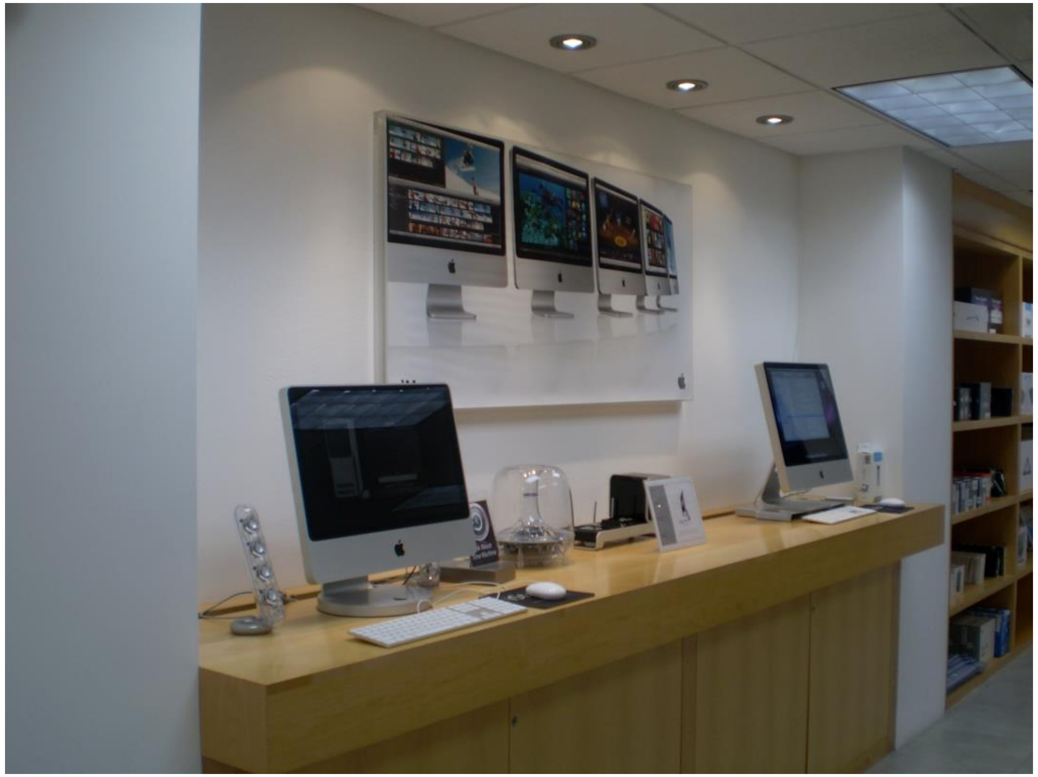
T. HOEFLER, M. PUESCHEL

Lecture 4: Languages, Fast Locks, and Lock-free

Teaching assistant: Salvatore Di Girolamo

Motivational video: https://www.youtube.com/watch?v=1o4YViBAGU0

So, a computing scientist entered a store….



http://archive.constantcontact.com/fs042/1101916237075/archive/1102594461324.html

http://bitchmagazine.org/post/beyond-the-panel-an-interview-with-danielle-corsetto-of-girls-with-slingshots

Adopted from Jose Nelson Amaral, “How did this get published?”, LCTES’12

So, a computing scientist entered a store….

http://bitchmagazine.org/post/beyond-the-panel-an-interview-with-danielle-corsetto-of-girls-with-slingshots

Adopted from Jose Nelson Amaral, “How did this get published?”, LCTES’12

So, a computing scientist entered a store….

$ 3,000.00

http://bitchmagazine.org/post/beyond-the-panel-an-interview-with-danielle-corsetto-of-girls-with-slingshots

Adopted from Jose Nelson Amaral, “How did this get published?”, LCTES’12

So, a computing scientist entered a store….

$ 3,000.00

$ 200.00

http://bitchmagazine.org/post/beyond-the-panel-an-interview-with-danielle-corsetto-of-girls-with-slingshots

Adopted from Jose Nelson Amaral, “How did this get published?”, LCTES’12

So, a computing scientist entered a store….

$ 3,000.00

$ 200.00

http://bitchmagazine.org/post/beyond-the-panel-an-interview-with-danielle-corsetto-of-girls-with-slingshots

Adopted from Jose Nelson Amaral, “How did this get published?”, LCTES’12

So, a computing scientist entered a store….

$ 3,000.00

$ 200.00

http://bitchmagazine.org/post/beyond-the-panel-an-interview-with-danielle-corsetto-of-girls-with-slingshots

Adopted from Jose Nelson Amaral, “How did this get published?”, LCTES’12

So, a computing scientist entered a store….

$ 3,000.00

$ 200.00

They want $2,700 for the server and $100 for the iPod.

http://bitchmagazine.org/post/beyond-the-panel-an-interview-with-danielle-corsetto-of-girls-with-slingshots

Adopted from Jose Nelson Amaral, “How did this get published?”, LCTES’12

So, a computing scientist entered a store….

$ 3,000.00

$ 200.00

They want $2,700 for the server and $100 for the iPod.

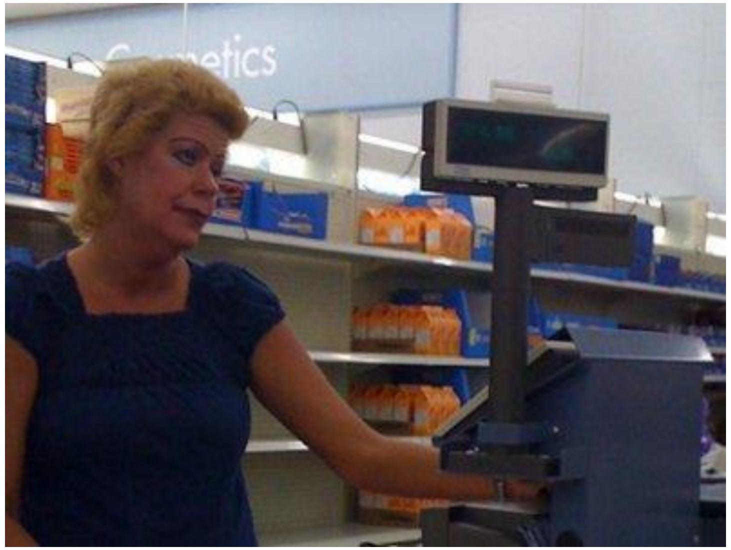
I will get both and pay only $2,240 altogether!

http://bitchmagazine.org/post/beyond-the-panel-an-interview-with-danielle-corsetto-of-girls-with-slingshots

Adopted from Jose Nelson Amaral, “How did this get published?”, LCTES’12

So, a computing scientist entered a store….

$ 200.00



$ 3,000.00

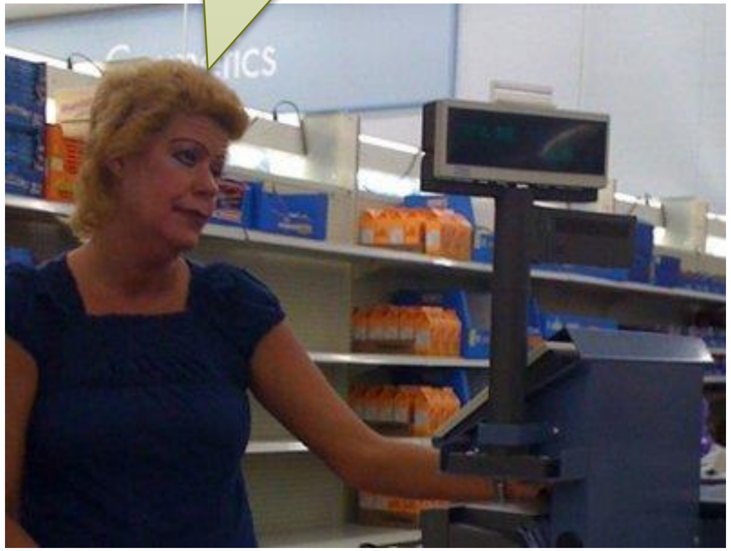
http://www.businessinsider.com/10-ways-to-fix-googles-busted-android-app-market-2010-1?op=1

http://bitchmagazine.org/post/beyond-the-panel-an-interview-with-danielle-corsetto-of-girls-with-slingshots

Adopted from Jose Nelson Amaral, “How did this get published?”, LCTES’12

So, a computing scientist entered a store….

Ma’am you are $560 short.



http://www.businessinsider.com/10-ways-to-fix-googles-busted-android-app-market-2010-1?op=1

50%
Discount
Buy Now!

$ 200.00

10%
Discount
Buy Now!

$ 3,000.00

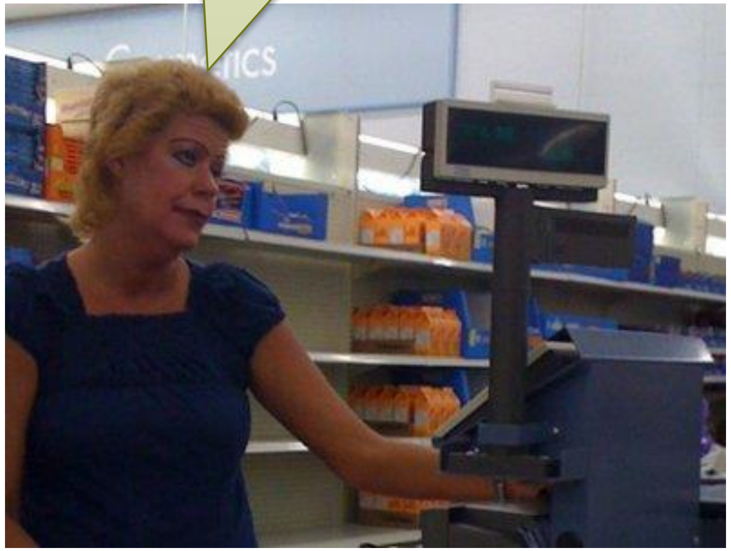
http://bitchmagazine.org/post/beyond-the-panel-an-interview-with-danielle-corsetto-of-girls-with-slingshots

Adopted from Jose Nelson Amaral, “How did this get published?”, LCTES’12

So, a computing scientist entered a store….

Ma’am you are $560 short.

But the average of 10% and 50% is 30% and 70% of $3,200 is $2,240.



http://www.businessinsider.com/10-ways-to-fix-googles-busted-android-app-market-2010-1?op=1

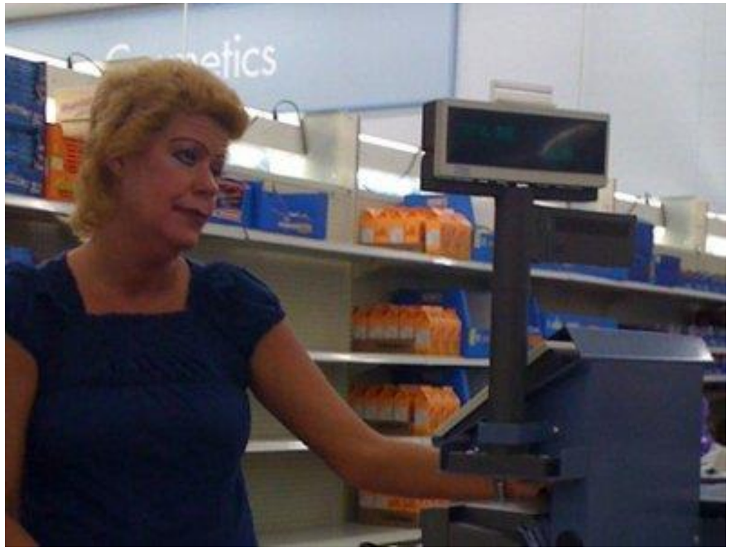
$ 200.00

$ 3,000.00

http://bitchmagazine.org/post/beyond-the-panel-an-interview-with-danielle-corsetto-of-girls-with-slingshots

Adopted from Jose Nelson Amaral, “How did this get published?”, LCTES’12

So, a computing scientist entered a store….



http://www.businessinsider.com/10-ways-to-fix-googles-busted-android-app-market-2010-1?op=1

$ 200.00

50%
Discount
Buy Now!

10%
Discount
Buy Now!

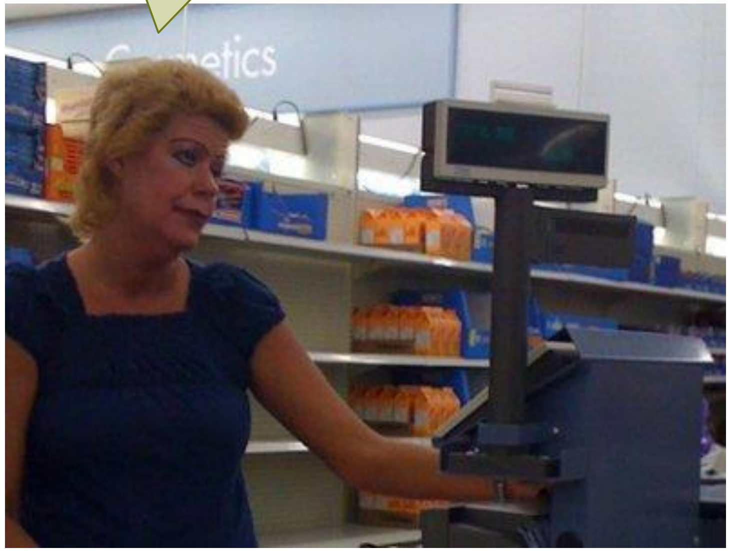
$ 3,000.00

http://bitchmagazine.org/post/beyond-the-panel-an-interview-with-danielle-corsetto-of-girls-with-slingshots

Adopted from Jose Nelson Amaral, “How did this get published?”, LCTES’12

So, a computing scientist entered a store….

Ma’am you cannot take the arithmetic average of percentages!



http://www.businessinsider.com/10-ways-to-fix-googles-busted-android-app-market-2010-1?op=1

$ 200.00

50%
Discount
Buy Now!

10%
Discount
Buy Now!

$ 3,000.00

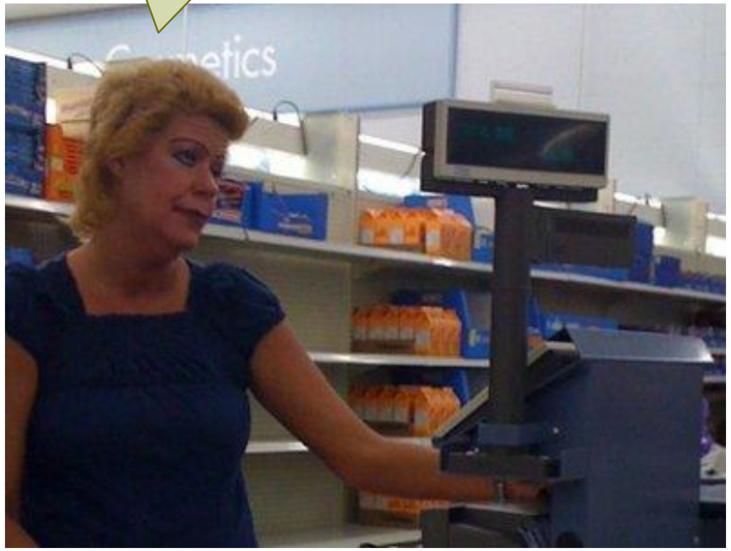
http://bitchmagazine.org/post/beyond-the-panel-an-interview-with-danielle-corsetto-of-girls-with-slingshots

Adopted from Jose Nelson Amaral, “How did this get published?”, LCTES’12

So, a computing scientist entered a store….

Ma’am you cannot take the arithmetic average of percentages!

But… I just came from at top CS conference in San Jose where they do it!



http://www.businessinsider.com/10-ways-to-fix-googles-busted-android-app-market-2010-1?op=1

$ 200.00

50%
Discount
Buy Now!

10%
Discount
Buy Now!

$ 3,000.00

http://bitchmagazine.org/post/beyond-the-panel-an-interview-with-danielle-corsetto-of-girls-with-slingshots

Adopted from Jose Nelson Amaral, “How did this get published?”, LCTES’12

Scientific Benchmarking: The fallacies of summarizing (Rules 3+4)

	System A			System B		
Testcase	I	II	III	I	II	III
Floating-point operations [Gflop]	10,0	15,0	20,0	10,0	15,0	20,0

TH, R. Belli: Scientific Benchmarking of Parallel Computing Systems, IEEE/ACM SC15 (full talk at https://www.youtube.com/watch?v=HwEpXIWAWTU)

Scientific Benchmarking: The fallacies of summarizing (Rules 3+4)

	System A			System B		
Testcase	I	II	III	I	II	III
Floating-point operations [Gflop]	10,0	15,0	20,0	10,0	15,0	20,0
Time [seconds]	**2,0**	**1,3**	**0,7**	**1,1**	**1,3**	**2,0**

TH, R. Belli: Scientific Benchmarking of Parallel Computing Systems, IEEE/ACM SC15 (full talk at https://www.youtube.com/watch?v=HwEpXIWAWTU)

Scientific Benchmarking: The fallacies of summarizing (Rules 3+4)

	System A			System B		
Testcase	I	II	III	I	II	III
Floating-point operations [Gflop]	10,0	15,0	20,0	10,0	15,0	20,0
Time [seconds]	**2,0**	**1,3**	**0,7**	**1,1**	**1,3**	**2,0**
Flop Rate [Gflop/s]	5,0	11,5	28,6	9,1	11,5	10,0

TH, R. Belli: Scientific Benchmarking of Parallel Computing Systems, IEEE/ACM SC15 (full talk at https://www.youtube.com/watch?v=HwEpXIWAWTU)

Scientific Benchmarking: The fallacies of summarizing (Rules 3+4)

	System A			System B		
Testcase	I	II	III	I	II	III
Floating-point operations [Gflop]	10,0	15,0	20,0	10,0	15,0	20,0
Time [seconds]	**2,0**	**1,3**	**0,7**	**1,1**	**1,3**	**2,0**
Flop Rate [Gflop/s]	5,0	11,5	28,6	9,1	11,5	10,0
Arithmetic Mean of Flop Rates [Gflop/s]	15,0			10,2		

TH, R. Belli: Scientific Benchmarking of Parallel Computing Systems, IEEE/ACM SC15 (full talk at https://www.youtube.com/watch?v=HwEpXIWAWTU)

Scientific Benchmarking: The fallacies of summarizing (Rules 3+4)

	System A			System B		
Testcase	I	II	III	I	II	III
Floating-point operations [Gflop]	10,0	15,0	20,0	10,0	15,0	20,0
Time [seconds]	**2,0**	**1,3**	**0,7**	**1,1**	**1,3**	**2,0**
Flop Rate [Gflop/s]	5,0	11,5	28,6	9,1	11,5	10,0
Arithmetic Mean of Flop Rates [Gflop/s]	15,0			10,2		
Harmonic Mean of Flop Rates [Gflop/s]	9,3			10,1		

TH, R. Belli: Scientific Benchmarking of Parallel Computing Systems, IEEE/ACM SC15 (full talk at https://www.youtube.com/watch?v=HwEpXIWAWTU)

Scientific Benchmarking: The fallacies of summarizing (Rules 3+4)

	System A			System B		
Testcase	I	II	III	I	II	III
Floating-point operations [Gflop]	10,0	15,0	20,0	10,0	15,0	20,0
Time [seconds]	**2,0**	**1,3**	**0,7**	**1,1**	**1,3**	**2,0**
Flop Rate [Gflop/s]	5,0	11,5	28,6	9,1	11,5	10,0
Arithmetic Mean of Flop Rates [Gflop/s]	15,0			10,2		
Harmonic Mean of Flop Rates [Gflop/s]	9,3			10,1		
Flop Rate by Dividing Totals [Gflop/s]	11,3			10,2		

TH, R. Belli: Scientific Benchmarking of Parallel Computing Systems, IEEE/ACM SC15 (full talk at https://www.youtube.com/watch?v=HwEpXIWAWTU)

RULE 3 and 4

Rule 3: *Use the arithmetic mean only for summarizing costs. Use the harmonic mean for summarizing rates.*

Rule 4: *Avoid summarizing ratios (e.g., speedup); summarize the costs or rates that the ratios base on instead. Only if these are not available use the geometric mean for summarizing ratios.*

					B	
Testcase						III
Floating-point o						20,0
Time [seconds]	**2,0**	**1,3**	**0,7**	**1,1**	**1,3**	**2,0**
Flop Rate [Gflop						10,0
Arithmetic Mea						
Harmonic Mear						
Flop Rate by Dividing Totals [Gflop/s]	11,3			10,2		

TH, R. Belli: Scientific Benchmarking of Parallel Computing Systems, IEEE/ACM SC15 (full talk at https://www.youtube.com/watch?v=HwEpXIWAWTU)

Administrivia

- **First project presentation: 10/29 (two weeks from now!)**
  - First presentation to gather feedback

    You already know what your peers are doing, now let us know

  - Some more ideas what to talk about:

    What tools/programming language/parallelization scheme do you use?

    Which architecture? (we only offer access to Xeon Phi, you may use different)

    How to verify correctness of the parallelization?

    How to argue about performance (bounds, what to compare to?)

    (Somewhat) realistic use-cases and input sets?

    What are the key concepts employed?

    What are the main obstacles?

Review of last lecture

- **Directory-based cache coherence**
  - Simple working with presence/dirty bits
  - Case study with Xeon Phi

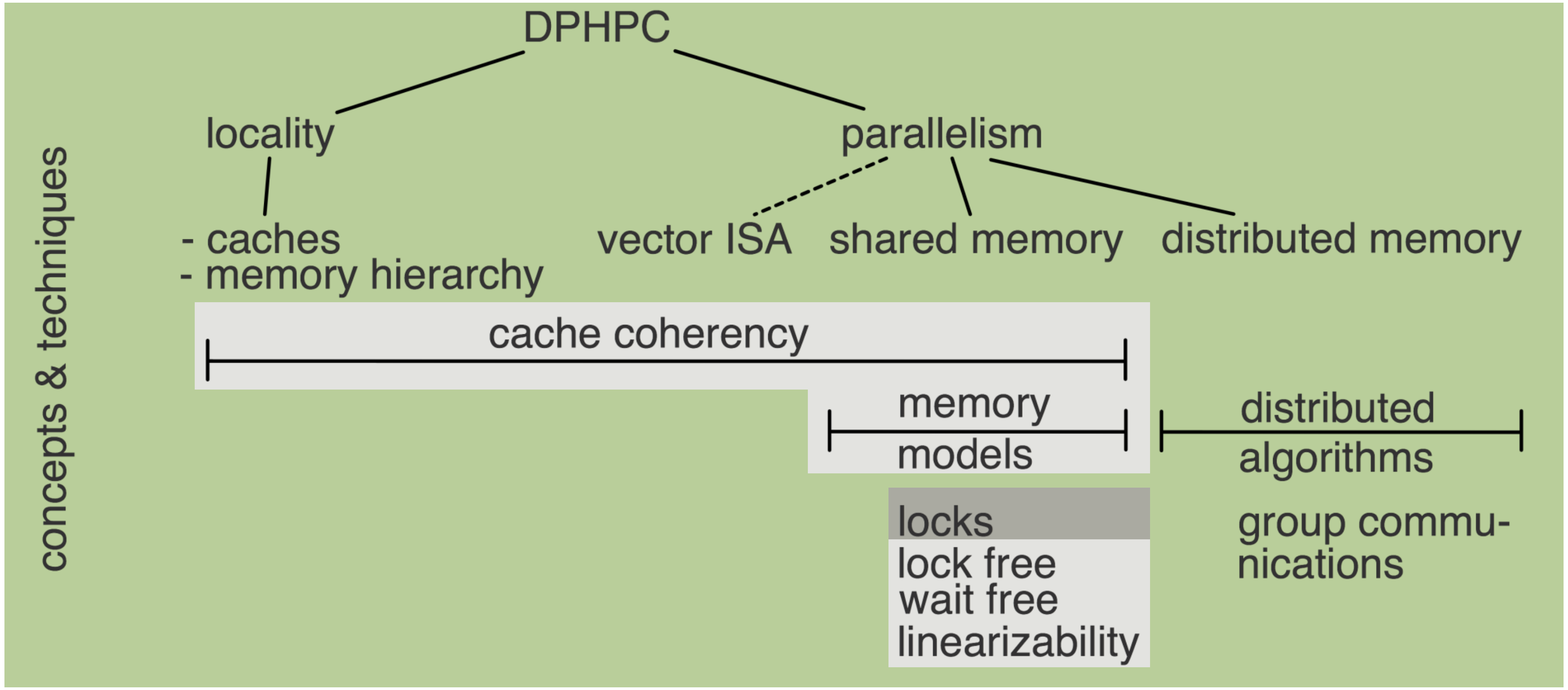
  Illustrates performance impact of the protocol and its importance!

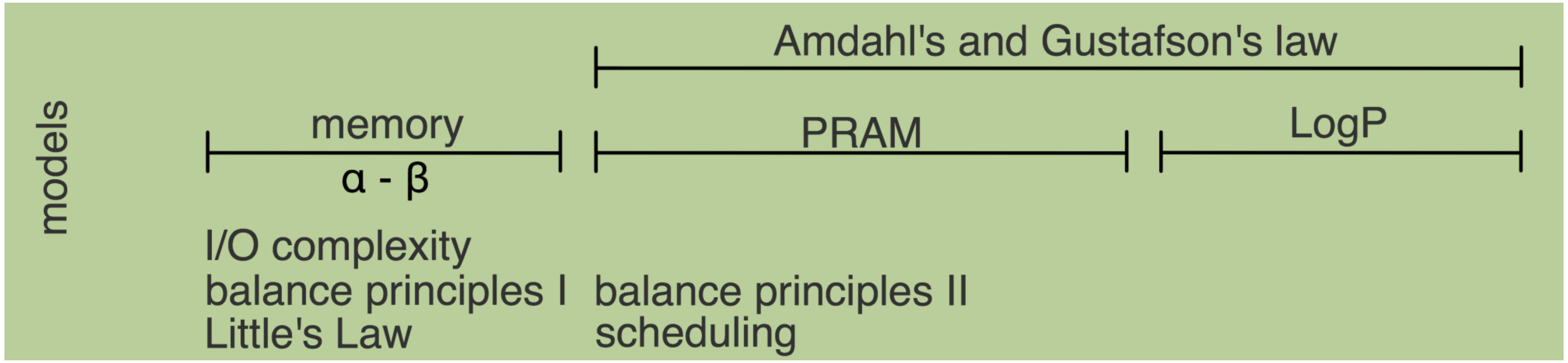
- **Memory models**
  - Ordering between accesses to different variables
  - Sequential consistency – nice but unrealistic

  Demonstrate how it prevents compiler and architectural optimizations

- **Practical memory models**
  - Overview of various models (TSO, PSO, RMO, … existing CPUs)
  - Case study of x86 (continuing today)

DPHPC Overview





Goals of this lecture

- **Recap: Correctness in parallel programs**
  - Covered in PP, here a slimmed down version to make the DPHPC lecture self-contained
    Watch for the green bar on the right side
- **Languages and Memory Models**
  - Java/C++ definition
- **Recap sequential consistency from the programmer's perspective**
  - Races (now in practice)
  - Synchronization variables (now in practice)
- **Mutual exclusion**
  - Recap – simple lock properties
  - Proving correctness in SC and memory models (x86)
  - Locks in practice – performance overhead of memory models!
- **Fast (actually practical) locks**
  - CLH – queue locks
  - MCS – "cache coherence optimal" queue locking

The Eight x86 Principles

1. “Reads are not reordered with other reads.” (R→R)

The Eight x86 Principles

1. “Reads are not reordered with other reads.” (R→R)
2. “Writes are not reordered with other writes.” (W→W)

The Eight x86 Principles

1. “Reads are not reordered with other reads.” (R→R)
2. “Writes are not reordered with other writes.” (W→W)
3. “Writes are not reordered with older reads.” (R→W)

The Eight x86 Principles

1. “Reads are not reordered with other reads.” (R→R)
2. “Writes are not reordered with other writes.” (W→W)
3. “Writes are not reordered with older reads.” (R→W)
4. “Reads may be reordered with older writes to different locations but not with older writes to the same location.” (NO W→R!)

The Eight x86 Principles

1. **“Reads are not reordered with other reads.” (R→R)**
2. **“Writes are not reordered with other writes.” (W→W)**
3. **“Writes are not reordered with older reads.” (R→W)**
4. **“Reads may be reordered with older writes to different locations but not with older writes to the same location.” (NO W→R!)**
5. **“In a multiprocessor system, memory ordering obeys causality.“ (memory ordering respects transitive visibility)**

The Eight x86 Principles

1. “Reads are not reordered with other reads.” (R→R)
2. “Writes are not reordered with other writes.” (W→W)
3. “Writes are not reordered with older reads.” (R→W)
4. “Reads may be reordered with older writes to different locations but not with older writes to the same location.” (NO W→R!)
5. “In a multiprocessor system, memory ordering obeys causality.“ (memory ordering respects transitive visibility)
6. “In a multiprocessor system, writes to the same location have a total order.” (implied by cache coherence)

The Eight x86 Principles

1. “Reads are not reordered with other reads.” (R→R)
2. “Writes are not reordered with other writes.” (W→W)
3. “Writes are not reordered with older reads.” (R→W)
4. “Reads may be reordered with older writes to different locations but not with older writes to the same location.” (NO W→R!)
5. “In a multiprocessor system, memory ordering obeys causality.“ (memory ordering respects transitive visibility)
6. “In a multiprocessor system, writes to the same location have a total order.” (implied by cache coherence)
7. “In a multiprocessor system, locked instructions have a total order.“ (enables synchronized programming!)

The Eight x86 Principles

1. “Reads are not reordered with other reads.” (R→R)
2. “Writes are not reordered with other writes.” (W→W)
3. “Writes are not reordered with older reads.” (R→W)
4. “Reads may be reordered with older writes to different locations but not with older writes to the same location.” (NO W→R!)
5. “In a multiprocessor system, memory ordering obeys causality.“ (memory ordering respects transitive visibility)
6. “In a multiprocessor system, writes to the same location have a total order.” (implied by cache coherence)
7. “In a multiprocessor system, locked instructions have a total order.“ (enables synchronized programming!)
8. “Reads and writes are not reordered with locked instructions. “ (enables synchronized programming!)

Principle 1 and 2

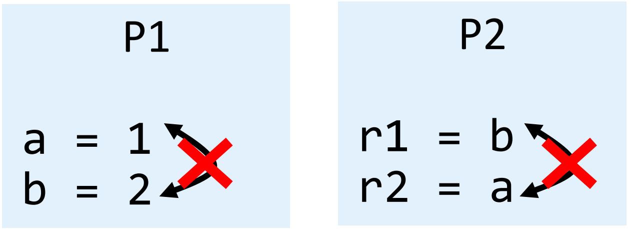
Reads are not reordered with other reads. (R→R)

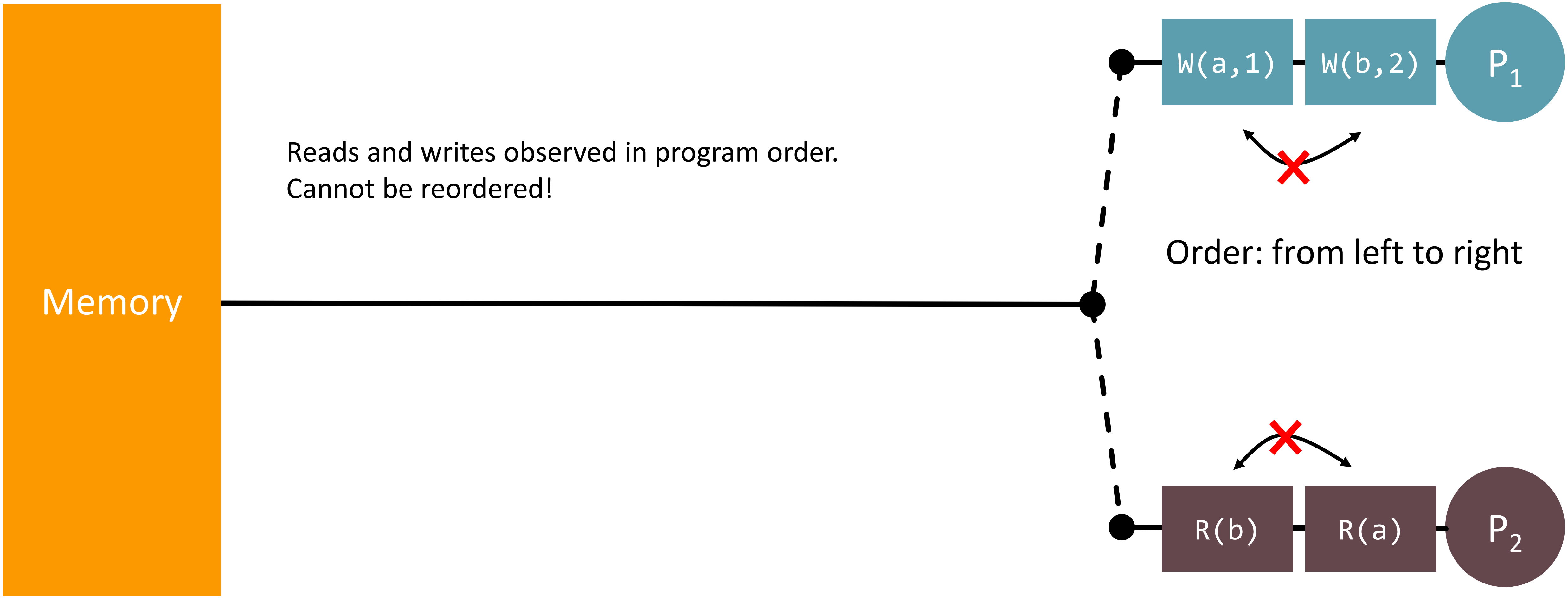
Writes are not reordered with other writes. (W→W)

All values zero initially. r1 and r2 are registers.

P1	P2

a = 1
b = 2
	r1 = b
	r2 = a





Reads and writes observed in program order.
Cannot be reordered!

Order: from left to right

Principle 1 and 2

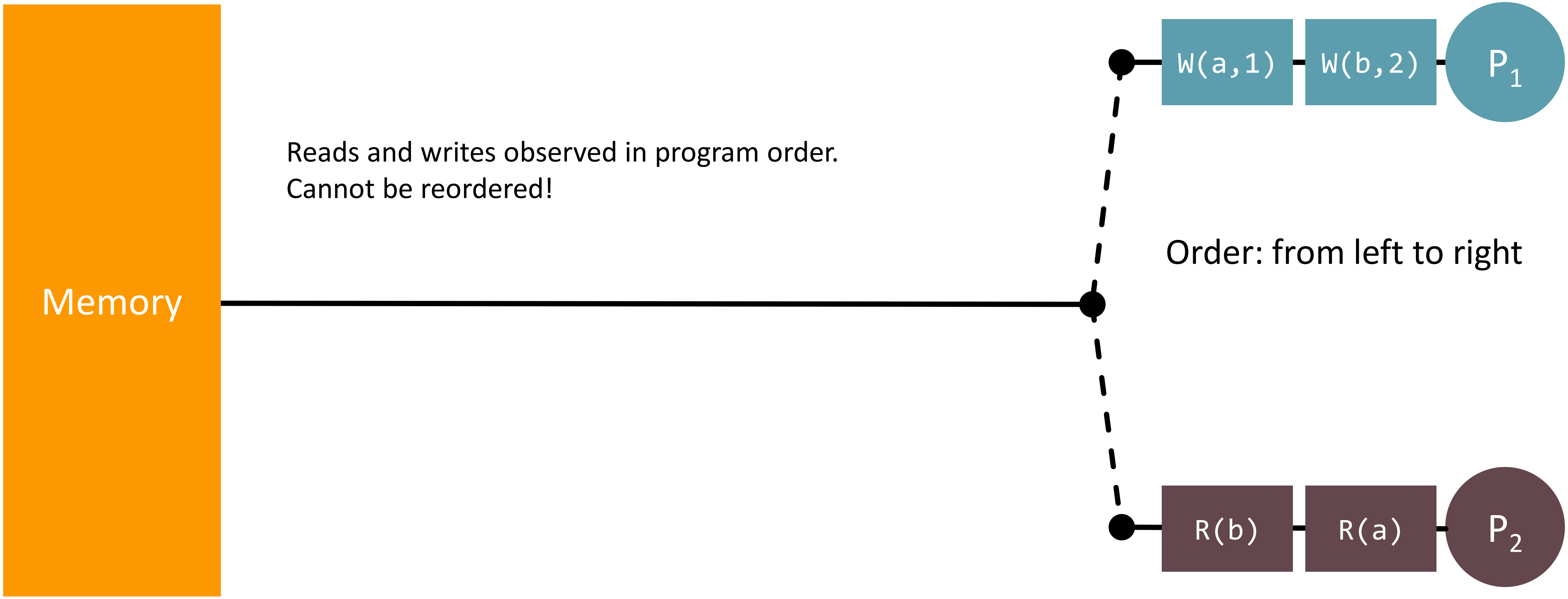
All values zero initially. r1 and r2 are registers.

Reads are not reordered with other reads. (R→R)

Writes are not reordered with other writes. (W→W)

```
P1

a = 1
b = 2
```

```
P2

r1 = b
r2 = a
```



Reads and writes observed in program order.
Cannot be reordered!

Order: from left to right

Principle 1 and 2

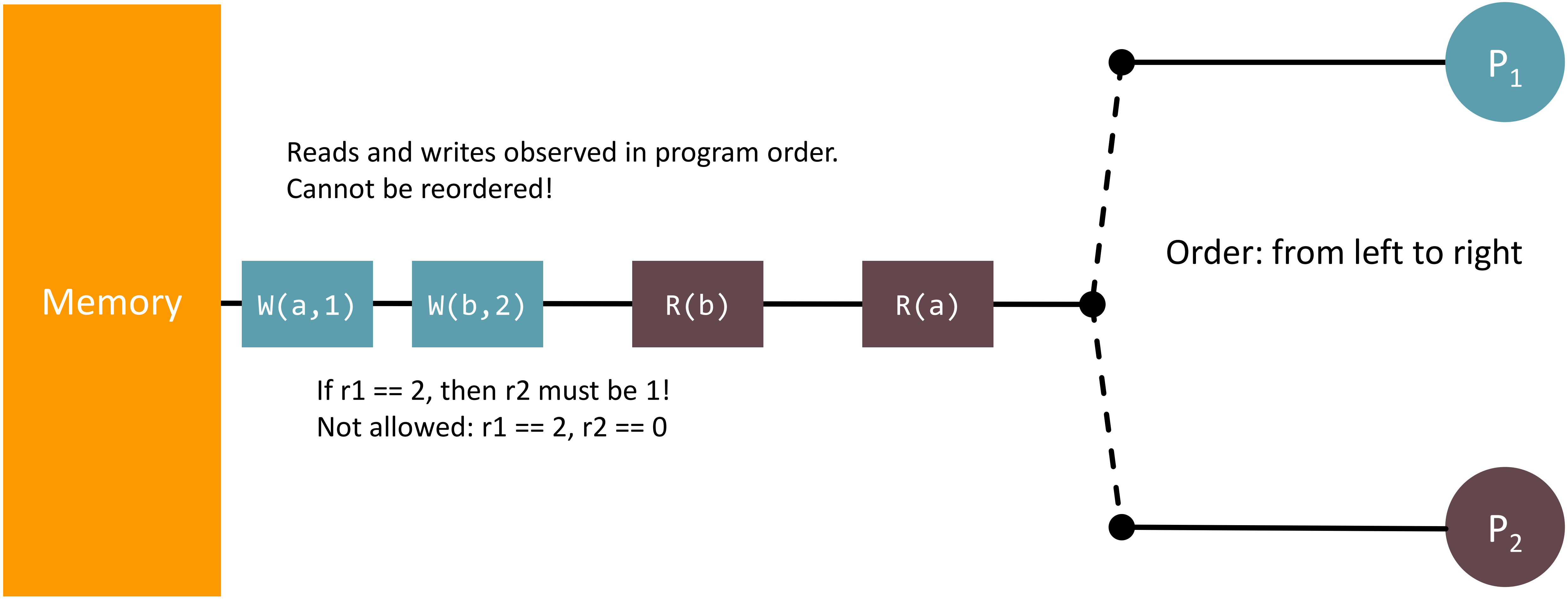
All values zero initially. r1 and r2 are registers.

```
P1

a = 1
b = 2
```

```
P2

r1 = b
r2 = a
```

Reads are not reordered with other reads. (R→R)

Writes are not reordered with other writes. (W→W)



Reads and writes observed in program order.
Cannot be reordered!

If r1 == 2, then r2 must be 1!
Not allowed: r1 == 2, r2 == 0

Order: from left to right

Principle 1 and 2

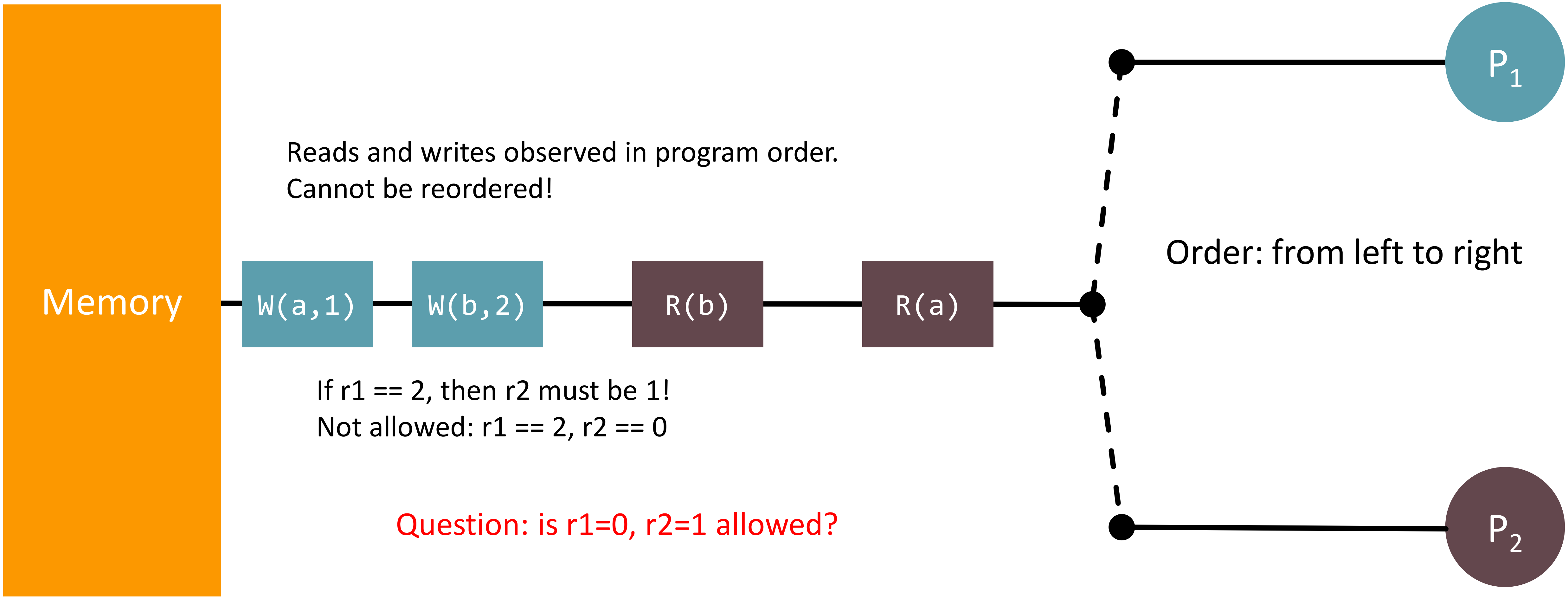
Reads are not reordered with other reads. (R→R)

Writes are not reordered with other writes. (W→W)

All values zero initially. r1 and r2 are registers.

P1	P2

`a = 1`
`b = 2`
	`r1 = b`
	`r2 = a`



Reads and writes observed in program order.
Cannot be reordered!

Order: from left to right

If r1 == 2, then r2 must be 1!
Not allowed: r1 == 2, r2 == 0

Question: is r1=0, r2=1 allowed?

Principle 3

Writes are not reordered with older reads. (R→W)

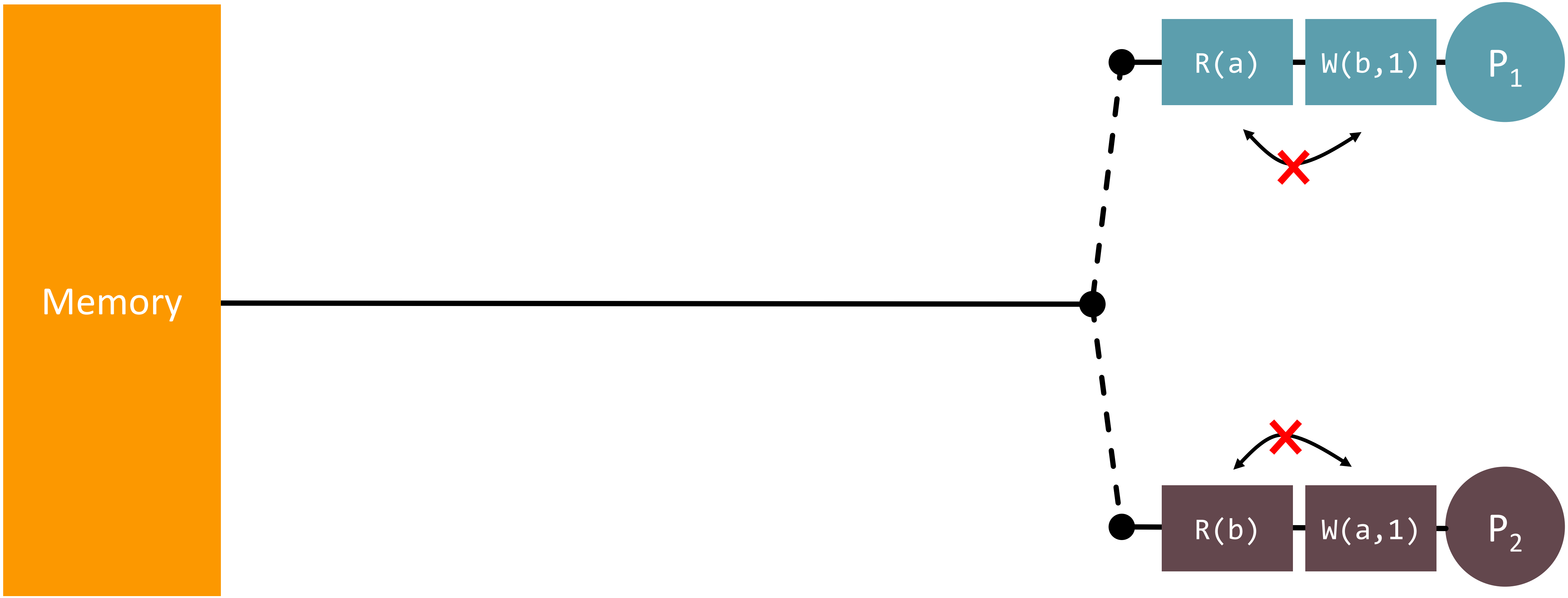
P1

```
r1 = a
b = 1
```

P2

```
r2 = b
a = 1
```

All values zero initially



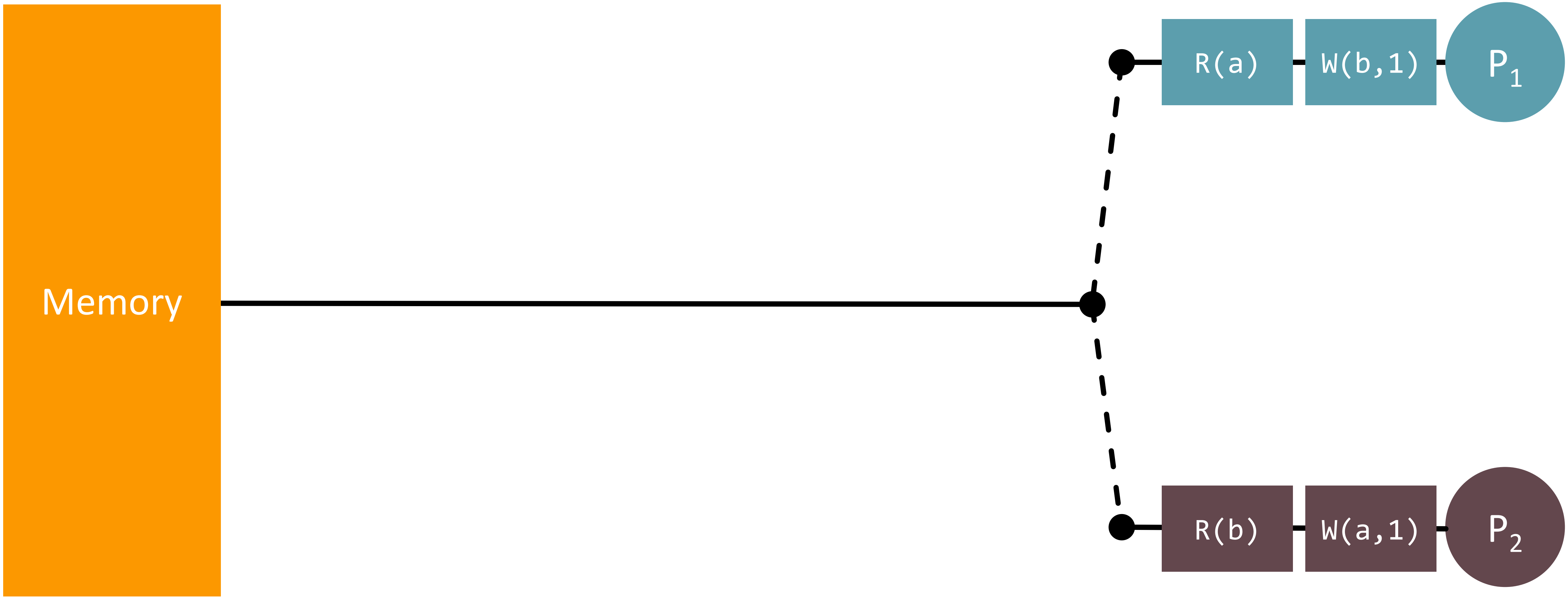
Principle 3

Writes are not reordered with older reads. (R→W)

```
P1

r1 = a
b = 1
```

```
P2

r2 = b
a = 1
```

All values zero initially



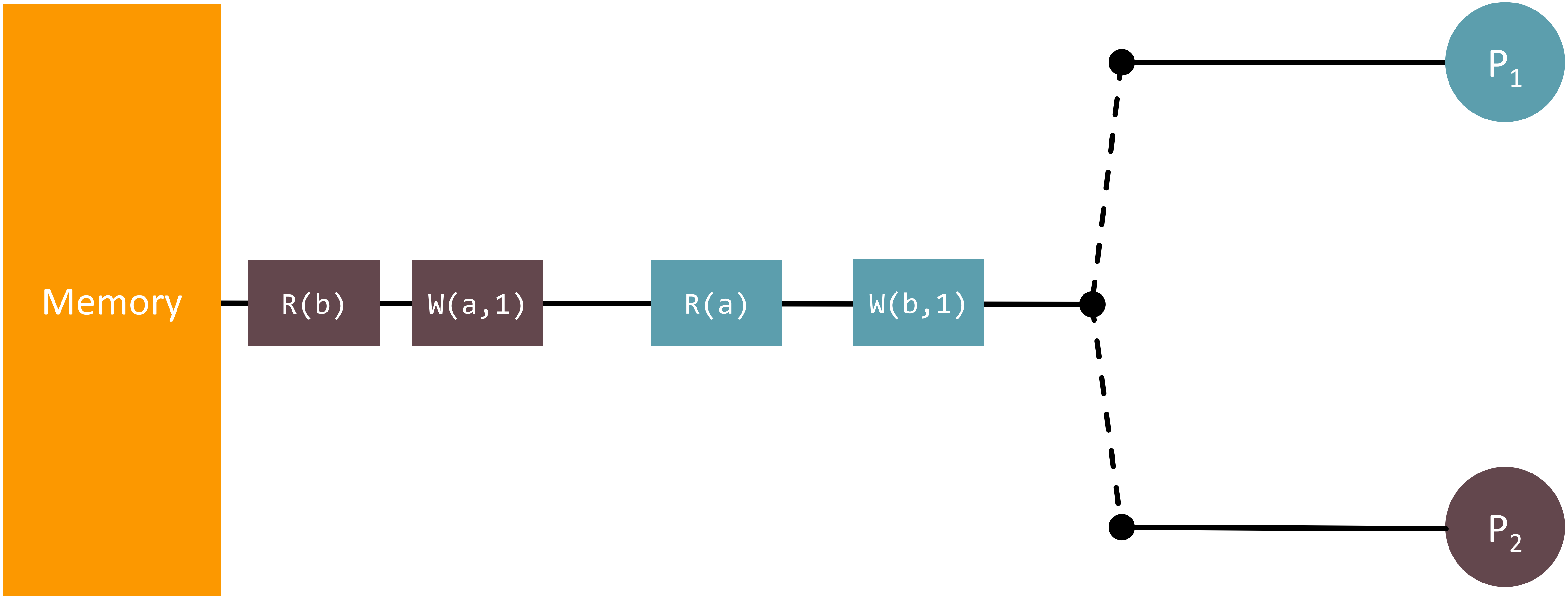
Principle 3

Writes are not reordered with older reads. (R→W)

```
P1

r1 = a
b = 1
```

```
P2

r2 = b
a = 1
```

All values zero initially



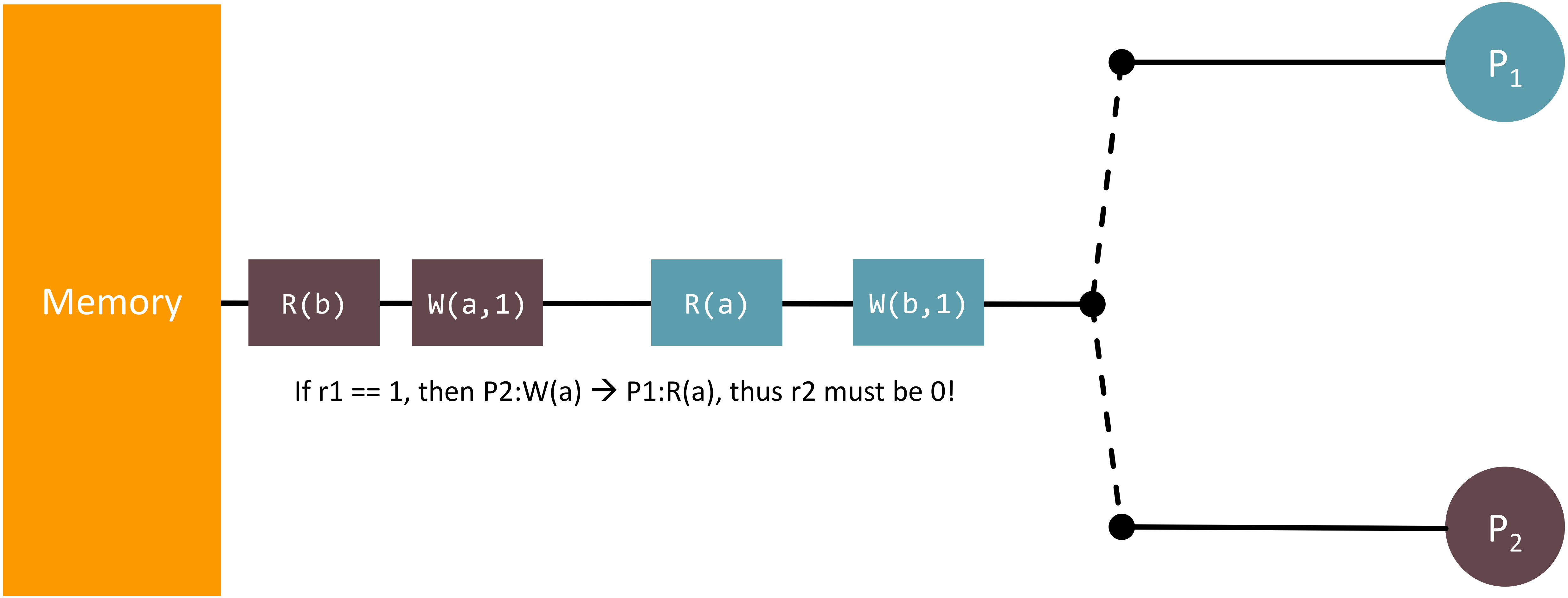
Principle 3

Writes are not reordered with older reads. (R→W)

```
P1

r1 = a
b = 1
```

```
P2

r2 = b
a = 1
```

All values zero initially



If r1 == 1, then P2:W(a) → P1:R(a), thus r2 must be 0!

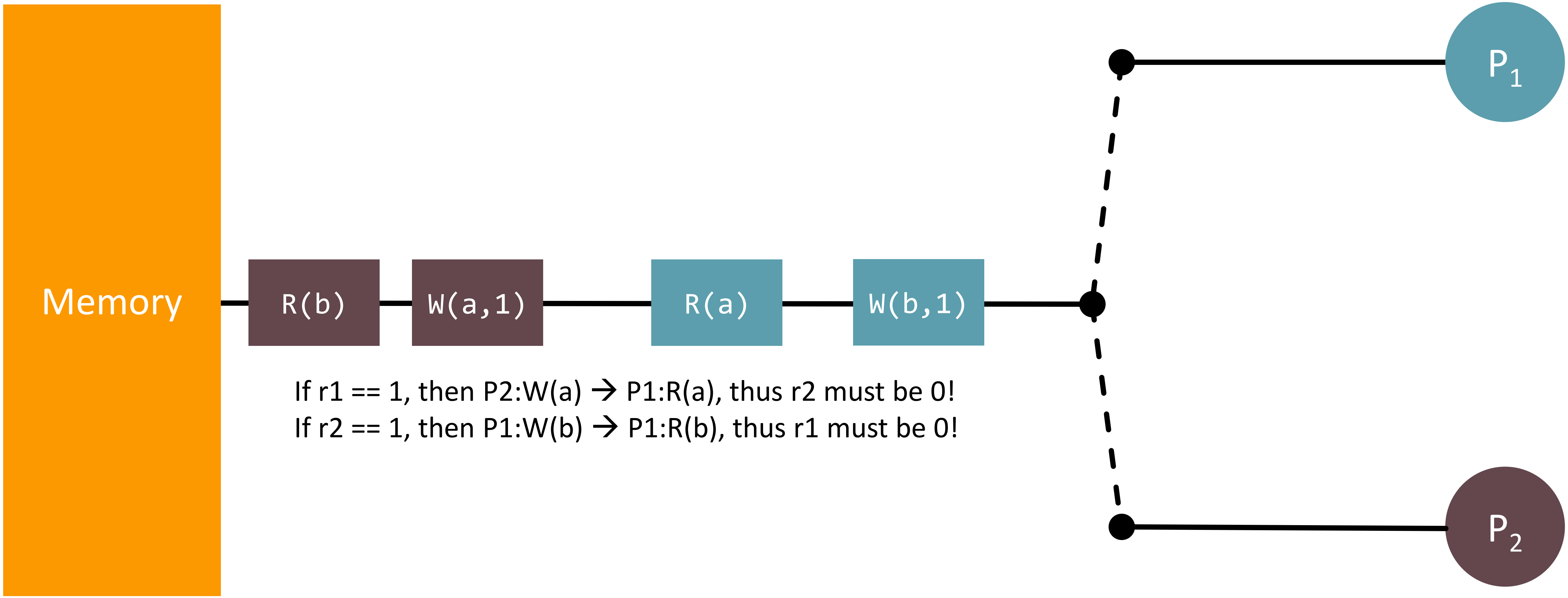
Principle 3

Writes are not reordered with older reads. (R→W)

```
P1

r1 = a
b = 1
```

```
P2

r2 = b
a = 1
```

All values zero initially



If r1 == 1, then P2:W(a) → P1:R(a), thus r2 must be 0!
If r2 == 1, then P1:W(b) → P1:R(b), thus r1 must be 0!

Principle 3

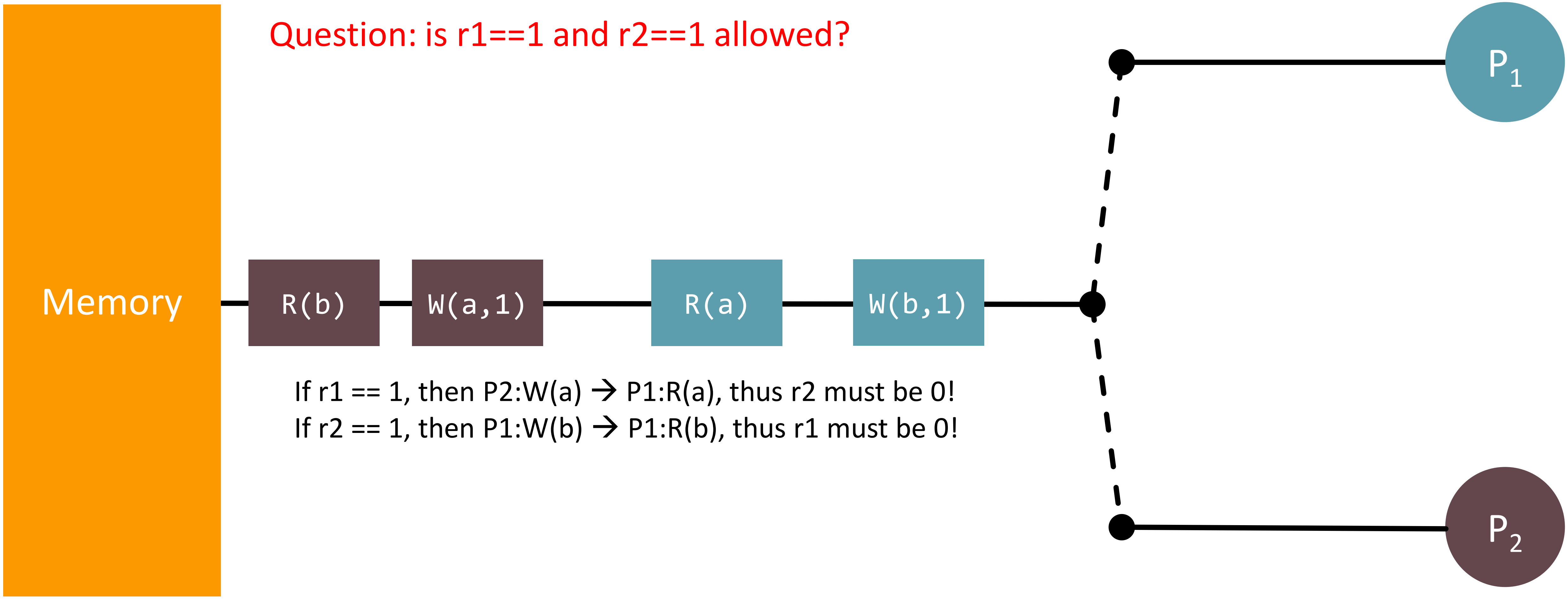
Writes are not reordered with older reads. (R→W)

```
P1

r1 = a
b = 1
```

```
P2

r2 = b
a = 1
```

All values zero initially

Question: is r1==1 and r2==1 allowed?



If r1 == 1, then P2:W(a) → P1:R(a), thus r2 must be 0!
If r2 == 1, then P1:W(b) → P1:R(b), thus r1 must be 0!

Principle 3

Writes are not reordered with older reads. (R→W)

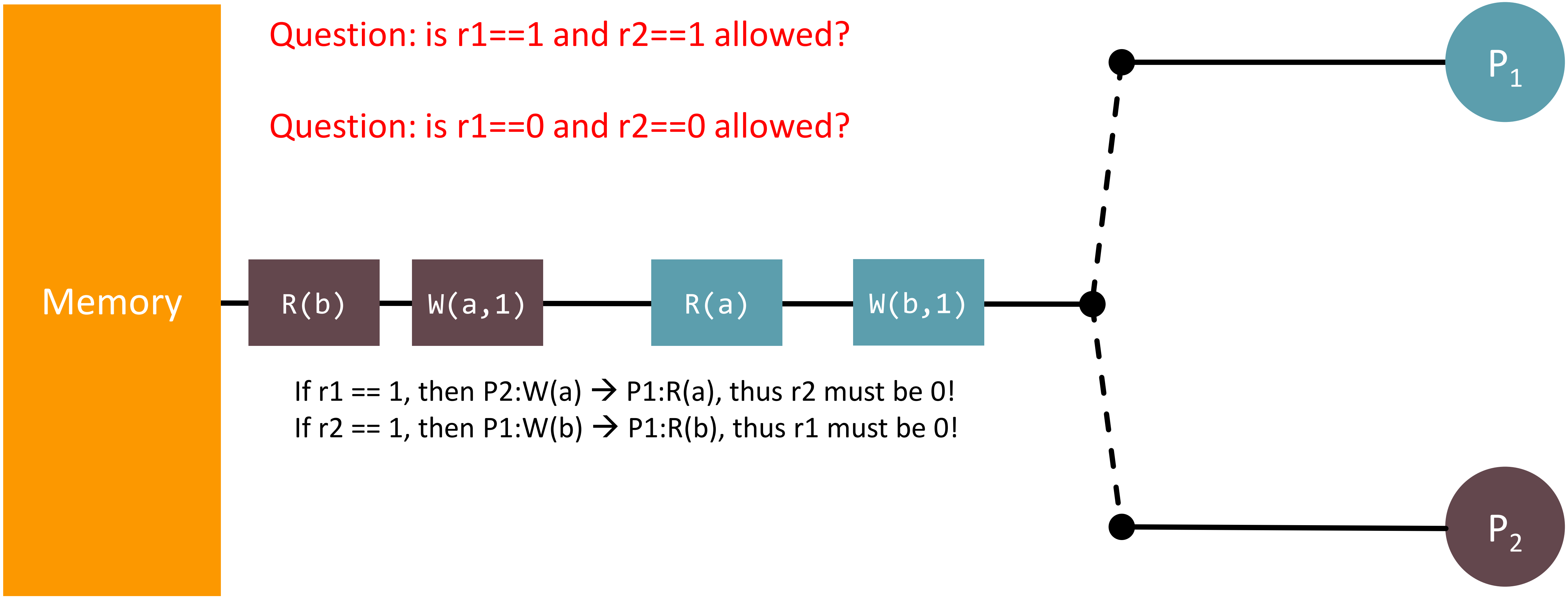
```
P1

r1 = a
b = 1
```

```
P2

r2 = b
a = 1
```

All values zero initially

Question: is r1==1 and r2==1 allowed?

Question: is r1==0 and r2==0 allowed?



If r1 == 1, then P2:W(a) → P1:R(a), thus r2 must be 0!
If r2 == 1, then P1:W(b) → P1:R(b), thus r1 must be 0!

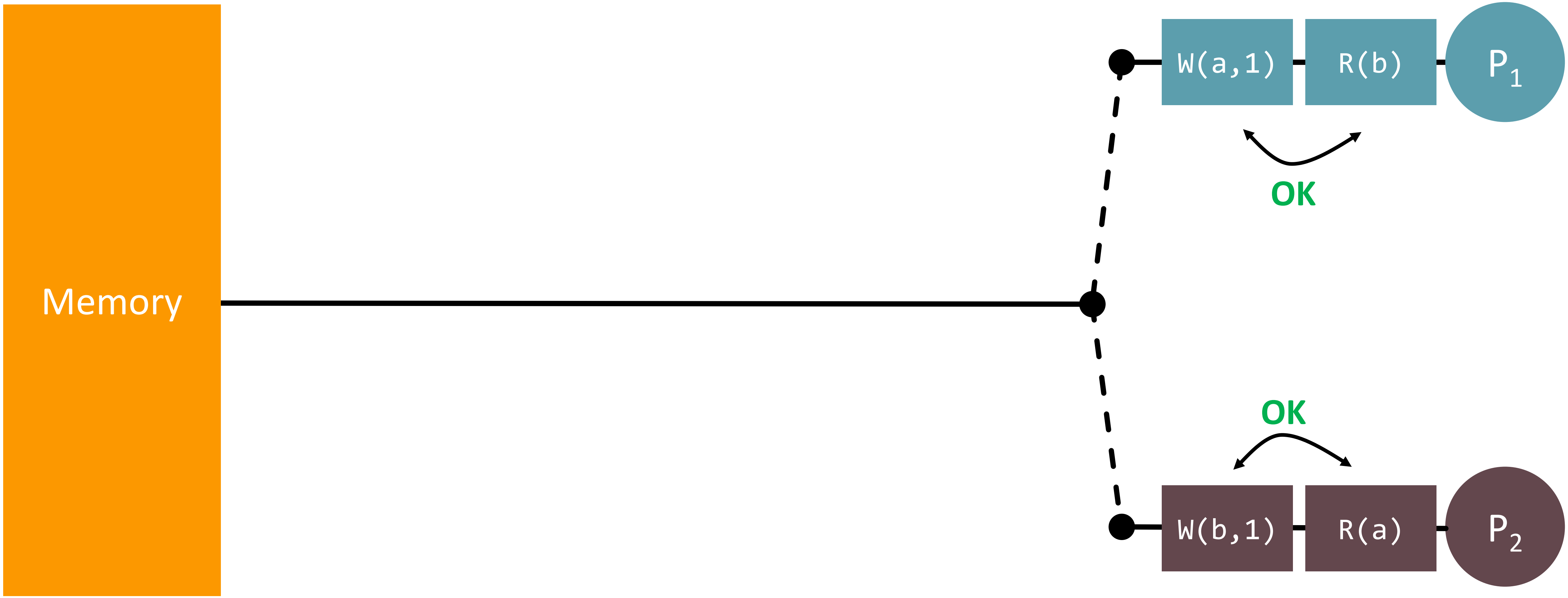
Principle 4

All values zero initially

Reads may be reordered with older writes to different locations but not with older writes to the same location. (NO W→R!)

```
P1

a  = 1
r1  = b
```

```
P2

b  = 1
r2  = a
```



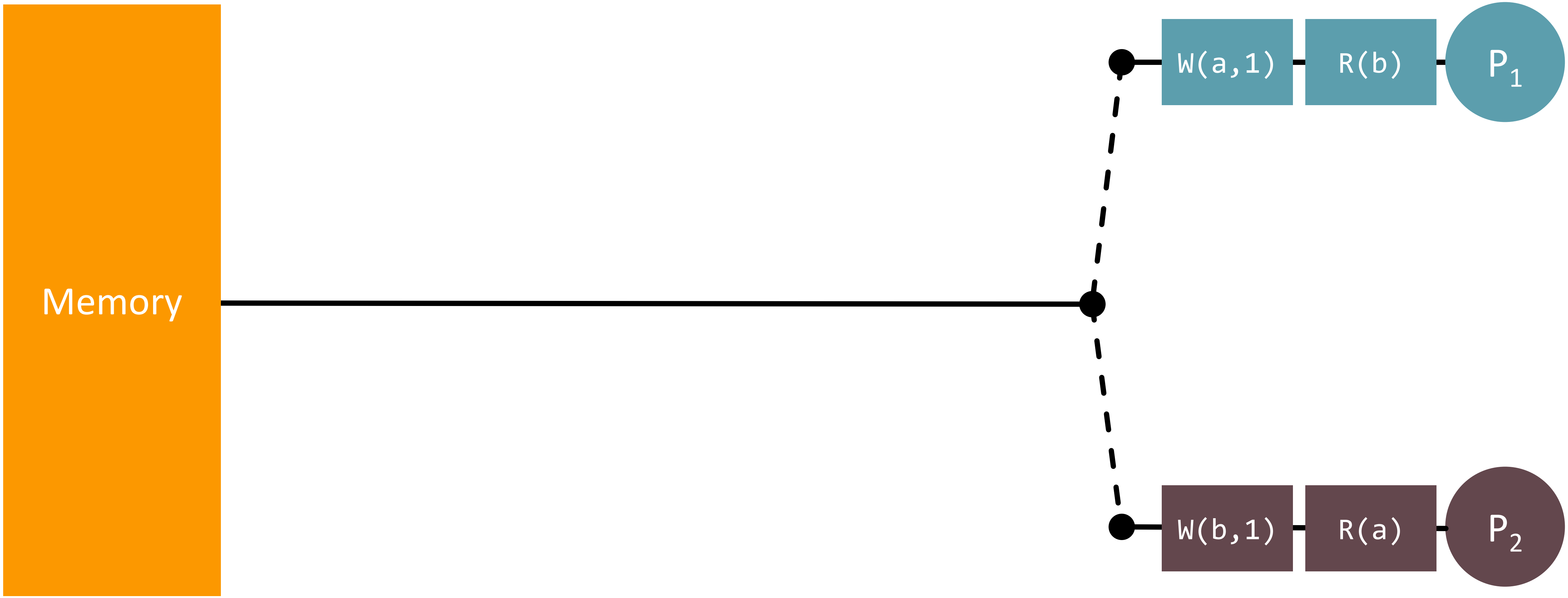
Principle 4

All values zero initially

Reads may be reordered with older writes to different locations but not with older writes to the same location. (NO W→R!)

```
P1

a  = 1
r1 = b
```

```
P2

b  = 1
r2 = a
```



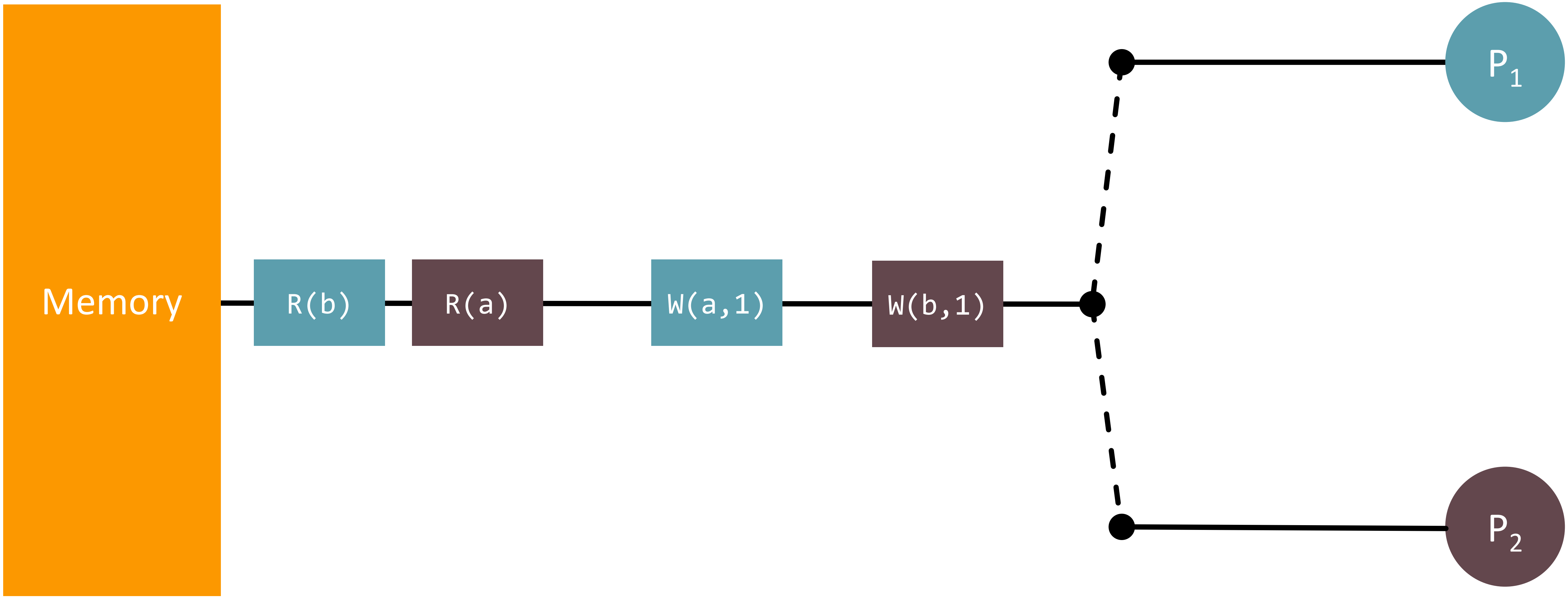
All values zero initially

Principle 4

Reads may be reordered with older writes to different locations but not with older writes to the same location. (NO W→R!)

P1	P2

a = 1
r1 = b
	b = 1
	r2 = a



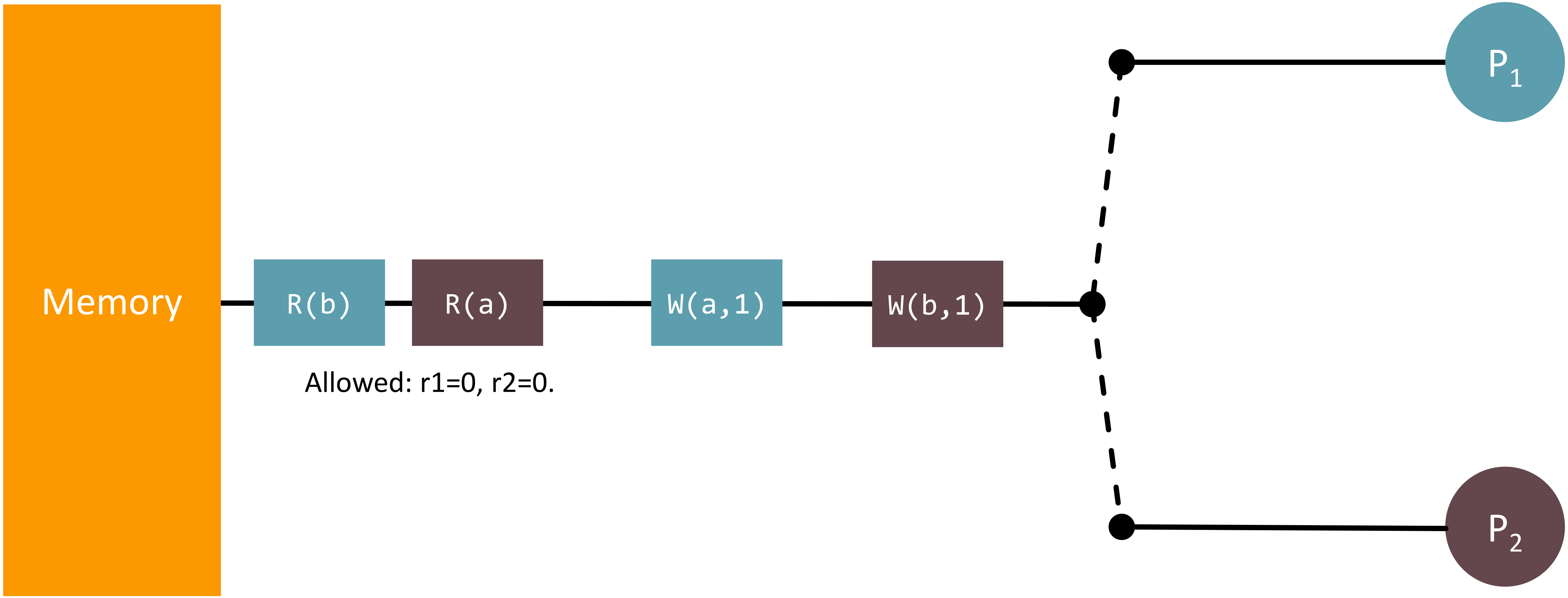
Principle 4

All values zero initially

Reads may be reordered with older writes to different locations but not with older writes to the same location. (NO W→R!)

```
P1

a = 1
r1 = b
```

```
P2

b = 1
r2 = a
```



Allowed: r1=0, r2=0.

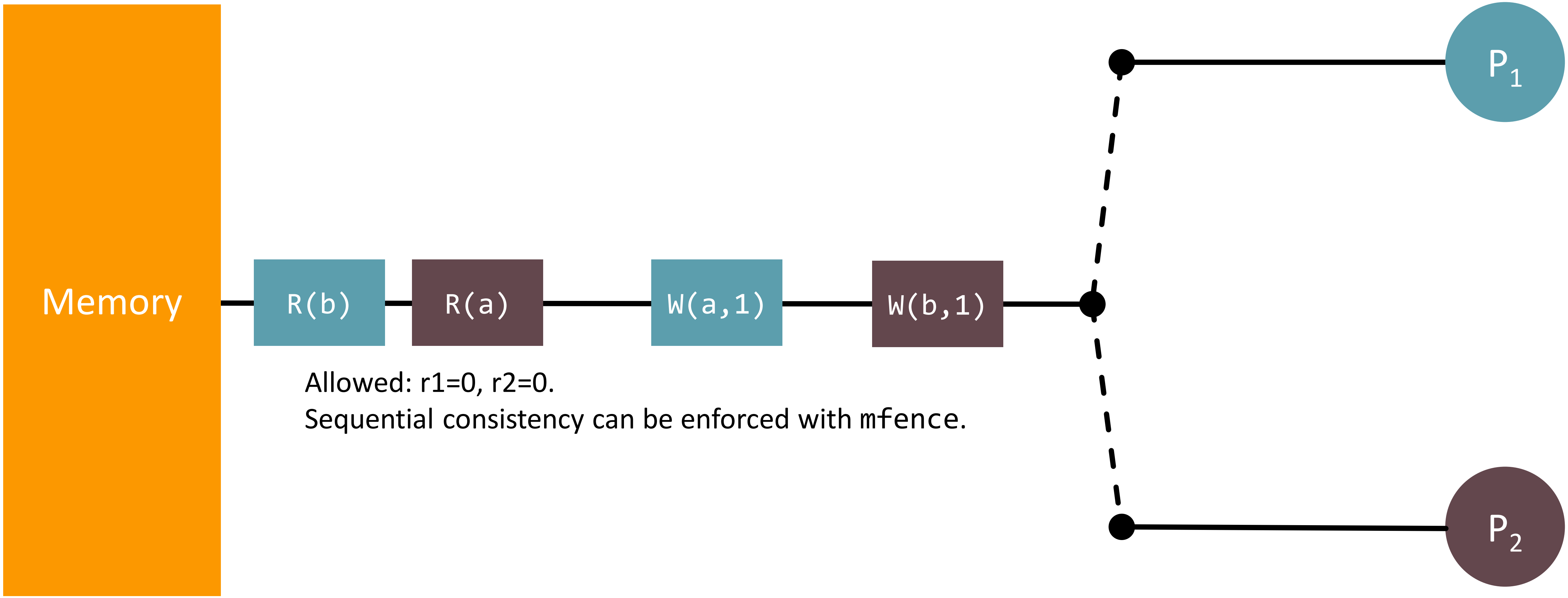
Principle 4

All values zero initially

Reads may be reordered with older writes to different locations but not with older writes to the same location. (NO W→R!)

```
P1

a  = 1
r1 = b
```

```
P2

b  = 1
r2 = a
```



Allowed: r1=0, r2=0.
Sequential consistency can be enforced with `mfence`.

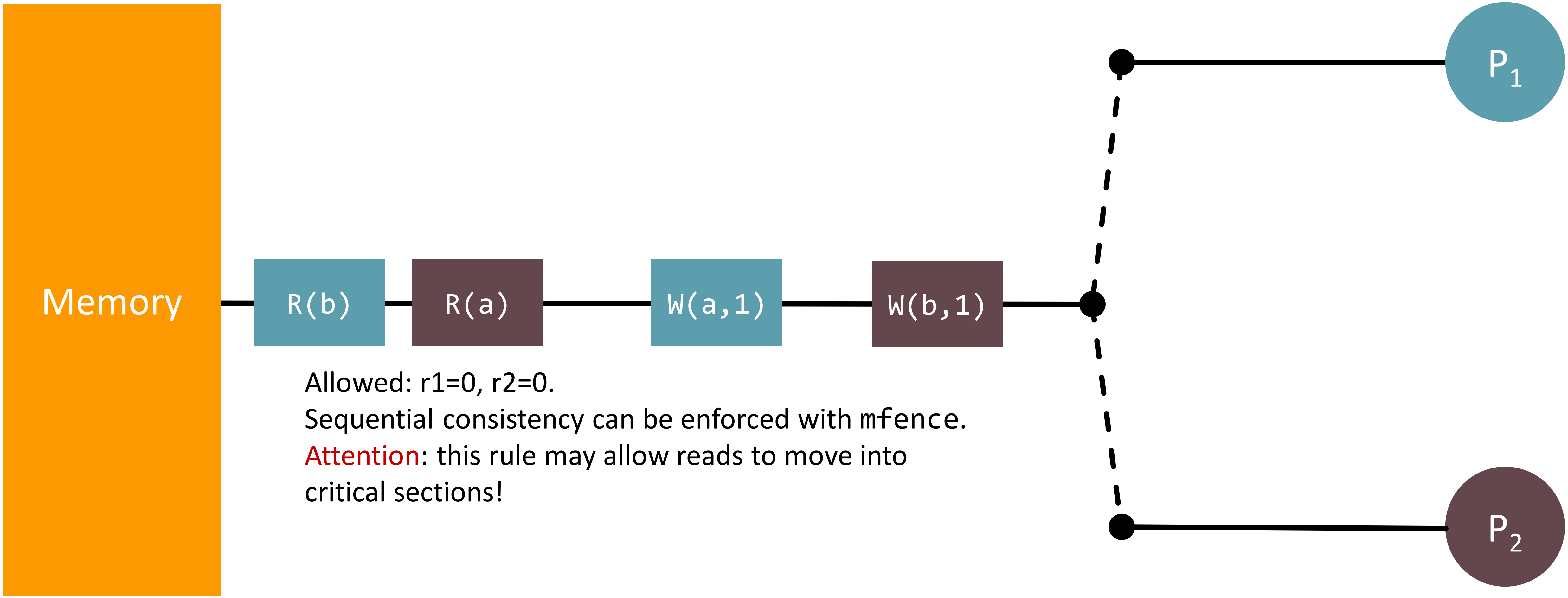
Principle 4

All values zero initially

Reads may be reordered with older writes to different locations but not with older writes to the same location. (NO W→R!)

P1	P2

`a = 1`
`r1 = b`
	`b = 1`
	`r2 = a`



Allowed: r1=0, r2=0.
Sequential consistency can be enforced with `mfence`.
Attention: this rule may allow reads to move into critical sections!

Principle 4

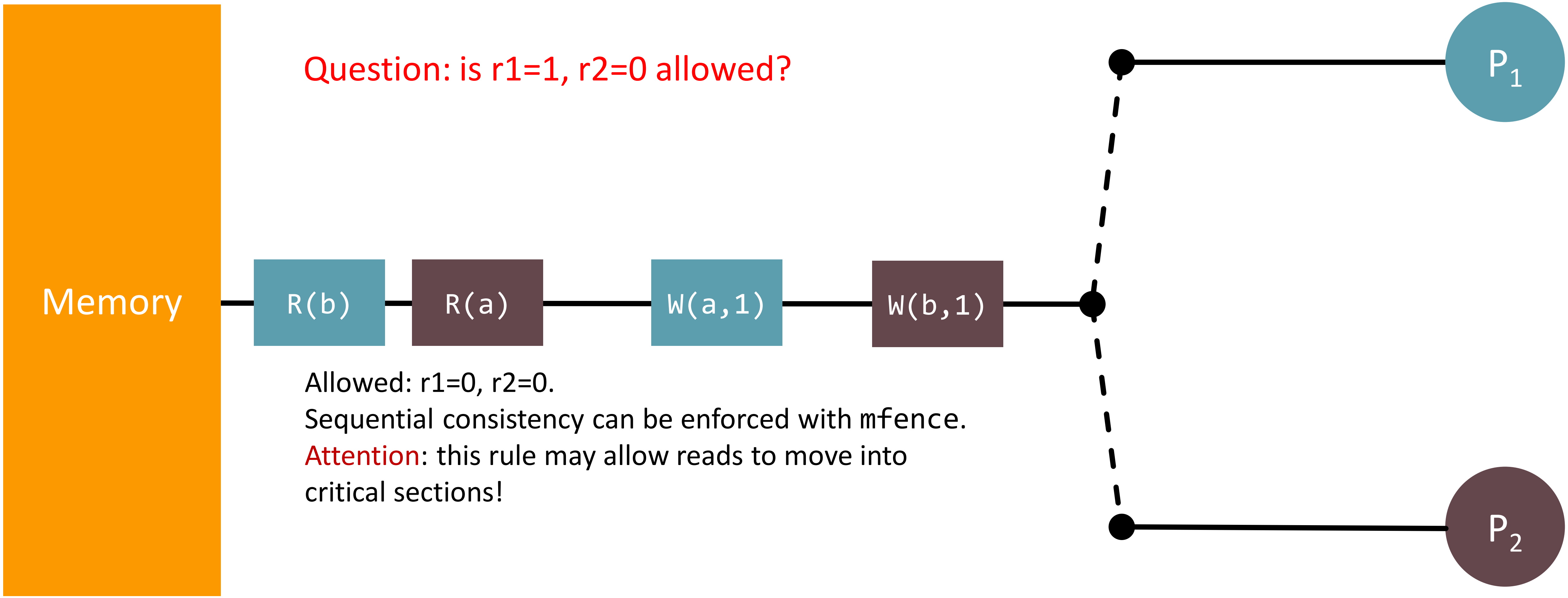
All values zero initially

Reads may be reordered with older writes to different locations but not with older writes to the same location. (NO W→R!)

```
P1

a  = 1
r1 = b
```

```
P2

b  = 1
r2 = a
```

Question: is r1=1, r2=0 allowed?



Allowed: r1=0, r2=0.
Sequential consistency can be enforced with `mfence`.
Attention: this rule may allow reads to move into critical sections!

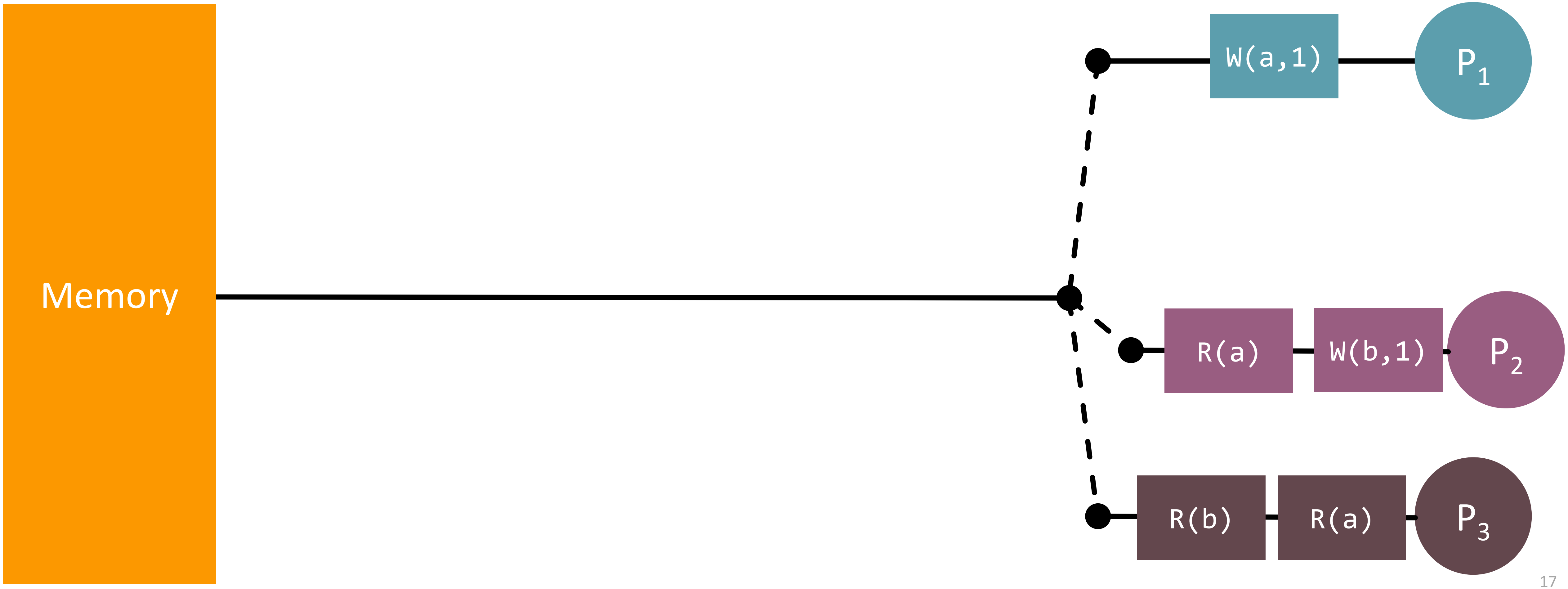
Principle 5

In a multiprocessor system, memory ordering obeys causality (memory ordering respects transitive visibility).

All values zero initially

P1	P2	P3

a = 1	r1 = a
	b = 1
		r2 = b
		r3 = a



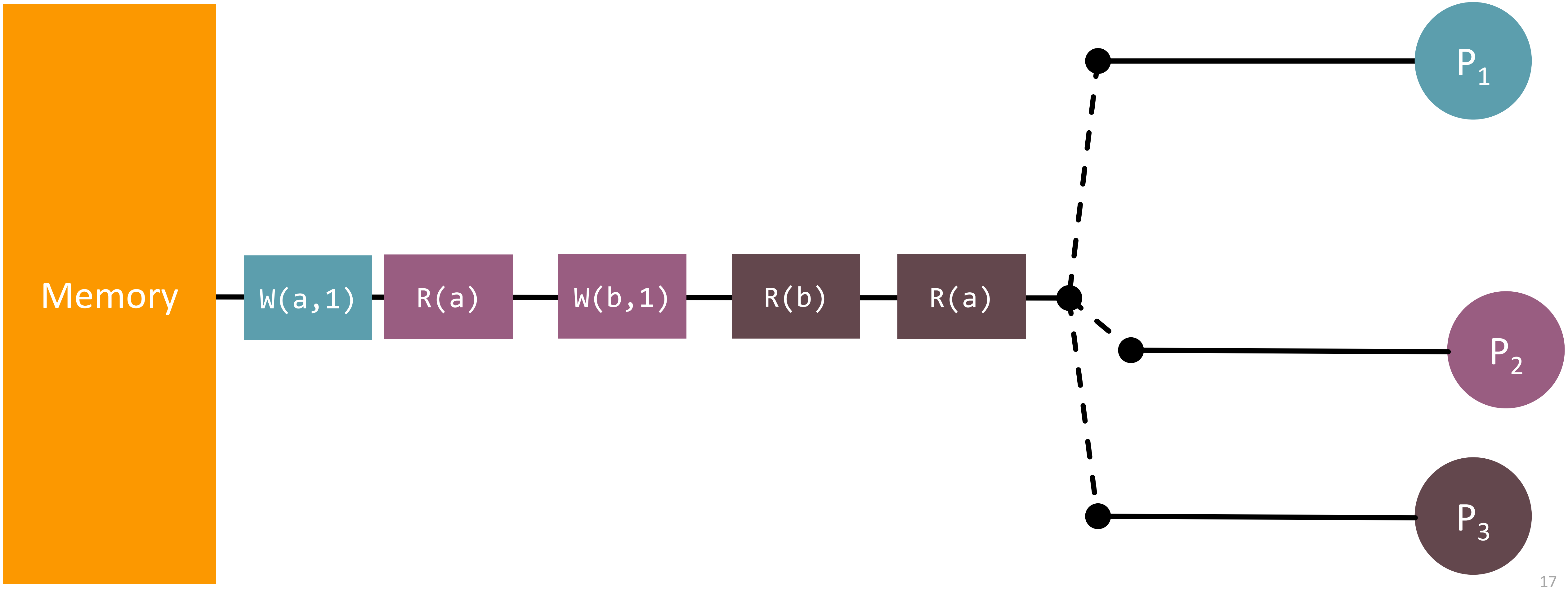
Principle 5

In a multiprocessor system, memory ordering obeys causality (memory ordering respects transitive visibility).

All values zero initially

P1	P2	P3

a = 1	r1 = a
	b = 1
		r2 = b
		r3 = a



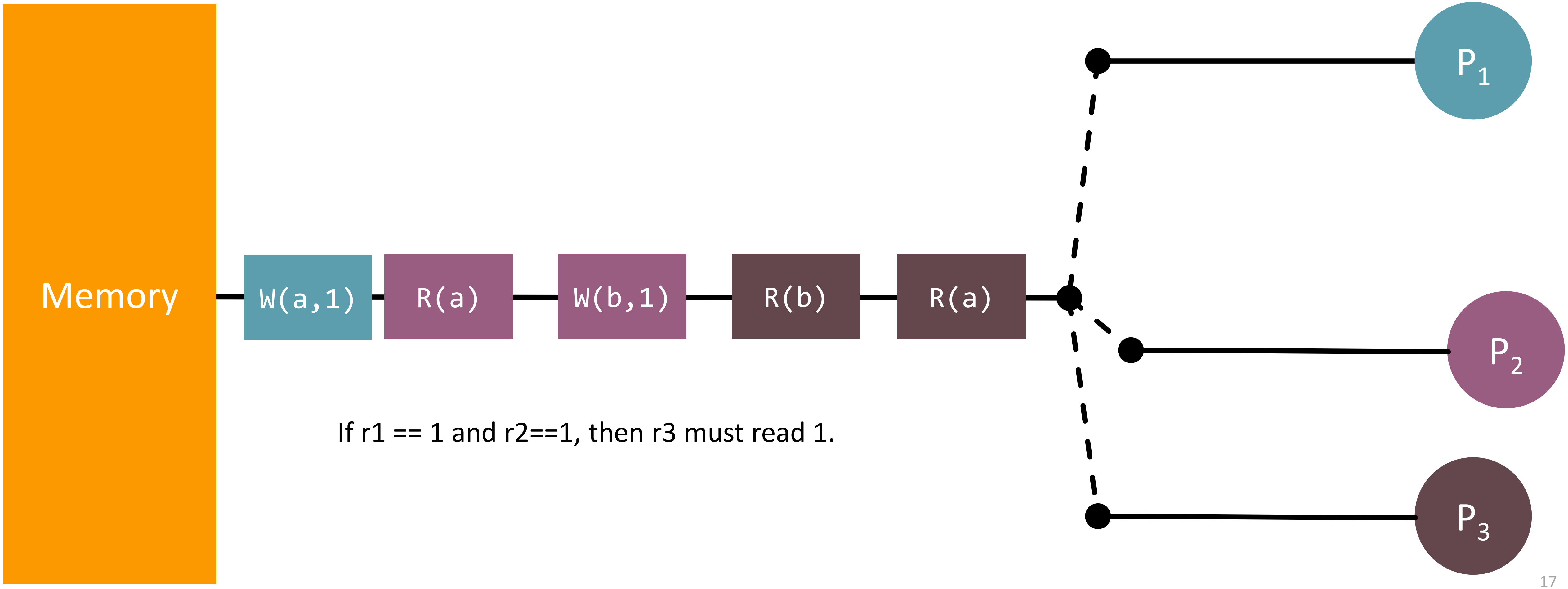
Principle 5

In a multiprocessor system, memory ordering obeys causality (memory ordering respects transitive visibility).

All values zero initially

P1	P2	P3

`a = 1`	`r1 = a`
	`b = 1`
		`r2 = b`
		`r3 = a`



If r1 == 1 and r2==1, then r3 must read 1.

Principle 5

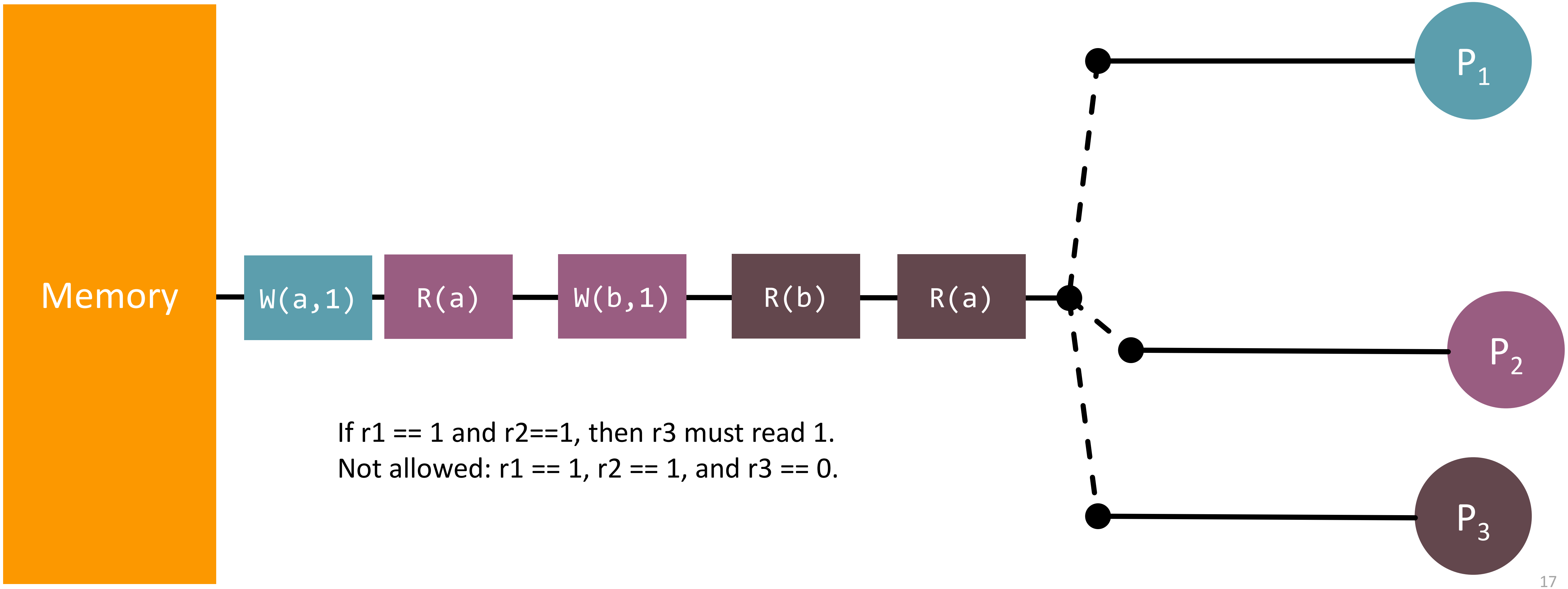
In a multiprocessor system, memory ordering obeys causality (memory ordering respects transitive visibility).

All values zero initially

```
P1

a = 1
```

```
P2

r1 = a
b = 1
```

```
P3


r2 = b
r3 = a
```



If r1 == 1 and r2==1, then r3 must read 1.
Not allowed: r1 == 1, r2 == 1, and r3 == 0.

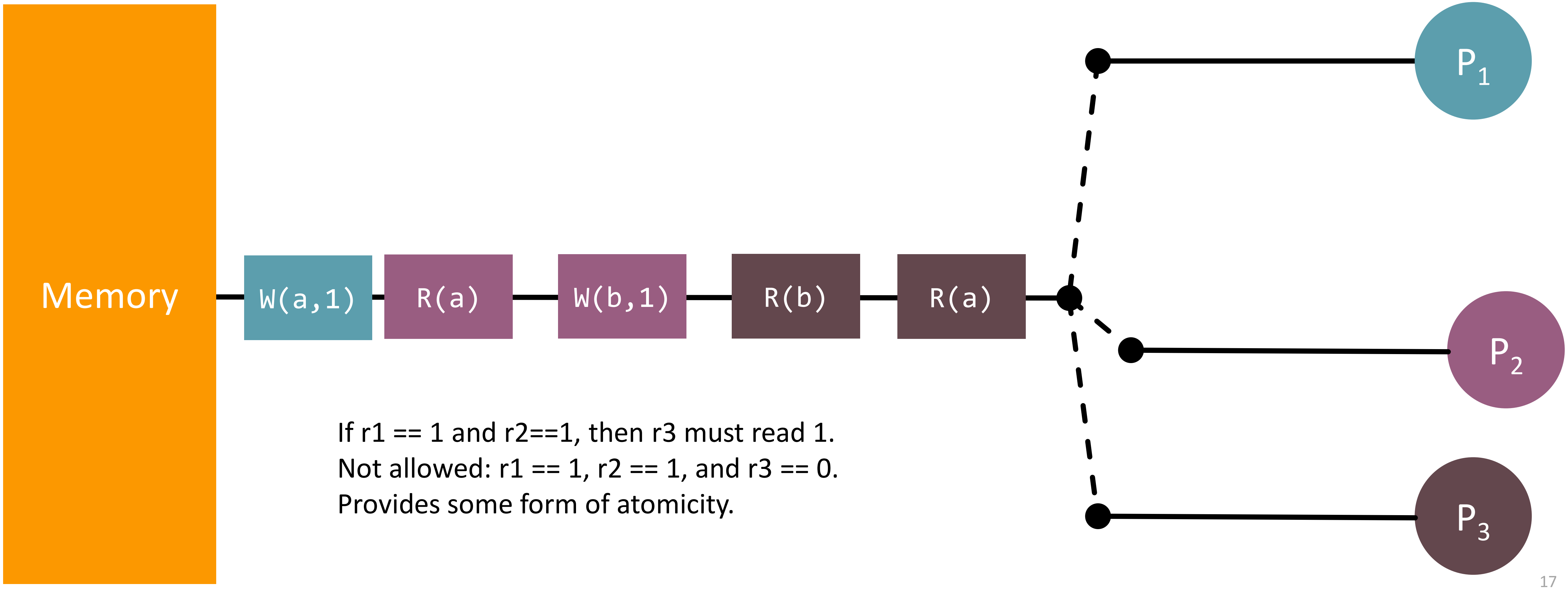
Principle 5

In a multiprocessor system, memory ordering obeys causality (memory ordering respects transitive visibility).

All values zero initially

P1	P2	P3

`a = 1`	`r1 = a`
	`b = 1`
		`r2 = b`
		`r3 = a`



If r1 == 1 and r2==1, then r3 must read 1.
Not allowed: r1 == 1, r2 == 1, and r3 == 0.
Provides some form of atomicity.

Principle 5

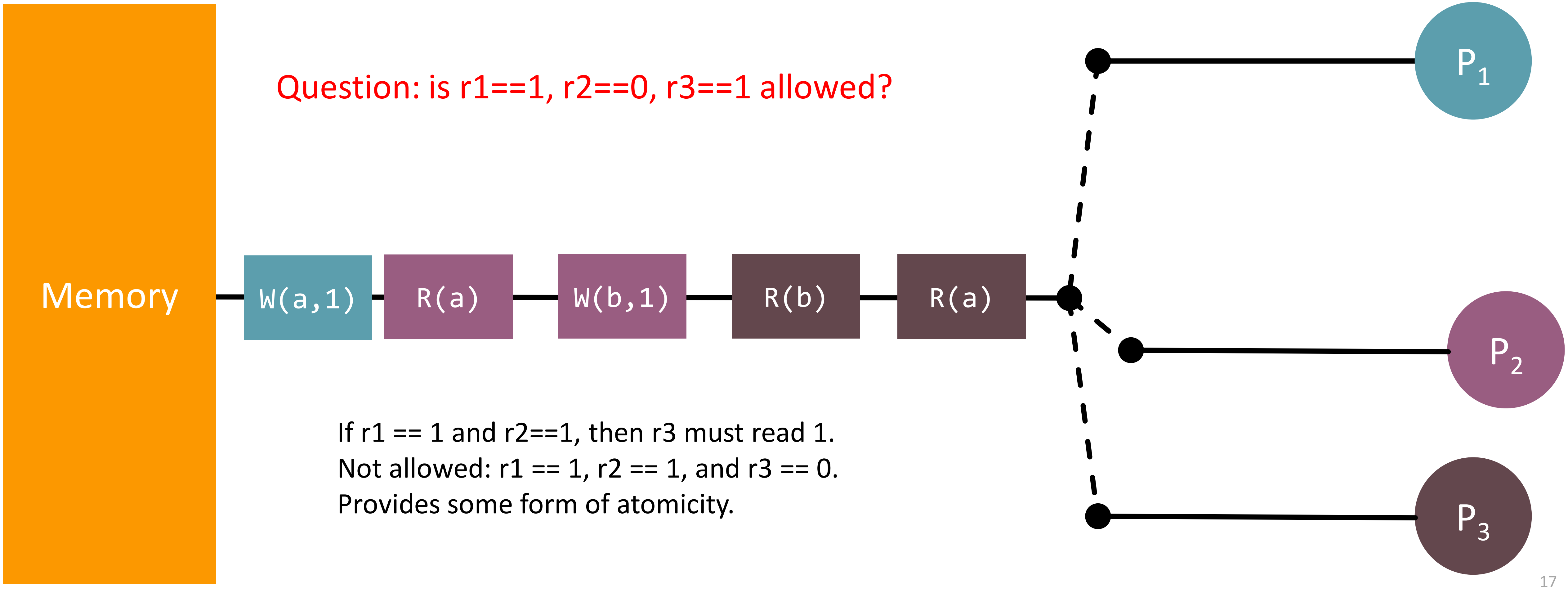
In a multiprocessor system, memory ordering obeys causality (memory ordering respects transitive visibility).

All values zero initially

P1	P2	P3

a = 1	r1 = a
	b = 1
		r2 = b
		r3 = a

Question: is r1==1, r2==0, r3==1 allowed?



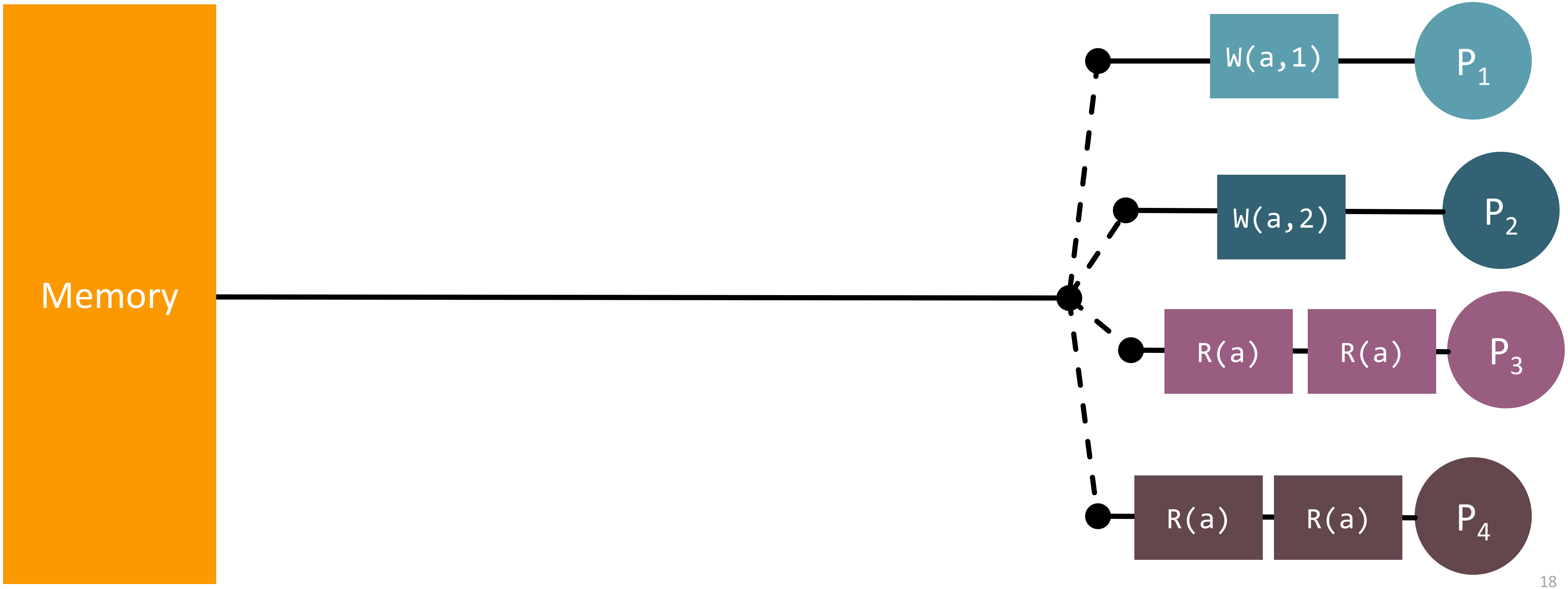
If r1 == 1 and r2==1, then r3 must read 1.
Not allowed: r1 == 1, r2 == 1, and r3 == 0.
Provides some form of atomicity.

Principle 6

In a multiprocessor system, writes to the same location have a total order (implied by cache coherence).

All values zero initially

P1	P2	P3	P4
a=1	a=2		
		r1 = a	r3 = a
		r2 = a	r4 = a



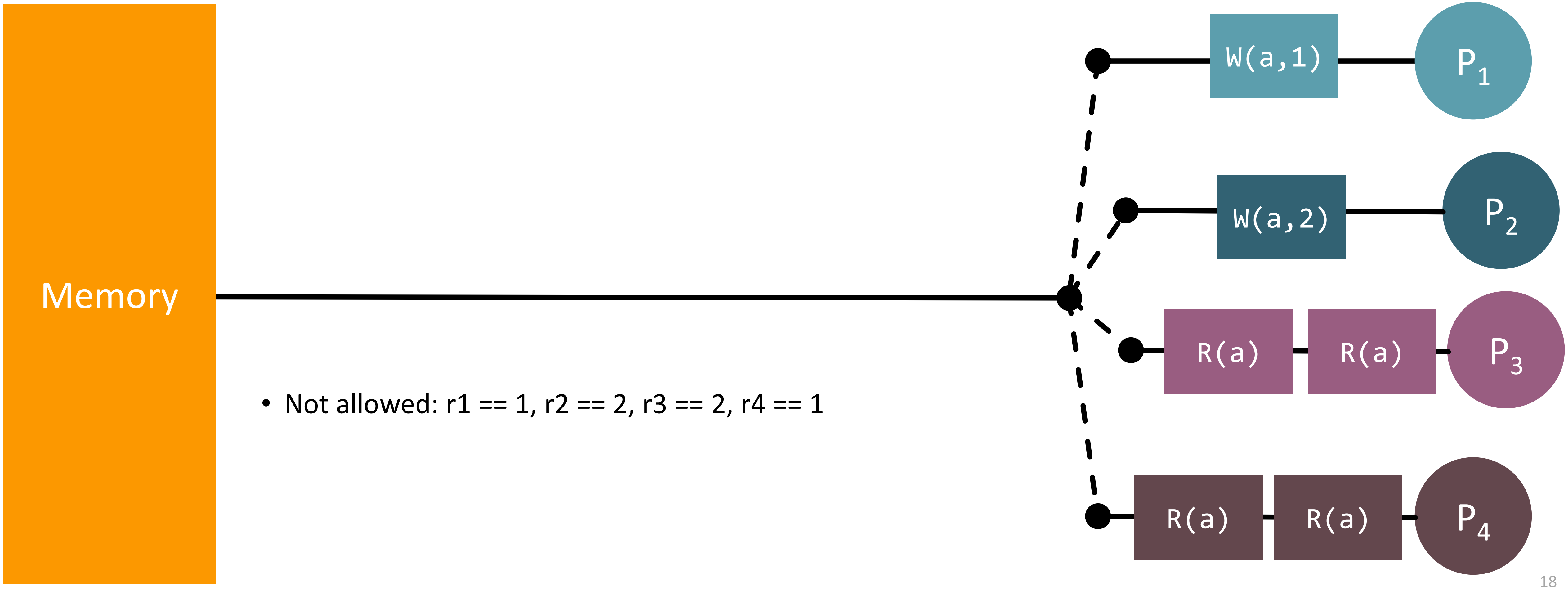
Principle 6

In a multiprocessor system, writes to the same location have a total order (implied by cache coherence).

All values zero initially

P1	P2	P3	P4

a=1	a=2	r1 = a
		r2 = a
			r3 = a
			r4 = a



- Not allowed: r1 == 1, r2 == 2, r3 == 2, r4 == 1

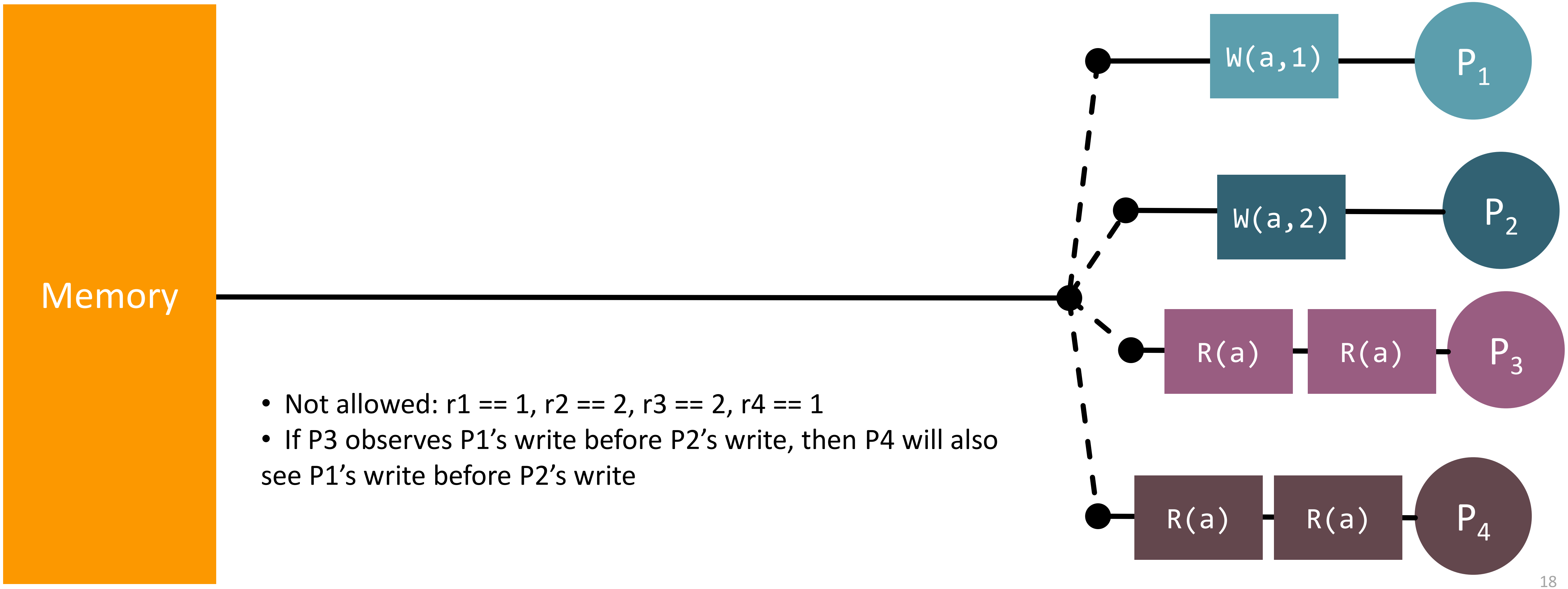
Principle 6

In a multiprocessor system, writes to the same location have a total order (implied by cache coherence).

All values zero initially

P1	P2	P3	P4

`a=1`	`a=2`	`r1 = a`
		`r2 = a`
			`r3 = a`
			`r4 = a`



- Not allowed: r1 == 1, r2 == 2, r3 == 2, r4 == 1
- If P3 observes P1’s write before P2’s write, then P4 will also see P1’s write before P2’s write

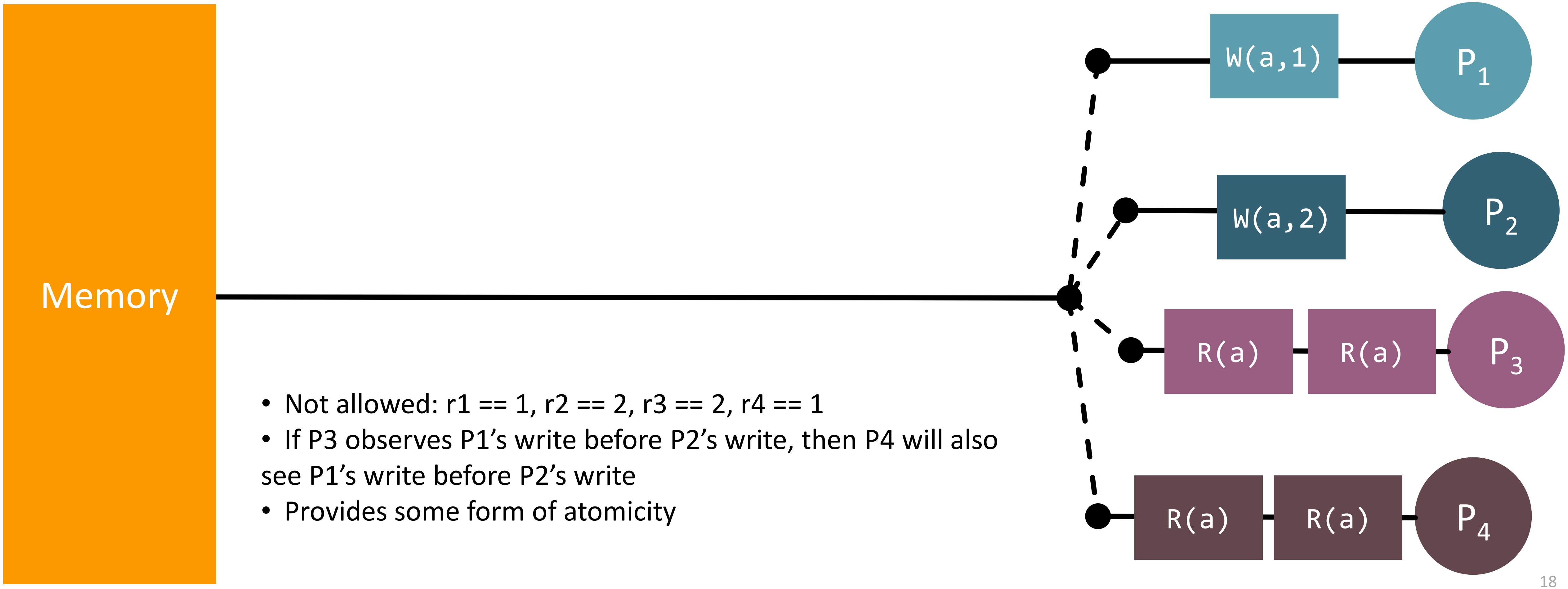
Principle 6

In a multiprocessor system, writes to the same location have a total order (implied by cache coherence).

All values zero initially

P1	P2	P3	P4
a=1	a=2		

		r1 = a
		r2 = a
			r3 = a
			r4 = a



- Not allowed: r1 == 1, r2 == 2, r3 == 2, r4 == 1
- If P3 observes P1’s write before P2’s write, then P4 will also see P1’s write before P2’s write
- Provides some form of atomicity

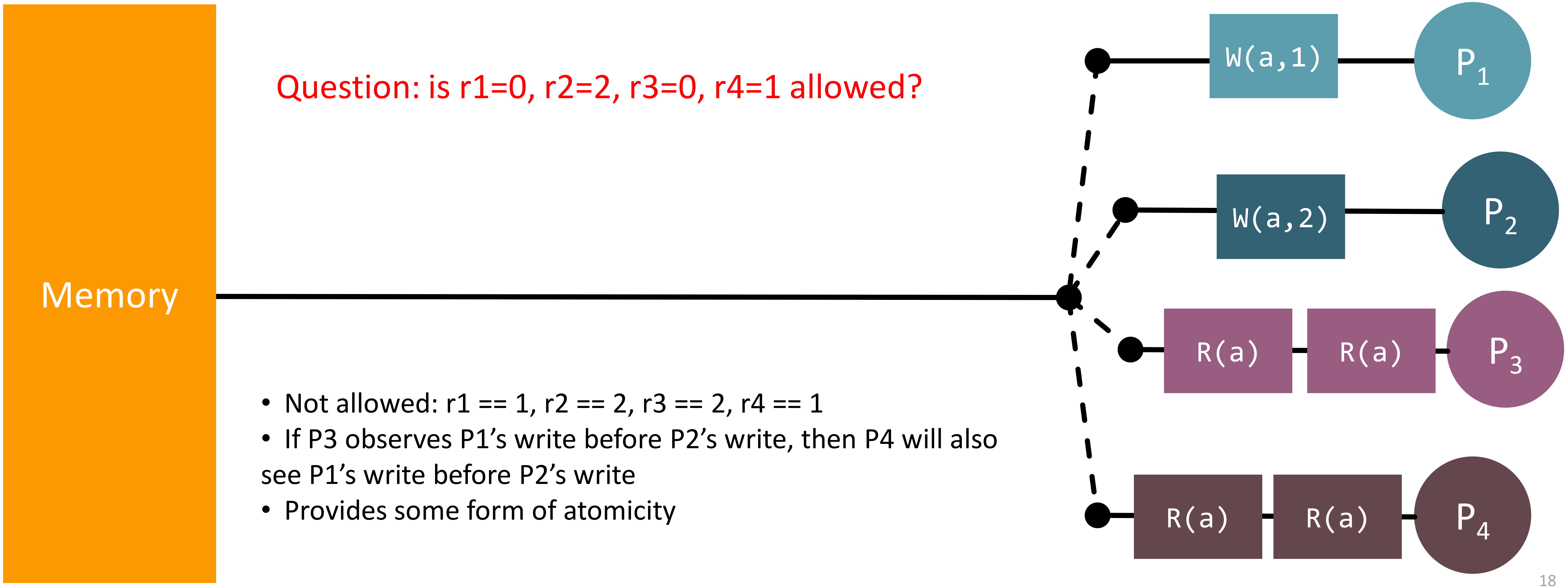
Principle 6

In a multiprocessor system, writes to the same location have a total order (implied by cache coherence).

All values zero initially

P1	P2	P3	P4

a=1	a=2	r1 = a
		r2 = a
			r3 = a
			r4 = a



Question: is r1=0, r2=2, r3=0, r4=1 allowed?

- Not allowed: r1 == 1, r2 == 2, r3 == 2, r4 == 1
- If P3 observes P1's write before P2's write, then P4 will also see P1's write before P2's write
- Provides some form of atomicity

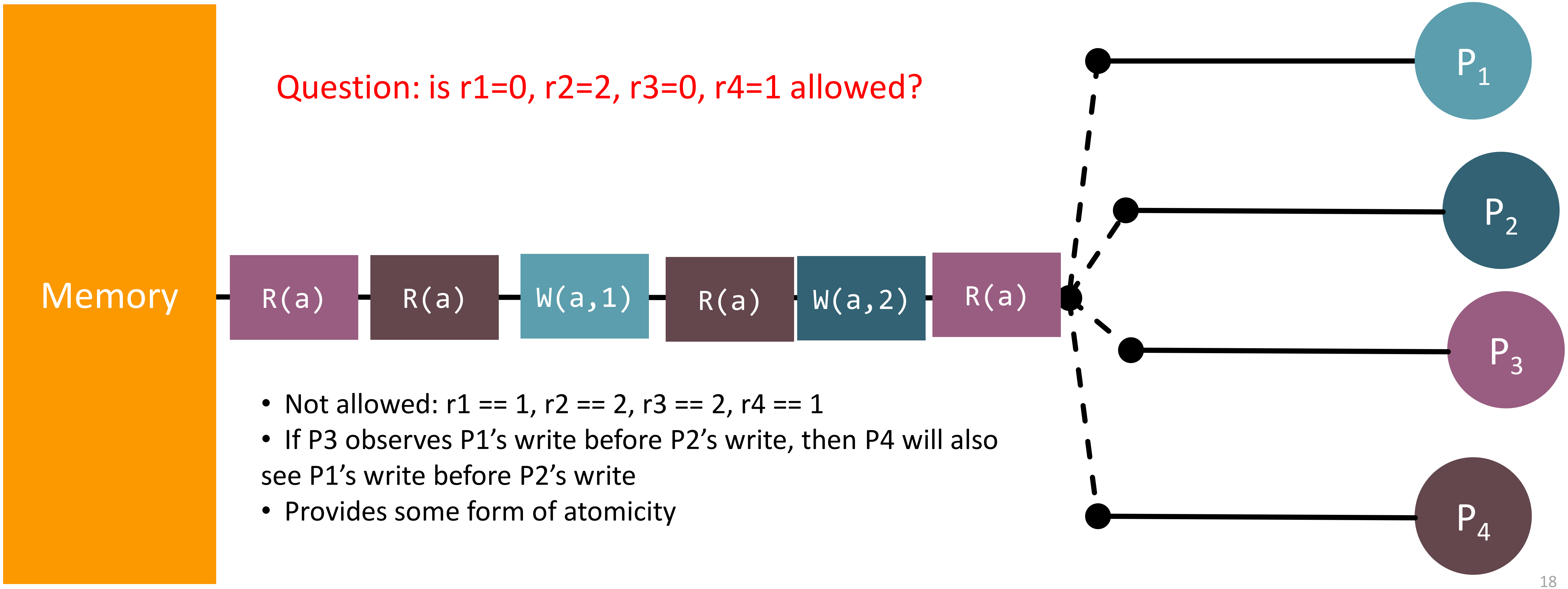
Principle 6

In a multiprocessor system, writes to the same location have a total order (implied by cache coherence).

All values zero initially

P1	P2	P3	P4

a=1	a=2	r1 = a
		r2 = a
			r3 = a
			r4 = a



Question: is r1=0, r2=2, r3=0, r4=1 allowed?

- Not allowed: r1 == 1, r2 == 2, r3 == 2, r4 == 1
- If P3 observes P1’s write before P2’s write, then P4 will also see P1’s write before P2’s write
- Provides some form of atomicity

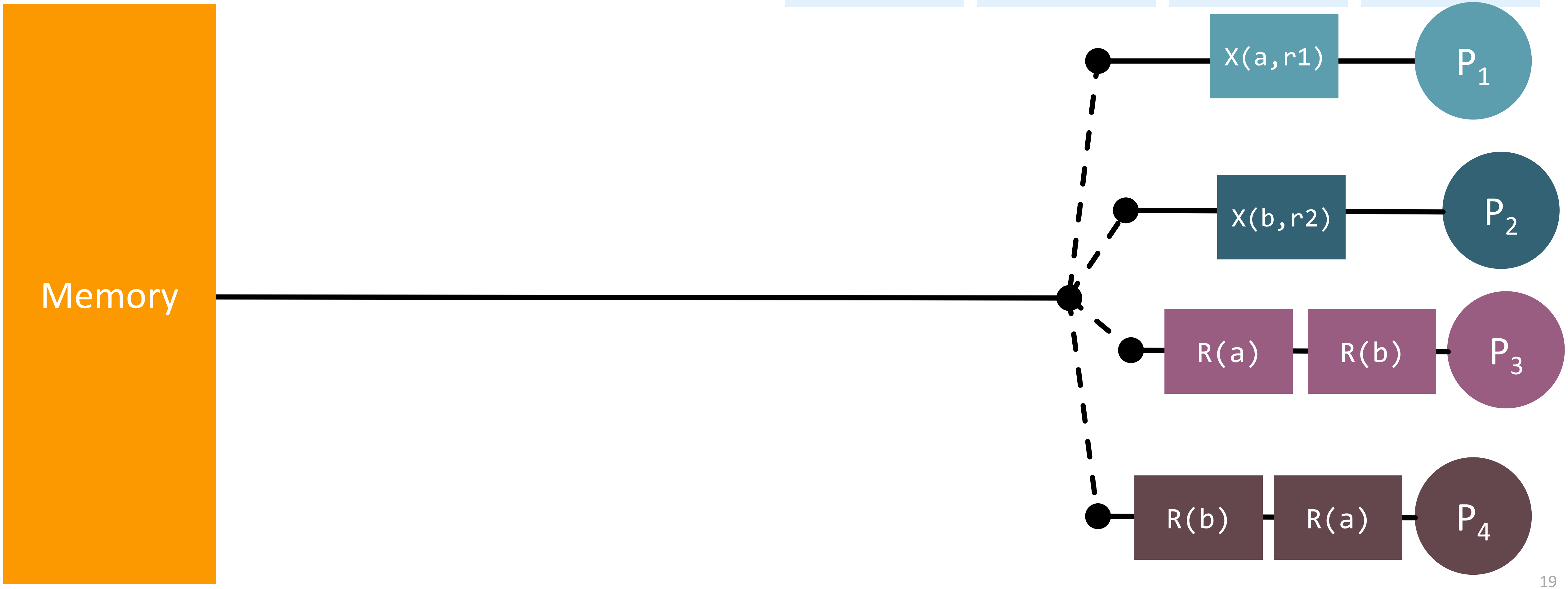
Principle 7

In a multiprocessor system, locked instructions have a total order. (enables synchronized programming!)

All values zero initially, registers r1==r2==1

P1	P2	P3	P4

xchg(a,r1)	xchg(b,r2)	r3 = a
		r4 = b
			r5 = b
			r6 = a

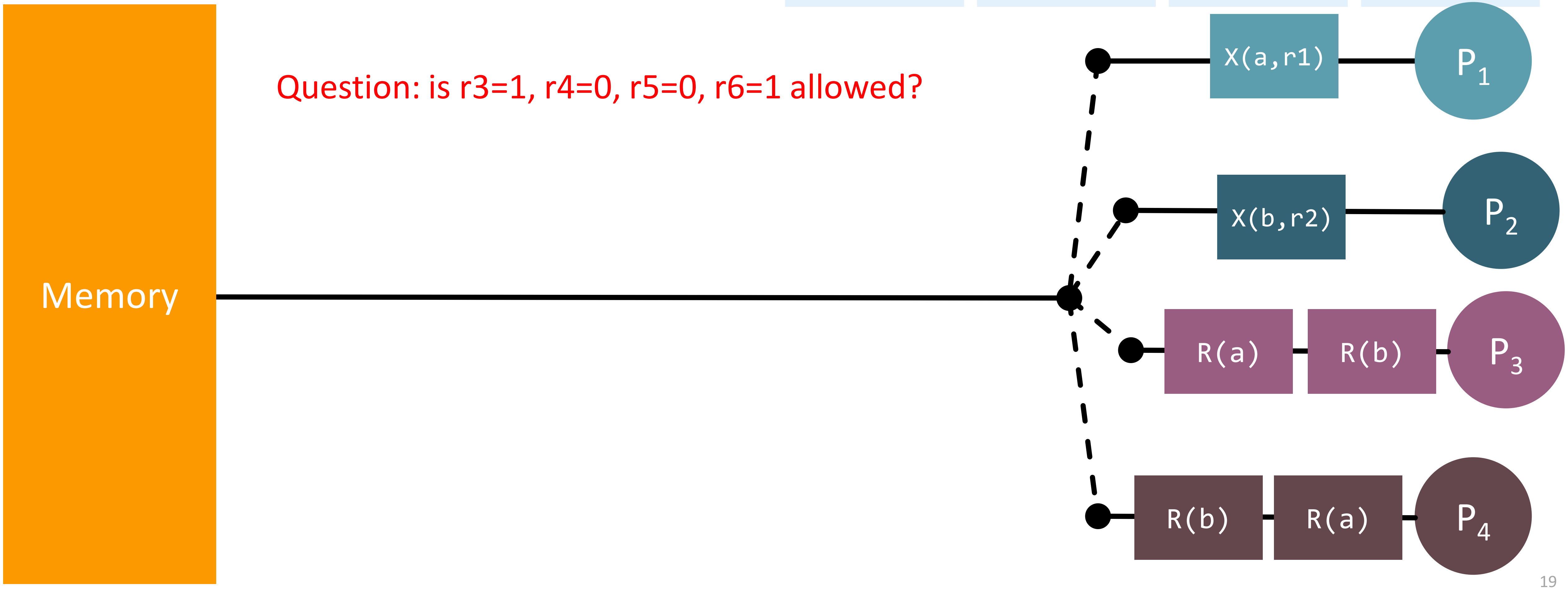


Principle 7

In a multiprocessor system, locked instructions have a total order. (enables synchronized programming!)

All values zero initially, registers r1==r2==1

P1	P2	P3	P4
`xchg(a,r1)`	`xchg(b,r2)`		
		`r3 = a`	`r5 = b`
		`r4 = b`	`r6 = a`



Question: is r3=1, r4=0, r5=0, r6=1 allowed?

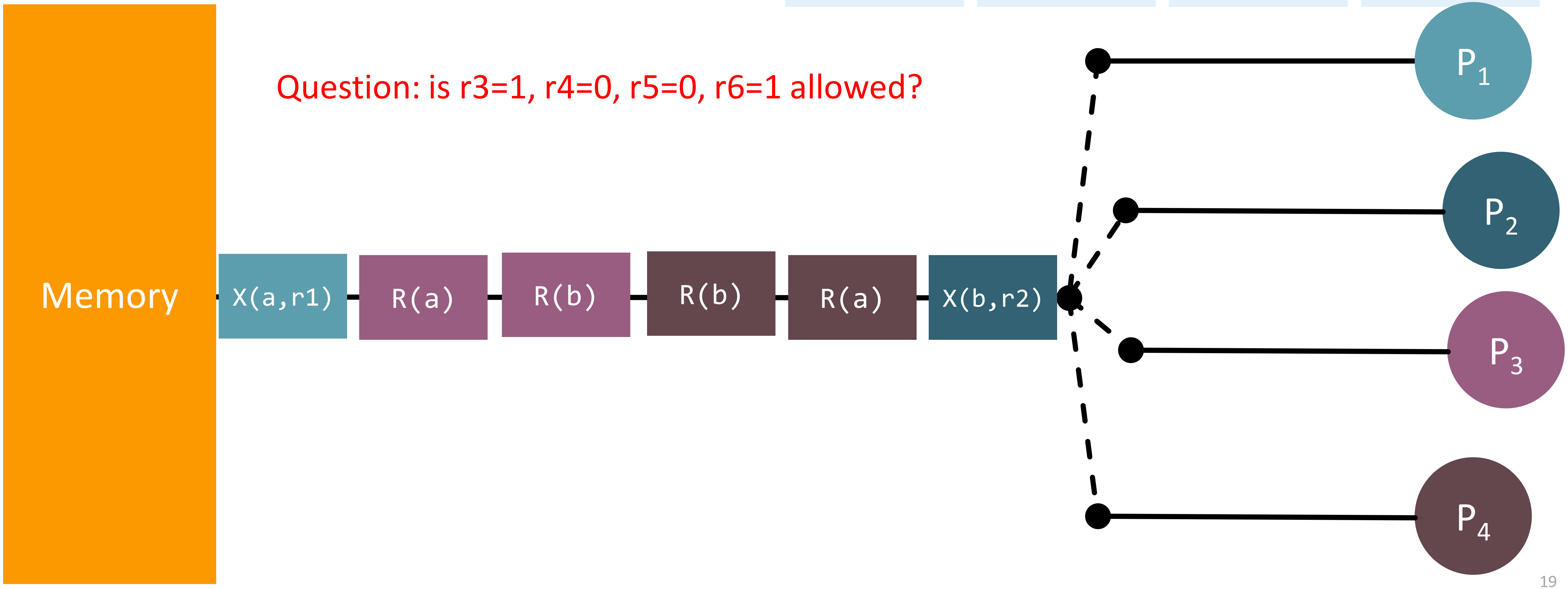
Principle 7

In a multiprocessor system, locked instructions have a total order. (enables synchronized programming!)

All values zero initially, registers r1==r2==1

P1	P2	P3	P4

`xchg(a,r1)`	`xchg(b,r2)`	`r3 = a`
		`r4 = b`
			`r5 = b`
			`r6 = a`



Question: is r3=1, r4=0, r5=0, r6=1 allowed?

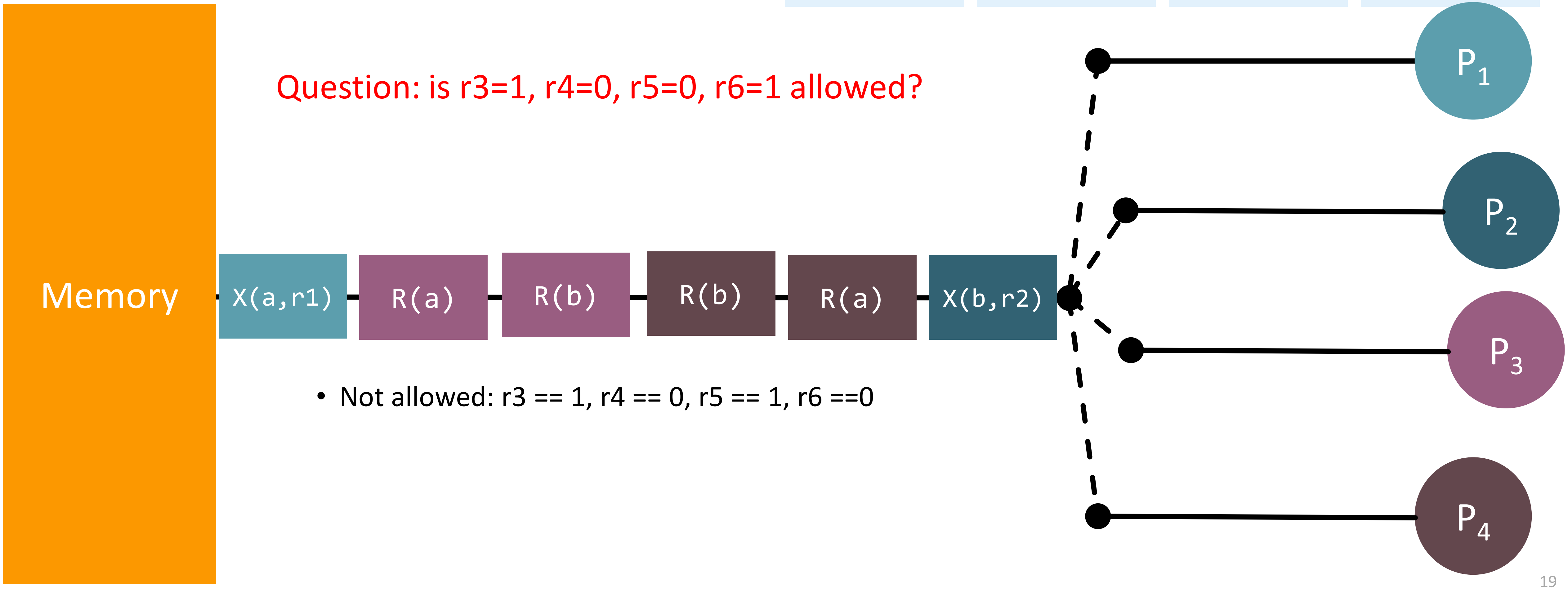
Principle 7

In a multiprocessor system, locked instructions have a total order. (enables synchronized programming!)

All values zero initially, registers r1==r2==1

P1	P2	P3	P4

`xchg(a,r1)`	`xchg(b,r2)`	`r3 = a`
		`r4 = b`
			`r5 = b`
			`r6 = a`



Question: is r3=1, r4=0, r5=0, r6=1 allowed?

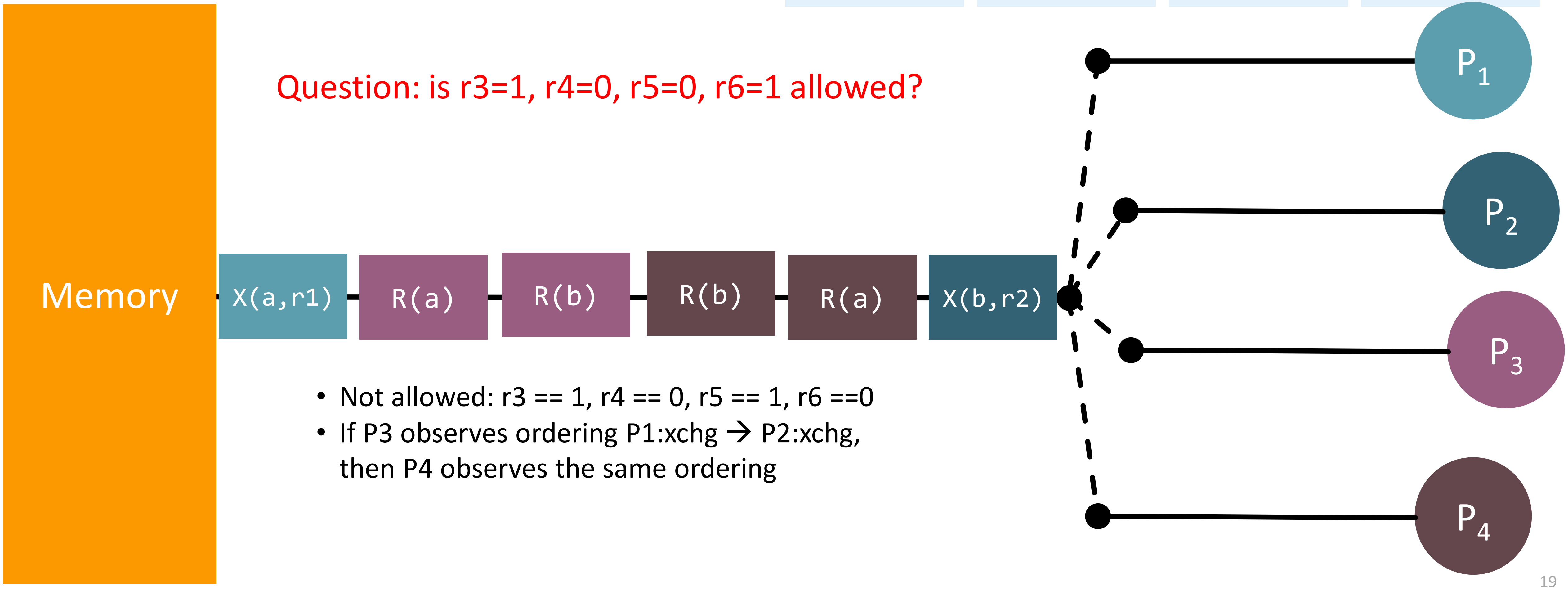
- Not allowed: r3 == 1, r4 == 0, r5 == 1, r6 ==0

Principle 7

In a multiprocessor system, locked instructions have a total order. (enables synchronized programming!)

All values zero initially, registers r1==r2==1

P1	P2	P3	P4
`xchg(a,r1)`	`xchg(b,r2)`		
		`r3 = a`	`r5 = b`
		`r4 = b`	`r6 = a`



Question: is r3=1, r4=0, r5=0, r6=1 allowed?

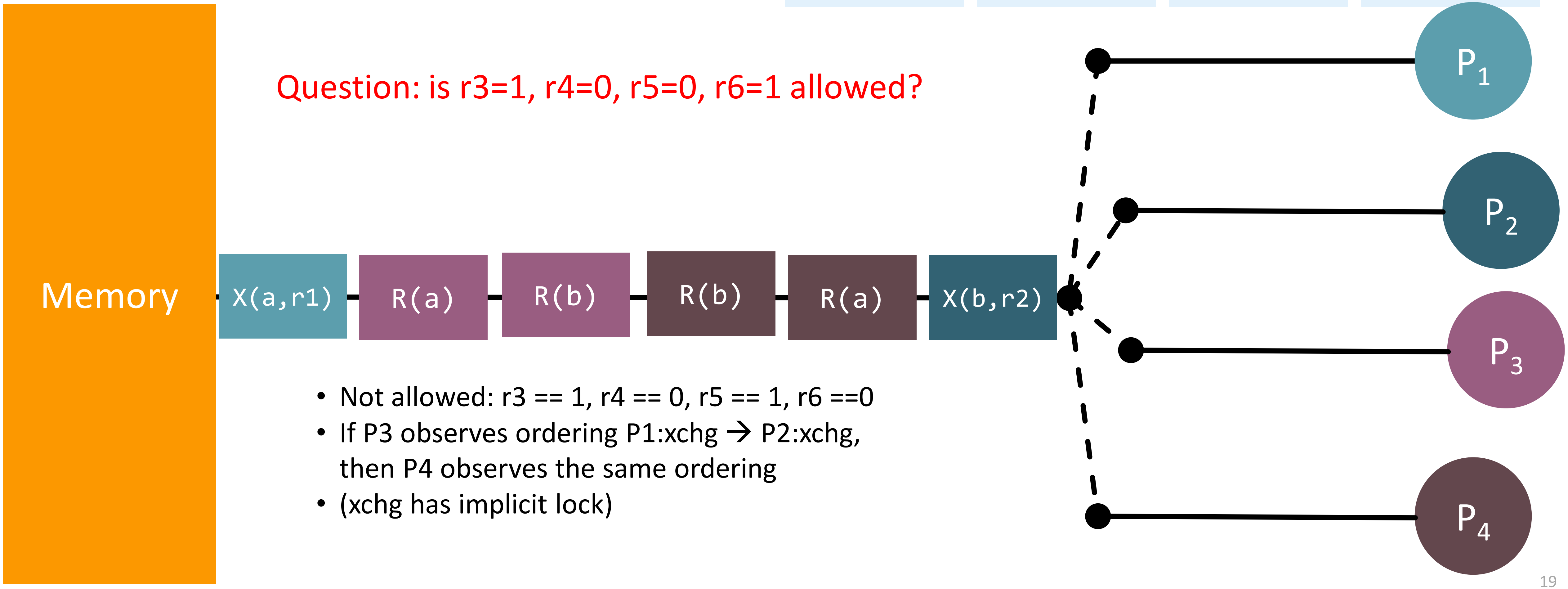
- Not allowed: r3 == 1, r4 == 0, r5 == 1, r6 ==0
- If P3 observes ordering P1:xchg → P2:xchg, then P4 observes the same ordering

Principle 7

In a multiprocessor system, locked instructions have a total order. (enables synchronized programming!)

All values zero initially, registers r1==r2==1

P1	P2	P3	P4
`xchg(a,r1)`	`xchg(b,r2)`		
		`r3 = a`	`r5 = b`
		`r4 = b`	`r6 = a`



Question: is r3=1, r4=0, r5=0, r6=1 allowed?

- Not allowed: r3 == 1, r4 == 0, r5 == 1, r6 ==0
- If P3 observes ordering P1:xchg → P2:xchg, then P4 observes the same ordering
- (xchg has implicit lock)

Principle 8

Reads and writes are not reordered with locked instructions.

(enables synchronized programming!)

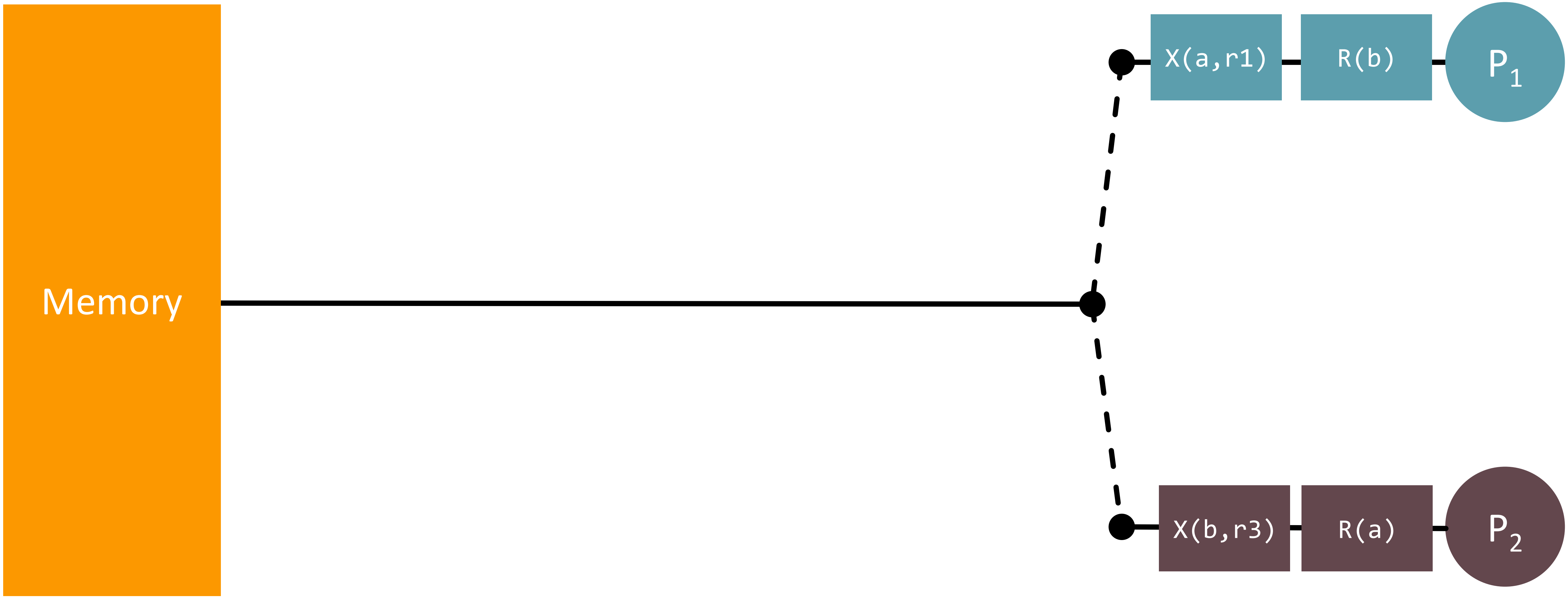
All values zero initially but r1 = r3 = 1

P1

```
xchg(a,r1)
r2 = b
```

P2

```
xchg(b,r3)
r4 = a
```



Principle 8

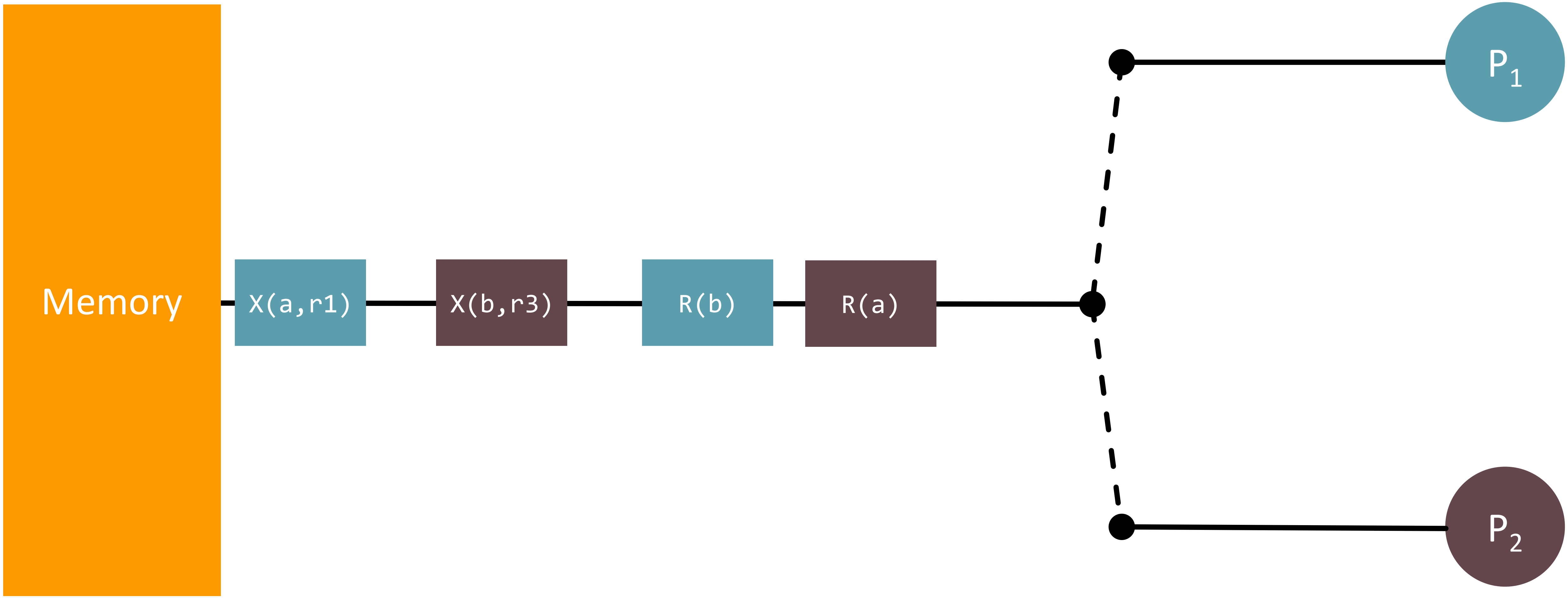
Reads and writes are not reordered with locked instructions.

(enables synchronized programming!)

All values zero initially but r1 = r3 = 1

P1	P2

`xchg(a,r1)`
`r2 = b`
	`xchg(b,r3)`
	`r4 = a`



Principle 8

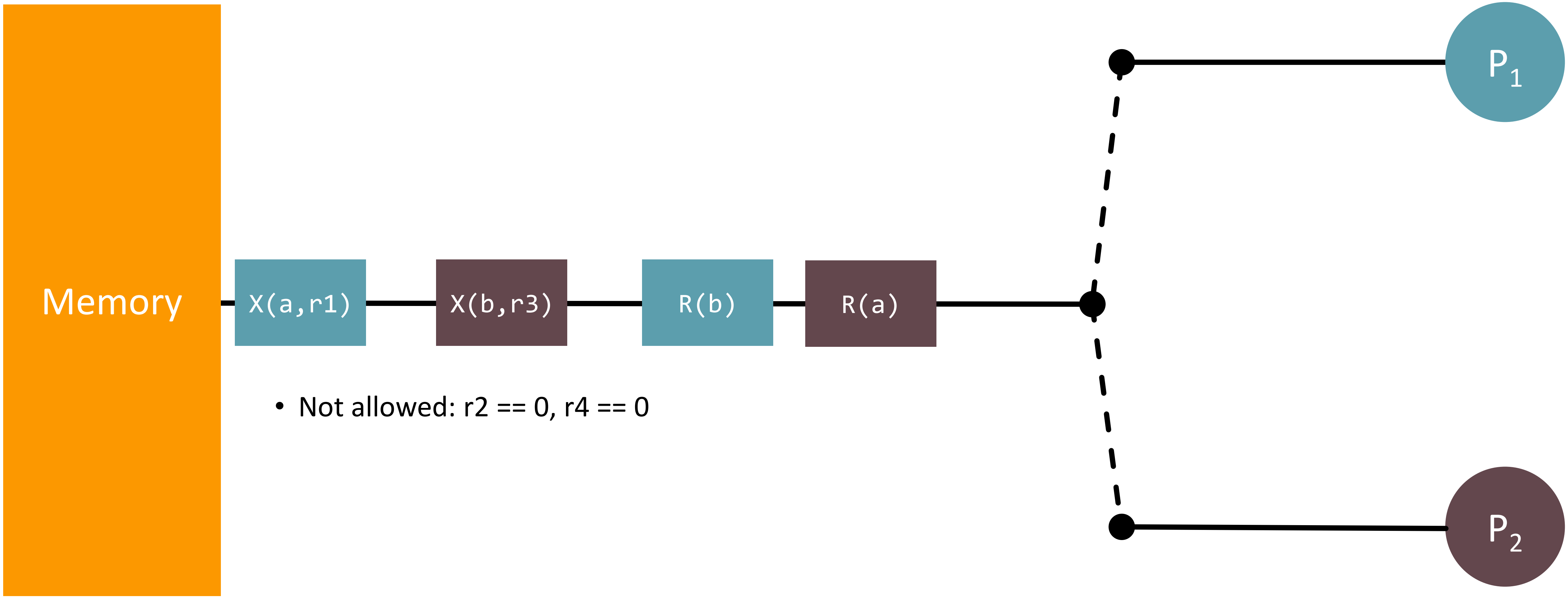
Reads and writes are not reordered with locked instructions.

(enables synchronized programming!)

All values zero initially but r1 = r3 = 1

```
P1

xchg(a,r1)
r2 = b
```

```
P2

xchg(b,r3)
r4 = a
```



- Not allowed: r2 == 0, r4 == 0

Principle 8

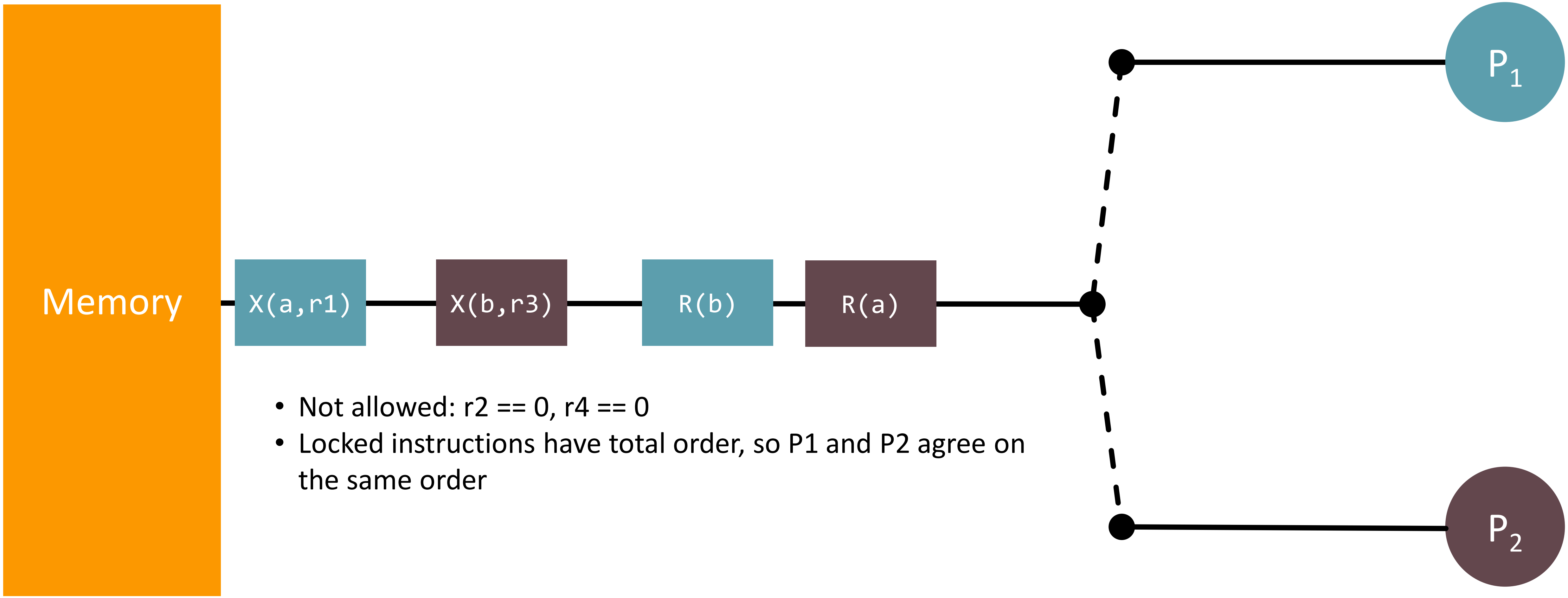
Reads and writes are not reordered with locked instructions.

(enables synchronized programming!)

All values zero initially but r1 = r3 = 1

P1	P2

`xchg(a,r1)`
`r2 = b`
	`xchg(b,r3)`
	`r4 = a`



- Not allowed: r2 == 0, r4 == 0
- Locked instructions have total order, so P1 and P2 agree on the same order

Principle 8

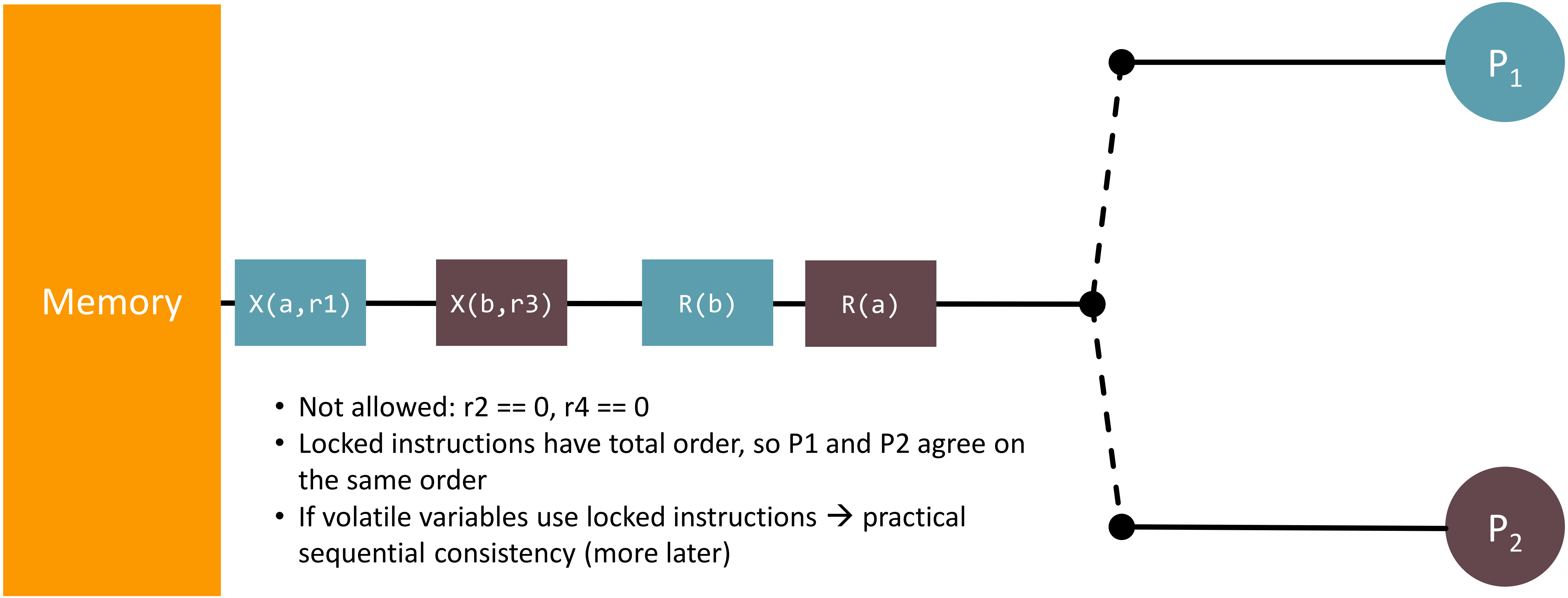
Reads and writes are not reordered with locked instructions.

(enables synchronized programming!)

All values zero initially but r1 = r3 = 1

P1	P2

`xchg(a,r1)`
`r2 = b`
	`xchg(b,r3)`
	`r4 = a`



- Not allowed: r2 == 0, r4 == 0
- Locked instructions have total order, so P1 and P2 agree on the same order
- If volatile variables use locked instructions → practical sequential consistency (more later)

An Alternative View: x86-TSO

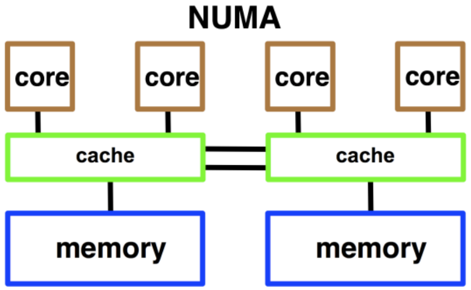
- **Sewell el al.: “x86-TSO: A Rigorous and Usable Programmer’s Model for x86 Multiprocessors”, CACM May 2010**

“[…] **real multiprocessors typically do not provide the sequentially consistent memory** that is assumed by most work on semantics and verification. Instead, they have relaxed memory models, varying in subtle ways between processor families, in which different hardware threads may have only loosely consistent views of a shared memory. Second, **the public vendor architectures, supposedly specifying what programmers can rely on, are often in ambiguous informal prose (a particularly poor medium for loose specifications), leading to widespread confusion**. [...] We present a new x86-TSO programmer’s model that, to the best of our knowledge, suffers from none of these problems. **It is mathematically precise** (rigorously defined in HOL4) but can be presented as an **intuitive abstract machine which should be widely accessible to working programmers**. […]”

An Alternative View: x86-TSO

- **Sewell el al.: “x86-TSO: A Rigorous and Usable Programmer’s Model for x86 Multiprocessors”, CACM May 2010**

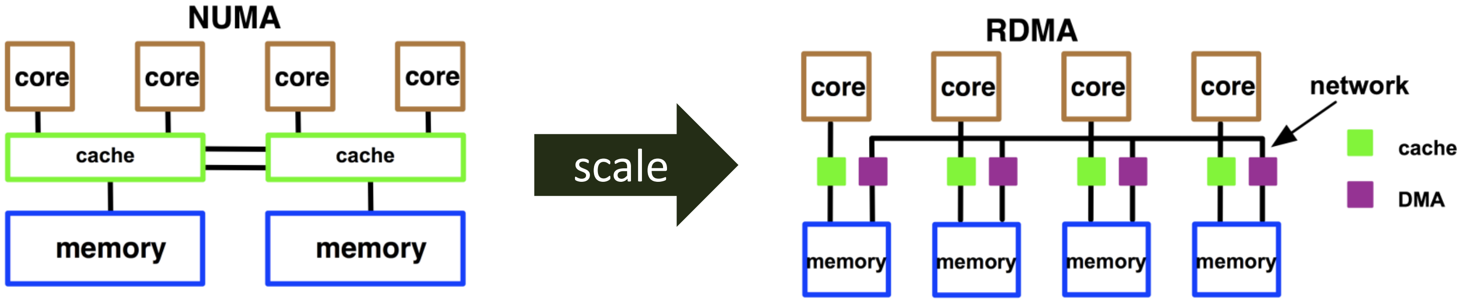
“[…] **real multiprocessors typically do not provide the sequentially consistent memory** that is assumed by most work on semantics and verification. Instead, they have relaxed memory models, varying in subtle ways between processor families, in which different hardware threads may have only loosely consistent views of a shared memory. Second, **the public vendor architectures, supposedly specifying what programmers can rely on, are often in ambiguous informal prose (a particularly poor medium for loose specifications), leading to widespread confusion**. [...] We present a new x86-TSO programmer’s model that, to the best of our knowledge, suffers from none of these problems. **It is mathematically precise** (rigorously defined in HOL4) but can be presented as an **intuitive abstract machine which should be widely accessible to working programmers**. […]”



An Alternative View: x86-TSO

- **Sewell el al.: “x86-TSO: A Rigorous and Usable Programmer’s Model for x86 Multiprocessors”, CACM May 2010**

“[…] **real multiprocessors typically do not provide the sequentially consistent memory** that is assumed by most work on semantics and verification. Instead, they have relaxed memory models, varying in subtle ways between processor families, in which different hardware threads may have only loosely consistent views of a shared memory. Second, **the public vendor architectures, supposedly specifying what programmers can rely on, are often in ambiguous informal prose (a particularly poor medium for loose specifications), leading to widespread confusion**. […] We present a new x86-TSO programmer’s model that, to the best of our knowledge, suffers from none of these problems. **It is mathematically precise** (rigorously defined in HOL4) but can be presented as an **intuitive abstract machine which should be widely accessible to working programmers**. […]”



Similar for RMA systems: A. Dan, P. Lam, T. Hoefler, A. Vechev: Modeling and Analysis of Remote Memory Access Programming, ACM OOPSLA’16, best paper

Notions of Correctness

- **We discussed so far:**
  - Read/write of the same location

    Cache coherence (write serialization and atomicity)
  - Read/write of multiple locations

    Memory models (visibility order of updates by cores)

- **Now one level up: objects (variables/fields with invariants defined on them)**
  - Invariants “tie” variables together
  - Sequential objects
  - Concurrent objects

Sequential Objects

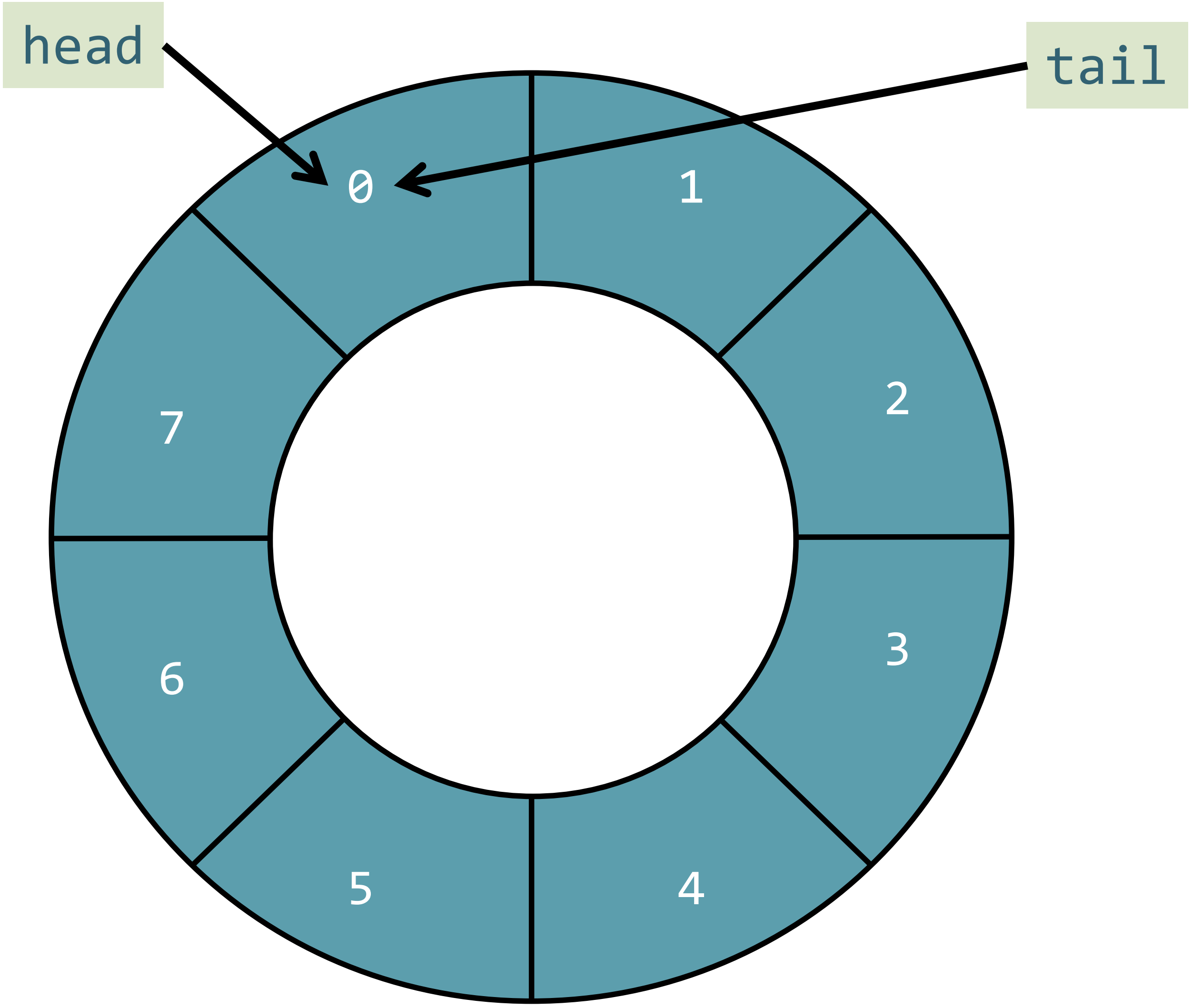
- **Each object has a type**
- **A type is defined by a class**
  - Set of fields forms the state of an object
  - Set of methods (or free functions) to manipulate the state

- **Remark**
  - An Interface is an abstract type that defines behavior

    A class implementing an interface defines several types

Running Example: FIFO Queue

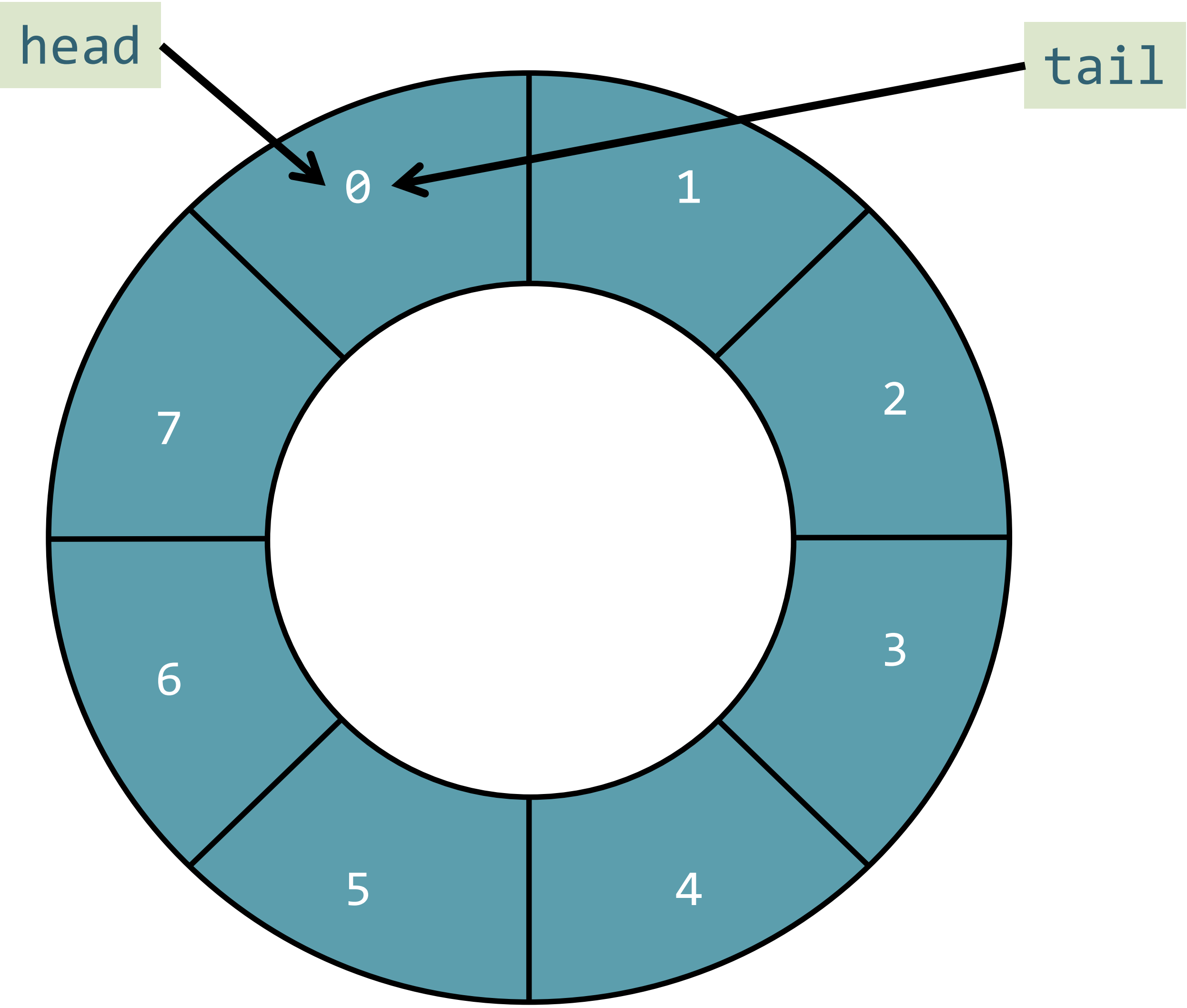
- **Insert elements at tail**
- **Remove elements from head**
    - Initial: `head = tail = 0`



`capacity = 8`

Running Example: FIFO Queue

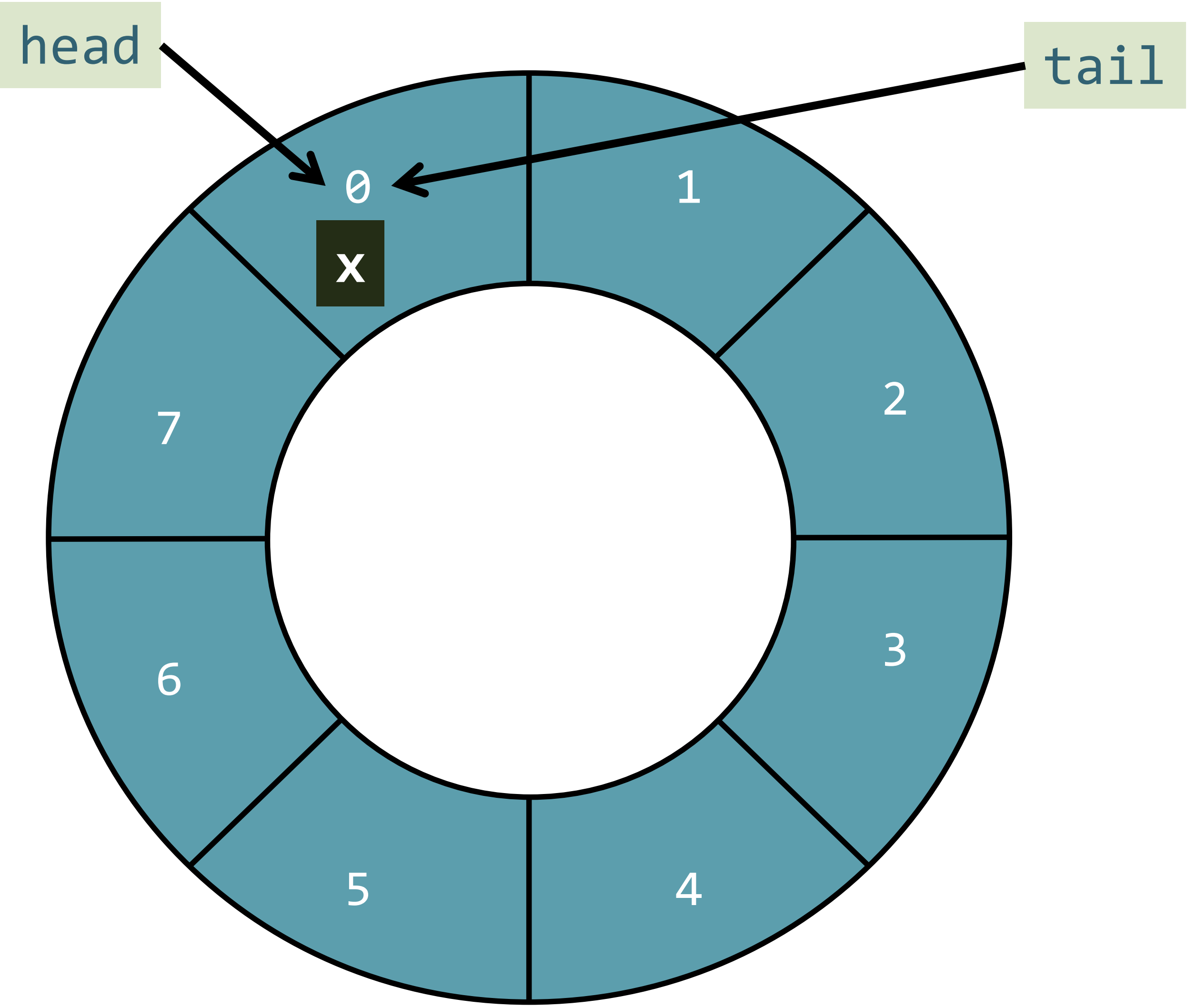
- **Insert elements at tail**
- **Remove elements from head**
  - Initial: `head = tail = 0`
  - `enq(x)`



`capacity = 8`

Running Example: FIFO Queue

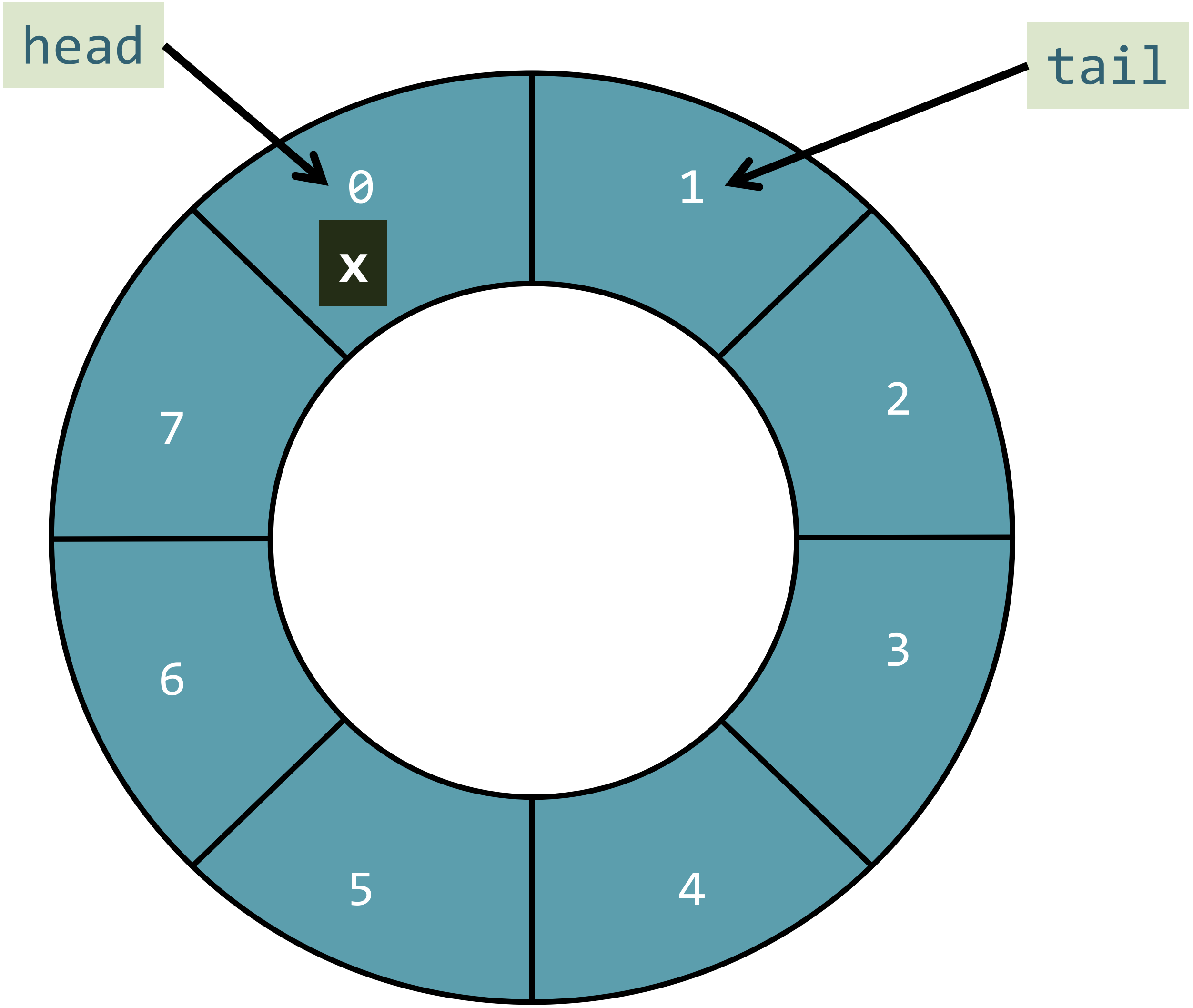
- **Insert elements at tail**
- **Remove elements from head**
    - Initial: `head = tail = 0`
    - `enq(x)`



capacity = 8

Running Example: FIFO Queue

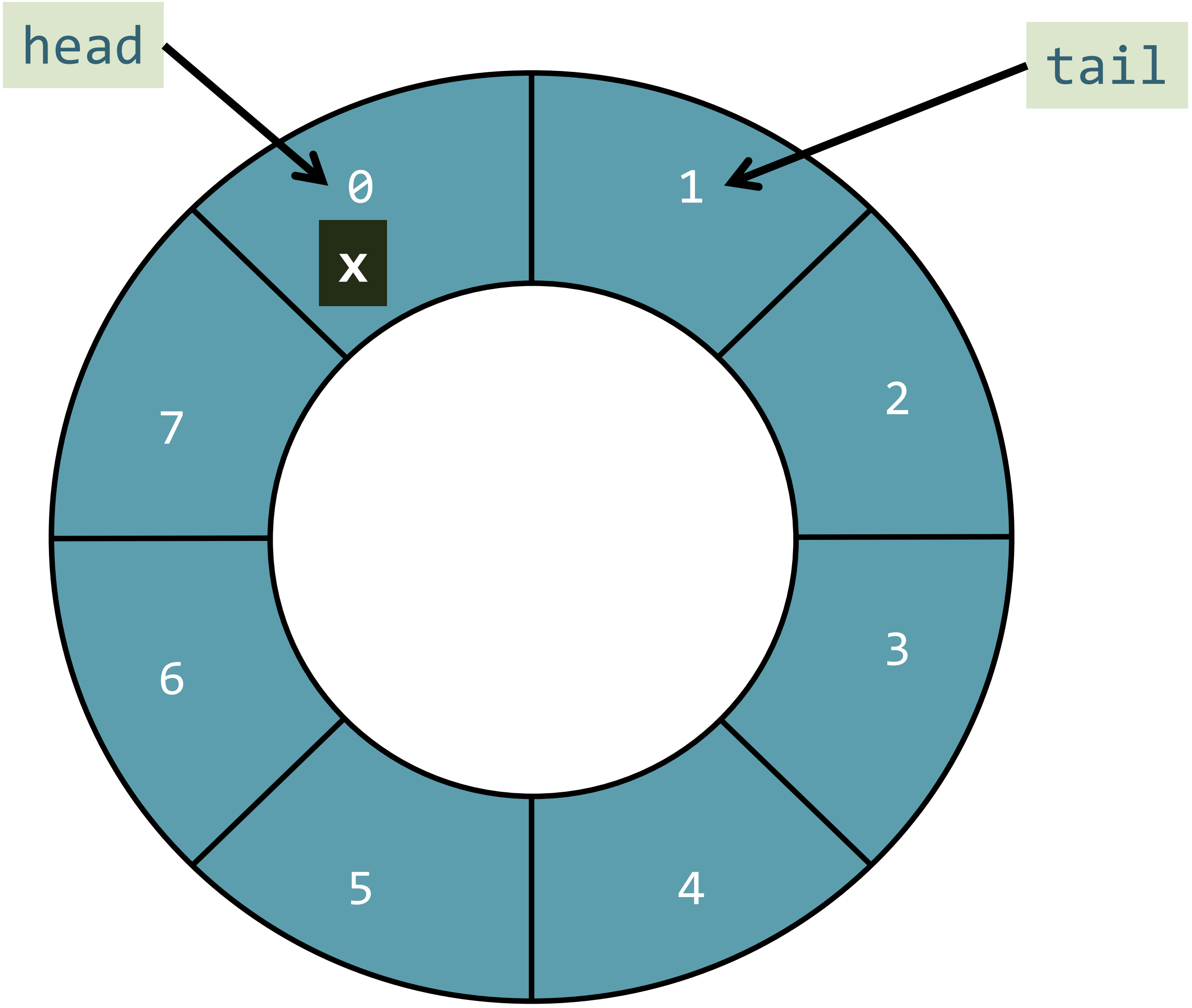
- **Insert elements at tail**
- **Remove elements from head**
  - Initial: `head = tail = 0`
  - `enq(x)`



`capacity = 8`

Running Example: FIFO Queue

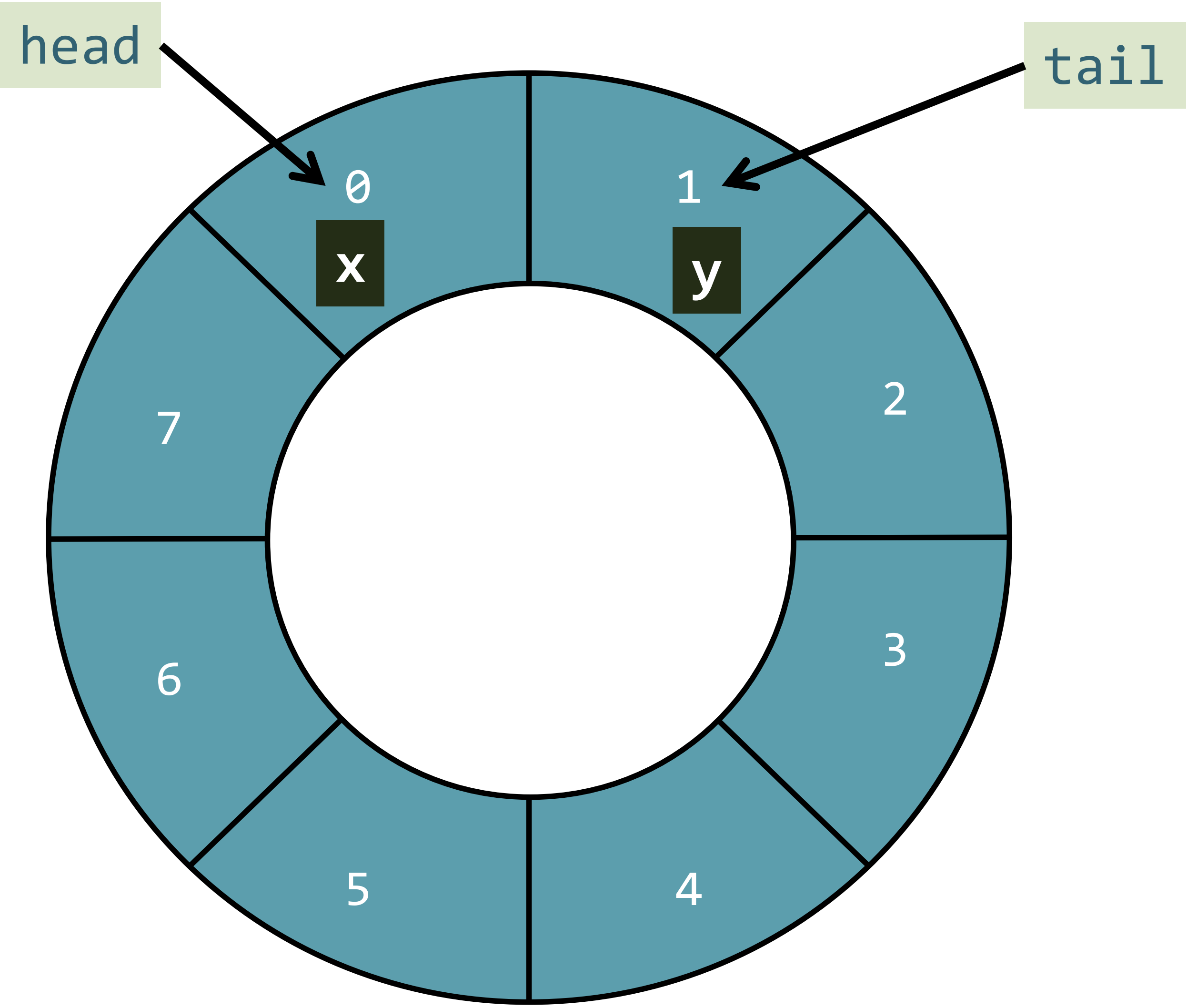
- **Insert elements at tail**
- **Remove elements from head**
  - Initial: `head = tail = 0`
  - `enq(x)`
  - `enq(y)`



`capacity = 8`

Running Example: FIFO Queue

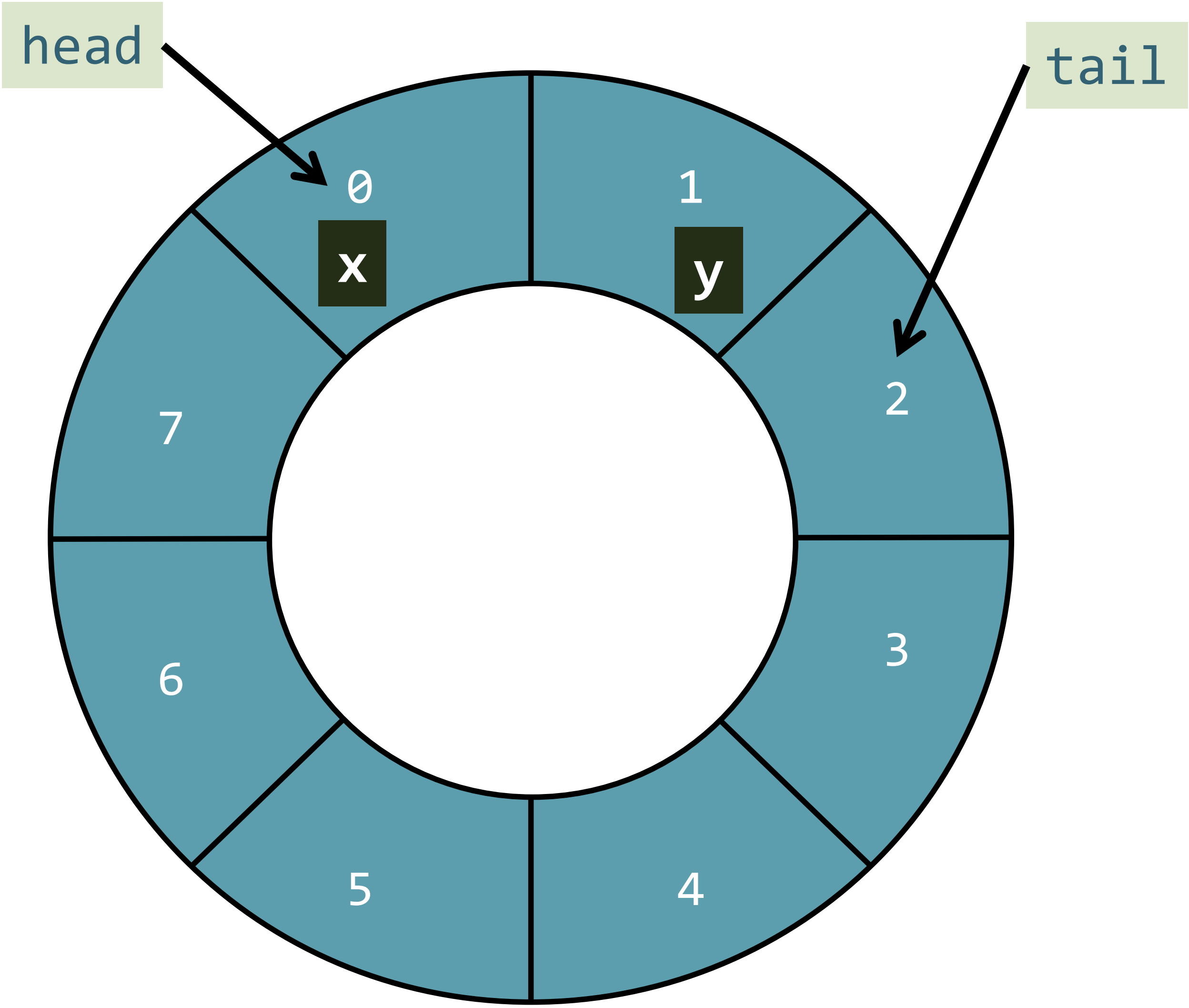
- **Insert elements at tail**
- **Remove elements from head**
  - Initial: `head = tail = 0`
  - `enq(x)`
  - `enq(y)`



`capacity = 8`

Running Example: FIFO Queue

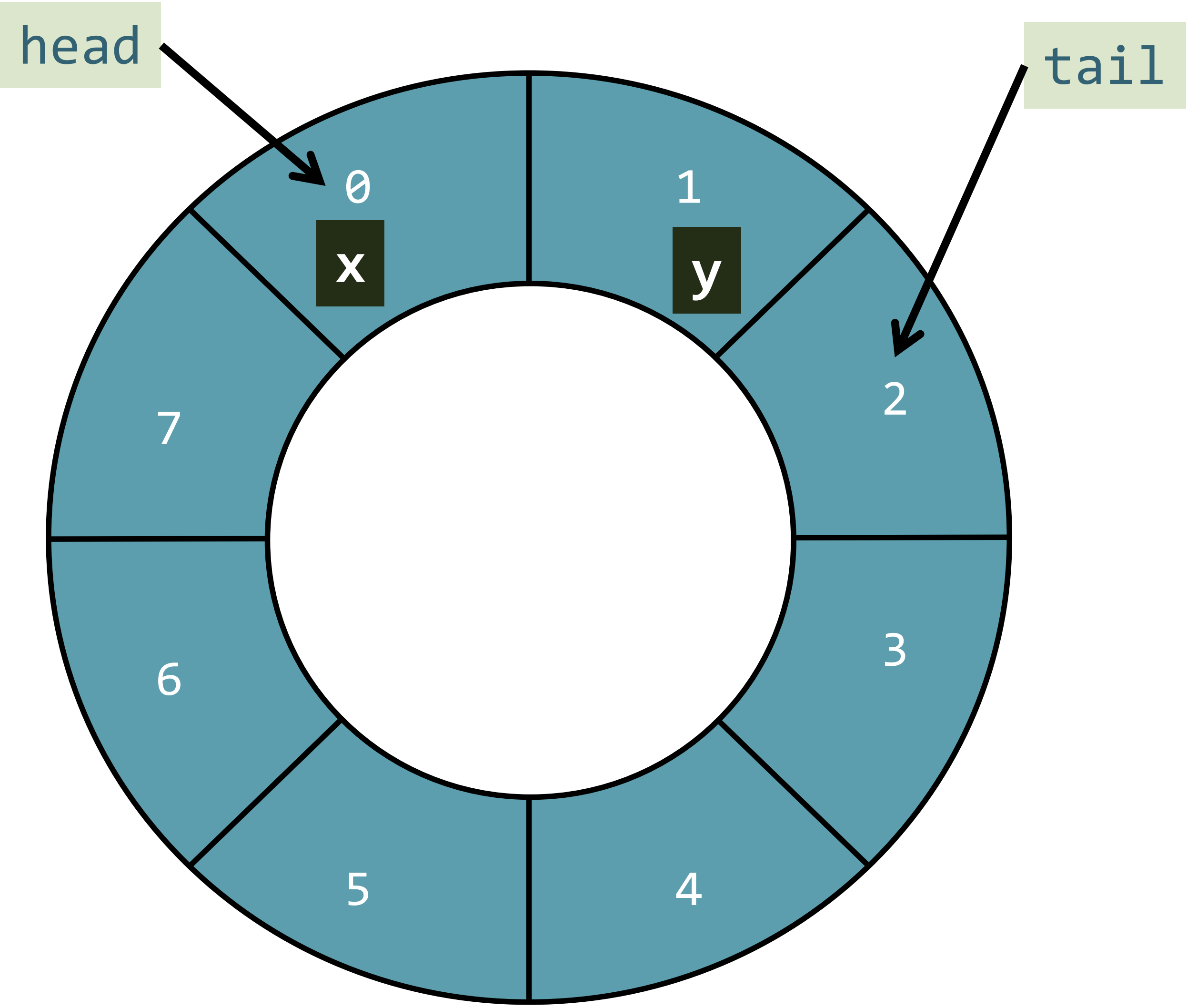
- **Insert elements at tail**
- **Remove elements from head**
    - Initial: `head = tail = 0`
    - `enq(x)`
    - `enq(y)`



capacity = 8

Running Example: FIFO Queue

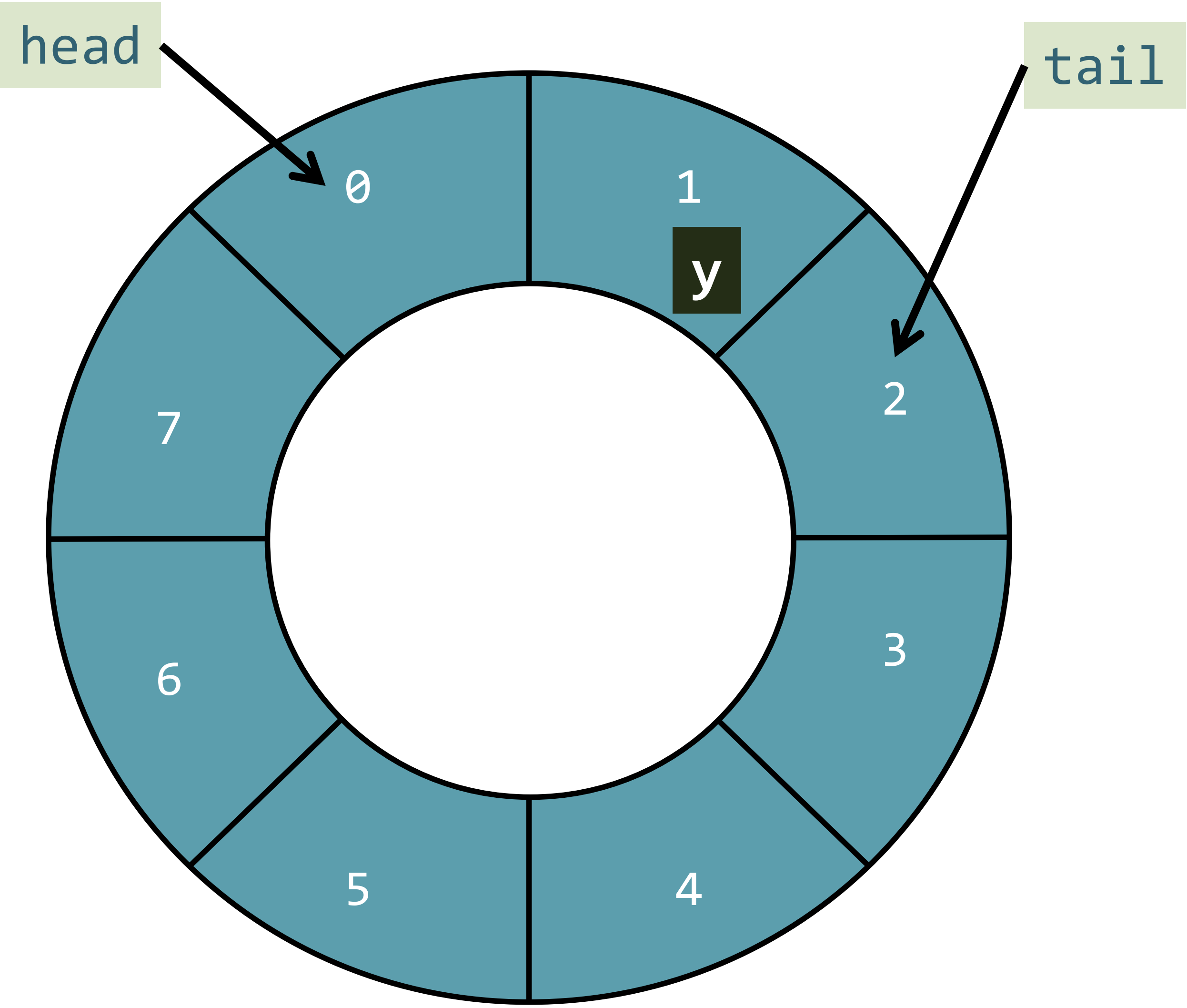
- **Insert elements at tail**
- **Remove elements from head**
  - Initial: `head = tail = 0`
  - `enq(x)`
  - `enq(y)`
  - `deq() [x]`



`capacity = 8`

Running Example: FIFO Queue

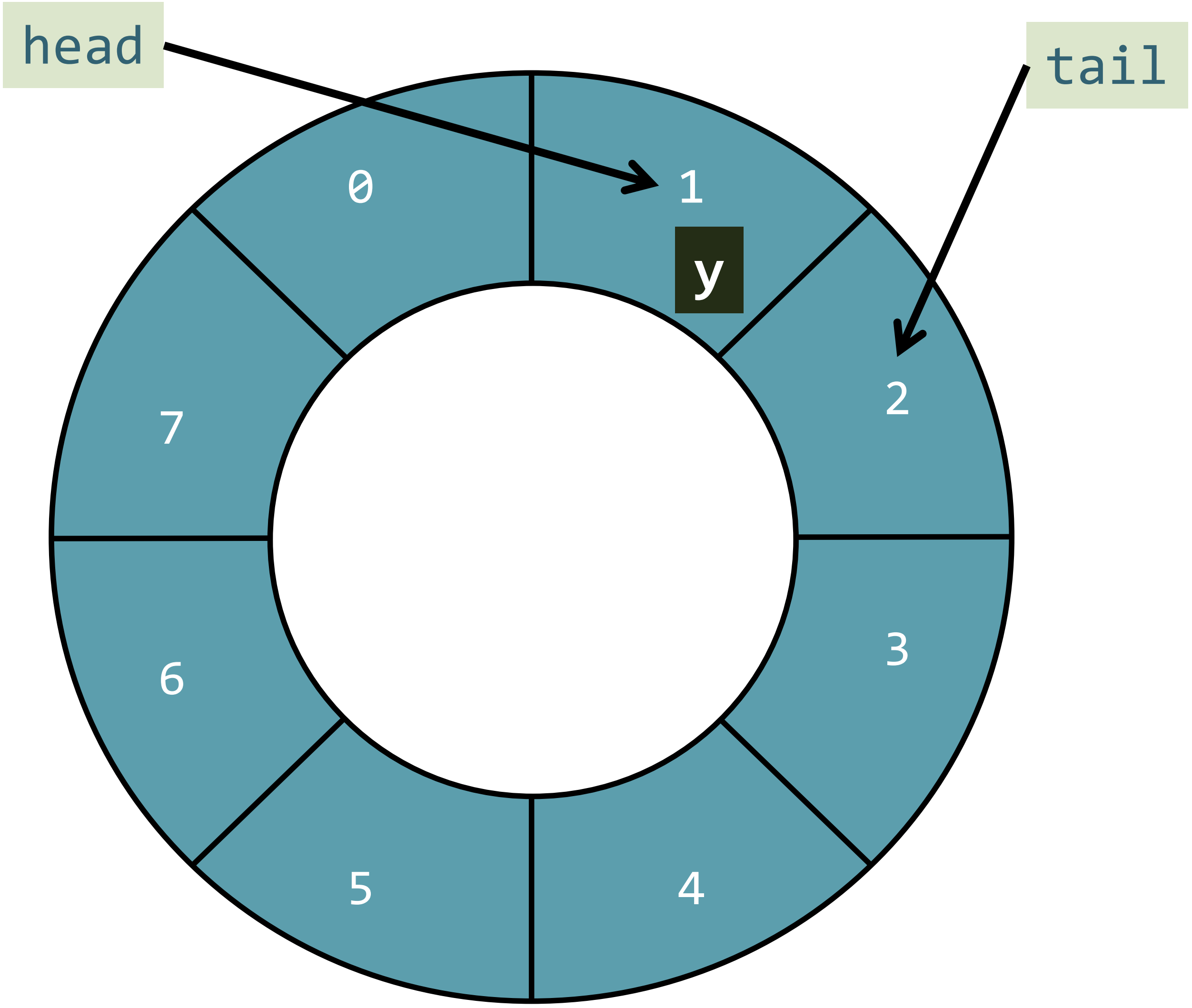
- **Insert elements at tail**
- **Remove elements from head**
  - Initial: `head = tail = 0`
  - `enq(x)`
  - `enq(y)`
  - `deq() [x]`



`capacity = 8`

Running Example: FIFO Queue

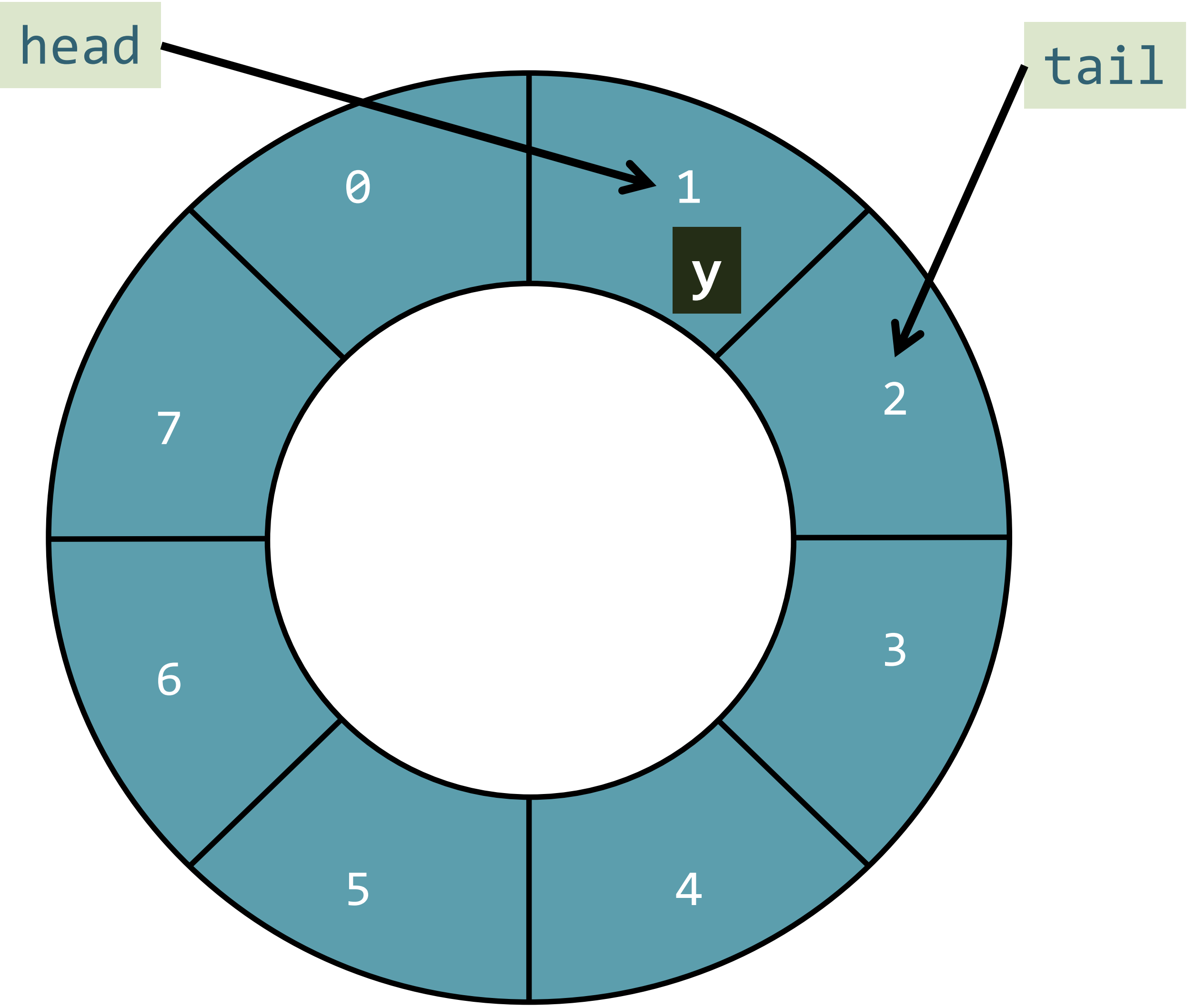
- **Insert elements at tail**
- **Remove elements from head**
  - Initial: `head = tail = 0`
  - `enq(x)`
  - `enq(y)`
  - `deq() [x]`



capacity = 8

Running Example: FIFO Queue

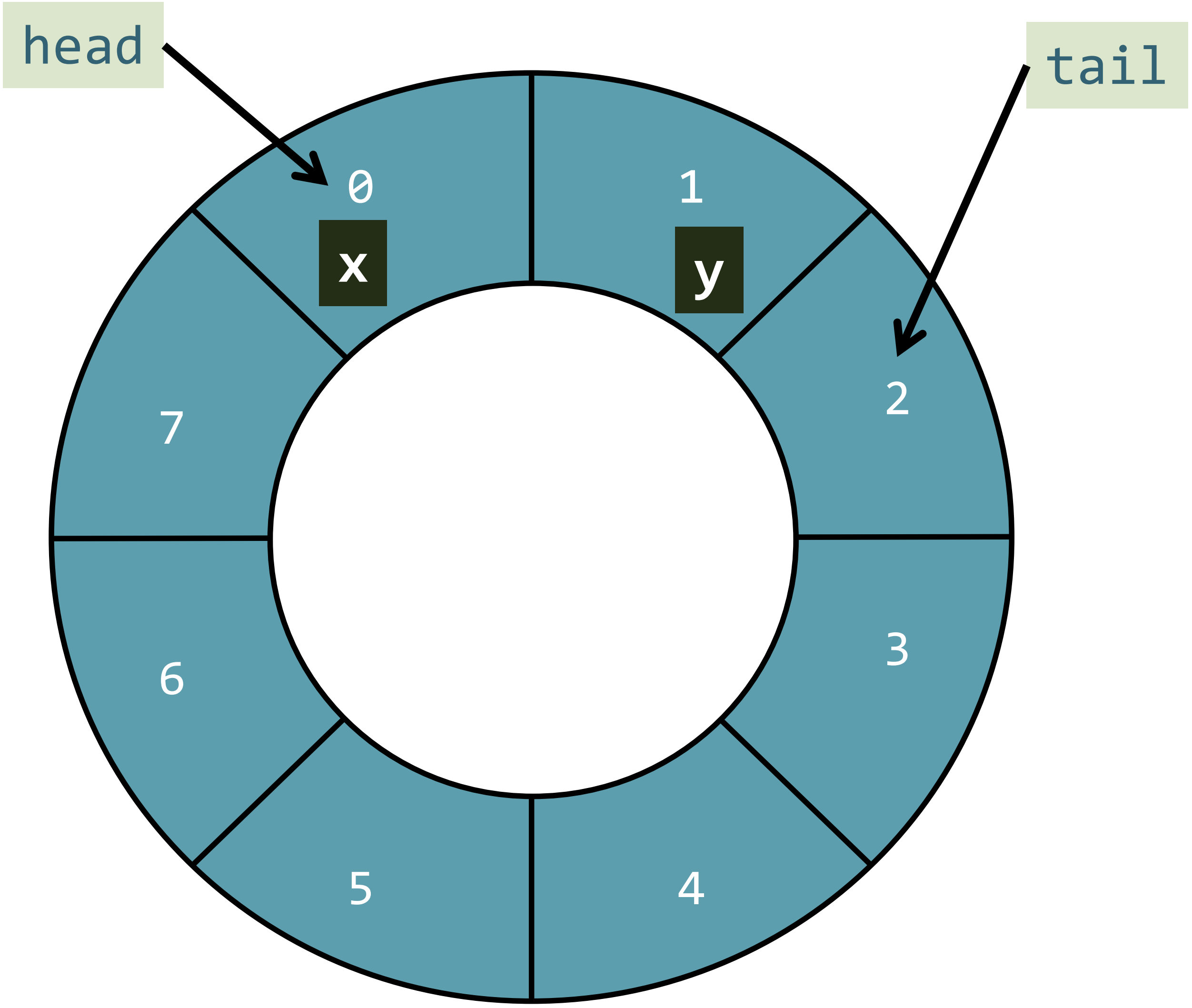
- **Insert elements at tail**
- **Remove elements from head**
  - Initial: `head = tail = 0`
  - `enq(x)`
  - `enq(y)`
  - `deq() [x]`
  - …



`capacity = 8`

Sequential Queue

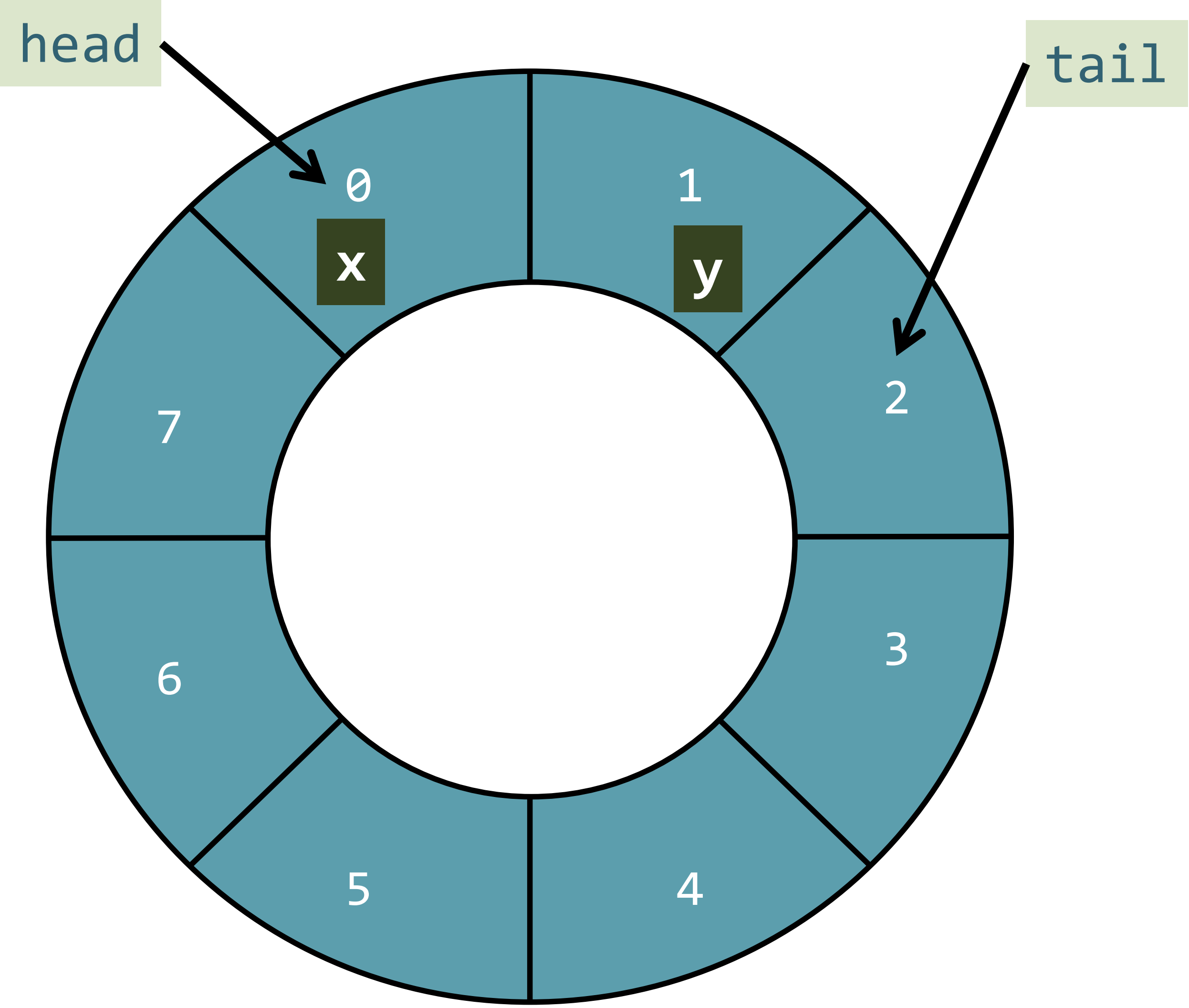
```
class Queue {
private:
  int head, tail;
  std::vector<Item> items;

public:
  Queue(int capacity) {
    head = tail = 0;
    items.resize(capacity);
  }

  // ...

};
```



capacity = 8

Sequential Queue

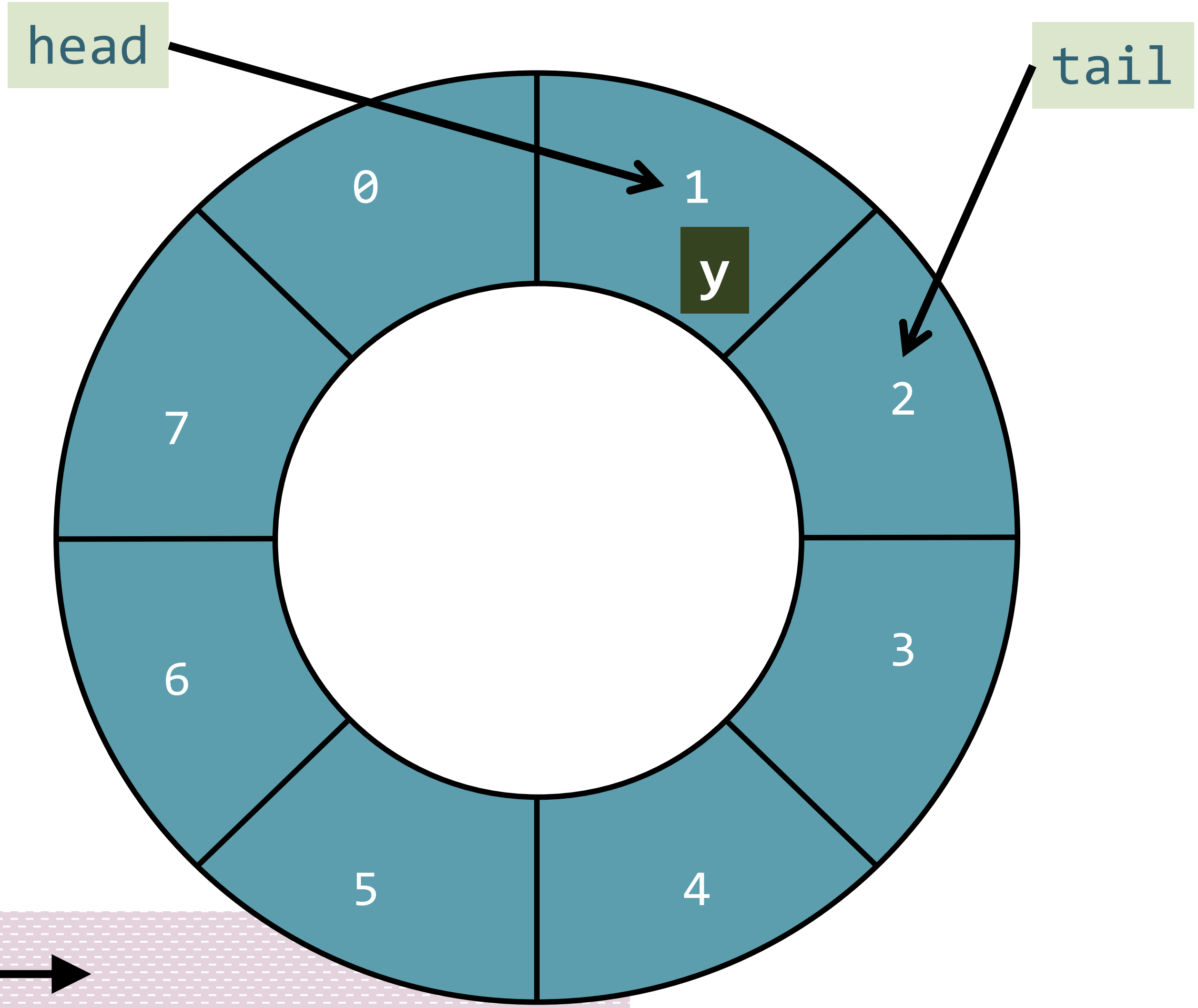
```
class Queue {
  // ...

public:
  void enq(Item x) {
    if((tail+1)%items.size() == head) {
      throw FullException;
    }
    items[tail] = x;
    tail = (tail+1)%items.size();
  }

  Item deq() {
    if(tail == head) {
      throw EmtpyException;
    }
    Item item = items[head];
    head = (head+1)%items.size();
    return item;
  }
};
```

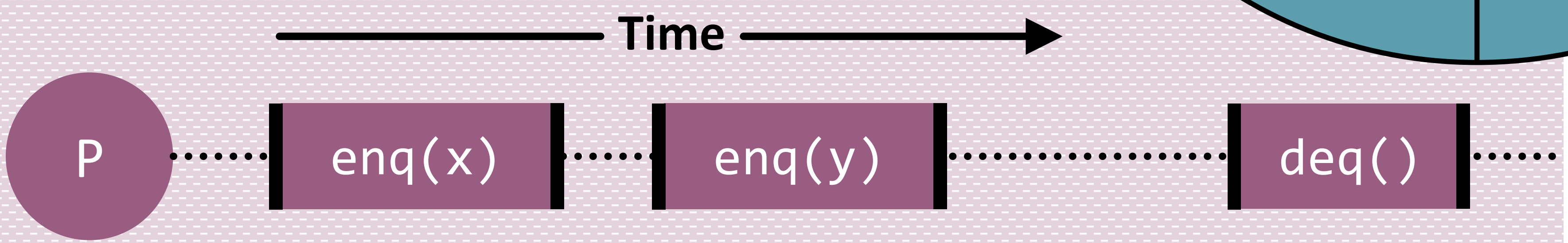


capacity = 8

Sequential Execution

- **(The) one process executes operations one at a time**
  - Sequential ☺
- **Semantics of operation defined by specification of the class**
  - Preconditions and postconditions
    e.g., Hoare logic

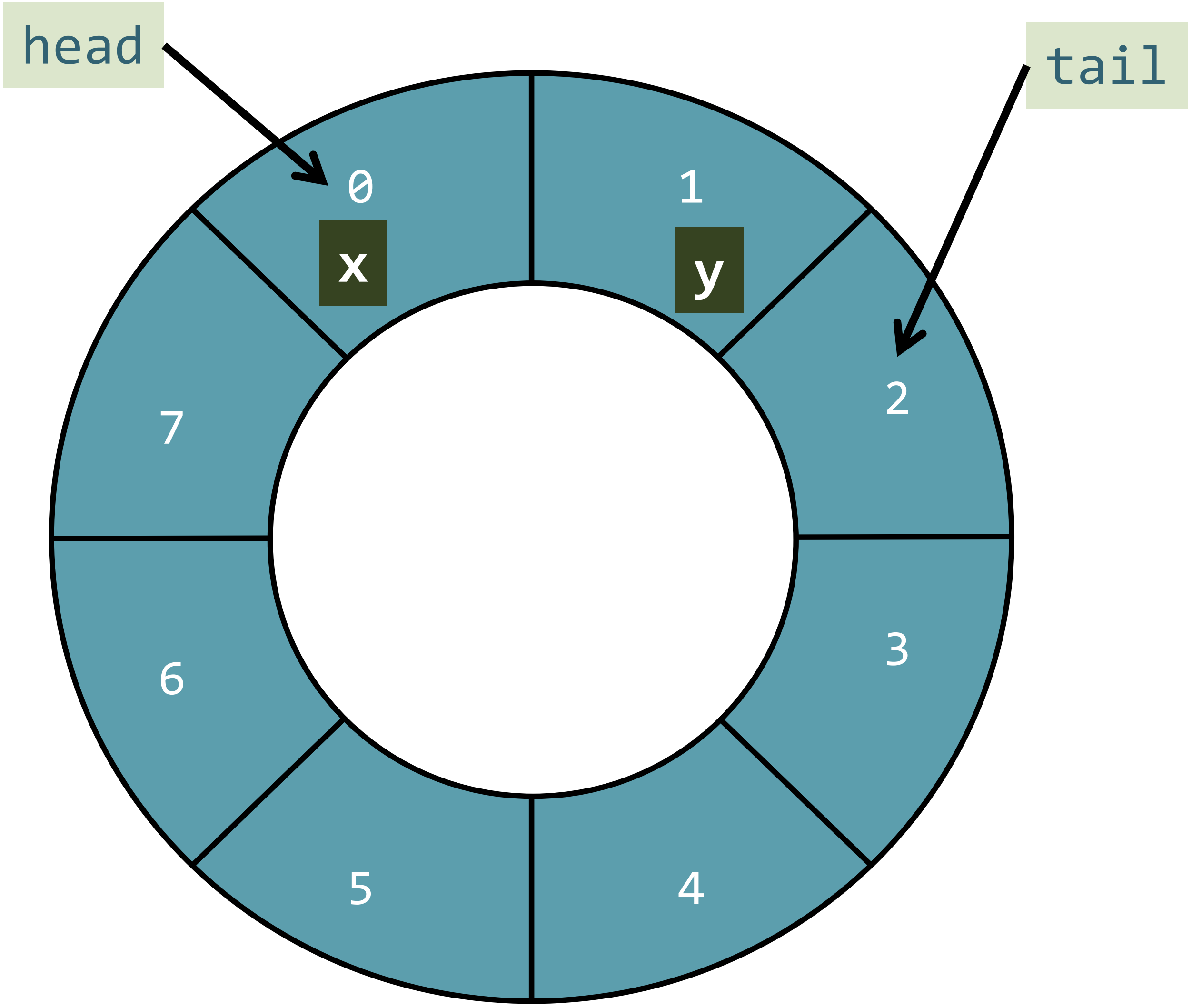




Design by Contract!

- **Preconditions:**
  - Specify conditions that must hold before method executes
  - Involve state and arguments passed
  - Specify obligations a client must meet before calling a method
- **Example: enq()**
  - Queue must not be full!

```
class Queue {
  // ...
  void enq(Item x) {
    assert((tail+1)%items.size() != head);
    // ...
  }
};
```

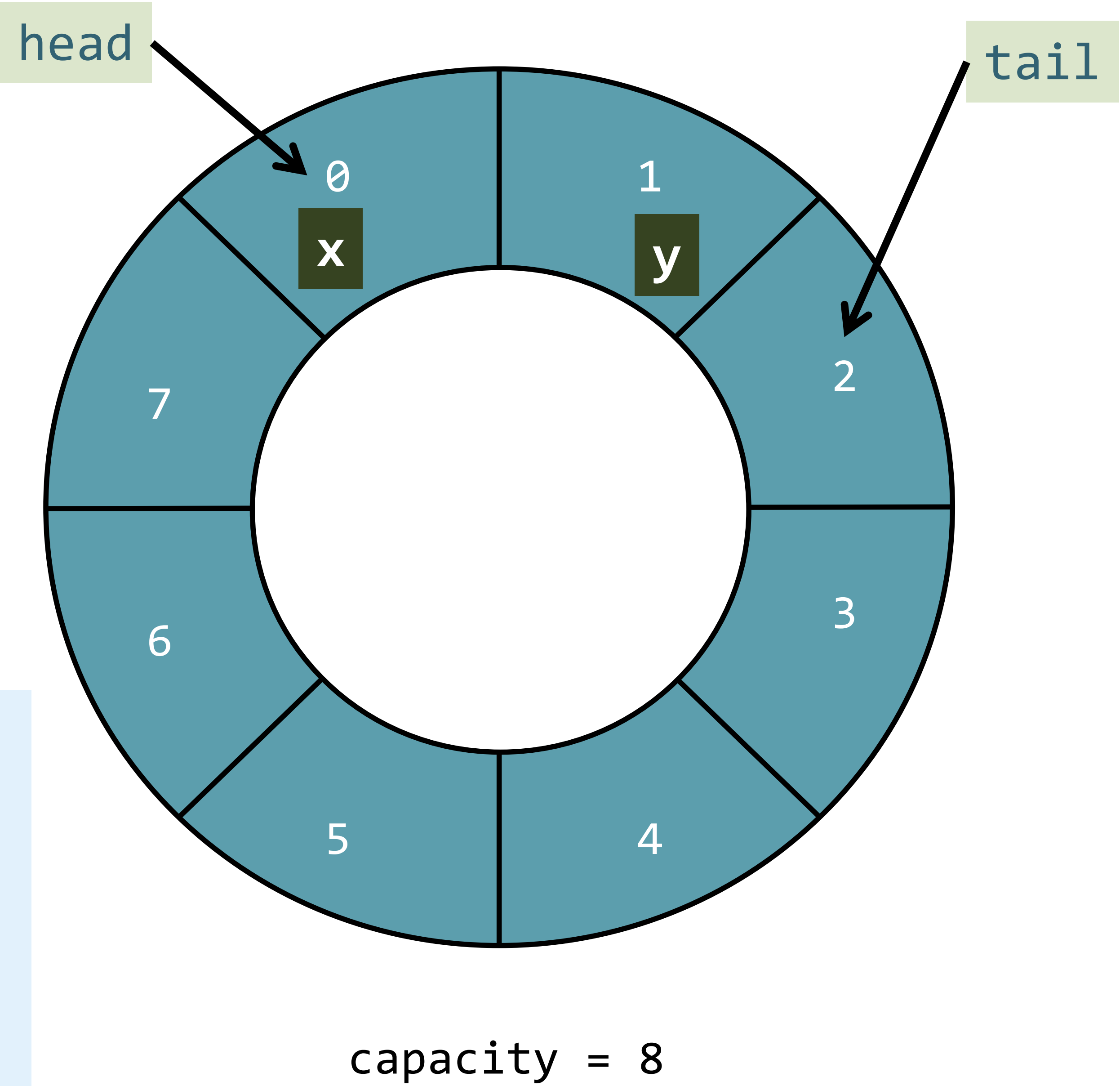


capacity = 8

Design by Contract!

- **Postconditions:**
  - Specify conditions that must hold after method executed
  - Involve old state and arguments passed
- **Example: enq()**
  - Queue must contain element!

```
class Queue {
  // ...
  void enq(Item x) {
    // ...
    assert(
      (tail == (old_tail + 1)%items.size()) &&
      (items[old_tail] == x) );
  }
};
```



Sequential specification

- **if(precondition)**
  - Object is in a **specified state**
- **then(postcondition)**
  - The method returns a particular value or
  - Throws a particular exception **and**
  - Leaves the object in a **specified state**

- **Invariants**
  - Specified conditions (e.g., object state) must hold **anytime** a client could invoke an objects method!

Advantages of sequential specification

- **State between method calls is defined**
  - Enables reasoning about objects
  - Interactions between methods captured by side effects on object state

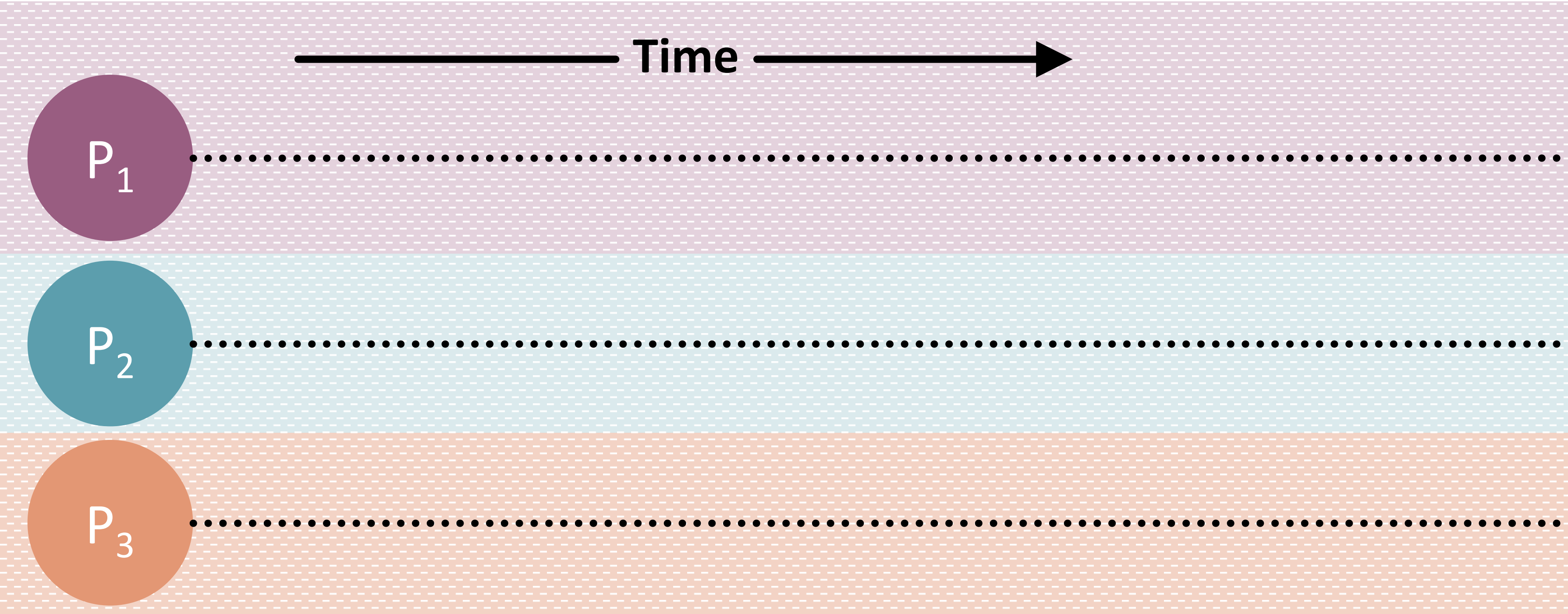
- **Enables reasoning about each method in isolation**
  - Contracts for each method
  - Local state changes global state

- **Adding new methods**
  - Only reason about state changes that the new method causes
  - If invariants are kept: **no need to check old methods**
  - **Modularity!**

Concurrent execution - State

- **Concurrent threads invoke methods on possibly shared objects**
  - At overlapping time intervals!

Property	Sequential	Concurrent
State	Meaningful only between method executions	Overlapping method executions → object may never be "between method executions"

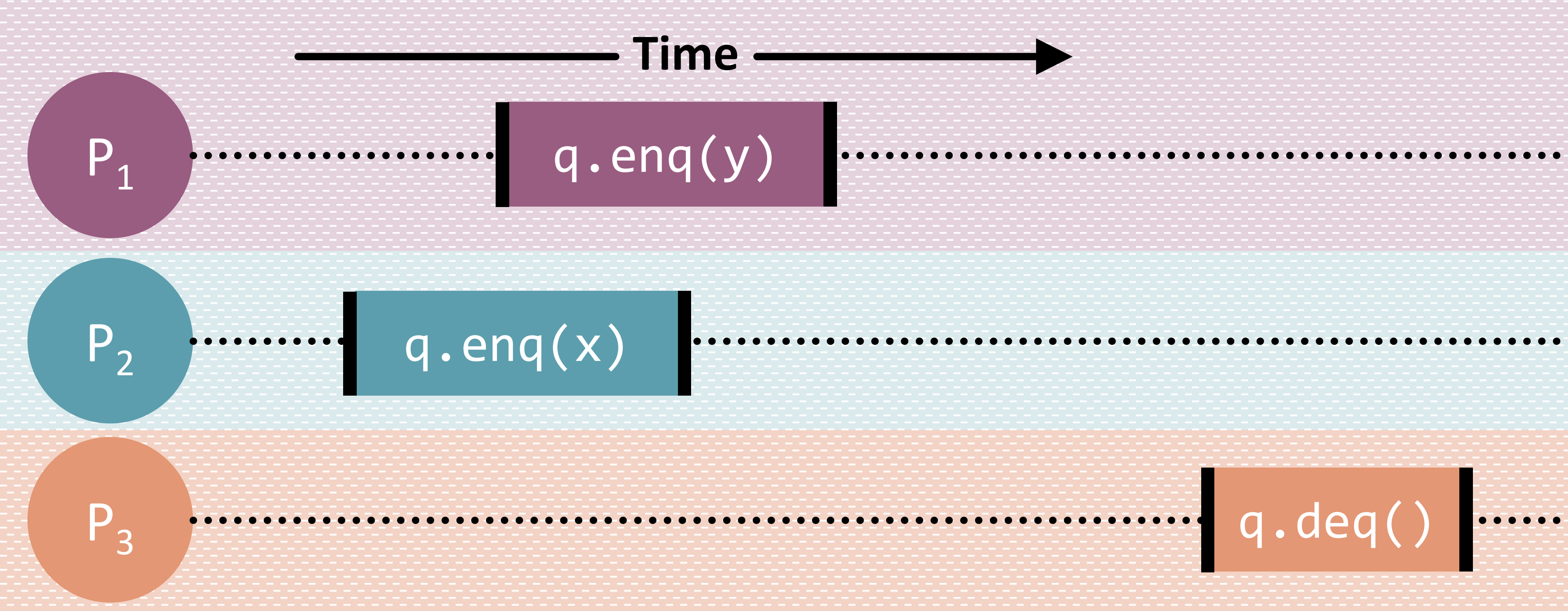


Each method execution takes some non-zero amount of time!

Concurrent execution - State

- **Concurrent threads invoke methods on possibly shared objects**
  - At overlapping time intervals!

Property	Sequential	Concurrent
State	Meaningful only between method executions	Overlapping method executions → object may never be "between method executions"



Each method execution takes some non-zero amount of time!

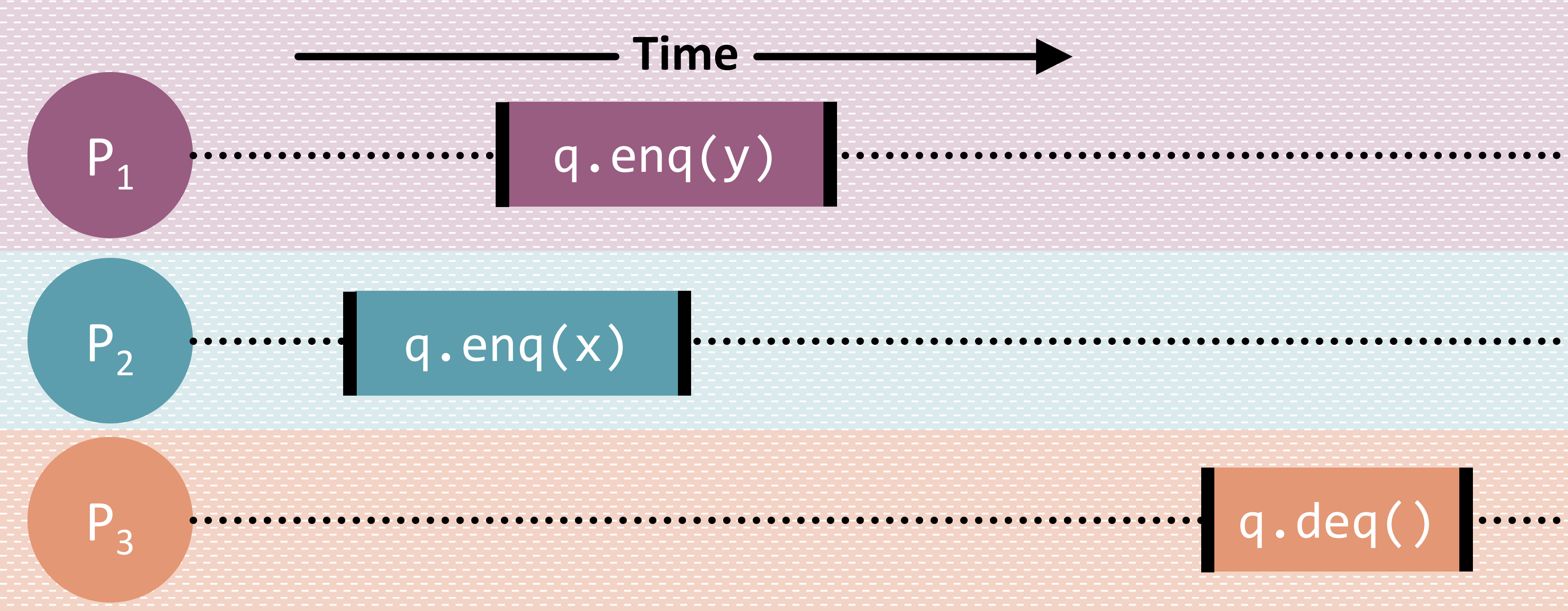
Concurrent execution - Reasoning

- **Reasoning must now include all possible interleavings**
  - Of changes caused by methods themselves

Property	Sequential	Concurrent
Reasoning	Consider each method in isolation; invariants on state before/after execution.	Need to consider all possible interactions; all intermediate states during execution

That is, now we have to consider what will happen if we execute:

- **enq()** concurrently with **enq()**
- **deq()** concurrently with **deq()**
- **deq()** concurrently with **enq()**



Each method execution takes some non-zero amount of time!

Concurrent execution - Method addition

- **Reasoning must now include all possible interleavings**
  - Of changes caused by and methods themselves

Property	Sequential	Concurrent
Add Method	Without affecting other methods; invariants on state before/after execution.	Everything can potentially interact with everything else

- **Consider adding a method that returns the last item enqueued**

```
Item peek() {
  if(tail == head) throw EmptyException;
  return items[head];
}
```

```
void enq(Item x) {
  items[tail] = x;
  tail = (tail+1) % items.size();
}
```

```
Item deq() {
  Item item = items[head];
  head = (head+1) % items.size();
}
```

  - If **peek()** and **enq()** run concurrently: what if tail has not yet been incremented?
  - If **peek()** and **deq()** run concurrently: what if last item is being dequeued?

Concurrent objects

- **How do we describe one?**
  - No pre-/postconditions ☹
- **How do we implement one?**
  - Plan for quadratic or exponential number of interactions and states
- **How do we tell if an object is correct?**
  - Analyze all quadratic or exponential interactions and states

Concurrent objects

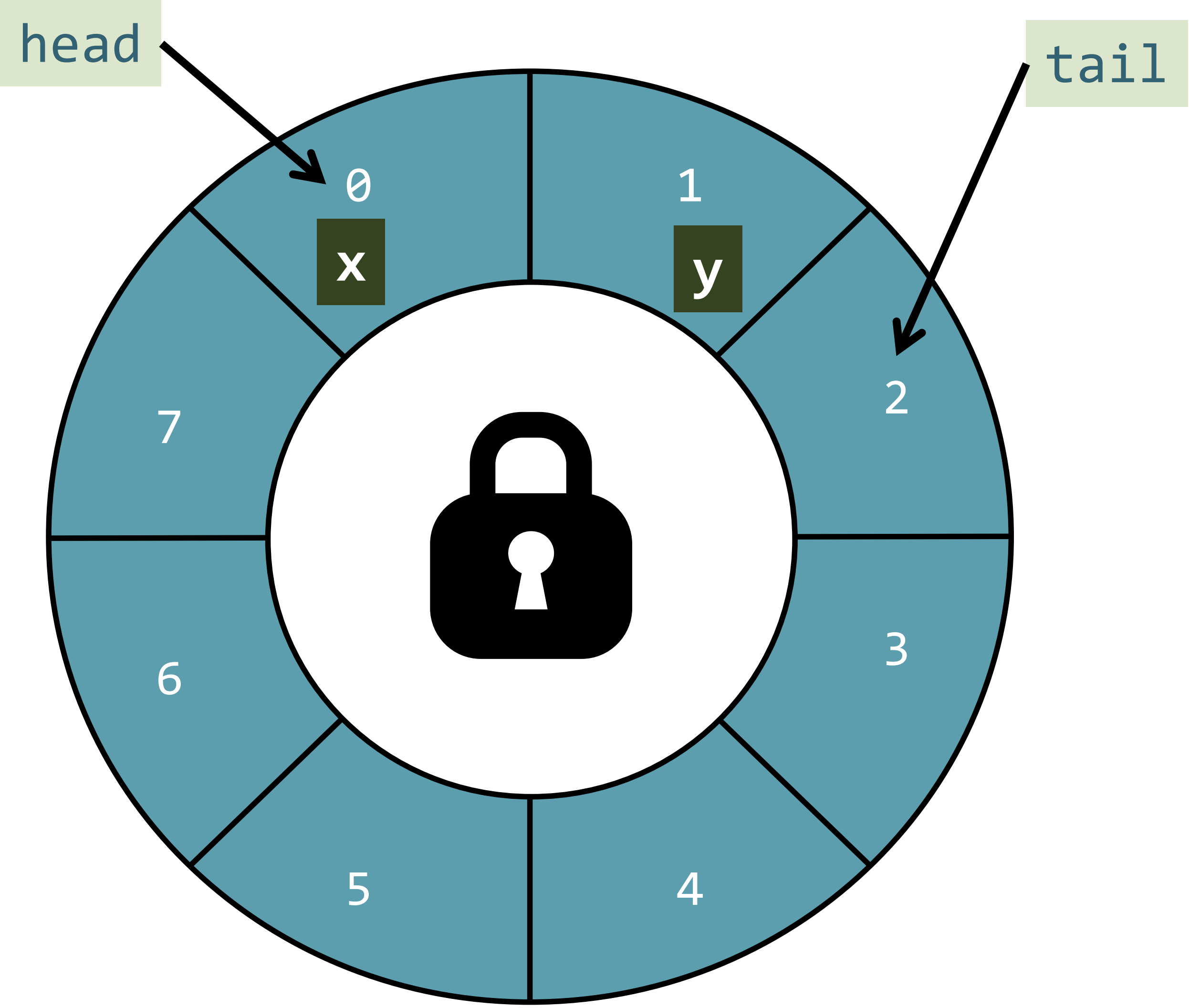
- **How do we describe one?**
  - No pre-/postconditions ☹
- **How do we implement one?**
  - Plan for quadratic or exponential number of interactions and states
- **How do we tell if an object is correct?**
  - Analyze all quadratic or exponential interactions and states

Is it time to panic for (parallel) software engineers?
Who has a solution?

Lock-based queue

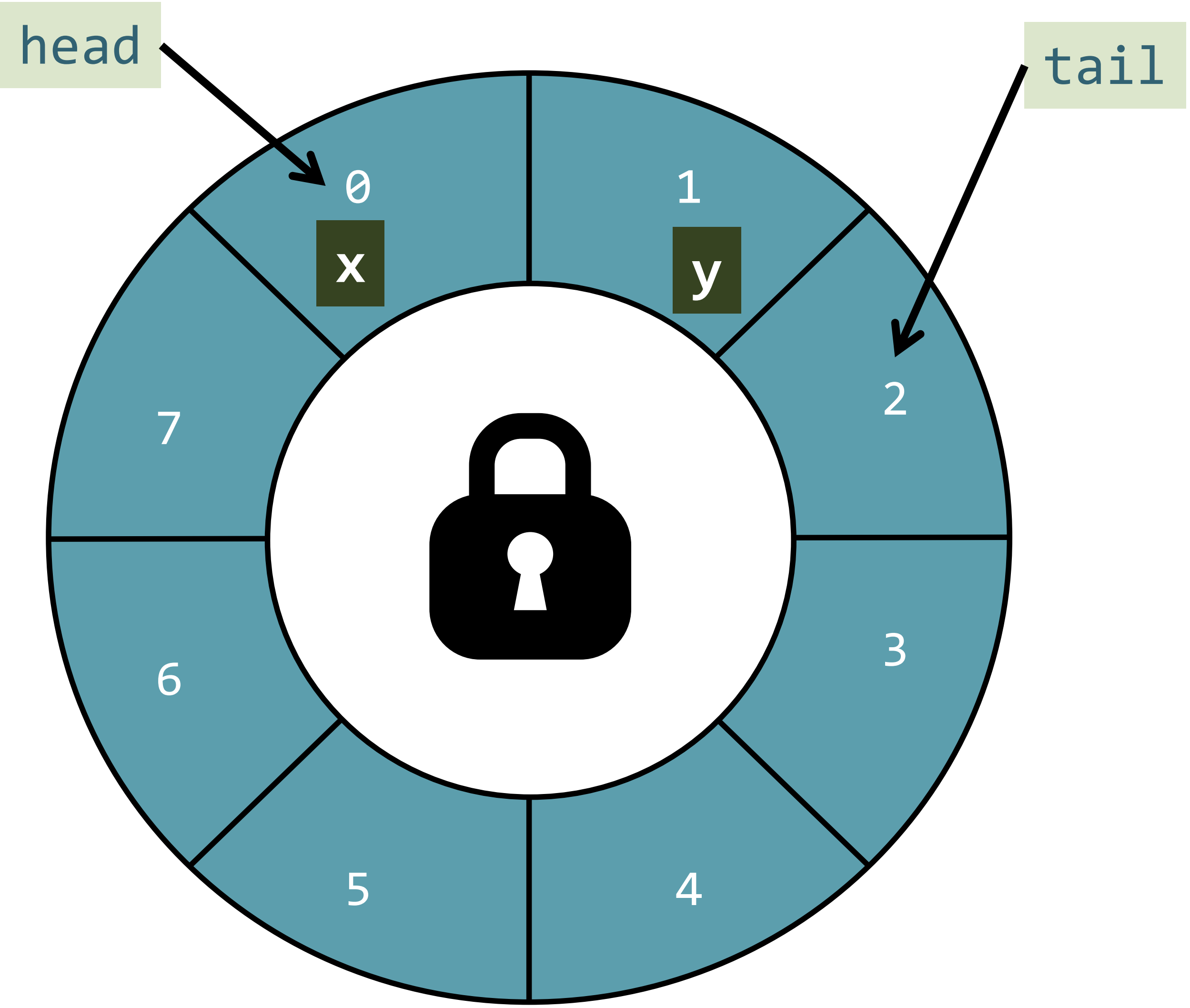
```
class Queue {
private:
  int head, tail;
  std::vector<Item> items;
  std::mutex lock;

public:
  Queue(int capacity) {
    head = tail = 0;
    items.resize(capacity);
  }
  // ...
};
```

We can use the lock to protect Queue's fields.



Lock-based queue

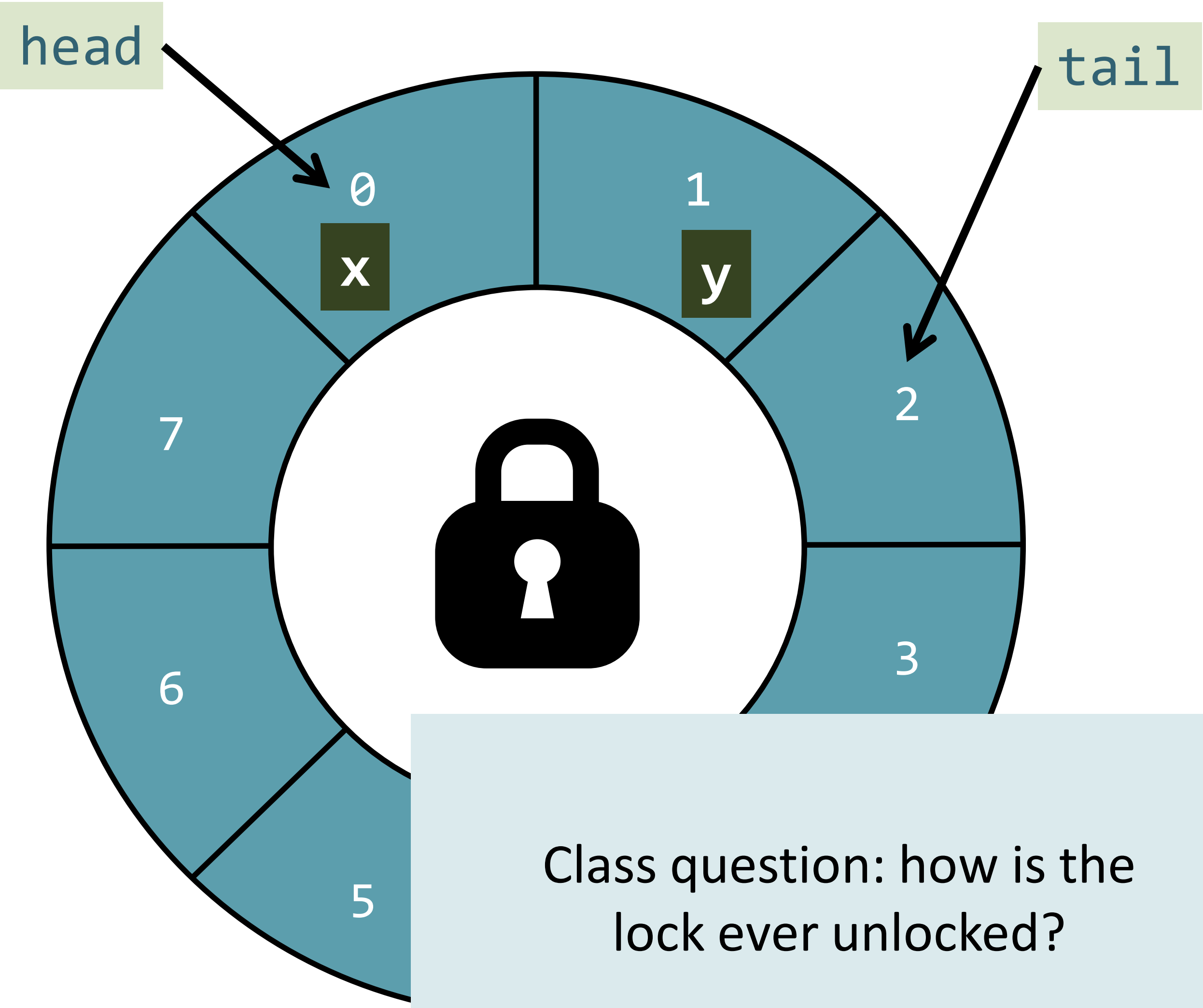
```
class Queue {
  // ...
public:
  void enq(Item x) {
    std::lock_guard<std::mutex> l(lock);
    if((tail+1)%items.size()==head) {
      throw FullException;
    }
    items[tail] = x;
    tail = (tail+1)%items.size();
  }

  Item deq() {
    std::lock_guard<std::mutex> l(lock);
    if(tail == head) {
      throw EmptyException;
    }
    Item item = items[head];
    head = (head+1)%items.size();
    return item;
  }
};
```



One of C++'s ways of implementing a **critical section**

Lock-based queue

```
class Queue {
  // ...
public:
  void enq(Item x) {
    std::lock_guard<std::mutex> l(lock);
    if((tail+1)%items.size()==head) {
      throw FullException;
    }
    items[tail] = x;
    tail = (tail+1)%items.size();
  }

  Item deq() {
    std::lock_guard<std::mutex> l(lock);
    if(tail == head) {
      throw EmptyException;
    }
    Item item = items[head];
    head = (head+1)%items.size();
    return item;
  }
};
```



Class question: how is the lock ever unlocked?

One of C++'s ways of implementing a **critical section**

C++ Resource Acquisition is Initialization

- **RAII – suboptimal name**
- **Can be used for locks (or any other resource acquisition)**
  - Constructor grabs resource
  - Destructor frees resource
- **Behaves as if**
  - Implicit unlock at end of block!
- **Main advantages**
  - Always unlock/free lock at exit
  - No “lost” locks due to exceptions or strange control flow (goto ☺)
  - Very easy to use

```
template <typename mutex_impl>
class lock_guard {
  mutex_impl& _mtx; // ref to the mutex

public:
  lock_guard(mutex_impl& mtx ) : _mtx(mtx) {
    _mtx.lock(); // lock mutex in constructor
  }

  ~lock_guard() {
    _mtx.unlock(); // unlock mutex in destructor
  }
};
```

Example execution

Example execution

enq( ) is called

```
void enq(Item x) {
```

Example execution

The lock is acquired

```
void enq(Item x) {
  std::lock_guard<std::mutex> l(lock);
```

Example execution

```
void enq(Item x) {
  std::lock_guard<std::mutex> l(lock);
  if((tail+1)%items.size()==head) {
```

Example execution

deq() is called by another thread

```
void enq(Item x) {
  std::lock_guard<std::mutex> l(lock);
  if((tail+1)%items.size()==head) {
    throw FullException;
```

```
Item deq() {
```

Example execution

deq() has to wait for the lock to be released

```
void enq(Item x) {
  std::lock_guard<std::mutex> l(lock);
  if((tail+1)%items.size()==head) {
    throw FullException;
  }
```

```
Item deq() {
  std::lock_guard<std::mutex> l(lock);
```

Example execution

deq( ) has to wait for the lock to be released

```
void enq(Item x) {
  std::lock_guard<std::mutex> l(lock);
  if((tail+1)%items.size()==head) {
    throw FullException;
  }
  items[tail] = x;
```

```
Item deq() {
  std::lock_guard<std::mutex> l(lock);
```

Example execution

deq() has to wait for the lock to be released

```
void enq(Item x) {
  std::lock_guard<std::mutex> l(lock);
  if((tail+1)%items.size()==head) {
    throw FullException;
  }
  items[tail] = x;
  tail = (tail+1)%items.size();
```

```
Item deq() {
  std::lock_guard<std::mutex> l(lock);
```

Example execution

enq() releases the lock; deq() acquires it and proceeds.

```
void enq(Item x) {
  std::lock_guard<std::mutex> l(lock);
  if((tail+1)%items.size()==head) {
    throw FullException;
  }
  items[tail] = x;
  tail = (tail+1)%items.size();
}
```

```
Item deq() {
  std::lock_guard<std::mutex> l(lock);
```

Example execution

enq() releases the lock; deq() acquires it and proceeds.

```
void enq(Item x) {
  std::lock_guard<std::mutex> l(lock);
  if((tail+1)%items.size()==head) {
    throw FullException;
  }
  items[tail] = x;
  tail = (tail+1)%items.size();
}
```

```
Item deq() {
  std::lock_guard<std::mutex> l(lock);



  if(tail == head) {
```

Example execution

enq() releases the lock; deq() acquires it and proceeds.

```
void enq(Item x) {
  std::lock_guard<std::mutex> l(lock);
  if((tail+1)%items.size()==head) {
    throw FullException;
  }
  items[tail] = x;
  tail = (tail+1)%items.size();
}
```

```



Item deq() {
  std::lock_guard<std::mutex> l(lock);



  if(tail == head) {
    throw EmptyException;
```

Example execution

enq() releases the lock; deq() acquires it and proceeds.

```
void enq(Item x) {
  std::lock_guard<std::mutex> l(lock);
  if((tail+1)%items.size()==head) {
    throw FullException;
  }
  items[tail] = x;
  tail = (tail+1)%items.size();
}
```

```
Item deq() {
  std::lock_guard<std::mutex> l(lock);



  if(tail == head) {
    throw EmptyException;
  }
```

Example execution

enq() releases the lock; deq() acquires it and proceeds.

```
void enq(Item x) {
  std::lock_guard<std::mutex> l(lock);
  if((tail+1)%items.size()==head) {
    throw FullException;
  }
  items[tail] = x;
  tail = (tail+1)%items.size();
}
```

```
Item deq() {
  std::lock_guard<std::mutex> l(lock);



  if(tail == head) {
    throw EmptyException;
  }
  Item item = items[head];
```

Example execution

enq() releases the lock; deq() acquires it and proceeds.

```
void enq(Item x) {
  std::lock_guard<std::mutex> l(lock);
  if((tail+1)%items.size()==head) {
    throw FullException;
  }
  items[tail] = x;
  tail = (tail+1)%items.size();
}
```

```
Item deq() {
  std::lock_guard<std::mutex> l(lock);


  if(tail == head) {
    throw EmptyException;
  }
  Item item = items[head];
  head = (head+1)%items.size();
```

Example execution

enq() releases the lock; deq() acquires it and proceeds.

```
void enq(Item x) {
  std::lock_guard<std::mutex> l(lock);
  if((tail+1)%items.size()==head) {
    throw FullException;
  }
  items[tail] = x;
  tail = (tail+1)%items.size();
}
```

```
Item deq() {
  std::lock_guard<std::mutex> l(lock);


  if(tail == head) {
    throw EmptyException;
  }
  Item item = items[head];
  head = (head+1)%items.size();
  return item;
```

Example execution

deq() releases the lock

```
void enq(Item x) {
  std::lock_guard<std::mutex> l(lock);
  if((tail+1)%items.size()==head) {
    throw FullException;
  }
  items[tail] = x;
  tail = (tail+1)%items.size();
}
```

```
Item deq() {
  std::lock_guard<std::mutex> l(lock);



  if(tail == head) {
    throw EmptyException;
  }
  Item item = items[head];
  head = (head+1)%items.size();
  return item;
}
```

Example execution

```
void enq(Item x) {
  std::lock_guard<std::mutex> l(lock);
  if((tail+1)%items.size()==head) {
    throw FullException;
  }
  items[tail] = x;
  tail = (tail+1)%items.size();
}
```

```
Item deq() {
  std::lock_guard<std::mutex> l(lock);



  if(tail == head) {
    throw EmptyException;
  }
  Item item = items[head];
  head = (head+1)%items.size();
  return item;
}
```

Example execution

```
void enq(Item x) {
  std::lock_guard<std::mutex> l(lock);
  if((tail+1)%items.size()==head) {
    throw FullException;
  }
  items[tail] = x;
  tail = (tail+1)%items.size();
}
```

```
Item deq() {
  std::lock_guard<std::mutex> l(lock);

  if(tail == head) {
    throw EmptyException;
  }
  Item item = items[head];
  head = (head+1)%items.size();
  return item;
}
```

enq(x)

deq()

Methods effectively execute one after another, sequentially.

Correctness – end of interlude

- **Is the locked queue correct?**
  - Yes, only one thread has access if locked correctly
  - Allows us again to reason about pre- and postconditions
  - Smells a bit like sequential consistency, no?
- **Class question: What is the problem with this approach?**

Correctness – end of interlude

- **Is the locked queue correct?**
  - Yes, only one thread has access if locked correctly
  - Allows us again to reason about pre- and postconditions
  - Smells a bit like sequential consistency, no?
- **Class question: What is the problem with this approach?**
  - Same as for SC ☺

It does not scale!
What is the solution here?

Back to memory models: Language Memory Models

- **Which transformations/reorderings can be applied to a program**
- **Affects platform/system**
  - Compiler, (VM), hardware
- **Affects programmer**
  - What are possible semantics/output
  - Which communication between threads is legal?
- **Without memory model**
  - Impossible to even define “legal” or “semantics” when data is accessed concurrently
- **A memory model is a contract**
  - Between platform and programmer

History of Memory Models

- **Java's original memory model was broken [1]**
  - Difficult to understand => widely violated
  - Did not allow reorderings as implemented in standard VMs
  - Final fields could appear to change value without synchronization
  - Volatile writes could be reordered with normal reads and writes

  => counter-intuitive for most developers
- **Java memory model was revised [2]**
  - Java 1.5 (JSR-133)
  - Still some issues (operational semantics definition [3])
- **C/C++ didn't even have a memory model until much later**
  - Not able to make any statement about threaded semantics!
  - Introduced in C++11 and C11
  - Based on experience from Java, much more conservative

[1] Pugh: "The Java Memory Model is Fatally Flawed", CCPE 2000

[2] Manson, Pugh, Adve: "The Java memory model", POPL'05

[3] Aspinall, Sevcik: "Java memory model examples: Good, bad and ugly", VAMP'07

Everybody wants to optimize

- **Language constructs for synchronization**
  - Java: volatile, synchronized, …
  - C++: atomic, (**NOT volatile**!), mutex, …

- **Without synchronization (defined language-specific)**
  - Compiler, (VM), architecture
  - Reorder and appear to reorder memory operations
  - Maintain sequential semantics per thread
  - Other threads may observe any order (have seen examples before)

Java and C++ High-level overview

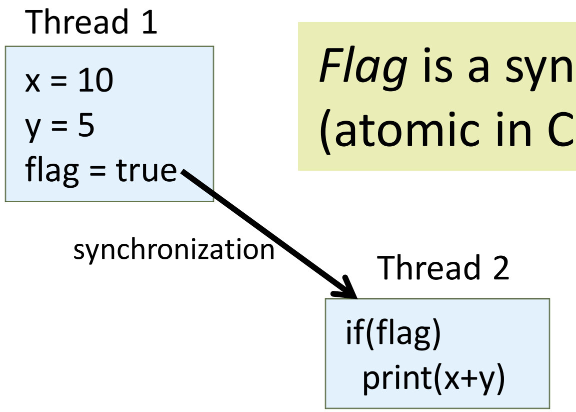
- **Relaxed memory model**
  - No global visibility ordering of operations
  - Allows for standard compiler optimizations
- **But**
  - Program order for each thread (sequential semantics)
  - Partial order on memory operations (with respect to synchronizations)
  - Visibility function defined
- **Correctly synchronized programs**
  - Guarantee sequential consistency
- **Incorrectly synchronized programs**
  - Java: maintain safety and security guarantees

    Type safety etc. (require behavior bounded by causality)
  - C++: undefined behavior

    No safety (anything can happen/change)

Communication between threads: Intuition

- **Not guaranteed unless by:**
  - Synchronization
  - Volatile/atomic variables
  - Specialized functions/classes (e.g., java.util.concurrent, …)



```
x = 10
y = 5
flag = true
```

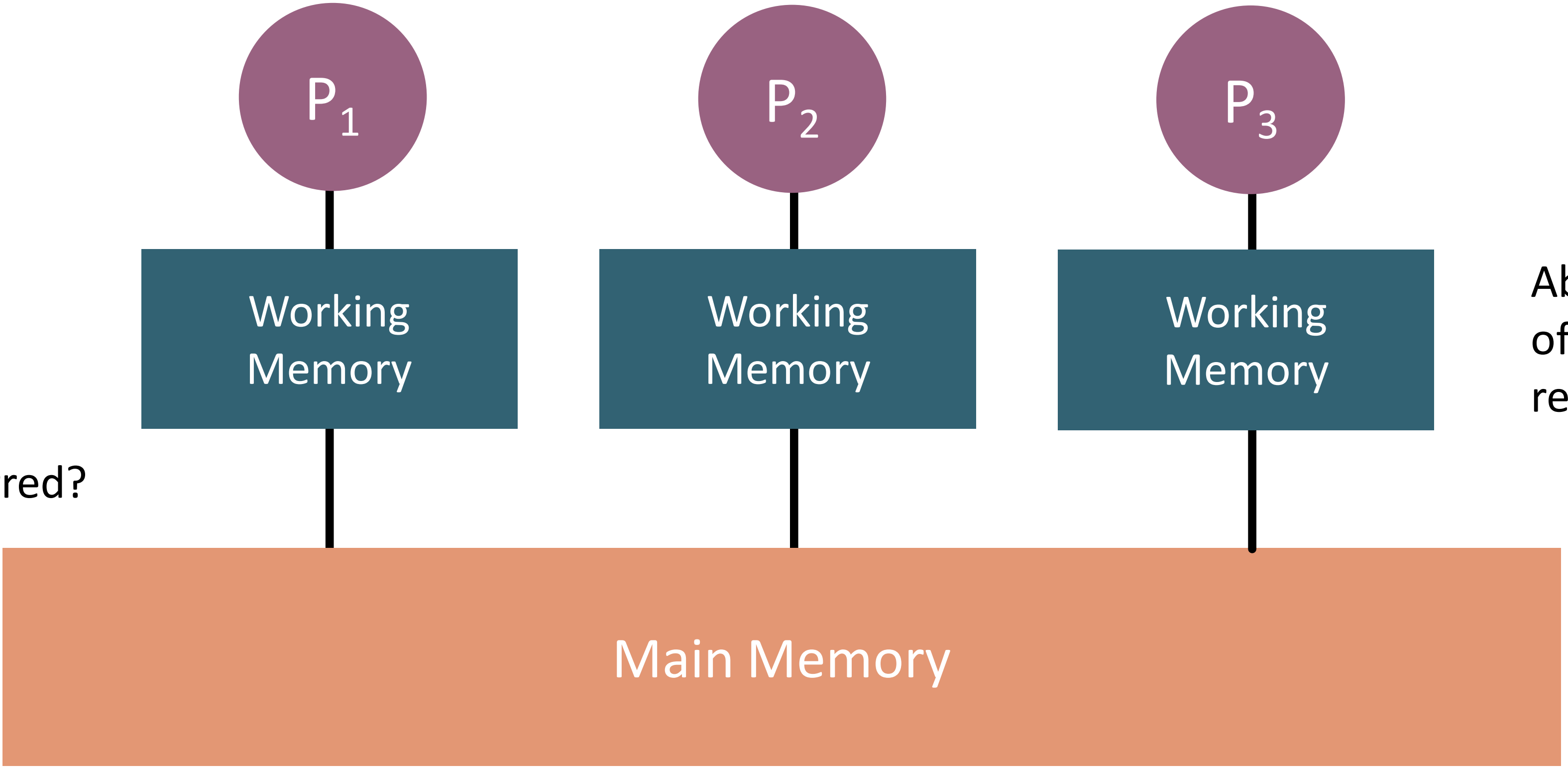
```
if(flag)
  print(x+y)
```

Flag is a synchronization variable (atomic in C++, volatile in Java),

i.e., all memory written by T1 must be visible to T2 after it reads the value true for *flag*!

Recap: Memory Model (Intuition)

- **Abstract relation between threads and memory**
  - Local thread view!



When are values transferred?

Abstraction of caches and registers.

- **Does not talk about classes, objects, methods, …**
  - Linearizability is a higher-level concept!

Locks synchronize threads *and* memory!

- **Java**

```
synchronized (lock) {
  // critical region
}
```

  - Synchronized methods as syntactic sugar

- **C++ (RAII)**

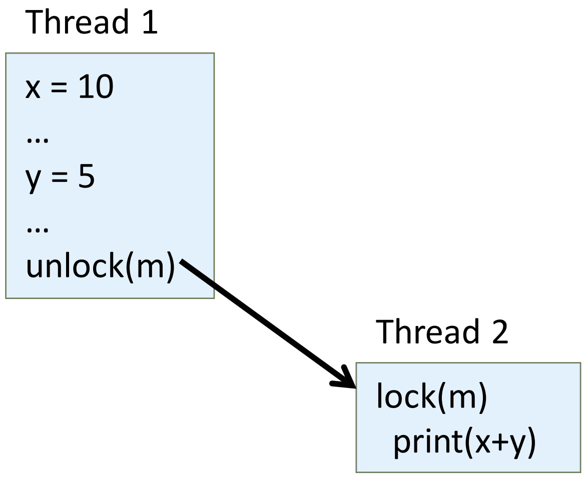
```
{
  unique_lock<mutex> l(lock);
  // critical region
}
```

  - Many flexible variants

- Semantics:
  - mutual exclusion
  - at most one thread may hold a lock at a time
  - a thread B trying to acquire a lock held by thread A blocks until thread A releases the lock
  - note: threads may wait forever (no progress guarantee!)

Memory model semantics of locks

- **Similar to synchronization variables**



Thread 1

```
x = 10
…
y = 5
…
unlock(m)
```

Thread 2

```
lock(m)
  print(x+y)
```

  - All memory accesses before an unlock …
  - are ordered before and are visible to …
  - any memory access after a matching lock!

Memory model semantics of synchronization variables

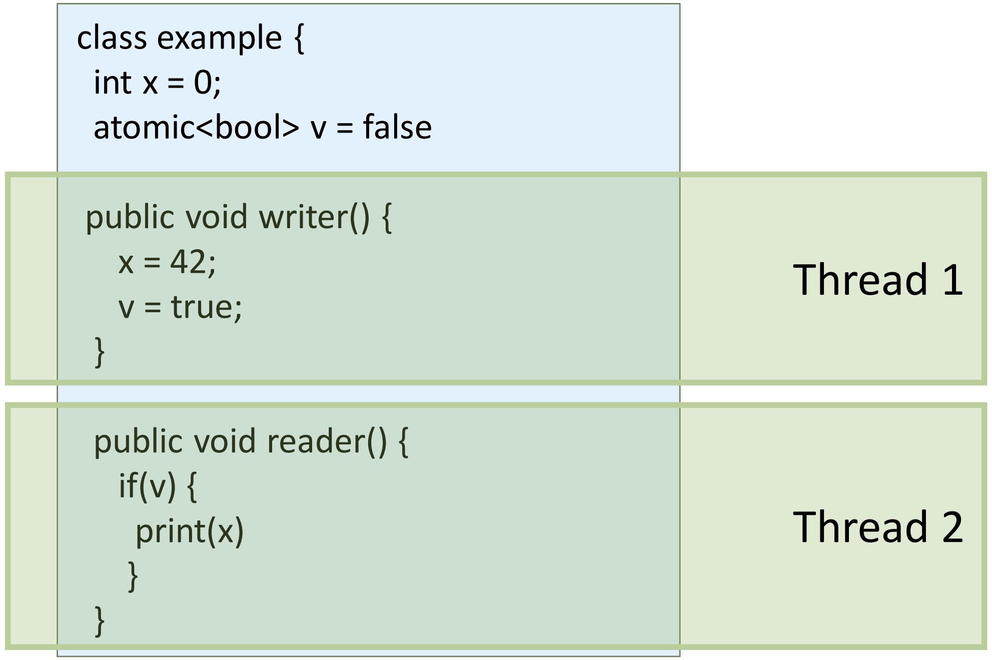
- **Variables can be declared volatile (Java) or atomic (C++)**

- **Reads and writes to synchronization variables**
  - Are totally ordered with respect to all threads
  - Must not be reordered with normal reads and writes

- **Compiler**
  - Must not allocate synchronization variables in registers
  - Must not swap variables with synchronization variables
  - May need to issue memory fences/barriers
  - …

Memory model semantics of synchronization variables

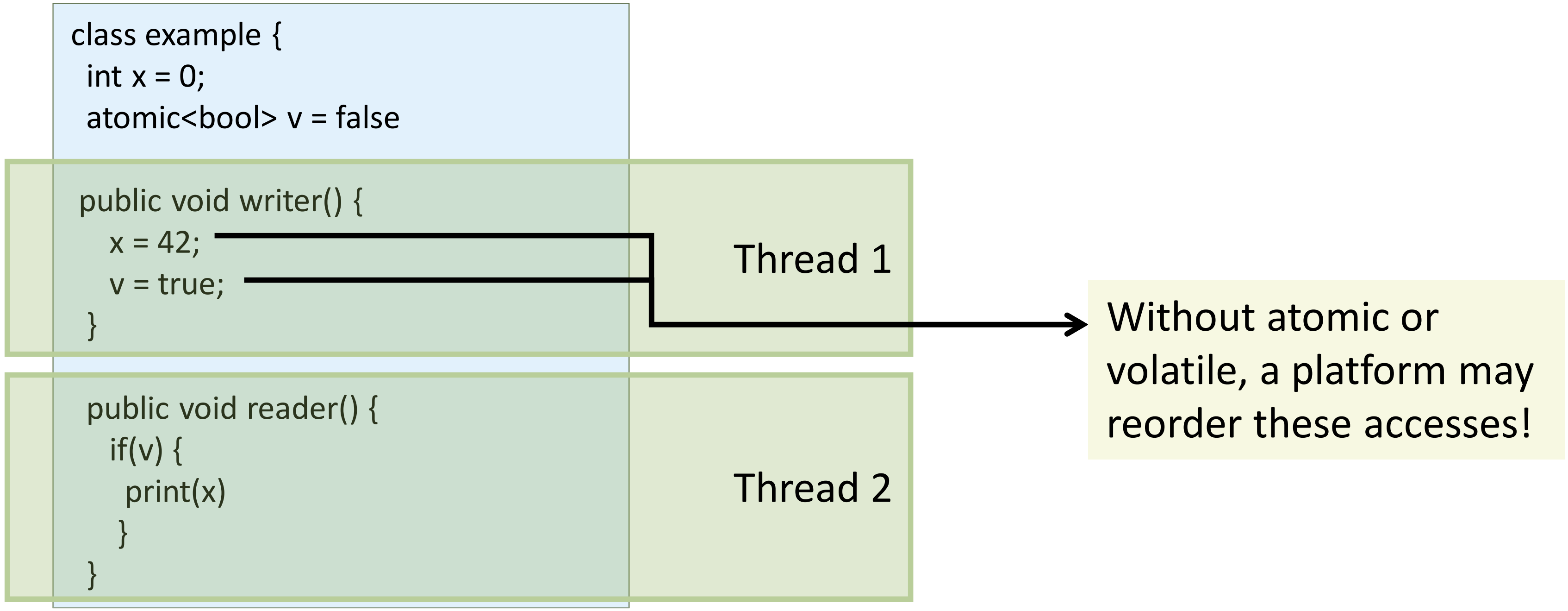
- **Write to a synchronization variable**
  - Similar memory semantics as unlock (no process synchronization!)
- **Read from a synchronization variable**
  - Similar memory semantics as lock (no process synchronization!)



```
class example {
 int x = 0;
 atomic<bool> v = false

  public void writer() {
     x = 42;
     v = true;
  }

  public void reader() {
     if(v) {
       print(x)
     }
  }
```

Memory model semantics of synchronization variables

- **Write to a synchronization variable**
  - Similar memory semantics as unlock (no process synchronization!)
- **Read from a synchronization variable**
  - Similar memory semantics as lock (no process synchronization!)

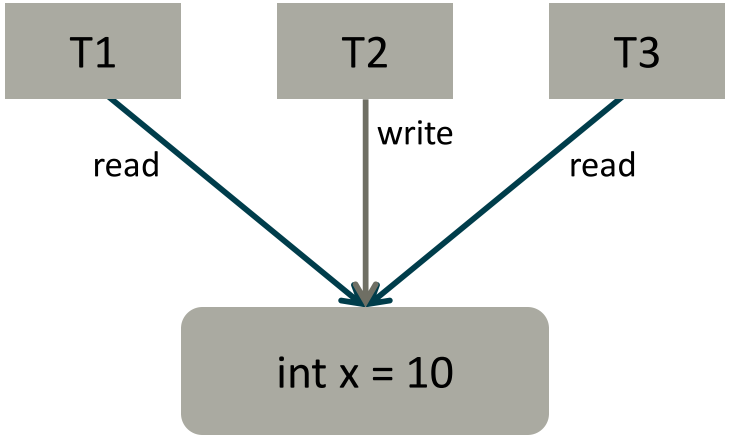


```
class example {
 int x = 0;
 atomic<bool> v = false

  public void writer() {
     x = 42;
     v = true;
  }

  public void reader() {
     if(v) {
       print(x)
     }
  }
```

Without atomic or volatile, a platform may reorder these accesses!

Intuitive memory model rules

- **Java/C++: Correctly synchronized programs will execute sequentially consistent**
- **Correctly synchronized = data-race free**
  - iff all sequentially consistent executions are free of data races
- **Two accesses to a shared memory location form a data race in the execution of a program if**
  - The two accesses are from different threads
  - At least one access is a write and
  - The accesses are not synchronized



Case Study: Implementing locks - lecture goals

- **Among the simplest concurrency constructs**
  - Yet, complex enough to illustrate many optimization principles
- **Goal 1: You understand locks in detail**
  - Requirements / guarantees
  - Correctness / validation
  - Performance / scalability

    Why you do not want to use them in many cases!
- **Goal 2: Acquire the ability to design your own locks (and other constructs)**
  - Understand techniques and weaknesses/traps
  - Extend to other concurrent algorithms

    Issues are very much the same
- **Goal 3: Understand the complexity of shared memory!**
  - Memory models in realistic settings

Preliminary Comments

- **All code examples are in C/C++ style**
  - Neither C (<11) nor C++ (<11) had a clear memory model
  - C++ is one of the languages of choice in HPC
  - Consider source as exemplary (and pay attention to the memory model)!

    In fact, many/most textbook examples are incorrect in anything but sequential consistency!

    In fact, you'll most likely not need those algorithms, but the principles will be useful!
- **x86 is really only used because it is common**
  - This does not mean that we consider the ISA or memory model elegant!
  - We assume atomic memory (or registers)!

    Usually given on x86 (easy to enforce)
- **Number of threads/processes is p, tid is the thread id**

Recap Concurrent Updates

```
const int n=1000;
volatile int a=0;
for (int i=0; i<n; ++i)
  a++;
```

- **Multi-threaded execution!**
  - Demo: value of a for p=1?

Recap Concurrent Updates

```
const int n=1000;
volatile int a=0;
for (int i=0; i<n; ++i)
  a++;
```

- **Multi-threaded execution!**
  - Demo: value of a for p=1?
  - Demo: value of a for p>1?

Recap Concurrent Updates

```
const int n=1000;
volatile int a=0;
for (int i=0; i<n; ++i)
  a++;
```

- **Multi-threaded execution!**
  - Demo: value of a for p=1?
  - Demo: value of a for p>1?

    Why? Isn’t it a single instruction?

Recap Concurrent Updates

```
const int n=1000;
volatile int a=0;
for (int i=0; i<n; ++i)
  a++;
```

gcc -O3

```
     movl  $1000, %eax     // i=n=1000
.L2:
     movl  (%rdx), %ecx    // ecx = *a
     addl  $1, %ecx        // ecx++
     subl  $1, %eax        // i—
     movl  %ecx, (%rdx)    // *a = ecx
     jne   .L2             // loop if i>0
```

- **Multi-threaded execution!**
  - Demo: value of a for p=1?
  - Demo: value of a for p>1?

    Why? Isn’t it a single instruction?

Recap Concurrent Updates

```
const int n=1000;
volatile int a=0;
for (int i=0; i<n; ++i)
  a++;
```

gcc -O3

```
      movl  $1000, %eax      // i=n=1000
.L2:
      movl  (%rdx), %ecx     // ecx = *a
      addl  $1, %ecx         // ecx++
      subl  $1, %eax         // i—
      movl  %ecx, (%rdx)     // *a = ecx
      jne   .L2              // loop if i>0
```

- **Multi-threaded execution!**
  - Demo: value of a for p=1?
  - Demo: value of a for p>1?

    Why? Isn't it a single instruction?

```
const int n=1000;
std::atomic<int> a;
a=0;
for (int i=0; i<n; ++i)
  a++;
```

Recap Concurrent Updates

```
const int n=1000;
volatile int a=0;
for (int i=0; i<n; ++i)
  a++;
```

gcc -O3

```
    movl  $1000, %eax     // i=n=1000
.L2:
    movl  (%rdx), %ecx    // ecx = *a
    addl  $1, %ecx        // ecx++
    subl  $1, %eax        // i—
    movl  %ecx, (%rdx)    // *a = ecx
    jne   .L2             // loop if i>0
```

- **Multi-threaded execution!**
  - Demo: value of a for p=1?
  - Demo: value of a for p>1?

  Why? Isn't it a single instruction?

```
const int n=1000;
std::atomic<int> a;
a=0;
for (int i=0; i<n; ++i)
  a++;
```

g++ -O3

```
    movl  $1000, %eax    // i=n=1000
    movl  $0, -24(%rsp)  // a = 0
    mfence               // a is visible!
.L2:
    lock addl $1 , -24(%rsp)  // (*a)++
    subl  $1, %eax        // i—
    jne   .L2             // loop if i>0
```

One instruction less! Performance!?

- run with larger n (10^8)
- Compiler: gcc version 4.9.2 (enabled c++11 support, -O3)
- Single-threaded execution only!

```
const int n = 1e8;
volatile int a=0;
for (int i=0; i<n; ++i)
  a++;
```

```
const int n = 1e8;
std::atomic<int> a;
a=0;
for (int i=0; i<n; ++i)
  a++;
```

Schweizer, Besta, Hoefler: “Evaluating the Cost of Atomic Operations on Modern Architectures”, ACM PACT’15

One instruction less! Performance!?

- **run with larger n (10^8)**
- **Compiler: gcc version 4.9.2 (enabled c++11 support, -O3)**
- **Single-threaded execution only!**

```
const int n = 1e8;
volatile int a=0;
for (int i=0; i<n; ++i)
  a++;
```

Demo: 0.17s

```
const int n = 1e8;
std::atomic<int> a;
a=0;
for (int i=0; i<n; ++i)
  a++;
```

Schweizer, Besta, Hoefler: “Evaluating the Cost of Atomic Operations on Modern Architectures”, ACM PACT’15

One instruction less! Performance!?

- **run with larger n (10^8)**
- **Compiler: gcc version 4.9.2 (enabled c++11 support, -O3)**
- **Single-threaded execution only!**

```
const int n = 1e8;
volatile int a=0;
for (int i=0; i<n; ++i)
  a++;
```

Demo: 0.17s

```
const int n = 1e8;
std::atomic<int> a;
a=0;
for (int i=0; i<n; ++i)
  a++;
```

Guess!

Schweizer, Besta, Hoefler: “Evaluating the Cost of Atomic Operations on Modern Architectures”, ACM PACT’15

One instruction less! Performance!?

- **run with larger n (10^8)**
- **Compiler: gcc version 4.9.2 (enabled c++11 support, -O3)**
- **Single-threaded execution only!**

```
const int n = 1e8;
volatile int a=0;
for (int i=0; i<n; ++i)
  a++;
```

Demo: 0.17s

```
const int n = 1e8;
std::atomic<int> a;
a=0;
for (int i=0; i<n; ++i)
  a++;
```

Guess!

Demo: 0.55s

Schweizer, Besta, Hoefler: “Evaluating the Cost of Atomic Operations on Modern Architectures”, ACM PACT’15

Some Statistics

- **Nondeterministic execution**
  - Result depends on timing (probably not desired)
- **What do you think are the most significant results?**
  - Running two threads on Core i5 dual core
  - a=1000? 2000? 1500? 1223? 1999?

```
const int n=1000;
volatile int a=0;
for (int i=0; i<n; ++i)
  a++;
```

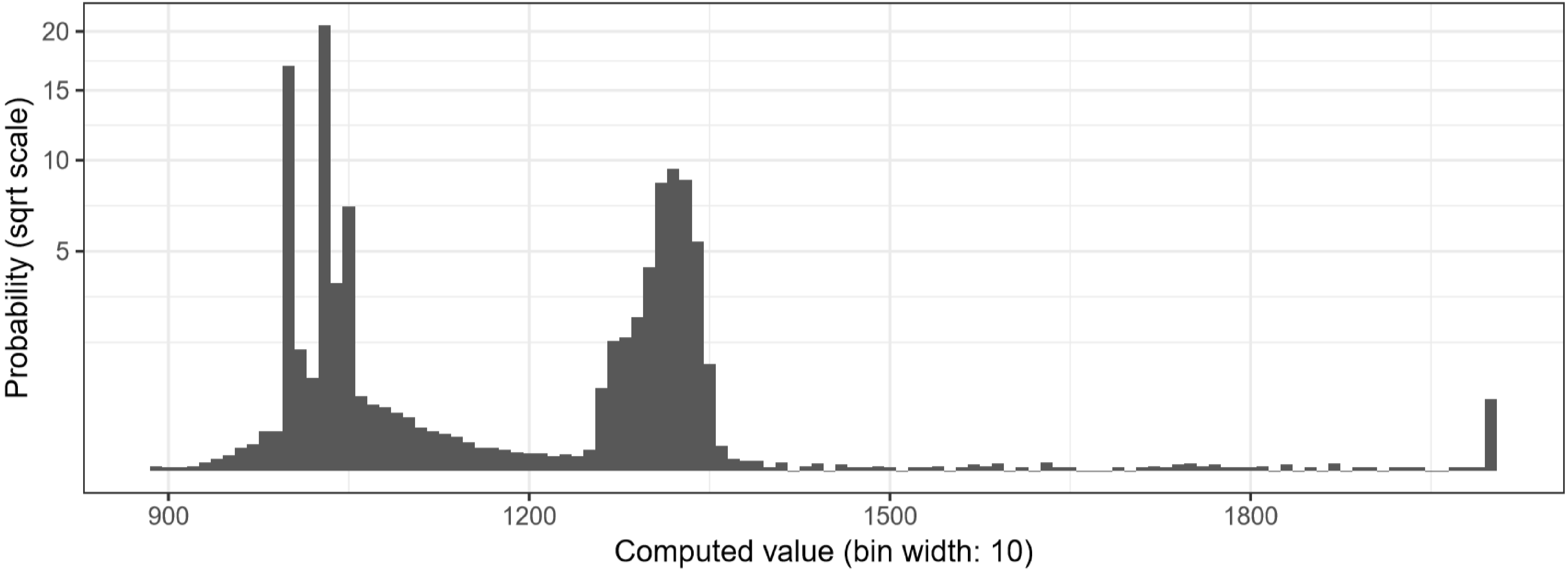
Some Statistics

- **Nondeterministic execution**
  - Result depends on timing (probably not desired)
- **What do you think are the most significant results?**
  - Running two threads on Core i5 dual core
  - a=1000? 2000? 1500? 1223? 1999?

    Demo!

```
const int n=1000;
volatile int a=0;
for (int i=0; i<n; ++i)
  a++;
```

Some Statistics

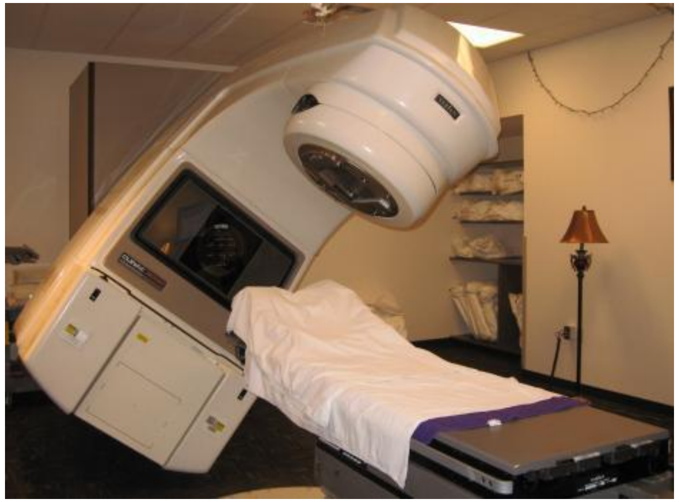


Conflicting Accesses

- **(recap) two memory accesses conflict if they can happen at *the same time* (in happens-before) and one of them is a write (store)**
- **Such a code is said to have a “race condition”**
  - Also data-race
  - Trivia around races:

    The Therac-25 killed three people
    due to a race

    A data-race lead to a large blackout
    in 2003, leaving 55 million people
    without power causing $1bn damage
- **Can be avoided by critical regions**
  - Mutually exclusive access to a set of operations



More formal: Mutual Exclusion

- **Control access to a critical region**
  - Memory accesses of all processes happen in program order (a partial order, many interleavings)

    An execution history defines a total order of memory accesses
  - Some subsets of memory accesses (issued by the same process) need to happen atomically (thread a’s memory accesses may not be interleaved with other thread’s accesses)

    To achieve linearizability!

    We need to restrict the valid executions
- **→ Requires synchronization of some sort**
  - Many possible techniques (e.g., TM, CAS, T&S, …)
  - We first discuss locks which have wait semantics

More formal: Mutual Exclusion

- **Control access to a critical region**
  - Memory accesses of all processes happen in program order (a partial order, many interleavings)

    An execution history defines a total order of memory accesses
  - Some subsets of memory accesses (issued by the same process) need to happen atomically (thread a's memory accesses may not be interleaved with other thread's accesses)

    To achieve linearizability!

    We need to restrict the valid executions
- **→ Requires synchronization of some sort**
  - Many possible techniques (e.g., TM, CAS, T&S, …)
  - We first discuss locks which have wait semantics

```
 movl      $1000, %eax     // i=1000
.L2:
     movl  (%rdx), %ecx    // ecx = *a
     addl  $1, %ecx        // ecx++
     subl  $1, %eax        // i—
     movl  %ecx, (%rdx)    // *a = ecx
     jne   .L2             // loop if i>0
```

Fixing it with locks

```
const int n=1000;
volatile int a=0;
omp_lock_t lck;
for (int i=0; i<n; ++i) {
  omp_set_lock(&lck);
  a++;
  omp_unset_lock(&lck);
}
```

```
    movl  $1000, %ebx     // i=1000
.L2:
    movq  0(%rbp), %rdi   // (SystemV CC)
    call  omp_set_lock    // get lock
    movq  0(%rbp), %rdi   // (SystemV CC)
    movl  (%rax), %edx    // edx = *a
    addl  $1, %edx        // edx++
    movl  %edx, (%rax)    // *a = edx
    call  omp_unset_lock  // release lock
    subl  $1, %ebx        // i—
    jne   .L2             // repeat if i>0
```

- What must the functions lock and unlock guarantee?

Fixing it with locks

```
const int n=1000;
volatile int a=0;
omp_lock_t lck;
for (int i=0; i<n; ++i) {
  omp_set_lock(&lck);
  a++;
  omp_unset_lock(&lck);
}
```

gcc -O3

```
     movl  $1000, %ebx      // i=1000
.L2:
     movq  0(%rbp), %rdi    // (SystemV CC)
     call  omp_set_lock     // get lock
     movq  0(%rbp), %rdi    // (SystemV CC)
     movl  (%rax), %edx     // edx = *a
     addl  $1, %edx         // edx++
     movl  %edx, (%rax)     // *a = edx
     call  omp_unset_lock // release lock
     subl  $1, %ebx         // i—
     jne   .L2              // repeat if i>0
```

- What must the functions lock and unlock guarantee?
  - #1: prevent two threads from simultaneously entering CR

    i.e., accesses to CR must be mutually exclusive!

Fixing it with locks

```
const int n=1000;
volatile int a=0;
omp_lock_t lck;
for (int i=0; i<n; ++i) {
  omp_set_lock(&lck);
  a++;
  omp_unset_lock(&lck);
}
```

gcc -O3

```
    movl  $1000, %ebx      // i=1000
.L2:
    movq  0(%rbp), %rdi    // (SystemV CC)
    call  omp_set_lock     // get lock
    movq  0(%rbp), %rdi    // (SystemV CC)
    movl  (%rax), %edx     // edx = *a
    addl  $1, %edx         // edx++
    movl  %edx, (%rax)     // *a = edx
    call  omp_unset_lock   // release lock
    subl  $1, %ebx         // i—
    jne   .L2              // repeat if i>0
```

- What must the functions lock and unlock guarantee?
  - #1: prevent two threads from simultaneously entering CR

    i.e., accesses to CR must be mutually exclusive!
  - #2: ensure consistent memory

    i.e., stores must be globally visible before new lock is granted!

Fixing it with locks

```
const int n=1000;
volatile int a=0;
omp_lock_t lck;
for (int i=0; i<n; ++i) {
  omp_set_lock(&lck);
  a++;
  omp_unset_lock(&lck);
}
```

```
      movl   $1000, %ebx      // i=1000
.L2:
      movq   0(%rbp), %rdi    // (SystemV CC)
      call   omp_set_lock     // get lock
      movq   0(%rbp), %rdi    // (SystemV CC)
      movl   (%rax), %edx     // edx = *a
      addl   $1, %edx         // edx++
      movl   %edx, (%rax)     // *a = edx
      call   omp_unset_lock   // release lock
      subl   $1, %ebx         // i—
      jne    .L2              // repeat if i>0
```

- What must the functions lock and unlock guarantee?
  - #1: prevent two threads from simultaneously entering CR

    i.e., accesses to CR must be mutually exclusive!
  - #2: ensure consistent memory

    i.e., stores must be globally visible before new lock is granted!
- Any performance guesses (remember, 0.23s → 0.78s for atomics)

Fixing it with locks

```c
const int n=1000;
volatile int a=0;
omp_lock_t lck;
for (int i=0; i<n; ++i) {
  omp_set_lock(&lck);
  a++;
  omp_unset_lock(&lck);
}
```

gcc -O3

```
      movl   $1000, %ebx       // i=1000
.L2:
      movq   0(%rbp), %rdi     // (SystemV CC)
      call   omp_set_lock      // get lock
      movq   0(%rbp), %rdi     // (SystemV CC)
      movl   (%rax), %edx      // edx = *a
      addl   $1, %edx          // edx++
      movl   %edx, (%rax)      // *a = edx
      call   omp_unset_lock    // release lock
      subl   $1, %ebx          // i—
      jne    .L2               // repeat if i>0
```

- What must the functions lock and unlock guarantee?
  - #1: prevent two threads from simultaneously entering CR

    i.e., accesses to CR must be mutually exclusive!
  - #2: ensure consistent memory

    i.e., stores must be globally visible before new lock is granted!
- Any performance guesses (remember, 0.23s → 0.78s for atomics)
  - 2.26s

Lock Overview

- **Lock/unlock or acquire/release**
  - Lock/acquire: before entering CR
  - Unlock/release: after leaving CR
- **Semantics:**
  - Lock/unlock pairs have to match
  - Between lock/unlock, a thread holds the lock

Desired Lock Properties

?

Desired Lock Properties

- **Mutual exclusion**
  - Only one thread is on the critical region
- **Consistency**
  - Memory operations are visible when critical region is left
- **Progress**
  - If any thread a is not in the critical region, it cannot prevent another thread b from entering
- **Starvation-freedom (implies deadlock-freedom)**
  - If a thread is requesting access to a critical region, then it will eventually be granted access
- **Fairness**
  - A thread a requested access to a critical region before thread b. Did is also granted access to this region before b?
- **Performance**
  - Scaling to large numbers of contending threads

Simplified Notation (cf. Histories)

- **Time defined by precedence (a total order on events)**
  - Events are instantaneous (linearizable)
  - Threads produce sequences of events $a_0, a_1, a_2, \ldots$
  - Program statements may be repeated, denote i-th instance of a as a^i
  - Event a occurs before event b: a → b
  - An interval (a,b) is the duration between events a → b
  - Interval I_1=(a,b) precedes interval I_2=(c,d) iff b → c
- **Critical regions**
  - A critical region CR is an interval (a,b), where a is the first operation in the CR and b the last
- **Mutual exclusion**
  - Critical regions CR_A and CR_B are mutually exclusive if:

    Either $CR_A \rightarrow CR_B$ or $CR_B \rightarrow CR_A$ for all valid executions!
- **Assume atomic registers (for now)**

Peterson's Two-Thread Lock (1981)

- **Combines the first lock (request access) with the second lock (grant access)**

```
volatile int flag[2];
volatile int victim;

void lock() {
  int j = 1 - tid;
  flag[tid] = 1; // I’m interested
  victim = tid; // other goes first
  while (flag[j] && victim == tid) {}; // wait
}

void unlock() {
  flag[tid] = 0; // I’m not interested
}
```

Proof Correctness

- **Intuition:**
  - Victim is written once
  - Pick thread that wrote victim last
  - Show thread must have read flag==0
  - Show that no sequentially consistent schedule permits that

Starvation Freedom

- **(recap) definition: Every thread that calls lock() eventually gets the lock.**
  - Implies deadlock-freedom!
- **Is Peterson's lock starvation-free?**

```
volatile int flag[2];
volatile int victim;

void lock() {
  int j = 1 - tid;
  flag[tid] = 1; // I’m interested
  victim = tid; // other goes first
  while (flag[j] && victim == tid) {}; // wait
}

void unlock() {
  flag[tid] = 0; // I’m not interested
}
```

Proof Starvation Freedom

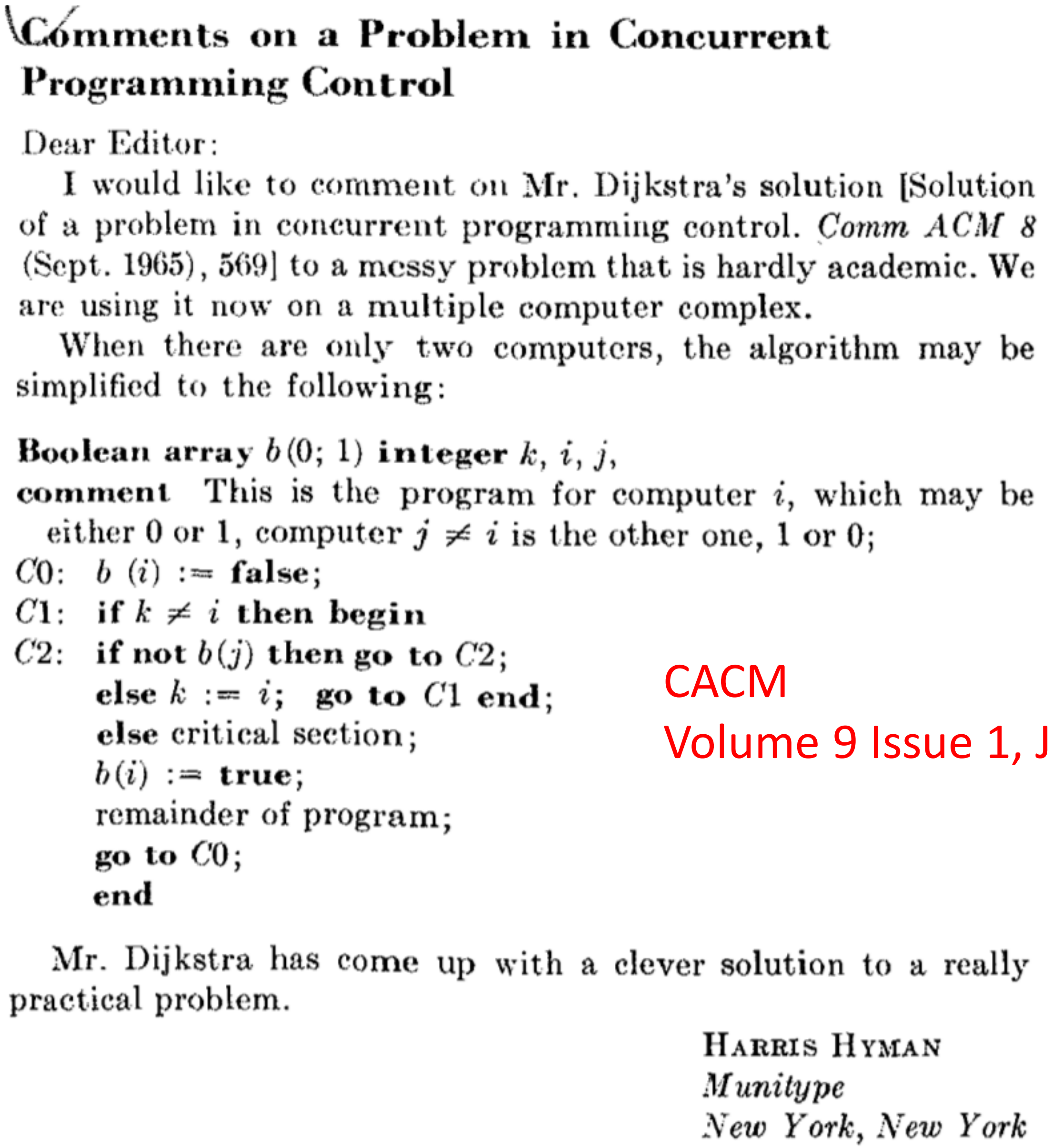
- **Intuition:**
  - Threads can only wait/starve in while()

    Until flag==0 or victim==other
  - Other thread enters lock() → sets victim to other

    Will definitely “unstuck” first thread
  - So other thread can only be stuck in lock()

    Will wait for victim==other, victim cannot block both threads → one must leave!

Case Study: Automatic Reasoning about Semantics



Comments on a Problem in Concurrent Programming Control

Dear Editor:

I would like to comment on Mr. Dijkstra's solution [Solution of a problem in concurrent programming control. *Comm ACM 8* (Sept. 1965), 569] to a messy problem that is hardly academic. We are using it now on a multiple computer complex.

When there are only two computers, the algorithm may be simplified to the following:

```
Boolean array b(0; 1) integer k, i, j,
comment  This is the program for computer i, which may be
   either 0 or 1, computer j ≠ i is the other one, 1 or 0;
C0:  b (i) := false;
C1:  if k ≠ i then begin
C2:  if not b(j) then go to C2;
     else k := i;  go to C1 end;
     else critical section;
     b(i) := true;
     remainder of program;
     go to C0;
     end
```

Mr. Dijkstra has come up with a clever solution to a really practical problem.

HARRIS HYMAN
Munitype
New York, New York

CACM
Volume 9 Issue 1, Jan. 1966

Case Study: Automatic Reasoning about Semantics

- **Is the proposed algorithm correct?**
  - We may proof it manually

    Using tools from the last lecture

    → *reason about the state space of H*

Case Study: Automatic Reasoning about Semantics

- **Is the proposed algorithm correct?**
  - We may proof it manually

    Using tools from the last lecture

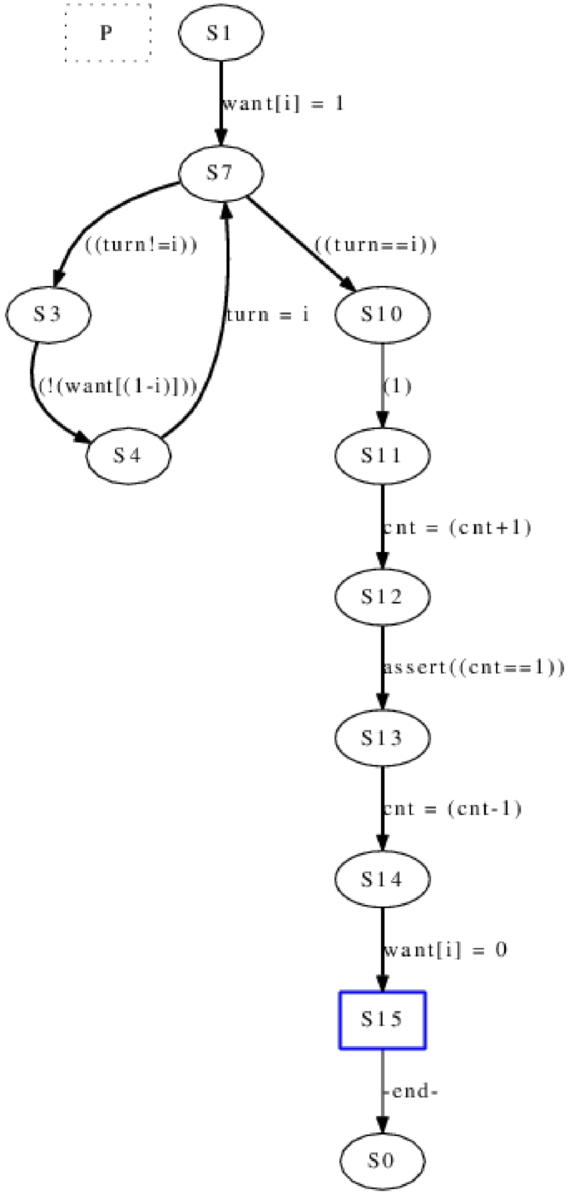
    → reason about the state space of H
  - Or use automated proofs (model checking)

    E.g., SPIN (Promela syntax)

```
bool want[2];
bool turn;
byte cnt;

proctype P(bool i)
{
 want[i] = 1;
 do
 :: (turn != i) ->
    (!want[1-i]);
    turn = i
 :: (turn == i) ->
    break
 od;
 skip; /* critical section */
 cnt = cnt+1;
 assert(cnt == 1);
 cnt = cnt-1;
 want[i] = 0
}

init { run P(0); run P(1) }
```

Case Study: Automatic Reasoning about Semantics

- **Spin tells us quickly that it found a problem**
  - A sequentially consistent order that violates mutual exclusion!
- **It’s not always that easy**
  - This example comes from the SPIN tutorial
  - More than two threads make it much more demanding!
- **More in the recitation session!**



Back to Peterson in Practice … on x86

- **Implement and run our little counter on x86**
- **100000 iterations**
  - 1.6 · 10^{-6}% errors
  - What is the problem?

```
volatile int flag[2];
volatile int victim;

void lock() {
  int j = 1 - tid;
  flag[tid] = 1; // I’m interested
  victim = tid; // other goes first
  while (flag[j] && victim == tid) {}; // wait
}

void unlock() {
  flag[tid] = 0; // I’m not interested
}
```

Peterson in Practice … on x86

- **Implement and run our little counter on x86**
- **100000 iterations**
  - $1.6 \cdot 10^{-6}$% errors
  - What is the problem?

    No sequential consistency for W(v) and R(flag[j])

```
volatile int flag[2];
volatile int victim;

void lock() {
  int j = 1 - tid;
  flag[tid] = 1; // I'm interested
  victim = tid; // other goes first
  asm("mfence");
  while (flag[j] && victim == tid) {}; // wait
}

void unlock() {
  flag[tid] = 0; // I'm not interested
}
```

Peterson in Practice … on x86

- **Implement and run our little counter on x86**
- **Many iterations**
  - $1.6 \cdot 10^{-6}$% errors
  - What is the problem?

    No sequential consistency for W(v) and R(flag[j])
  - Still $1.3 \cdot 10^{-6}$%

    Why?

```
volatile int flag[2];
volatile int victim;

void lock() {
  int j = 1 - tid;
  flag[tid] = 1; // I’m interested
  victim = tid; // other goes first
  asm(“mfence”);
  while (flag[j] && victim == tid) {}; // wait
}

void unlock() {
  flag[tid] = 0; // I’m not interested
}
```

Peterson in Practice … on x86

- **Implement and run our little counter on x86**
- **Many iterations**
  - $1.6 \cdot 10^{-6}$% errors
  - What is the problem?

    No sequential consistency for W(v) and R(flag[j])
  - Still $1.3 \cdot 10^{-6}$%

    Why?

    Reads may slip into CR!

```
volatile int flag[2];
volatile int victim;

void lock() {
  int j = 1 - tid;
  flag[tid] = 1; // I’m interested
  victim = tid; // other goes first
  asm(“mfence”);
  while (flag[j] && victim == tid) {}; // wait
}

void unlock() {
  asm(“mfence”);
  flag[tid] = 0; // I’m not interested
}
```

Peterson in Practice … on x86

- **Implement and run our little counter on x86**
- **Many iterations**
  - $1.6 \cdot 10^{-6}$% errors
  - What is the problem?

    No sequential consistency for W(v) and R(flag[j])
  - Still $1.3 \cdot 10^{-6}$%

    Why?

    Reads may slip into CR!

```
volatile int flag[2];
volatile int victim;

void lock() {
  int j = 1 - tid;
  flag[tid] = 1; // I’m interested
  victim = tid; // other goes first
  asm(“mfence”);
  while (flag[j] && victim == tid) {}; // wait
}

void unlock() {
  asm(“mfence”);
  flag[tid] = 0; // I’m not interested
}
```

The compiler may inline this function ☺

Correct Peterson Lock on x86

- **Unoptimized (naïve sprinkling of mfences)**
- **Performance:**
  - No mfence

    375ns
  - mfence in lock

    379ns
  - mfence in unlock

    404ns
  - Two mfence

    427ns (+14%)

```
volatile int flag[2];
volatile int victim;

void lock() {
  int j = 1 - tid;
  flag[tid] = 1; // I’m interested
  victim = tid; // other goes first
  asm(“mfence”);
  while (flag[j] && victim == tid) {}; // wait
}

void unlock() {
  asm(“mfence”);
  flag[tid] = 0; // I’m not interested
}
```

Hardware Support?

- **Hardware atomic operations:**
  - Test&Set

    Write const to memory while returning the old value
  - Atomic swap

    Atomically exchange memory and register
  - Fetch&Op

    Get value and apply operation to memory location
  - Compare&Swap

    Compare two values and swap memory with register if equal
  - Load-linked/Store-Conditional LL/SC

    Loads value from memory, allows operations, commits only if no other updates committed → mini-TM
  - Intel TSX (transactional synchronization extensions)

    Hardware-TM (roll your own atomic operations)

Relative Power of Synchronization

- **Design-Problem I: Multi-core Processor**
  - Which atomic operations are useful?
- **Design-Problem II: Complex Application**
  - What atomic should I use?
- **Concept of "consensus number" C if a primitive can be used to solve the "consensus problem" in a finite number of steps (even if threads stop)**
  - atomic registers have C=1 (thus locks have C=1!)
  - TAS, Swap, Fetch&Op have C=2
  - CAS, LL/SC, TM have C=∞

Test-and-Set Locks

- **Test-and-Set semantics**
  - Memoize old value
  - Set fixed value TASval (true)
  - Return old value
- **After execution:**
  - Post-condition is a fixed (constant) value!

```
bool TestAndSet (bool *flag) {
  bool old = *flag;
  *flag = true;
  return old;
} // all atomic!
```

Test-and-Set Locks

- Assume TASval indicates “locked”
- Write something else to indicate “unlocked”
- TAS until return value is != TASval (1 in this example)

- When will the lock be granted?
- Does this work well in practice?

```
volatile int lck = 0;

void lock() {
  while (TestAndSet(&lck) == 1);
}

void unlock() {
  lck = 0;
}
```

Cacheline contention (or: why I told you about MESI and friends)

- **On x86, the XCHG instruction is used to implement TAS**
  - x86 lock is implicit in xchg!
- **Cacheline is read and written**
  - Ends up in exclusive state, invalidates other copies
  - Cacheline is “thrown” around uselessly
  - High load on memory subsystem

  x86 lock is essentially a full memory barrier ☹

```
movl   $1, %eax
xchg   %eax, (%ebx)
```

Test-and-Test-and-Set (TATAS) Locks

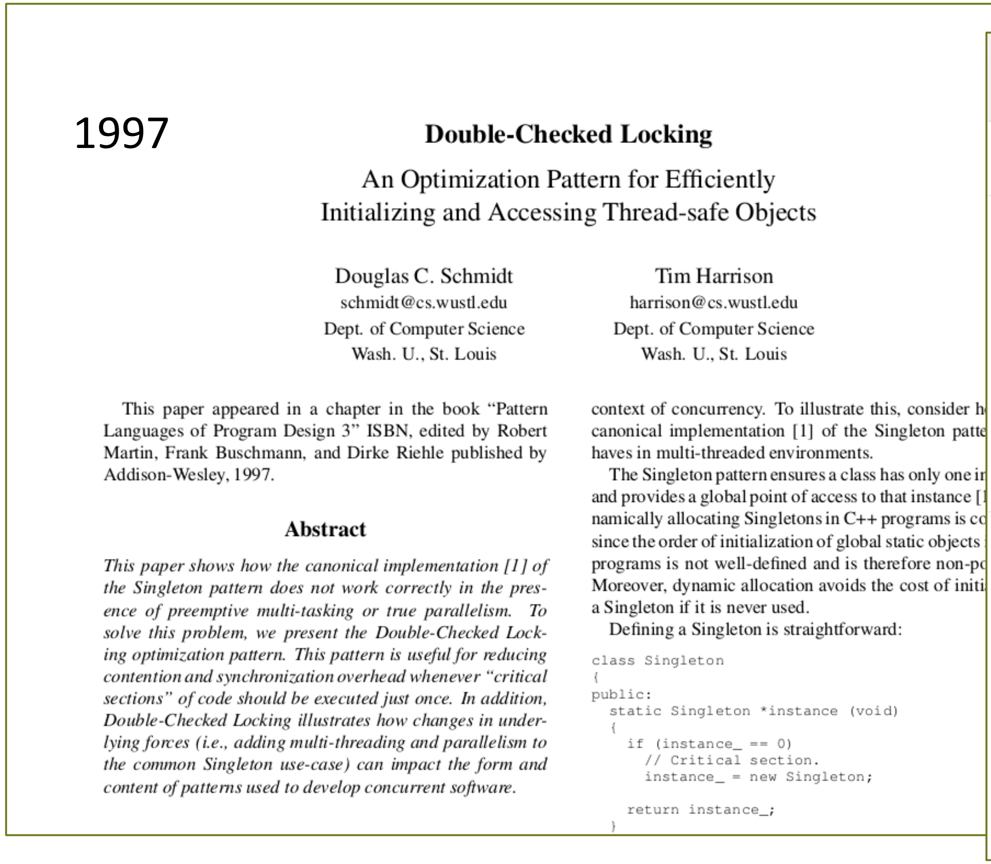
- **Spinning in TAS is not a good idea**
- **Spin on cache line in shared state**
  - All threads at the same time, no cache coherency/memory traffic

- **Danger!**
  - Efficient but use with great care!
  - Generalizations are very dangerous

```
volatile int lck = 0;

void lock() {
  do {
     while (lck == 1);
  } while (TestAndSet(&lck) == 1);
}

void unlock() {
  lck = 0;
}
```

Warning: Even Experts get it wrong!

- **Example: Double-Checked Locking**



1997

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

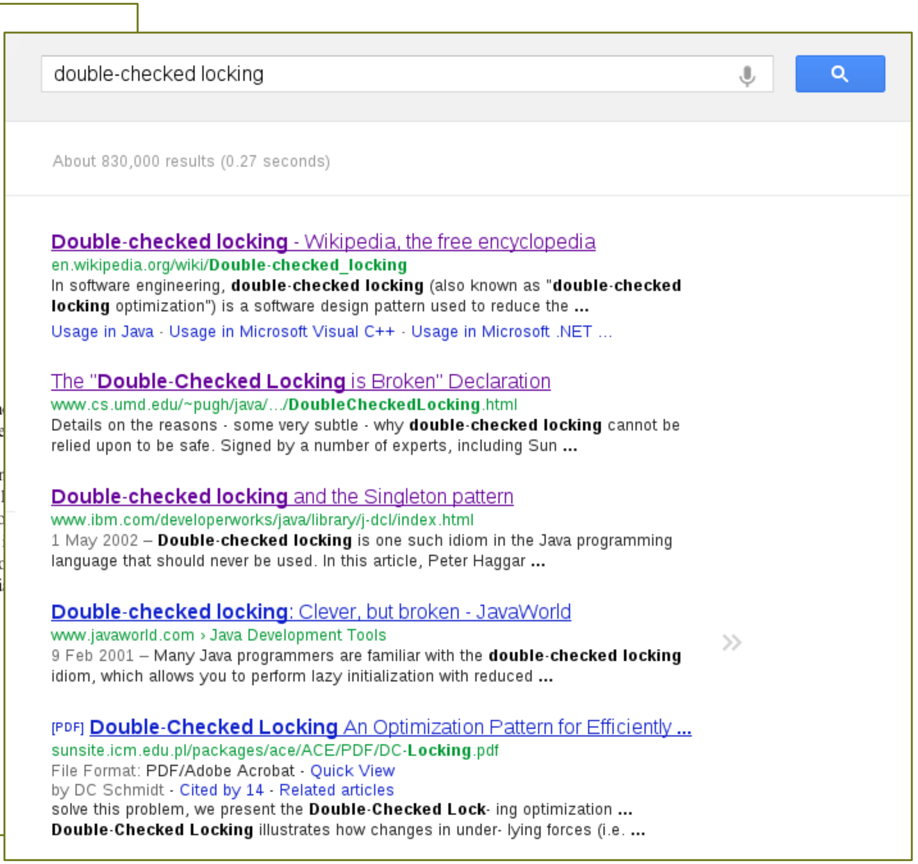
This paper shows how the canonical implementation [1] of the Singleton pattern does not work correctly in the presence of preemptive multi-tasking or true parallelism. To solve this problem, we present the Double-Checked Locking optimization pattern. This pattern is useful for reducing contention and 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 h canonical implementation [1] of the Singleton patte haves in multi-threaded environments.

The Singleton pattern ensures a class has only one i and provides a global point of access to that instance [1 namically allocating Singletons in C++ programs is co since the order of initialization of global static objects programs is not well-defined and is therefore non-po Moreover, dynamic allocation avoids the cost of initi 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!

Contention?

- **Do TATAS locks still have contention?**

```
volatile int lck = 0;

void lock() {
  do {
    while (lck == 1);
  } while (TestAndSet(&lck) == 1);
}

void unlock() {
  lck = 0;
}
```

Contention?

- **Do TATAS locks still have contention?**
- **When lock is released, k threads fight for cache line ownership**
  - One gets the lock, all get the CL exclusively (serially!)
  - What would be a good solution? (think “collision avoidance”)

```
volatile int lck = 0;

void lock() {
  do {
    while (lck == 1);
  } while (TestAndSet(&lck) == 1);
}

void unlock() {
  lck = 0;
}
```

TAS Lock with Exponential Backoff

- **Exponential backoff eliminates contention statistically**
  - Locks granted in unpredictable order
  - Starvation possible but unlikely

    How can we make it even less likely?

```
volatile int lck = 0;

void lock() {
  while (TestAndSet(&lck) == 1) {
    wait(time);
    time *= 2; // double waiting time
  }
}

void unlock() {
  lck = 0;
}
```

Similar to: T. Anderson: “The performance of spin lock alternatives for shared-memory multiprocessors”, TPDS, Vol. 1 Issue 1, Jan 1990

TAS Lock with Exponential Backoff

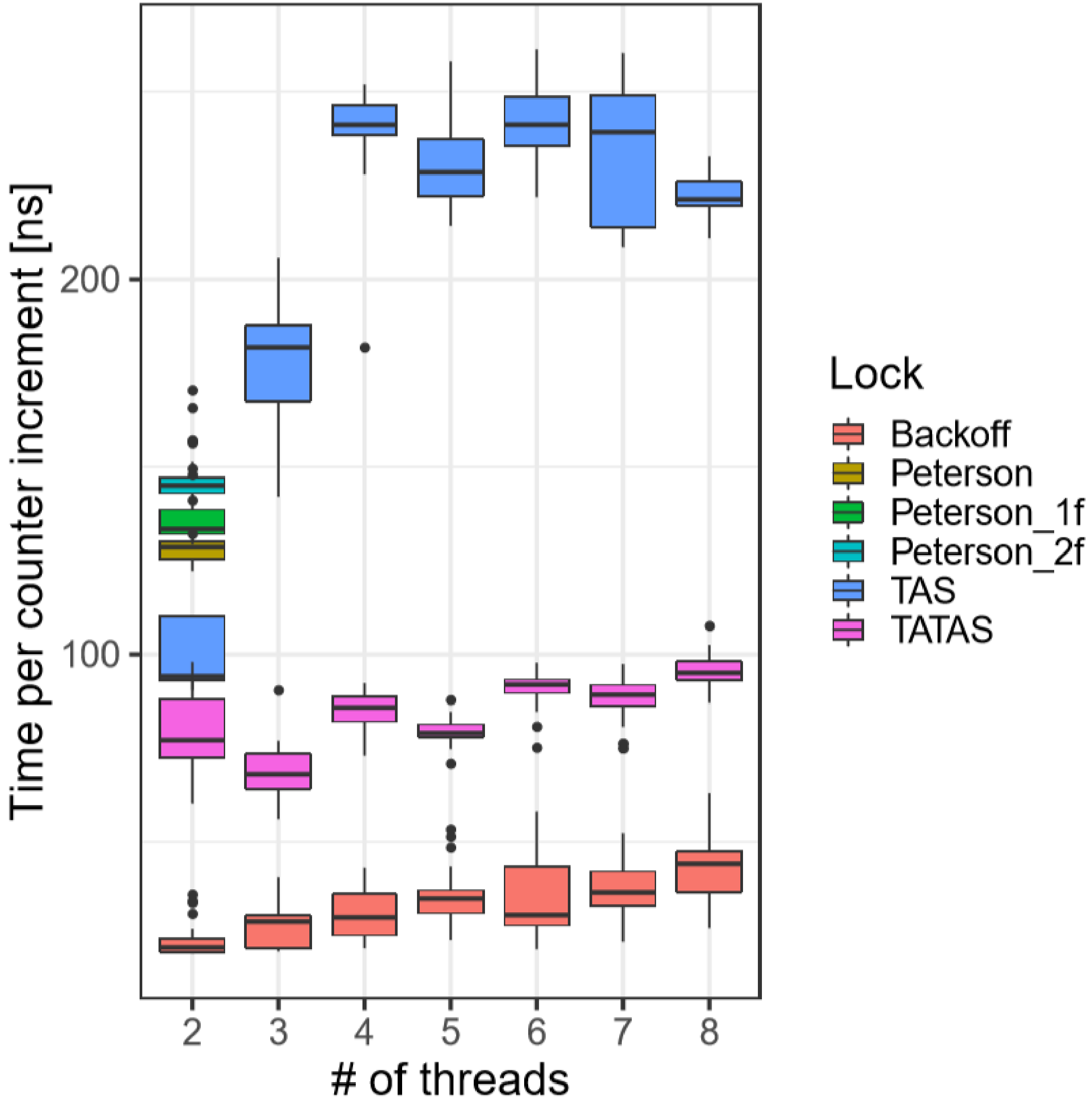
- **Exponential backoff eliminates contention statistically**
  - Locks granted in unpredictable order
  - Starvation possible but unlikely

    Maximum waiting time makes it less likely

```
volatile int lck = 0;
const int maxtime=1000;

void lock() {
  while (TestAndSet(&lck) == 1) {
    wait(time);
    time = min(time * 2, maxtime);
  }
}

void unlock() {
  lck = 0;
}
```

Similar to: T. Anderson: “The performance of spin lock alternatives for shared-memory multiprocessors”, TPDS, Vol. 1 Issue 1, Jan 1990

Comparison of TAS Locks



Improvements?

- **Are TAS locks perfect?**
  - What are the two biggest issues?

Improvements?

- **Are TAS locks perfect?**
  - What are the two biggest issues?
    - Cache coherency traffic (contending on same location with expensive atomics)

      -- or --

    - Critical section underutilization (waiting for backoff times will delay entry to CR)
- **What would be a fix for that?**
  - How is this solved at airports and shops (often at least)?

Improvements?

- **Are TAS locks perfect?**
  - What are the two biggest issues?
    - Cache coherency traffic (contending on same location with expensive atomics)

      -- or --
    - Critical section underutilization (waiting for backoff times will delay entry to CR)
- **What would be a fix for that?**
  - How is this solved at airports and shops (often at least)?
- **Queue locks -- Threads enqueue**
  - Learn from predecessor if it’s their turn
  - Each threads spins at a different location
  - FIFO fairness

Array Queue Lock

- **Array to implement queue**
  - Tail-pointer shows next free queue position
  - Each thread spins on own location

    CL padding!
  - index[] array can be put in TLS

```
volatile int array[n] = {1,0,…,0};
volatile int index[n] = {0,0,…,0};
volatile int tail = 0;

void lock() {
  index[tid] = GetAndInc(tail) % n;
  while (!array[index[tid]]); // wait to receive lock
}

void unlock() {
  array[index[tid]] = 0; // I release my lock
  array[(index[tid] + 1) % n] = 1; // next one
}
```

Array Queue Lock

- **Array to implement queue**
  - Tail-pointer shows next free queue position
  - Each thread spins on own location

    CL padding!
  - index[] array can be put in TLS
- **So are we done now?**
  - What’s wrong?

```
volatile int array[n] = {1,0,…,0};
volatile int index[n] = {0,0,…,0};
volatile int tail = 0;

void lock() {
  index[tid] = GetAndInc(tail) % n;
  while (!array[index[tid]]); // wait to receive lock
}

void unlock() {
  array[index[tid]] = 0; // I release my lock
  array[(index[tid] + 1) % n] = 1; // next one
}
```

Array Queue Lock

- **Array to implement queue**
  - Tail-pointer shows next free queue position
  - Each thread spins on own location

    CL padding!
  - index[] array can be put in TLS
- **So are we done now?**
  - What’s wrong?
  - Synchronizing M objects requires Θ(NM) storage
  - What do we do now?

```
volatile int array[n] = {1,0,…,0};
volatile int index[n] = {0,0,…,0};
volatile int tail = 0;

void lock() {
  index[tid] = GetAndInc(tail) % n;
  while (!array[index[tid]]); // wait to receive lock
}

void unlock() {
  array[index[tid]] = 0; // I release my lock
  array[(index[tid] + 1) % n] = 1; // next one
}
```

CLH Lock (1993)

- **List-based (same queue principle)**
- **Discovered twice by Craig, Landin, Hagersten 1993/94**
- **2N+3M words**
  - N threads, M locks
- **Requires thread-local qnode pointer**
  - Can be hidden!

```
typedef struct qnode {
  struct qnode *prev;
  int succ_blocked;
} qnode;

qnode *lck = new qnode; // node owned by lock

void lock(qnode *lck, qnode *qn) {
  qn->succ_blocked = 1;
  qn->prev = FetchAndSet(lck, qn);
  while (qn->prev->succ_blocked);
}

void unlock(qnode **qn) {
  qnode *pred = (*qn)->prev;
  (*qn)->succ_blocked = 0;
  *qn = pred;
}
```

CLH Lock (1993)

- **Qnode objects represent thread state!**
  - succ_blocked == 1 if waiting or acquired lock
  - succ_blocked == 0 if released lock
- **List is implicit!**
  - One node per thread
  - Spin location changes

    NUMA issues (cacheless)
- **Can we do better?**

```
typedef struct qnode {
  struct qnode *prev;
  int succ_blocked;
} qnode;

qnode *lck = new qnode; // node owned by lock

void lock(qnode *lck, qnode *qn) {
  qn->succ_blocked = 1;
  qn->prev = FetchAndSet(lck, qn);
  while (qn->prev->succ_blocked);
}

void unlock(qnode **qn) {
  qnode *pred = (*qn)->prev;
  (*qn)->succ_blocked = 0;
  *qn = pred;
}
```

MCS Lock (1991)

- **Make queue explicit**
  - Acquire lock by appending to queue
  - Spin on own node until locked is reset
- **Similar advantages as CLH but**
  - Only 2N + M words
  - Spinning position is fixed!
    Benefits cache-less NUMA
- **What are the issues?**

```
typedef struct qnode {
  struct qnode *next;
  int succ_blocked;
} qnode;

qnode *lck = NULL;

void lock(qnode *lck, qnode *qn) {
  qn->next = NULL;
  qnode *pred = FetchAndSet(lck, qn);
  if(pred != NULL) {
    qn->locked = 1;
    pred->next = qn;
    while(qn->locked);
} }

void unlock(qnode * lck, qnode *qn) {
  if(qn->next == NULL) { // if we're the last waiter
    if(CAS(lck, qn, NULL)) return;
    while(qn->next == NULL); // wait for pred arrival
  }
  qn->next->locked = 0; // free next waiter
  qn->next = NULL;
}
```

MCS Lock (1991)

- **Make queue explicit**
  - Acquire lock by appending to queue
  - Spin on own node until locked is reset
- **Similar advantages as CLH but**
  - Only 2N + M words
  - Spinning position is fixed!
    Benefits cache-less NUMA
- **What are the issues?**
  - Releasing lock spins
  - More atomics!

```
typedef struct qnode {
  struct qnode *next;
  int succ_blocked;
} qnode;

qnode *lck = NULL;

void lock(qnode *lck, qnode *qn) {
  qn->next = NULL;
  qnode *pred = FetchAndSet(lck, qn);
  if(pred != NULL) {
    qn->locked = 1;
    pred->next = qn;
    while(qn->locked);
} }

void unlock(qnode * lck, qnode *qn) {
  if(qn->next == NULL) { // if we're the last waiter
    if(CAS(lck, qn, NULL)) return;
    while(qn->next == NULL); // wait for pred arrival
  }
  qn->next->locked = 0; // free next waiter
  qn->next = NULL;
}
```

Lessons Learned!

- **Key Lesson:**
  - Reducing memory (coherency) traffic is most important!
  - Not always straight-forward (need to reason about CL states)

- **MCS: 2006 Dijkstra Prize in distributed computing**
  - *“an outstanding paper on the principles of distributed computing, whose significance and impact on the theory and/or practice of distributed computing has been evident for at least a decade”*
  - *“probably the most influential practical mutual exclusion algorithm ever”*
  - *“vastly superior to all previous mutual exclusion algorithms”*
  - fast, fair, scalable → widely used, always compared against!

Time to Declare Victory?

- **Down to memory complexity of 2N+M**
  - Probably close to optimal
- **Only local spinning**
  - Several variants with low expected contention
- **But: we assumed sequential consistency ☹**
  - Reality causes trouble sometimes
  - Sprinkling memory fences may harm performance
  - Open research on minimally-synching algorithms!

    Come and talk to me if you’re interested

Fighting CPU waste: Condition Variables

- **Allow threads to yield CPU and leave the OS run queue**
  - Other threads can get them back on the queue!
- **cond_wait(cond, lock) – yield and go to sleep**
- **cond_signal(cond) – wake up sleeping threads**
- **Wait and signal are OS calls**
  - Often expensive, which one is more expensive?

Fighting CPU waste: Condition Variables

- **Allow threads to yield CPU and leave the OS run queue**
  - Other threads can get them back on the queue!
- **cond_wait(cond, lock) – yield and go to sleep**
- **cond_signal(cond) – wake up sleeping threads**
- **Wait and signal are OS calls**
  - Often expensive, which one is more expensive?

    Wait, because it has to perform a full context switch

When to Spin and When to Block?

- **Spinning consumes CPU cycles but is cheap**
  - “Steals” CPU from other threads
- **Blocking has high one-time cost and is then free**
  - Often hundreds of cycles (trap, save TCB …)
  - Wakeup is also expensive (latency)

    Also cache-pollution
- **Strategy:**
  - Poll for a while and then block

    But what is a “while”??

When to Spin and When to Block?

When to Spin and When to Block?

- **Optimal time depends on the future**
  - When will the active thread leave the CR?
  - Can compute optimal offline schedule

    Q: What is the optimal offline schedule (assuming we know the future, i.e., when the lock will become available)?

When to Spin and When to Block?

- **Optimal time depends on the future**
  - When will the active thread leave the CR?
  - Can compute optimal offline schedule

    Q: What is the optimal offline schedule (assuming we know the future, i.e., when the lock will become available)?
  - Actual problem is an online problem
- **Competitive algorithms**
  - An algorithm is c-competitive if for a sequence of actions x and a constant a holds:

    $C(x) \leq c * C_{opt}(x) + a$
  - What would a good spinning algorithm look like and what is the competitiveness?

Competitive Spinning

- **If T is the overhead to process a wait, then a locking algorithm that spins for time T before it blocks is 2-competitive!**
  - Karlin, Manasse, McGeoch, Owicki: “Competitive Randomized Algorithms for Non-Uniform Problems”, SODA 1989
- **If randomized algorithms are used, then e/(e-1)-competitiveness (~1.58) can be achieved**
  - See paper above!

Remember: lock-free vs. wait-free

- **A lock-free method**
  - guarantees that infinitely often **some** method call finishes in a finite number of steps
- **A wait-free method**
  - guarantees that **each** method call finishes in a finite number of steps (implies lock-free)

- **Synchronization instructions are not equally powerful!**
  - Indeed, they form an infinite hierarchy; no instruction (primitive) in level x can be used for lock-/wait-free implementations of primitives in level z>x.

Concept: Consensus Number

- **Each level of the hierarchy has a “consensus number” assigned.**
  - Is the maximum number of threads for which primitives in level x can solve the consensus problem
- **The consensus problem:**
  - Has single function: decide(v)
  - Each thread calls it at most once, the function returns a value that meets two conditions:

    consistency: all threads get the same value

    validity: the value is some thread’s input
  - Simplification: binary consensus (inputs in {0,1})

Understanding Consensus

- **Can a particular class solve n-thread consensus wait-free?**
  - A class C solves n-thread consensus if there exists a consensus protocol using **any number** of objects of class C and **any number** of atomic registers
  - The protocol has to be wait-free (bounded number of steps per thread)
  - The consensus number of a class C is the largest n for which that class solves n-thread consensus (may be infinite)
  - Assume we have a class D whose objects can be constructed from objects out of class C. If class C has consensus number n, what does class D have?

Starting simple …

- **Binary consensus with two threads (A, B)!**
  - Each thread moves until it decides on a value
  - May update shared objects
  - Protocol state = state of threads + state of shared objects
  - Initial state = state before any thread moved
  - Final state = state after all threads finished
  - States form a tree, wait-free property guarantees a finite tree

    Example with two threads and two moves each!

Atomic Registers

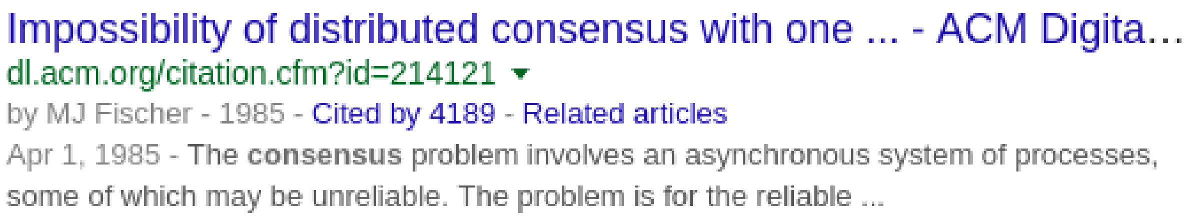
- **Theorem [Herlihy’91]: Atomic registers have consensus number one**
  - I.e., they cannot be used to solve even two-thread consensus! Really?

Atomic Registers

- **Theorem [Herlihy'91]: Atomic registers have consensus number one**
  - I.e., they cannot be used to solve even two-thread consensus! Really?
- **Proof outline:**
  - Assume arbitrary consensus protocol, thread A, B
  - Run until it reaches critical state where next action determines outcome (show that it must have a critical state first)
  - Show all options using atomic registers and show that they cannot be used to determine one outcome for all possible executions!
    1) *Any thread reads (other thread runs solo until end)*
    2) *Threads write to different registers (order doesn't matter)*
    3) *Threads write to same register (solo thread can start after each write)*

Atomic Registers

- **Theorem [Herlihy'91]: Atomic registers have consensus number one**
- **Corollary: It is impossible to construct a wait-free implementation of any object with consensus number of >1 using atomic registers**
  - "perhaps one of the most striking impossibility results in Computer Science" (Herlihy, Shavit)
  - → We need hardware atomics or Transactional Memory!
- **Proof technique borrowed from:**



Impossibility of distributed consensus with one ... - ACM Digita…
dl.acm.org/citation.cfm?id=214121 ▾
by MJ Fischer - 1985 - Cited by 4189 - Related articles
Apr 1, 1985 - The **consensus** problem involves an asynchronous system of processes, some of which may be unreliable. The problem is for the reliable ...

- **Very influential paper, always worth a read!**
  - Nicely shows proof techniques that are central to parallel and distributed computing!

Other Atomic Operations

- **Simple RMW operations (Test&Set, Fetch&Op, Swap, basically all functions where the op commutes or overwrites) have consensus number 2!**
  - Similar proof technique (bivalence argument)
- **CAS and TM have consensus number ∞**
  - Constructive proof!

Compare and Set/Swap Consensus

```
const int first = -1
volatile int thread = -1;
int proposed[n];

int decide(v) {
  proposed[tid] = v;
  if(CAS(thread, first, tid))
    return  v; // I won!
  else
     return proposed[thread]; // thread won
}
```

- **CAS provides an infinite consensus number**
  - Machines providing CAS are **asynchronous** computation equivalents of the Turing Machine
  - I.e., any concurrent object can be implemented in a wait-free manner (not necessarily fast!)

Now you know everything ☺

- **Not really … ;-)**
  - We’ll argue more about **performance** now!
- **But you have all the tools for:**
  - Efficient locks
  - Efficient lock-based algorithms
  - Efficient lock-free algorithms (or even wait-free)
  - Reasoning about parallelism!
- **What now?**
  - A different class of problems

    Impact on wait-free/lock-free on actual performance is not well understood
  - Relevant to HPC, applies to shared and distributed memory

    → *Group communications*