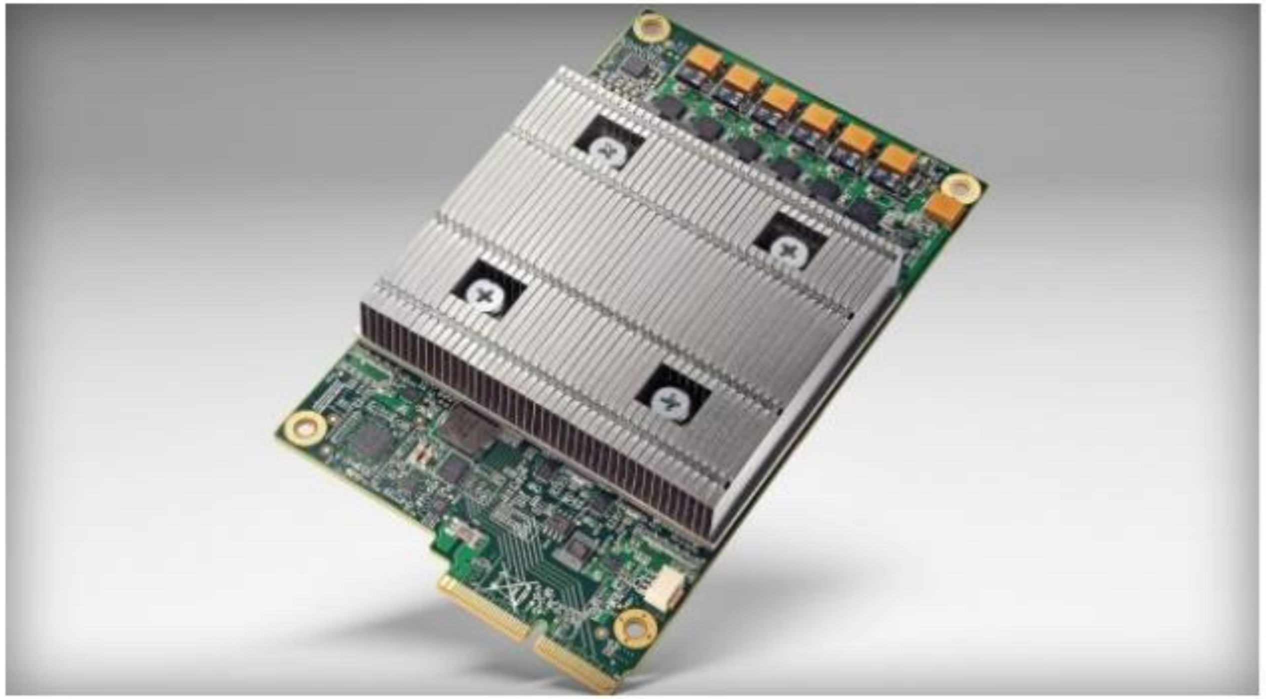
TORSTEN HOEFLER

Parallel Programming, Spr. 2019, Lecture 14: Data Races, Solving Mutual Exclusion with Atomic Registers

Google's dedicated TensorFlow processor, or TPU, crushes Intel, Nvidia in inference workloads

By Joel Hruska on April 6, 2017 at 9:48 am | 23 Comments

2.2K shares



Several years ago, Google began working on its own custom software for machine learning and artificial intelligence workloads, dubbed TensorFlow. Last year, the company announced that it had designed its own tensor processing unit (TPU), an ASIC designed for high throughput of low-precision arithmetic. Now, Google has released some performance data for their TPU and how it compares to Intel's Haswell CPUs and Nvidia's

In-Datacenter Performance Analysis of a Tensor Processing Unit™

Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon

Google, Inc., Mountain View, CA USA

Email: {jouppi, cliffy, nishantpatil, davidpatterson} @google.com

To appear at the 44th International Symposium on Computer Architecture (ISCA), Toronto, Canada, June 26, 2017.

Abstract

Many architects believe that major improvements in cost-energy-performance must now come from domain-specific hardware. This paper evaluates a custom ASIC—called a *Tensor Processing Unit (TPU)*— deployed in datacenters since 2015 that accelerates the inference phase of neural networks (NN). The heart of the TPU is a 65,536 8-bit MAC matrix multiply unit that offers a peak throughput of 92 TeraOps/second (TOPS) and a large (28 MiB) software-managed on-chip memory. The TPU's deterministic execution model is a better match to the 99th-percentile response-time requirement of our NN applications than are the time-varying optimizations of CPUs and GPUs (caches, out-of-order execution, multithreading, multiprocessing, prefetching, …) that help average throughput more than guaranteed latency. The lack of such features helps explain why, despite having myriad MACs and a big memory, the TPU is relatively small and low power. We compare the TPU to a server-class Intel Haswell CPU and an Nvidia K80 GPU, which are contemporaries deployed in the same datacenters. Our workload, written in the high-level TensorFlow framework, uses production NN applications (MLPs, CNNs, and LSTMs) that represent 95% of our datacenters' NN inference demand. Despite low utilization for some applications, the TPU is on average about 15X - 30X faster than its contemporary GPU or CPU, with TOPS/Watt about 30X - 80X higher. Moreover, using the GPU's GDDR5 memory in the TPU would triple achieved TOPS and raise TOPS/Watt to nearly 70X the GPU and 200X the CPU.

Index terms–DNN, MLP, CNN, RNN, LSTM, neural network, domain-specific architecture, accelerator

Learning goals for today

So far:

- **Programming with locks and critical sections**
- **Key guidelines and trade-offs**
- **Bad interleavings (high level races)**

Now:

- The unfortunate reality of parallel programming in practice – memory models
- **Why you must avoid data races** (= low level races / memory reorderings)
- Implementation of a Mutex with Atomic Registers

  Dekker's algorithm

  Peterson's algorithm

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

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

Motivation

```
class C {
  private int x = 0;
  private int y = 0;
  Thread 1
    x = 1;  Ⓐ
    y = 1;  Ⓑ
  }
  Thread 2
    int a = y;  Ⓒ
    int b = x;  Ⓓ
    assert(b >= a);
  }
}
```

Can this fail?

There is no *interleaving* of f and g that would cause the assertion to fail:

Ⓐ Ⓑ Ⓒ Ⓓ ✓
Ⓐ Ⓒ Ⓑ Ⓓ ✓
Ⓐ Ⓒ Ⓓ Ⓑ ✓
Ⓒ Ⓐ Ⓑ Ⓓ ✓
Ⓒ Ⓐ Ⓓ Ⓑ ✓
Ⓒ Ⓓ Ⓐ Ⓑ ✓

Proof by exhaustion (or full enumeration)!

A little combinatorial excursion

- **Assuming 2 threads and k statements each, how many interleavings are there?**
  - Any ideas?
- **Hint 1**
  - The merged list has length k+k=2k
  - Once we know which k positions in the merged list are occupied with elements from thread 1 (or 2) then the interleaving is determined!
  - How many are those?
- **Hint 2**
  - This is equivalent to sampling without replacement (draw the k positions out of 2k total)

    “Ziehen ohne Zuruecklegen”
  - $\binom{2k}{k} = O\left(\frac{4^n}{\sqrt{2n}}\right)$

- **If you cannot sleep tonight:**
  - Generalize this to n threads ☺

Another proof

```
class C {
  private int x = 0;
  private int y = 0;
  Thread 1
    x = 1;
    y = 1;
  }
  Thread 2
    int a = y;
    int b = x;
    assert(b >= a);
  }
}
```

There is no interleaving of `f` and `g` causing the assertion to fail

Another proof (by contradiction):

Assume `b<a` ⇒ `a==1` and `b==0`.

But if `a==1` ⇒ `y=1` *happened before* `a=y`.
And if `b==0` ⇒ `b=x` *happened before* `x=1`.

Because we assume that programs execute in order:

`a=y` *happened before* `b=x`
`x=1` *happened before* `y=1`

So by transitivity,
a=y happened before b=x happened before x=1 happened before y=1 happened before a=y ⇒ Contradiction ↯

Let’s try that on my laptop

Why it still can fail: Memory reordering

Rule of thumb: Compiler and hardware allowed to make changes that do not affect the *semantics* of a *sequentially* executed program

```
void f() {
        x = 1;
        y = x+1;
        z = x+1;
}
```

semantically equivalent?

```
void f() {
        x = 1;
        z = x+1;
        y = x+1;
}
```

semantically equivalent?

```
void f() {
        x = 1;
        z = 2;
        y = 2;
}
```

In a **sequential** world!

Memory reordering: A software view

Modern compilers do not give guarantees that a global ordering of memory accesses is provided:

- **Some memory accesses may be even optimized away completely!**

- **Class question: why?**

- **Huge potential for optimizations – and for errors, when you make the wrong assumptions**
  - Dead code elimination
  - Register hoisting
  - Locality optimizations
  - … many more (beyond this basic class)

Example: Fail with self-made rendezvous (C / GCC)

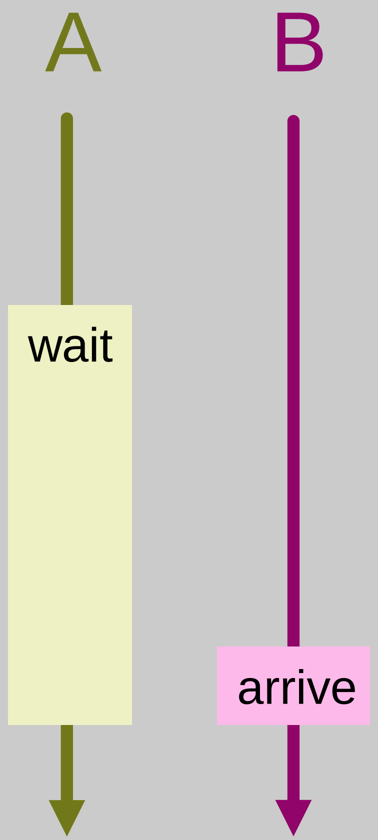
```
int x;

void wait() {
  x = 1;
  while(x==1);
}
```

```
void arrive(){
  x = 2;
}
```

Consider
thread A calling wait and
thread B subsequently calling arrive.

What would you naively expect?



Example: Fail with self-made rendezvous (C / GCC)

```
int x;

void wait() {
  x = 1;
  while(x==1);
}
```

```
void arrive(){
  x = 2;
}
```

Assembly without optimization

```
movl   $0x1, x
test:
mov     x, %eax
cmp     $0x1, %eax
je      test
```

je: jump (only) if equal, i.e., if cmp yields true

```
movl $0x2, x
```

Assembly with optimization

```
movl   $0x1, x
test:
jmp    test
```

jmp: jump always

```
movl $0x2, x
```

Memory reordering: A hardware view

Modern multiprocessors do not enforce global ordering of all instructions:

- **What they actually guarantee varies widely!**

- **Class question: why?**

- **For performance!**
  - Most processors have a pipelined architecture and can execute (parts of) multiple instructions simultaneously. They can (and will) even reorder instructions internally.
  - Each processor has a local cache, and thus loads/stores to shared memory can become visible to other processors at different times

Memory hierachy (one core)

ALUs

Registers

L1 Cache

L2 Cache

System Memory

fast, low latency, high cost, low capacity

slow, high latency latency, low cost, high capacity

Memory hierachy (many cores)

ALUs | ALUs | ... | ALUs

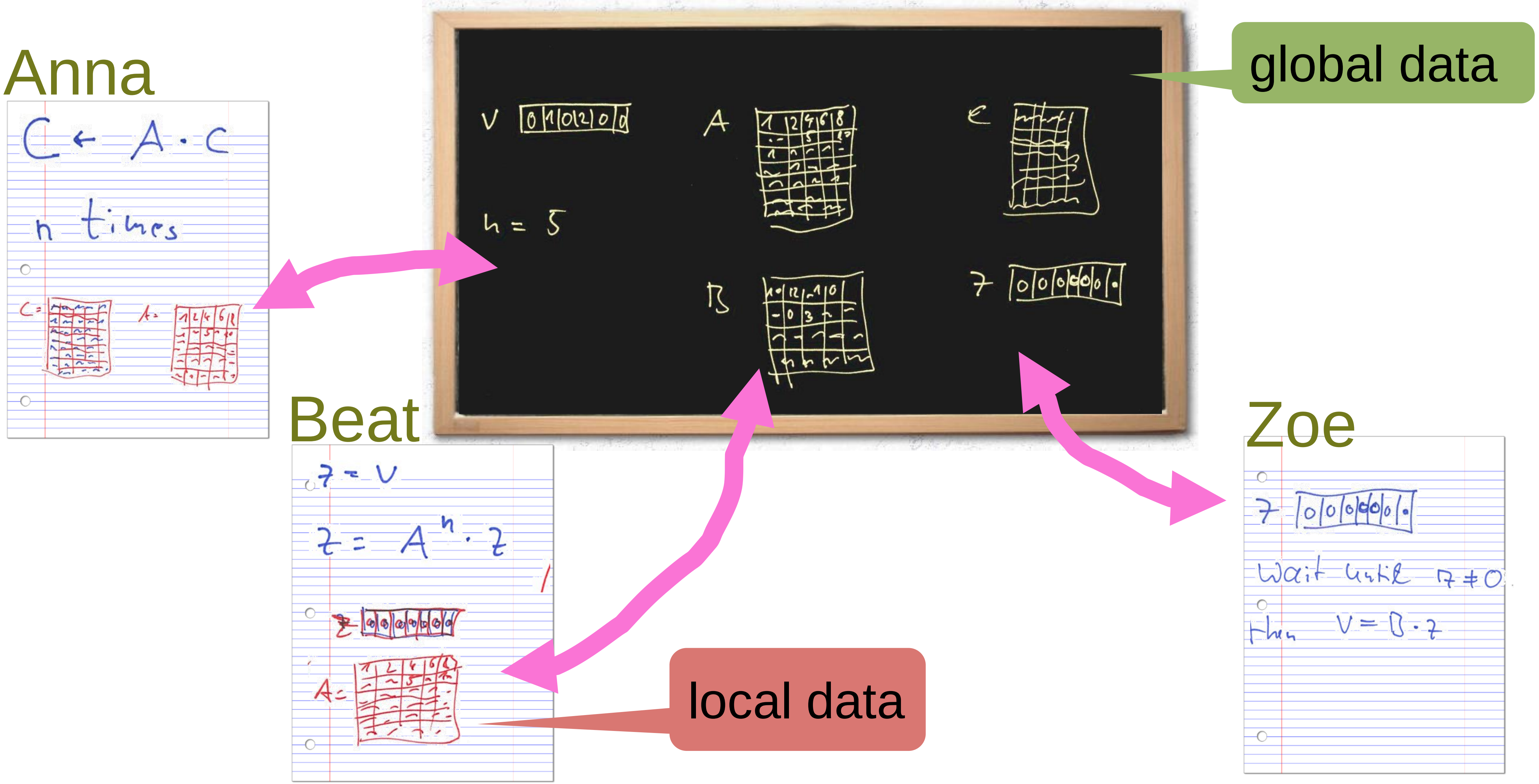
Registers | Registers | ... | Registers

L1 Cache | L1 Cache | ... | L1 Cache

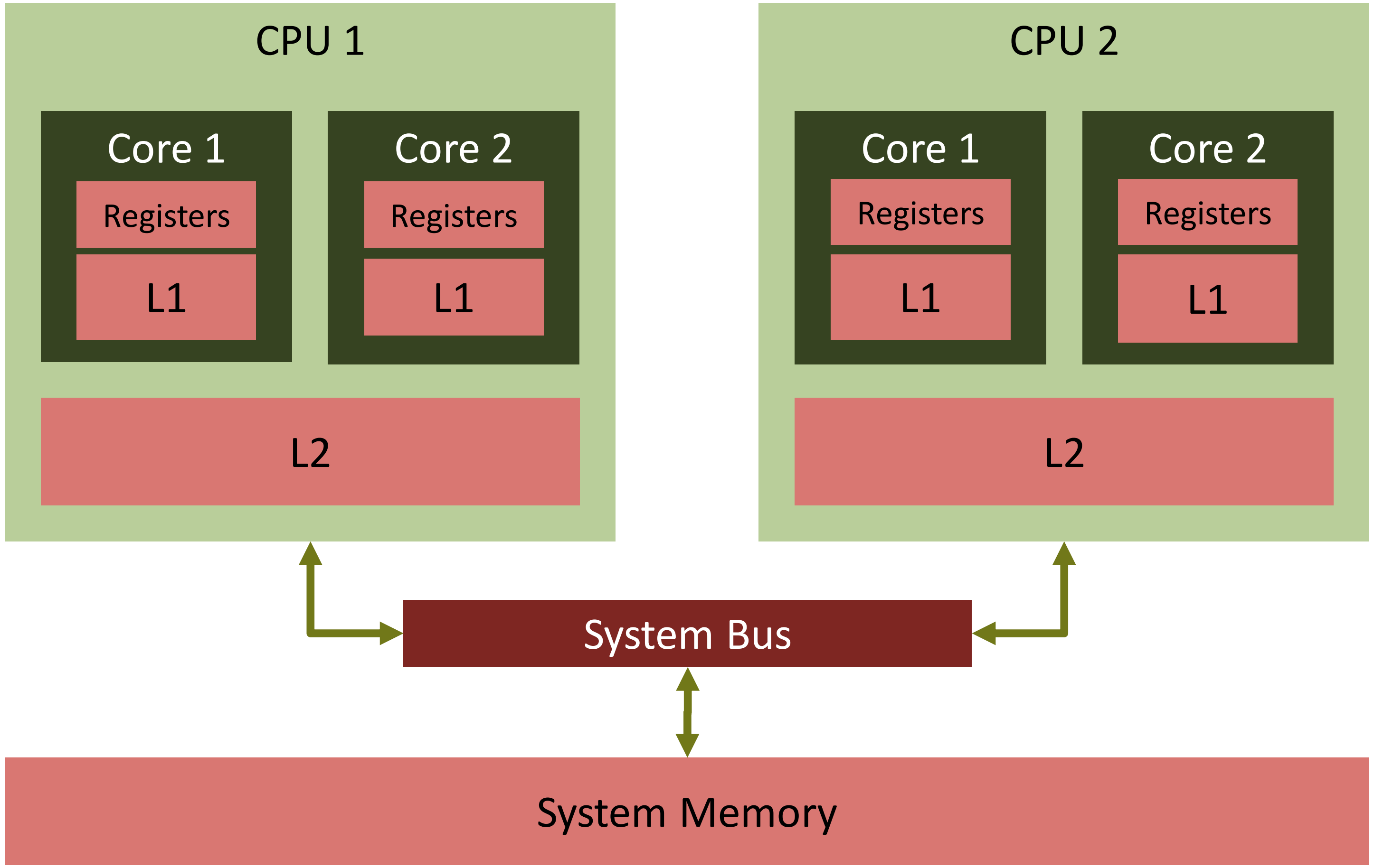
Shared L2 Cache

System Memory

A real-life analogy



Sharing memory (schematically)



Memory models

The exact behavior of threads interacting via shared memory usually depends on hardware, runtime system, and programming language.

A memory model (e.g., of a programming language like Java) provides (often minimal) guarantees for the effects of memory operations.

- **leaving open optimization possibilities for hardware and compiler**
- **but including guidelines for writing correct multithreaded programs**

Will come back to this later.

合同 (contract)

Implications

We need to learn (a bit) more about Java's Memory Model.

For now, we know that Java gives certain guarantees in the presence of synchronization.

Fixing our example

```
class C {
   private int x = 0;
   private int y = 0;
   void f() {
      synchronized(this) { x = 1; }
      synchronized(this) { y = 1; }
   }
   void g() {
      int a, b;
      synchronized(this) { a = y; }
      synchronized(this) { b = x; }
      assert(b >= a);
   }
}
```

- Can use synchronization to avoid data races
- Then, indeed, the assertion cannot fail

Another fix

```
class C {
   private volatile int x = 0;
   private volatile int y = 0;
   void f() {
      x = 1;
      y = 1;
   }
   void g() {
      int a = y;
      int b = x;
      assert(b >= a);
   }
}
```

- Java has `volatile` fields: accesses do not count as data races
- Implementation: slower than regular fields, faster than locks
- Really for experts: avoid them; use standard libraries instead
- And why do you need code like this anyway?

More realistic example of code that is wrong

```
class C {
  boolean stop = false;

  void f() {
    while(!stop) {
      // draw a monster
    }
  }

  void g() {
    stop = didUserQuit();
  }
}
```

Thread 1: `f()`

Thread 2: `g()`

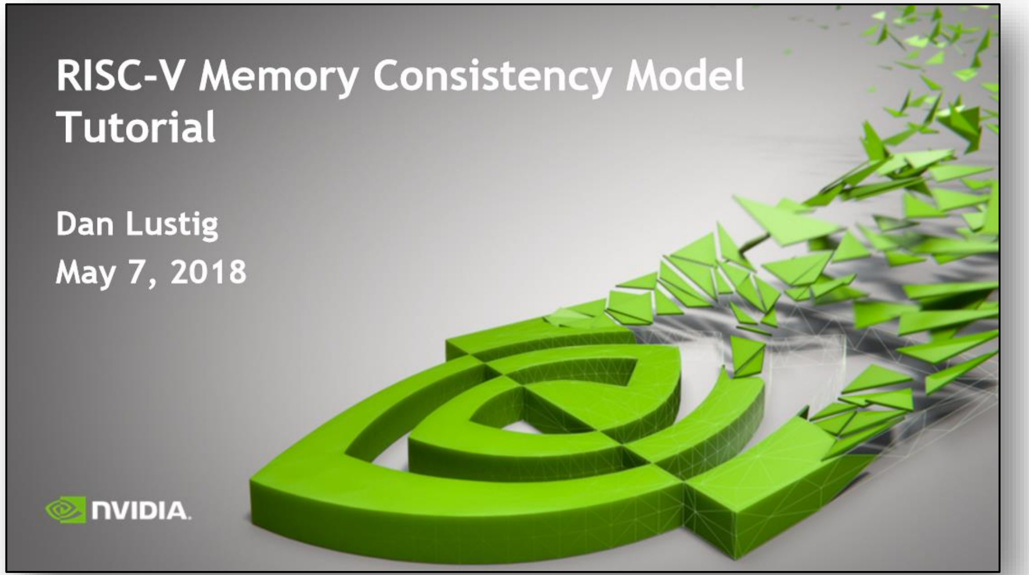
No *guarantee* Thread 1 will *ever* stop.

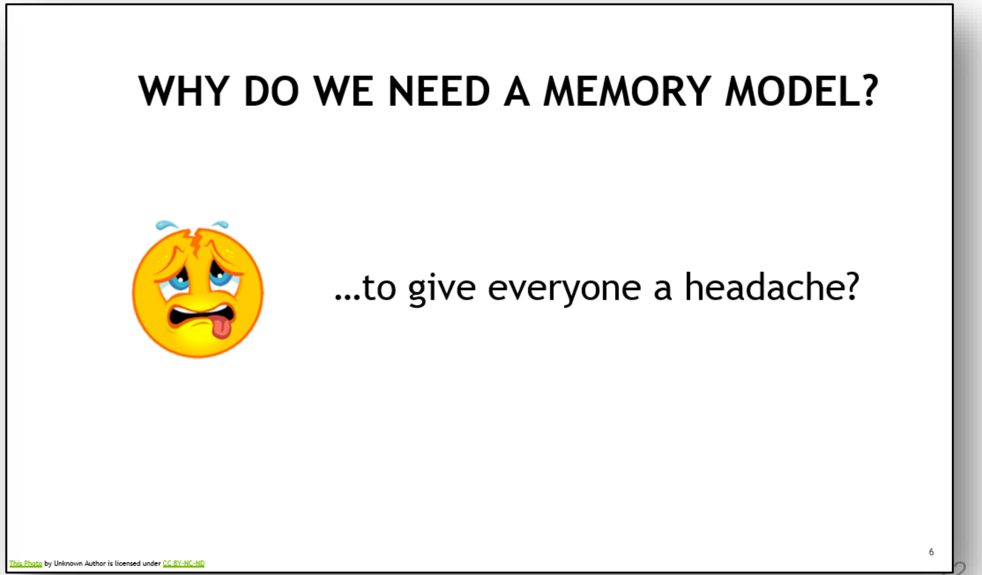
But honestly it will "*likely* work in practice"

What did we learn?

- **Compilers and computer architectures will change orders of memory operations**
  - Consistent with sequential semantics!
  - May impact parallel execution ☹
- **There are some language constructs that forbid such reordering**
  - We saw synchronized and volatile in Java
  - But what do they really mean?
  - Now we need to dig a bit deeper (I’d rather not but have to)

    It’s quite complex!
- **Memory models**





Why (architectural) memory models? For real …

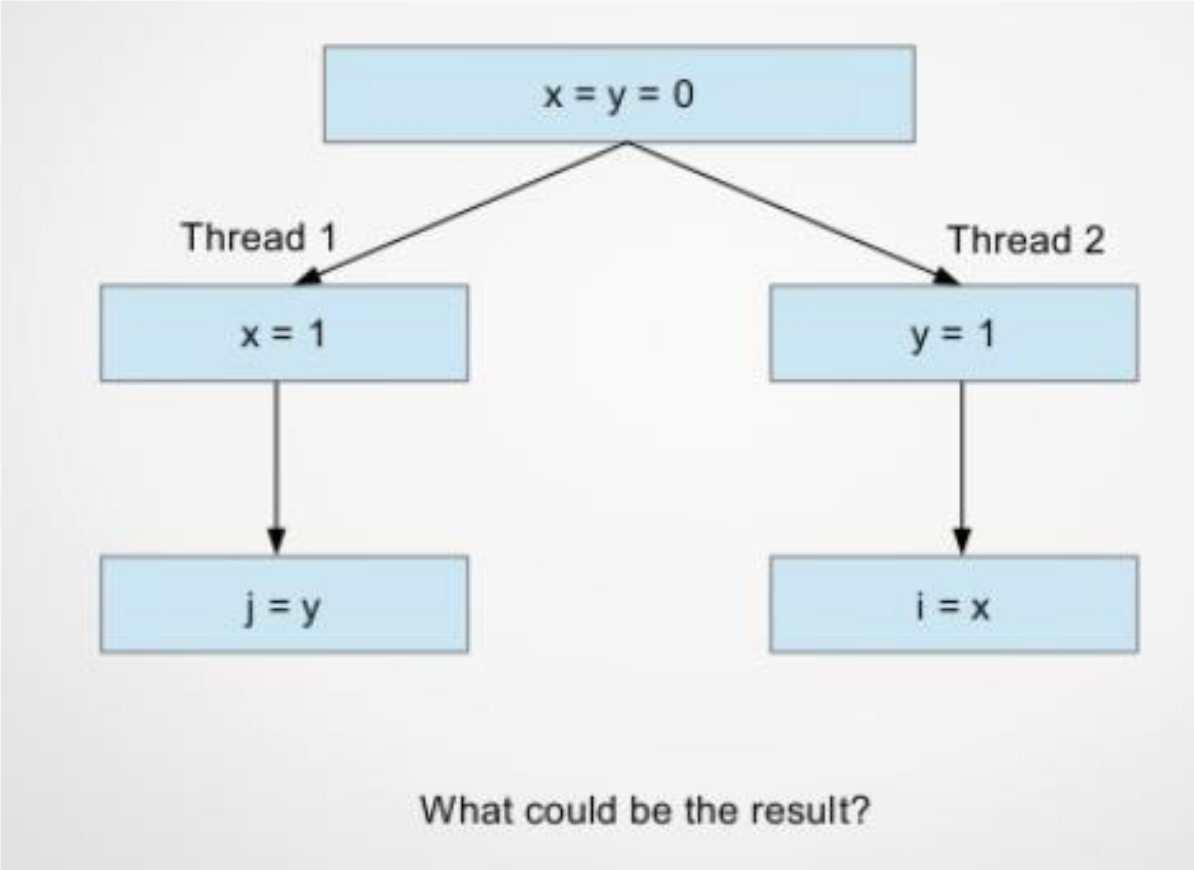
- **You expect instructions to be executed in program order?**
- **But your compiler, your CPU, and your DRAM reorder! For better performance.**
- **What will be reordered depends on hardware, e.g., AMD86 is different than ARM.**
  - In single threaded programs this does not cause problems.
- **But let's see a shared-memory multithreading example using x86**

Memory ordering in some architectures[7][8]

Type	Alpha	ARMv7	PA-RISC	POWER	SPARC			x86		AMD64	IA-64	z/Architecture
					RMO	PSO	TSO		oostore[a]			
Loads reordered after loads	Y	Y	Y	Y	Y				Y		Y	
Loads reordered after stores	Y	Y	Y	Y	Y				Y		Y	
Stores reordered after stores	Y	Y	Y	Y	Y	Y			Y		Y	
Stores reordered after loads	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Atomic reordered with loads	Y	Y		Y	Y						Y	
Atomic reordered with stores	Y	Y		Y	Y	Y					Y	
Dependent loads reordered	Y											
Incoherent instruction cache pipeline	Y	Y		Y	Y	Y	Y	Y	Y		Y	

Source: wikipedia

Why memory models, x86 example



Answer:
i=1, j=1
i=0, j=1
i=1, j=0
i=0, j=0 (but why?)

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 ☺)

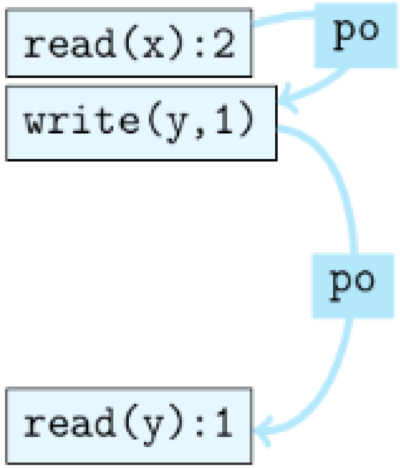
- **JMM defines *Actions:* `read(x):1` "read variable x, the value read is 1"**

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

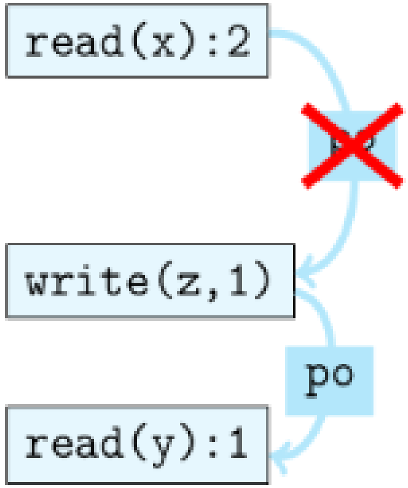
JMM: Program Order (PO)

- **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

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

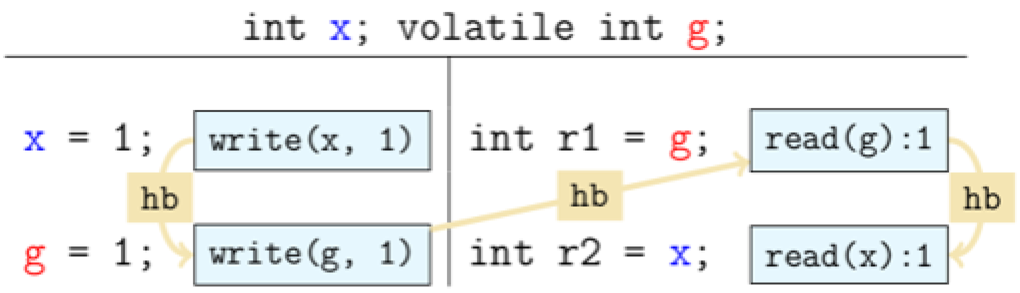
Exercise: List all outcomes (r1,r2) allowed by the JMM.

- **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

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

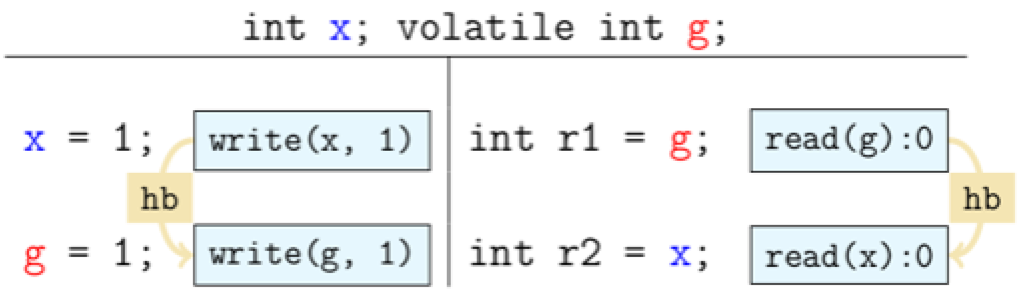
- **SW only pairs the specific actions which "see" each other**
- **A volatile write to x synchronizes with subsequent read of x (subsequent in SO)**
- **The transitive closure of PO and SW forms HB**
- **HB consistency: When reading a variable, we see either the last write (in HB) or any other unordered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

Example



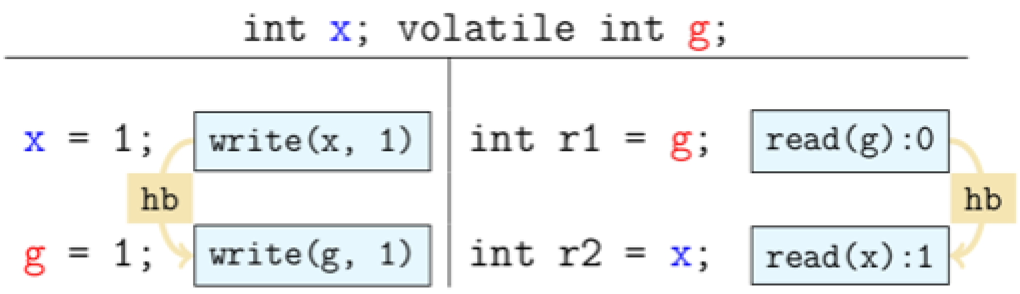
Case 1: HB consistent, observe the latest write in $\xrightarrow{\text{hb}}$

$(r1, r2) = (1, 1)$



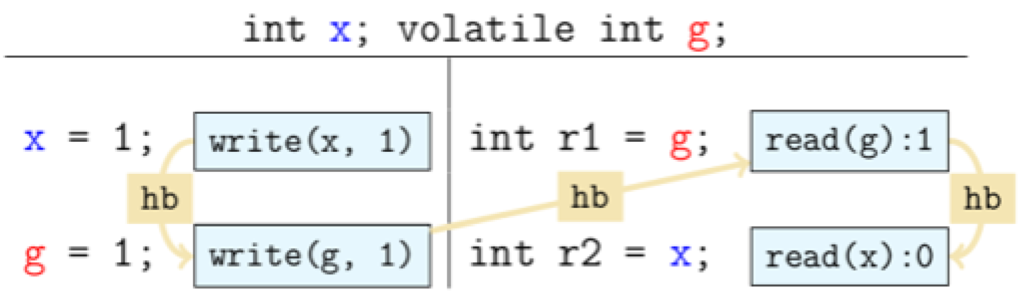
Case 2: HB consistent, observe the default value

$(r1, r2) = (0, 0)$



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!

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

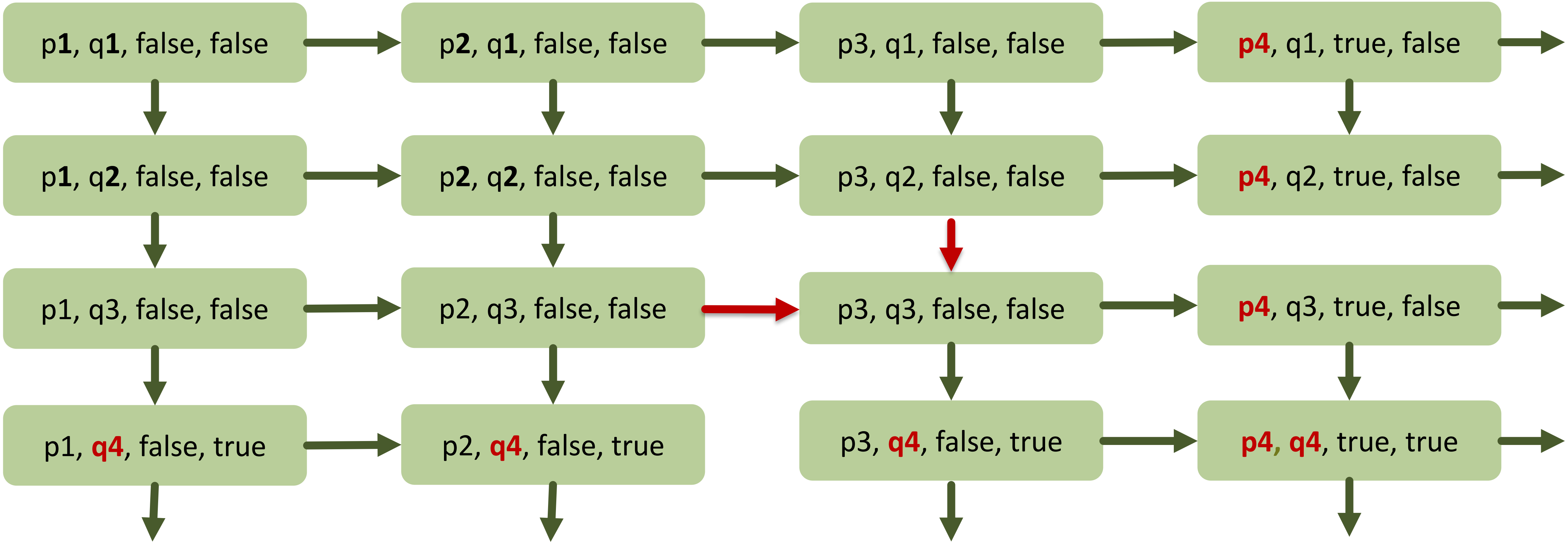
Do you see the problem?

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

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

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



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**

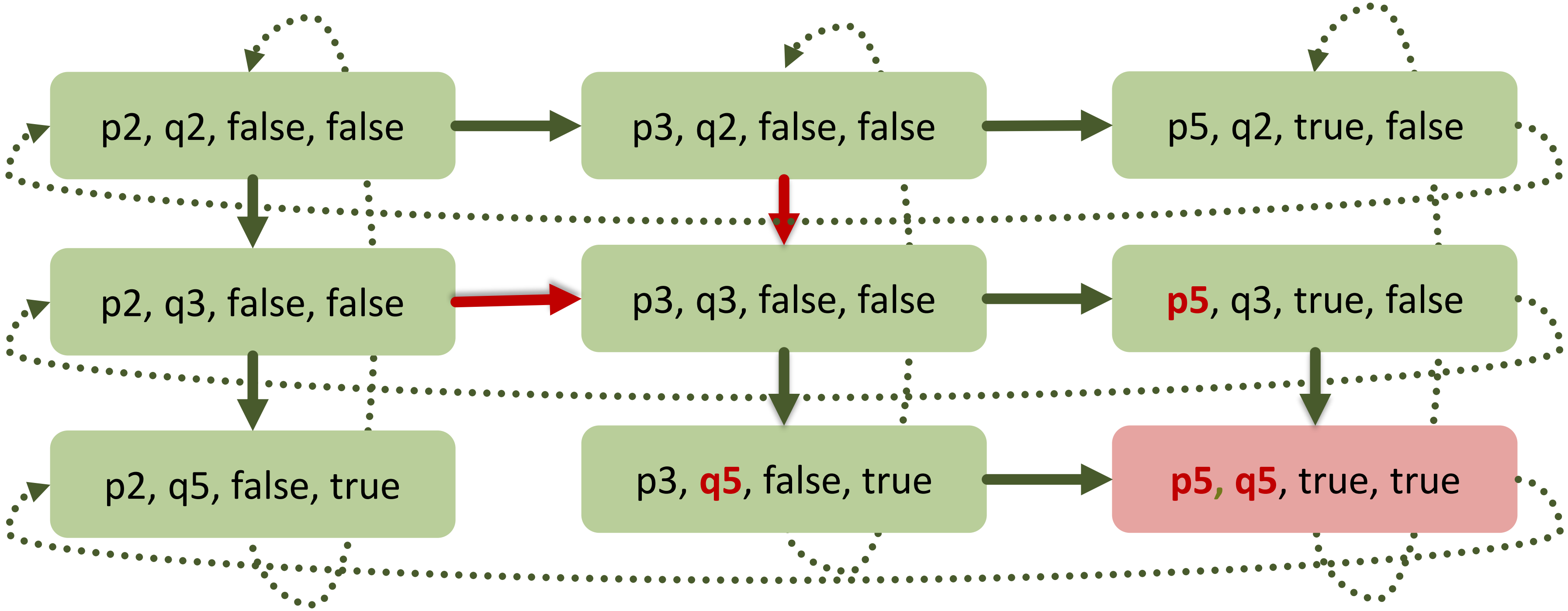
loop

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:



no mutual exclusion !

Mutual exclusion for 2 processes -- 2nd Try

volatile boolean wantp=false, wantq=false

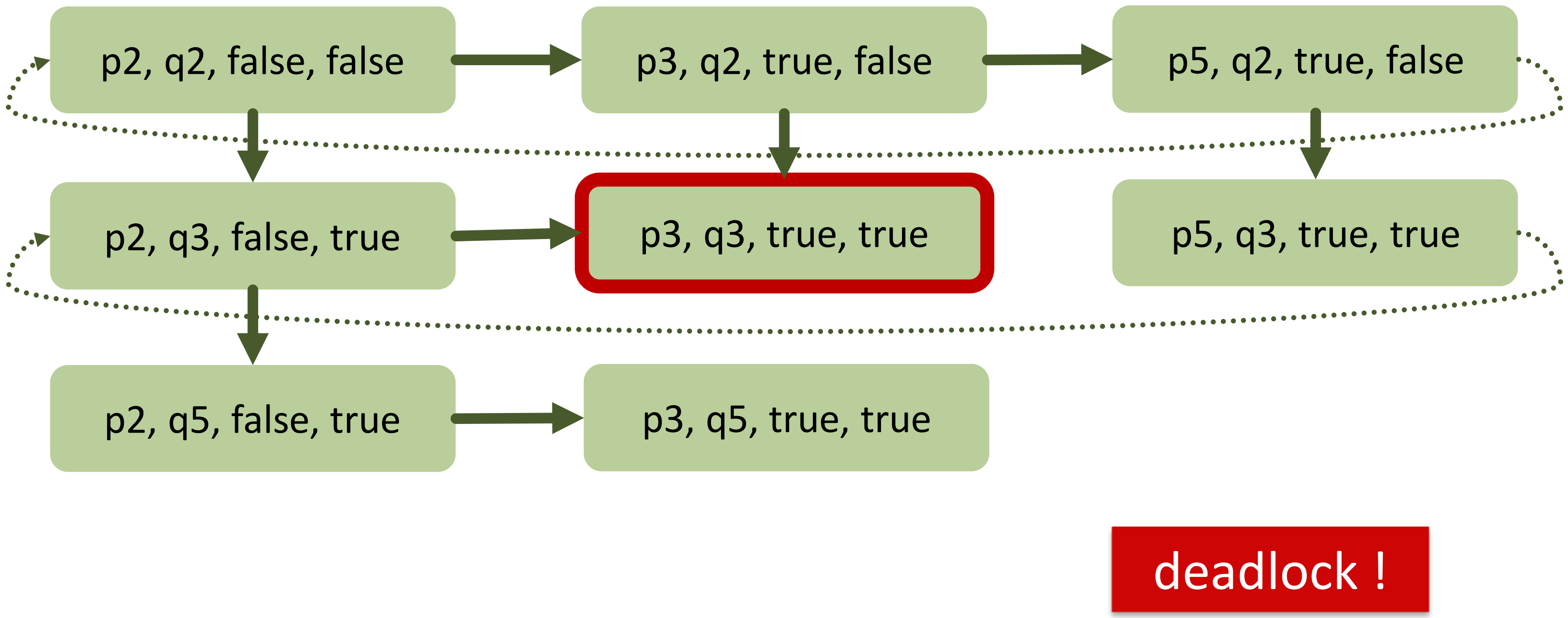
Do you see the problem?

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 wantq = true	**3** while(wantp) while(wantq)	**4** critical section	**5** wantp = false wantq = false



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

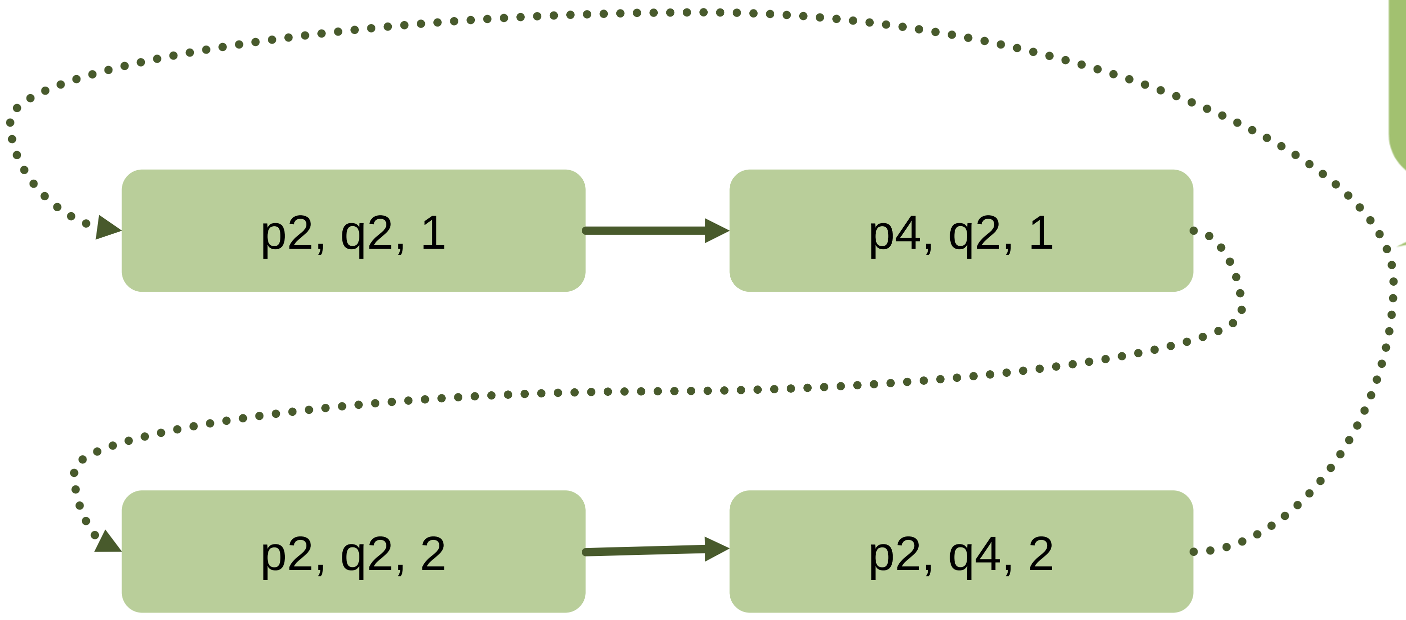
local variables

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

How?

Requires some notation first.

Events and precedence

Threads produce a sequence of events

P produces events $p_0, p_1, \ldots$

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: p_i^j

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

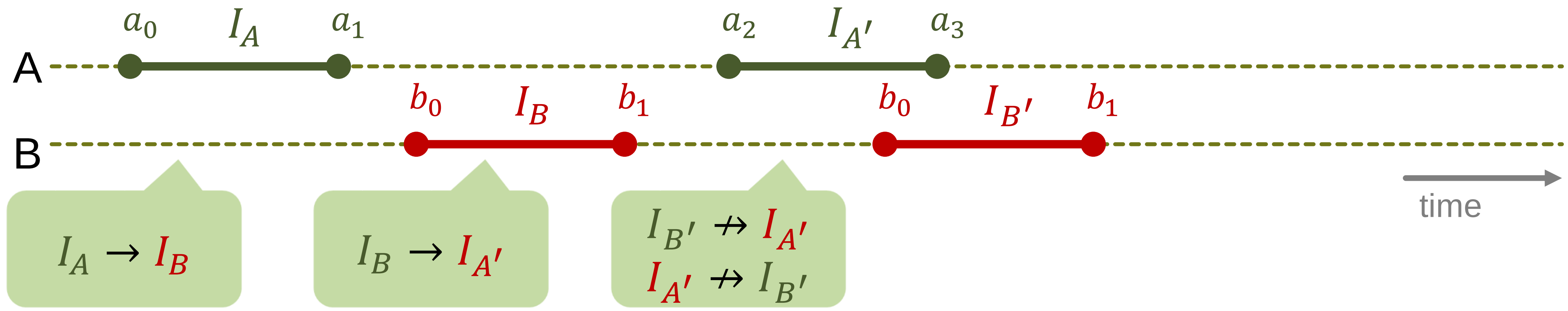
Precedence relation: we write $a \to b$ when a occurs before b.

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

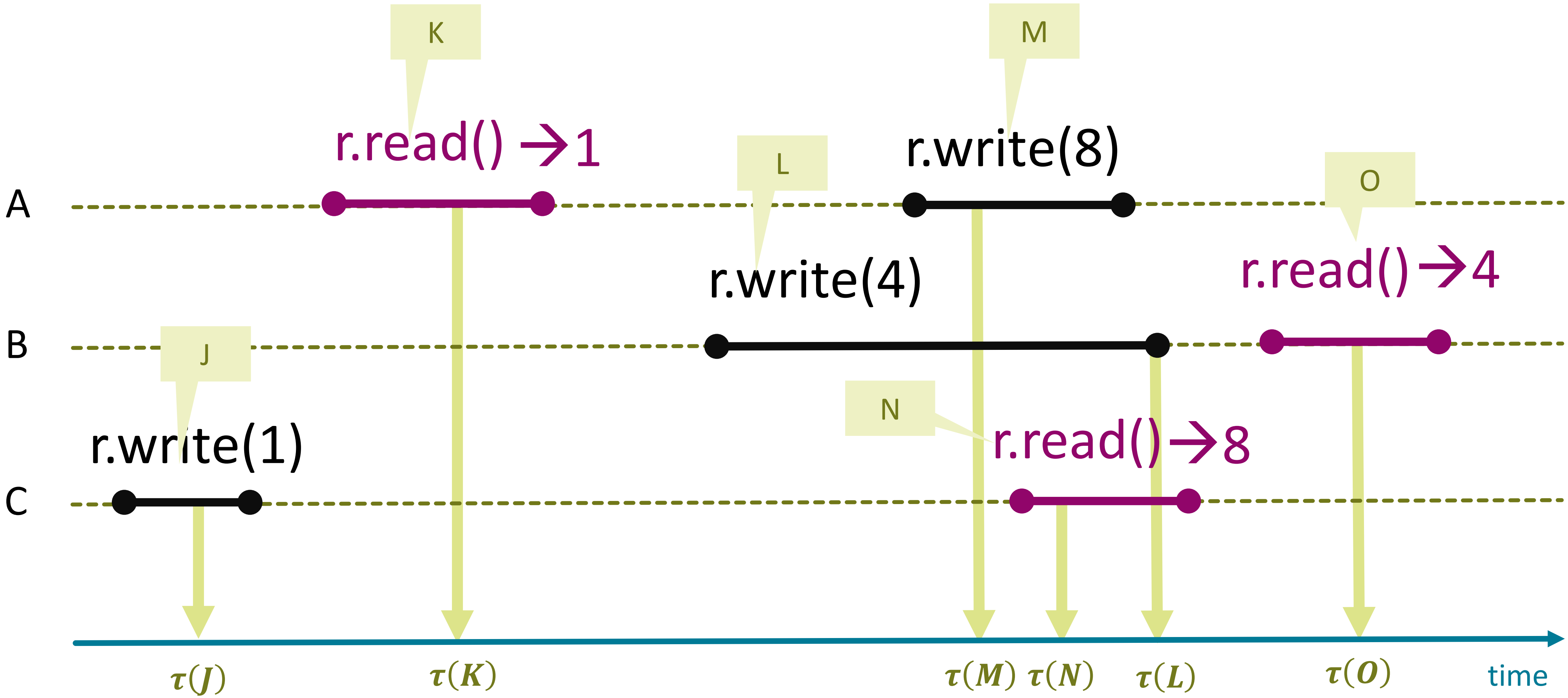
Register: basic memory object, can be shared or not
i.e., in this context register $\neq$ register of a CPU

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.

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){}
CS_P
flag[P] = false
```

Proof: Mutual exclusion (Peterson)

By contradiction: assume concurrent CS_P and CS_Q [A]

Assume without loss of generality:

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

From the code:

A + C ⇒ must read false

B ⇒ must read P [C]

$$W_P(flag[P]=true) \rightarrow W_P(victim = P) \rightarrow R_P(flag[Q]) \rightarrow R_P(victim) \rightarrow CS_P$$

transitivity of "→ "
⇒ must read true ⚡

"write of P"

$$W_Q(flag[Q]=true) \rightarrow W_Q(victim = Q) \rightarrow R_Q(flag[P]) \rightarrow R_Q(victim) \rightarrow CS_Q$$

"read of Q"

Proof: Freedom from starvation

```
flag[P] = true
victim = P
while (flag[Q] && victim == P){}
CS_P
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 production programs it is recommended to use Java's **AtomicInteger** and **AtomicIntegerArray**.

More than two threads

- How to extend Peterson’s lock to more than 2 threads?
- Think about it, I will present a solution in tomorrow's lecture.