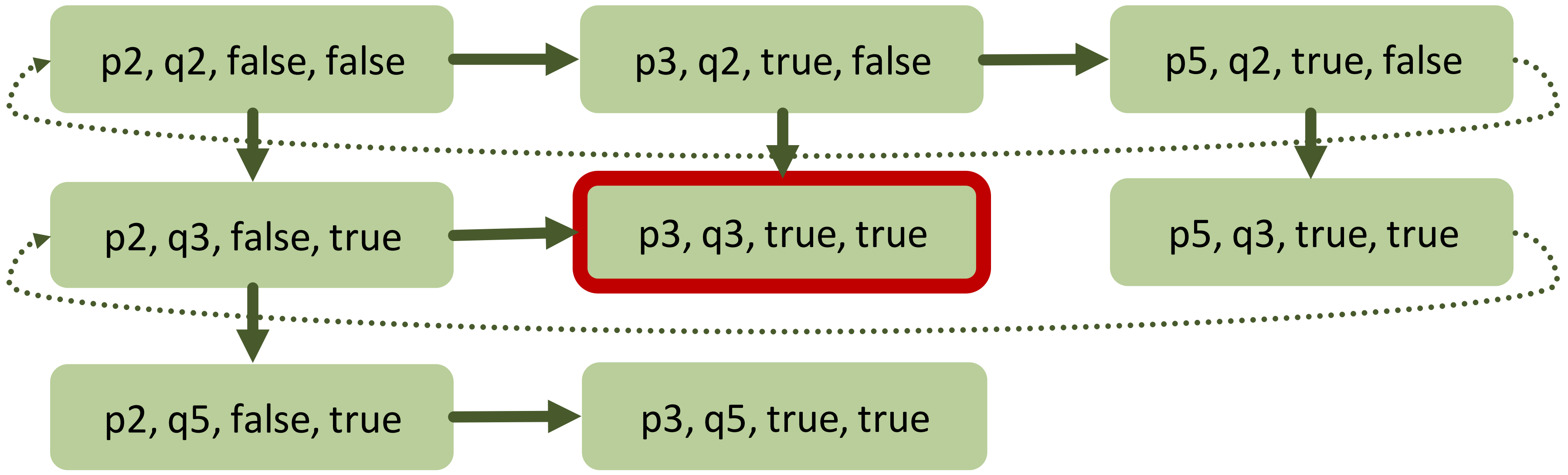
Do you see the problem?

```
Process P
local variables
loop
p1      non-critical section
p2      wantp = true
p3      while(wantq);
p4      critical section
p5      wantp = false
```

```
Process Q
local variables
loop
q1      non-critical section
q2      wantq = true
q3      while(wantp):
q4      critical section
q5      wantq = false
```

## State space diagram [p, q, wantp, wantq]

| | | | | |
|---|---|---|---|---|
| **1** non-critical section | **2** wantp = true<br>wantq = true | **3** while(wantp)<br>while(wantq) | **4** critical section | **5** wantp = false<br>wantq = false |



deadlock !

## Mutual exclusion for 2 processes -- 3rd Try

```
volatile int turn = 1;
```

**Process P**

**local variables**

**loop**

```
p1    non-critical section
p2    while(turn != 1);
p3    critical section
p4    turn = 2
```

**Process Q**

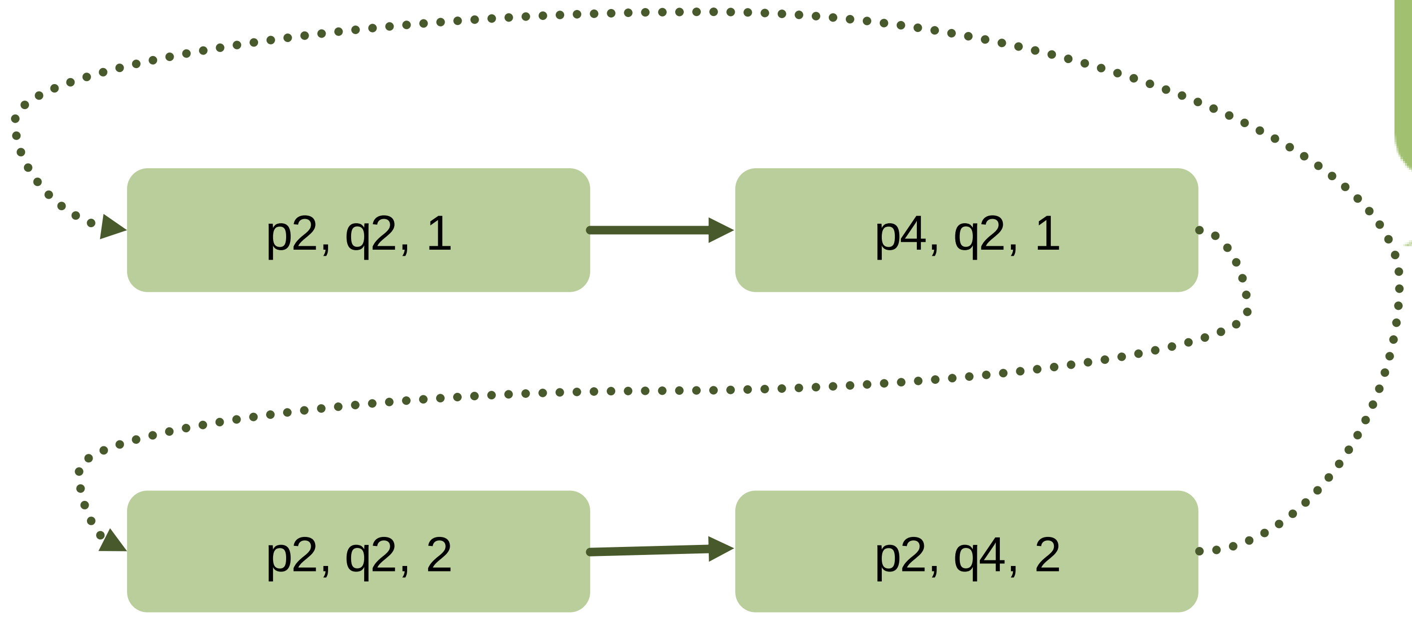
**local variables**

**loop**

```
q1    non-critical section
q2    while(turn != 2);
q3    critical section
q4    turn = 1
```

Do you see the problem?

## State space diagram [p, q, turn]



We have not made any assumptions about progress outside of the CS...

starvation!

## A combination of the tries 2 and 3: Decker's Algorithm

volatile boolean wantp=false, wantq=false, integer turn= 1

```
Process P
loop
    non-critical section
    wantp = true
    while (wantq) {
        if (turn == 2) {
            wantp = false;
            while(turn != 1);
            wantp = true; }}
    critical section
    turn = 2
    wantp = false
```

only when q tries to get lock

and q has preference

let q proceed

and wait

and try again

```
Process Q
loop
    non-critical section
    wantq = true
    while (wantp) {
        if (turn == 1) {
            wantq = false
            while(turn != 2);
            wantq = true; }}
    critical section
    turn = 1
    wantq = false
```

## More concise than Decker: Peterson Lock

```
let P=1, Q=2; volatile boolean array flag[1..2] = [false, false];
volatile integer victim = 1
```

**Process P (1)**

```
loop
    non-critical section
    flag[P] = true
    victim = P
    while(flag[Q] && victim == P);
    critical section
    flag[P] = false
```

I am interested

but you go first

We both are interested

And you go first

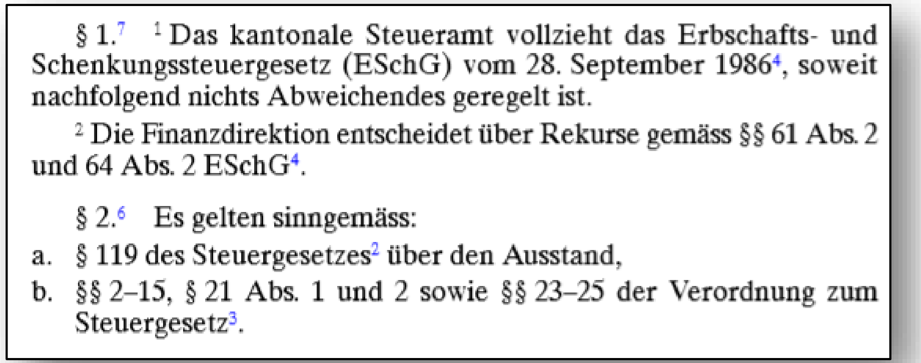
**Process Q (2)**

```
loop
    non-critical section
    flag[Q] = true
    victim = Q
    while(flag[P] && victim == Q);
    critical section
    flag[Q] = false
```

## We want to prove ...

**that the Peterson Lock satisfies mutual exclusion**

**and that it is starvation free**



§ 1.[7] [1] Das kantonale Steueramt vollzieht das Erbschafts- und Schenkungssteuergesetz (ESchG) vom 28. September 1986[4], soweit nachfolgend nichts Abweichendes geregelt ist.

[2] Die Finanzdirektion entscheidet über Rekurse gemäss §§ 61 Abs. 2 und 64 Abs. 2 ESchG[4].

§ 2.[6] Es gelten sinngemäss:

a. § 119 des Steuergesetzes[2] über den Ausstand,

b. §§ 2–15, § 21 Abs. 1 und 2 sowie §§ 23–25 der Verordnung zum Steuergesetz[3].

**How?**

**Requires some notation first.**

## Events and precedence

**Threads produce a sequence of events**

P produces events $\boldsymbol{p_0, p_1, \ldots}$

**e.g.,** $p_1$ = "flag[P] = true"

programs usually consist of loops, therefore we might need to count occurences

**j-th occurence of event i in thread P:** $\boldsymbol{p_i^j}$

**e.g.,** $p_5^3$ **= "flag[P] = false" in the third iteration**

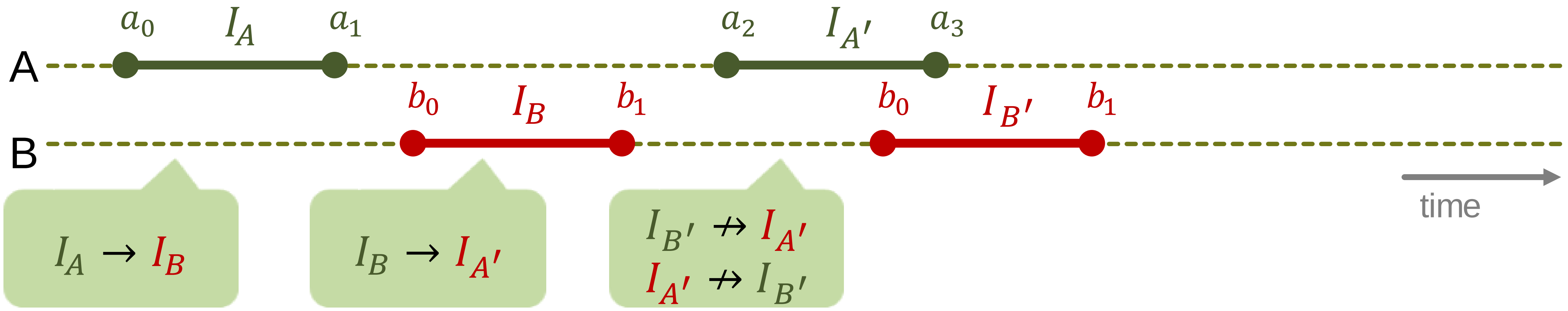
Precedence relation: we write $\boldsymbol{a \rightarrow b}$ when a occurs before b.

Note that the precedence relation "→" is a total order for events.

## Intervals

$(a_0, a_1)$: **interval of events** $a_0, a_1$ **with** $a_0 \rightarrow a_1$

**With** $I_A = (a_0, a_1)$ **and** $I_B = (b_0, b_1)$ **we write** $\boldsymbol{I_A \rightarrow I_B}$ **if** $\boldsymbol{a_1 \rightarrow b_0}$



we say "$I_A$ ***precedes*** $I_B$" **and** "$I_{B'}$ **and** $I_{A'}$ **are** ***concurrent***"

## Atomic register

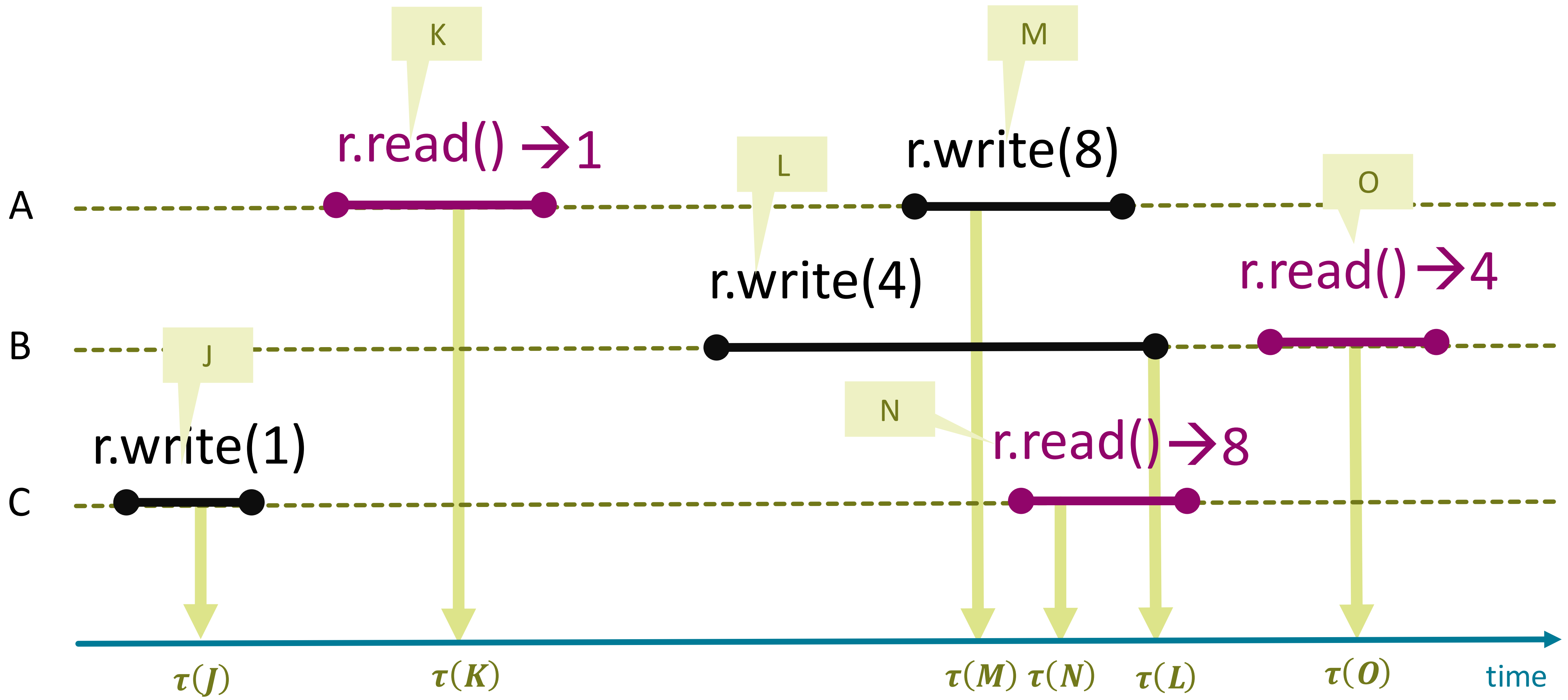
Register: basic memory object, can be shared or not
i.e., in this context register $\neq$ register of a CPU

Register *r* : operations *r.read()* and *r.write(v)*

Atomic Register:

- An invocation *J* of *r.read* or *r.write* takes effect at a single point $\tau(J)$ in time
- $\tau(J)$ always lies between start and end of the operation *J*
- Two operations *J* and *K* on the same register always have a different effect time $\tau(J) \neq \tau(K)$
- An invocation *J* of *r.read()* returns the value *v* written by the invocation *K* of *r.write(v)* with closest preceding effect time $\tau(K)$

## Example


K
M
L
O
J
N
r.read() →1
r.write(8)
r.write(4)
r.read() →4
r.write(1)
r.read() →8
A
B
C
$\tau(J)$
$\tau(K)$
$\tau(M)$
$\tau(N)$
$\tau(L)$
$\tau(O)$
time

## Atomic register

**Assumptions for Atomic Registers justify to treat operations on them as events taking place at a single point in time.**

**We will use this in the following proofs.**

**Note that even with atomic registers there can still be non-determinism of programs because nothing is said about the order of effect times for concurrent operations.**

```
flag[P] = true
victim = P
while (flag[Q] && victim == P){}
CSP
flag[P] = false
```

## Proof: Mutual exclusion (Peterson)

**By contradiction: assume concurrent $CS_P$ and $CS_Q$ [A]**

**Assume without loss of generality:**

$$W_Q(\text{victim=Q}) \rightarrow W_P(\text{victim=P}) \quad [B]$$

A + C ⇒ must read false

B ⇒ must read P [C]

**From the code:**

$$\begin{tikzcd}
W_P(\text{flag[P]=true}) \arrow[r] & W_P(\text{victim = P}) \arrow[r] & R_P(\text{flag[Q]}) \arrow[r] & R_P(\text{victim}) \arrow[r] & CS_P \\
W_Q(\text{flag[Q]=true}) \arrow[r] & W_Q(\text{victim = Q}) \arrow[r] \arrow[u] & R_Q(\text{flag[P]}) \arrow[r] & R_Q(\text{victim}) \arrow[r] & CS_Q
\end{tikzcd}$$

"write of P"

transitivity of "→ "
⇒ must read true ϟ

"read of Q"

## Proof: Freedom from starvation

```
flag[P] = true
victim = P
while (flag[Q] && victim == P){}
CSP
flag[P] = false
```

**By (exhaustive) contradition**

**Assume without loss of generality that P runs forever in its lock loop, waiting until `flag[Q]==false` or `victim != P`.**

**Possibilities for Q:**

**stuck in nonCS**

⇒ flag[Q] = false and P can continue. Contradiction.

**repeatedly entering and leaving its CS**

⇒ sets victim to Q when entering.

Now victim cannot be changed ⇒ P can continue. Contradiction.

**stuck in its lock loop waiting until `flag[P]==false` or `victim != Q`.**

But `victim == P` and `victim == Q` cannot hold at the same time. Contradiction.

## Peterson in Java

```
class PetersonLock
{
        volatile boolean flag[] = new boolean[2];
        volatile int victim;

        public void Acquire(int id)
        {
                flag[id] = true;
                victim = id;
                while (flag[1-id] && victim == id);
        }

        public void Release(int id)
        {
                flag[id] = false;
        }
}
```

Volatile reference to an array and not an array of volatile variables!
This example may work in practice.
However, for a correct program we need to use to use Java's **AtomicInteger** and **AtomicIntegerArray**.

## The Filter Lock

Extension of Peterson's lock to n processes

Every thread *t* knows its level in the filter *level[t]*

In order to enter CS, a thread has to elevate all levels.

For each level, we use Peterson's mechanism to filter at most one thread, if other threads are at higher level.

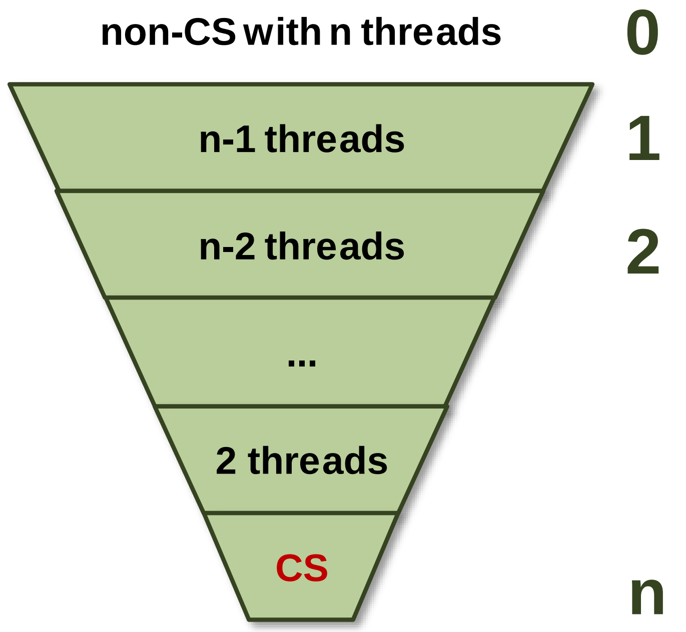
For every level *l* there is one victim *victim[l]* that has to let others pass in case of conflicts.

## The Filter Lock

```
int[] level(#threads), int[] victim(#threads)



lock(me) {
   for (int i=1; i<n; ++i) {
      level[me] = i;
      victim[i] = me;
      while (∃k ≠ me: level[k] >= i && victim[i] == me) {};
   }
}

unlock(me) {
   level[me] = 0;
}
```

Other threads are at same or higher level

And I have to wait



## FilterLock in Java

```
import java.util.concurrent.atomic.AtomicIntegerArray;
class FilterLock{
   AtomicIntegerArray level;
   AtomicIntegerArray victim;
   volatile int n;

   FilterLock(int n) {
      this.n = n;
      level = new AtomicIntegerArray(n);
      victim = new AtomicIntegerArray(n);
   }
   ...
```

## FilterLock in Java

```
...
    // ∃k ≠ me: level[k] >= i (lev)
    boolean Others(int me, int lev) {
        for (int k = 0; k < n; ++k)
            if (k != me && level.get(k) >= lev) return true;
        return false;
    }
    public void Acquire(int me) {
        for (int lev = 1; lev < n; ++lev) {
            level.set(me, lev);
            victim.set(lev, me);
            while(me == victim.get(lev) && Others(me,lev));
        }
    }
    public void Release(int me) {
        level.set(me, 0);
    }
}
```

Again: I (as a thread) can make progress if
(a) Another thread wants to enter my level or
(b) No more threads are in front of me
This works because there are at most n threads in the system.

## Fairness

**Divide lock implementation (preprotocol) into two parts**

- **doorway interval $D$: finite number of steps**
- **waiting interval $W$: unbounded number of steps**

**A lock algorithm is first-come-first-served when for two processes A and B it holds that**

If $D_A^j \rightarrow D_B^k$ **then** $CS_A^j \rightarrow CS_B^k$

## The Filter Lock

satisfies mutual exclusion

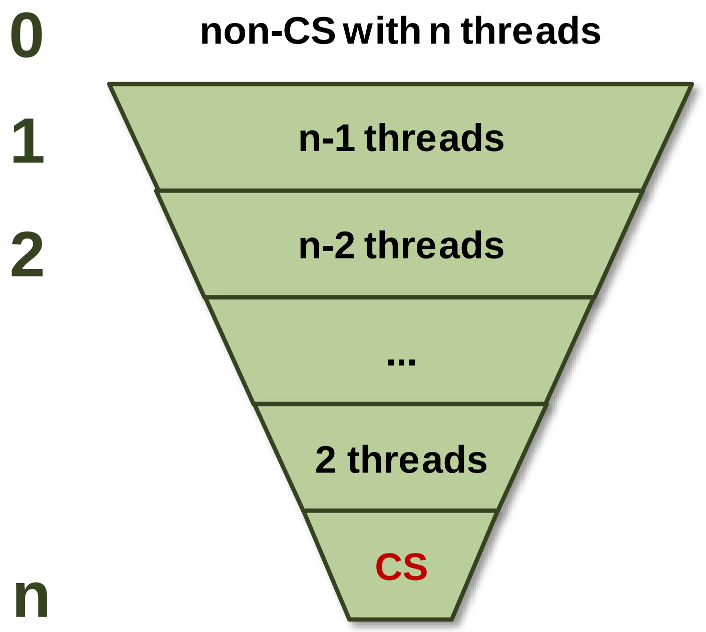
is deadlock free (how to prove?)

is starvation free (how to prove?)

but: is it also fair?

no: the filter lock is not first-come-first-serve

What else is bad about this lock?



## A small detour: Safe and Regular Registers

**Question:**

- **Is it possible to construct mutual exclusion with non-atomic registers?**

**Surprisingly: yes**

- **It is possible with registers fulfilling the weakest possible conditions that appear to be still useful in a concurrent setup.**

## Safe SWMR Register

**Register r: basic memory object, can be shared or not,**
**operations *r.read()* and *r.write(v)*.**

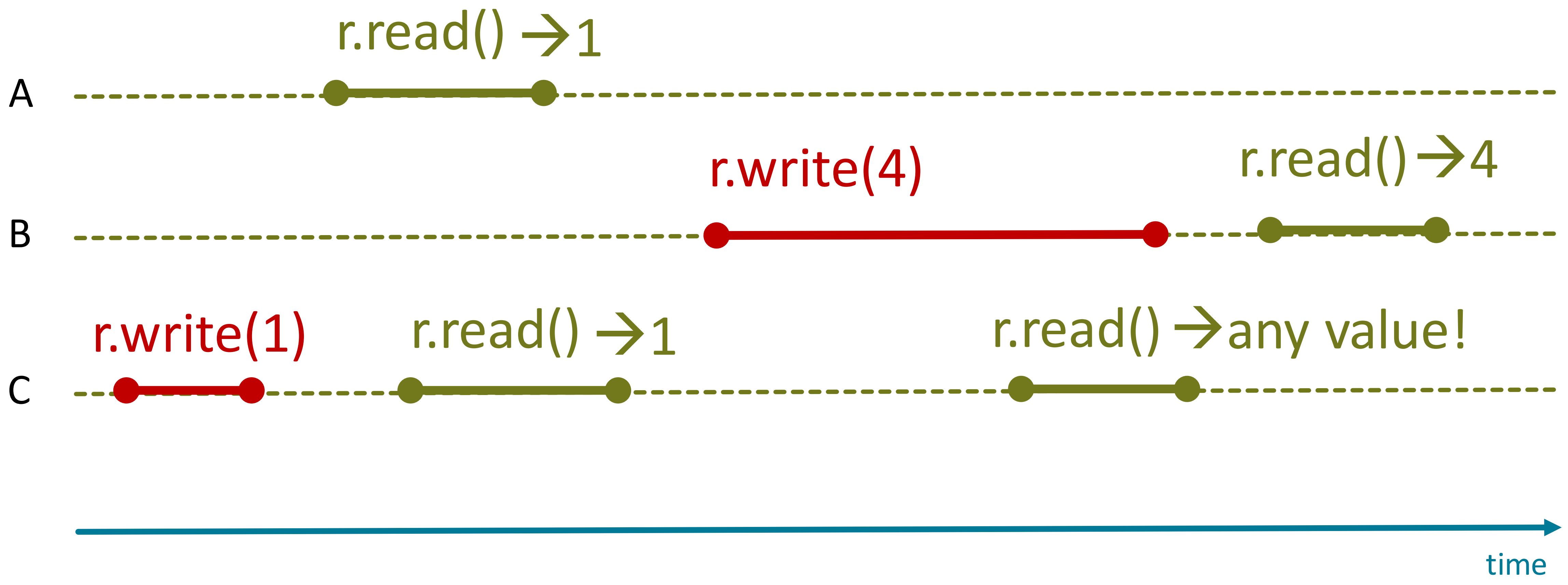
**SWMR (Single Writer Multiple Reader): only one concurrent write but multiple concurrent reads allowed.**

***Safe* Register**

- **any read not concurrent with a write returns the current value of r**
- **any read concurrent with a write can return *any value* of the domain of r**
  **if any read concurrent with writes can only return a value of one of the values (previous, new) then the register is called *regular***

The notion "safe" is historically motivated but actually misleading.

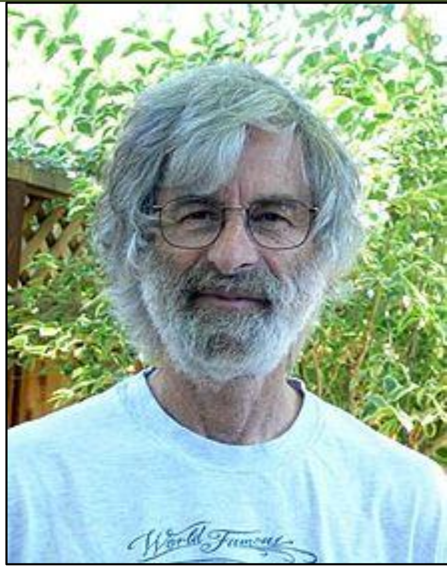
## Example


r.read() →1
A
r.write(4)
r.read() →4
B
r.write(1)
r.read() →1
r.read() →any value!
C
time

## Mutual Exclusion for n processes: Bakery Algorithm (1974)

A process is required to take a numbered ticket with value greater than all outstanding tickets

CS Entry: Wait until ticket number is lowest


Lamport, Turing award 2013

## Bakery algorithm (two processes, simplified)

```
volatile int np = 0, nq = 0
```

```
Process P
loop
    non-critical section
    np = nq + 1
    while (nq != 0 && nq < np);
    critical section
    np = 0
```

Q also wants access

and Q has an earlier ticket

```
Process Q
loop
    non-critical section
    nq = np + 1
    while (np != 0 && np <= nq);
    critical section
    nq = 0
```

np == nq can happen
➔ global ordering of processes

## Bakery algorithm (n processes)

```
integer array[0..n-1] label = [0,...,0]
boolean array[0..n-1] flag =  [false, ..., false]
```

SWMR «ticket number»

SWMR «I want the lock»

```
lock(me):
   flag[me] = true;
   label[me] = max(label[0], ... , label[n-1]) + 1;
   while (∃k ≠ me: flag[k] && (k,label[k]) <_l (me,label[me])) {};

unlock(me):
   flag[me] = false;
```

$(k, l_k) <_l (j, l_j) \Leftrightarrow l_k < l_j \text{ or } (l_k = l_j \text{ and } k < j)$

## Bakery Lock in Java

Nice lock! But which problem remains?

```
class BakeryLock
{
    AtomicIntegerArray flag;
    // there is no AtomicBooleanArray
    AtomicIntegerArray label;
    final int n;

    BakeryLock(int n) {
        this.n = n;
        flag = new AtomicIntegerArray(n);
        label = new AtomicIntegerArray(n);
    }

    int MaxLabel() {
        int max = label.get(0);
        for (int i = 1; i<n; ++i)
            max = Math.max(max, label.get(i));
        return max;
    }
    ...
```

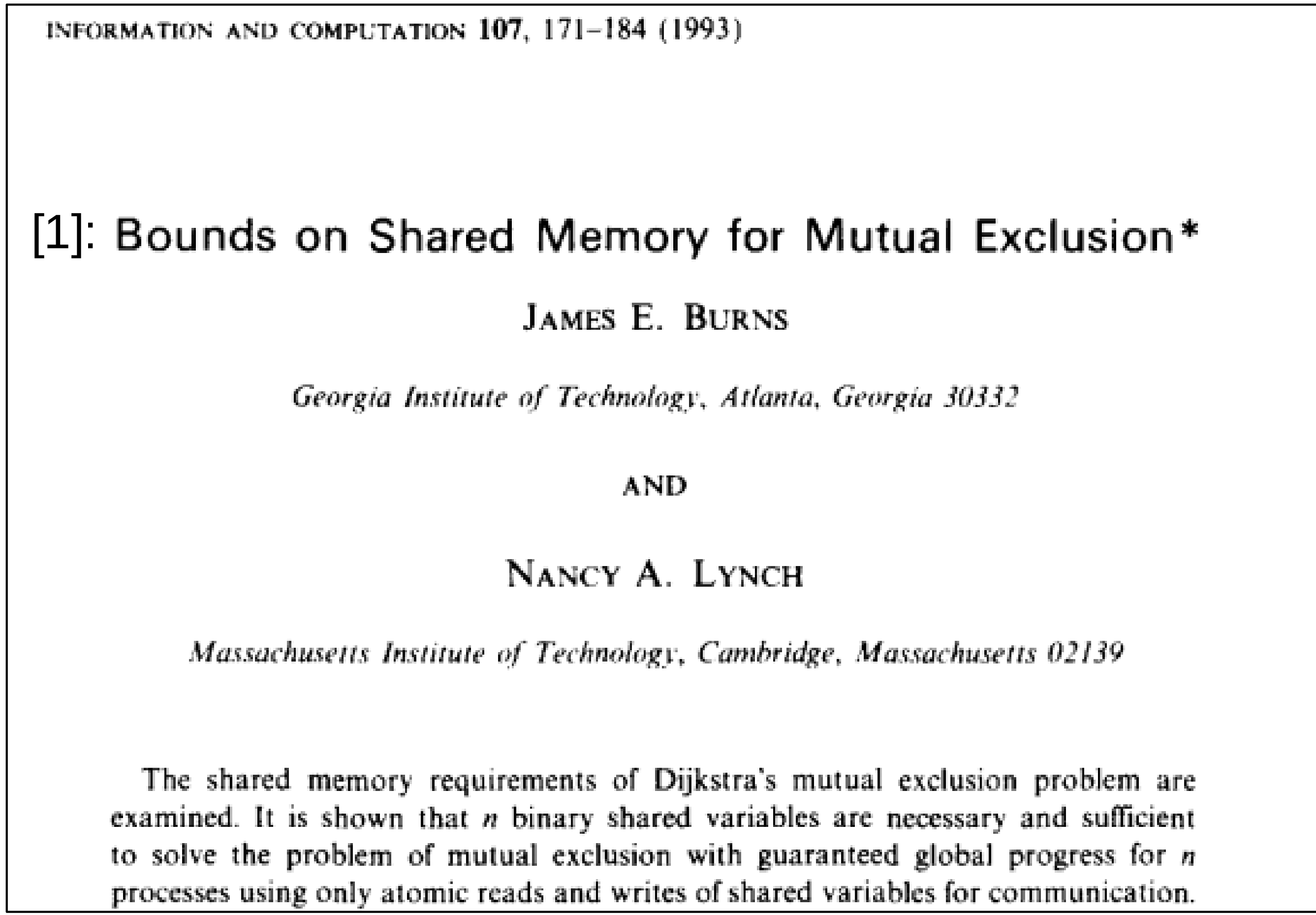
```
    boolean Conflict(int me) {
        for (int i = 0; i < n; ++i)
            if (i != me && flag.get(i) != 0) {
                int diff = label.get(i) - label.get(me);
                if (diff < 0 || diff == 0 && i < me)
                    return true;
            }
        return false;
    }

    public void Acquire(int me) {
        flag.set(me,1);
        label.set(me, MaxLabel() + 1);
        while(Conflict(me));
    }

    public void Release(int me) {
        flag.set(me, 0);
    }
}
```

## In general

**Shared memory locations come in different variants**

- **Multi-Reader-Single-Writer (flag[])**
- **Multi-Reader-Multi-Writer (victim[])**

- **Theorem 5.1 in [1]: *"If S is a [atomic] read/write system with at least two processes and S solves mutual exclusion with global progress [deadlock-freedom], then S must have at least as many variables as processes"***


INFORMATION AND COMPUTATION 107, 171–184 (1993)

[1]: Bounds on Shared Memory for Mutual Exclusion*

JAMES E. BURNS

*Georgia Institute of Technology, Atlanta, Georgia 30332*

AND

NANCY A. LYNCH

*Massachusetts Institute of Technology, Cambridge, Massachusetts 02139*

The shared memory requirements of Dijkstra's mutual exclusion problem are examined. It is shown that $n$ binary shared variables are necessary and sufficient to solve the problem of mutual exclusion with guaranteed global progress for $n$ processes using only atomic reads and writes of shared variables for communication.

## We have constructed something …

**… that may not quite fulfil its purpose!**

**AND we cannot do better, can we?**

- **Is mutual exclusion really implemented like this?**
  - NO! Why?

    - *space lower bound linear in the number of maximum threads!*

    - *without precautions (volatile variables) our assumptions on memory reordering does not hold. Memory barriers in hardware are expensive.*

    - *algorithms are not wait-free (more later)*

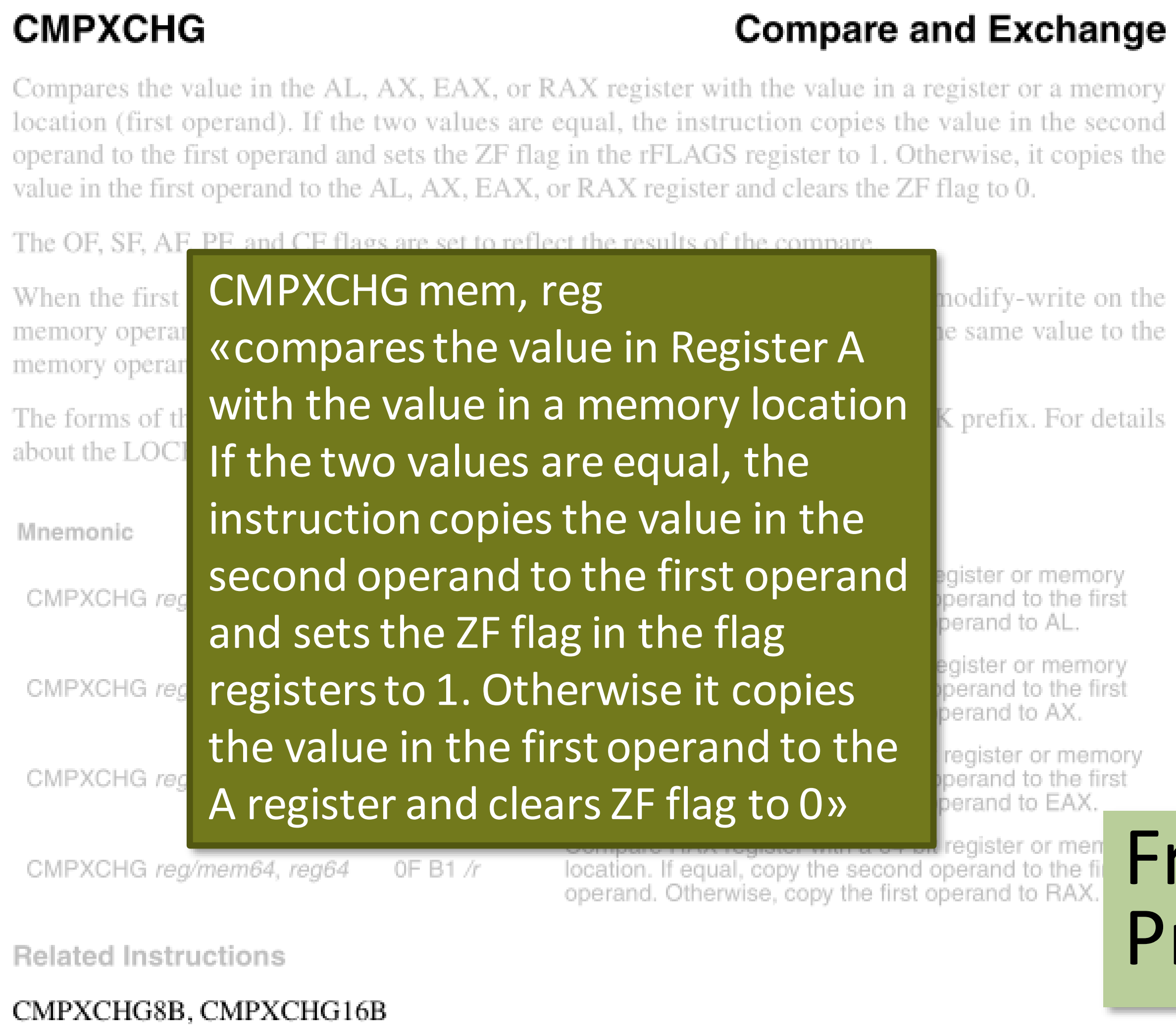
    - *modern multiprocessor architectures provide special instructions for atomically reading and writing at once!*

- **But we proved that we cannot do better. What now!?**
  - Change (extend) the model with architecture engineering!

## Hardware Support for Parallelism

Read-Modify-Write Operations

## Hardware support for atomic operations: Example (x86)



**CMPXCHG** **Compare and Exchange**

Compares the value in the AL, AX, EAX, or RAX register with the value in a register or a memory location (first operand). If the two values are equal, the instruction copies the value in the second operand to the first operand and sets the ZF flag in the rFLAGS register to 1. Otherwise, it copies the value in the first operand to the AL, AX, EAX, or RAX register and clears the ZF flag to 0.

The OF, SF, AF, PF, and CF flags are set to reflect the results of the compare.

When the first … modify-write on the memory operan… e same value to the memory operan…

The forms of th… K prefix. For details about the LOCI…

Mnemonic

CMPXCHG *reg* … egister or memory … perand to the first … perand to AL.

CMPXCHG *reg* … egister or memory … perand to the first … perand to AX.

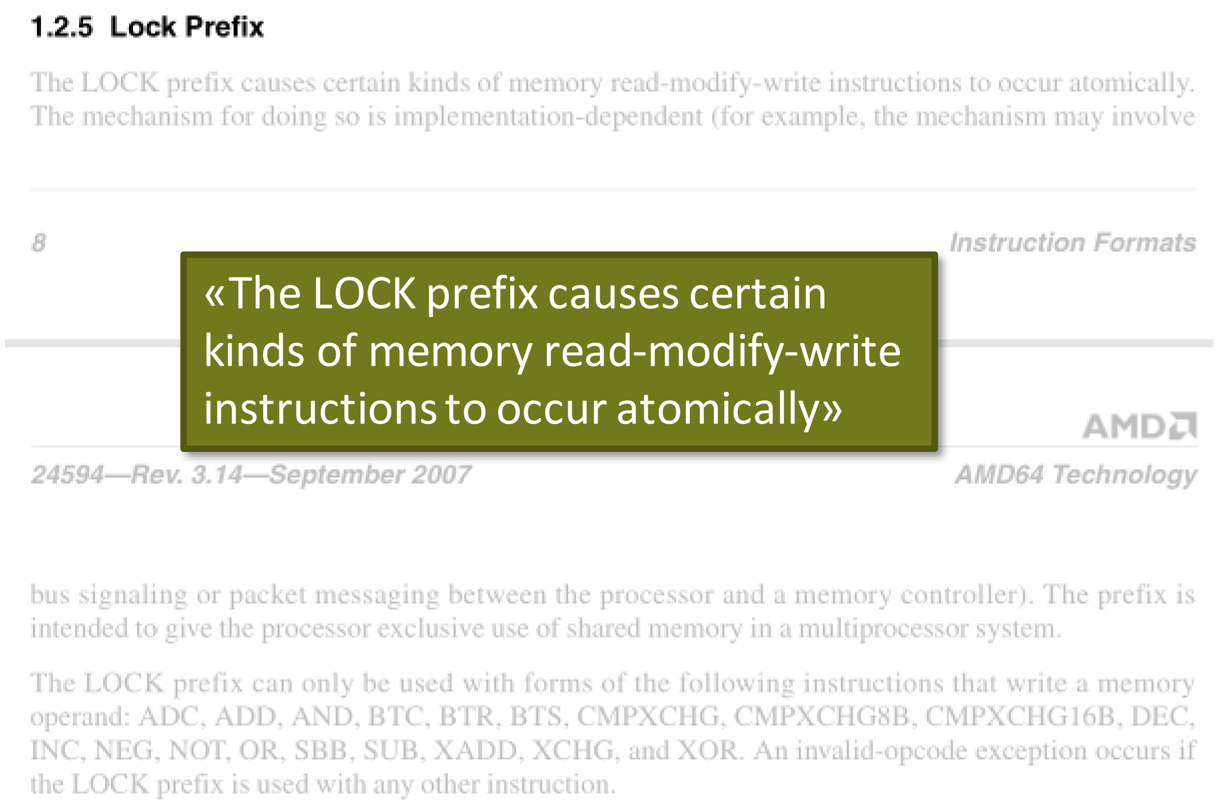
CMPXCHG *reg* … register or memory … perand to the first … perand to EAX.

CMPXCHG *reg/mem64, reg64* 0F B1 /r Compare RAX register with a 64-bit register or mem… location. If equal, copy the second operand to the fi… operand. Otherwise, copy the first operand to RAX.

Related Instructions

CMPXCHG8B, CMPXCHG16B

CMPXCHG mem, reg
«compares the value in Register A with the value in a memory location If the two values are equal, the instruction copies the value in the second operand to the first operand and sets the ZF flag in the flag registers to 1. Otherwise it copies the value in the first operand to the A register and clears ZF flag to 0»



**1.2.5 Lock Prefix**

The LOCK prefix causes certain kinds of memory read-modify-write instructions to occur atomically. The mechanism for doing so is implementation-dependent (for example, the mechanism may involve

8 *Instruction Formats*

AMD

*24594—Rev. 3.14—September 2007* *AMD64 Technology*

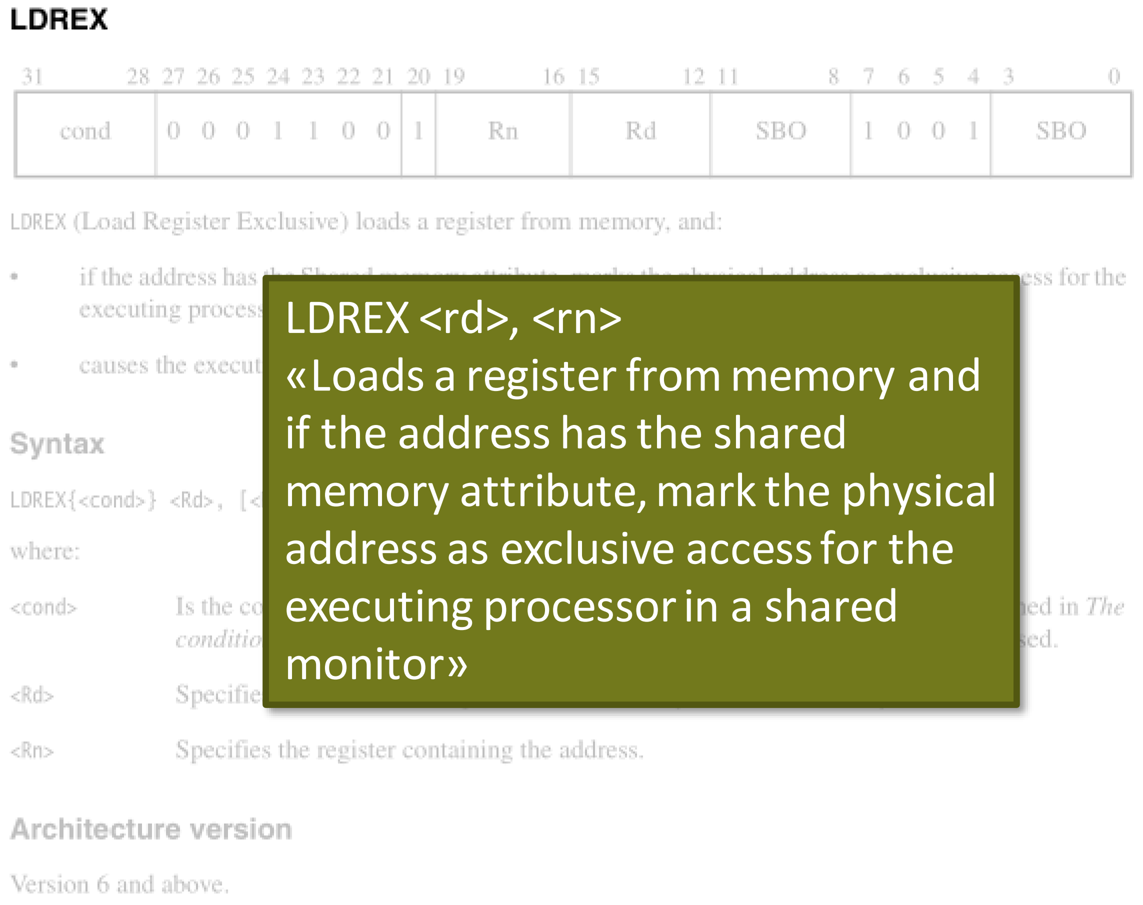
bus signaling or packet messaging between the processor and a memory controller). The prefix is intended to give the processor exclusive use of shared memory in a multiprocessor system.

The LOCK prefix can only be used with forms of the following instructions that write a memory operand: ADC, ADD, AND, BTC, BTR, BTS, CMPXCHG, CMPXCHG8B, CMPXCHG16B, DEC, INC, NEG, NOT, OR, SBB, SUB, XADD, XCHG, and XOR. An invalid-opcode exception occurs if the LOCK prefix is used with any other instruction.

«The LOCK prefix causes certain kinds of memory read-modify-write instructions to occur atomically»

From the AMD64 Architecture Programmer's Manual

## Hardware support for atomic operations: Example (ARM)



**LDREX**

| 31–28 | 27 26 25 24 23 22 21 | 20 | 19–16 | 15–12 | 11–8 | 7 6 5 4 | 3–0 |
|---|---|---|---|---|---|---|---|
| cond | 0 0 0 1 1 0 0 | 1 | Rn | Rd | SBO | 1 0 0 1 | SBO |

LDREX (Load Register Exclusive) loads a register from memory, and:

- if the address has [illegible] ess for the executing process
- causes the execut

**Syntax**

LDREX{<cond>} <Rd>, [<

where:

<cond> Is the co [illegible] ed in *The conditio* [illegible] sed.

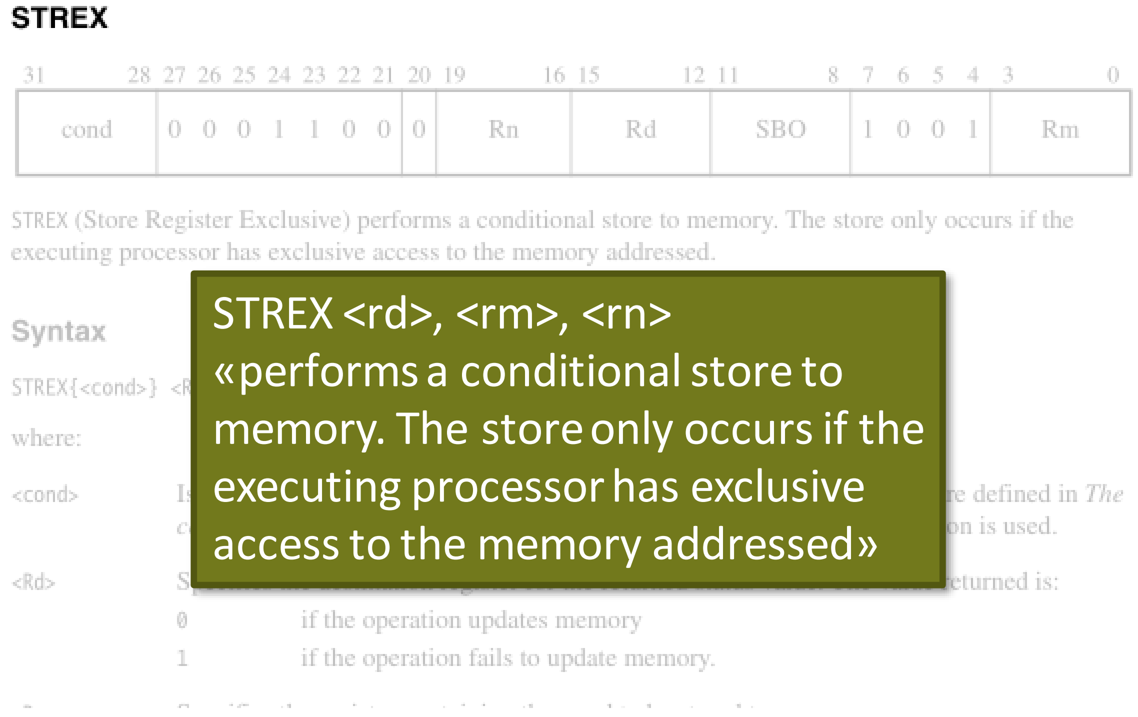
<Rd> Specifie

<Rn> Specifies the register containing the address.

**Architecture version**

Version 6 and above.

LDREX <rd>, <rn>
«Loads a register from memory and if the address has the shared memory attribute, mark the physical address as exclusive access for the executing processor in a shared monitor»



**STREX**

| 31–28 | 27 26 25 24 23 22 21 | 20 | 19–16 | 15–12 | 11–8 | 7 6 5 4 | 3–0 |
|---|---|---|---|---|---|---|---|
| cond | 0 0 0 1 1 0 0 | 0 | Rn | Rd | SBO | 1 0 0 1 | Rm |

STREX (Store Register Exclusive) performs a conditional store to memory. The store only occurs if the executing processor has exclusive access to the memory addressed.

**Syntax**

STREX{<cond>} <R

where:

<cond> Is [illegible] re defined in *The* c [illegible] on is used.

<Rd> S [illegible] eturned is:

| | |
|---|---|
| 0 | if the operation updates memory |
| 1 | if the operation fails to update memory. |

STREX <rd>, <rm>, <rn>
«performs a conditional store to memory. The store only occurs if the executing processor has exclusive access to the memory addressed»

From the ARM Architecture Reference Manual

## Hardware support for atomic operations

**Typical instructions**

**Test-And-Set (TAS)**

Example TSL register, flag (Motorola 68000)

**Compare-And-Swap (CAS)**

Example: LOCK CMPXCHG (Intel x86)

Example: CASA (Sparc)

**Load Linked / Store Conditional**

Example LDREX/STREX (ARM)

Example LL / SC (MIPS, POWER, RISC V)

Atomic instructions are typically much slower than simple read & write operations [1]!

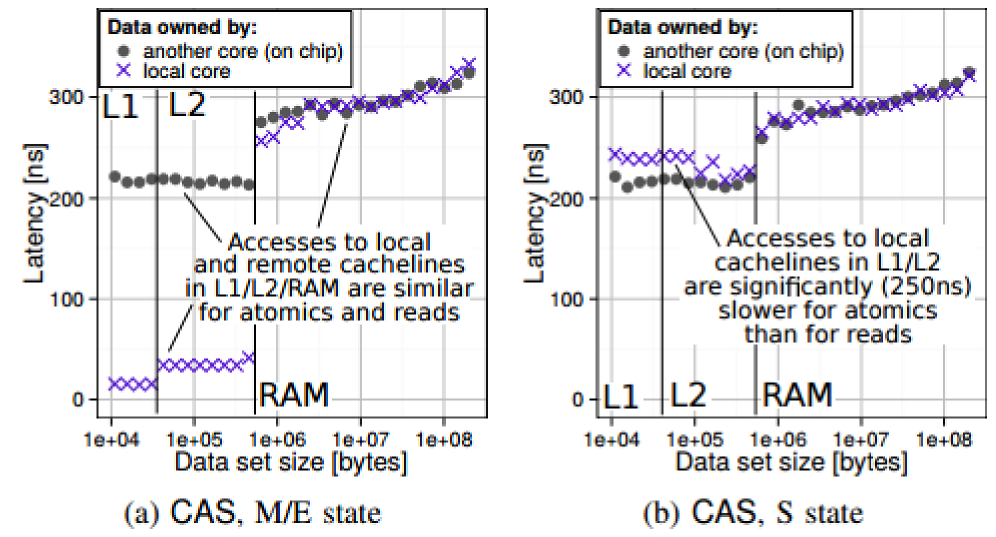


Fig. 6: The comparison of the latency of CAS on Xeon Phi. The requesting core accesses its own cache lines (local) and cache lines of a different core from the same chip (on chip).

## Semantics

```
boolean TAS (memref s)
atomic
    if (mem[s] == 0) {
        mem[s] = 1;
        return true;
    } else
        return false;
```

```
int CAS (memref a, int old, int new)
atomic
    oldval = mem[a];
    if (old == oldval)
        mem[a] = new;
    return oldval;
```

## TAS and CAS

are Read-Modify-Write operations

enable implementation of a mutex with O(1) space
(in contrast to Filter lock, Bakery lock etc.)

are needed for lock-free programming (later in this course)

## Implementation of a spinlock using simple atomic operations

### Test and Set (TAS)

**Init (lock)**

```
    lock = 0;
```

**Acquire (lock)**

```
    while !TAS(lock); // wait
```

**Release (lock)**

```
    lock = 0;
```

### Compare and Swap (CAS)

**Init (lock)**

```
    lock = 0;
```

**Acquire (lock)**

```
    while (CAS(lock, 0, 1) != 0);
```

**Release (lock)**

```
    CAS(lock, 1, 0);
```

**ignore result**

## Read-Modify-Write in Java

## Let's try it.

**Need support for atomic operations on a high level.**

**Available in Java (from JDK 5) with class**

```
        java.util.concurrent.atomic.AtomicBoolean
```

```
Operations
        boolean set();
        boolean get();
        boolean compareAndSet(boolean expect, boolean update);
        boolean getAndSet(boolean newValue);
```

atomically set to value `update` iff current value is `expect`. Return true on success.

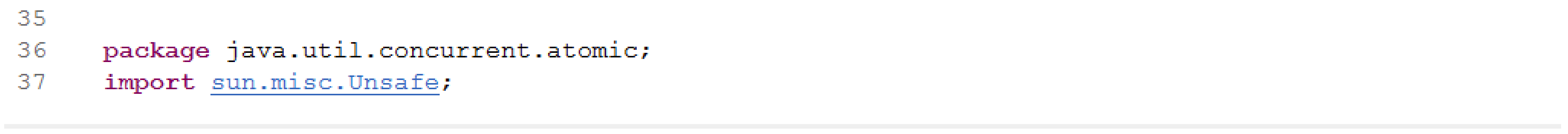
sets `newValue` and returns previous value.

## How does this work?

- The JVM bytecode does not offer atomic operations like CAS.
  [It does, however, support monitors via instructions monitorenter, monitorexit, we will understand this later]
- But there is a (yet undocumented) class `sun.misc.Unsafe` offering direct mappings from java to underlying machine / OS.
- Direct mapping to hardware is not guaranteed –
  operations on AtomicBoolean are not guaranteed lock-free

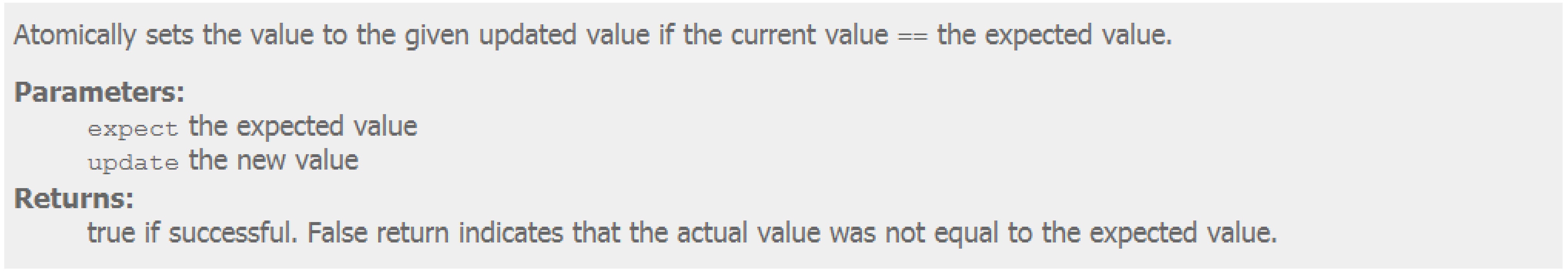
## For experts: java.util.concurrent.atomic.AtomicInteger

**(source: grepcode.com)**



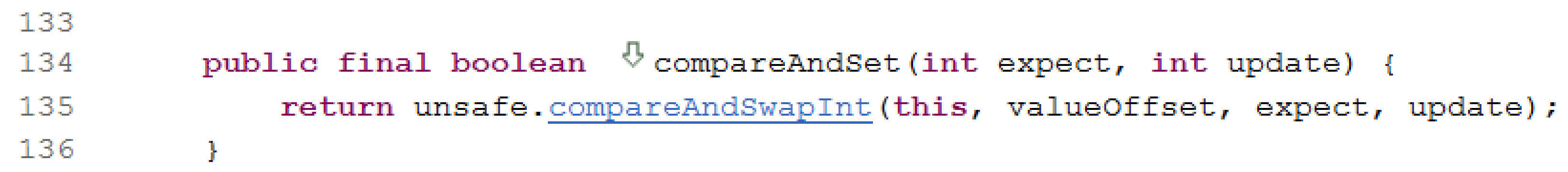
```java

package java.util.concurrent.atomic;
import sun.misc.Unsafe;
```

…



Atomically sets the value to the given updated value if the current value == the expected value.

**Parameters:**
`expect` the expected value
`update` the new value
**Returns:**
true if successful. False return indicates that the actual value was not equal to the expected value.



```java

    public final boolean compareAndSet(int expect, int update) {
        return unsafe.compareAndSwapInt(this, valueOffset, expect, update);
    }
```

## TASLock in Java

```
public class TASLock implements Lock {
    AtomicBoolean state = new AtomicBoolean(false);

    public void lock() {
        while(state.getAndSet(true)) {}
    }

    public void unlock() {
        state.set(false);
    }
    ...
}
```

Spinlock:

Try to get the lock.

Keep trying until the lock is acquired (return value is false).

unlock

release the lock (set to false)

## Measurement

TAS

n = 1, elapsed= 224, normalized= 224

n = 2, elapsed= 719, normalized= 359

n = 3, elapsed= 1914, normalized= 638

n = 4, elapsed= 3373, normalized= 843

n = 5, elapsed= 4330, normalized= 866

n = 6, elapsed= 6075, normalized= 1012

n = 7, elapsed= 8089, normalized= 1155

n = 8, elapsed= 10369, normalized= 1296

n = 16, elapsed= 41051, normalized= 2565

n = 32, elapsed= 156207, normalized= 4881

n = 64, elapsed= 619197, normalized= 9674

- run n threads
- each thread acquires and releases the TASLock a million times
- repeat scenario ten times and add up runtime
- record time per thread

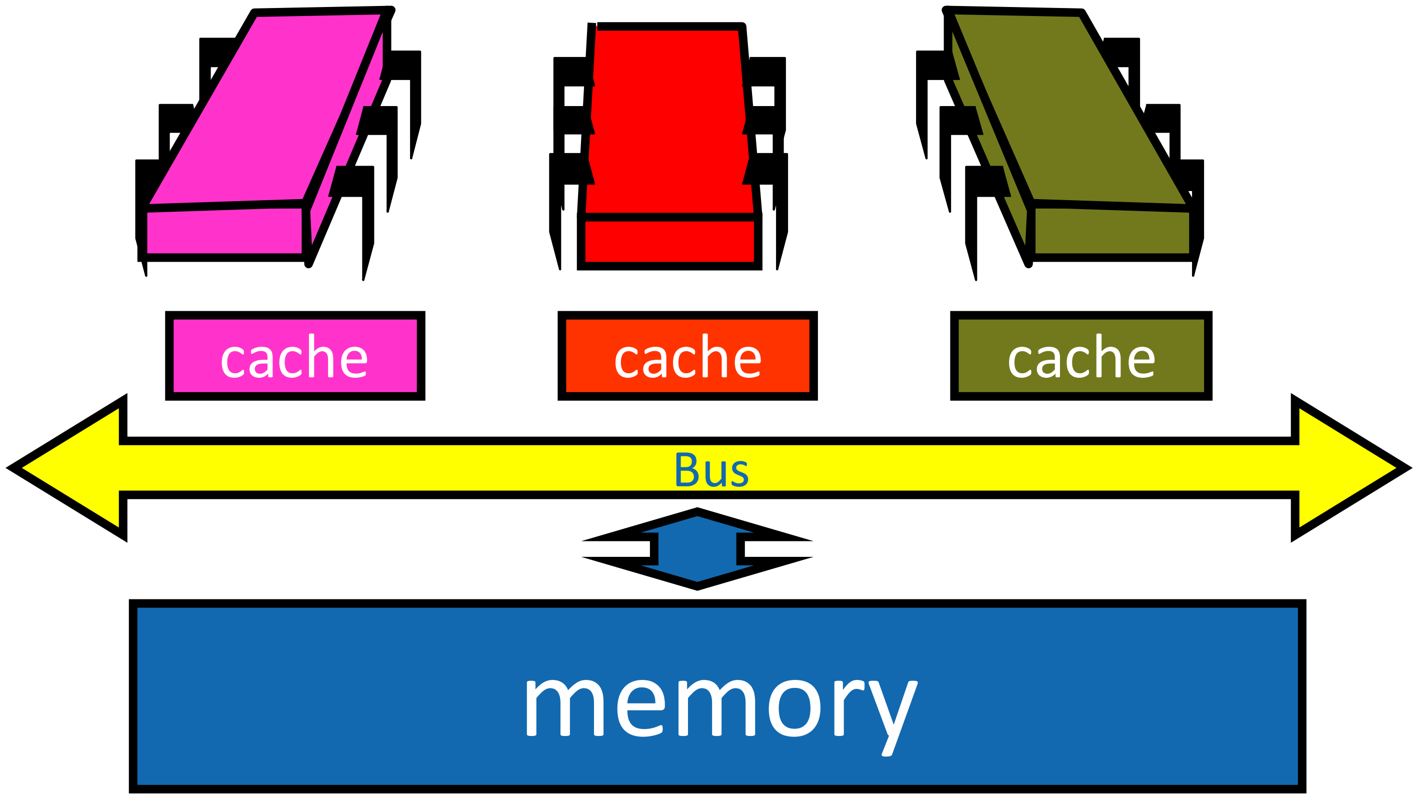
Intel core i7@3.4 GHz, 4 cores + HT

## Why?

sequential bottleneck

contention: threads fight for the bus during call of getAndSet()

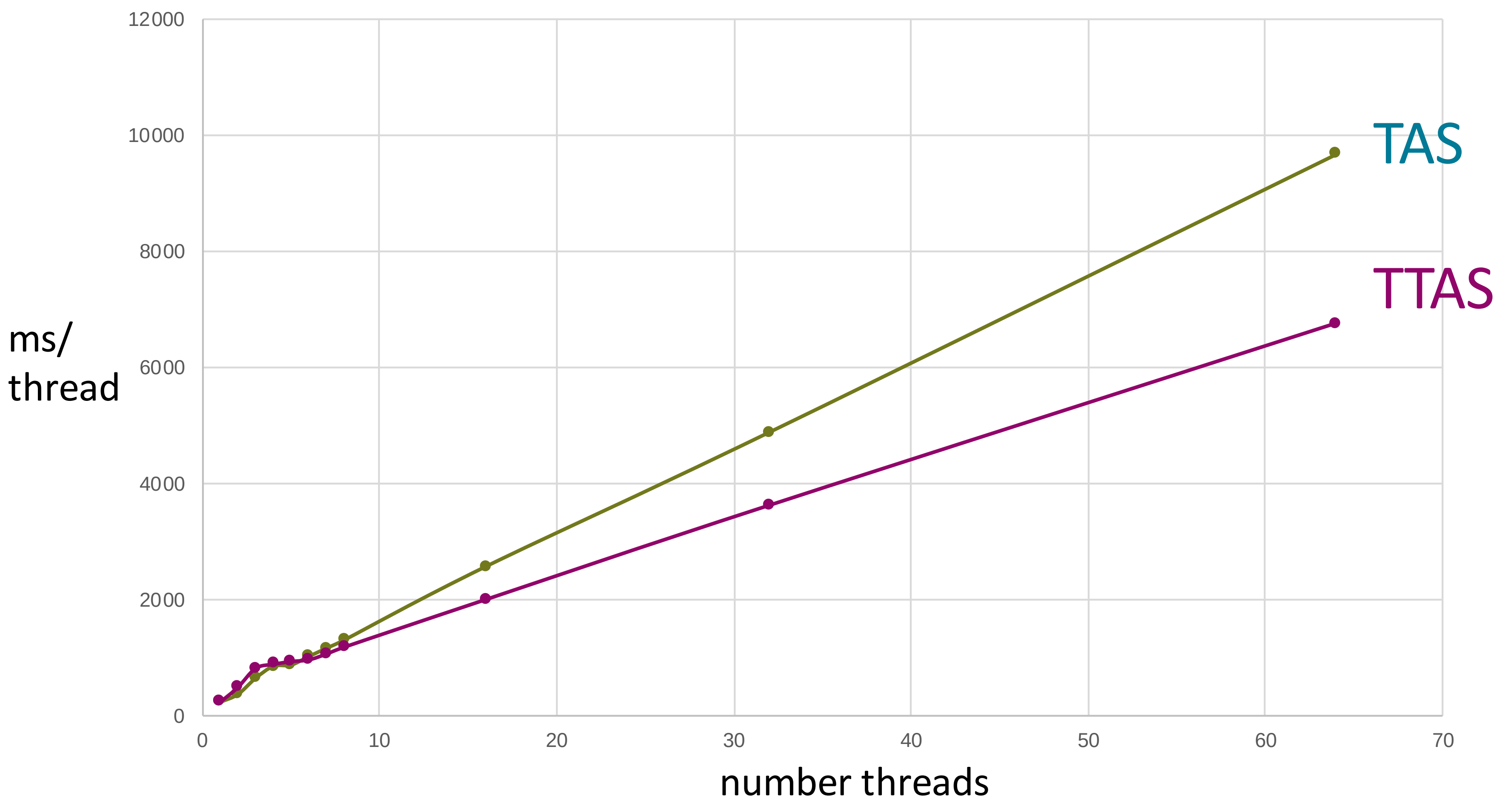
cache coherency protocol invalidates cached copies of the lock on other processors



## Test-and-Test-and-Set (TATAS) Lock

```
public void lock()
{
    do
        while(state.get()) {}
    while (!state.compareAndSet(false, true));
}

public void unlock()
{
    state.set(false);
}
```

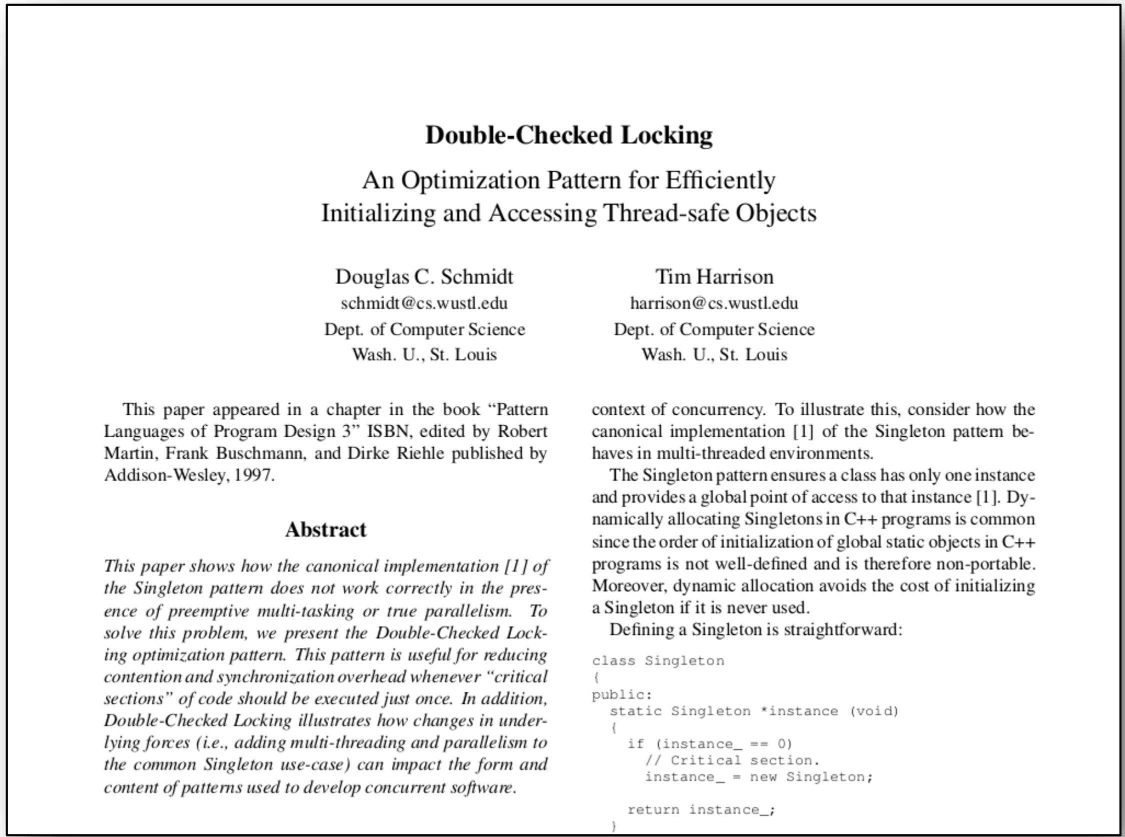
## Measurement



note that this varies strongly between machines and JVM implementations and even between runs. Take it as a qualitative statement

## TATAS does not generalize

- **Example: Double-Checked Locking**



**Double-Checked Locking**

An Optimization Pattern for Efficiently
Initializing and Accessing Thread-safe Objects

Douglas C. Schmidt
schmidt@cs.wustl.edu
Dept. of Computer Science
Wash. U., St. Louis

Tim Harrison
harrison@cs.wustl.edu
Dept. of Computer Science
Wash. U., St. Louis

This paper appeared in a chapter in the book "Pattern Languages of Program Design 3" ISBN, edited by Robert Martin, Frank Buschmann, and Dirke Riehle published by Addison-Wesley, 1997.

**Abstract**

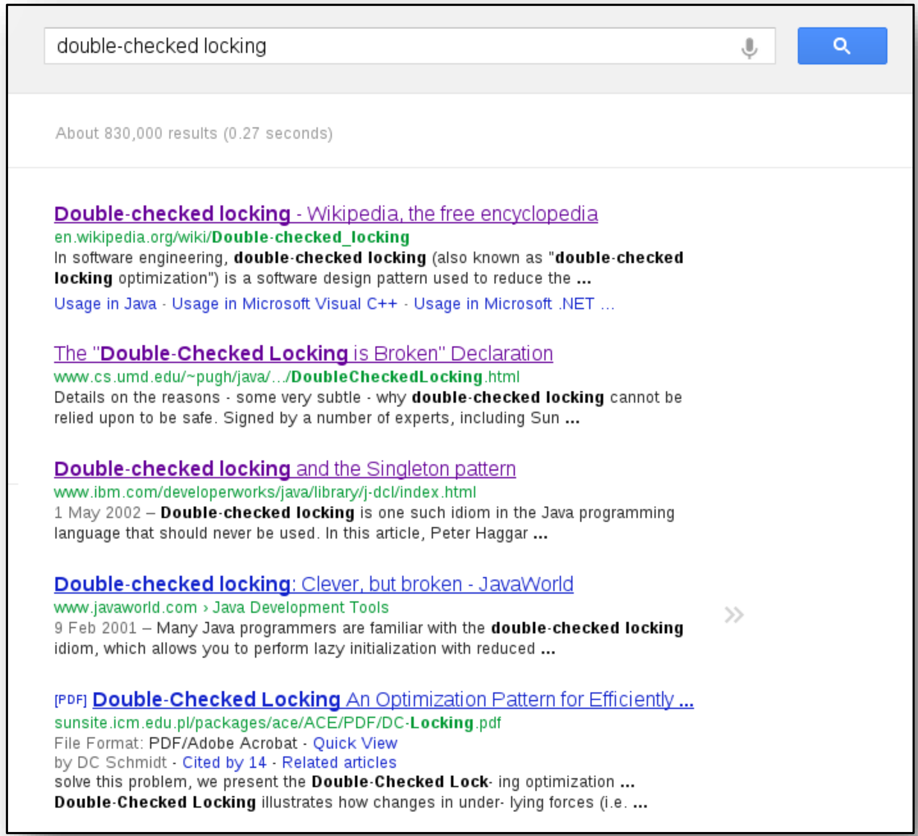
*This paper shows how the canonical implementation [1] of the Singleton pattern does not work correctly in the presence of preemptive multi-tasking or true parallelism. To solve this problem, we present the Double-Checked Locking optimization pattern. This pattern is useful for reducing contention and synchronization overhead whenever "critical sections" of code should be executed just once. In addition, Double-Checked Locking illustrates how changes in underlying forces (i.e., adding multi-threading and parallelism to the common Singleton use-case) can impact the form and content of patterns used to develop concurrent software.*

context of concurrency. To illustrate this, consider how the canonical implementation [1] of the Singleton pattern behaves in multi-threaded environments.

The Singleton pattern ensures a class has only one instance and provides a global point of access to that instance [1]. Dynamically allocating Singletons in C++ programs is common since the order of initialization of global static objects in C++ programs is not well-defined and is therefore non-portable. Moreover, dynamic allocation avoids the cost of initializing a Singleton if it is never used.

Defining a Singleton is straightforward:

```
class Singleton
{
public:
  static Singleton *instance (void)
  {
    if (instance_ == 0)
      // Critical section.
      instance_ = new Singleton;

    return instance_;
  }
```



double-checked locking

About 830,000 results (0.27 seconds)

Double-checked locking - Wikipedia, the free encyclopedia
en.wikipedia.org/wiki/Double-checked_locking
In software engineering, **double-checked locking** (also known as "**double-checked locking** optimization") is a software design pattern used to reduce the ...
Usage in Java - Usage in Microsoft Visual C++ - Usage in Microsoft .NET ...

The "Double-Checked Locking is Broken" Declaration
www.cs.umd.edu/~pugh/java/.../DoubleCheckedLocking.html
Details on the reasons - some very subtle - why **double-checked locking** cannot be relied upon to be safe. Signed by a number of experts, including Sun ...

Double-checked locking and the Singleton pattern
www.ibm.com/developerworks/java/library/j-dcl/index.html
1 May 2002 – **Double-checked locking** is one such idiom in the Java programming language that should never be used. In this article, Peter Haggar ...

Double-checked locking: Clever, but broken - JavaWorld
www.javaworld.com › Java Development Tools
9 Feb 2001 – Many Java programmers are familiar with the **double-checked locking** idiom, which allows you to perform lazy initialization with reduced ...

[PDF] Double-Checked Locking An Optimization Pattern for Efficiently ...
sunsite.icm.edu.pl/packages/ace/ACE/PDF/DC-Locking.pdf
File Format: PDF/Adobe Acrobat - Quick View
by DC Schmidt - Cited by 14 - Related articles
solve this problem, we present the **Double-Checked Lock**- ing optimization ...
**Double-Checked Locking** illustrates how changes in under- lying forces (i.e. ...

Problem: Memory ordering leads to race-conditions!

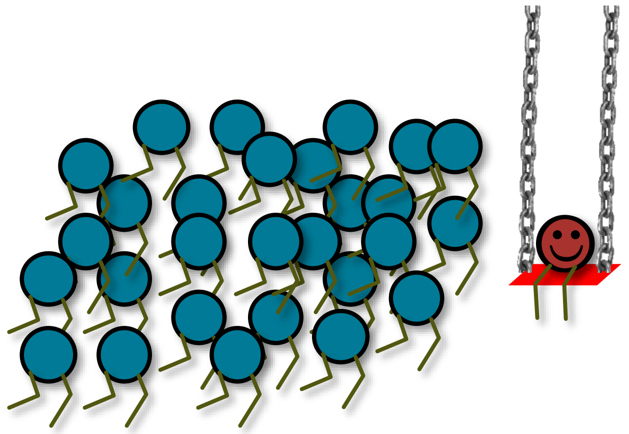
## TATAS with backoff

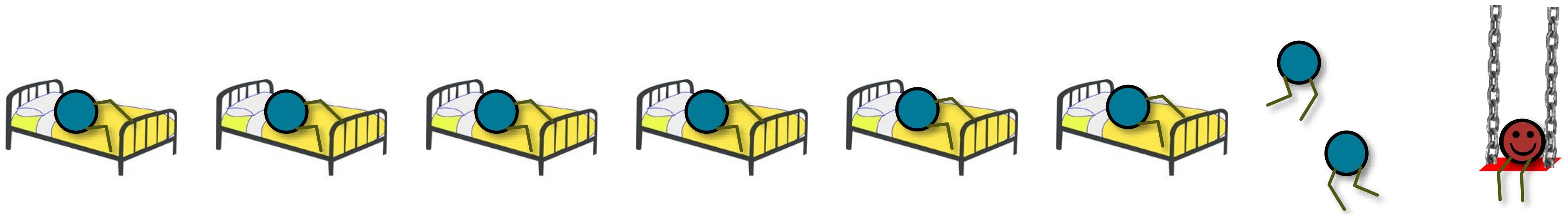
### Observation

- (too) many threads fight for access to the same resource
- slows down progress globally and locally

### Solution

- threads go to sleep with random duration
- increase expected duration each time the resource is not free





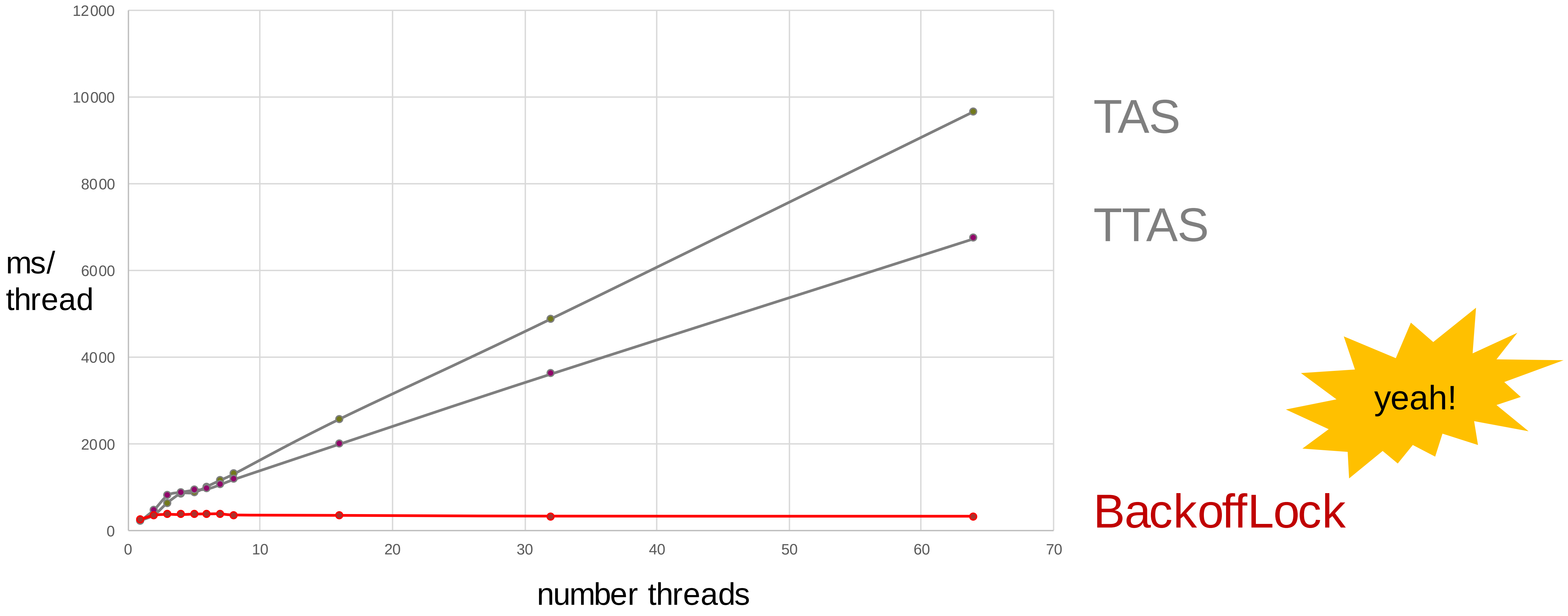
## Lock with Backoff

```
public void lock() {
    Backoff backoff = null;
    while (true) {
        while (state.get()) {};         // spin reading only (TTAS)
        if (!state.getAndSet(true))     // try to acquire, returns previous val
            return;
        else { // backoff on failure
            try {
                if (backoff == null)    // allocation only on demand
                    backoff = new Backoff(MIN_DELAY, MAX_DELAY);
                backoff.backoff();
            } catch (InterruptedException ex) {}
        }
    }
}
```

## exponential backoff

```
class Backoff
{...
      public void backoff() throws InterruptedException {
            int delay = random.nextInt(limit);
            if (limit < maxDelay) { // double limit if less than max
                  limit = 2 * limit;
            }
            Thread.sleep(delay);
      }
}
```

## Measurement



## Summary

- Implementation of spinlocks in software.
- Spinlocks vs. scheduled locks.
- Atomic operations in hardware and Java.
- Next time: higher level abstractions: monitors / semaphores etc.