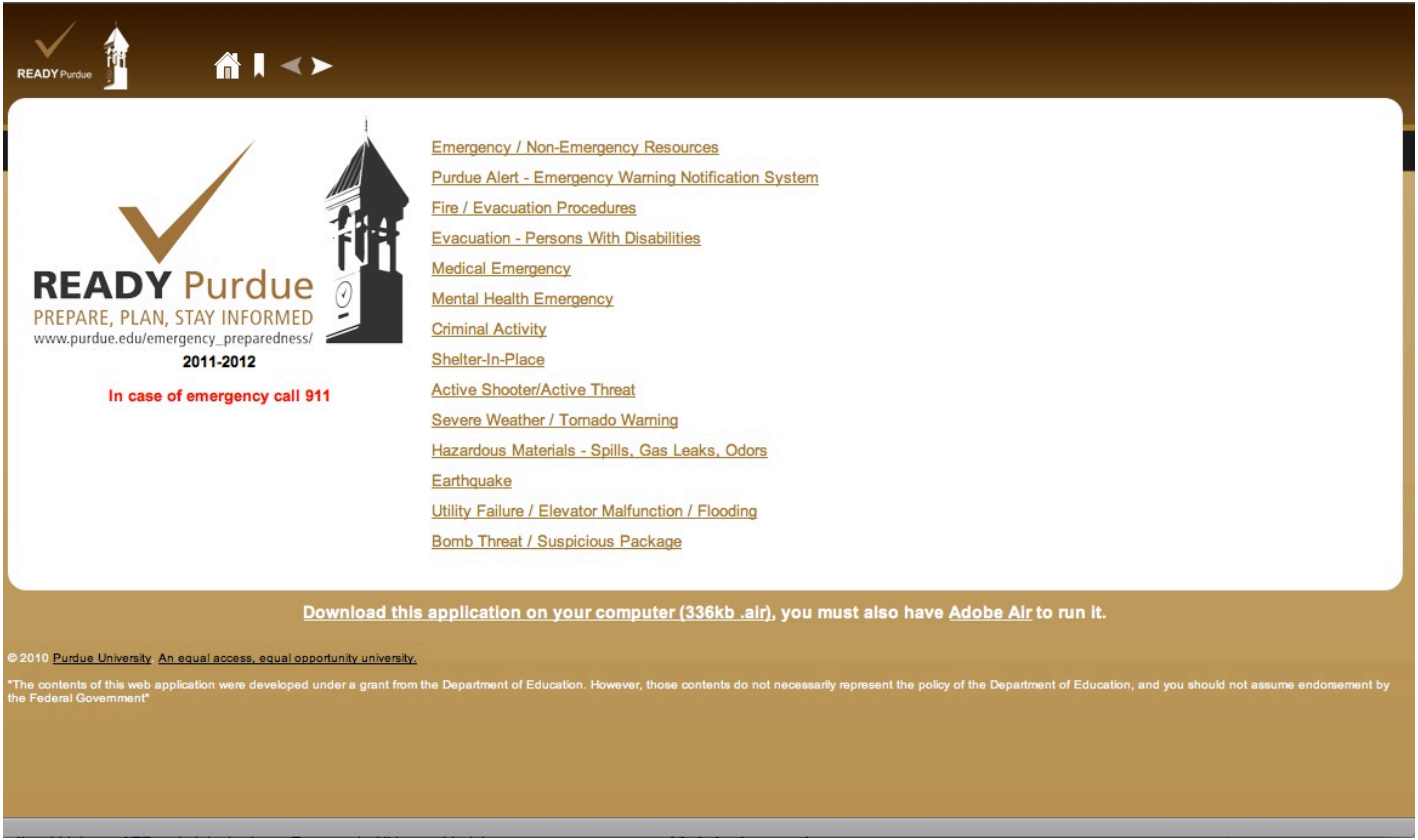


http://www.purdue.edu/emergency_preparedness/flipchart/

counseling available at http://www.purdue.edu/caps/

Three issues with shared memory programming

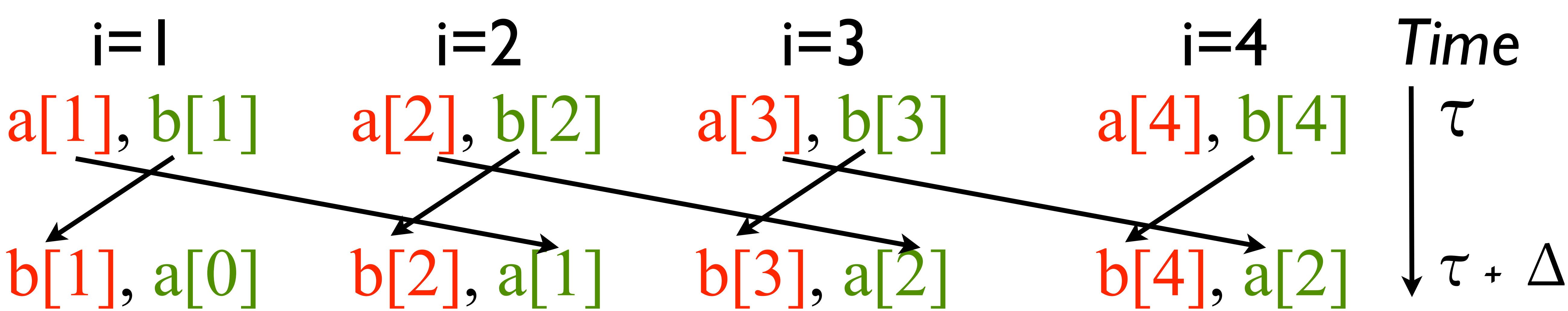
- We would like to execute programs whose different "parts" share data
- This raises three issues
    1. If operations are not commutative they must execute them in the order specified by the program
    2. If operations are commutative they can execute in any order as long as one operation appears to execute after the other has finished
    3. Loads of variable values from memory must get the last value written
- We will now discuss these in more detail

Barriers and loop distribution to enforce orders

Consider the program

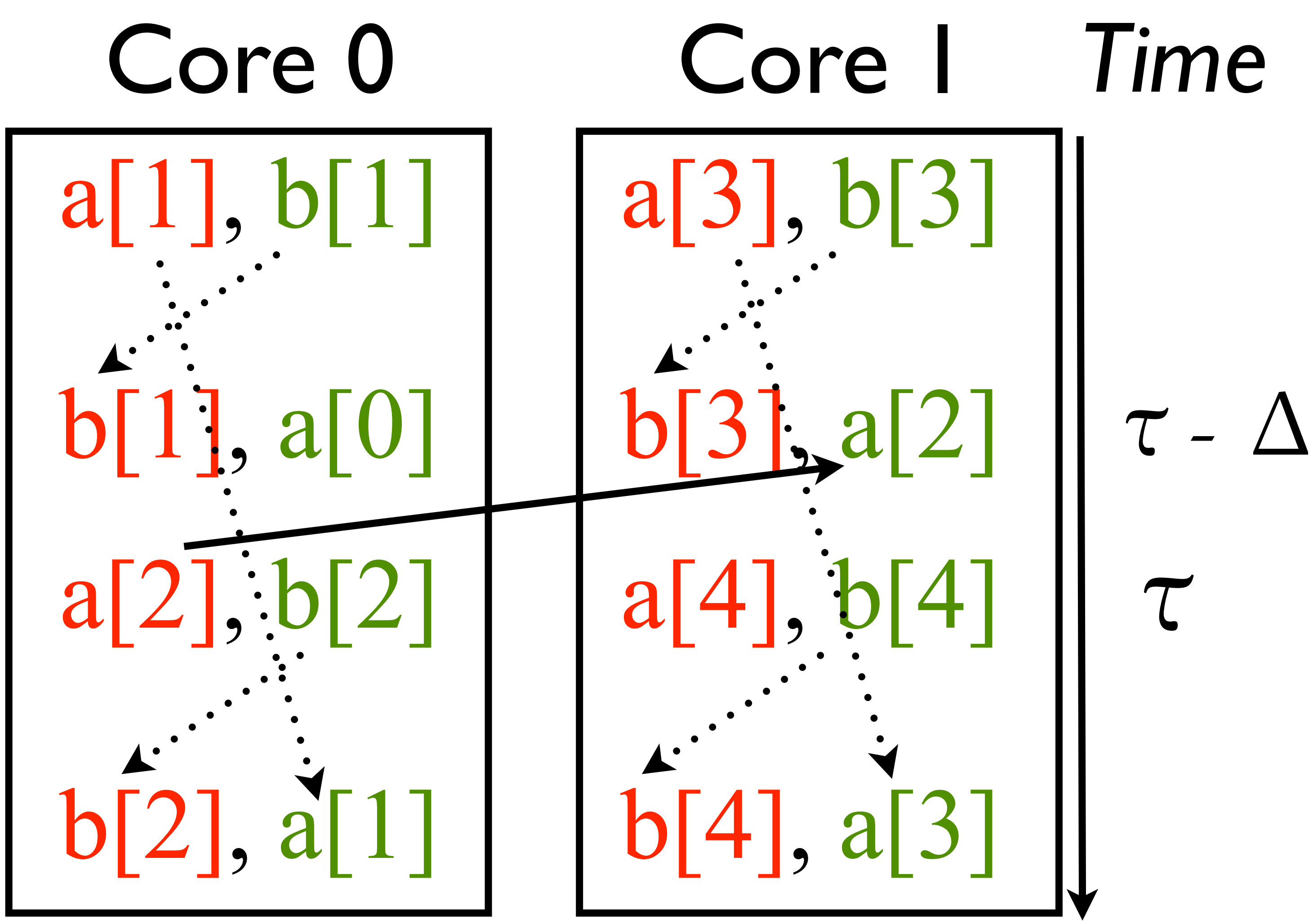
```
for (int i = 1; i < 4; i++) {
   a[i] = b[i] + . . .;
   b[i] = a[i-1]
}
```

Orders that must be honored are shown as arrows



Execute on two cores

```
for (int i = 1; i < 4; i++) {
    a[i] = b[i] + . . .;
    b[i] = a[i-1]
}
```



Dashed orders enforced by sequential execution within a core

a[2] written in Core 0 must travel back in time to reach read of a[2]. Need to force this order

First step
Distribute the loop

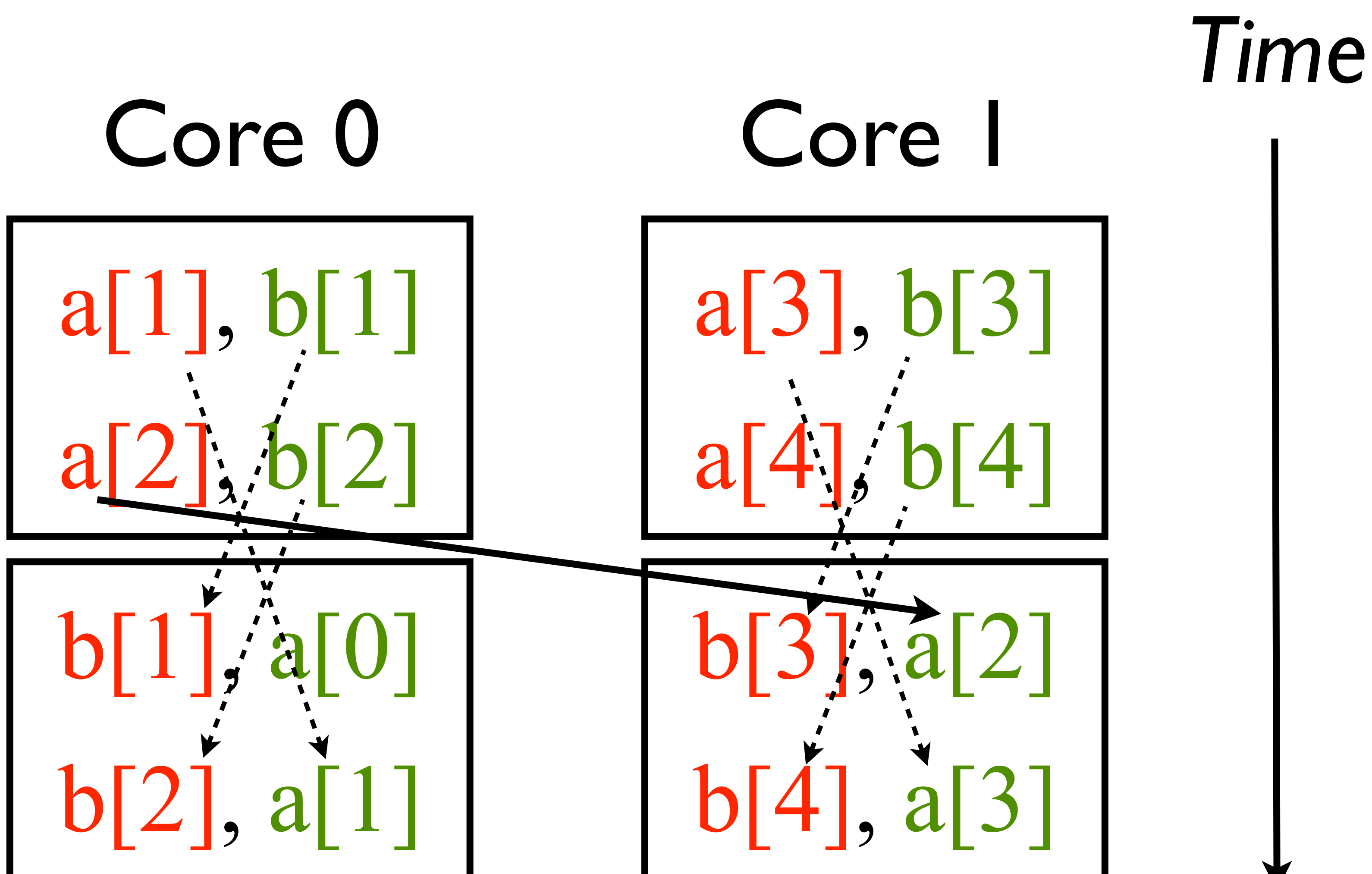
```
for (int i = 1; i < 4; i++) {
   a[i] = b[i] + . . .;
}
for (int i = 1; i < 4; i++) {
   b[i] = a[i-1]
}
```

Distribute the loop over the statements in the loop.

Try running this on two cores

```
for (int i = 1; i < 4; i++) {
   a[i] = b[i] + . . .;
}
for (int i = 1; i < 4; i++) {
   b[i] = a[i-1]
}
```

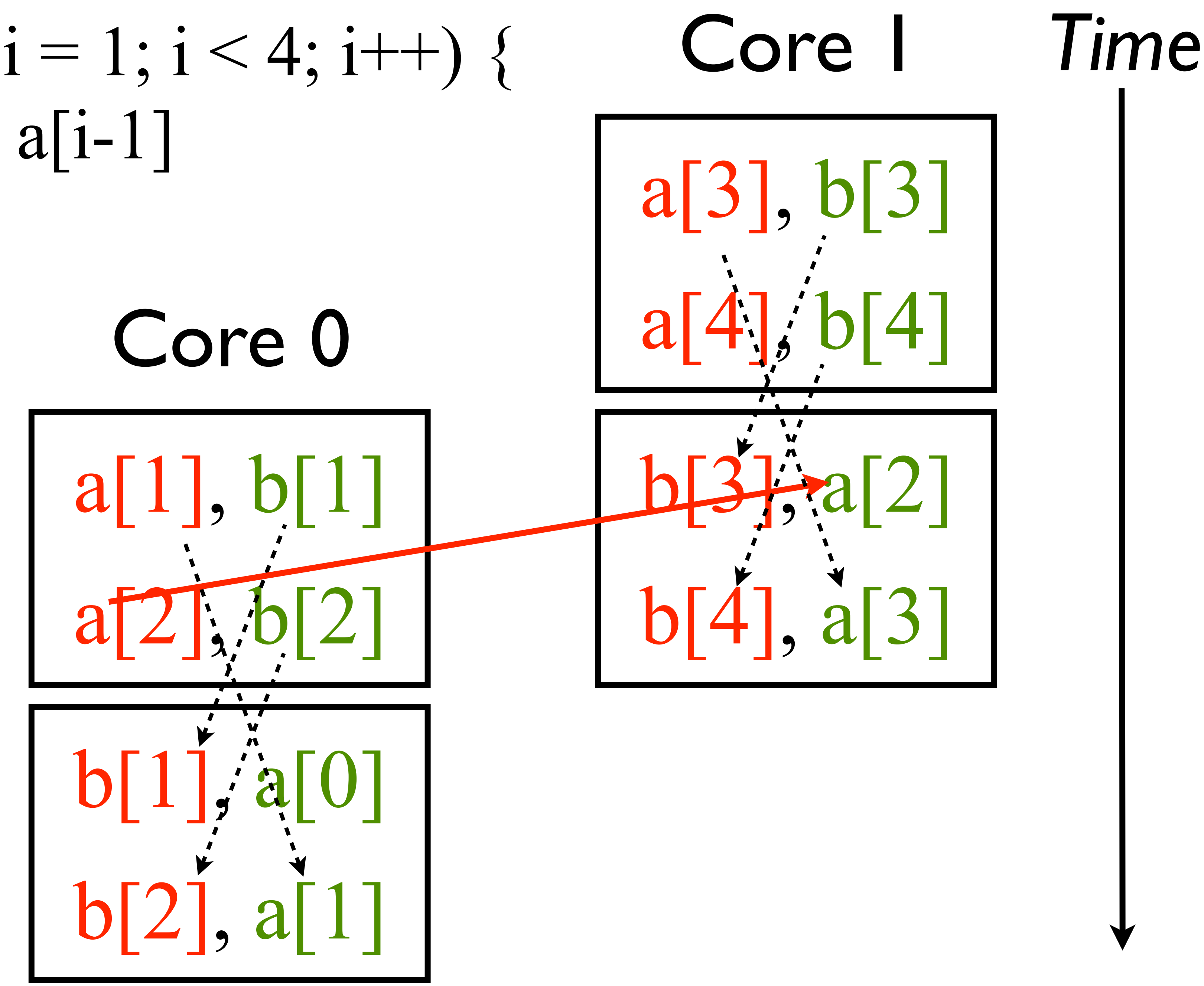
In a perfect world, this happens



Try running this on two cores

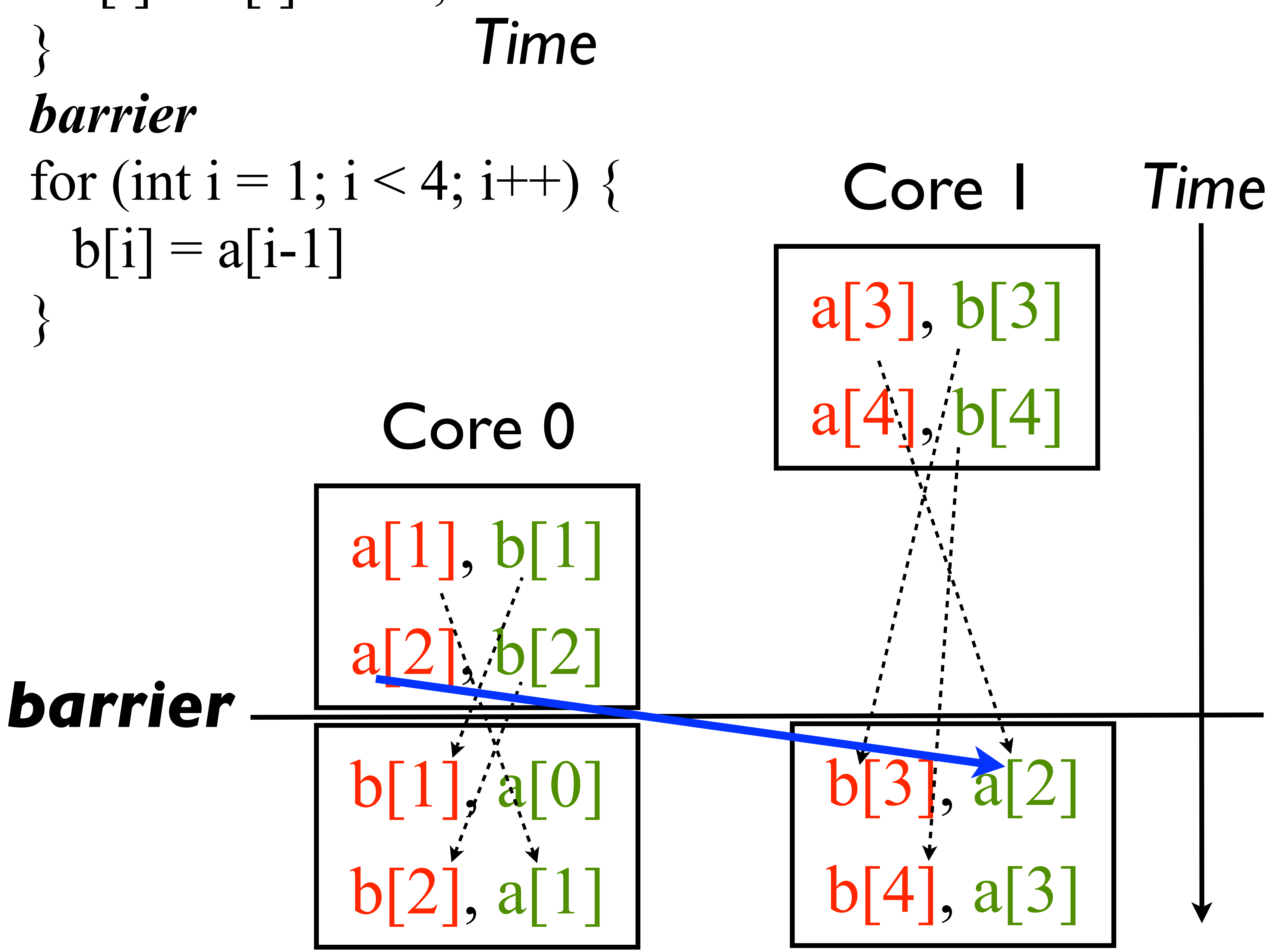
```
for (int i = 1; i < 4; i++) {
  a[i] = b[i] + . . .;
}
for (int i = 1; i < 4; i++) {
  b[i] = a[i-1]
}
```

In the world that exists, this can happen



A *barrier* is needed

for (int i = 1; i < 4; i++) {
  a[i] = b[i] + . . .;
}
barrier
for (int i = 1; i < 4; i++) {
  b[i] = a[i-1]
}



Barrier semantics

- Given a barrier b in each thread T
  - As each thread reaches the barrier, it *enters* it and waits until all threads have entered the barrier
  - When all threads have entered the barrier, all threads can proceed past the barrier
- Dangers in barriers
  - If one or more threads don't enter the barrier, the threads in it are stuck!

Note: not all loops can be distributed!

```
for (int i = 1; i < 4; i++) {
   a[i] = b[i-1] + . . .;
   b[i] = a[i-1]
}
```

```
a[1] = b[0] + . . .;
b[1] = a[0]
a[2] = b[1] + . . .;
b[2] = a[1]
a[3] = b[2] + . . .;
b[3] = a[2]
a[4] = b[3] + . . .;
b[4] = a[3]
```

```
for (int i = 1; i < 4; i++) {
   a[i] = b[i-1] + . . .;
}
for (int i = 1; i < 4; i++) {
   b[i] = a[i-1]
}
```

Illegal

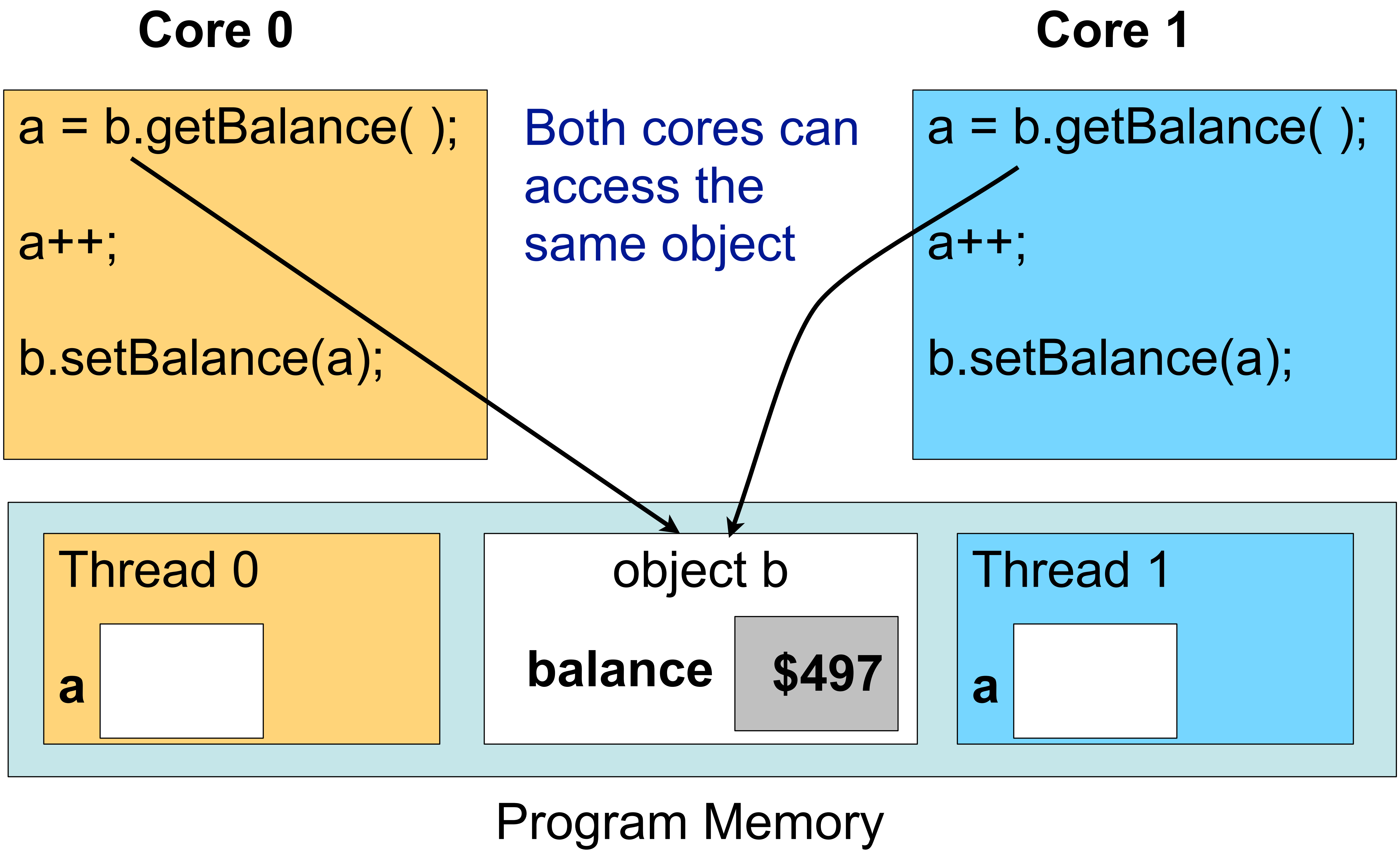
```
a[1] = b[0] + . . .;
a[2] = b[1] + . . .;
a[3] = b[2] + . . .;
a[4] = b[3] + . . .;
b[1] = a[0]
b[2] = a[1]
b[3] = a[2]
b[4] = a[3]
```

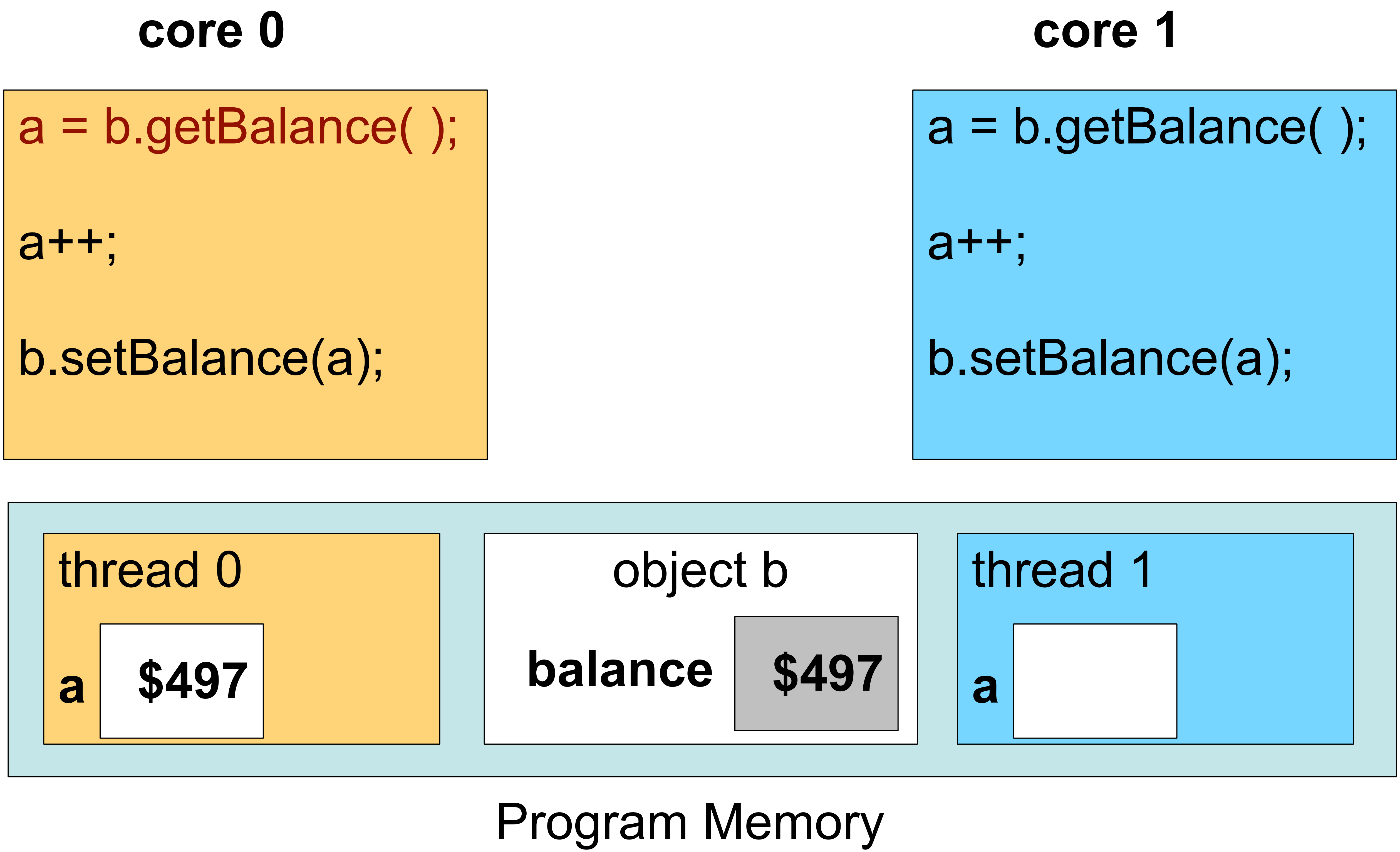
The order of reads and writes to b changes

Atomicity
Useful with commutative operations

- An operation is *atomic* if it executes as if nothing else happens to the state of the memory used by the operation during the execution of the instruction
- Shared memory programming models give *synchronization* instructions that allow us to specify that a collection of operations should be executed atomically
- If operations that can commute are executed atomically, they can be executed in any order and give the correct outcome
- We will now look at an example that shows the importance of atomic execution

Atomicity -- operations execute as though no changes are made to their operands during executions




core 0
core 1
a = b.getBalance( );
a++;
b.setBalance(a);
a = b.getBalance( );
a++;
b.setBalance(a);
thread 0
a
$497
object b
balance
$497
thread 1
a
Program Memory

core 0

```
a = b.getBalance( );

a++;

b.setBalance(a);
```

core 1

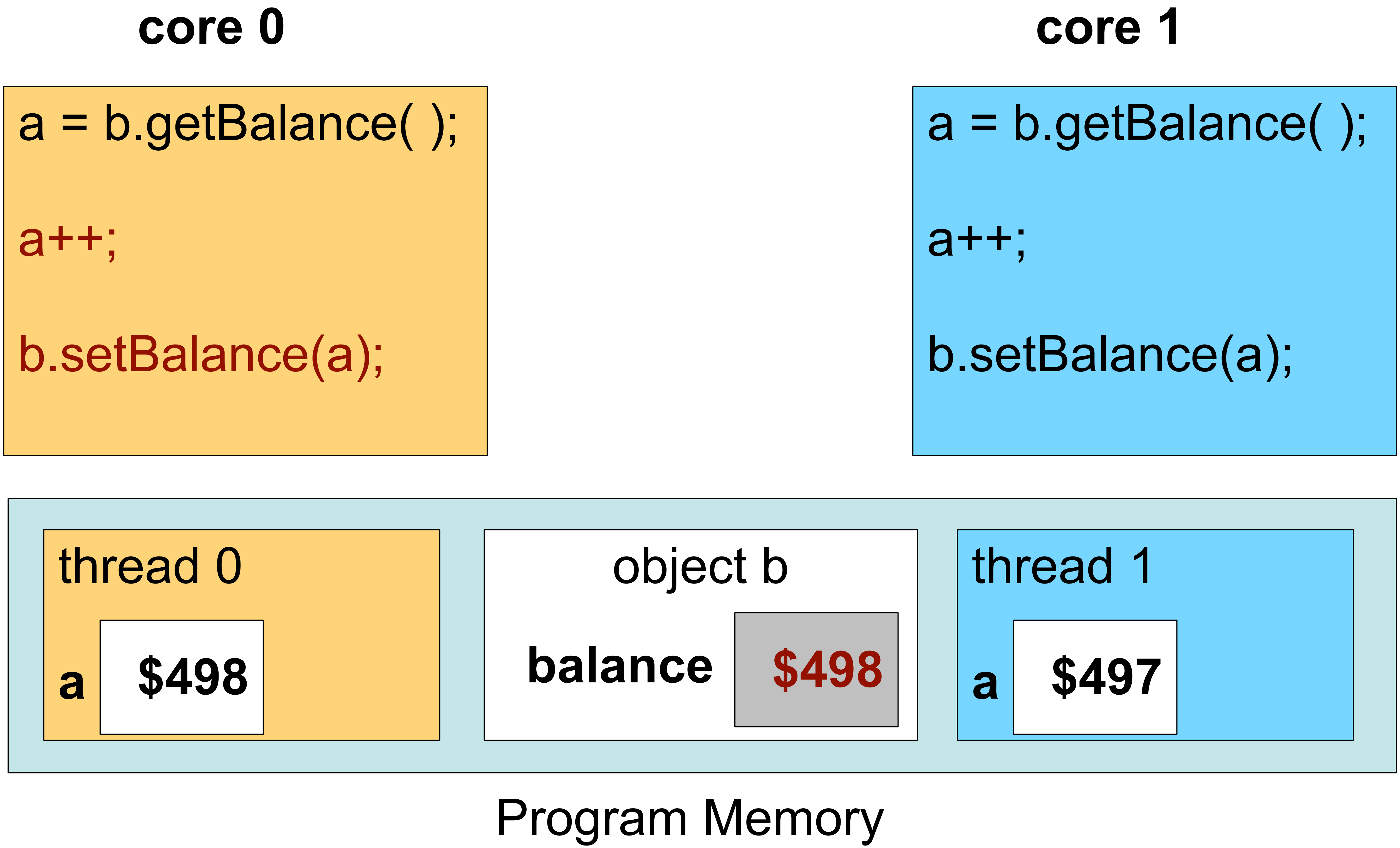
```
a = b.getBalance( );

a++;

b.setBalance(a);
```

thread 0	object b	thread 1
a **$497**	**balance** **$497**	**a** **$497**

Program Memory


core 0
core 1
a = b.getBalance( );
a++;
b.setBalance(a);
a = b.getBalance( );
a++;
b.setBalance(a);
thread 0
a $498
object b
balance $498
thread 1
a $497
Program Memory

core 0

```
a = b.getBalance( );

a++;

b.setBalance(a);
```

The end result probably should have been $499. One update is lost.

core 1

```
a = b.getBalance( );

a++;

b.setBalance(a);
```

thread 0

a **$498**

object b

balance **$498**

thread 1

a **$498**

Program Memory

synchronization enforces atomicity

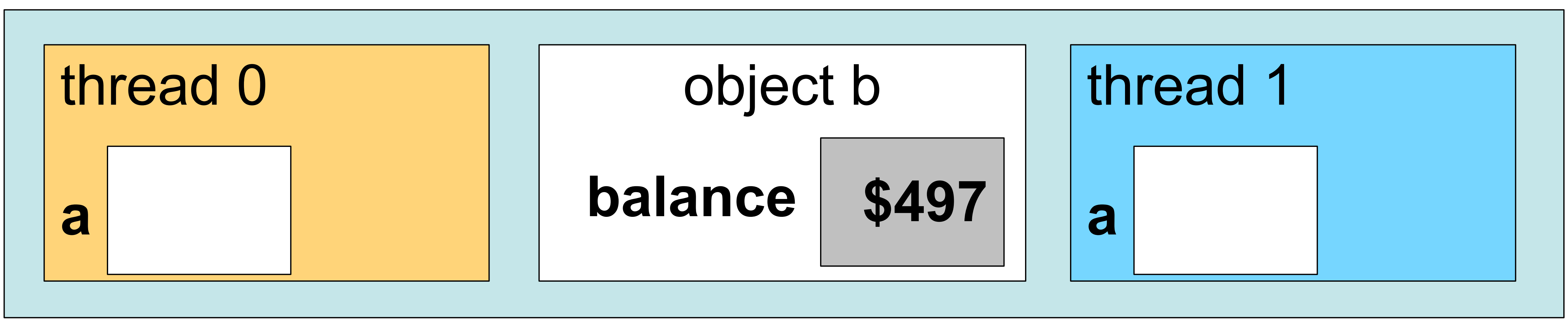
core 0

```
synchronized(b) {
   a = b.getBalance();
   a++;
   b.setBalance(a);
}
```

Make them *atomic* using synchronized

core 1

```
synchronized(b) {
   a = b.getBalance();
   a++;
   b.setBalance(a);
}
```



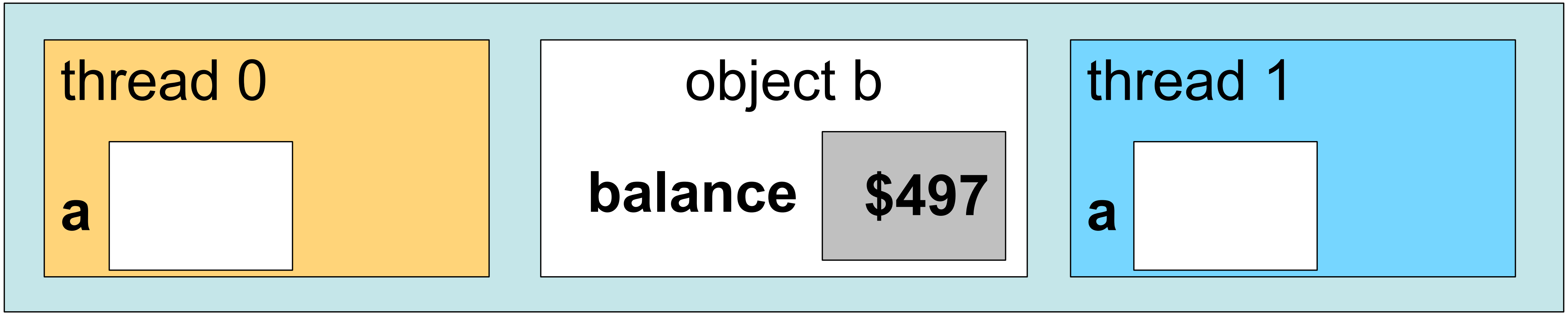
Program Memory

One thread acquires the lock

```
synchronized(b) {
   a = b.getBalance();
   a++;
   b.setBalance(a);
}
```

```
synchronized(b) {
   a = b.getBalance();
   a++;
   b.setBalance(a);
}
```

The other thread waits until the lock is free



One thread acquires the lock

```
synchronized(b) {
   a = b.getBalance();
   a++;
   b.setBalance(a);
}
```

```
synchronized(b) {
   a = b.getBalance();
   a++;
   b.setBalance(a);
}
```

The other thread waits until the lock is free

thread 0

a

object b

balance **$498**

thread 1

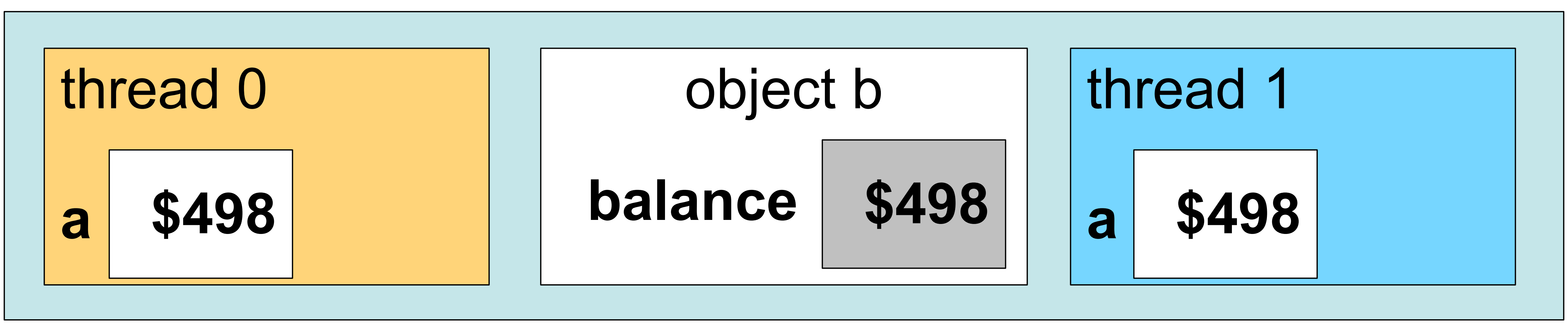
a **$498**

One thread acquires the lock

```
synchronized(b) {
   a = b.getBalance();
   a++;
   b.setBalance(a);
}
```

```
synchronized(b) {
   a = b.getBalance();
   a++;
   b.setBalance(a);
}
```

The other thread waits until the lock is free

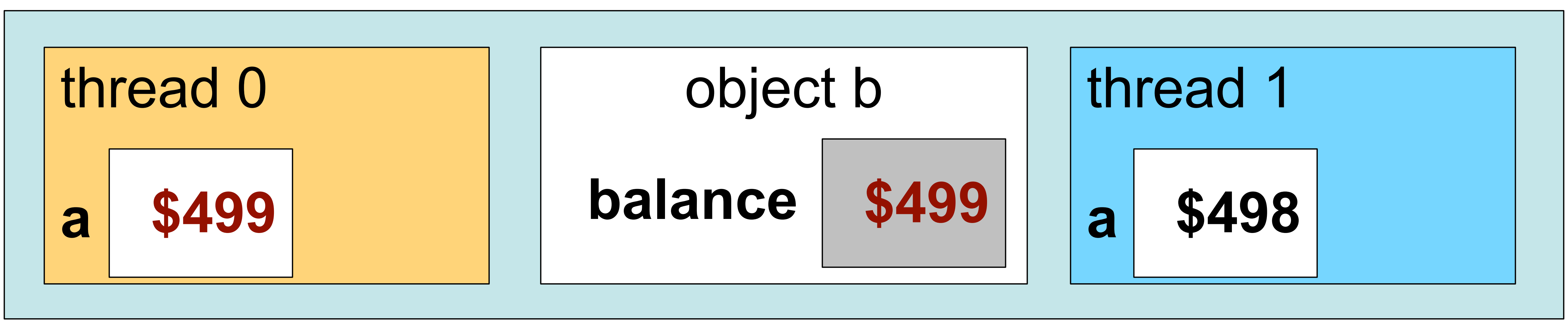


One thread acquires the lock

```
synchronized(b) {
   a = b.getBalance();
   a++;
   b.setBalance(a);
}
```

```
synchronized(b) {
   a = b.getBalance();
   a++;
   b.setBalance(a);
}
```

The other thread waits until the lock is free



Ensuring the last value written is read

A short architectural overview

Warning: gross simplifications to follow

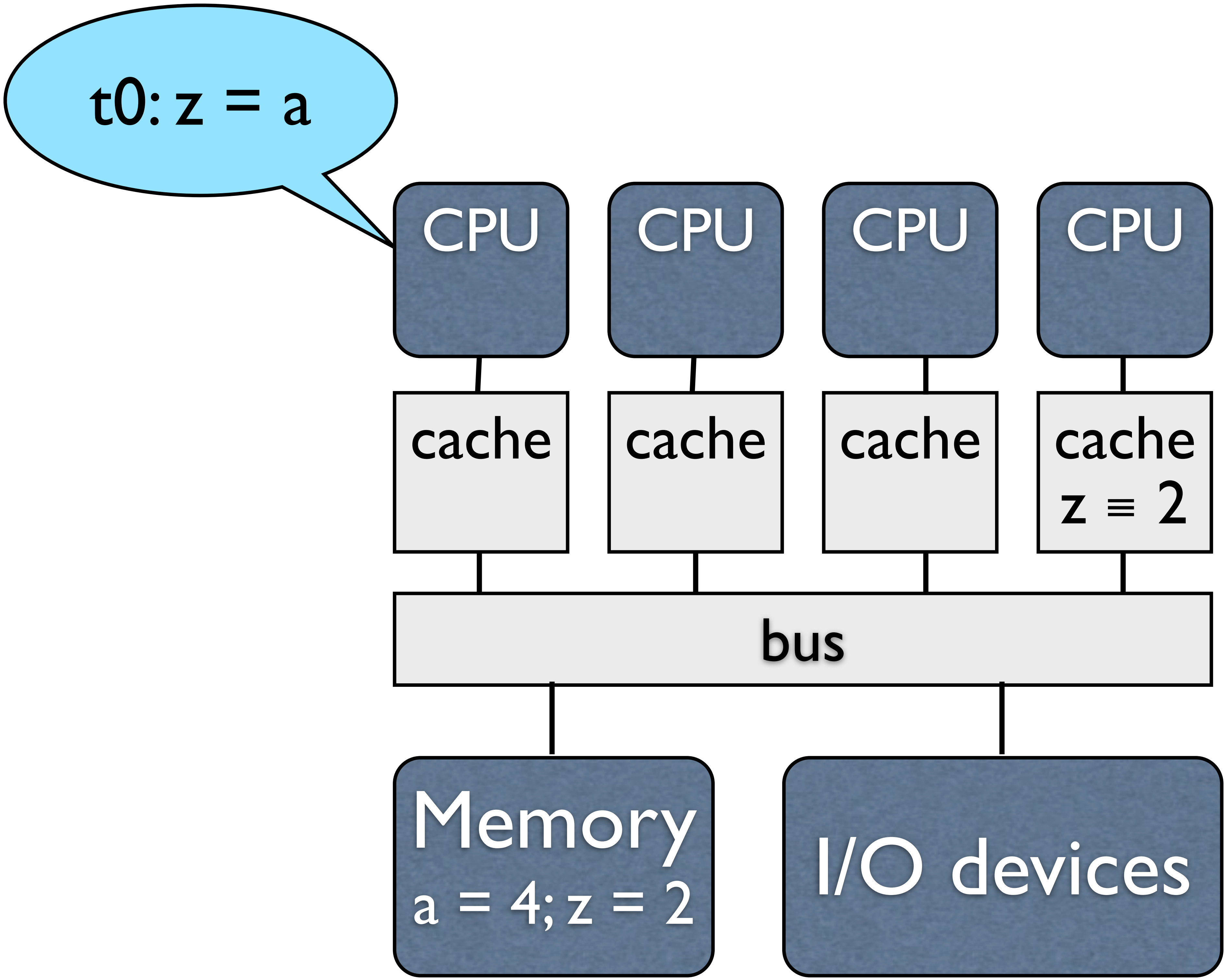
Multiprocessor (shared memory multiprocessor)

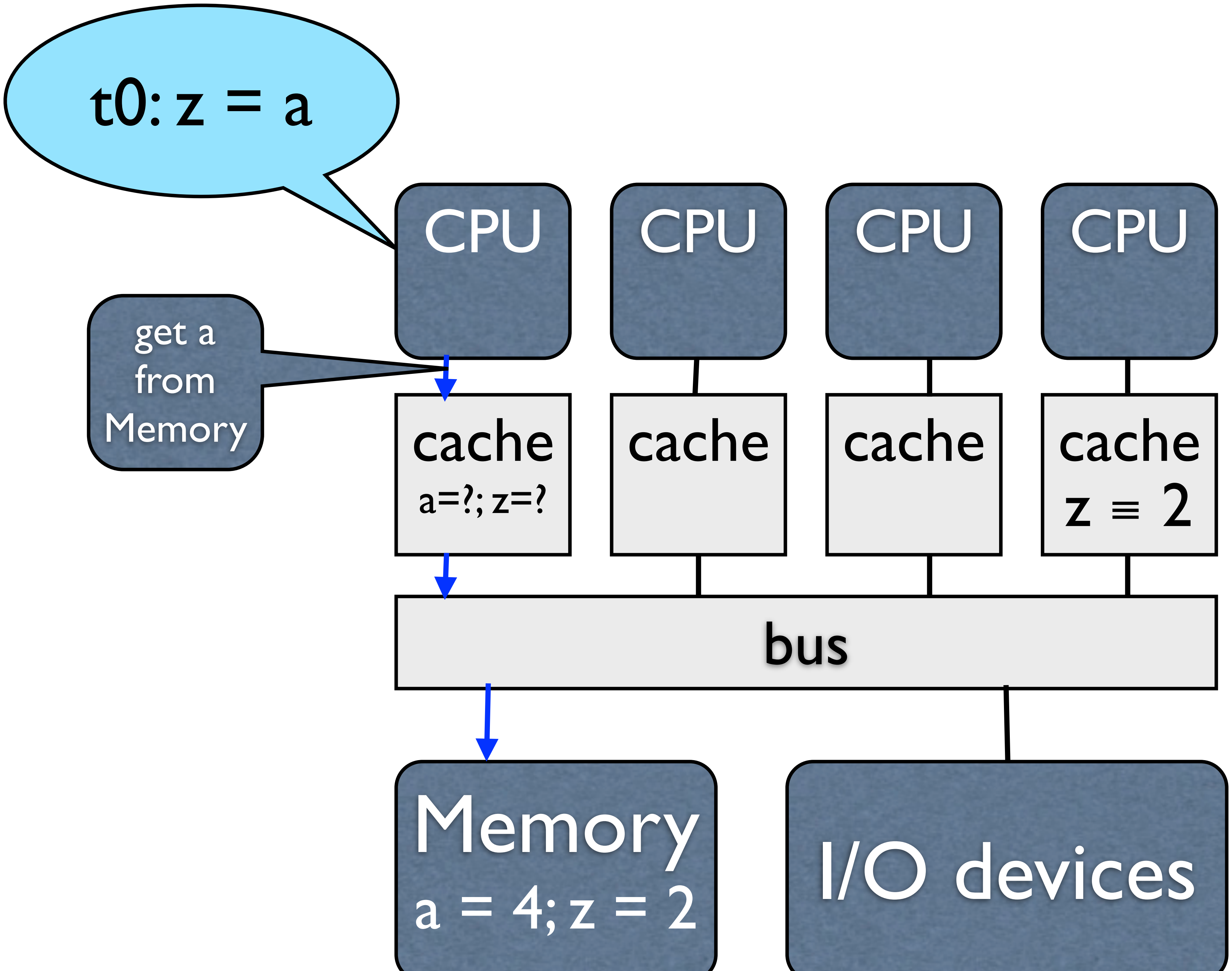
- Multiple CPUs with a shared memory (or multiple cores in the same CPU)
- The same address on two different processors points to the same memory location
  - Multicores are a version of this
- If multiple processors are used, they are connected to a *shared bus* which allows them to communicate with one another via the shared memory
- Two variants:
  - *Uniform memory access*: all processors access all memory in the same amount of time
  - *Non-uniform memory access*: different processors may see different times to access some memory.

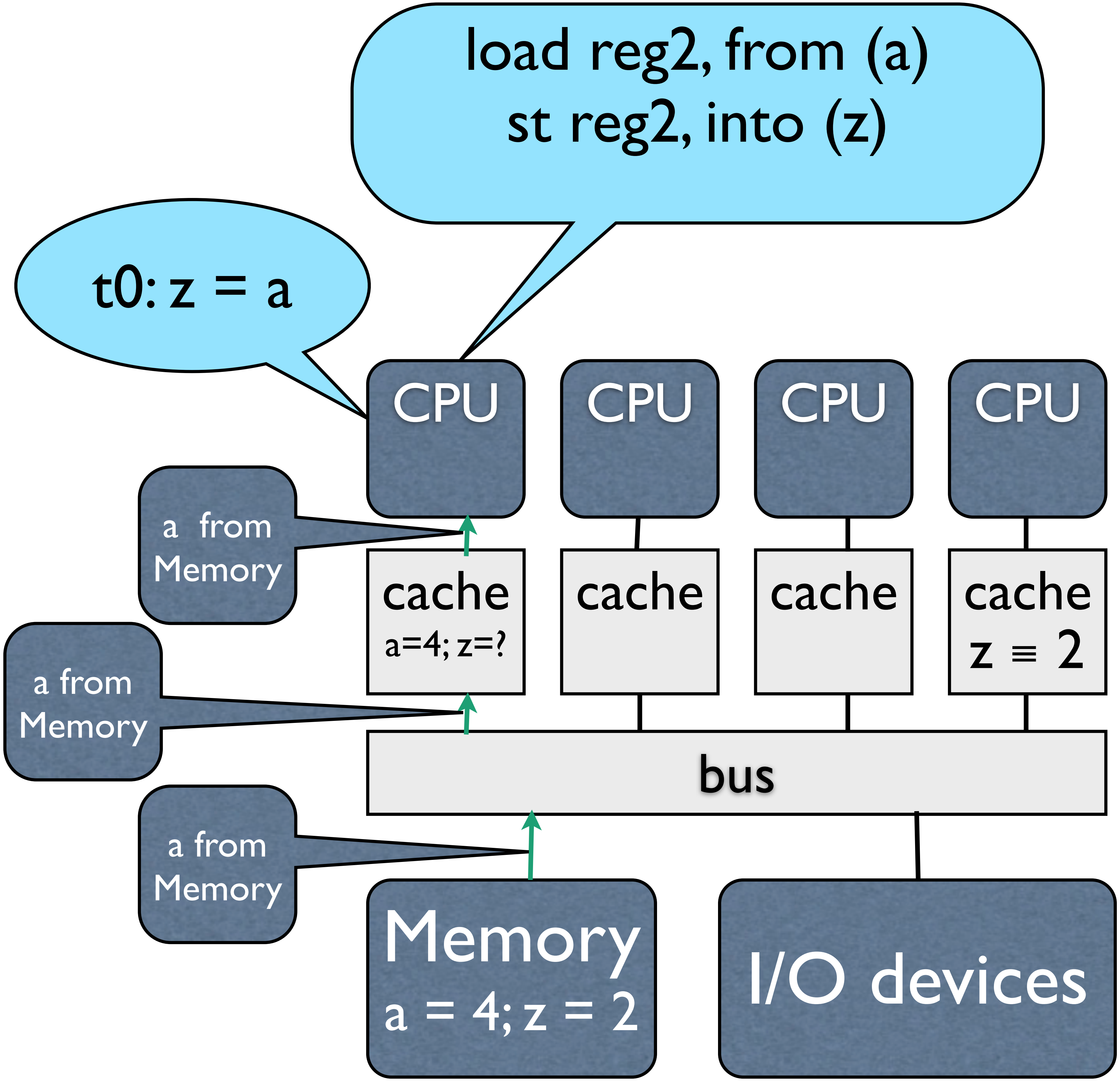
Memory hierarchies

- A computer typically has a hierarchy of memory
  - Larger memories are slower, smaller memories are faster
  - From slow to fast a typical hierarchy is
    - disk 33 - 65 million cycles, terabytes;
    - ram 180 cycles, megabytes;
    - l3 cache 36 cycles, 2 to 20MB;
    - l2 cache (12 cycles , 256KB;
    - l1 cache (4 cycles, 64 KB;
    - registers (1 cycles, 1 word to a quad-word)

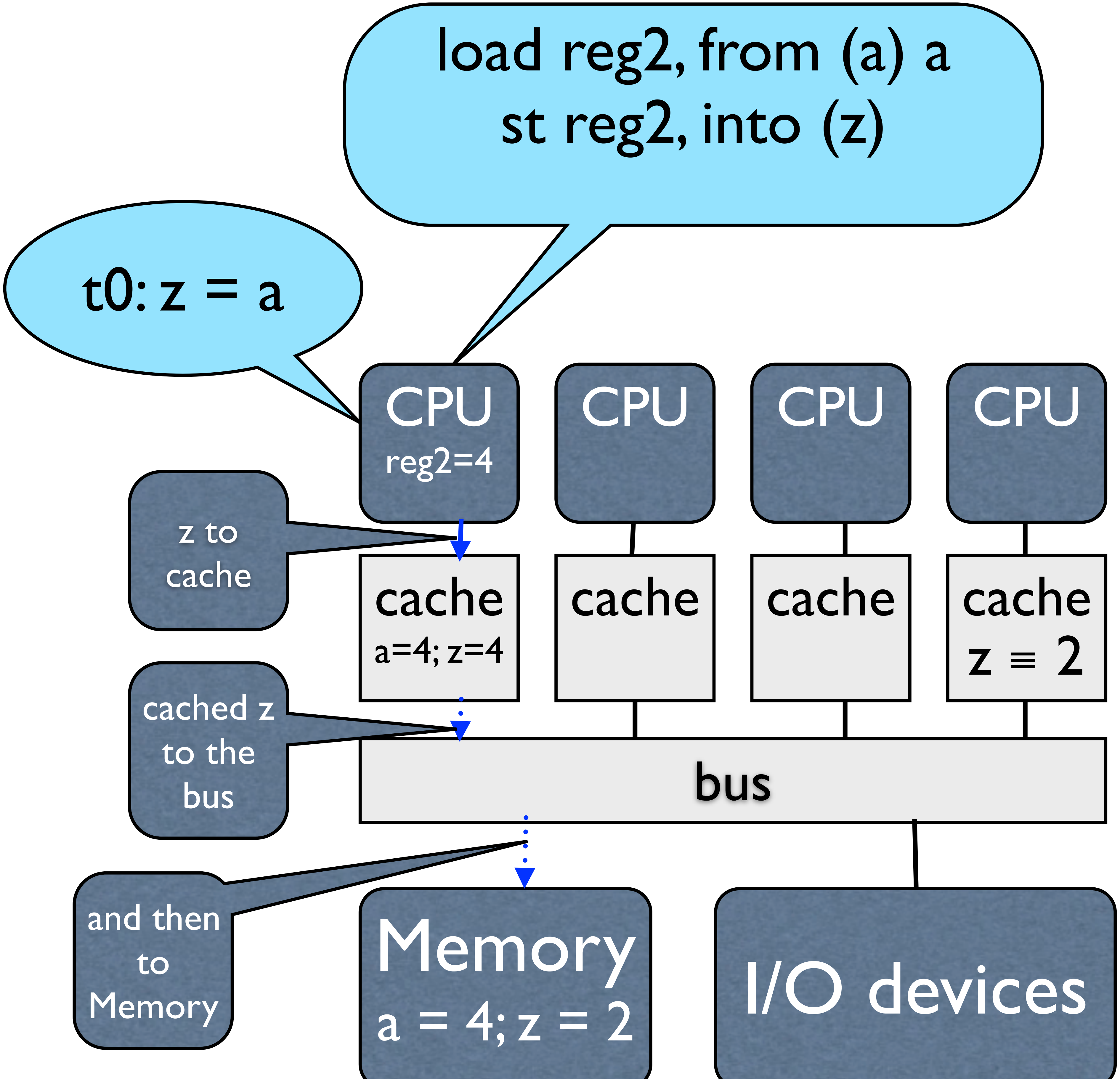
Coherence is needed



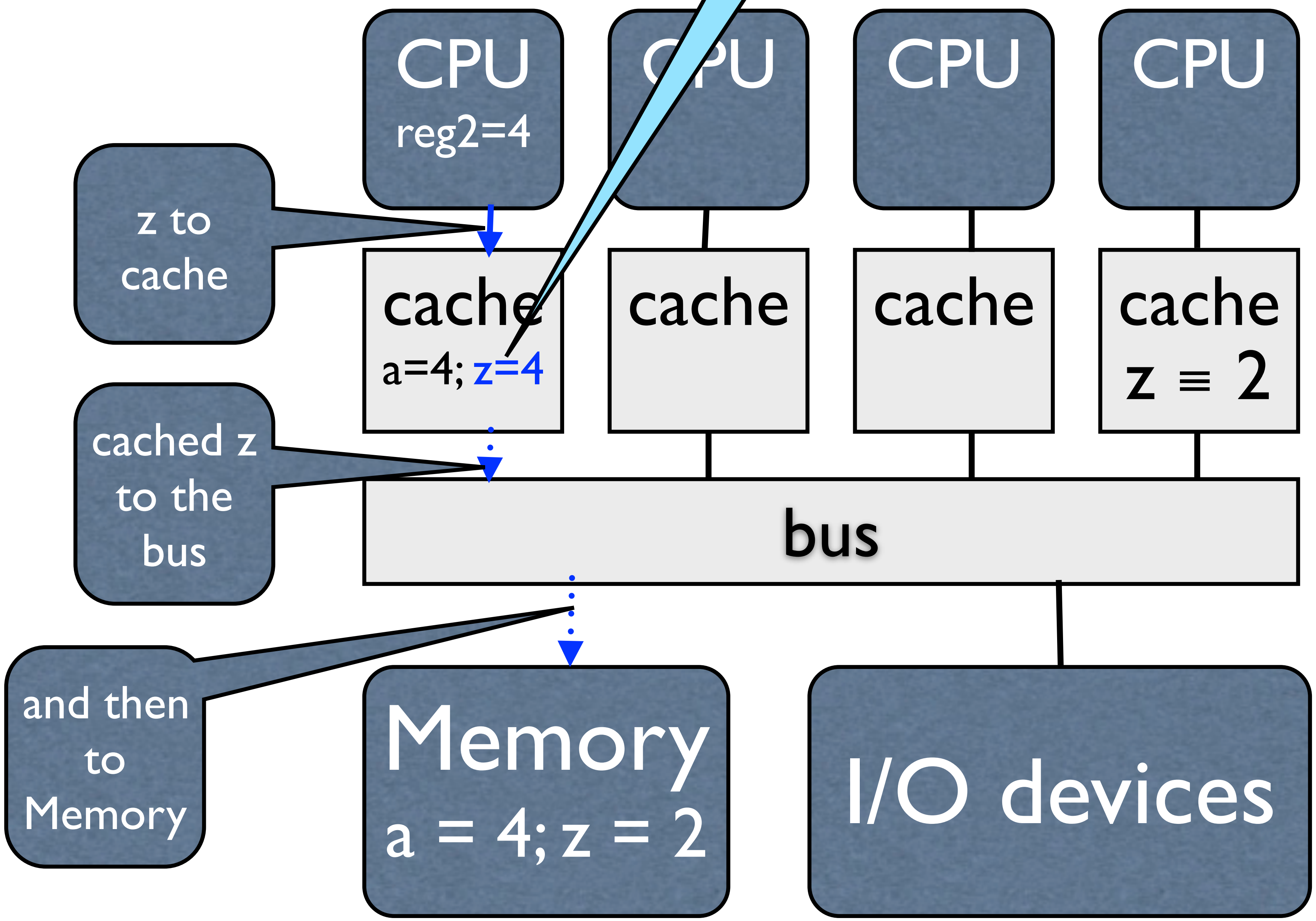

t0: z = a
get a from Memory
CPU
CPU
CPU
CPU
cache
a=?; z=?
cache
cache
cache
z ≡ 2
bus
Memory
a = 4; z = 2
I/O devices

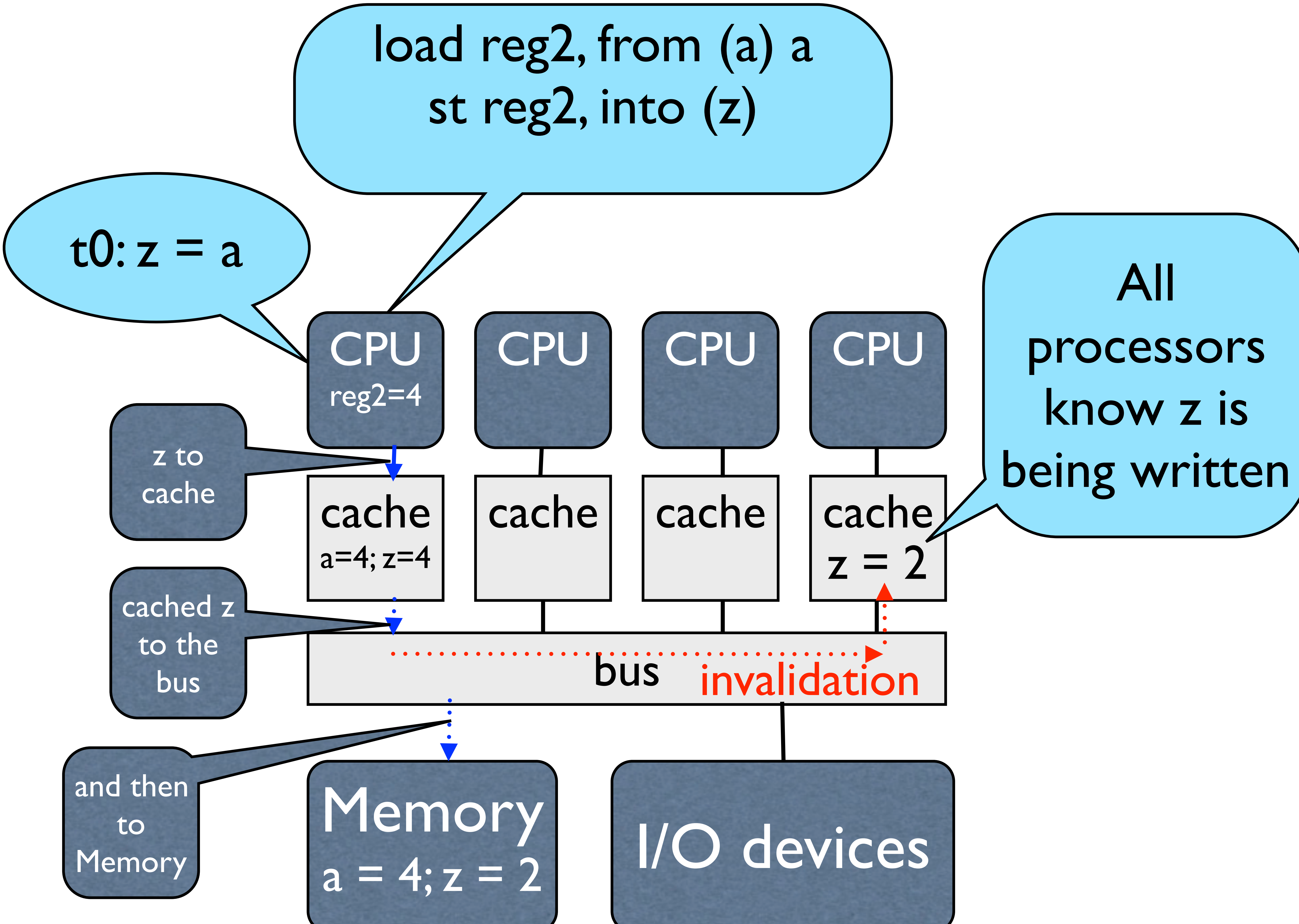


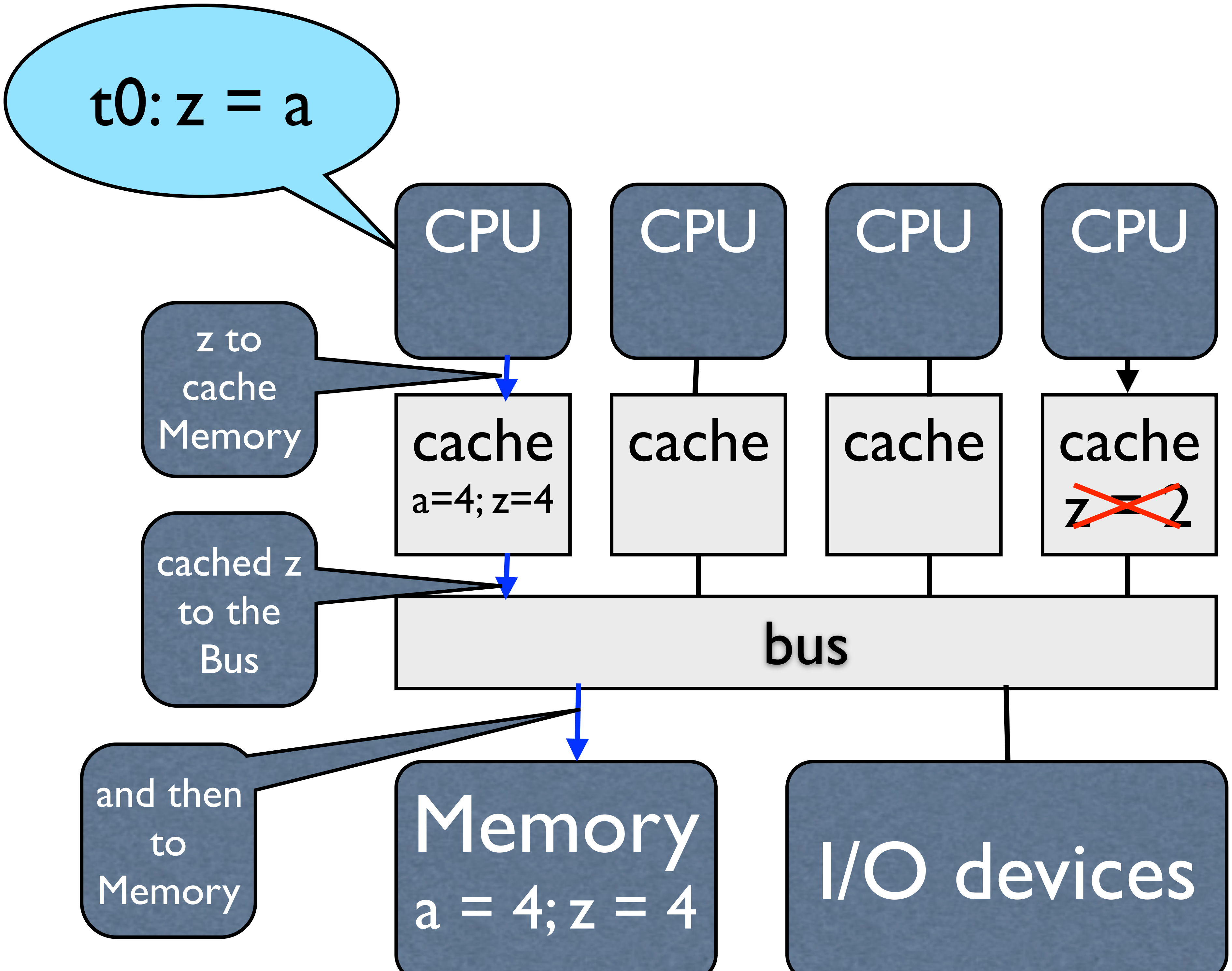
load reg2, from (a)
st reg2, into (z)
t0: z = a
CPU
CPU
CPU
CPU
a from Memory
cache
a=4; z=?
cache
cache
cache
z ≡ 2
a from Memory
bus
a from Memory
Memory
a = 4; z = 2
I/O devices

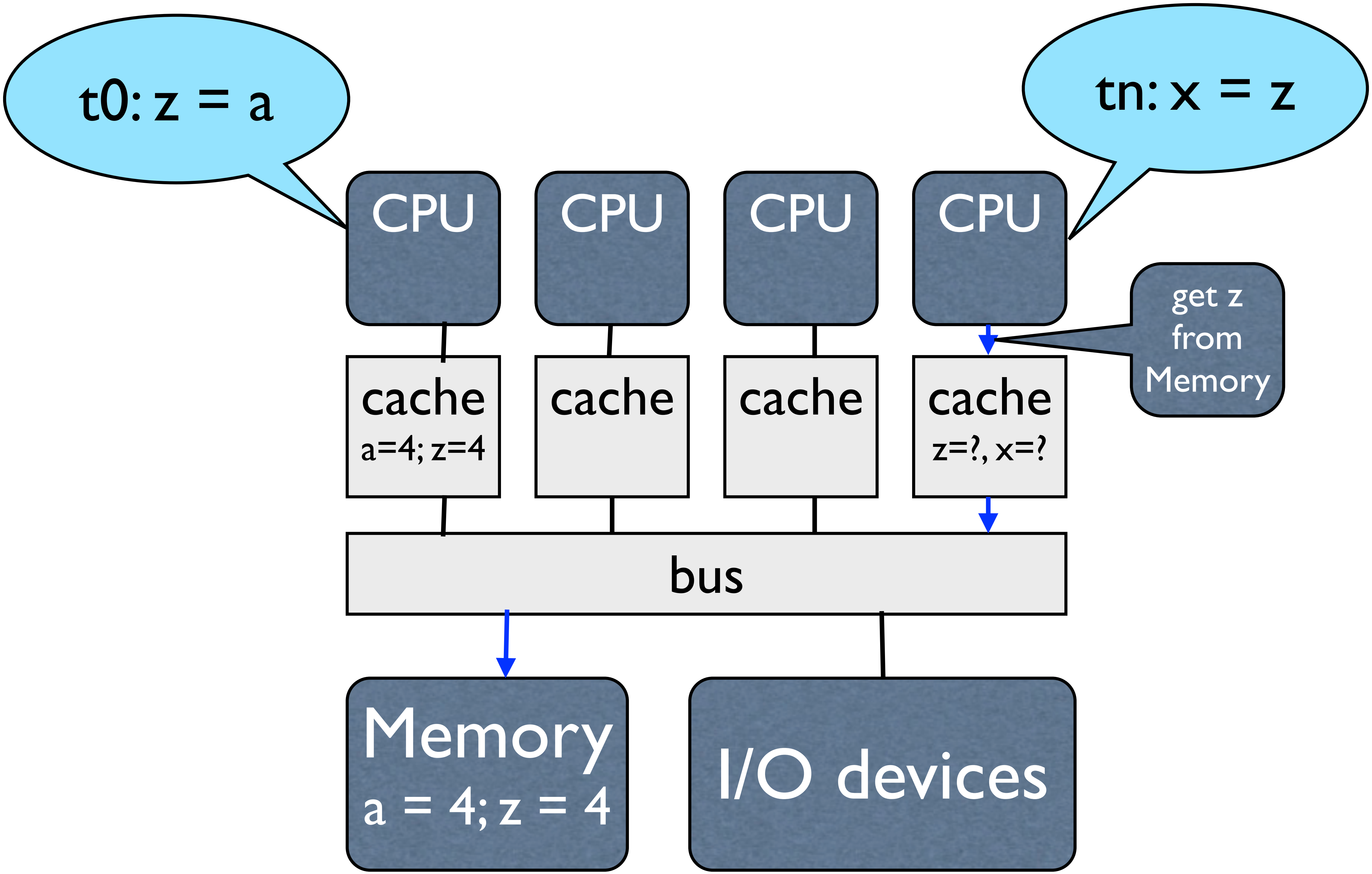

load reg2, from (a) a
st reg2, into (z)
t0: z = a
CPU
reg2=4
CPU
CPU
CPU
z to cache
cache
a=4; z=4
cache
cache
cache
z ≡ 2
cached z to the bus
bus
and then to Memory
Memory
a = 4; z = 2
I/O devices

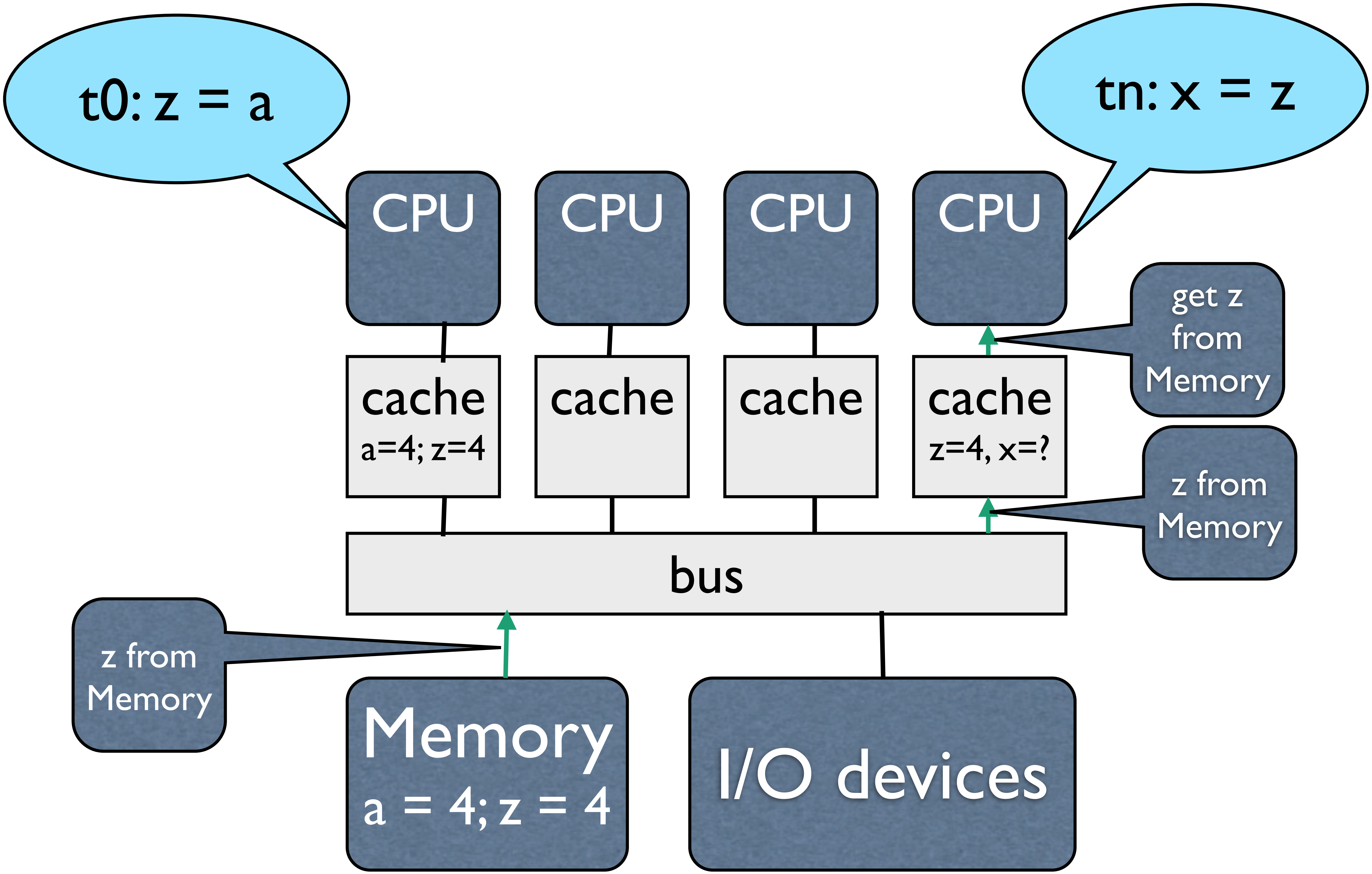
Writes are special. Many processors can read a variable (memory location) with no problem. Only one should be writing it. If many processors simultaneously write a variable values will be lost.

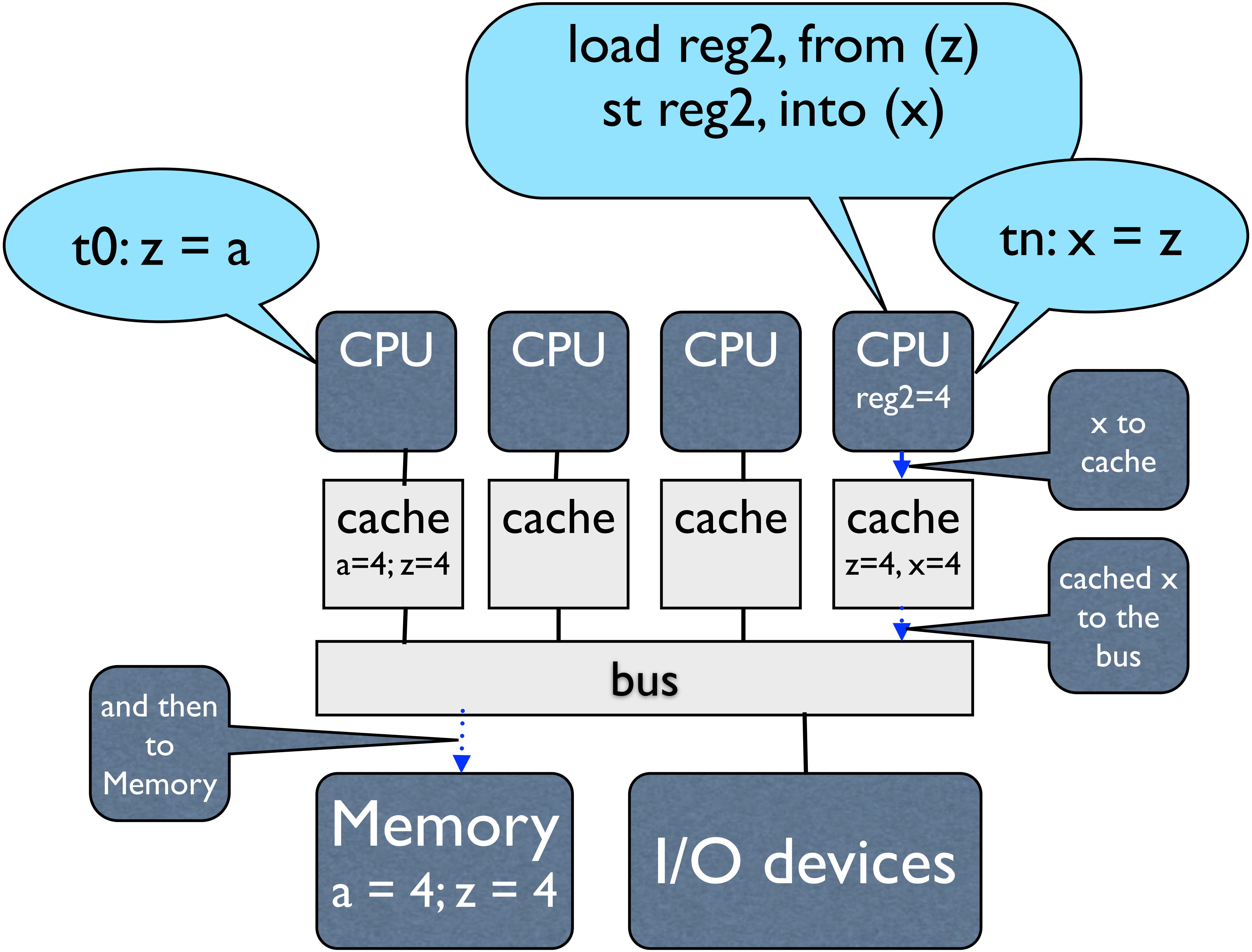


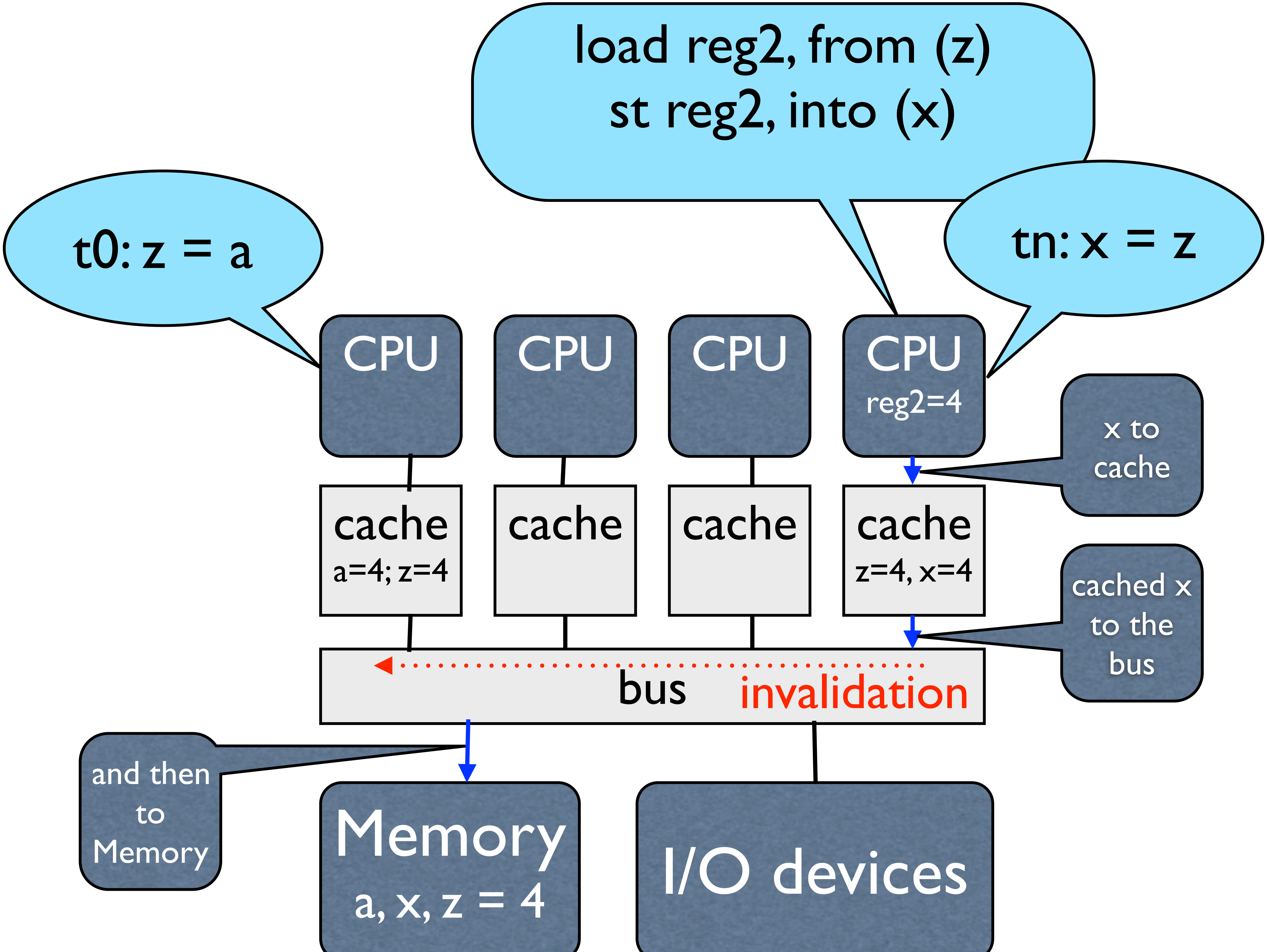

load reg2, from (a) a
st reg2, into (z)
t0: z = a
CPU
reg2=4
CPU
CPU
CPU
All processors know z is being written
z to cache
cache
a=4; z=4
cache
cache
cache
z = 2
cached z to the bus
bus
invalidation
and then to Memory
Memory
a = 4; z = 2
I/O devices


t0: z = a
CPU
CPU
CPU
CPU
z to cache Memory
cache
a=4; z=4
cache
cache
cache
~~z = 2~~
cached z to the Bus
bus
and then to Memory
Memory
a = 4; z = 4
I/O devices

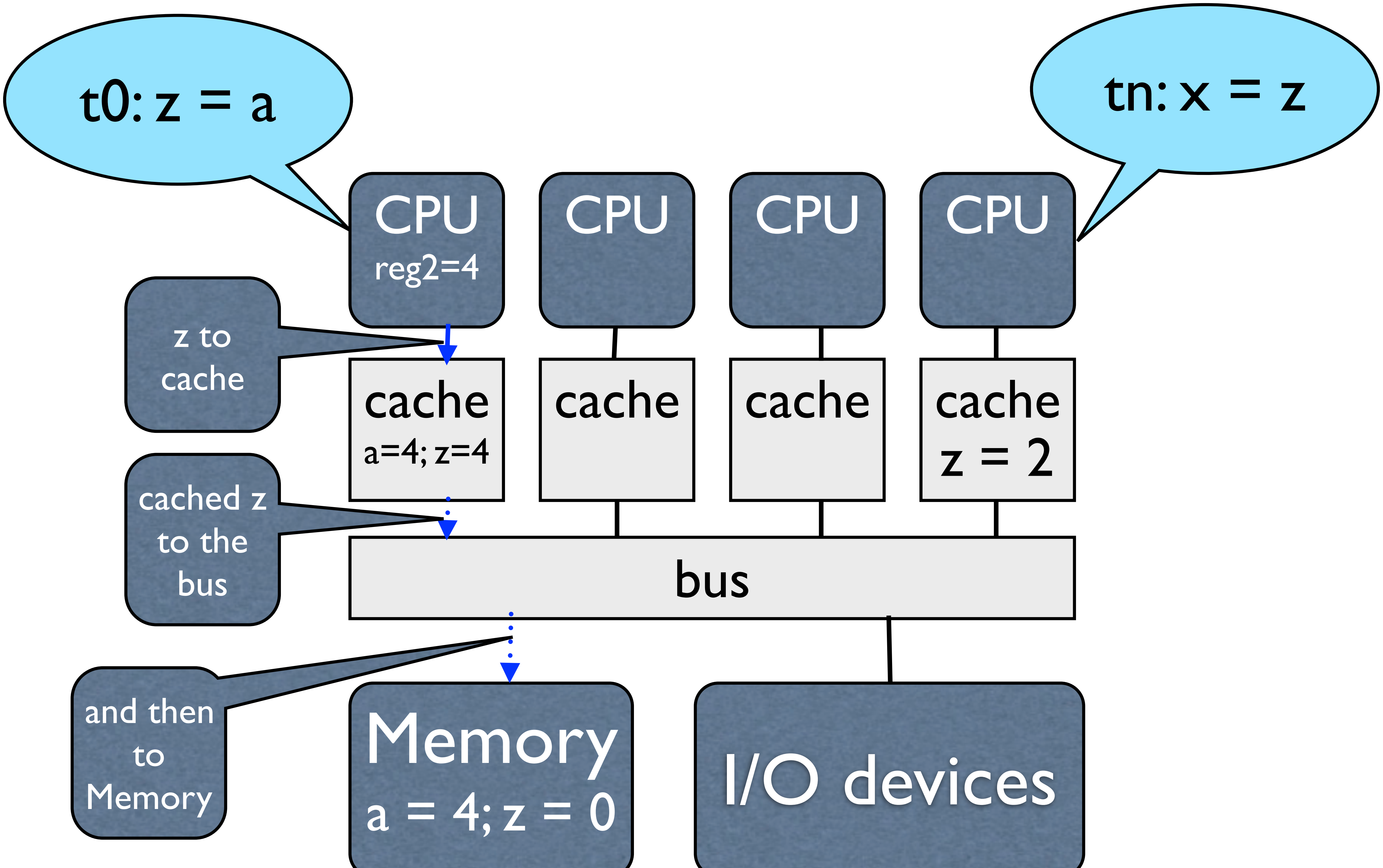

t0: z = a
tn: x = z
CPU
CPU
CPU
CPU
get z from Memory
cache
a=4; z=4
cache
cache
cache
z=?, x=?
bus
Memory
a = 4; z = 4
I/O devices


t0: z = a
tn: x = z
CPU
CPU
CPU
CPU
get z from Memory
cache
a=4; z=4
cache
cache
cache
z=4, x=?
z from Memory
bus
z from Memory
Memory
a = 4; z = 4
I/O devices

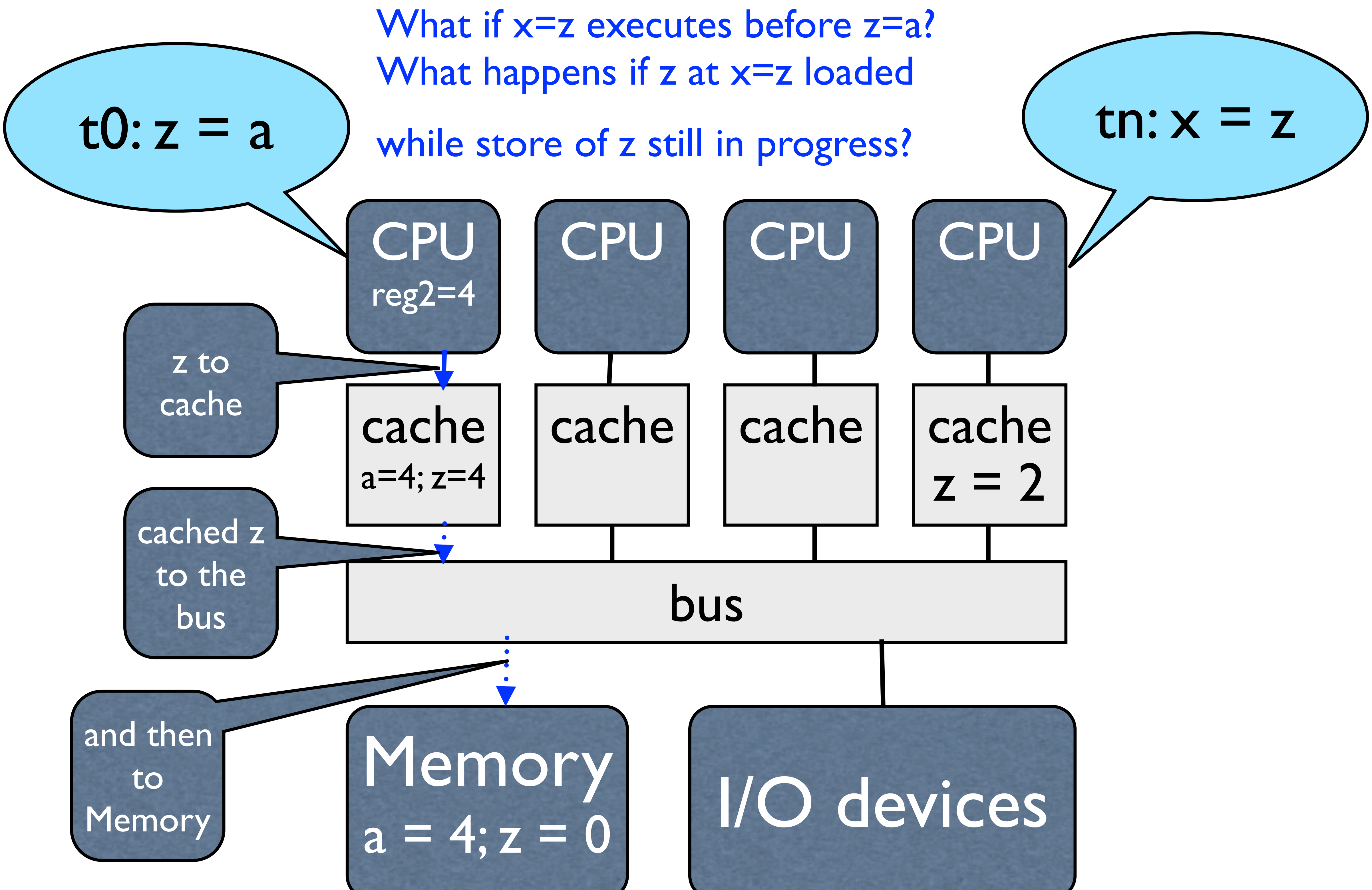

load reg2, from (z)
st reg2, into (x)
t0: z = a
tn: x = z
CPU
CPU
CPU
CPU
reg2=4
x to cache
cache
a=4; z=4
cache
cache
cache
z=4, x=4
cached x to the bus
bus
and then to Memory
Memory
a = 4; z = 4
I/O devices


load reg2, from (z)
st reg2, into (x)
t0: z = a
tn: x = z
CPU
CPU
CPU
CPU
reg2=4
x to cache
cache
a=4; z=4
cache
cache
cache
z=4, x=4
cached x to the bus
bus
invalidation
and then to Memory
Memory
a, x, z = 4
I/O devices

Hardware makes sure a core/processor reads the latest value assigned to memory (cache coherence)



Software has to make sure operations occur in the right order across threads/processors



Shared memory programming models provide the ability to write correct shared memory programs

- Can either be a language, language extension, library or a combination
  - Java is a language and associated *virtual machine* that provides runtime support
  - OpenMP is a language extension (for C/C++ and Fortran) and an associated library (or *runtime*)
  - Pthreads (or *Posix Threads*) is a library with C/C++ and Fortran bindings

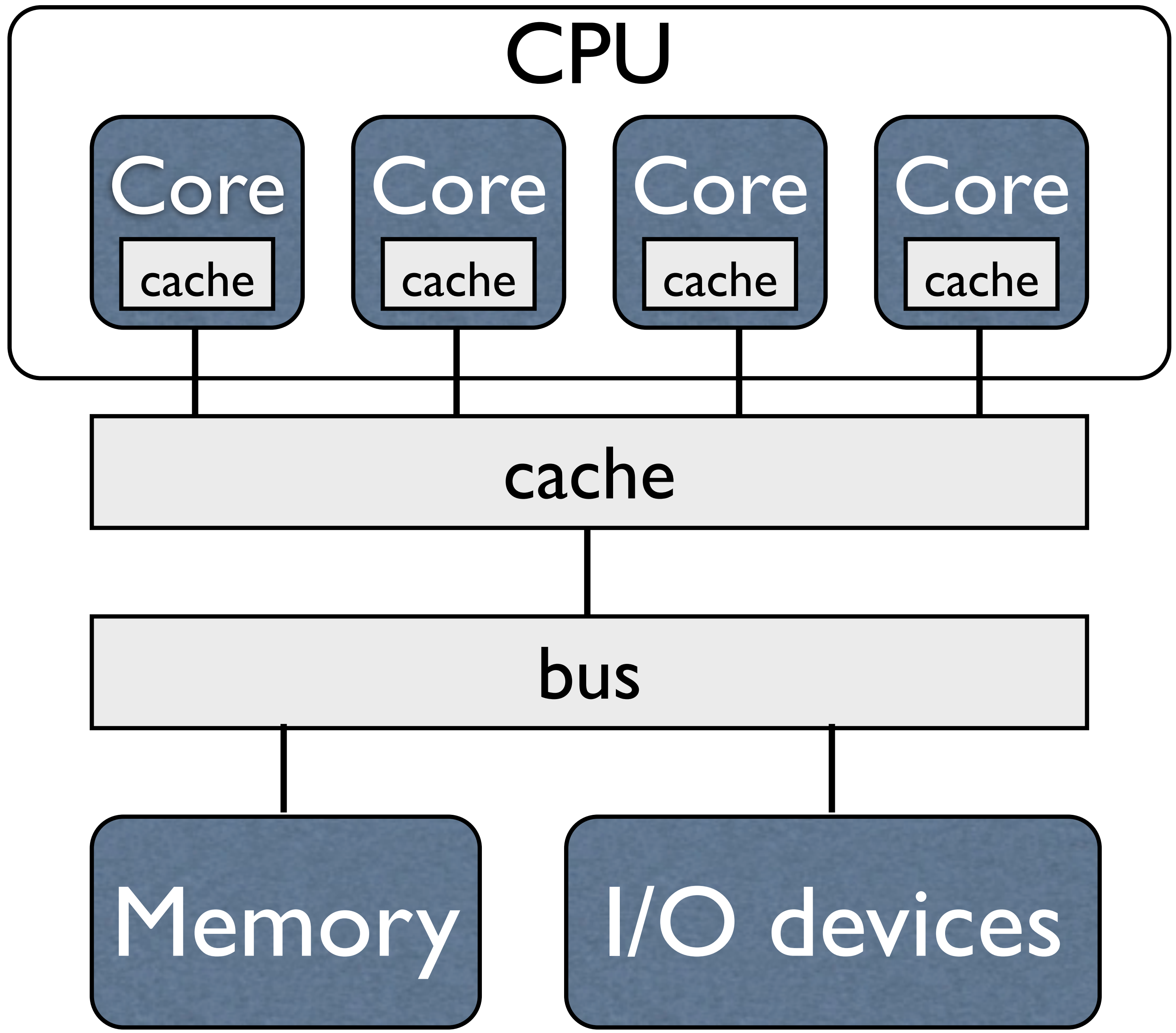
The slide that was here moved

- *A Uniform Memory Access shared memory machine*

Different kinds of shared memory machines

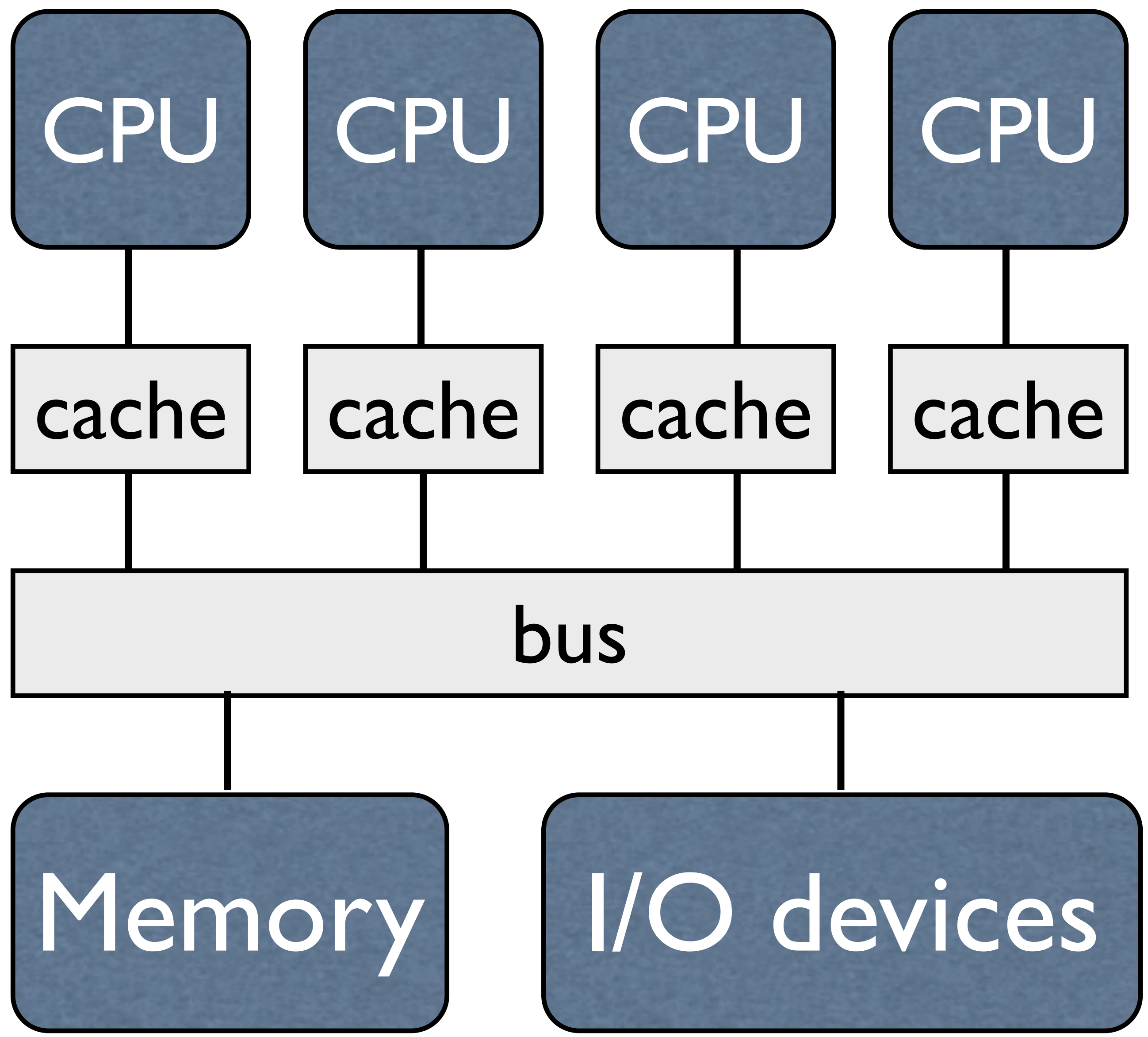
Multicore machines usually share at least one *level* of cache

All cores access global memory at the same speed

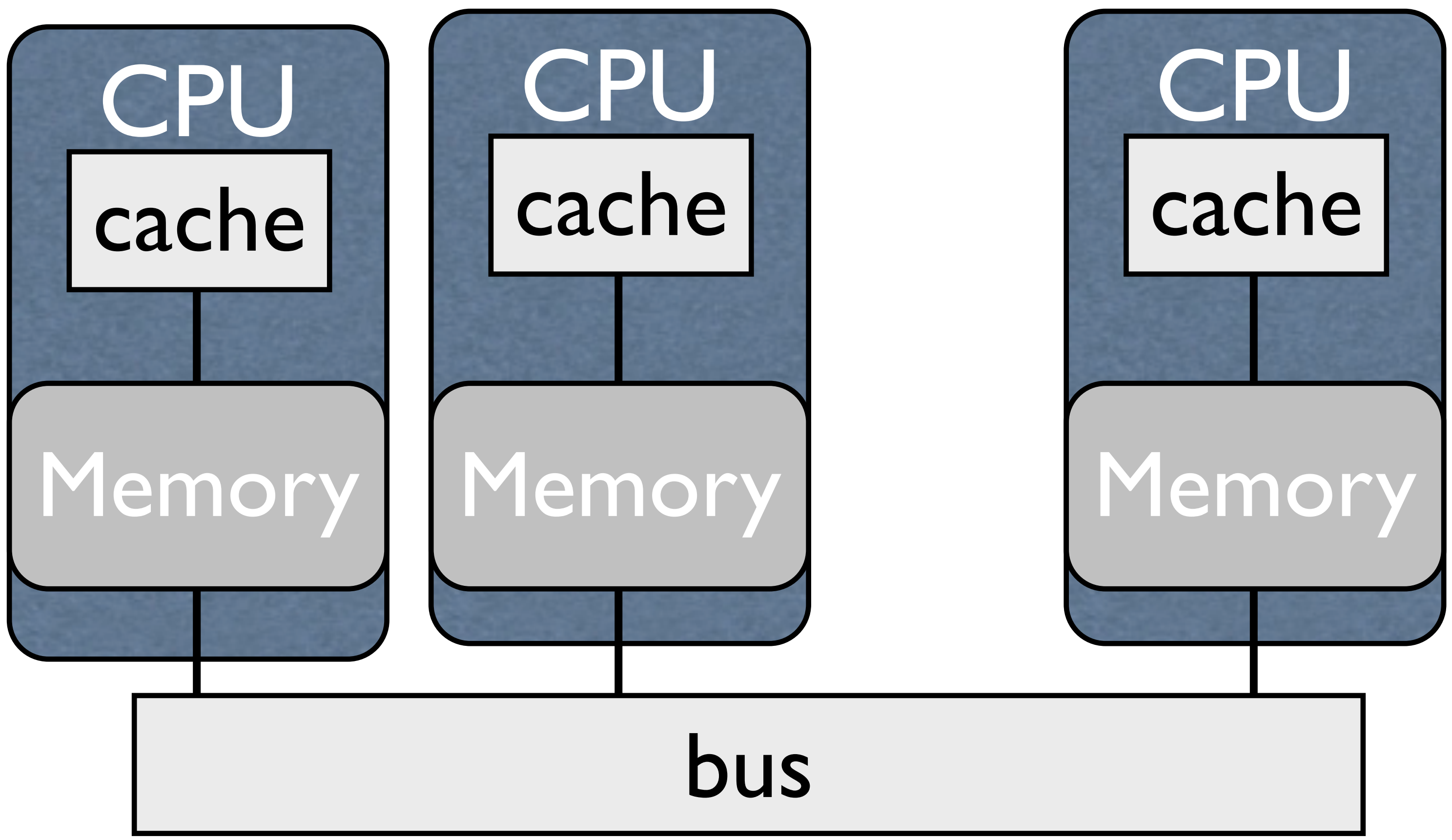


A Uniform Memory Access shared memory machine

All processors access global memory at the same speed



A NUMA shared memory machine



Processors will access their memory faster than their neighbors memory. Software and special hardware gives the illusion of a large shared address space.

A programming model must provide a way of specifying

- what parts of the program execute in parallel with one another
- how the work is distributed across different cores
- the order that multiple reads and writes to the same memory will take place
- that a sequence of accesses to a variable will occur *atomically* or without interference from other threads.
- **And,** ideally, it will do this while giving *good performance* and allowing *maintainable programs* to be written.

OpenMP

- Open *M*ulti-*P*rocessor
  - targets multicores and multi-processor shared memory machines
  - An open standard, not controlled by any manufacturer
- Allows loop-by-loop & region-by-region parallelization of sequential programs.

What executes in parallel?

```
c = 57.0;
for (i=0; i < n; i++) {
  a[i] = c[i] + a[i]*b[i]
}
```

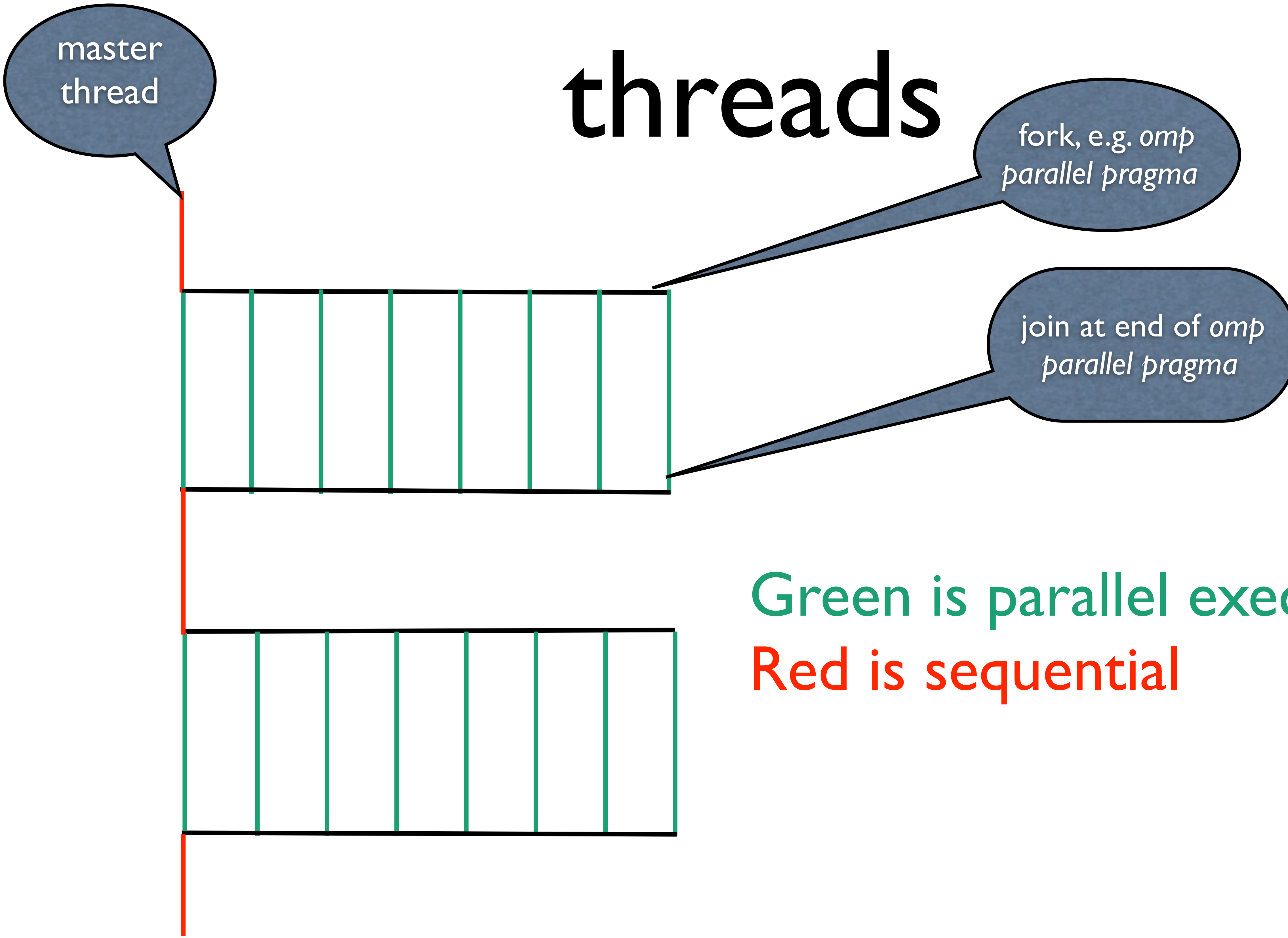
```
c = 57.0
#pragma omp parallel for
for (i=0; i < n; i++) {
  a[i] = c[i] + a[i]*b[i]
}
```

- *pragma* appears like a comment to a non-OpenMP compiler
- pragma requests parallel code to be produced for the following for loop

Threads and processes
The building blocks of shared memory programs

- Threads and processes are typically operating system entities and concepts
- A *process* has its own address space and owns a typically *virtualized* copy of the machine when executing
  - processes may own one or more threads
- A *thread* shares its address space with it's owning process and all other threads owned by the same process
  - each thread has its own copy of registers
  - local variables can be created that are accessible only by the thread
  - threads are the fundamental building block of parallel shared memory programs

Parallel execution using threads



Green is parallel execution
Red is sequential

How is the work distributed across different cores?

~~c = 57.0~~

```
#pragma omp parallel for schedule(static)
for (i=0; i < n; i++) {
   a[i] = c[i] + a[i]*b[i]
}
```

- Split the loop into chunks of contiguous iterations with approximately *t/n* iterations per chunk
- Thus, if 4 threads and 100 iterations, thread one would get iterations 0:24, thread 2 25:49, and so forth
- Other scheduling strategies supported.

The order that reads and writes to memory occur

~~c = 57.0~~

```
#pragma omp parallel for schedule(static)
for (i=0; i < n; i++) {
   a[i] = c[i] + a[i]*b[i]
}
```

barrier

```
#pragma omp parallel for schedule(static)
for (i=0; i < n; i++) {
   a[i] = c[i] + a[i]*b[i]
}
```

- Within an iteration, access to data appears in-order
- Across iterations, no order is implied. *Races* lead to undefined programs
- Across loops, an implicit *barrier* prevents a loop from starting execution until all iterations and writes (stores) to memory in the previous loop are finished
- Parallel constructs execute after preceding sequential constructs finish

Relaxing the order that reads and writes to memory occur

~~c = 57.0~~

```
#pragma omp parallel for schedule(static) nowait
for (i=0; i < n; i++) {
   a[i] = c[i] + a[i]*b[i]
}
#pragma omp parallel for schedule(static)
for (i=0; i < n; i++) {
   a[i] = c[i] + a[i]*b[i]
}
```

The *nowait* clause allows a thread to begin executing its part of the loop as soon as it finishes its part of the preceding computation

Note, however, that the barrier is associated with the first loop.

Accessing variables without interference from other threads -- forcing atomicity

```
#pragma omp parallel for
for (i=0; i < n; i++) {
   a = a + b[i]
}
```

Dangerous -- all iterations are updating *a* at the same time -- a *race* (or *data race*).

```
#pragma omp parallel for
for (i=0; i < n; i++) {
#pragma omp critical
   a = a + b[i];
}
```

Stupid but correct -- *critical* pragma allows only one thread to execute the *a* = statement at a time.   Is *very inefficient!*

OpenMP in more detail