Software Performance Engineering

LECTURE 16

Nondeterministic Parallel Programming

Xuhao Chen

November 20, 2025

Determinism

- Definition. A program is deterministic on a given input if every memory location is updated with the same sequence of values in every execution
  - The program always behaves the same way.
  - Two different memory locations may be updated in different orders, but each location always sees the same sequence of updates.

Advantage: DEBUGGING!

A Cilk program with no determinacy races is deterministic.

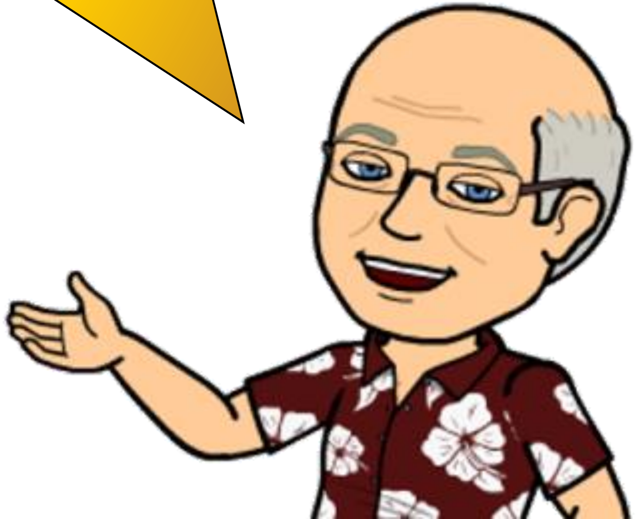
- Cilksan can help you avoid nondeterminacy bugs.

Golden Rule of Parallel Programming

Never write nondeterministic parallel programs

They can exhibit anomalous behaviors, and it's hard to debug them.

But what if determinism inhibits performance?



Silver Rule of Parallel Programming

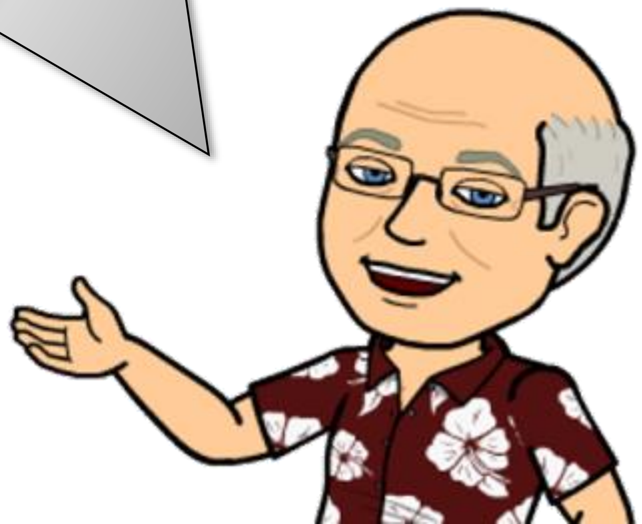
Never write nondeterministic parallel programs

— but if you must* —

always devise a test strategy
to manage the nondeterminism!

Typical test strategies

- Turn off nondeterminism.
- Encapsulate nondeterminism.
- Substitute a deterministic alternative.
- Use analysis tools.



---

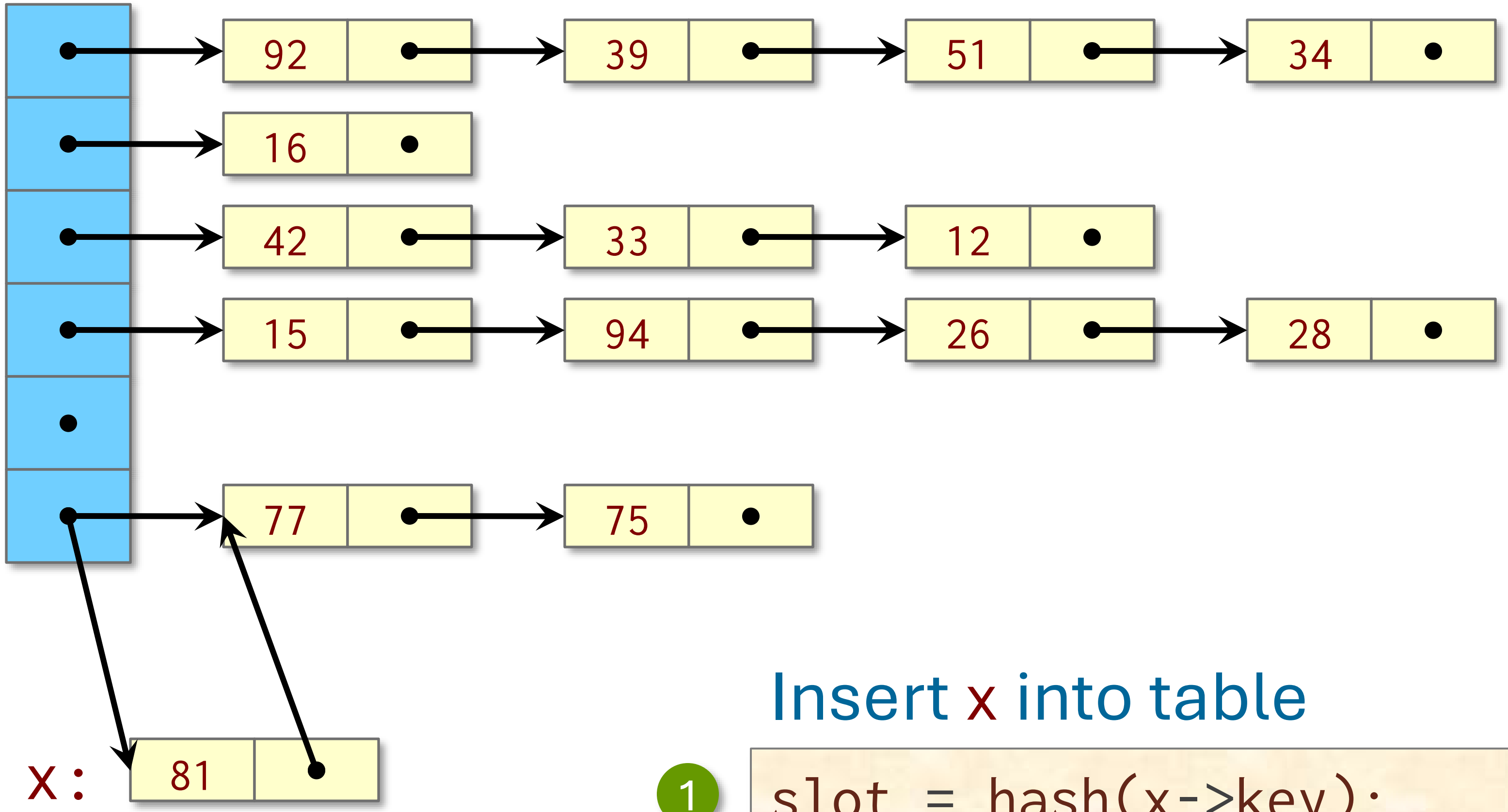
*e.g., for performance.

ENTER AT YOUR OWN RISK

Outline

- **Atomicity & Mutual Exclusion**
- Implementation of Mutexes
- Locking Anomaly: Contention
- Locking Anomaly: Deadlock
- Locking Anomaly: Convoying

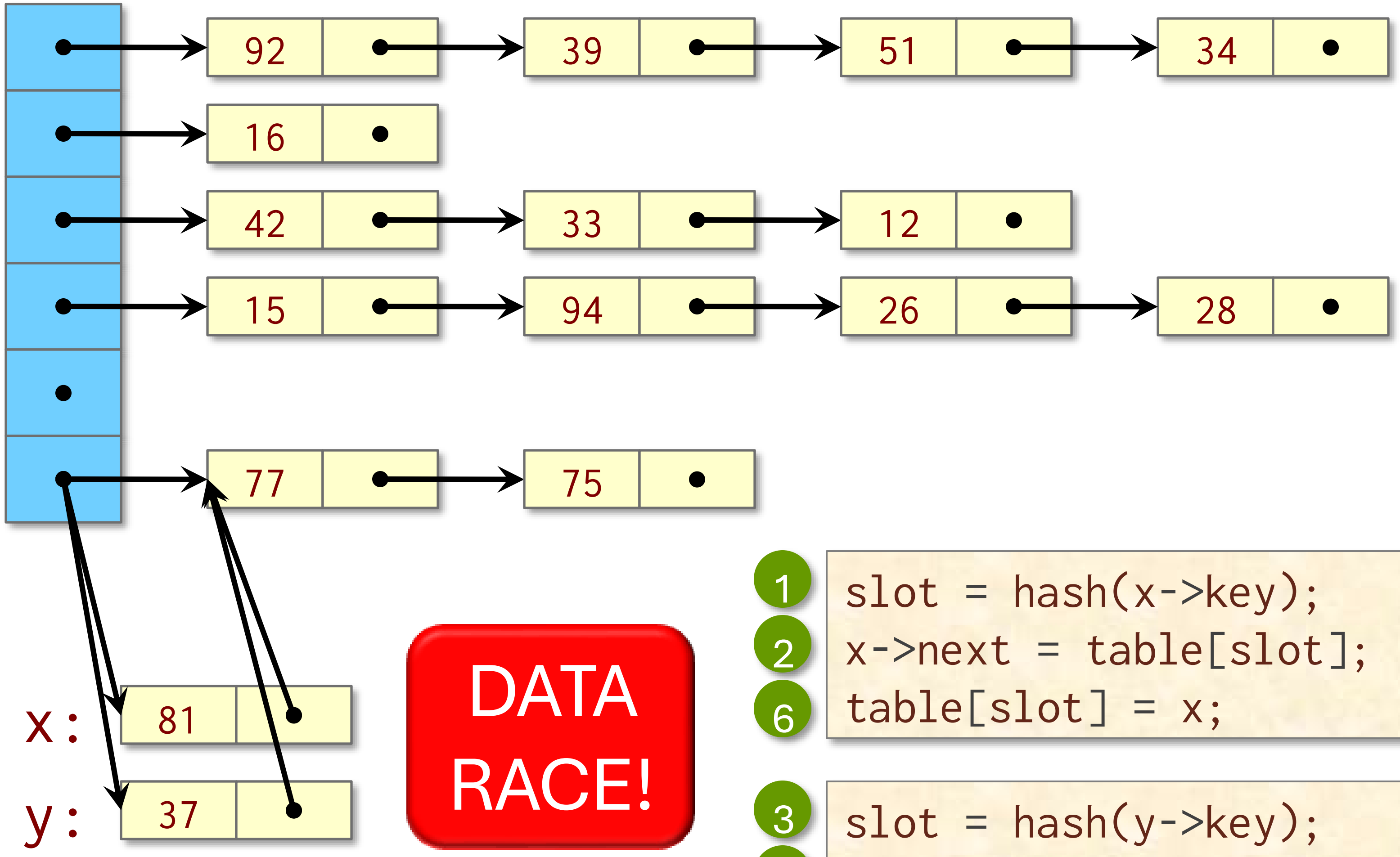
Example: Hash Table



Insert x into table

```
slot = hash(x->key);
x->next = table[slot];
table[slot] = x;
```

Concurrent Hash Table




```
1 slot = hash(x->key);
2 x->next = table[slot];
6 table[slot] = x;
```

```
3 slot = hash(y->key);
4 y->next = table[slot];
5 table[slot] = y;
```

Atomicity and Mutexes

- **Definition.** A sequence of instructions is *atomic* if the rest of the system never views them as partially executed.
  - At any moment, either no instructions in the sequence have executed or all of them have executed
- **Definition.** A *critical section* is a piece of code that accesses a shared data structure which must not be accessed by two or more strands at the same time (*mutual exclusion*).
- **Definition.** A *mutex* is an object with `lock()` and `unlock()` functions.
  - An attempt by a strand to lock an already locked mutex causes that strand to *block* (i.e., wait) until the mutex is unlocked

Concurrent Hash Table

- Modified hash-table code
  - Introduce a mutex L
  - Lock L before executing the critical section
  - Unlock L after executing the critical section.

critical section

```
slot = hash(x->key);
lock(&L);
  x->next = table[slot];
  table[slot] = x;
unlock(&L);
```

Performance problem

Only one strand can insert into the hash table at a time.

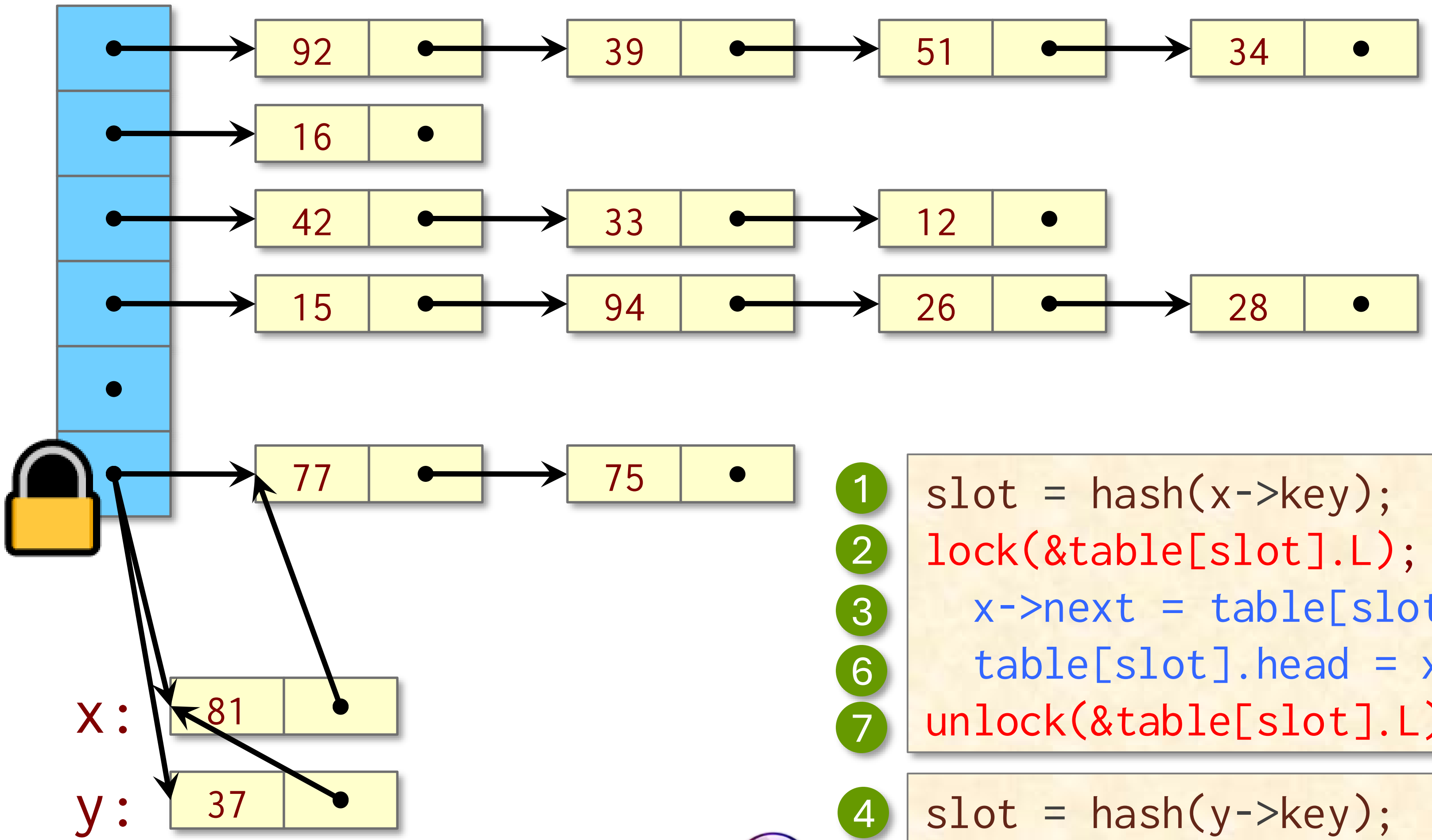
Concurrent Hash Table II

- Idea: One mutex per slot
  - Make each slot a `struct` with a mutex L and a pointer head to the slot contents

critical section

```
slot = hash(x->key);
lock(&table[slot].L);
  x->next = table[slot].head;
  table[slot].head = x;
unlock(&table[slot].L);
```

Concurrent Hash Table with Mutexes




```
1  slot = hash(x->key);
2  lock(&table[slot].L);
3    x->next = table[slot].head;
6    table[slot].head = x;
7  unlock(&table[slot].L);
```

```
4  slot = hash(y->key);
5  lock(&table[slot].L);
8    y->next = table[slot].head;
9    table[slot].head = y;
10 unlock(&table[slot].L);
```

Q: Is this table deterministic?

NO!

Recall: Determinacy Races

Definition. A determinacy race occurs when two logically parallel instructions access the same memory location and at least one of the instructions performs a write.

- A program execution with no determinacy races means that the program is deterministic on that input.
- The program always behaves the same on that input, no matter how it is scheduled and executed.
- If a determinacy race exists in an ostensibly deterministic program (e.g., a program with no mutexes), Cilksan guarantees to find such a race

Data Races

Definition. A data race occurs when two logically parallel strands holding no locks in common access the same memory location and at least one of the strands performs a write.

Although data-race-free programs obey atomicity constraints, they can still be nondeterministic, because acquiring a lock can cause a determinacy race with another lock acquisition.

WARNING: Codes that use locks are nondeterministic by intention, and they invalidate Cilksan’s guarantee.

No Data Races ≠ No Bugs

Example

```
slot = hash(x->key);

lock(&table[slot].L);
  x->next = table[slot].head;
unlock(&table[slot].L);

lock(&table[slot].L);
  table[slot].head = x;
unlock(&table[slot].L);
```

Nevertheless, the presence of mutexes and the absence of data races at least means that the programmer thought about the issue.

"Benign" Races

Example: Identify the set of digits in an array.

A: 4, 1, 0, 4, 3, 3, 4, 6, 1, 9, 1, 9, 6, 6, 6, 3, 4

```
for (int i = 0; i < 10; ++i) {
  digits[i] = 0;
}
cilk_for (int i = 0; i < N; ++i) {
  digits[A[i]] = 1;   // benign race
}
```

digits:

1	1	0	1	1	0	1	0	0	1
0	1	2	3	4	5	6	7	8	9

CAUTION: This code only works correctly if the hardware writes the array elements atomically (e.g., it may race on byte values for some architectures).

“Benign” Races

Example: Identify the set of digits in an array.

A: 4, 1, 0, 4, 3, 3, 4, 6, 1, 9, 1, 9, 6, 6, 6, 3, 4

```
for (int i = 0; i < 10; ++i) {
  digits[i] = 0;
}
cilk_for (int i = 0; i < N; ++i) {
  digits[A[i]] = 1;   // benign race
}
```

digits:

1	1	0	1	1	0	1	0	0	1
0	1	2	3	4	5	6	7	8	9

- Cilksan allows turn off race detection for intentional races: dangerous but practical
- Better solutions exist, e.g., fake locks in Intel’s Cilkscreen (see Intel Cilk Plus User's Guide)

Outline

- Atomicity & Mutual Exclusion
- **Implementation of Mutexes**
- Locking Anomaly: Contention
- Locking Anomaly: Deadlock
- Locking Anomaly: Convoying

Properties of Mutexes

- Yielding/spinning
  - A yielding mutex returns control to the operating system when it blocks.
  - A spinning mutex consumes processor cycles while blocked

- Reentrant/nonreentrant
  - A reentrant mutex allows a thread that is already holding a lock to acquire it again.
  - A nonreentrant mutex deadlocks if the thread attempts to reacquire a mutex it already holds

- Fair/unfair
  - An unfair mutex lets any blocked thread go next.
  - One type of fair mutex allows the thread that has been waiting the longest in first after unlock (it places blocked threads on a FIFO queue)

Spinning Mutex

```
Spin_Mutex:
      cmp 0, mutex ; Check if *mutex is free
      je Get_Mutex
      pause ; x86 hack to unconfuse pipeline
      jmp Spin_Mutex
Get_Mutex:
      mov 1, %eax
      xchg mutex, %eax ; Try to get mutex
      cmp 0, %eax ; Test if successful
      jne Spin_Mutex
Critical_Section:
      <critical-section code>
      mov 0, mutex ; Release mutex
```

Any performance issue?

Key property: **xchg** is an atomic exchange operation.

Yielding Mutex

```
Spin_Mutex:
      cmp 0, mutex ; Check if *mutex is free
      je Get_Mutex
      call thread_yield ; Yield the current quantum
      jmp Spin_Mutex
Get_Mutex:
      mov 1, %eax
      xchg mutex, %eax ; Try to get mutex
      cmp 0, %eax ; Test if successful
      jne Spin_Mutex
Critical_Section:
      <critical-section code>
      mov 0, mutex ; Release mutex
```

Is this always a win?

Competitive Mutex

Competing goals:

- To claim mutex soon after it is released
- To behave nicely and waste few cycles

IDEA: Spin for a while, and then yield.

How long to spin?

As long as a context switch takes → you wait no more than 2× the optimal time

- If the mutex is released while spinning, optimal.
- If the mutex is released after yield, ≤ 2× optimal.

Randomized algorithm [KMMO94]

A clever randomized algorithm can achieve a competitive ratio of $e/(e-1) \approx 1.58$

Outline

- Atomicity & Mutual Exclusion
- Implementation of Mutexes
- **Locking Anomaly: Contention**
- Locking Anomaly: Deadlock
- Locking Anomaly: Convoying

Summing Example

```c
int compute(const el_t *v);
const size_t n = 1000000;
extern el_t myArray[n];

int main() {
  int result = 0;
  for (size_t i = 0; i < n; ++i) {
    result += compute(&myArray[i]);
  }
  printf("The result is: %d\n", result);

  return 0;
}
```

Summing Example in Cilk

```
int compute(const el_t *v);
const size_t n = 1000000;
extern el_t myArray[n];

int main() {
  int result = 0;
  cilk_for (size_t i = 0; i < n; ++i) {
    result += compute(&myArray[i]);
  }
  printf("The result is: %d\n", result);

  return 0;
}
```

Assume $\Theta(1)$ work

Race!

Should we use a spin mutex or a yield mutex?

Work/span theory

$T_1(n) = \Theta(n)$

$T_\infty(n) = \Theta(\lg n)$

$T_P(n) = O(n/P + \lg n)$

Mutex Solution

```
#include <pthread.h>
int compute(const el_t *v);
const size_t n = 1000000;
extern el_t myArray[n];

int main() {
  int result = 0;
  pthread_spinlock_t slock;
  pthread_spin_init(&slock, 0);
  cilk_for (size_t i = 0; i < n; ++i) {
    pthread_spin_lock(&slock);
    result += compute(&myArray[i]);
    pthread_spin_unlock(&slock);
  }
  printf("The result is: %d\n", result);

  return 0;
}
```

lock

Sequential Bottleneck
⇒ no parallelism!

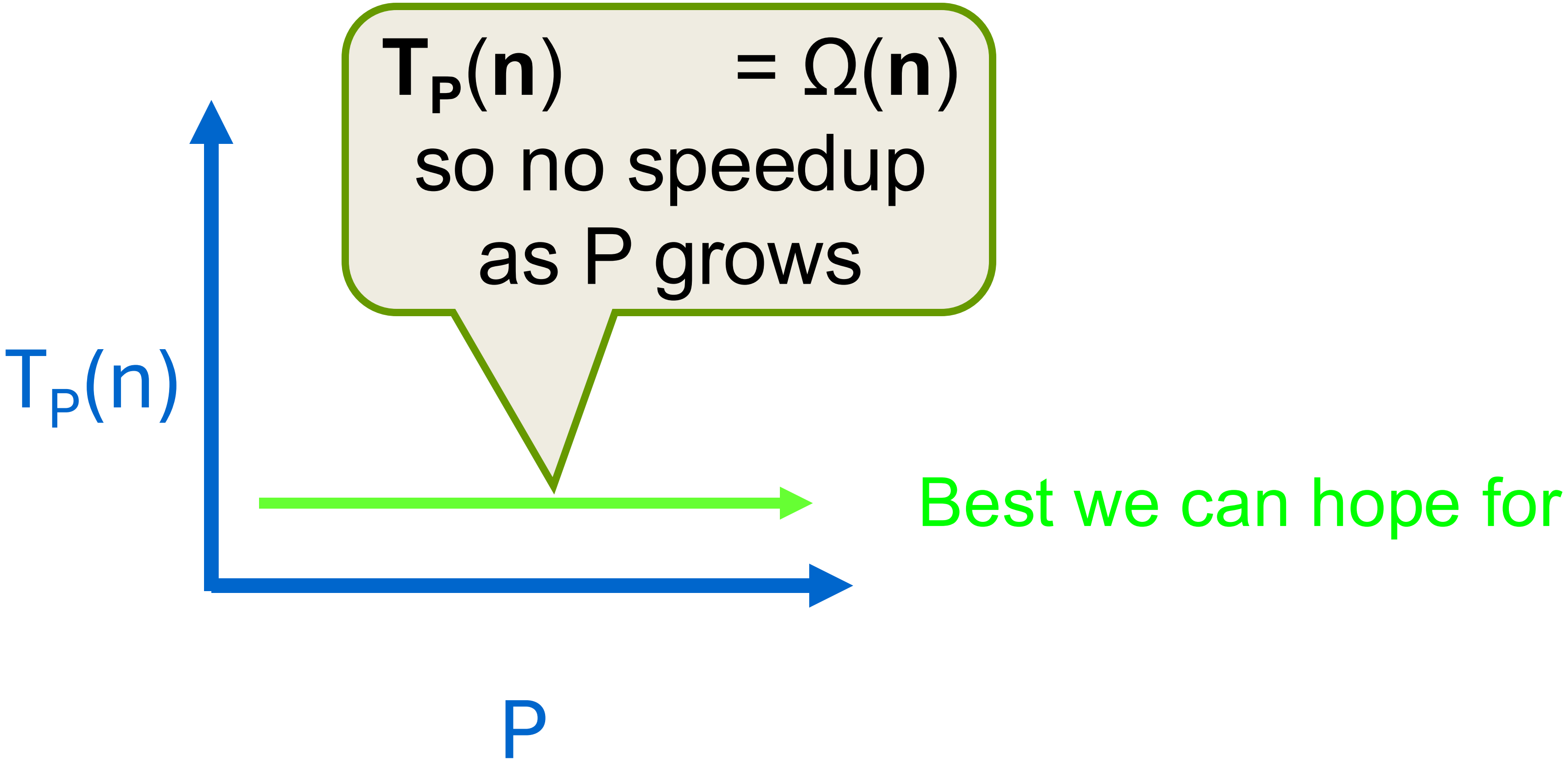
Bottleneck

$T_1(n) = \Theta(n)$

$T_\infty(n) = \Theta(\lg n)$

$T_P(n) = \Omega(n)$

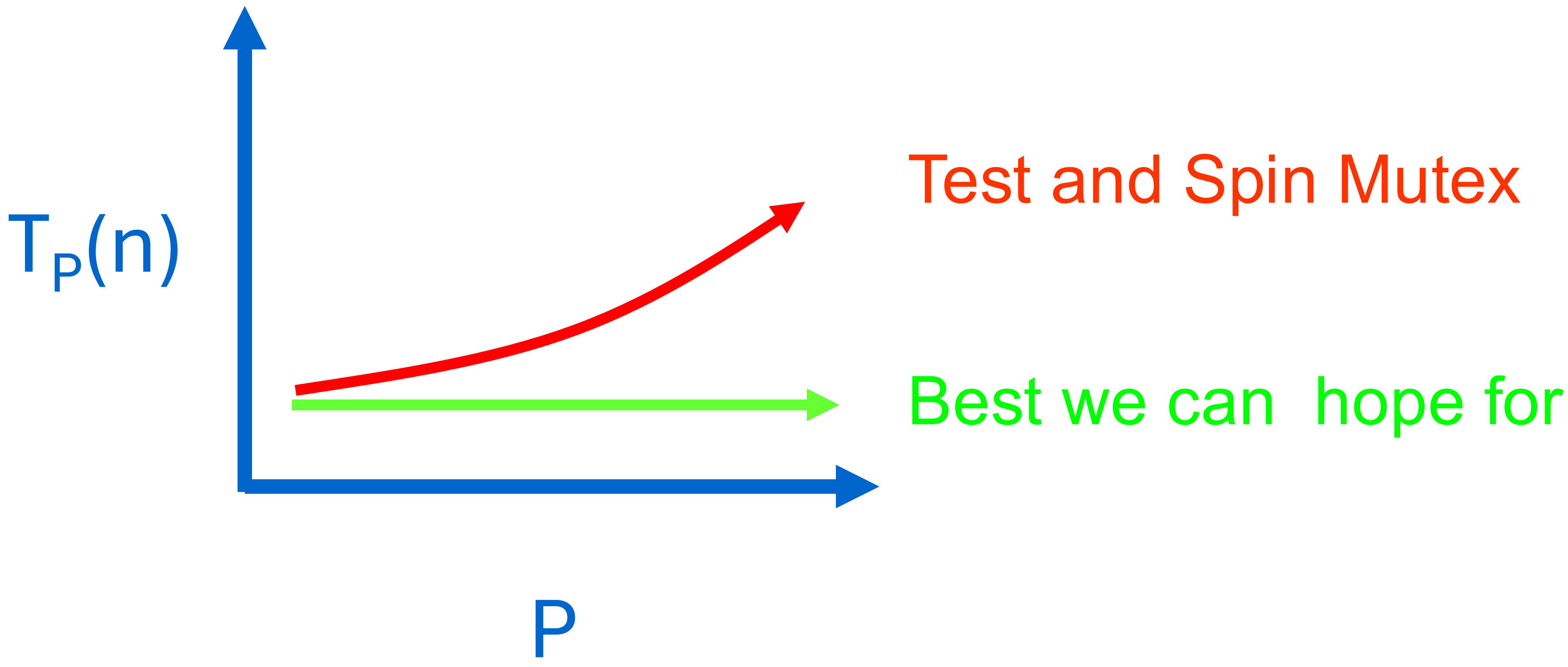
Sequential Bottleneck



Test and Spin Mutex

```
Spin_Mutex:
      cmp 0, mutex ; Check if *mutex is free
      je Get_Mutex
      pause ; x86 hack to unconfuse pipeline
      jmp Spin_Mutex
Get_Mutex:
      mov 1, %eax
      xchg mutex, %eax ; Try to get mutex
      cmp 0, %eax ; Test if successful
      jne Spin_Mutex
Critical_Section:
      <critical-section code>
      mov 0, mutex ; Release mutex
```

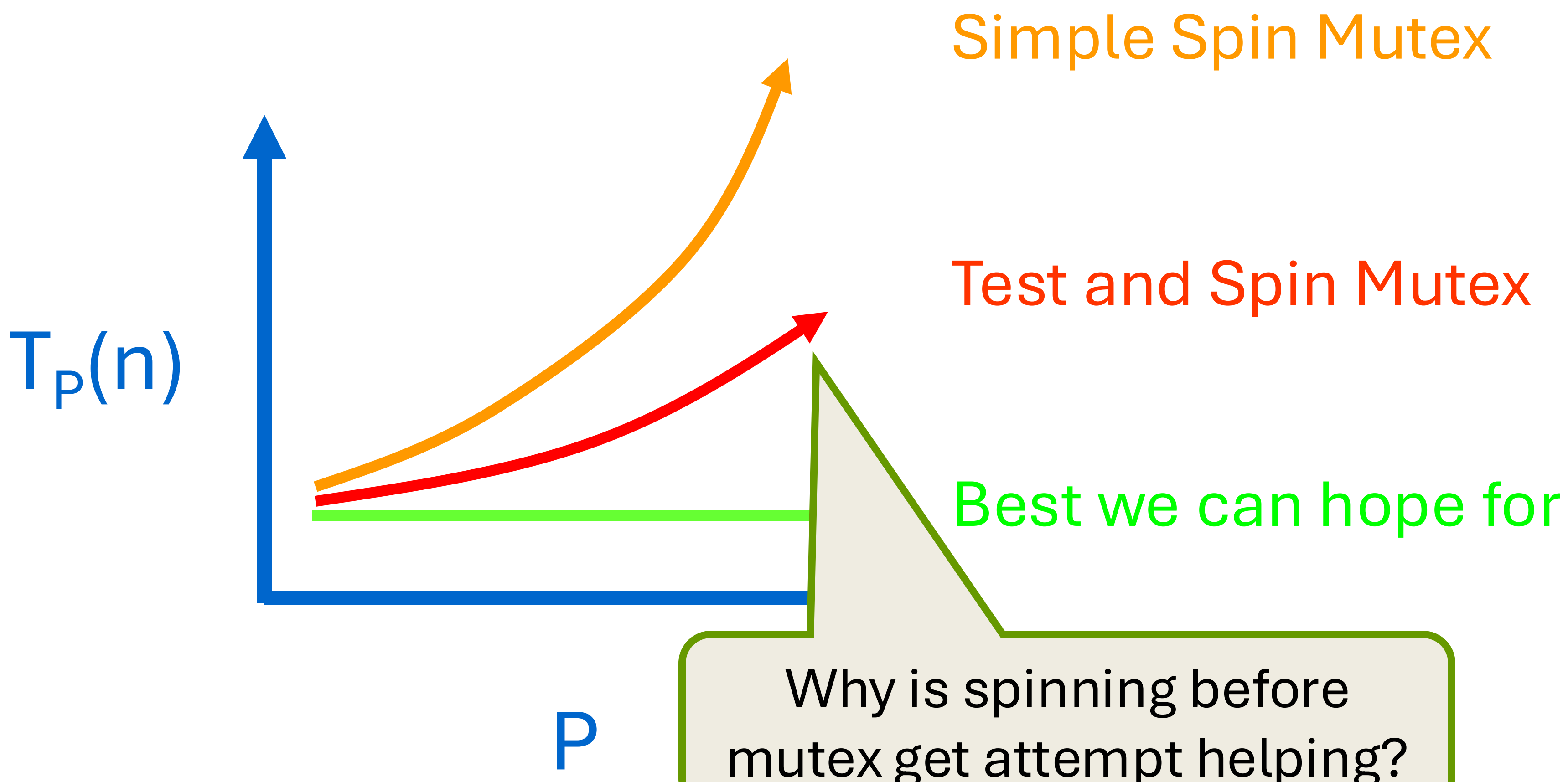
Mystery?



Simple Spin Mutex

```
Spin_Mutex:
      cmp 0, mutex ; Check if *mutex is free
      je Get_Mutex
      pause ; x86 hack to unconfuse pipeline
      jmp Spin_Mutex
Get_Mutex:
      mov 1, %eax
      xchg mutex, %eax ; Try to get mutex
      cmp 0, %eax ; Test if successful
      jne Get_Mutex
Critical_Section:
      <critical-section code>
      mov 0, mutex ; Release mutex
```

Mystery?



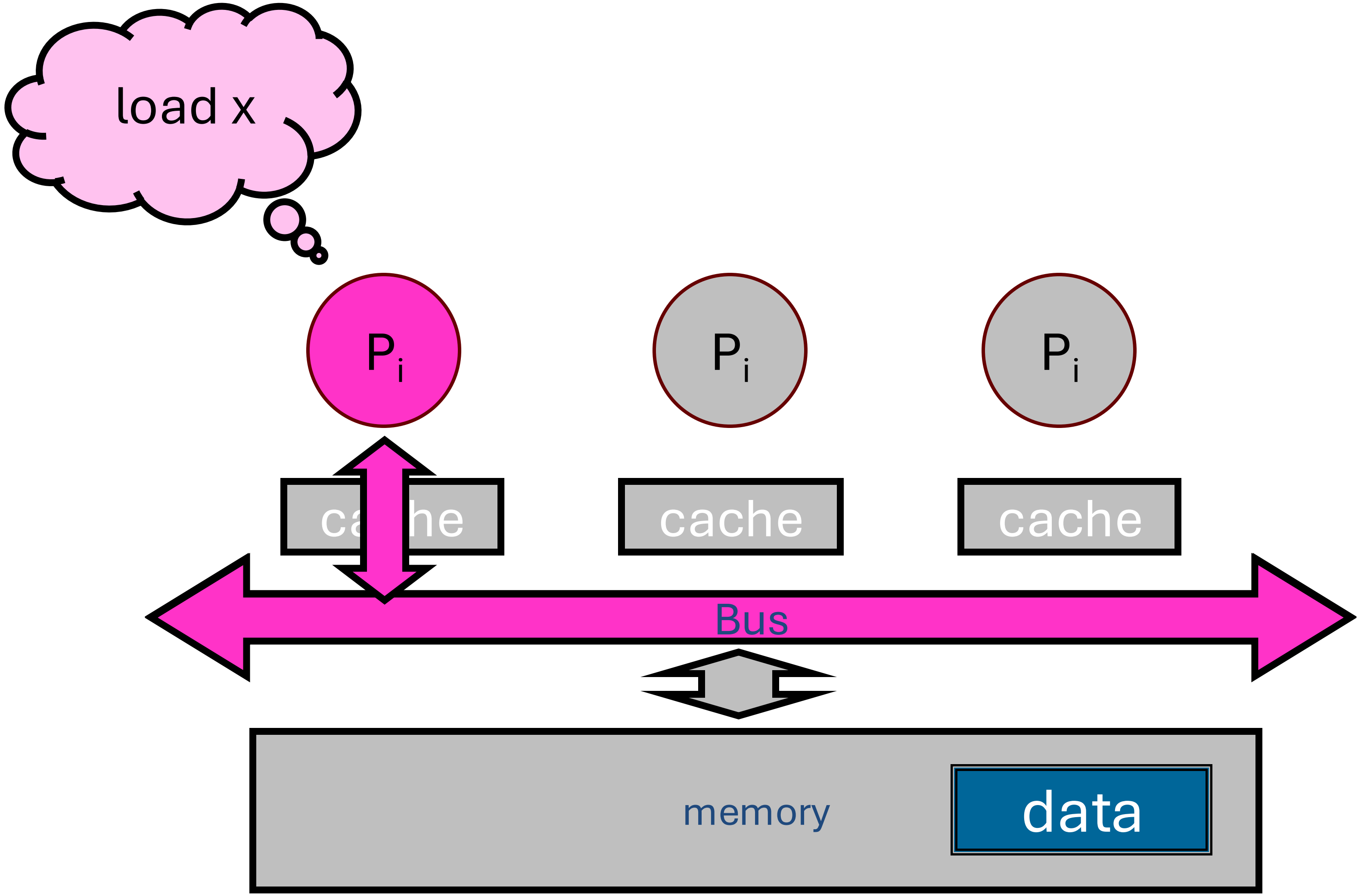
Why is spinning before mutex get attempt helping?

MESI Cache Coherence

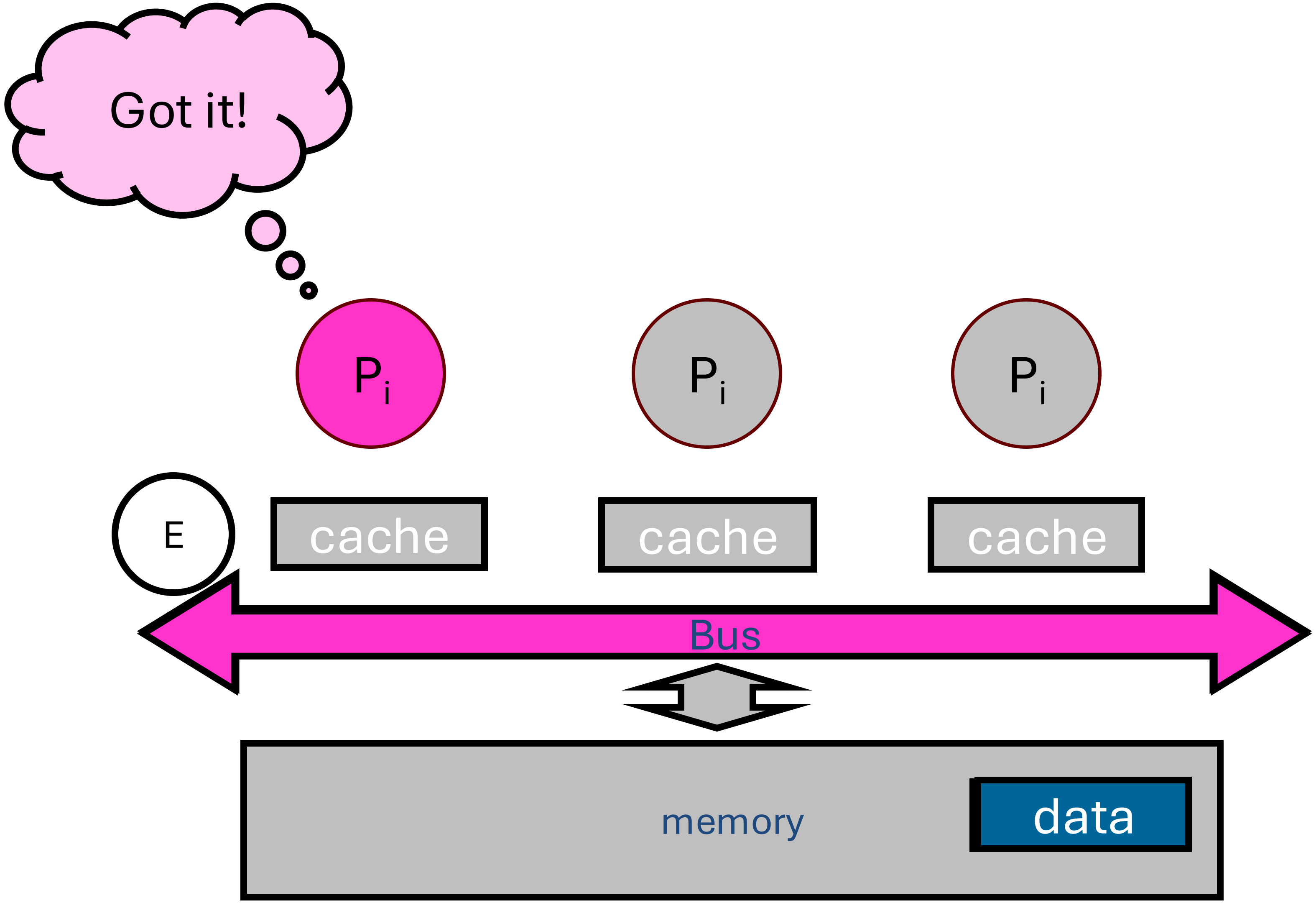
MESI: A slight variant of MSI protocol in Lecture 6

- Modified
  - Have modified cached data, must write back to memory
- Exclusive
  - Not modified, I have only copy
- Shared
  - Not modified, may be cached elsewhere
- Invalid
  - Cache contents not meaningful

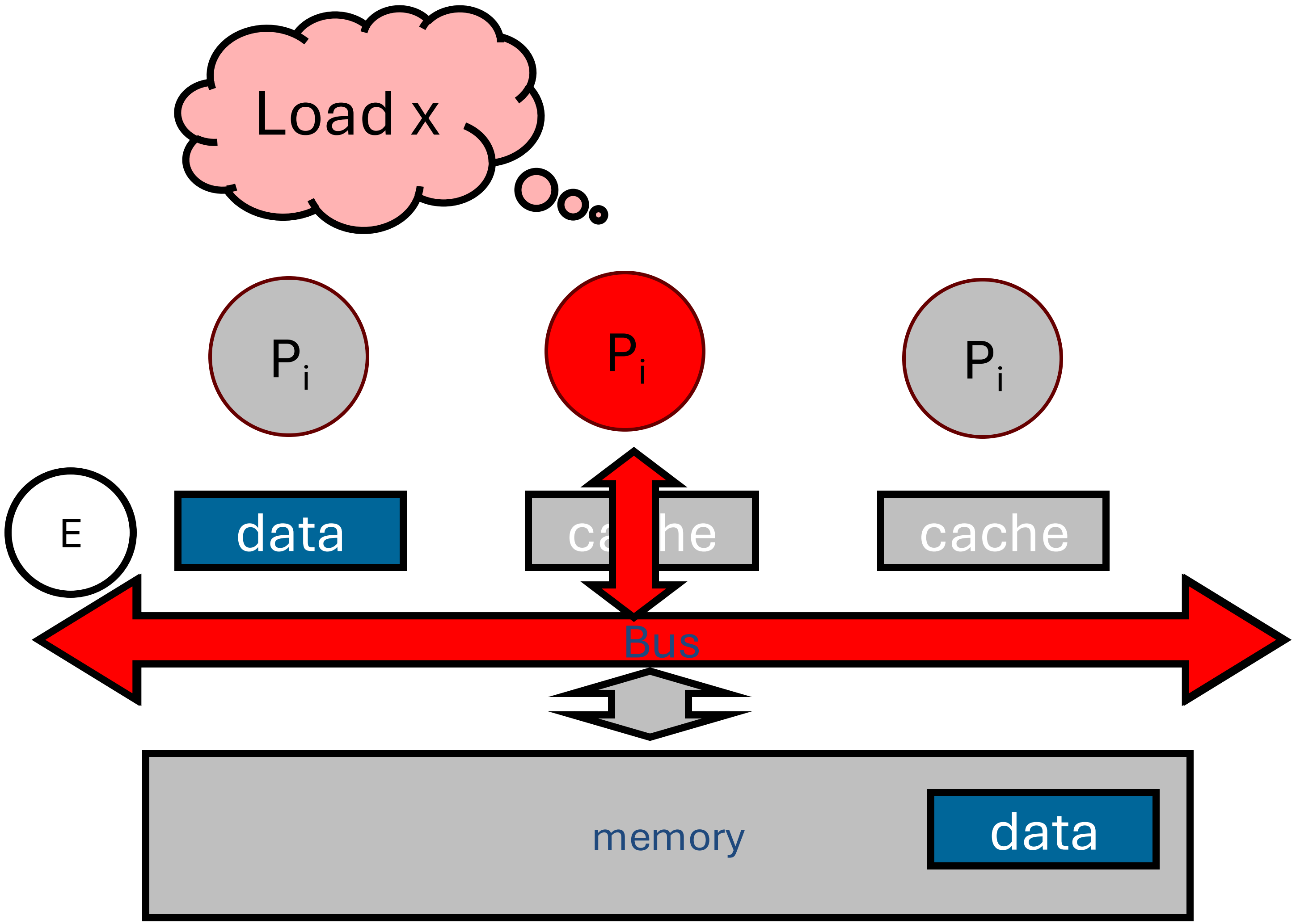
Processor Issues Load Request



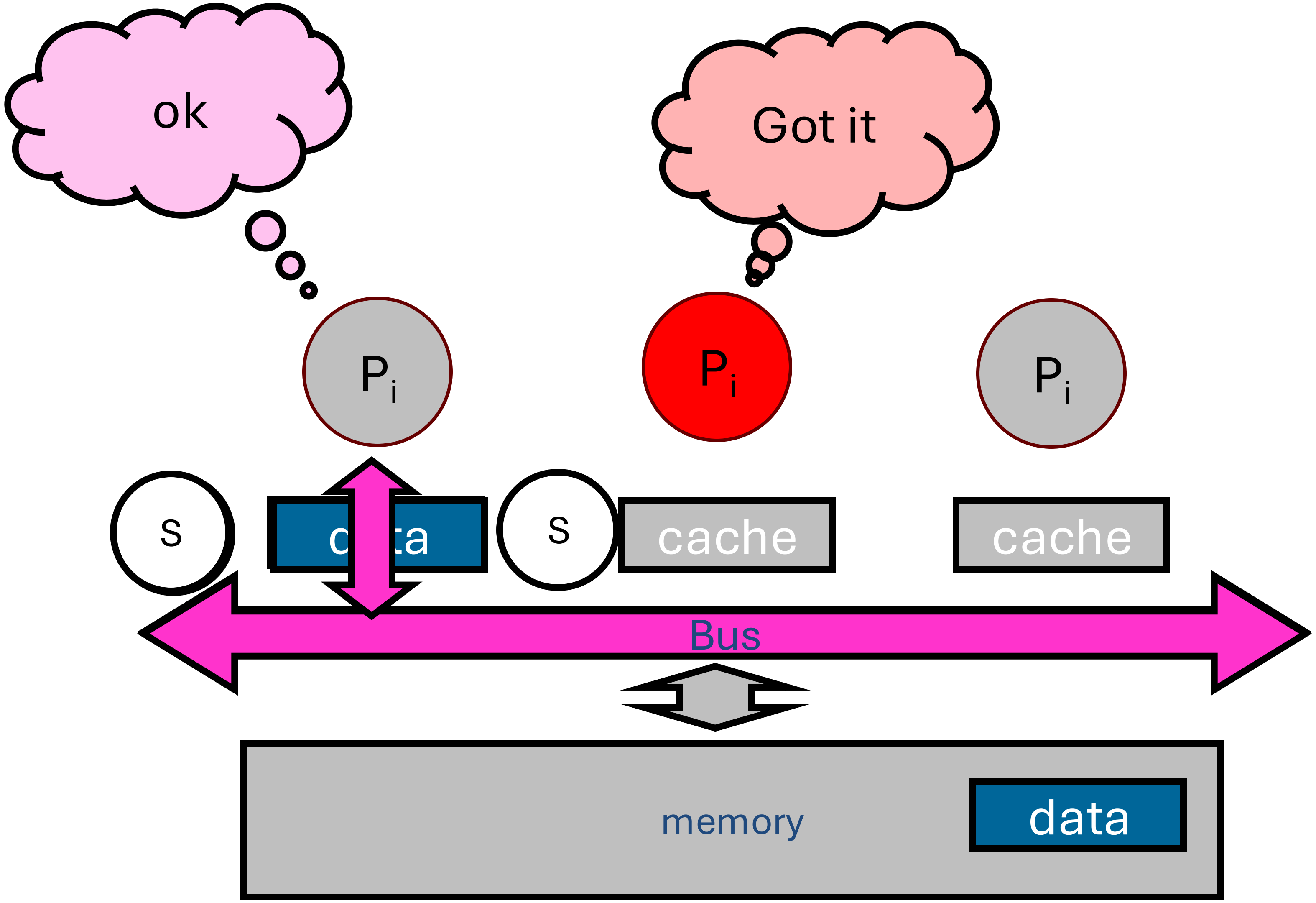
Memory Responds



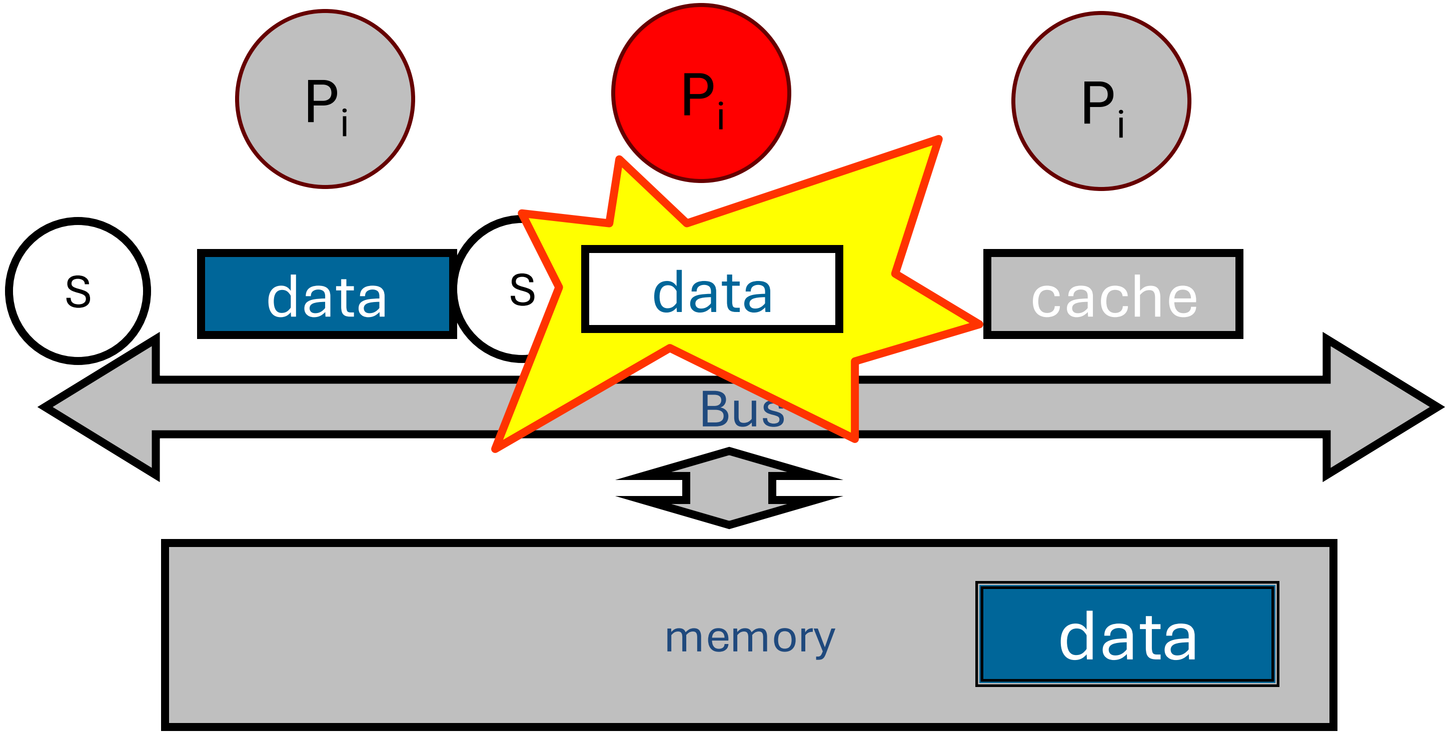
Processor Issues Load Request



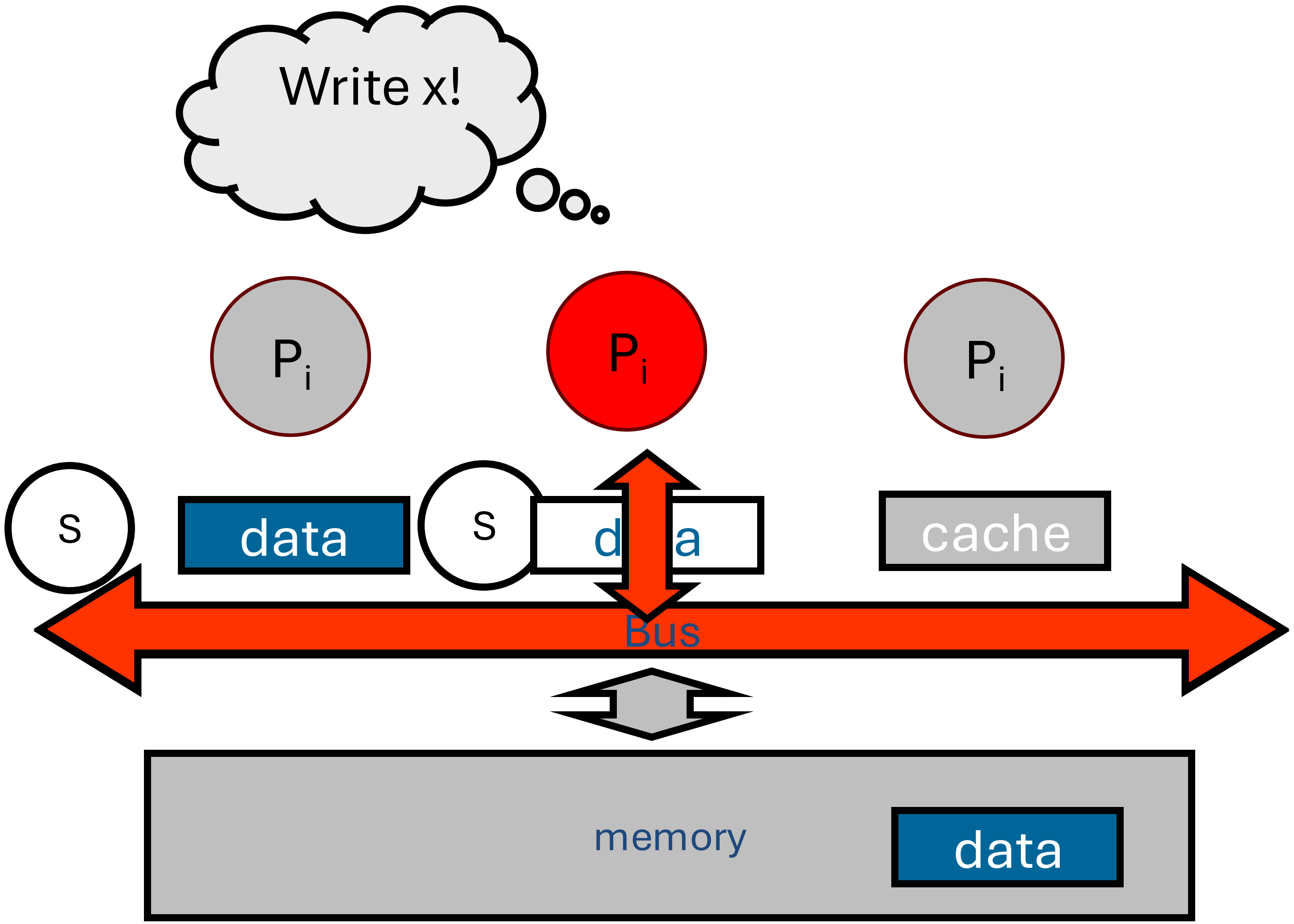
Other Processor Responds



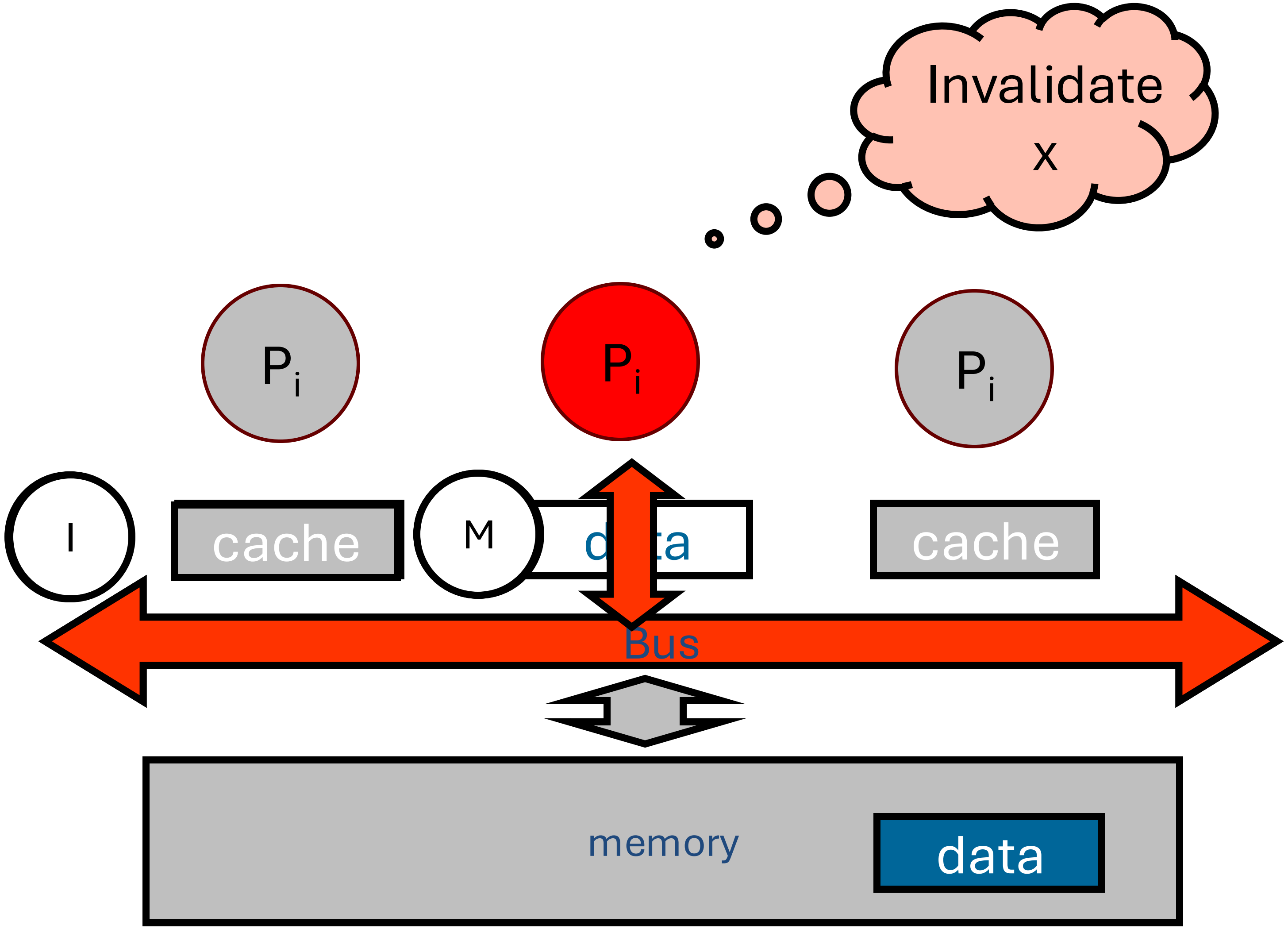
Modify Cached Data



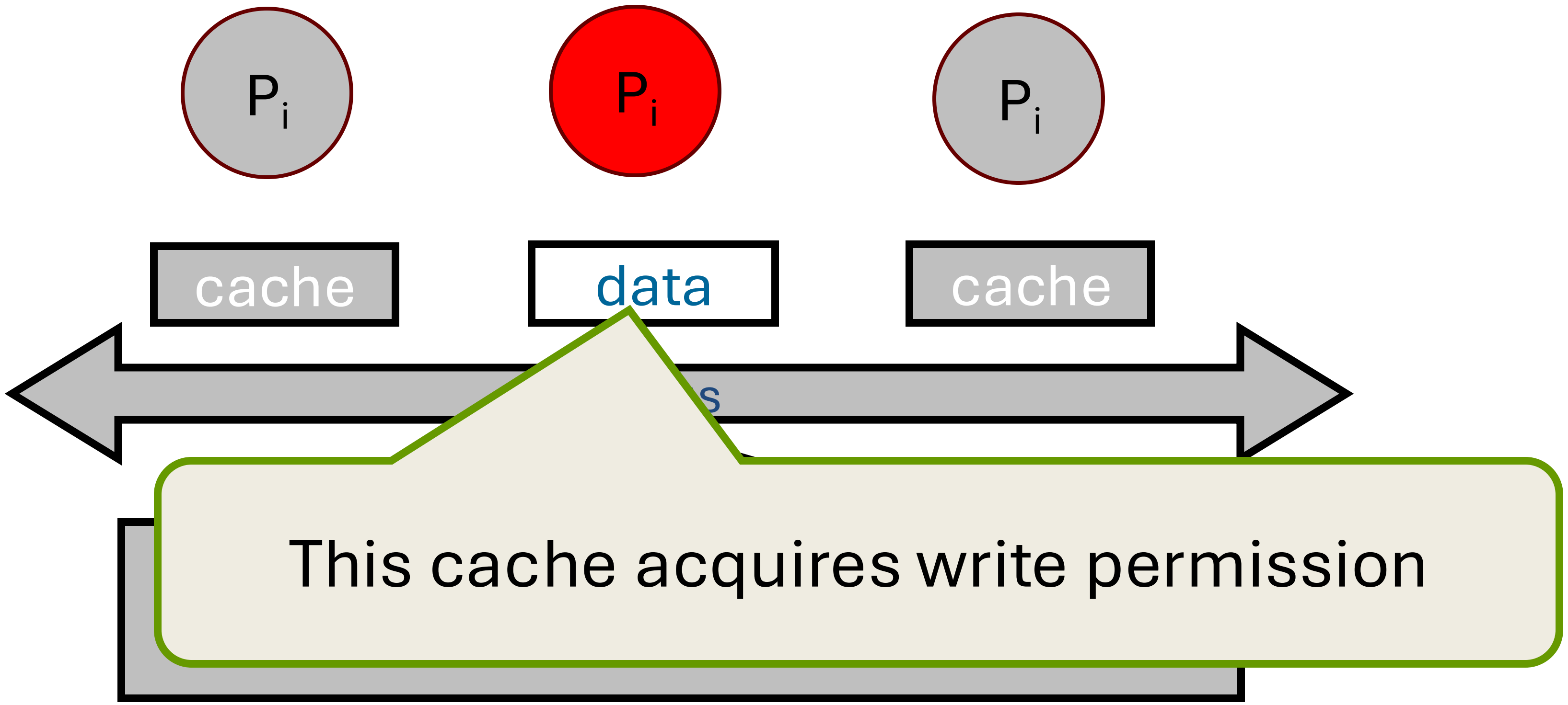
Write-Through Cache



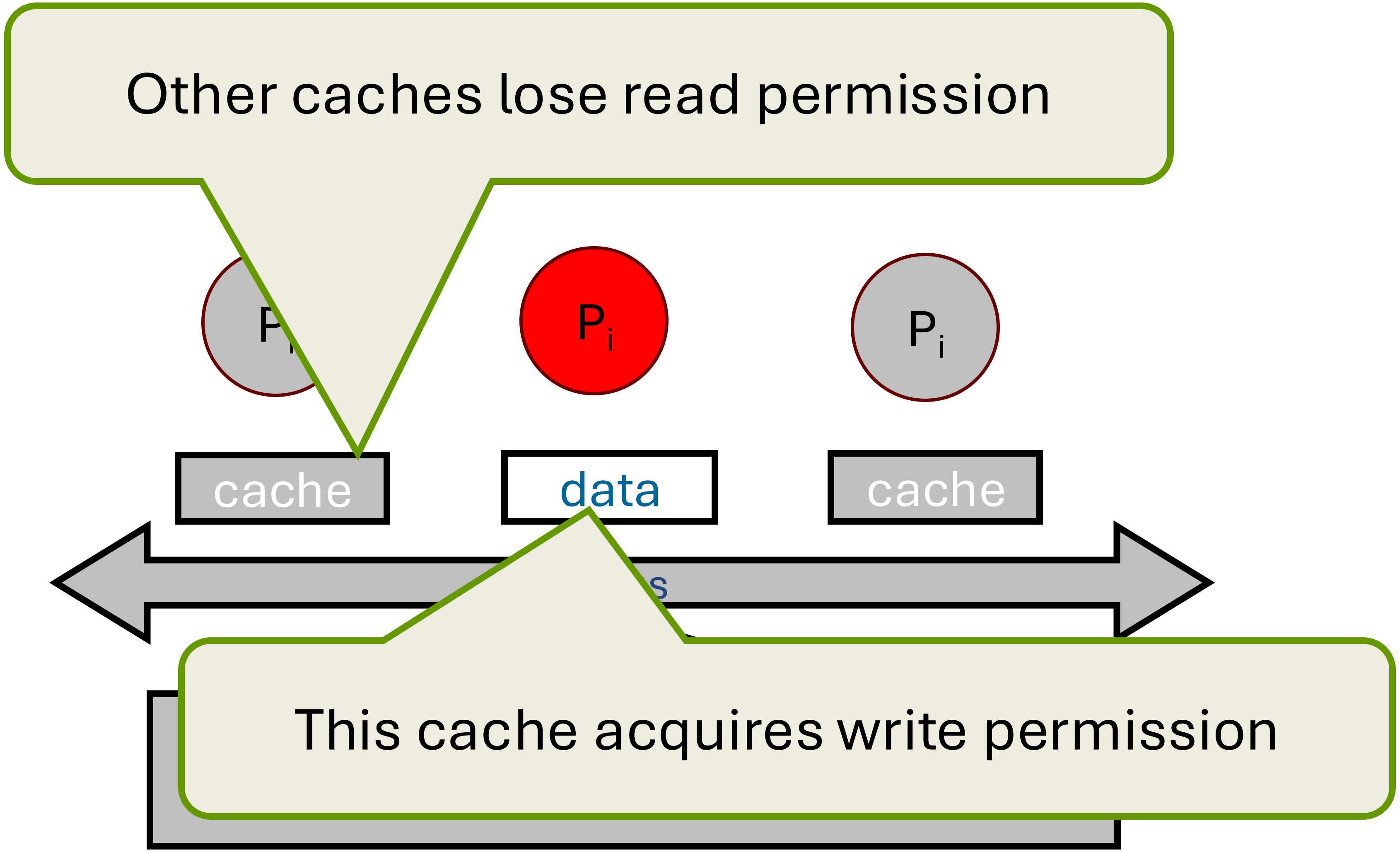
Write Back Cache



Invalidate

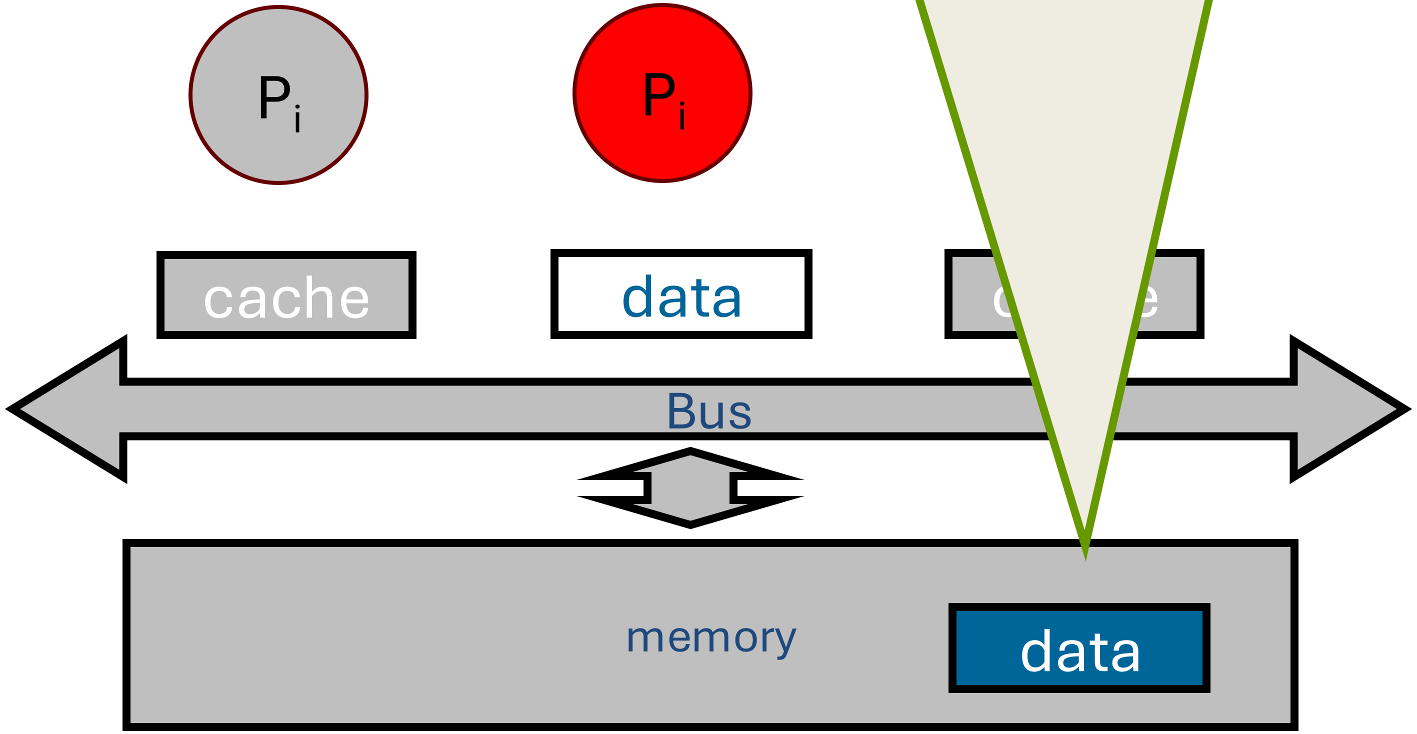

P_i
P_i
P_i
cache
data
cache
This cache acquires write permission

Invalidate

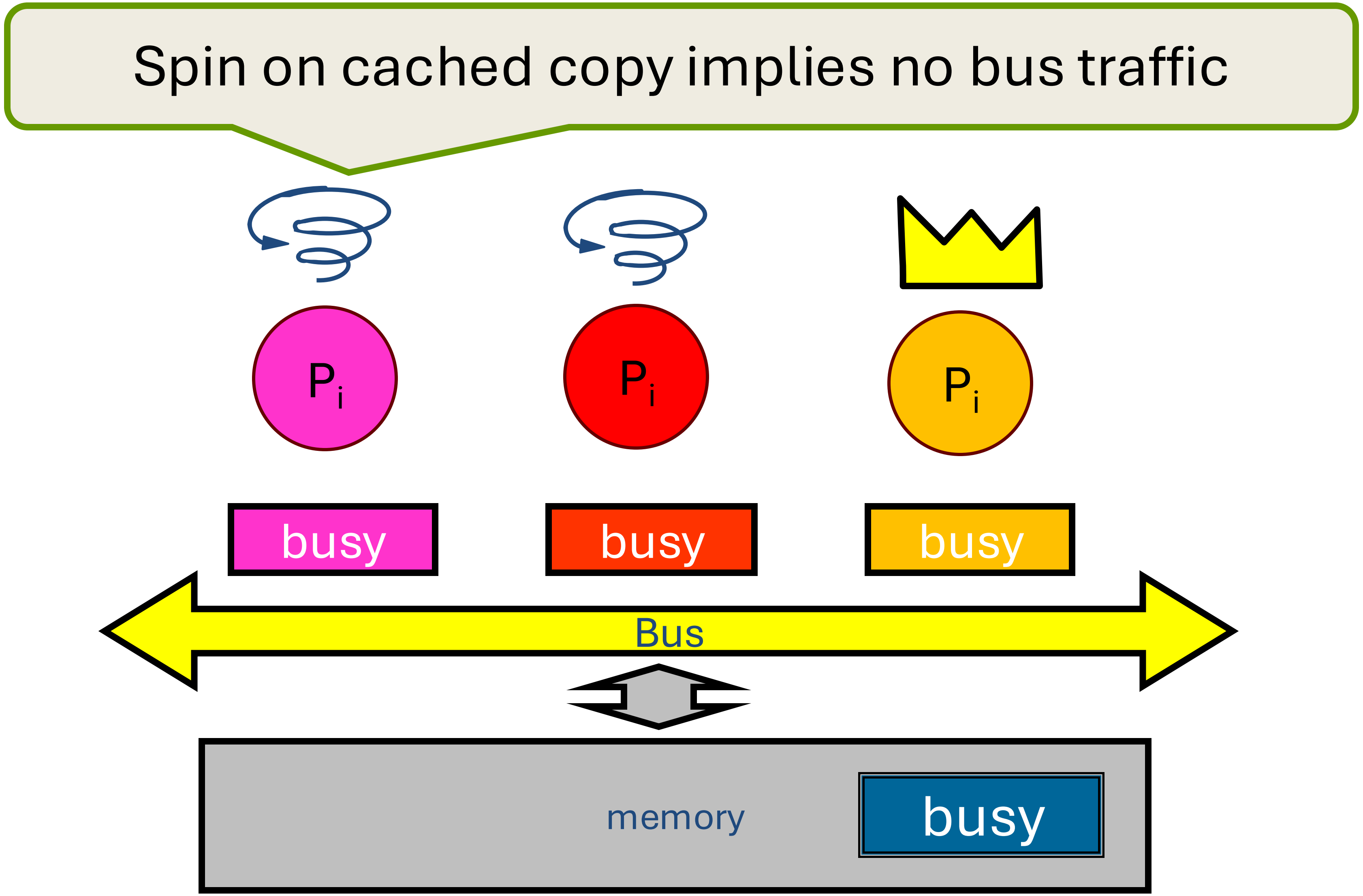

Other caches lose read permission
Pi
Pi
cache
data
cache
This cache acquires write permission

Invalidate

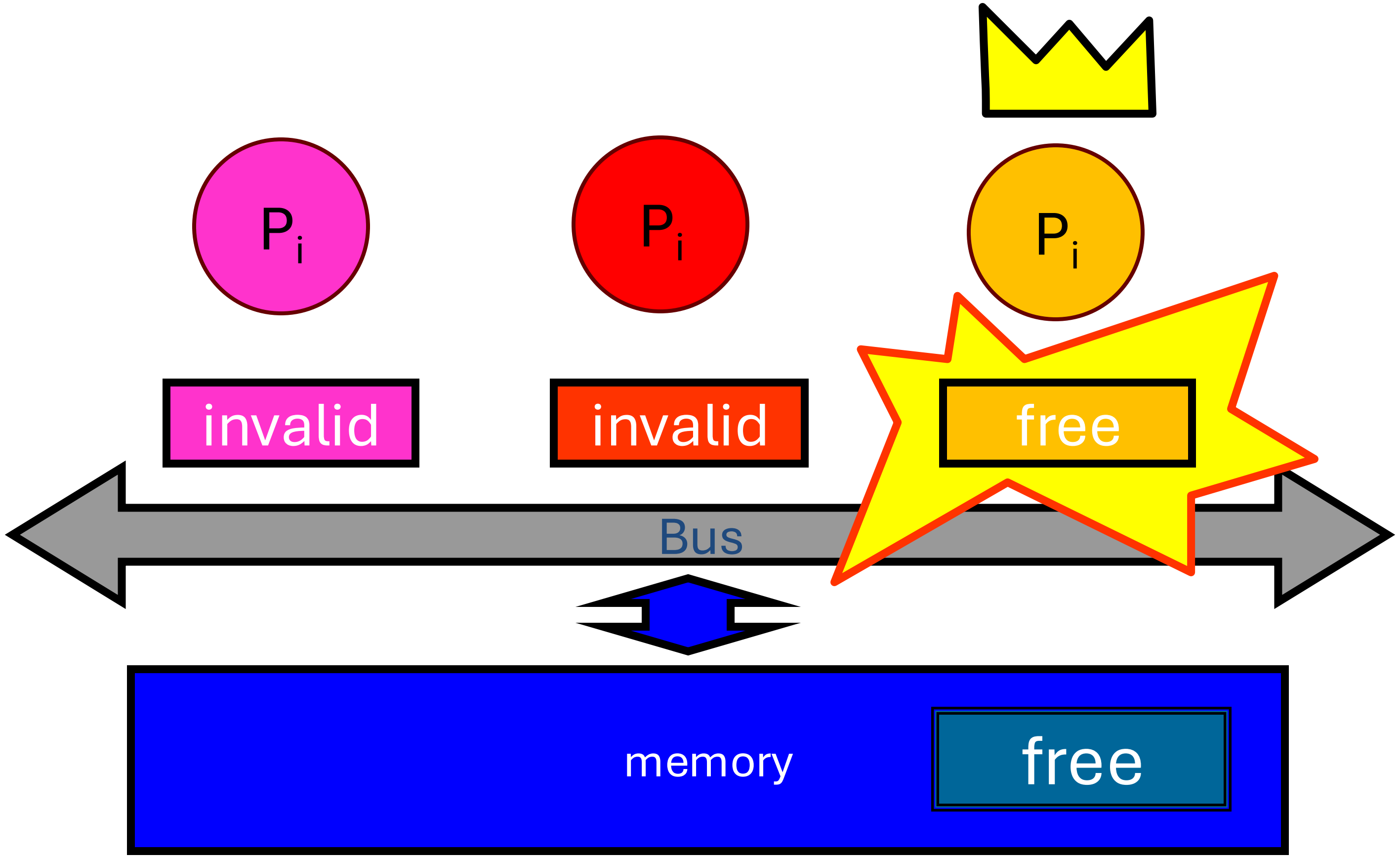
Memory provides data only if not present in any cache, so no need to change it now (expensive)



Spin Mutex: Local Spinning

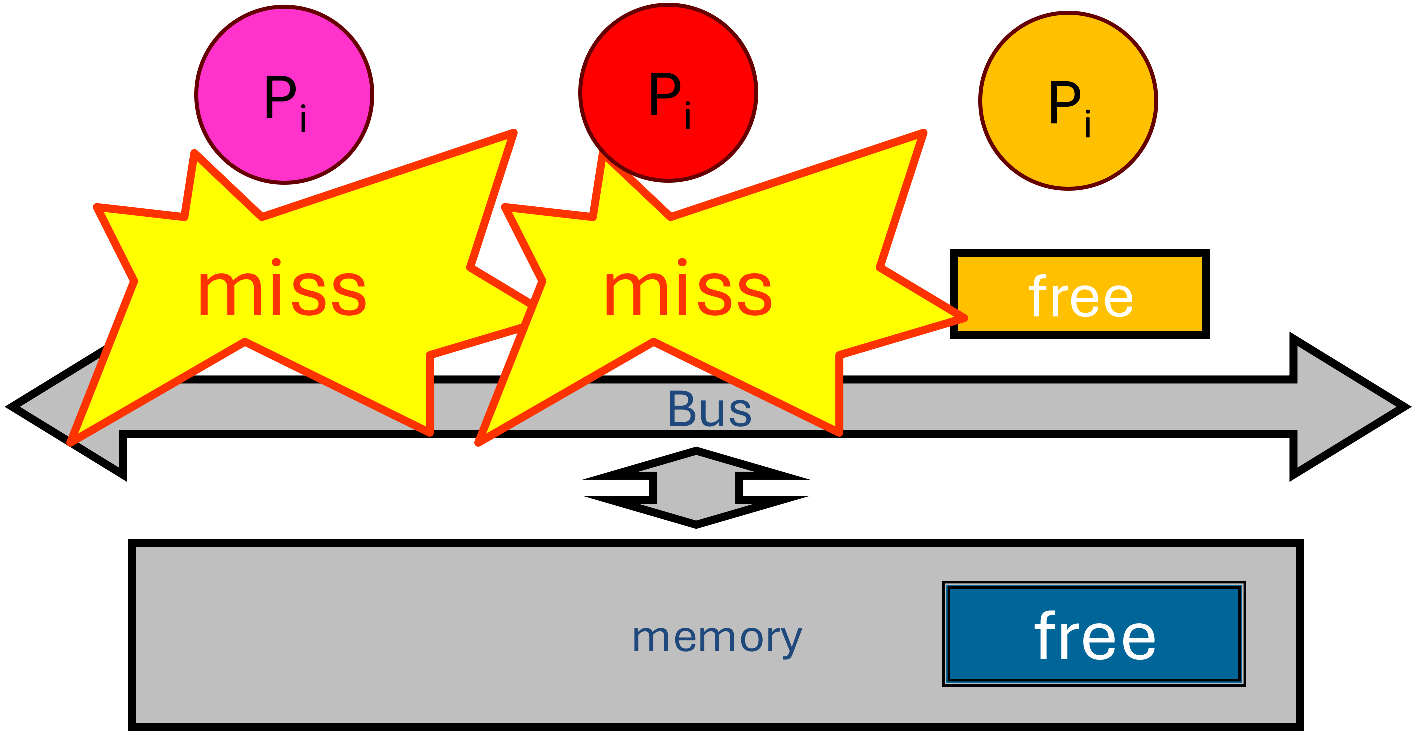


On Release



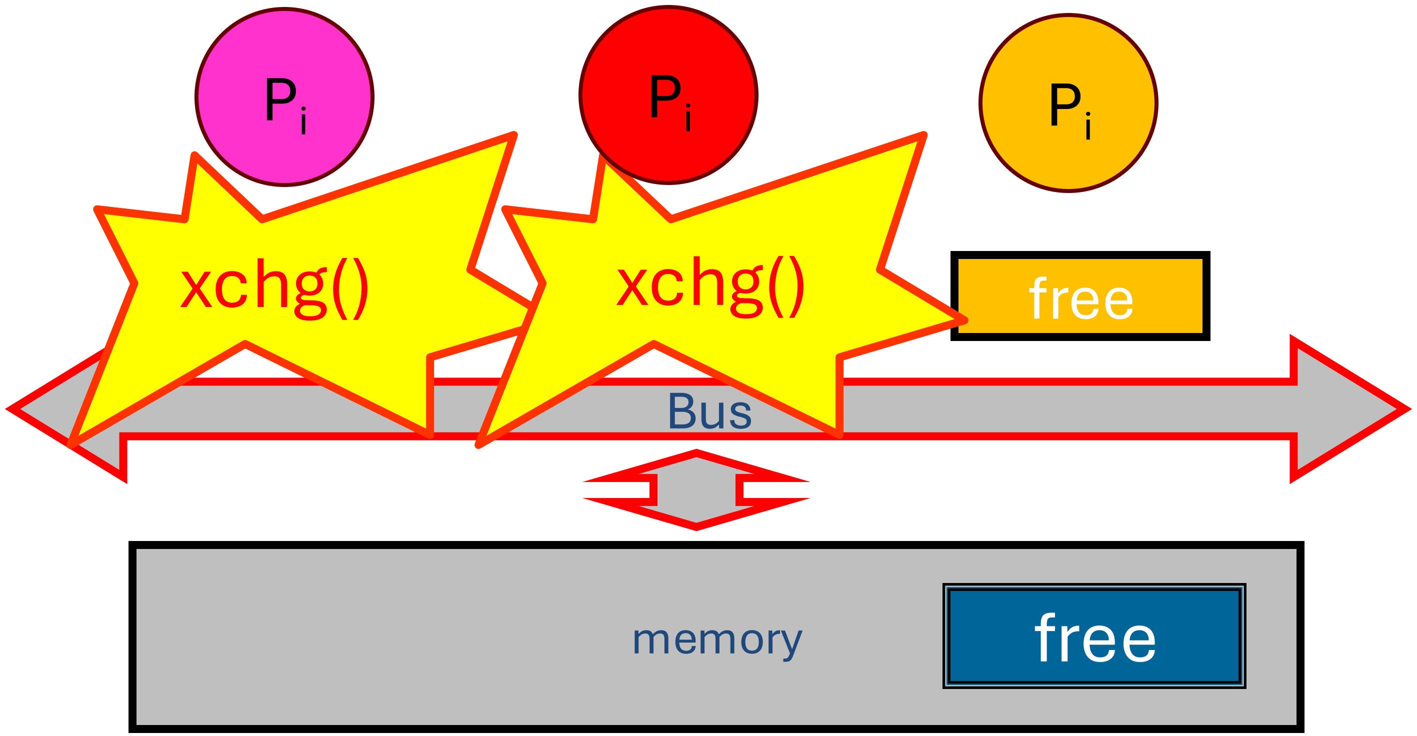
On Release

- Everyone misses, rereads



On Release

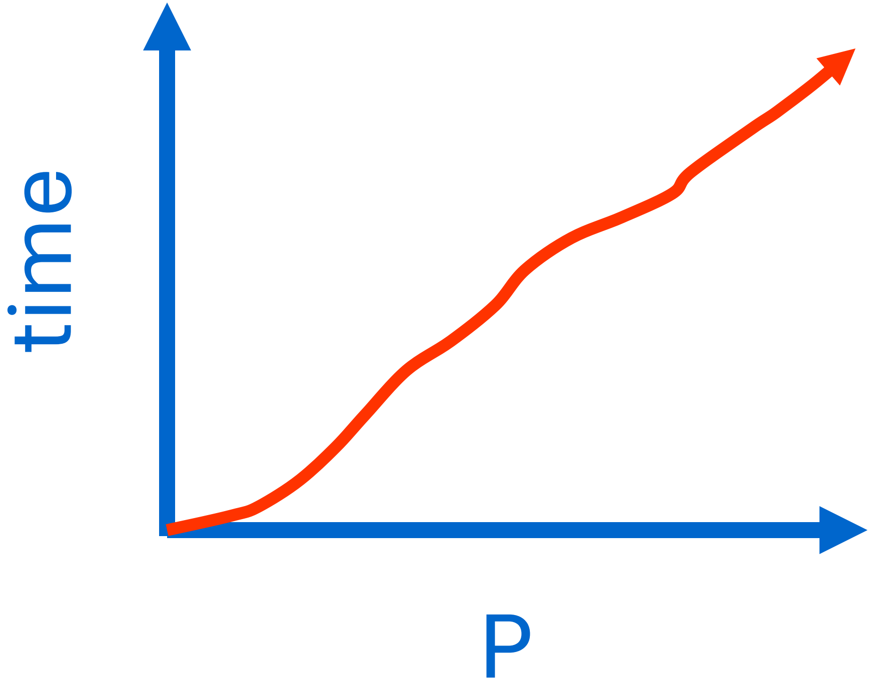
- Everyone tries to get lock



What happens on Release?

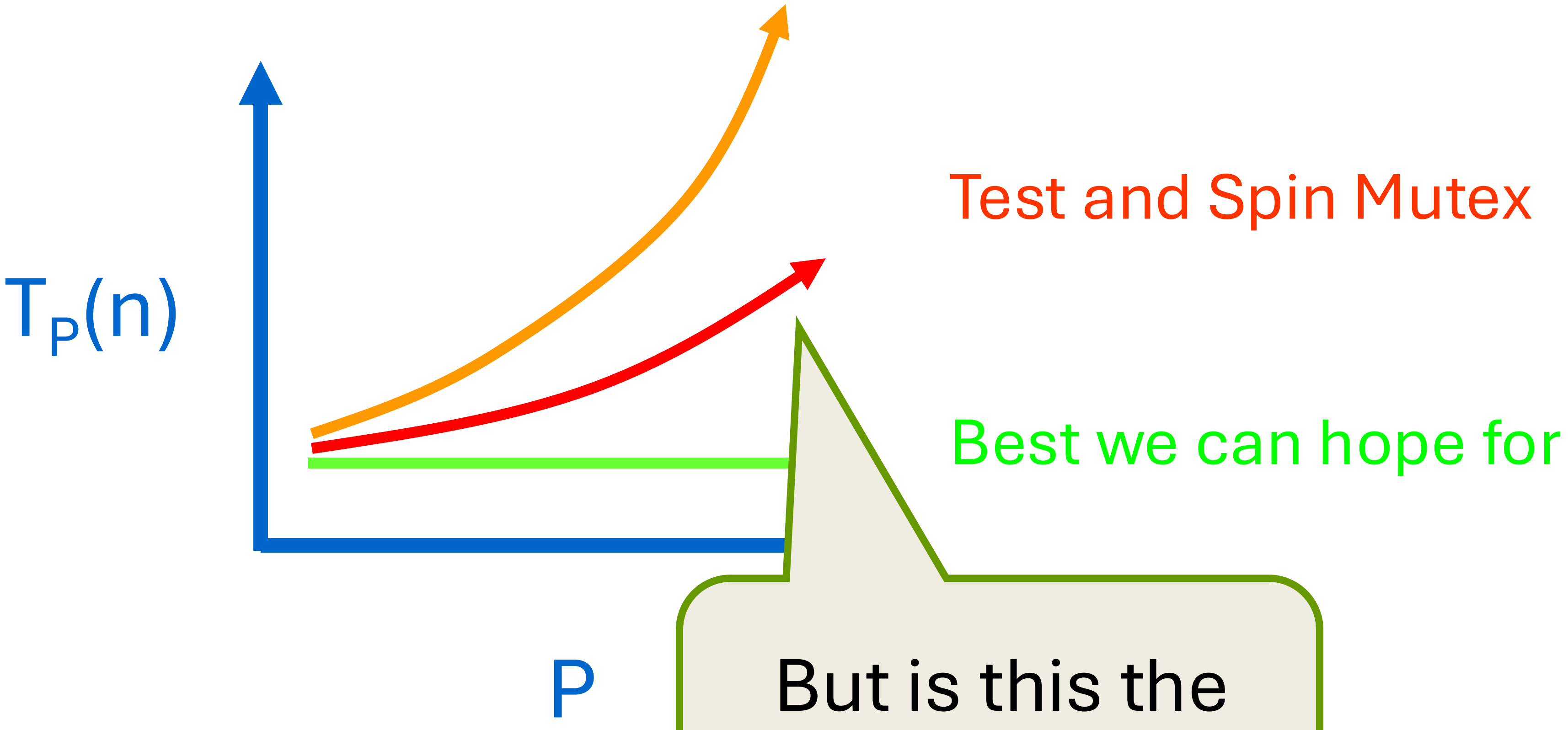
- Everyone misses
  - Reads satisfied sequentially
- Everyone does xchg
  - Invalidates others' caches
- Eventually quiesces after lock acquired
  - How long does this take?

Quiescence Time



Can increase linearly with the number of processors for bus architecture

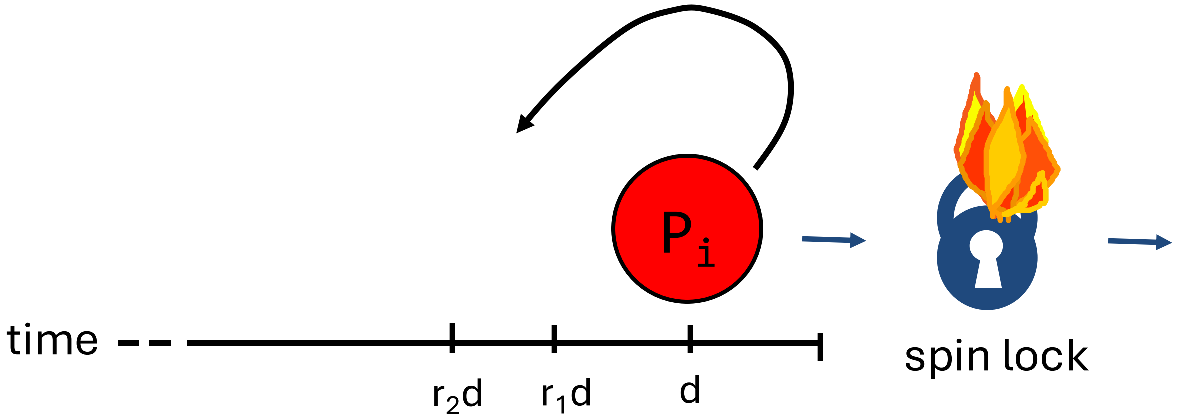
Mystery Explained



But is this the best we can do?

Backoff Mutex

- If the lock looks free, but I fail to get it ⟹
- There must be contention ⟹
- Better to back off than to collide again



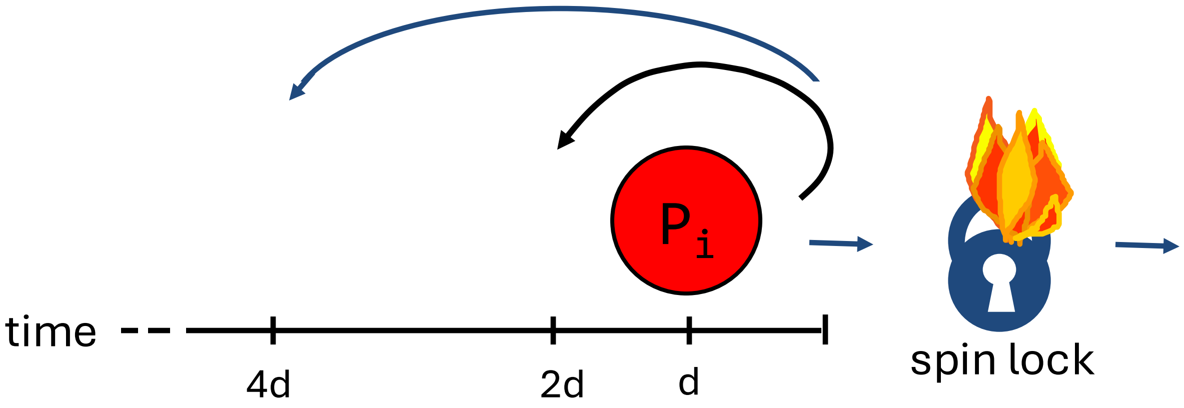
Dynamic Backoff Mutex

```
Spin_Mutex:
      cmp 0, mutex ; Check if *mutex is free
      je Get_Mutex
      pause ; insert here dynamic delay
      jmp Spin_Mutex
Get_Mutex:
      mov 1, %eax
      xchg mutex, %eax ; Try to g
      cmp 0, %eax ; Test if successful
      jne Spin_Mutex
Critical_Section:
      <critical-section code>
      mov 0, mutex ; Release mutex
```

Replace pause with delay adjusted to fit contention level

Exponential Backoff

- If I fail to get a spin lock
  - Wait random duration before retry
  - Each subsequent failure doubles expected wait



There are many other types of locks to address such issues...

Outline

- Atomicity & Mutual Exclusion
- Implementation of Mutexes
- Locking Anomaly: Contention
- **Locking Anomaly: Deadlock**
- Locking Anomaly: Convoying

Deadlock

Holding more than one lock at a time can be dangerous:

Thread 1

```
lock(&A);
lock(&B);
  ⟨critical section⟩
unlock(&B);
unlock(&A);
```

Thread 2

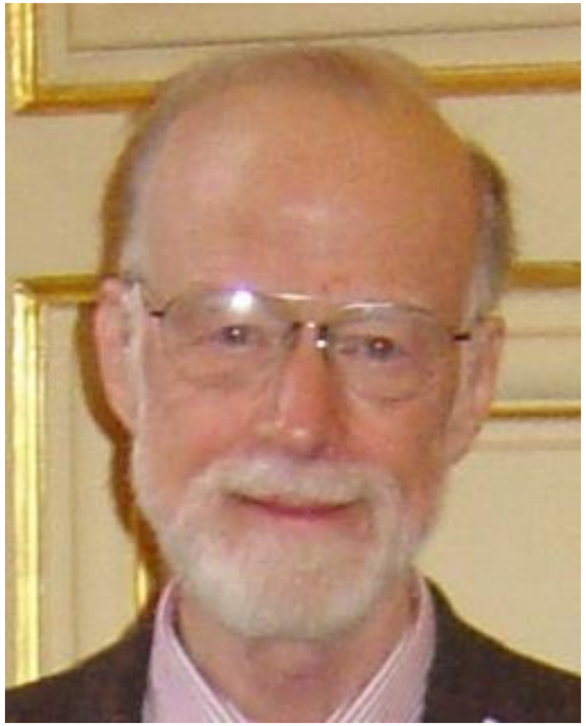
```
lock(&B);
lock(&A);
  ⟨critical section⟩
unlock(&A);
unlock(&B);
```

The ultimate loss of performance!

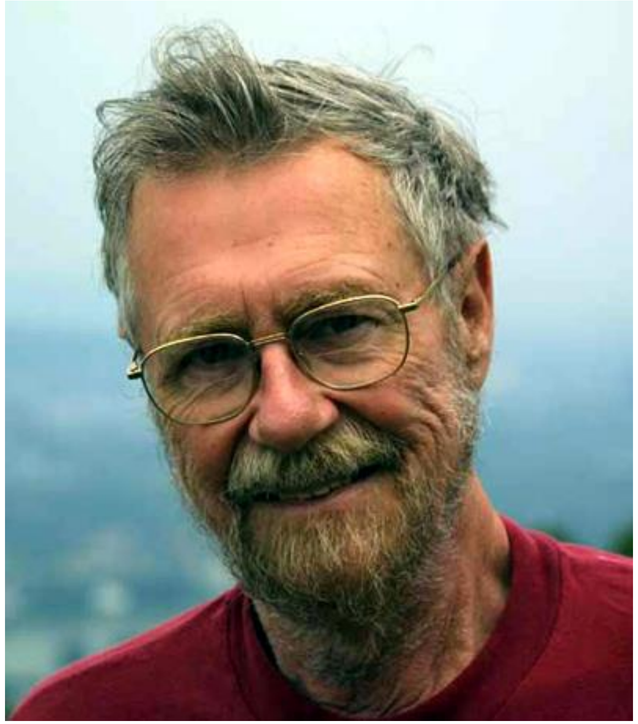
Conditions for Deadlock

1. Mutual exclusion — Each thread claims exclusive control over the resources it holds.
2. Nonpreemption — Each thread does not release the resources it holds until it completes its use of them.
3. Circular waiting — A cycle of threads exists in which each thread is blocked waiting for resources held by the next thread in the cycle.

Dining Philosophers



C.A.R. (Tony) Hoare



Edsger Dijkstra

Illustrative story of deadlock told by Hoare,
based on an examination question by Dijkstra.
The story has been embellished over the years by many retellers.

Dining Philosophers

Each of n philosophers needs the two chopsticks on either side of their plate to eat their noodles.

Philosopher i

```
while (1) {
  think();
  lock(&chopstick[i].L);
  lock(&chopstick[(i+1)%n].L);
    eat();
  unlock(&chopstick[i].L);
  unlock(&chopstick[(i+1)%n].L);
}
```

~~Dining~~ Starving Philosophers

Each of n philosophers needs the two chopsticks on either side of their plate to eat their noodles.

One day they all pick up their left chopsticks simultaneously

Philos

```
while (1) {
  think();
  lock(&chopstick[i].L);
  lock(&chopstick[(i+1)%n].L);
    eat();
  unlock(&chopstick[i].L);
  unlock(&chopstick[(i+1)%n].L);
}
```

Preventing Deadlock

Theorem. Assume that we can linearly order the mutexes $L_1 \lessdot L_2 \lessdot \cdots \lessdot L_n$ so that whenever a thread holds a mutex L_i and attempts to lock another mutex L_j, we have $L_i \lessdot L_j$. Then, no deadlock can occur.

Proof. Suppose that a cycle of waiting exists. Consider the thread in the cycle that holds the "largest" mutex L_{max} in the ordering, and suppose that it is waiting on a mutex L held by the next thread in the cycle. Then, we must have $L_{max} \lessdot L$. Contradiction. ■

Dining Philosophers

Philosopher **i**

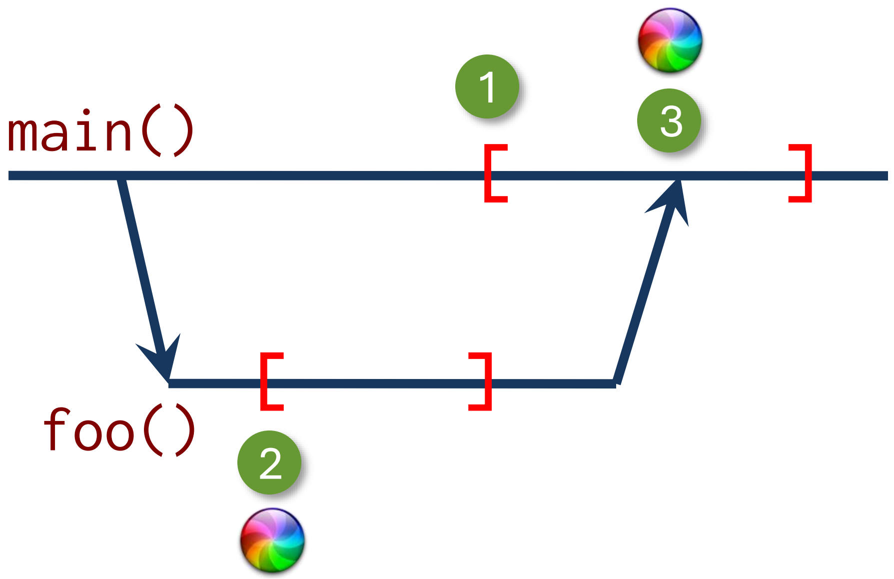
```
while (1) {
  think();
  lock(&chopstick[min(i,(i+1)%n)].L);
  lock(&chopstick[max(i,(i+1)%n)].L);
    eat();
  unlock(&chopstick[i].L);
  unlock(&chopstick[(i+1)%n].L);
}
```

Everyone grabs left then right, except philosopher n-1 : they're always going to reach for the right first

Deadlocking Cilk with just one lock

```
void main() {
  cilk_scope {
    cilk_spawn foo();
    lock(&L); 1
  } 3
  unlock(&L);
}

void foo() {
  lock(&L); 2
  unlock(&L);
}
```



- Don’t hold mutexes across joins (scope end)!
- Hold mutexes only within `cilk_scope`’s
- As always, try to avoid nondeterministic programming (but not always possible)

Outline

- Atomicity & Mutual Exclusion
- Implementation of Mutexes
- Locking Anomaly: Contention
- Locking Anomaly: Deadlock
- **Locking Anomaly: Convoying**

Convoying

A lock convoy occurs when multiple threads of equal priority contend repeatedly for the same lock.

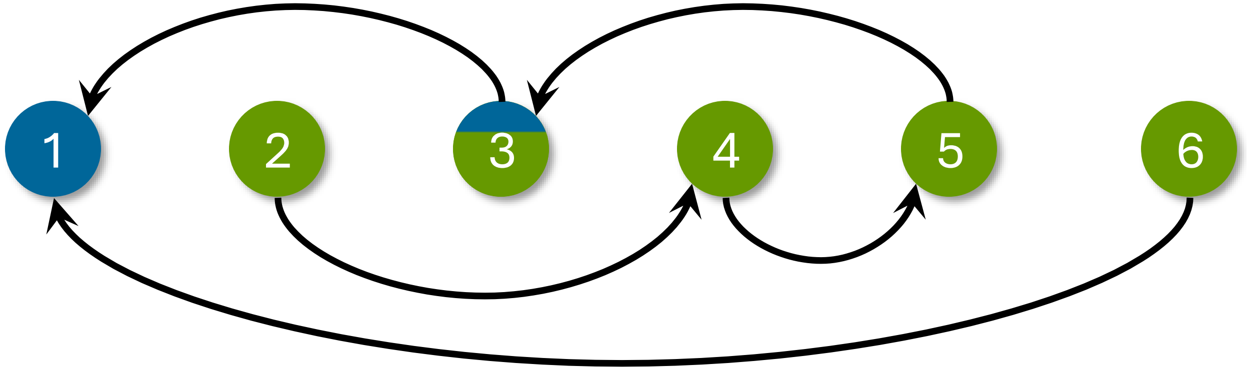
Example: Performance bug in MIT Cilk

When random work-stealing, each thief grabs a mutex on its victim's deque:

- If victim's deque is empty, the thief releases the mutex and tries again at random
- If victim's deque contains work, the thief steals the topmost frame and then releases the mutex

PROBLEM: At start-up, most thieves quickly converge on the worker containing the initial strand, creating a convoy.

Performance Bug in MIT-Cilk



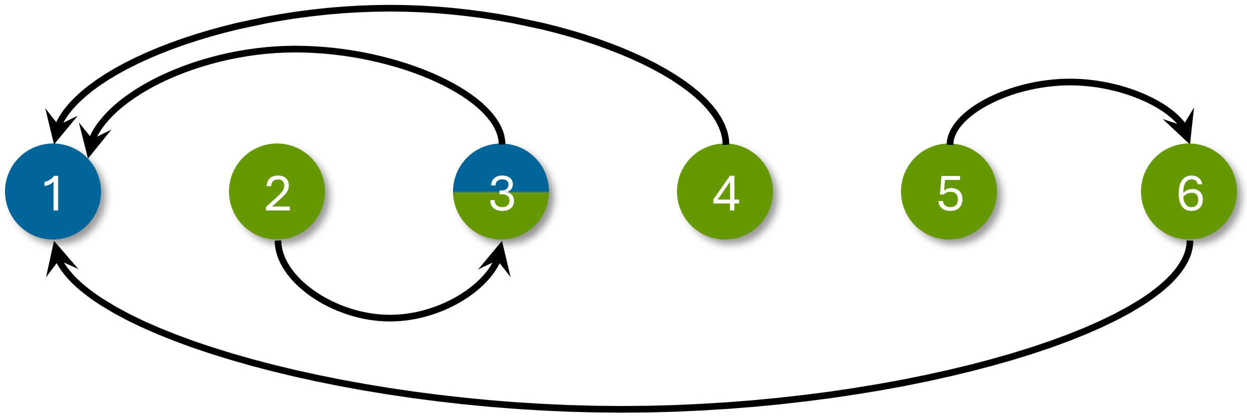
: busy worker

3 : successful steal in progress

: idle worker

: dependency from onto the lock on 's deque

Performance Bug in MIT-Cilk



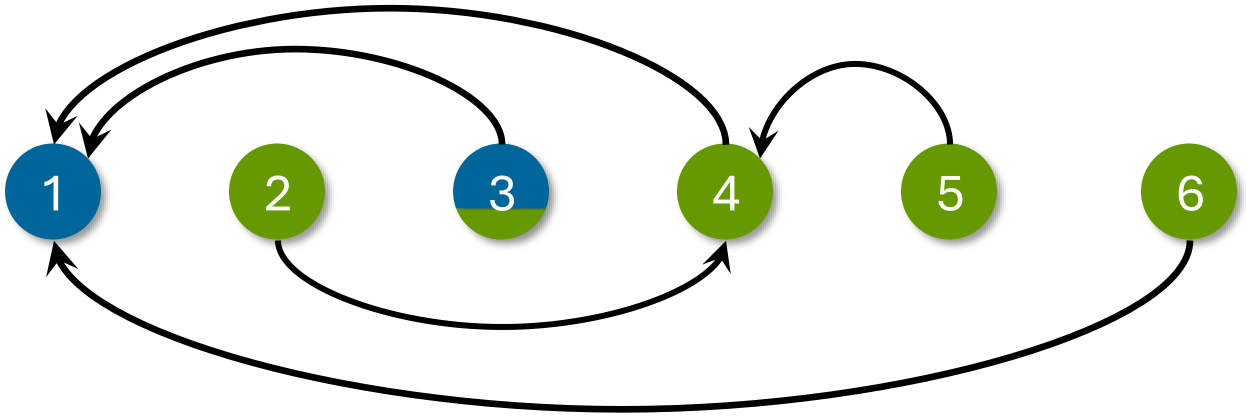
: busy worker

: idle worker

3 : successful steal in progress

: dependency from onto the lock on ‘s deque

Performance Bug in MIT-Cilk



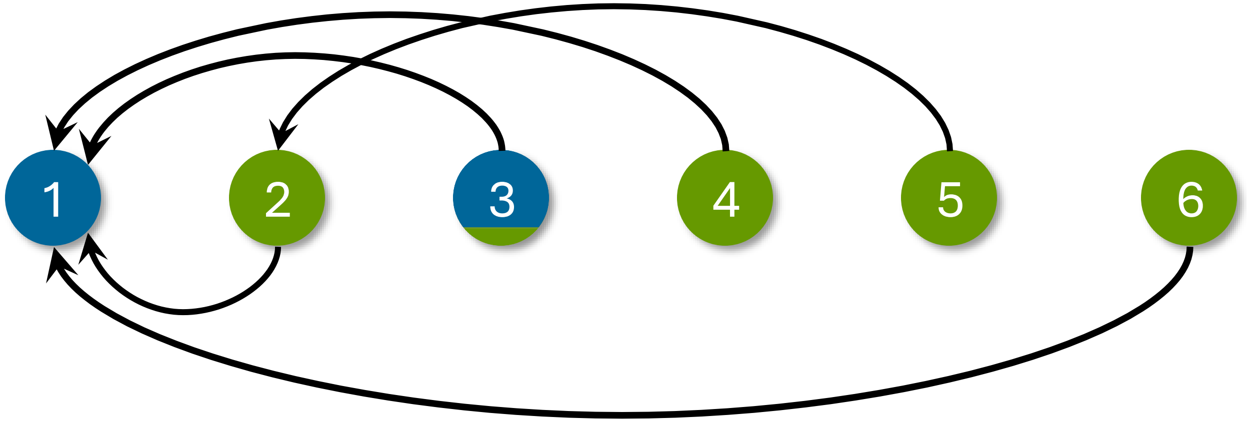
: busy worker

: idle worker

: successful steal in progress

: dependency from onto the lock on 's deque

Performance Bug in MIT-Cilk



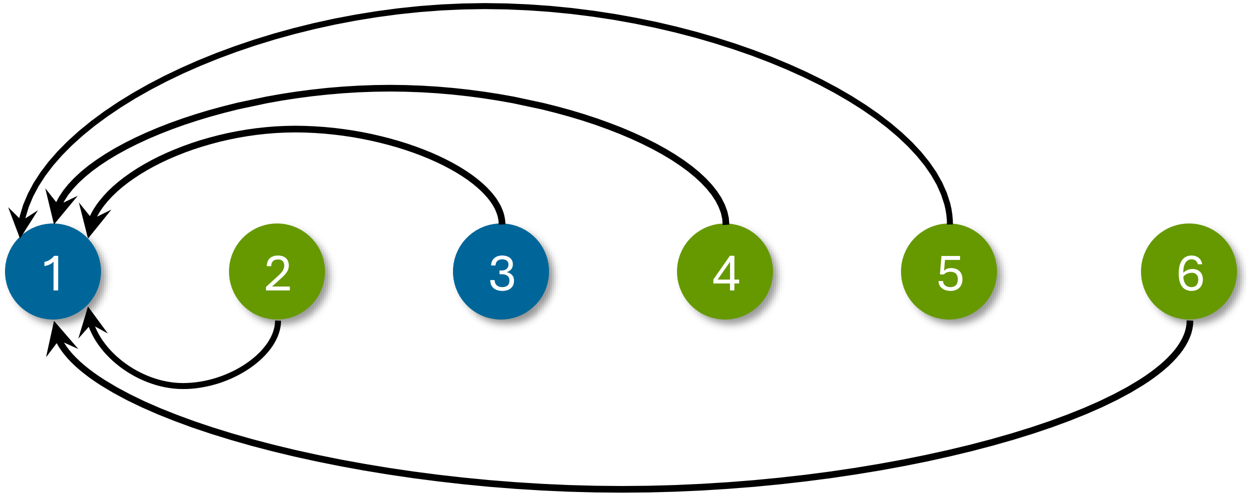
: busy worker

3 : successful steal in progress

: idle worker

: dependency from onto the lock on 's deque

Performance Bug in MIT-Cilk



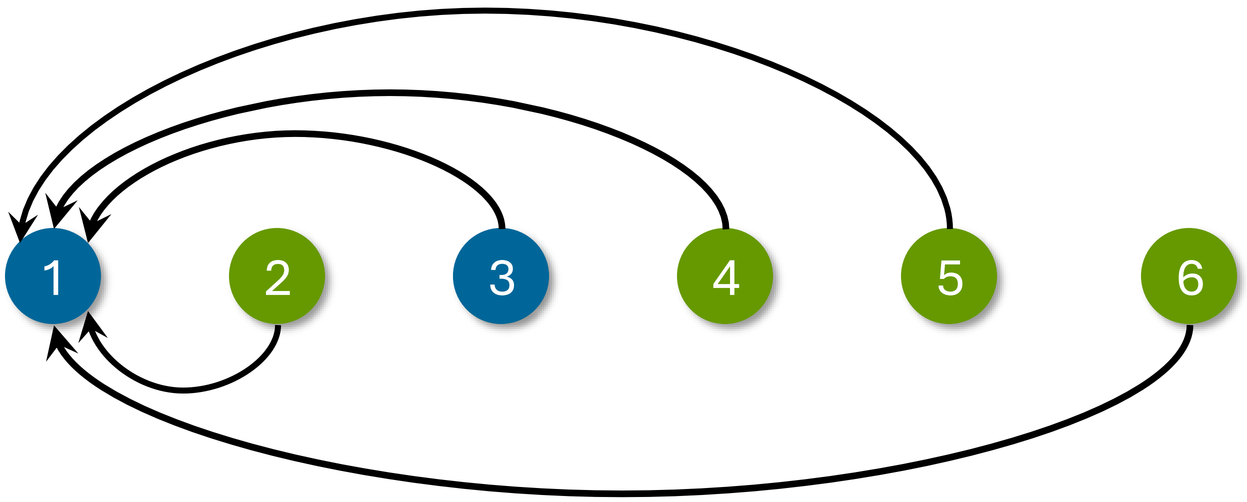
: busy worker

3 : successful steal in progress

: idle worker

: dependency from onto the lock on ‘s deque

Performance Bug in MIT-Cilk



The work now gets distributed slowly as each thief serially obtains Processor 1's mutex.

Solving the Convoying Problem

- Use the nonblocking function `try_lock()`, rather than `lock()`:
- `try_lock()` attempts to acquire the mutex and returns a flag indicating whether it was successful, but it does not block on an unsuccessful attempt

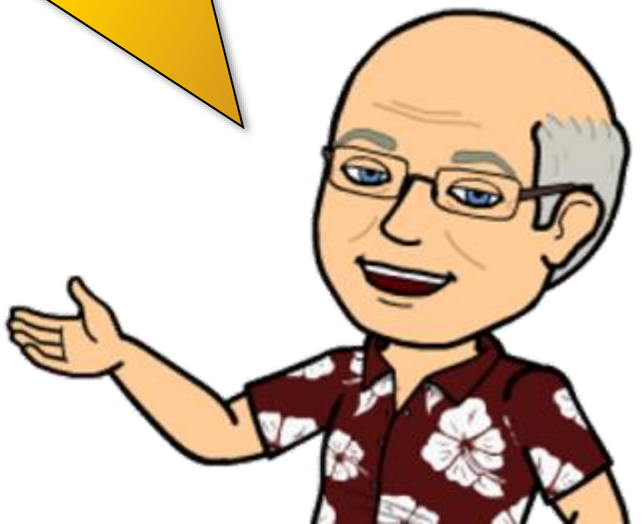
- In Cilk Plus, when a thief fails to acquire a mutex, it simply tries to steal again at random, rather than blocking.

Outline

- Atomicity & Mutual Exclusion
- Implementation of Mutexes
- Locking Anomaly: Contention
- Locking Anomaly: Deadlock
- Locking Anomaly: Convoying

Golden Rule of Parallel Programming

Never write nondeterministic parallel programs



Silver Rule of Parallel Programming

Never write nondeterministic parallel programs

— but if you must* —

always devise a test strategy to manage the nondeterminism!