EC 440 – Introduction to Operating Systems

Orran Krieger (BU)
Larry Woodman (Red Hat)

Slides from Christopher Kruegel, UCSB

Coding discussion

- How many of you have started?
- You have learned to use your tools, test driven development...
  - Please use and beat this up
- Nobody has yet posted tests for extra grades.
- We are building a repo for tests that we will be available soon...
  - You should be able to submit pull requests with new tests
  - You should be able to review each other's tests, and comment on them
  - We will select the best ones and incorporate them

Expectations HW

- You must only and exclusively submit code you wrote yourself.
- After the HW1 was submitted, we ran all solutions through software similarity detection.
  - It is clear that the majority of the class did the work themselves. Thank you!
  - There are a few situations where there are strong similarities to existing projects.
  - We are now collecting corpus of all previous years, and public shells, and running through similarity detection
- Why do we care?
  - You don't learn by doing this
  - Getting a good grade in this course is something to be proud of
  - You don't learn by doing this; you will be quizzed by future employers on what you did here...
  - In the long run it never works out.
- The end of this week I will be talking to anyone that the TFs were not confident fully understood their code and/or who's program was similar to other current or previous ones.
- Post a private note to Piazza if you want to talk to me first.

Review

- Bit about OS structure, history of OS and core services
- System calls, processes, threads, stacks and calling code conventions
- Scheduling policies, and a start to synchronization.
- Next:
  - go through example of how we expect you to be able to reason about synchronization
  - finish up synchronization
  - Discuss deadlock

Worksheet

```
monitor M
  condition cond1, cond2;
  function sub1();
  begin
    ...
    wait(cond1);
  end;
  function sub2();
  begin
    ...
    signal(cond1);
    ...
    wait(cond2);
  end;
  function sub3();
  begin
    ...
    signal(cond2);
    signal(cond2);
  end;
end;
```

- Process A is waiting on cond1 (in end of sub1)
- Process B is waiting on cond2 (in end of sub2)
- At time t0 process C calls M.sub2()
- At time t1 > t0 process D calls M.sub2()
- At time t2 > t1 process E calls M.sub3()

Problem:

- Assume all waiting queues are FIFO
- If Q has been waiting for condition "x" and P performs "signal(x)", consider two possible policies:
  - Policy 1: P waits until Q either leaves the monitor, or waits for another condition; or
  - Policy 2: Q waits until P either leaves the monitor, or waits for another condition
- Determine the order of execution of the processes

Policy 1 - Signalee runs

```
monitor M
  condition cond1, cond2;
  function sub1();
  begin
    ...
    wait(cond1);
  end;
  function sub2();
  begin
    ...
    signal(cond1);
    ...
    wait(cond2);
  end;
  function sub3();
  begin
    ...
    signal(cond2);
    signal(cond2);
  end;
end;
```

- P-A is on cond1 (in sub1)
- P-B is on cond2 (in sub2)
- At t0 P-C calls M.sub2()
- At t1 P-D calls M.sub2()
- At t2 P-E calls M.sub3()

C1

C2

M

Policy 1 - Signalee runs

```
monitor M
  condition cond1, cond2;
  function sub1();
  begin
    ...
    wait(cond1);
  end;
  function sub2();
  begin
    ...
    signal(cond1);
    ...
    wait(cond2);
  end;
  function sub3();
  begin
    ...
    signal(cond2);
    signal(cond2);
  end;
end;
```

- P-A is on cond1 (in sub1)
- P-B is on cond2 (in sub2)
- At t0 P-C calls M.sub2()
- At t1 P-D calls M.sub2()
- At t2 P-E calls M.sub3()

C1	P-A
C2	P-B
M	

Policy 1 - Signalee runs

```
monitor M
  condition cond1, cond2;
  function sub1();
  begin
    ...
    wait(cond1);
  end;
  function sub2();
  begin
    ...
    signal(cond1);
    ...
    wait(cond2);
  end;
  function sub3();
  begin
    ...
    signal(cond2);
    signal(cond2);
  end;
end;
```

- P-A is on cond1 (in sub1)
- P-B is on cond2 (in sub2)
- At t0 P-C calls M.sub2()
- At t1 P-D calls M.sub2()
- At t2 P-E calls M.sub3()

C1	P-A
C2	P-B
M	

```
monitor M
  condition cond1, cond2;
  function sub1();
  begin
    ...
    wait(cond1);
  end;
  function sub2();
  begin
    ...
    signal(cond1);
    ...
    wait(cond2);
  end;
  function sub3();
  begin
    ...
    signal(cond2);
    signal(cond2);
  end;
end;
```

- P-A is on cond1 (in sub1)
- P-B is on cond2 (in sub2)
- At t0 P-C calls M.sub2()
- At t1 P-D calls M.sub2()
- At t2 P-E calls M.sub3()

Policy 1 - Signalee runs

time t0

- C executes signal(cond1) and wakes up A

C1	P-A
C2	P-B
M	

```
monitor M
  condition cond1, cond2;
  function sub1();
  begin
    ...
    wait(cond1);
  end;
  function sub2();
  begin
    ...
    signal(cond1);
    ...
    wait(cond2);
  end;
  function sub3();
  begin
    ...
    signal(cond2);
    signal(cond2);
  end;
end;
```

- P-A is on cond1 (in sub1)
- P-B is on cond2 (in sub2)
- At t0 P-C calls M.sub2()
- At t1 P-D calls M.sub2()
- At t2 P-E calls M.sub3()

Policy 1 - Signalee runs

time t0

- C executes signal(cond1) and wakes up A
- C suspends and A starts executing sub1()

C1

C2 P-B

M P-C

```
monitor M
  condition cond1, cond2;
  function sub1();
  begin
    ...
    wait(cond1);
  end;
  function sub2();
  begin
    ...
    signal(cond1);
    ...
    wait(cond2);
  end;
  function sub3();
  begin
    ...
    signal(cond2);
    signal(cond2);
  end;
end;
```

- P-A is on cond1 (in sub1)
- P-B is on cond2 (in sub2)
- At t0 P-C calls M.sub2()
- At t1 P-D calls M.sub2()
- At t2 P-E calls M.sub3()

Policy 1 - Signalee runs

time t0

- C executes signal(cond1) and wakes up A
- C suspends and A starts executing sub1()
- A exits the monitor
- C restarts
- C waits on cond2 (after B)

C1

C2 P-B P-C

M

Policy 1 - Signalee runs

```
monitor M
  condition cond1, cond2;
  function sub1();
  begin
    ...
    wait(cond1);
  end;
  function sub2();
  begin
    ...
    signal(cond1);
    ...
    wait(cond2);
  end;
  function sub3();
  begin
    ...
    signal(cond2);
    signal(cond2);
  end;
end;
```

time t0

- C executes signal(cond1) and wakes up A
- C suspends and A starts executing sub1()
- A exits the monitor
- C restarts
- C waits on cond2 (after B)

time t1

- D enters the monitor with sub2()
- D executes signal(cond1) and nothing happens
- D waits on cond2 after (B and C)

- P-A is on cond1 (in sub1)
- P-B is on cond2 (in sub2)
- At t0 P-C calls M.sub2()
- At t1 P-D calls M.sub2()
- At t2 P-E calls M.sub3()

C1

C2 P-B P-C P-D

M

```
monitor M
  condition cond1, cond2;
  function sub1();
  begin
    ...
    wait(cond1);
  end;
  function sub2();
  begin
    ...
    signal(cond1);
    ...
    wait(cond2);
  end;
  function sub3();
  begin
    ...
    signal(cond2);
    signal(cond2);
  end;
end;
```

Policy 1 - Signalee runs

time t0

- C executes signal(cond1) and wakes up A
- C suspends and A starts executing sub1()
- A exits the monitor
- C restarts
- C waits on cond2 (after B)

time t1

- D enters the monitor with sub2()
- D executes signal(cond1) and nothing happens
- D waits on cond2 after (B and C)

time t2

- E enters the monitor with sub3()
- E executes the first signal on cond2 and wakes B
- E suspends and B starts

- P-A is on cond1 (in sub1)
- P-B is on cond2 (in sub2)
- At t0 P-C calls M.sub2()
- At t1 P-D calls M.sub2()
- At t2 P-E calls M.sub3()

C1		
C2	P-C	P-D
M	P-E	

Policy 1 - Signalee runs

```
monitor M
  condition cond1, cond2;
  function sub1();
  begin
    ...
    wait(cond1);
  end;
  function sub2();
  begin
    ...
    signal(cond1);
    ...
    wait(cond2);
  end;
  function sub3();
  begin
    ...
    signal(cond2);
    signal(cond2);
  end;
end;
```

time t0

- C executes signal(cond1) and wakes up A
- C suspends and A starts executing sub1()
- A exits the monitor
- C restarts
- C waits on cond2 (after B)

time t1

- D enters the monitor with sub2()
- D executes signal(cond1) and nothing happens
- D waits on cond2 after (B and C)

time t2

- E enters the monitor with sub3()
- E executes the first signal on cond2 and wakes B
- E suspends and B starts
- B exits the monitor and E restarts

- P-A is on cond1 (in sub1)
- P-B is on cond2 (in sub2)
- At t0 P-C calls M.sub2()
- At t1 P-D calls M.sub2()
- At t2 P-E calls M.sub3()

C1

C2 P-C P-D

M

Policy 1 - Signalee runs

```
monitor M
  condition cond1, cond2;
  function sub1();
  begin
    ...
    wait(cond1);
  end;
  function sub2();
  begin
    ...
    signal(cond1);
    ...
    wait(cond2);
  end;
  function sub3();
  begin
    ...
    signal(cond2);
    signal(cond2);
  end;
end;
```

- P-A is on cond1 (in sub1)
- P-B is on cond2 (in sub2)
- At t0 P-C calls M.sub2()
- At t1 P-D calls M.sub2()
- At t2 P-E calls M.sub3()

C1

C2 P-D

M P-E

time t0

- C executes signal(cond1) and wakes up A
- C suspends and A starts executing sub1()
- A exits the monitor
- C restarts
- C waits on cond2 (after B)

time t1

- D enters the monitor with sub2()
- D executes signal(cond1) and nothing happens
- D waits on cond2 after (B and C)

time t2

- E enters the monitor with sub3()
- E executes the first signal on cond2 and wakes B
- E suspends and B starts
- B exits the monitor and E restarts
- E executes the second signal and wakes C
- E suspends and C starts

```
monitor M
  condition cond1, cond2;
  function sub1();
  begin
    ...
    wait(cond1);
  end;
  function sub2();
  begin
    ...
    signal(cond1);
    ...
    wait(cond2);
  end;
  function sub3();
  begin
    ...
    signal(cond2);
    signal(cond2);
  end;
end;
```

- P-A is on cond1 (in sub1)
- P-B is on cond2 (in sub2)
- At t0 P-C calls M.sub2()
- At t1 P-D calls M.sub2()
- At t2 P-E calls M.sub3()

C1

C2 P-D

M

Policy 1 - Signalee runs

time t0

- C executes signal(cond1) and wakes up A
- C suspends and A starts executing sub1()
- A exits the monitor
- C restarts
- C waits on cond2 (after B)

time t1

- D enters the monitor with sub2()
- D executes signal(cond1) and nothing happens
- D waits on cond2 after (B and C)

time t2

- E enters the monitor with sub3()
- E executes the first signal on cond2 and wakes B
- E suspends and B starts
- B exits the monitor and E restarts
- E executes the second signal and wakes C
- E suspends and C starts
- C exits the monitor and E restarts
- E exits the monitor

Worksheet

```
monitor M
  condition cond1, cond2;
  function sub1();
  begin
    ...
    wait(cond1);
  end;
  function sub2();
  begin
    ...
    signal(cond1);
    ...
    wait(cond2);
  end;
  function sub3();
  begin
    ...
    signal(cond2);
    signal(cond2);
  end;
end;
```

- P-A is on cond1 (in sub1)
- P-B is on cond2 (in sub2)
- At t0 P-C calls M.sub2()
- At t1 P-D calls M.sub2()
- At t2 P-E calls M.sub3()

Policy 2 - Signaler runs first

- C executes signal(cond1) and wakes up A
- C continues until it waits on cond2 (after B)
- C suspends and A starts executing sub1()
- A exits the monitor
- D enters the monitor with sub2()
- D executes signal(cond1) and nothing happens
- D waits on cond2 after (B and C)
- E enters the monitor with sub3()
- E executes the first signal on cond2 and wakes B
- E executes the second signal on cond2 and wakes C
- E exits the monitor
- B starts
- B exits the monitor and C starts
- C exits the monitor

Solution

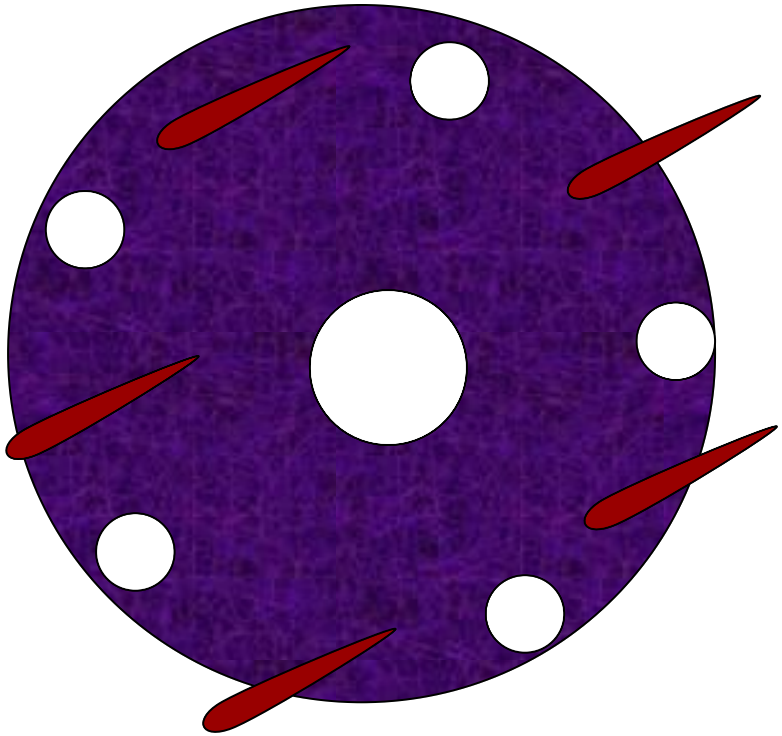
Policy 1- Signalee runs first

- C executes signal(cond1) and wakes up A
- C suspends and A starts executing sub1()
- A exits the monitor
- C restarts
- C waits on cond2 (after B)
- D enters the monitor with sub2()
- D executes signal(cond1) and nothing happens
- D waits on cond2 after (B and C)
- E enters the monitor with sub3()
- E executes the first signal on cond2 and wakes B
- E suspends and B starts
- B exits the monitor and E restarts
- E executes the second signal and wakes C
- E suspends and C starts
- C exits the monitor and E restarts
- E exits the monitor

Policy 2- Signaler runs first

- C executes signal(cond1) and wakes up A
- C continues until it waits on cond2 (after B)
- C suspends and A starts executing sub1()
- A exits the monitor
- D enters the monitor with sub2()
- D executes signal(cond1) and nothing happens
- D waits on cond2 after (B and C)
- E enters the monitor with sub3()
- E executes the first signal on cond2 and wakes B
- E executes the second signal on cond2 and wakes C
- E exits the monitor
- B starts
- B exits the monitor and C starts
- C exits the monitor

Dining Philosophers Problem



First Solution

```
philosopher(int i) {
 while (1) {
    think();
    take_chopstick(i);
    take_chopstick((i + 1) % N);
    eat();
    put_chopstick(i);
    put_chopstick((i + 1) % N);
  }
}
```

If all the philosopher take their left chopsticks they get stuck

Second Solution

```
philosopher(int i) {
 while (1) {
     think();
     take_chopstick(i);
     if (!available((i + 1) % N))
   {
        put_chopstick(i);
        continue();
    }
     take_chopstick((i + 1) % N);
     eat();
     put_chopstick(i);
     put_chopstick((i + 1) % N);
  }
}
```

It is possible that all the philosophers put down and pick up their chopsticks at the same time, leading to starvation

`think()` should be randomized

Third Solution

Use one mutex

- Do a `down()` when acquiring chopsticks
- Do an `up()` when releasing chopsticks

Problem:

Only one philosopher can eat at once

Fourth Solution

- Maintain state of philosophers
  - Switch to `HUNGRY` when ready to eat
  - Sleep if no chopsticks available
  - When finished wake up your neighbors
- Use one semaphore for each philosopher, to be used to suspend in case no chopsticks are available
- Use one mutex for critical regions
- Use `take_chopsticks/put_chopsticks` to acquire both chopsticks

Fourth Solution

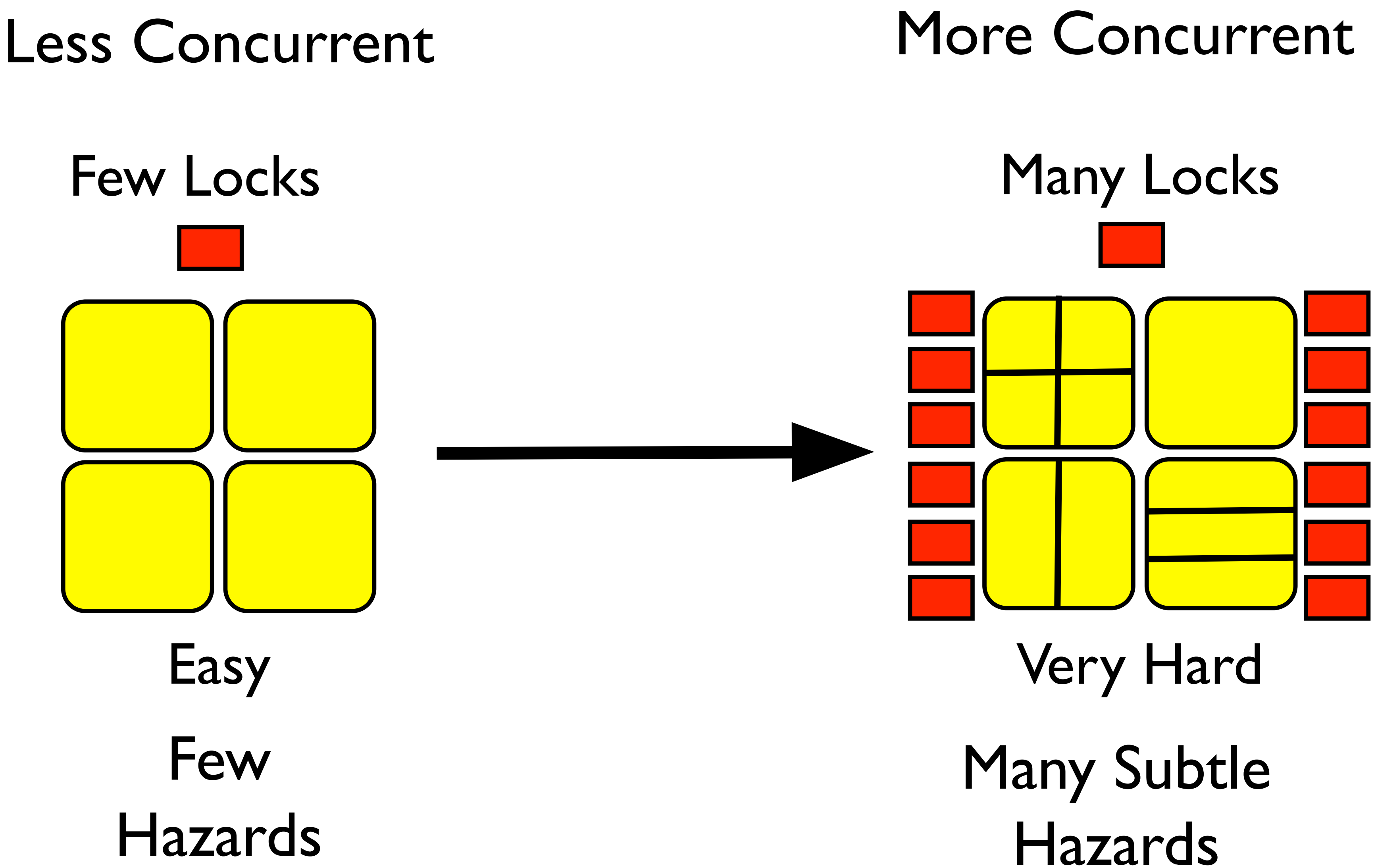
```
philosopher(i) {
  think();
  take_chopsticks(i);
  eat();
  put_chopsticks(i);
}
```

```
take_chopsticks(i) {
  mutex.down();
  state[i] = HUNGRY;
  test(i);
  mutex.up();
  philosopher[i].down();
}
```

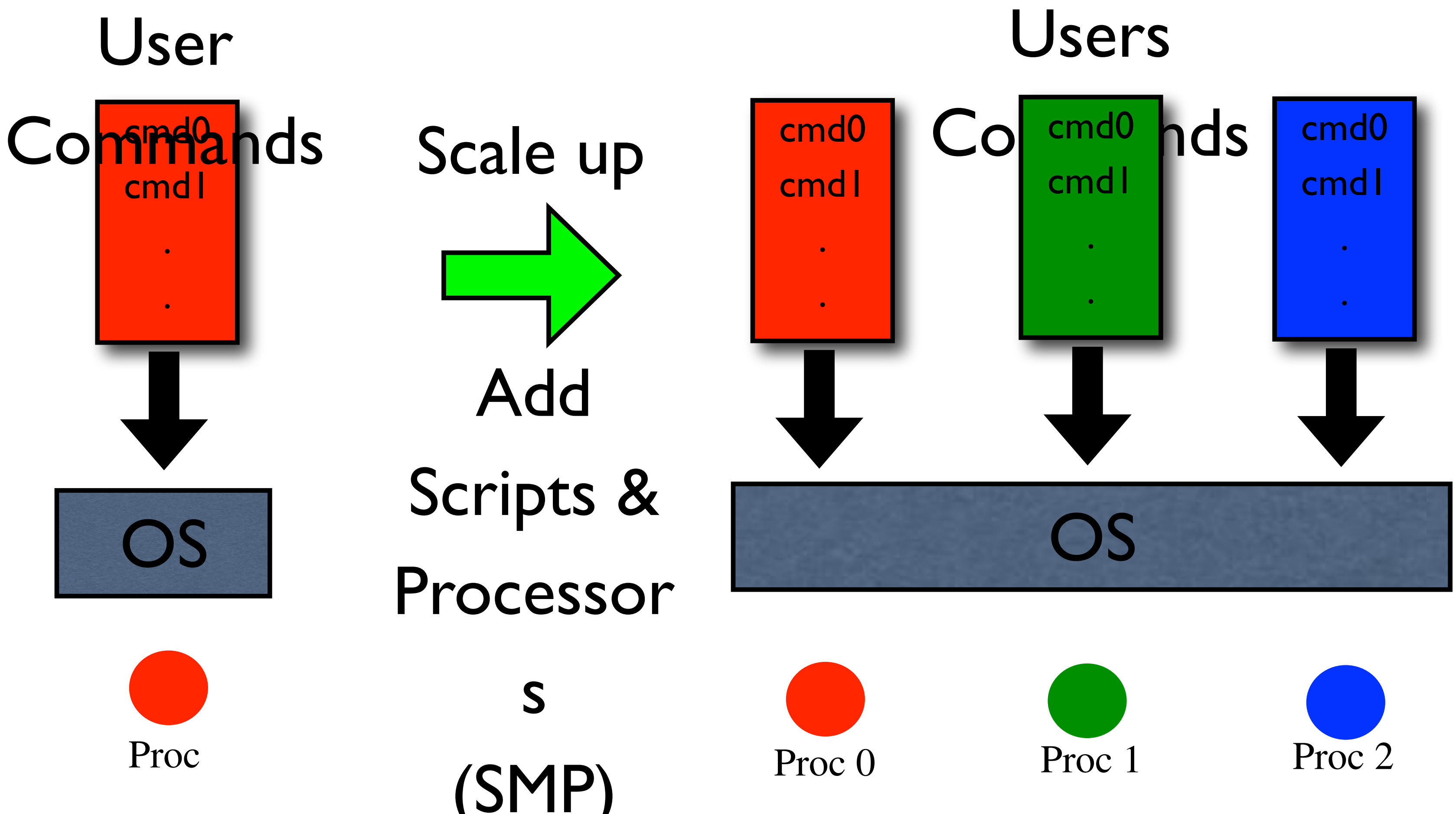
```
put_chopsticks(i) {
  mutex.down();
  state[i] = THINKING;
  test((i + 1) % N);
  test((i + N - 1) % N);
  mutex.up();
}
```

```
test(i) {
  if (state[i] == HUNGRY && state[(i + 1) % N] != EATING &&
                      state[(i + N - 1) % N] != EATING) {
        state[i] = EATING;
        philosopher[i].up();
  }
}
```

Lets discuss real world...

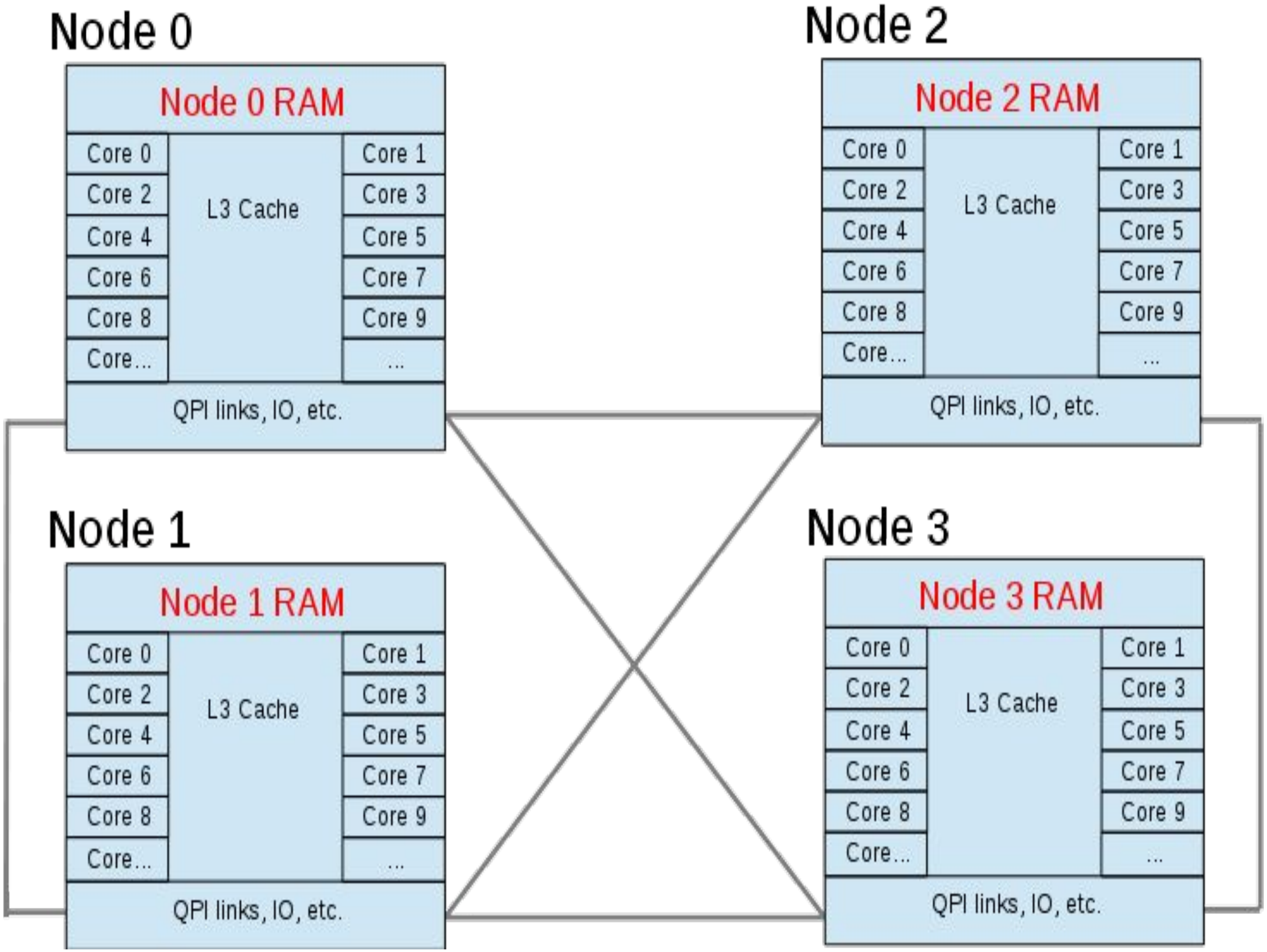

Less Concurrent
More Concurrent
Few Locks
Many Locks
Easy
Very Hard
Few
Hazards
Many Subtle
Hazards

Systems view of Scalability



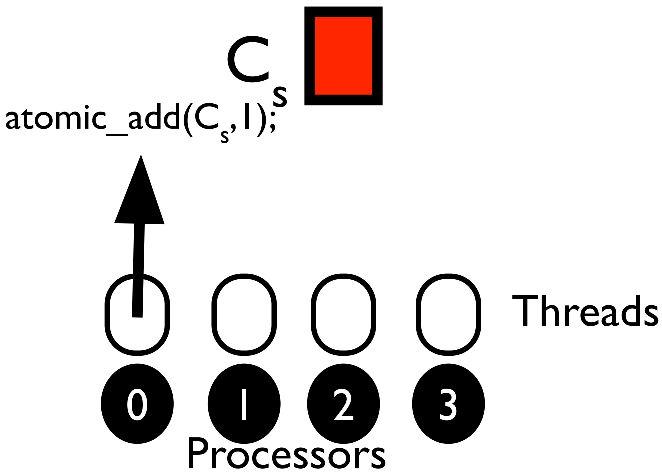
Measure throughput -- Does it scale?

Typical Four-Node NUMA System



Concurrent SW != Scalability

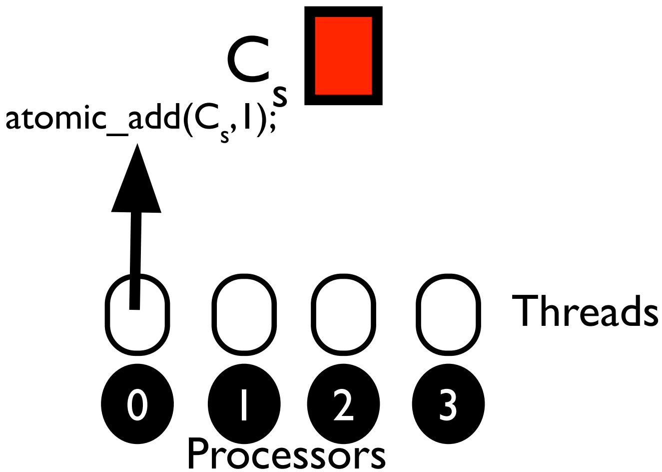
Shared Counter



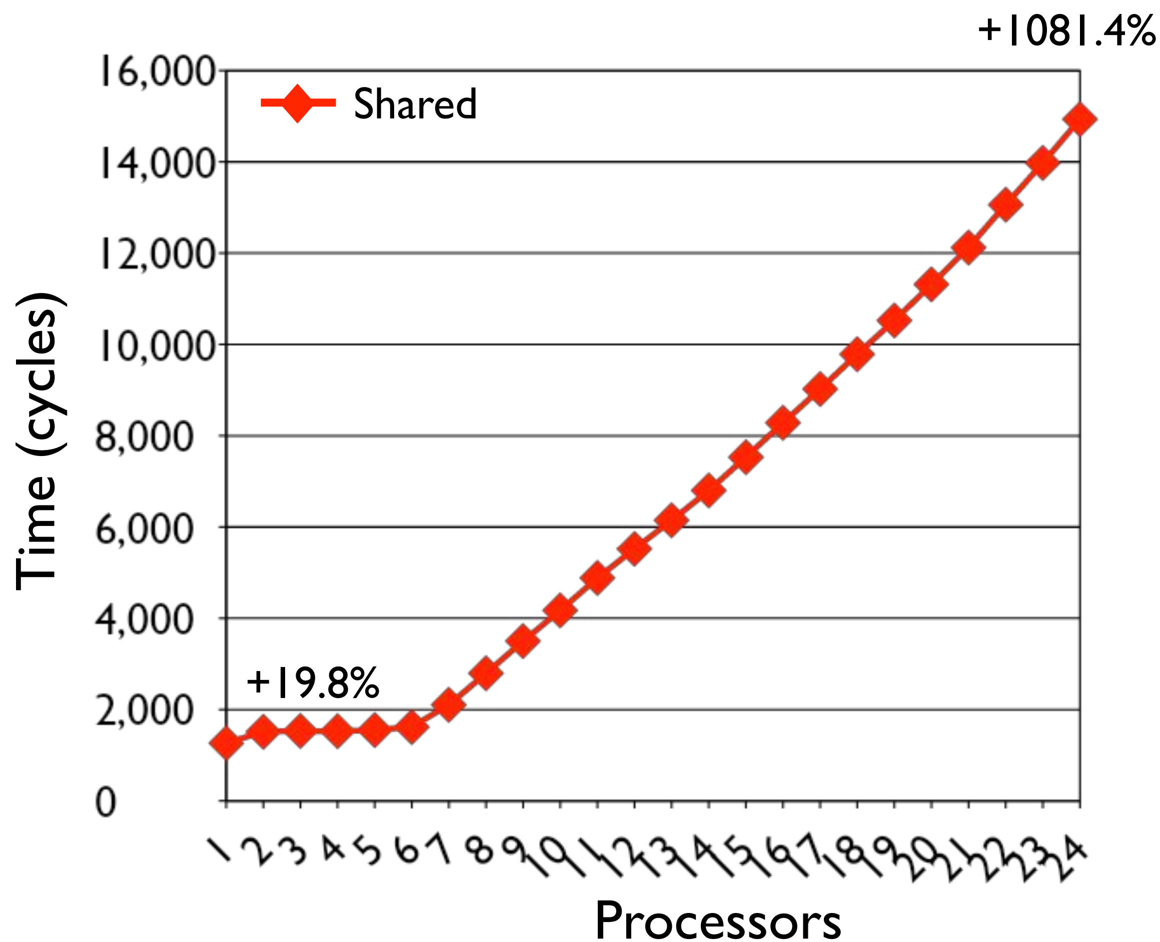
Uni-Processor atomic inc: 0.1 % of path

Concurrent SW != Scalability

Shared Counter

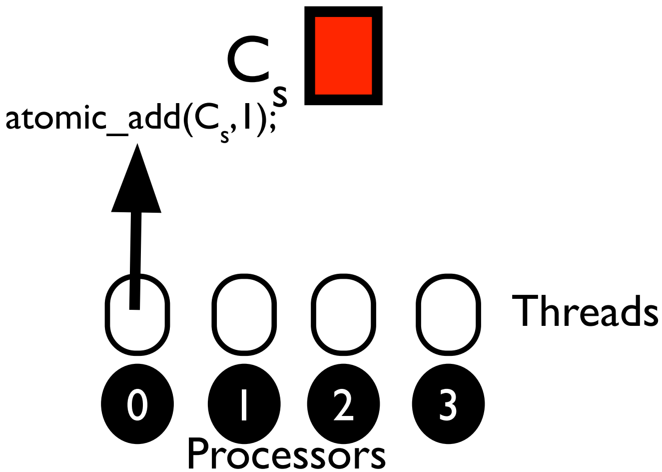


Uni-Processor atomic inc: 0.1 % of path

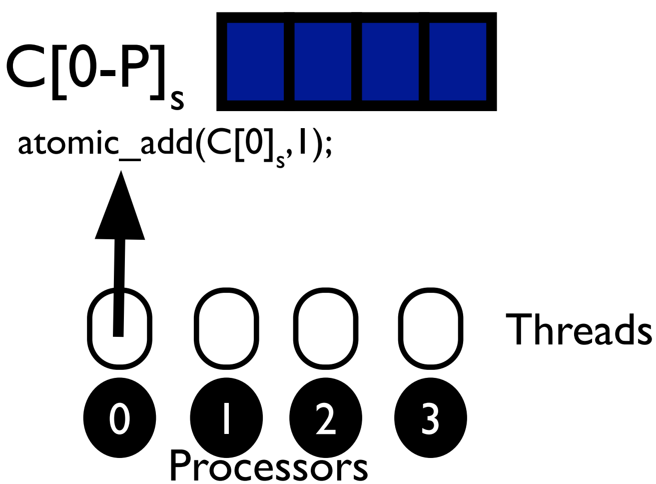


Concurrent SW != Scalability

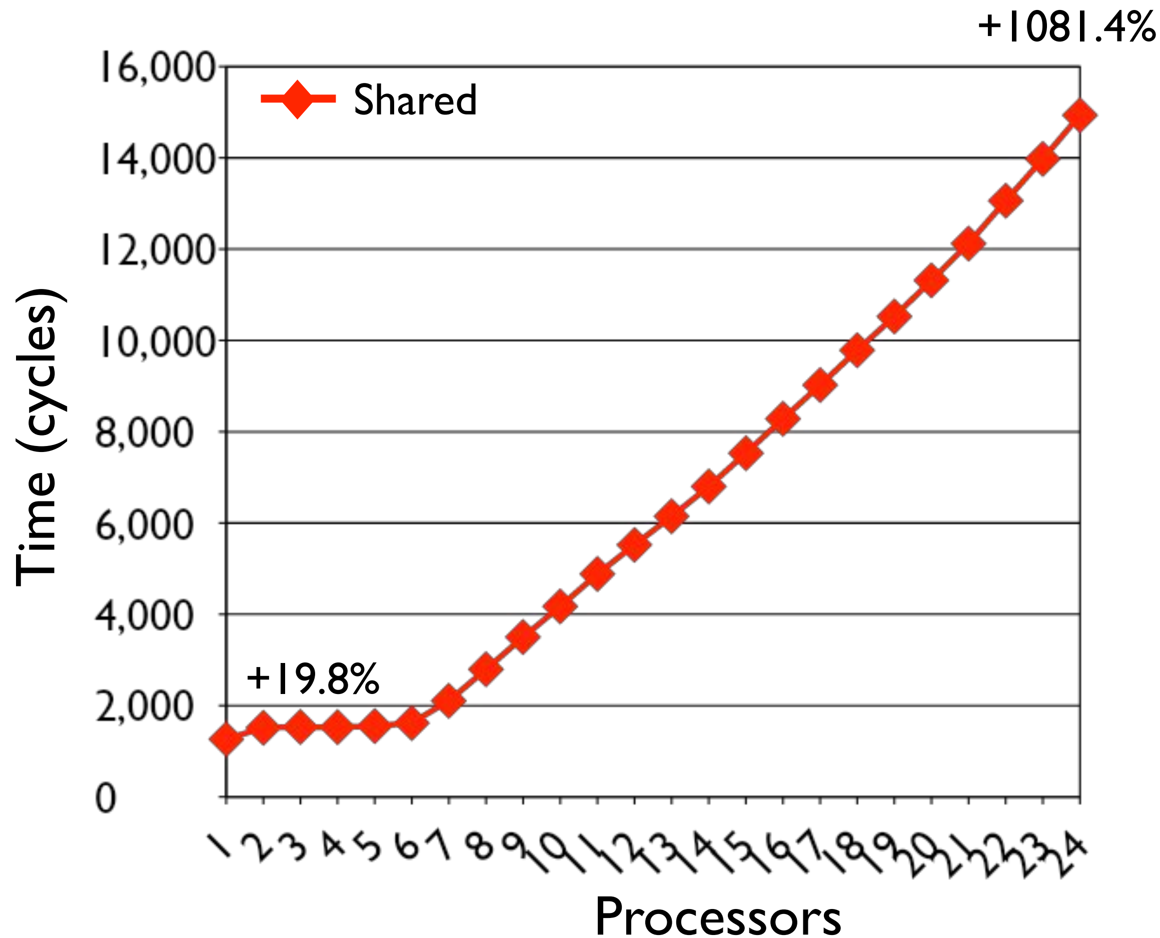
Shared Counter



Array of Counters

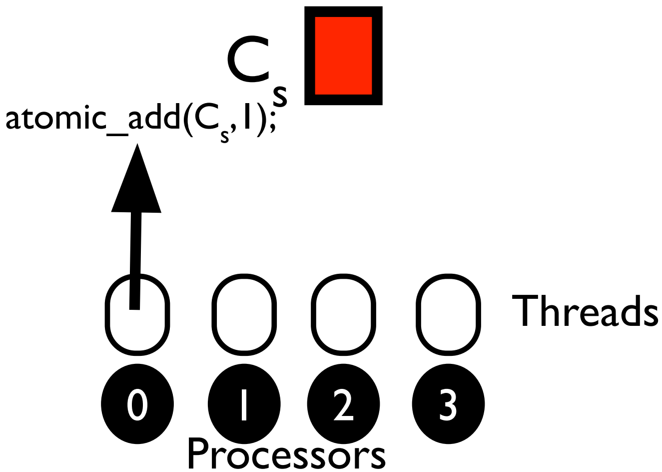


Uni-Processor atomic inc: 0.1 % of path

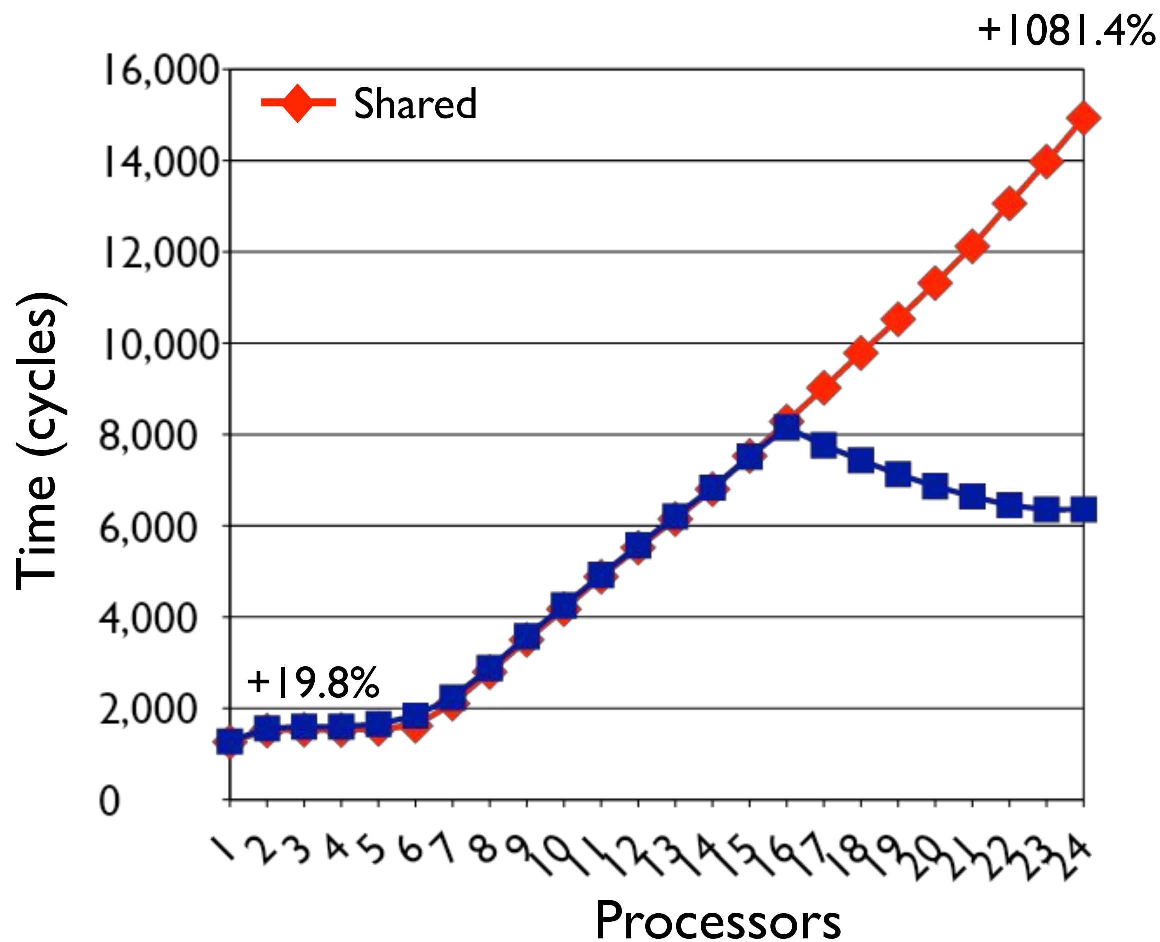


Concurrent SW != Scalability

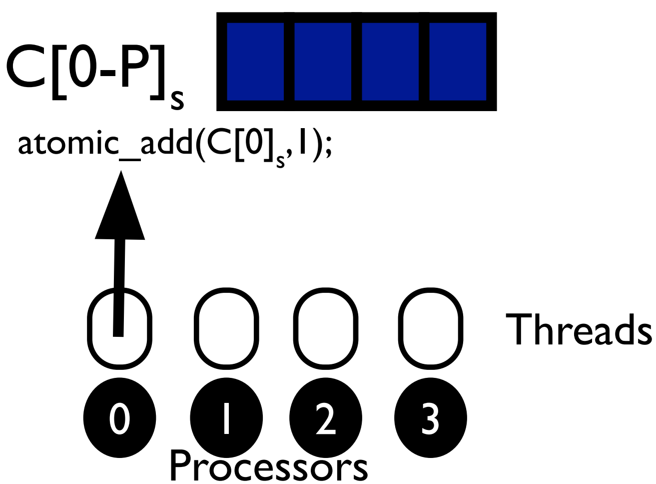
Shared Counter



Uni-Processor atomic inc: 0.1 % of path

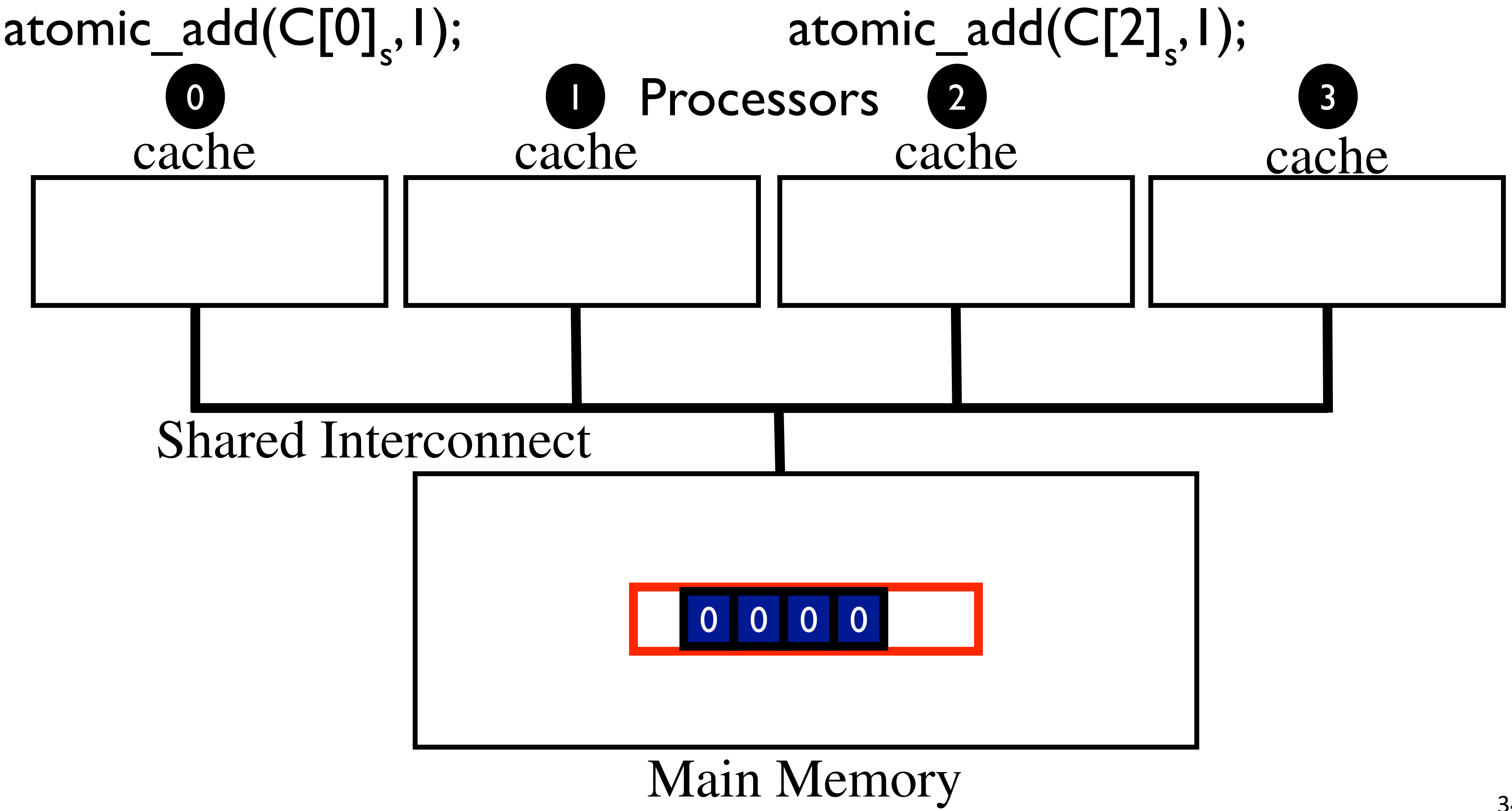


Array of Counters

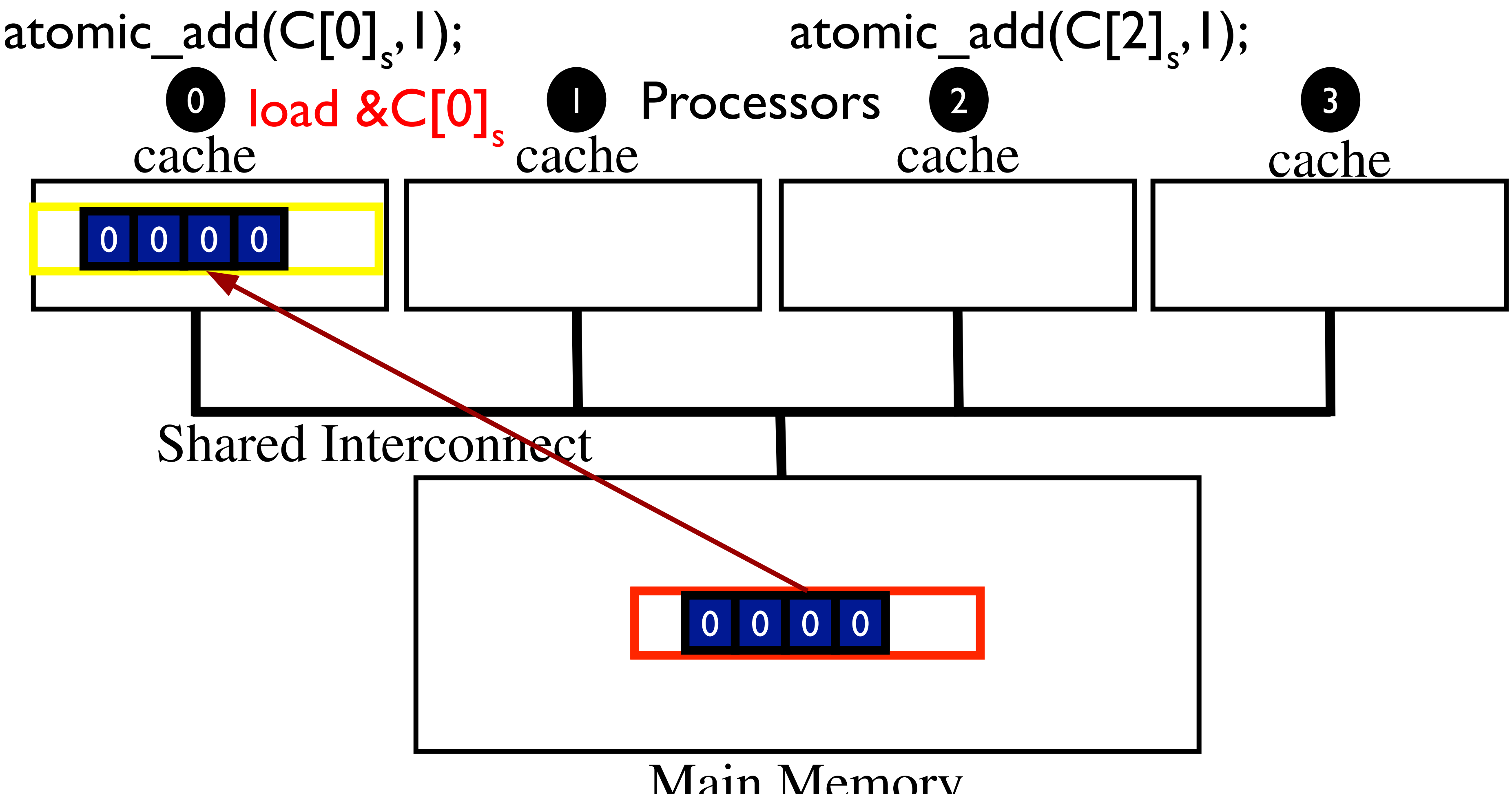


So what's the problem?

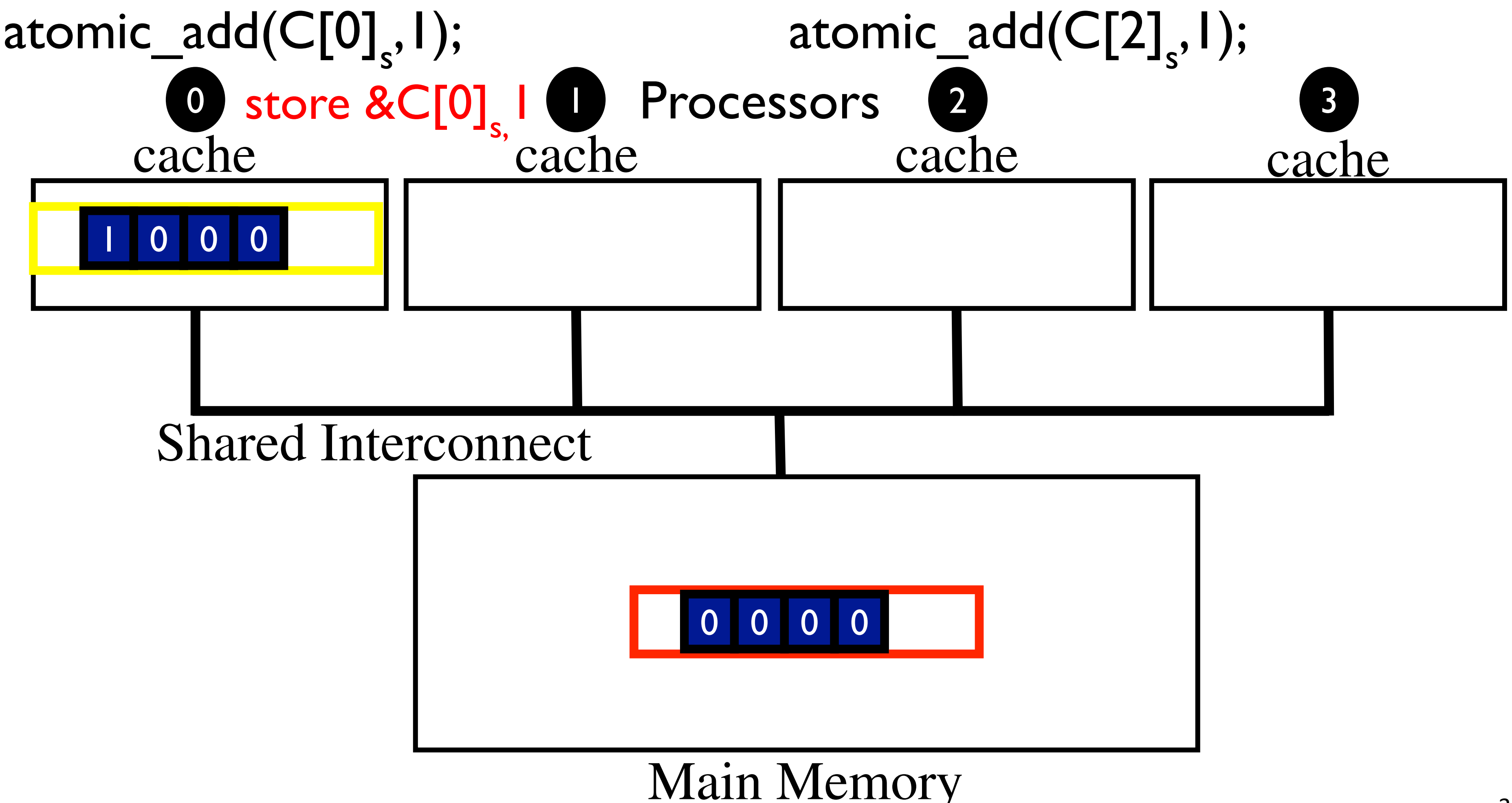
So what's the problem?



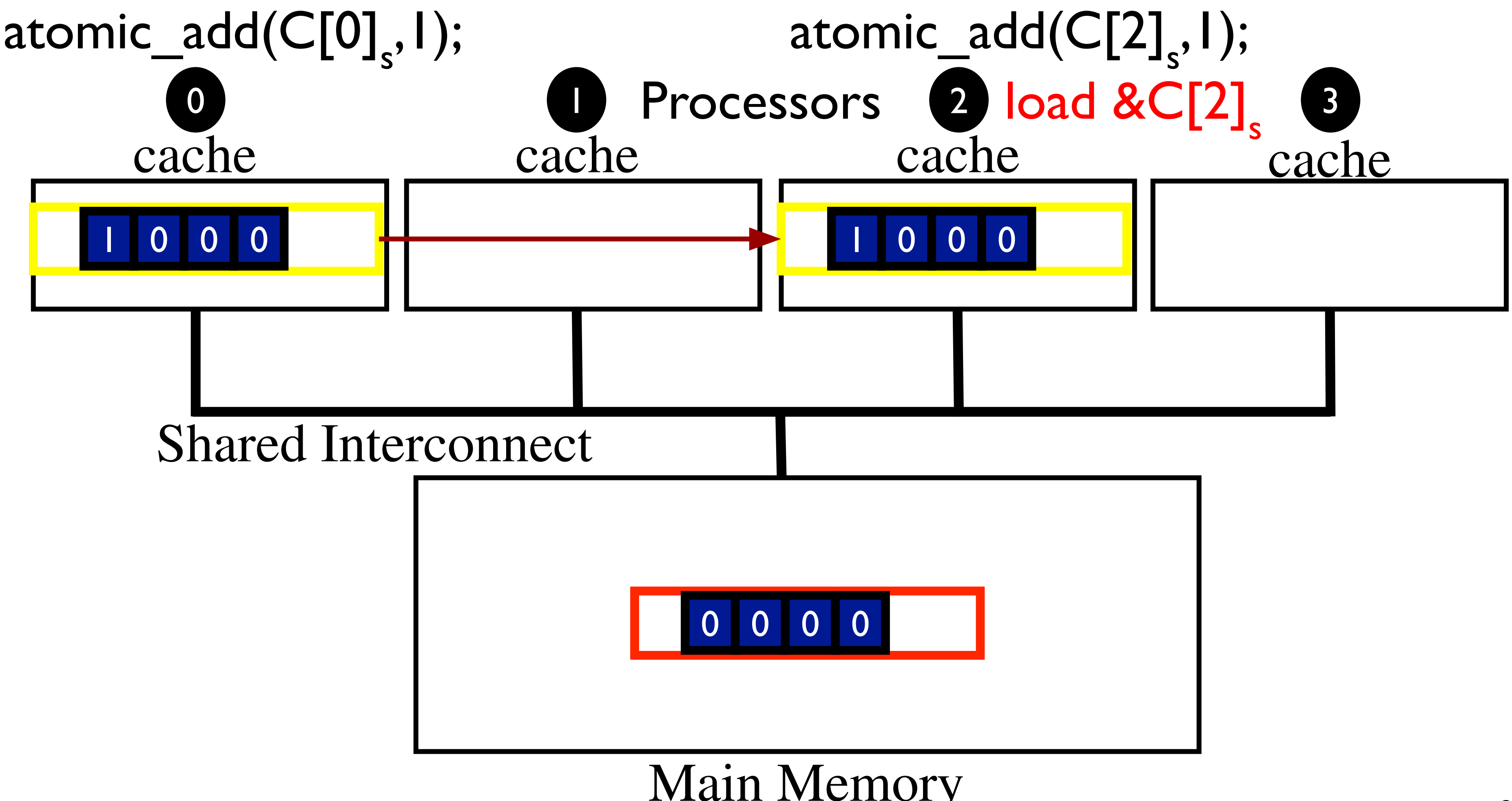
So what's the problem?



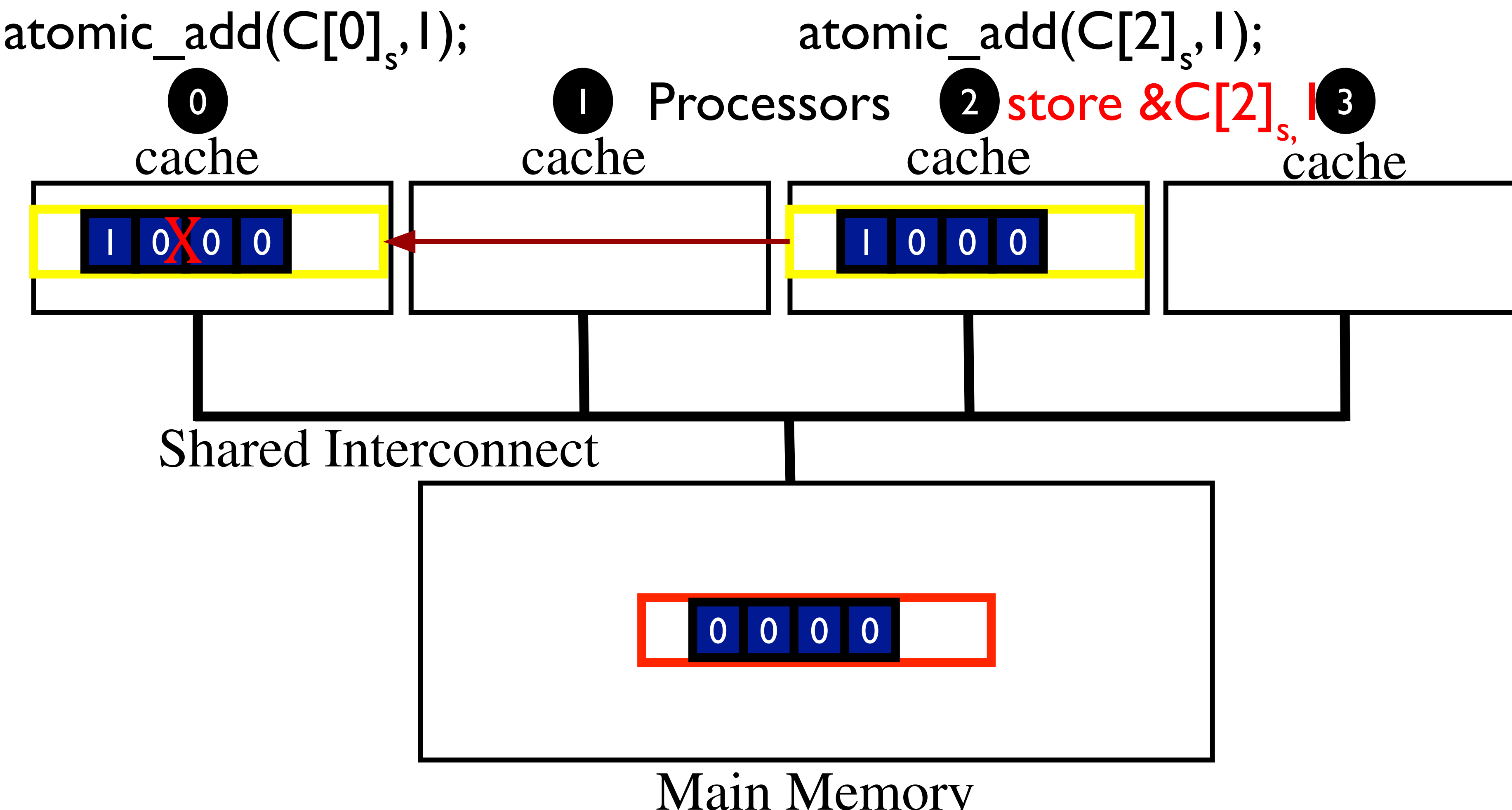
So what's the problem?



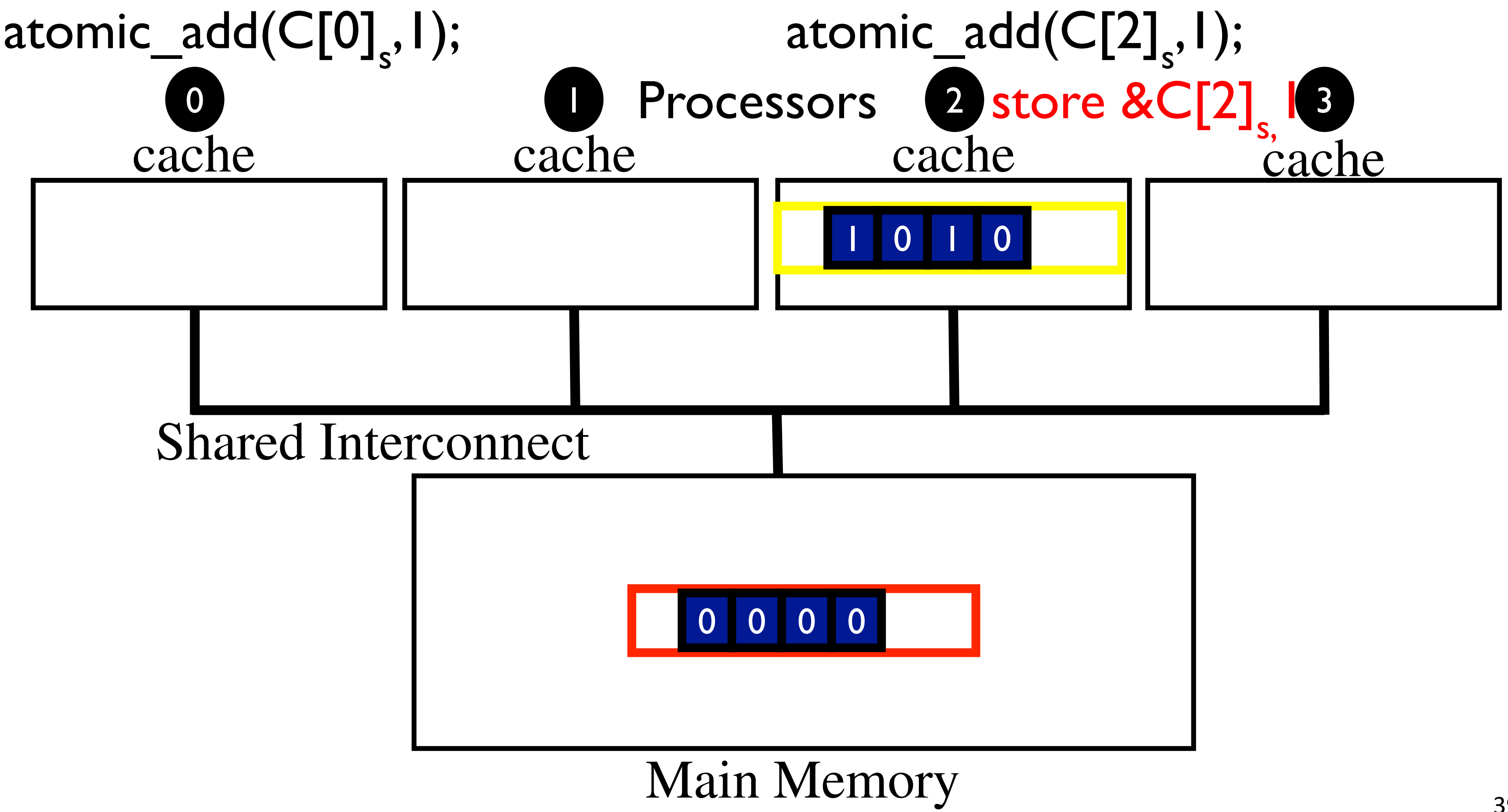
So what's the problem?



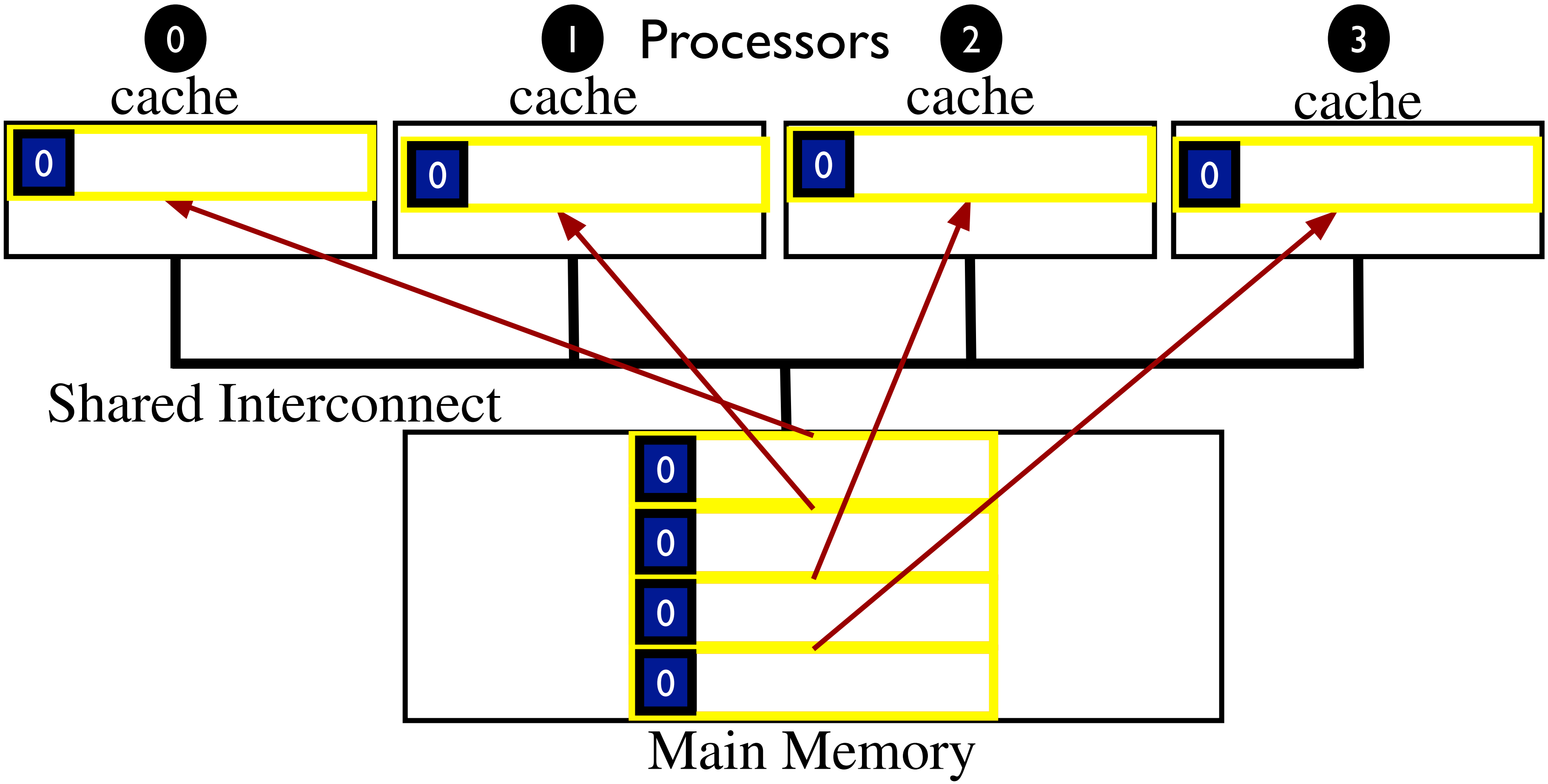
So what's the problem?



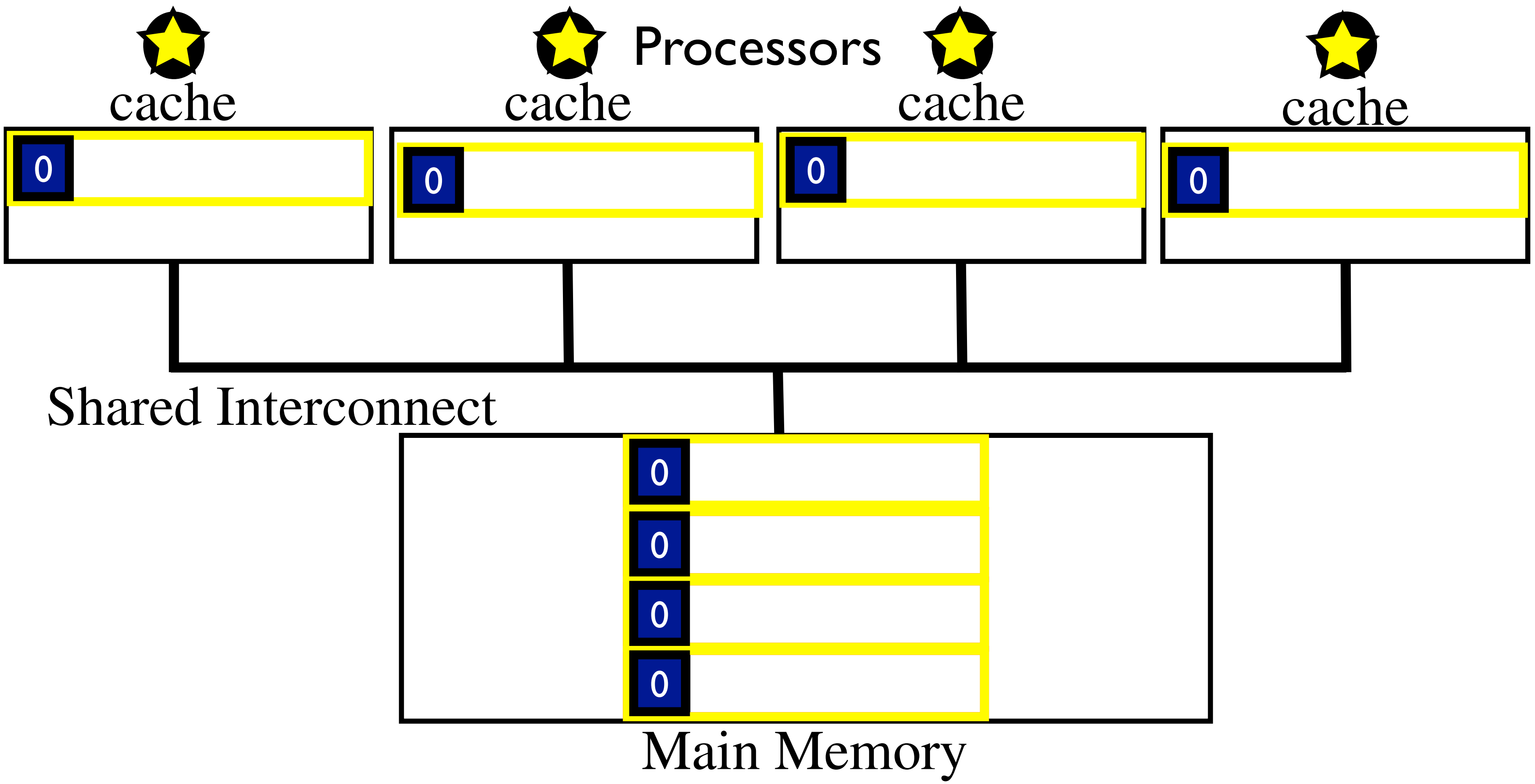
So what's the problem?



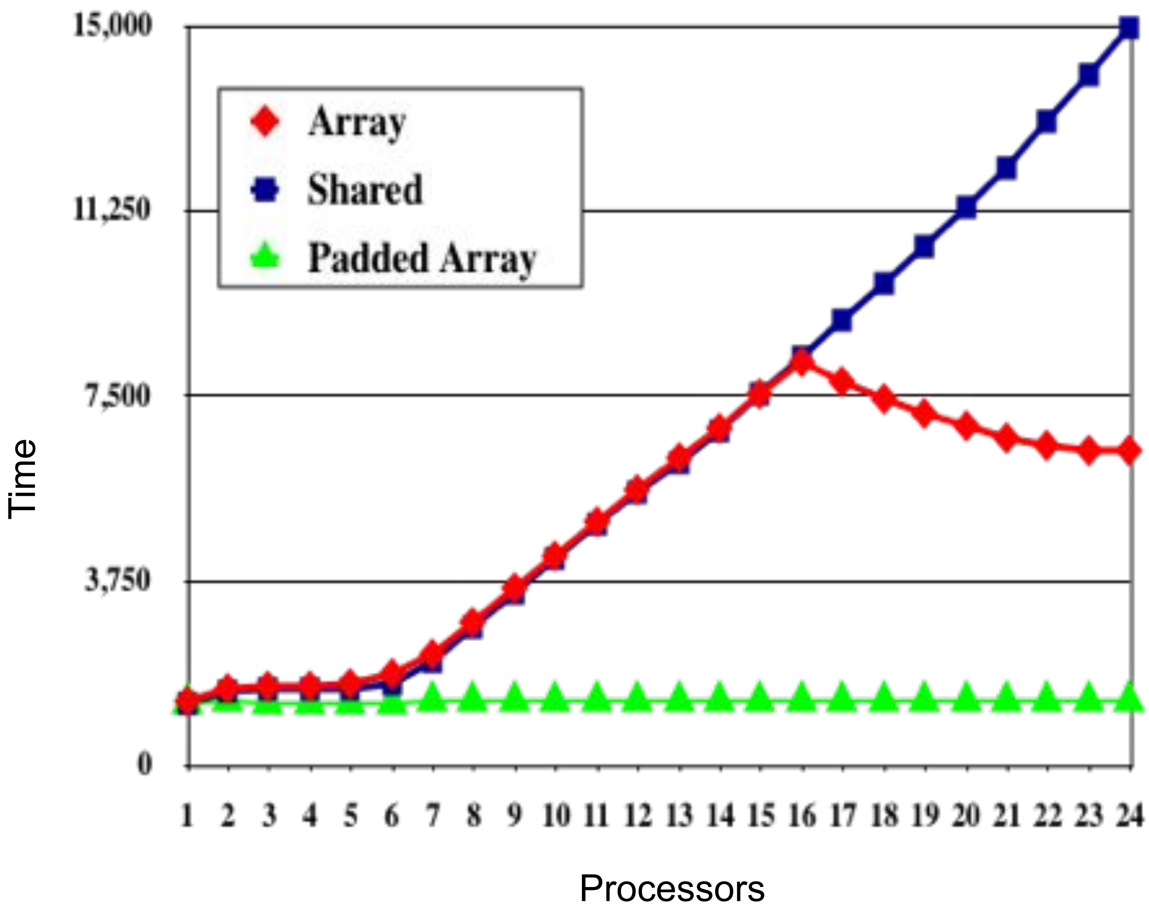
What if we pad to cache line



What if we pad to cache line



Result, the green line



False Sharing

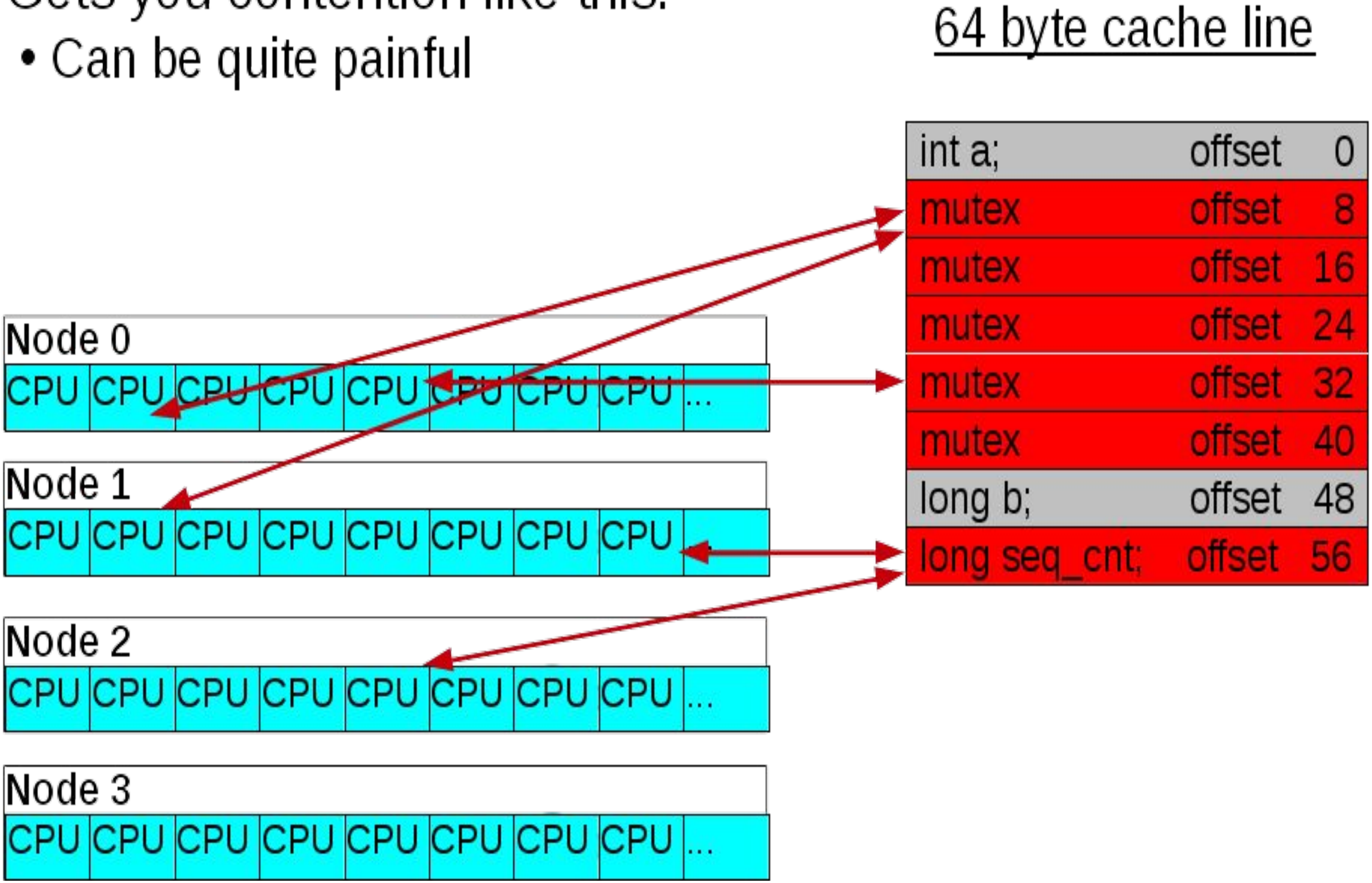
- Different threads sharing common data struct
- Different processes sharing common shared memory.

Ex: Two hotly contended data items sharing a 64-byte cacheline.

int a;	0	
pthread_mutex_t mutex1;	8	
	16	
	24	*Hot pthread mutex*
	32	
	40	
long b;	48	
long sequence_cnt;	56	*Hot variable*

Gets you contention like this:

- Can be quite painful



Split it up into two lines, with hot items in their own lines:

Hot mutex

Hot sequence counter

Cacheline 1	pthread_mutex_t mutex1;	0
		8
		16
		24
		32
	long a;	40
	long b;	48
	long cold_var;	56
Cacheline 2	long sequence_cnt;	0
With padding or cold variables	long pad1;	8
	long pad2;	16
	long pad3;	24
	long pad4;	32
	long pad5;	40
	long pad6;	48
	long pad7;	56

Future Red Hat update to perf: “c2c data sharing” tool

```
=============================================================================================
Cache                                                                                      CPU
 #   Refs    Stores    Data Address        Pid    Tid    Inst Address          Symbol     Object  Participants
=============================================================================================
0   118789  273709    0x602380            37878
     17734  136078    0x602380            37878  37878  0x401520              read_wrt_thread  a.out  0{0}
     13452  137631    0x602388            37878  37883  0x4015a0              read_wrt_thread  a.out  0{1}
     15134       0    0x6023a8            37878  37882  0x4011d7              reader_thread    a.out  1{5}
     14684       0    0x6023b0            37878  37880  0x4011d7              reader_thread    a.out  1{6}
     13864       0    0x6023b8            37878  37881  0x4011d7              reader_thread    a.out  1{7}

1       31      69    0xffff88023960df40  37878
        13      69    0xffff88023960df70  37878  ***    0xffffffff8109f8e5  update_cfs_rq_blocked_load      vmlinux  0{0,1,2};1{14,16}
        17       0    0xffff88023960df60  37878  ***    0xffffffff8109fc2e  __update_entity_load_avg_contrib  vmlinux  0{0,1,2};1{14,16}
         1       0    0xffff88023960df78  37878  37882  0xffffffff8109fc4e  __update_entity_load_avg_contrib  vmlinux  0{2}
```

<u>This shows who is contributing to the false sharing:</u>

- The hottest contended cachelines
- The process names, data addr, ip, pids, tids
- The node and CPU numbers they ran on,
- And how the cacheline is being accessed (read or write)
- Disassemble the binary to find the ip, and track back to the sources.

Techniques we use today

- Atomic operations, e.g., increment/decrement
- Per-core/per-socket locks/replication - data on different cache lines
- multiple reader and single writer locks
- Fine grained locks embedded in data/objects: cache-miss gets you the data and lock
  - Different data structures on allocated on different cache lines
- Scalable locks:
  - Ticketed spinlocks
  - MCS locks
    - enqueue blocked threads on list
    - each spins on a local variable
    - communication one-to-one
- Read Copy Update (RCU) synchronization
- lockless code implementation(if possible)
  - Example: Linux Kernel memory allocator

Ticketed spinlocks

- Problem:
  - starvation caused by locks allocated in preferred cache/memory.

    CPUs close to the memory the lock resides in will always get spinlock first.
- Solution: round-robin lock acquisition based on CPU ID.
  - spinlock():
    - if bitmask is clear

      acquire spinlock

      set CPUID bit in per-lock bitmask and spin
  - spinunlock()
    - clear CPUID bit in per-lock bitmask

      if bitmask is not clear

      grant lock to next CPU set in bitmask.

      else release spinlock

*Simple and stops starvation but is not fair.

MCS/queued spinlocks

- Problem:
  - ticketed spinlocks are unfair, round-robin rather than first-come-first-serve.
- Solution: build FIFO queue of CPUs trying to acquire lock.
  - spinlock():
    - if queue-is-empty

      acquire spinlock

      else

      insert CPU in per-lock queue and just spin
  - spinunlock()
    - if !queue-is-empty

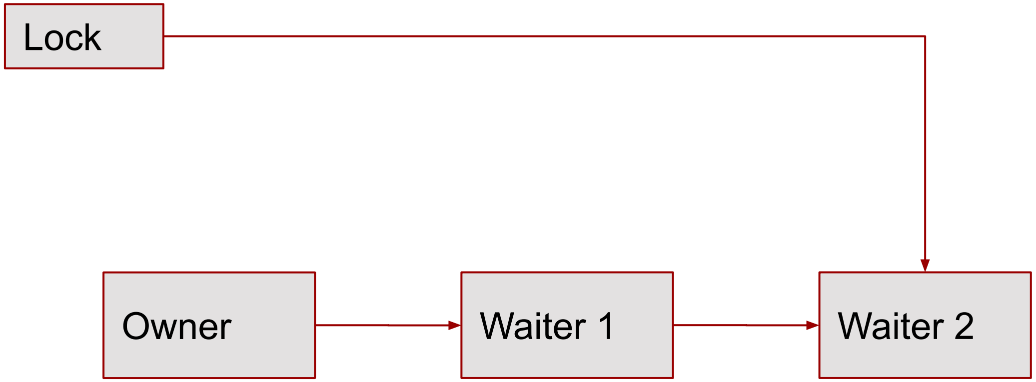
      pass spinlock to first CPU in queue

      else

      release spinlock

*Complex but stops starvation with fairness.

MCS spinlock



Key idea RCU

- Let multiple readers access data concurrently with one writer
- Writer copies data, modifies, atomically inserts modification
- Readers see consistent version,
  - either before writer or after writer
- Data not deleted until guaranteed all readers done

Avoiding Locks: Read-Copy-Update (1)

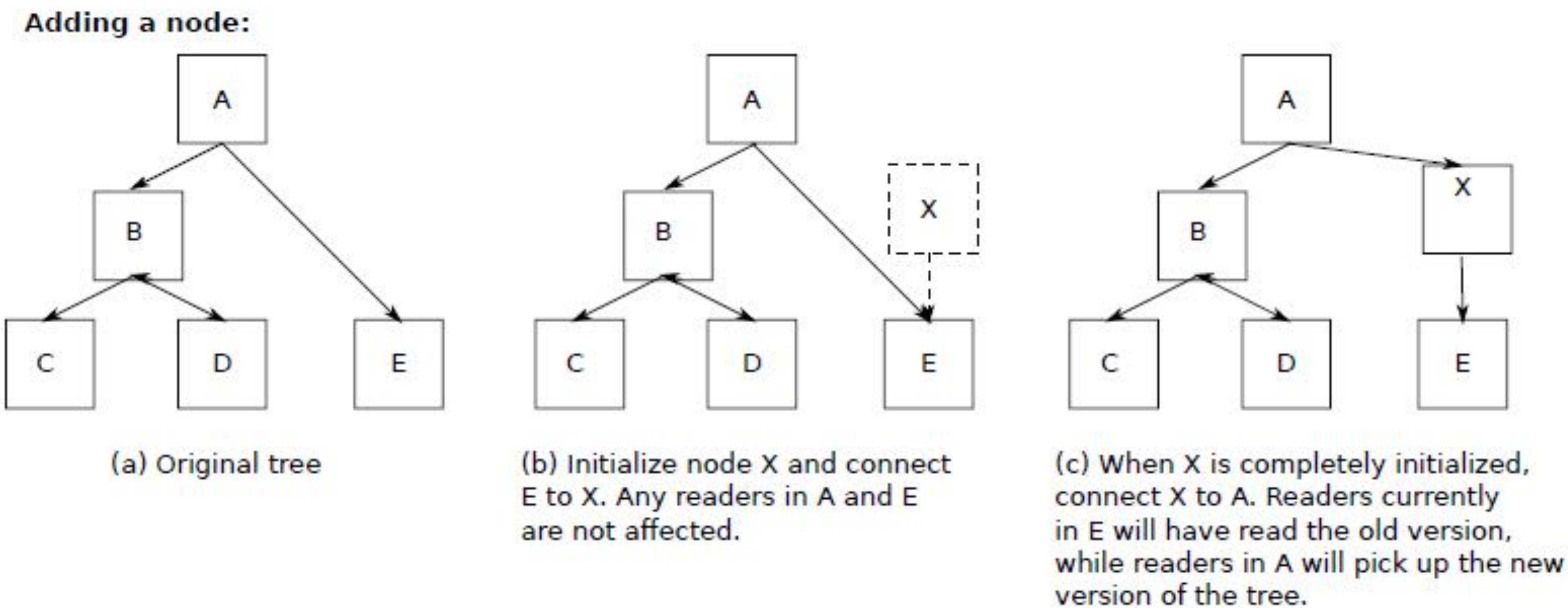
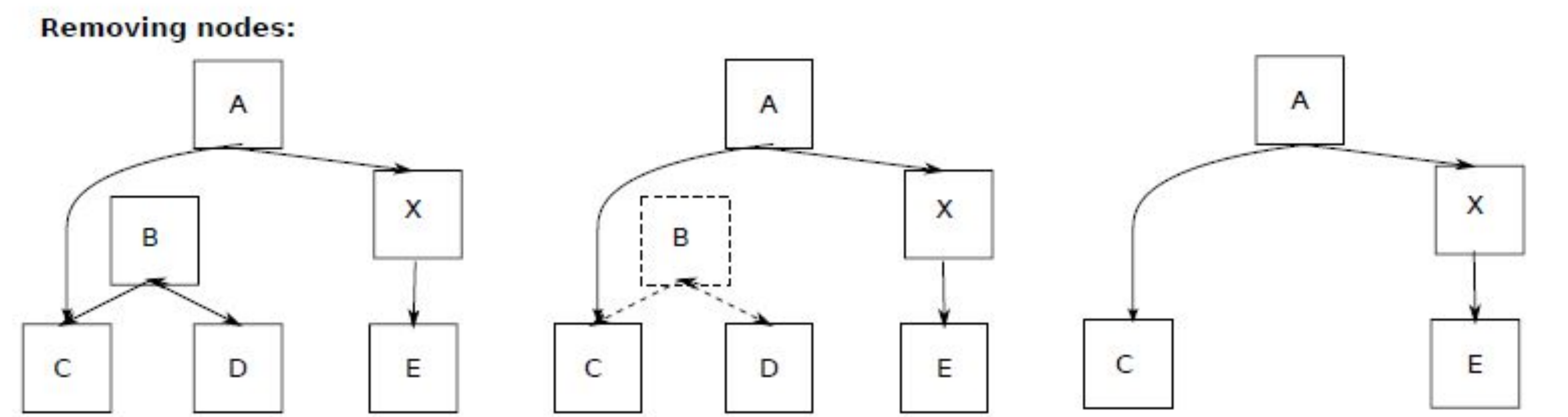


Figure 2-38. Read-Copy-Update: inserting a node in the tree and then removing a branch—all without locks

Avoiding Locks: Read-Copy-Update (2)



(d) Decouple B from A. Note that there may still be readers in B. All readers in B will see the old version of the tree, while all readers currently in A will see the new version.

(e) Wait until we are sure that all readers have left B and C. These nodes cannot be accessed by anymore.

(f) Now we can safely remove B and D

Figure 2-38. Read-Copy-Update: inserting a node in the tree and then removing a branch—all without locks

A bit of history

- Introduced concurrently in
  - Sequent's large MP ptx system
  - K42/Tornado systems Toronto/IBM
- We presented Ottawa Linux Symposium 2001
- Immediately started getting wide use in Linux
- Forced Paul to make it his PhD
- Subject of $3B SCO lawsuit in 2003
- oops
- Paul finally could release his PhD in 2004

Using RCU pervasively

- You already saw deadlock (topic of wed lecture) in dining philosophers
- We avoid deadlock in OS by acquiring locks in order
- Most ordering problems turn out to be existence locks, e.g., process doesn't go away while you are handling a page fault
- K42/Tornado had no global locks, no locking hierarchy, ... used RCU for everything...