

Distributed Shared Memory: Ivy

Brad Karp
UCL Computer Science



CS 0133
14th October 2019

Increasing Transparency: From RPC to Shared Memory

- In RPC, we've seen one way to split application across multiple nodes
 - Carefully specify interface between nodes
 - Explicitly communicate between nodes
 - **Transparent to programmer?**
- Can we hide all inter-node communication from programmer, and improve transparency?
 - Today's topic: Distributed Shared Memory

Ivy: Distributed Shared Memory

- Supercomputer: super-expensive 100-CPU machine, custom-built hardware
- Ivy: 100 cheap PCs and a LAN (all off-the-shelf hardware!)
- Both offer same easy view for programmer:
 - single, shared memory, visible to all CPUs

Distributed Shared Memory: Problem

- An application has a shared address space; **all memory locations accessible to all instructions**
- Divide code for application into pieces, assign one piece to each of several computers on a LAN
- Each computer has own **separate memory**
- Each piece of code may want to read or write any part of data
- **Where do you put the data?**

Distributed Shared Memory: Solution

- Goal: create illusion that all boxes share single memory, accessible by all
- Shared memory contents divided across nodes
 - Programmer needn't explicitly communicate among nodes
 - Pool memory of all nodes into one shared memory
- Performance? Correctness?
 - Far slower to read/write **across LAN** than read/write from/to local (same host's) memory
 - Remember NFS: caching should help
 - Remember NFS: caching complicates consistency!

Context: Parallel Computation

- Still need to divide program code across multiple CPUs
- Potential benefit: more total CPU power, so faster execution
- Potential risk: how will we know if distributed program executes correctly?
- To understand distributed shared memory, must understand what “correct” execution means...

Simple Case: Uniprocessor Correctness

- When you only have one processor, what does “correct” mean?
- Define “correct” **separately for each instruction**
- Each instruction takes machine from one state to another (e.g., ADD, LD, ST)
 - LD should return value of **most recent ST to same memory address**

Simple Case: Uniprocessor Correctness

“Correct” means:

Execution gives same result as if you ran one instruction at a time, waiting for each to complete

- Each instruction takes machine from one state to another (e.g., ADD, LD, ST)
 - LD should return value of **most recent ST to same memory address**

Why Define Correctness?

- Programmers want to be able to predict how CPU executes program!
 - ...to write correct program
- Note that modern CPUs **don't execute instructions one-at-a-time in program order**
 - Multiple instruction issue
 - Out-of-order instruction issue
- Nevertheless, CPUs must behave such that they obey uniprocessor correctness!

Distributed Correctness: Naïve Shared Memory

- Suppose we have multiple hosts with (for now) naïve shared memory
 - 3 hosts, each with one CPU, connected by Internet
 - Each host has local copy of all memory
 - Reads local, so very fast
 - Writes sent to other hosts (and execution continues immediately)
- **Is naïve shared memory correct?**

Example 1: Mutual Exclusion

Initialization: $x = y = 0$ on both CPUs

CPU0:

```
x = 1;
```

```
if (y == 0)
```

```
    critical section;
```

CPU1:

```
y = 1;
```

```
if (x == 0)
```

```
    critical section;
```

- Why is code correct?
 - If CPU0 sees $y == 0$, CPU1 can't have executed "y = 1"
 - So CPU1 will see $x == 1$, and can't enter critical section

Example 1: Mutual Exclusion

Initialization: $x = y = 0$ on both CPUs

CPU0:

$x = 1;$

if ($y == 0$)

critical section;

CPU1:

$y = 1;$

if ($x == 0$)

critical section;

So CPU0 and CPU1 cannot simultaneously enter critical section

1''

- So CPU1 will see $x == 1$, and can't enter critical section

Naïve Distributed Memory: Incorrect for Example 1

- Problem A:
 - CPU0 sends “write $x=1$ ”, reads local “ $y == 0$ ”
 - CPU1 reads local “ $x == 0$ ” before write arrives
- Local memory and slow writes cause disagreement about read/write order!
 - CPU0 thinks its “ $x = 1$ ” was before CPU1’s read of x
 - CPU1 thinks its read of x was before arrival of “write $x = 1$ ”
- Both CPU0 and CPU1 enter critical section!

Example 2: Data Dependencies

CPU0:

```
v0 = f0();
```

```
done0 = true;
```

CPU1:

```
while (done0 == false)
```

```
;
```

```
v1 = f1(v0);
```

```
done1 = true;
```

CPU2:

```
while (done1 == false)
```

```
;
```

```
v2 = f2(v0, v1);
```

Example 2: Data Dependencies

CPU0:

```
v0 = f0();  
done0 = true;
```

CPU1:

```
while (done0 == false)  
    ;  
v1 = f1(v0);  
done1 = true;
```

CPU2:

```
while (done1 == false)  
    ;  
v2 = f2(v0, v1);
```

Intent:

**CPU2 should run f2() with
results from CPU0 and CPU1
Waiting for CPU1 implies
waiting for CPU0**

Naïve Distributed Memory: Incorrect for Example 2

- Problem B:
 - CPU0's writes of `v0` and `done0` may be reordered by network, leaving `v0` unset, but `done0` true
- But even if each CPU sees each other CPU's writes in issue order...
- Problem C:
 - CPU2 sees CPU1's writes before CPU0's writes
 - i.e., CPU2 and CPU1 disagree on order of CPU0's and CPU1's writes

Naïve Distributed Memory: Incorrect for Example 2

- Problem B:

**Naïve distributed memory isn't correct
(Or we shouldn't expect code like these
examples to work...)**

CPU's writes in issue order...

- Problem C:
 - CPU2 sees CPU1's writes before CPU0's writes
 - i.e., CPU2 and CPU1 disagree on order of CPU0's and CPU1's writes

Distributed Correctness: Consistency Models

- How can we write correct distributed programs with shared storage?
- Need to define rules that memory system will follow
- Need to write programs with these rules in mind
- Rules are a consistency model
- Build memory system to obey model; programs that assume model then correct

How Do We Choose a Consistency Model?

- No such thing as “right” or “wrong” model
 - All models are artificial definitions
- Different models may be harder or easier to program for
 - Some models produce behavior that is more intuitive than others
- Different models may be harder or easier to implement efficiently
 - Performance vs. semantics trade-off, as with NFS/RPC

Back to Ivy: What's It Good For?

- Suppose you've got 100 PCs on a LAN and shared memory across all of them
- **Fast, parallel sorting program:**
 - Load entire array into shared memory
 - Each PC processes one section of array
 - On PC i:
 - sort own piece of array
 - done[i] = true;
 - wait for all done[] to be true
 - merge my piece of array with my neighbors'...

Partitioning Address Space: Fixed Approach

- Fixed approach:
 - First MB on host 0, 2nd on host 1, &c.
 - Send all reads and writes to “owner” of address
 - Each CPU read- and write-protects pages in address ranges held by other CPUs
 - Detect reads and writes to remote pages with VM hardware
- What if we placed pages on hosts poorly?
- Can't always predict which hosts will use which pages

Partitioning Address Space: Dynamic, Single-Copy Approach

- Move the page to the reading/writing CPU each time it is used
- CPU trying to read or write must find current owner, then take page from it
- Requires mechanism to find current location of page
- **What if many CPUs read same page?**

Partitioning Address Space: Dynamic, Multi-Copy Approach

- Move page for **writes**, but allow **read-only copies**
- When CPU reads page it doesn't have in its own local memory, find other CPU that most recently wrote to page
- Works if pages are **read-only and shared** or **read-write by one host**
- **Bad case: write sharing**
 - When does write sharing occur?
 - False sharing, too...

Simple Ivy: Centralized Manager (Section 3.1)

CPU0

lock	access	owner?

CPU1

lock	access	owner?

CPU2 / MGR

lock	access	owner?

ptable

- ptable (all CPUs)
access: R, W, or nil
owner: T or F
- info (MGR only)
copy_set: list of CPUs with read-only copies
owner: CPU that can write page

lock	copy_set	owner

info

Centralized Manager (2): Message Types Between CPUs

- RQ (read query, reader to MGR)
- RF (read forward, MGR to owner)
- RD (read data, owner to reader)
- RC (read confirm, reader to MGR)
- WQ (write query, writer to MGR)
- IV (invalidate, MGR to copy_set)
- IC (invalidate confirm, copy_set to MGR)
- WF (write forward, MGR to owner)
- WD (write data, owner to writer)
- WC (write confirm, writer to MGR)

Centralized Manager Example 1: Owned by CPU0, CPU1 wants to read

CPU0

lock	access	owner?
F	W	T
...		

CPU1

lock	access	owner?
F	nil	F
...		

CPU2 / MGR

lock	access	owner?
F	nil	F
...		

lock	copy_set	owner
F	{}	CPU0
...		

ptable

info

Centralized Manager Example 1: Owned by CPU0, CPU1 wants to read

CPU0

lock	access	owner?
F	W	T
...		

CPU1



lock	access	owner?
F	nil	F
...		

CPU2 / MGR

lock	access	owner?
F	nil	F
...		

lock	copy_set	owner
F	{ }	CPU0
...		

ptable

info

Centralized Manager Example 1: Owned by CPU0, CPU1 wants to read

CPU0

lock	access	owner?
F	W	T
...		

CPU1



lock	access	owner?
T	nil	F
...		

CPU2 / MGR

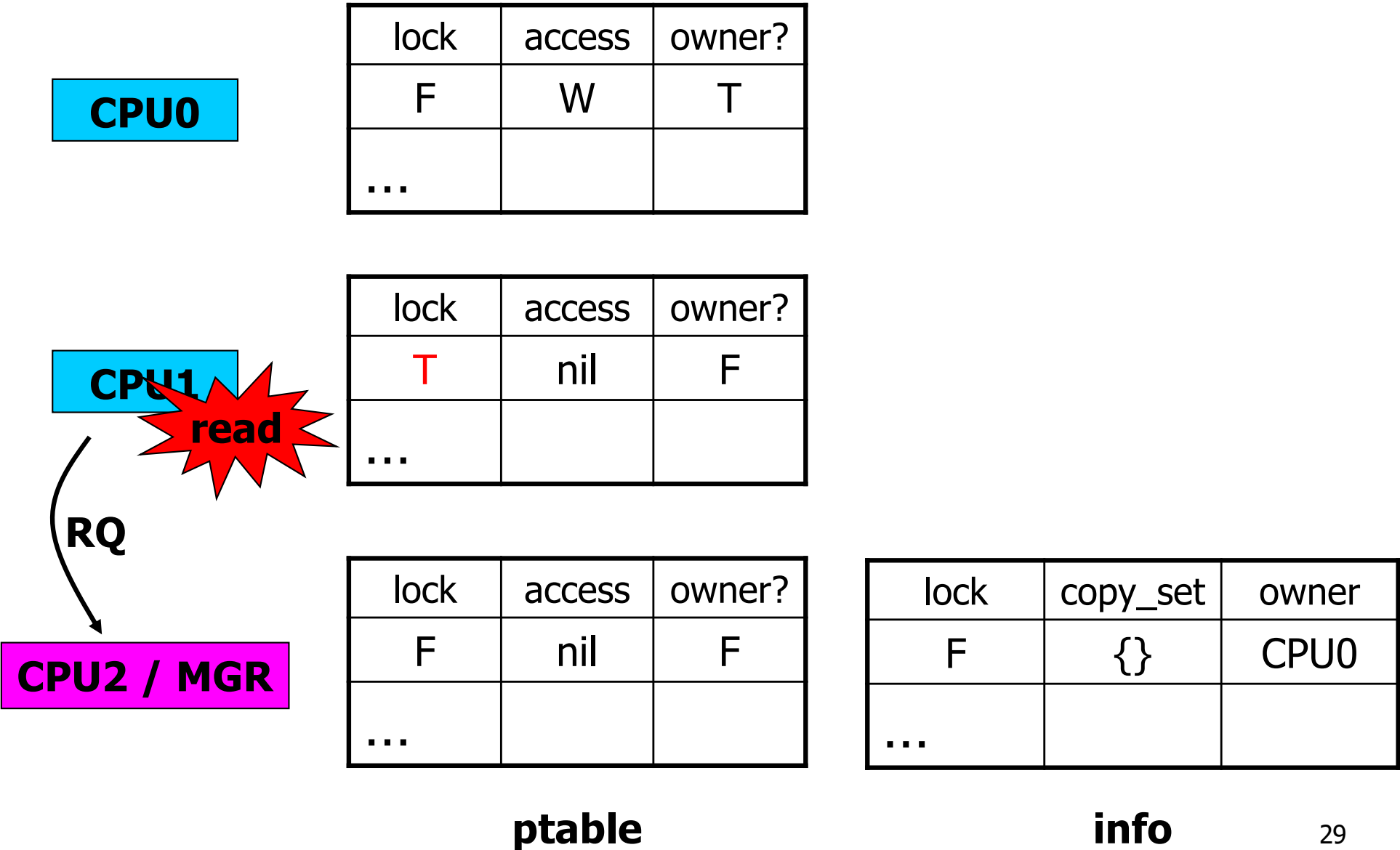
lock	access	owner?
F	nil	F
...		

lock	copy_set	owner
F	{ }	CPU0
...		

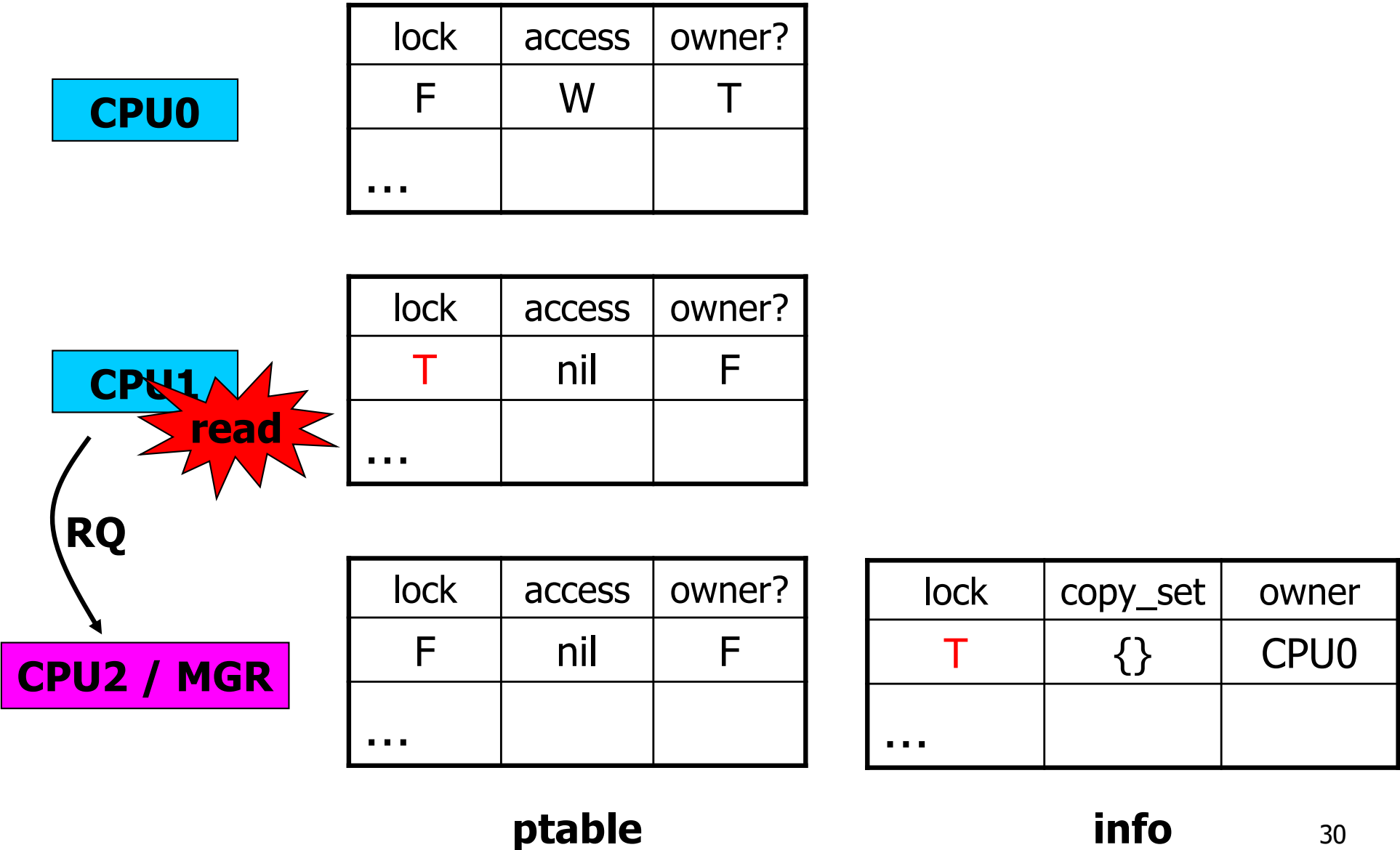
ptable

info

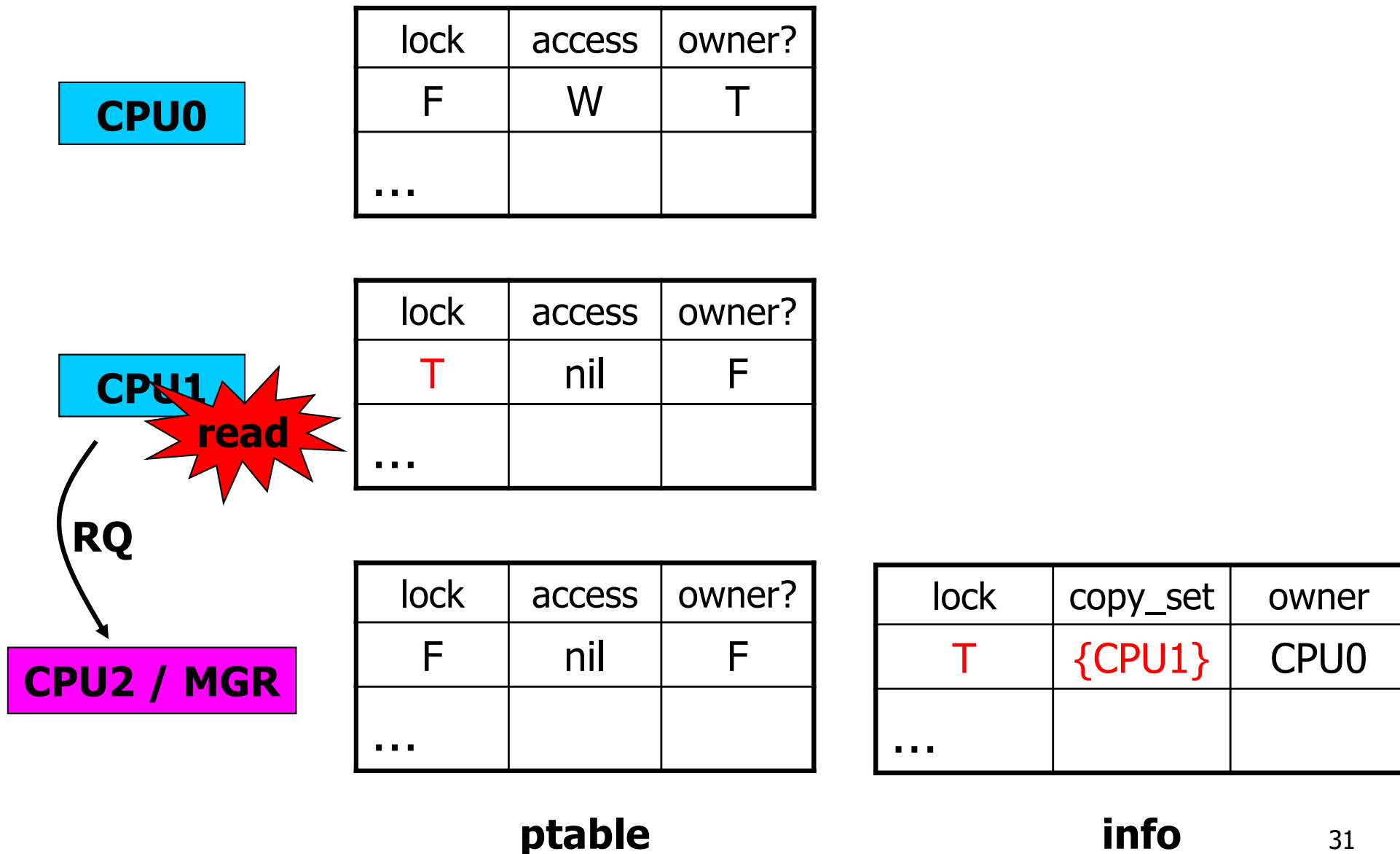
Centralized Manager Example 1: Owned by CPU0, CPU1 wants to read



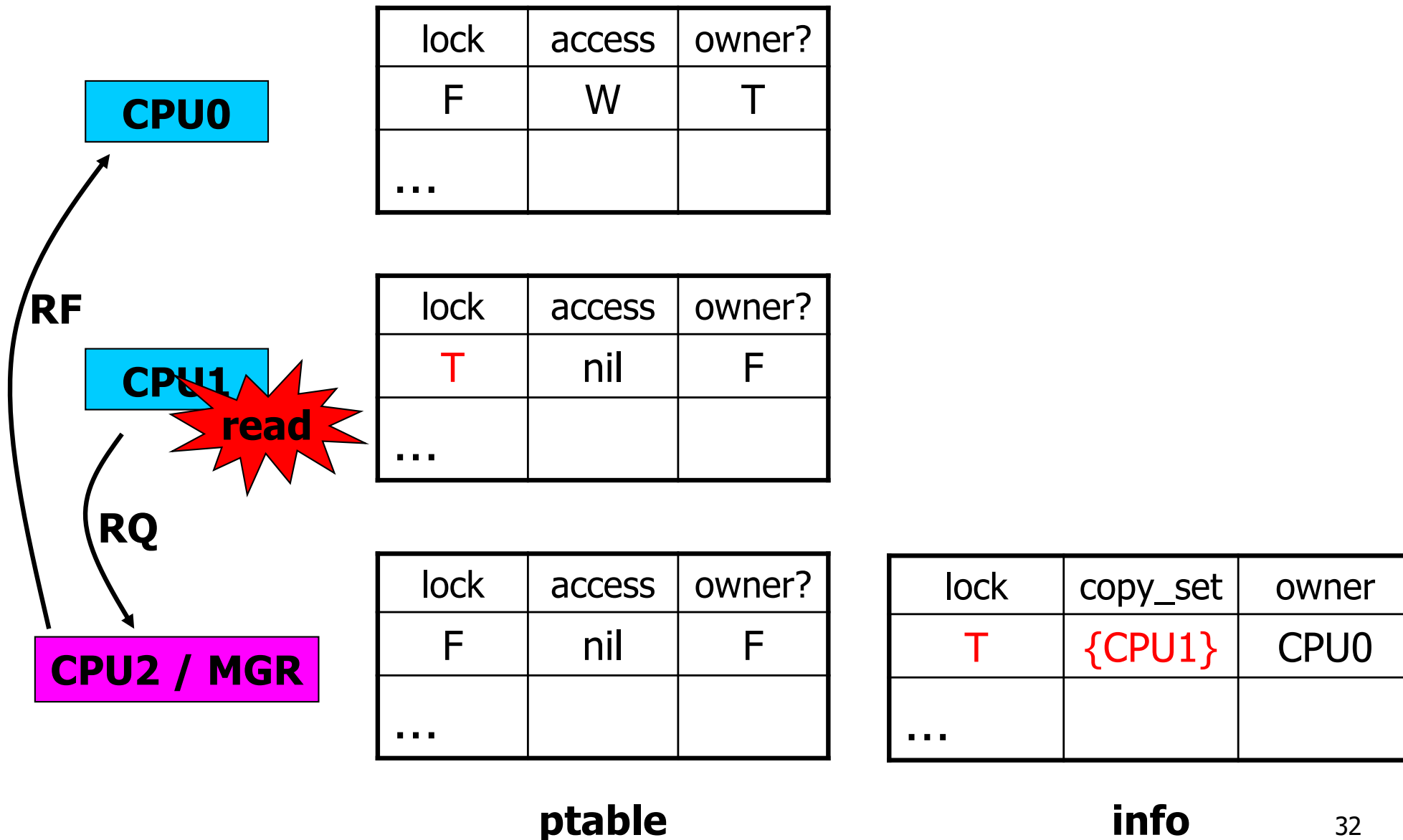
Centralized Manager Example 1: Owned by CPU0, CPU1 wants to read



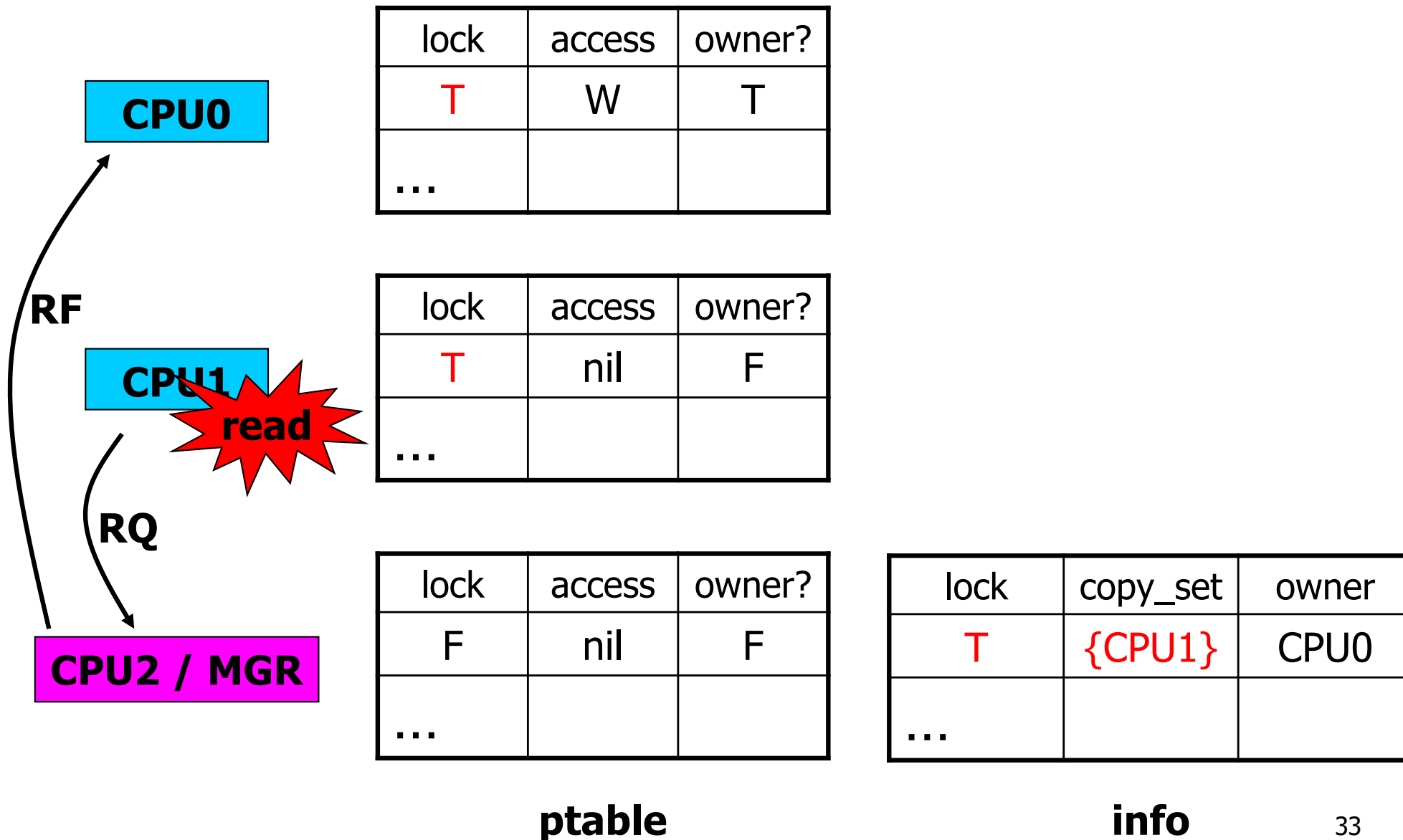
Centralized Manager Example 1: Owned by CPU0, CPU1 wants to read



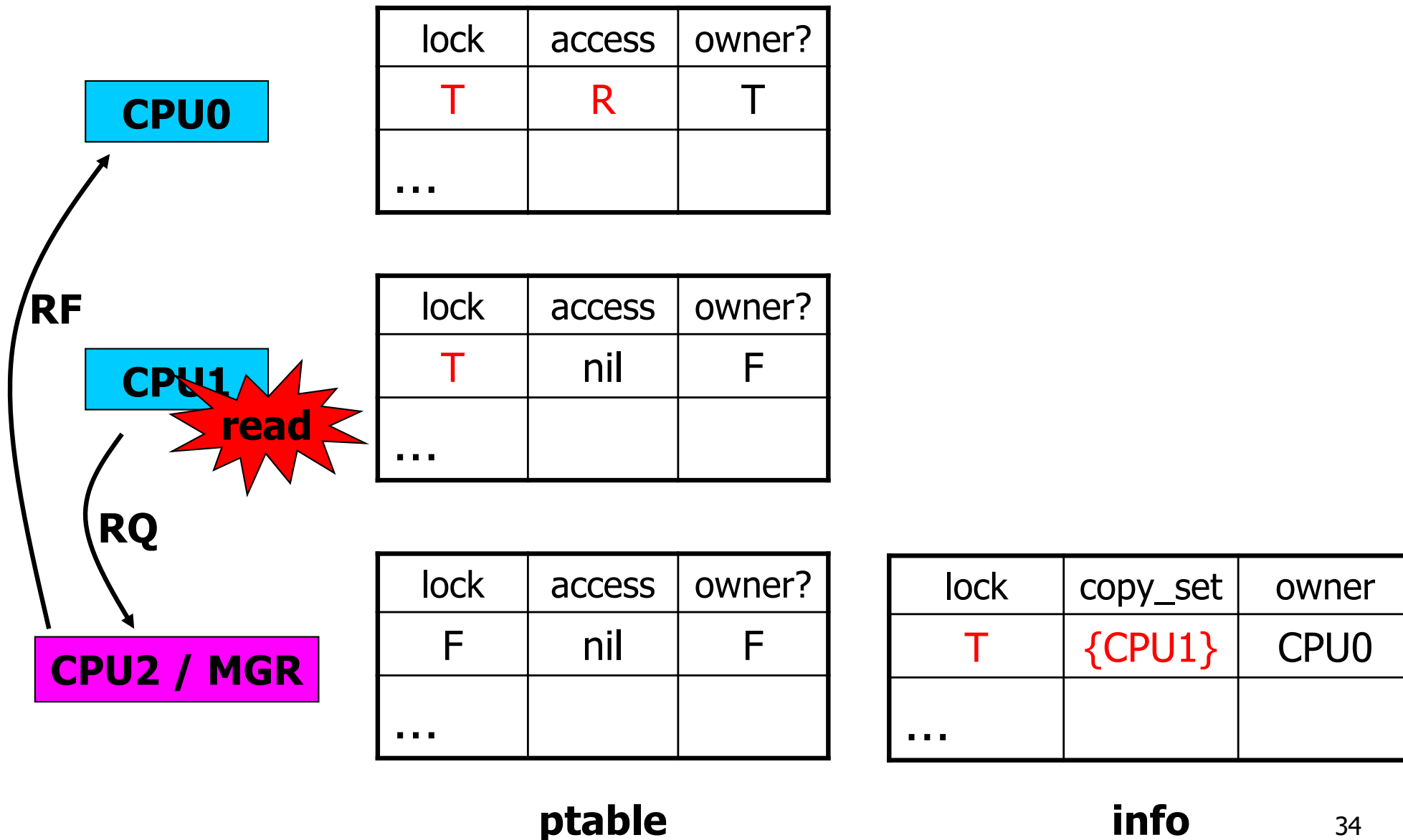
Centralized Manager Example 1: Owned by CPU0, CPU1 wants to read



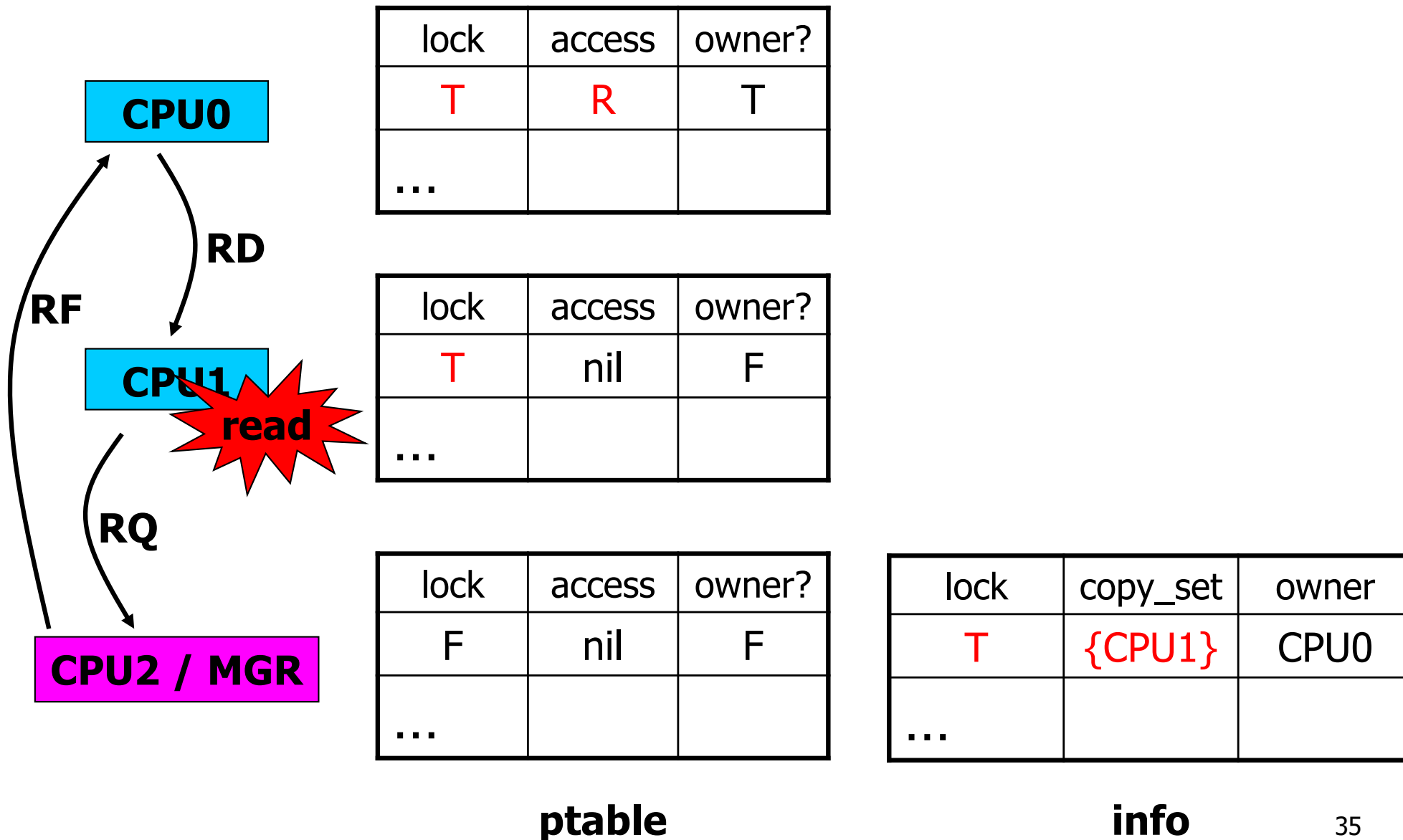
Centralized Manager Example 1: Owned by CPU0, CPU1 wants to read



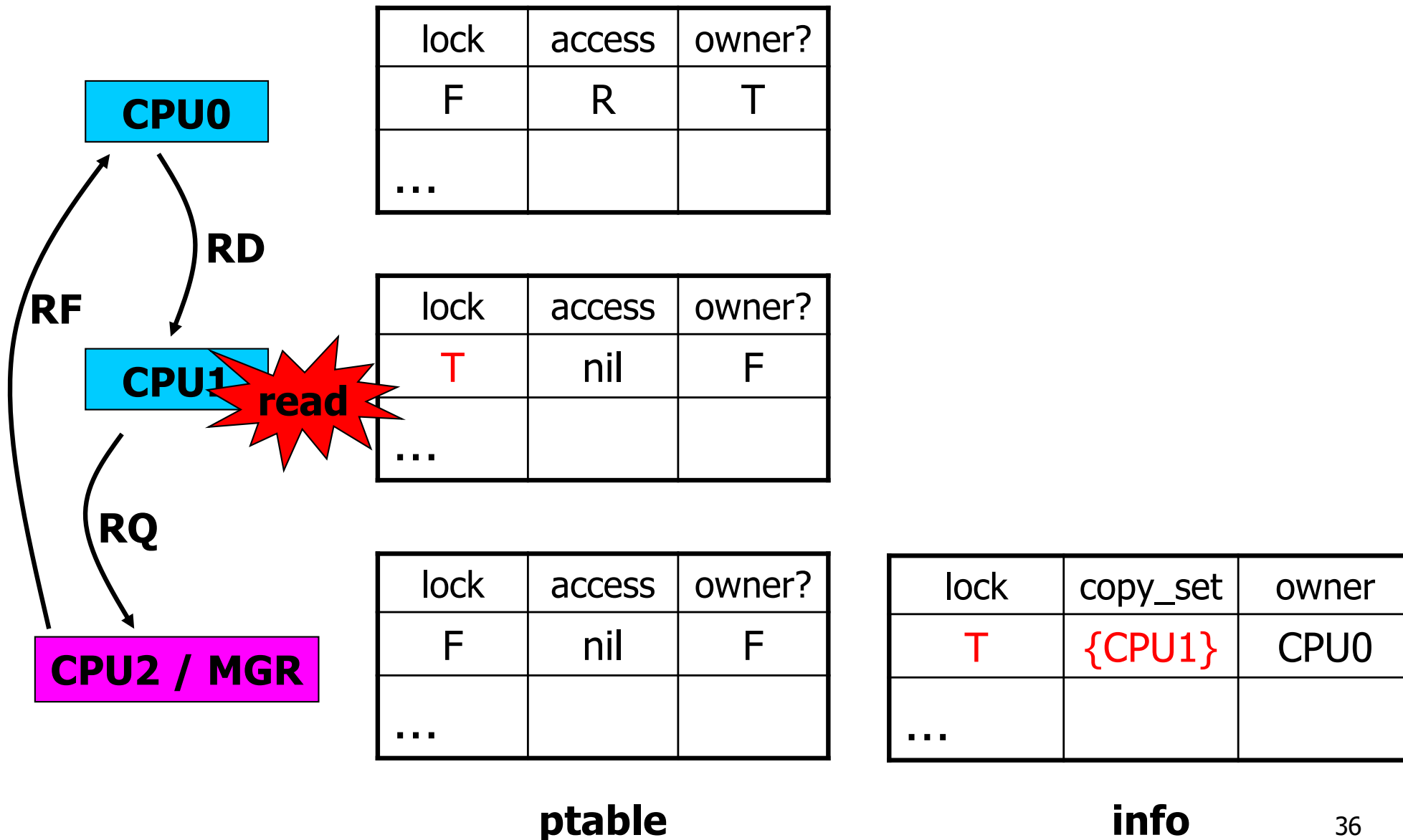
Centralized Manager Example 1: Owned by CPU0, CPU1 wants to read



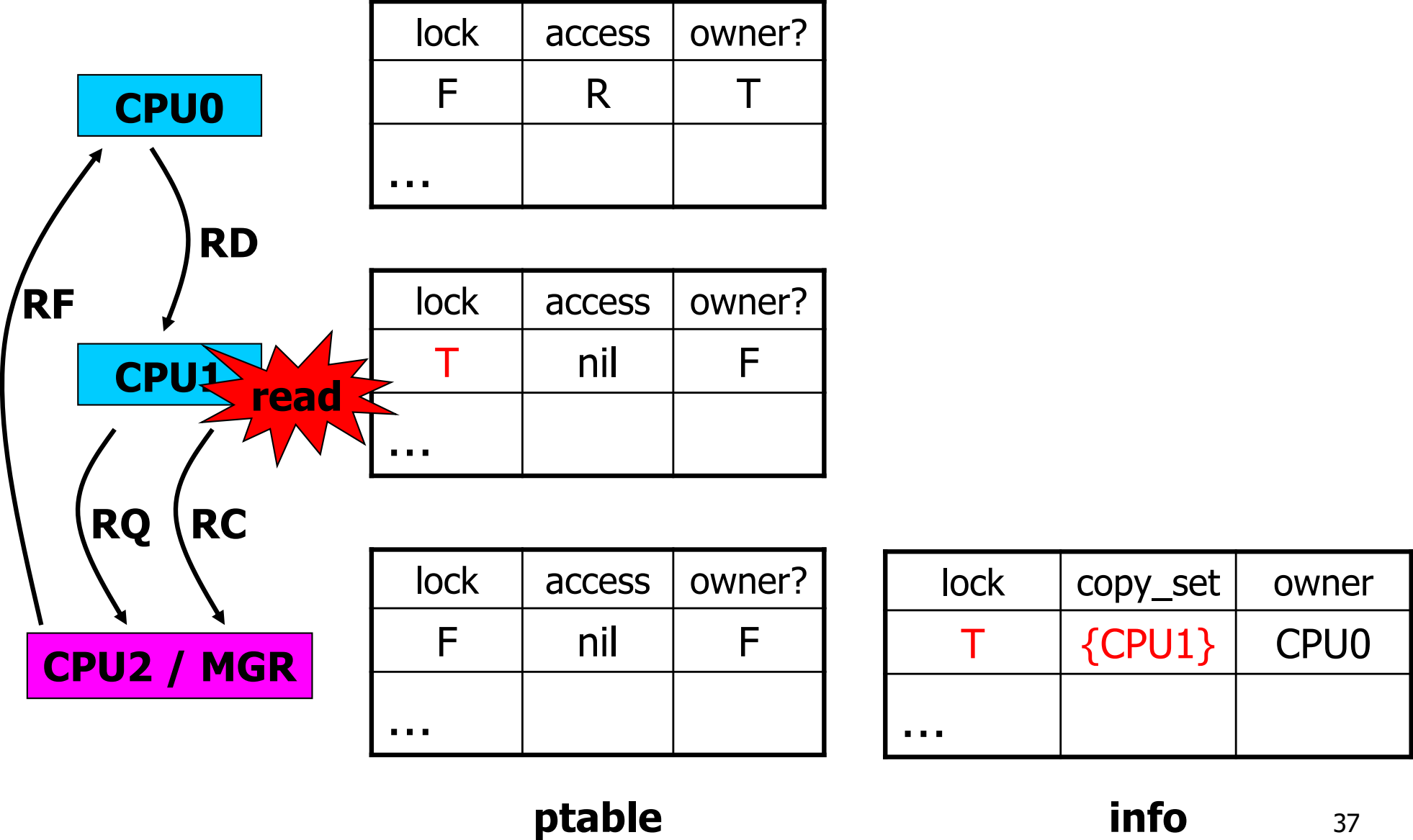
Centralized Manager Example 1: Owned by CPU0, CPU1 wants to read



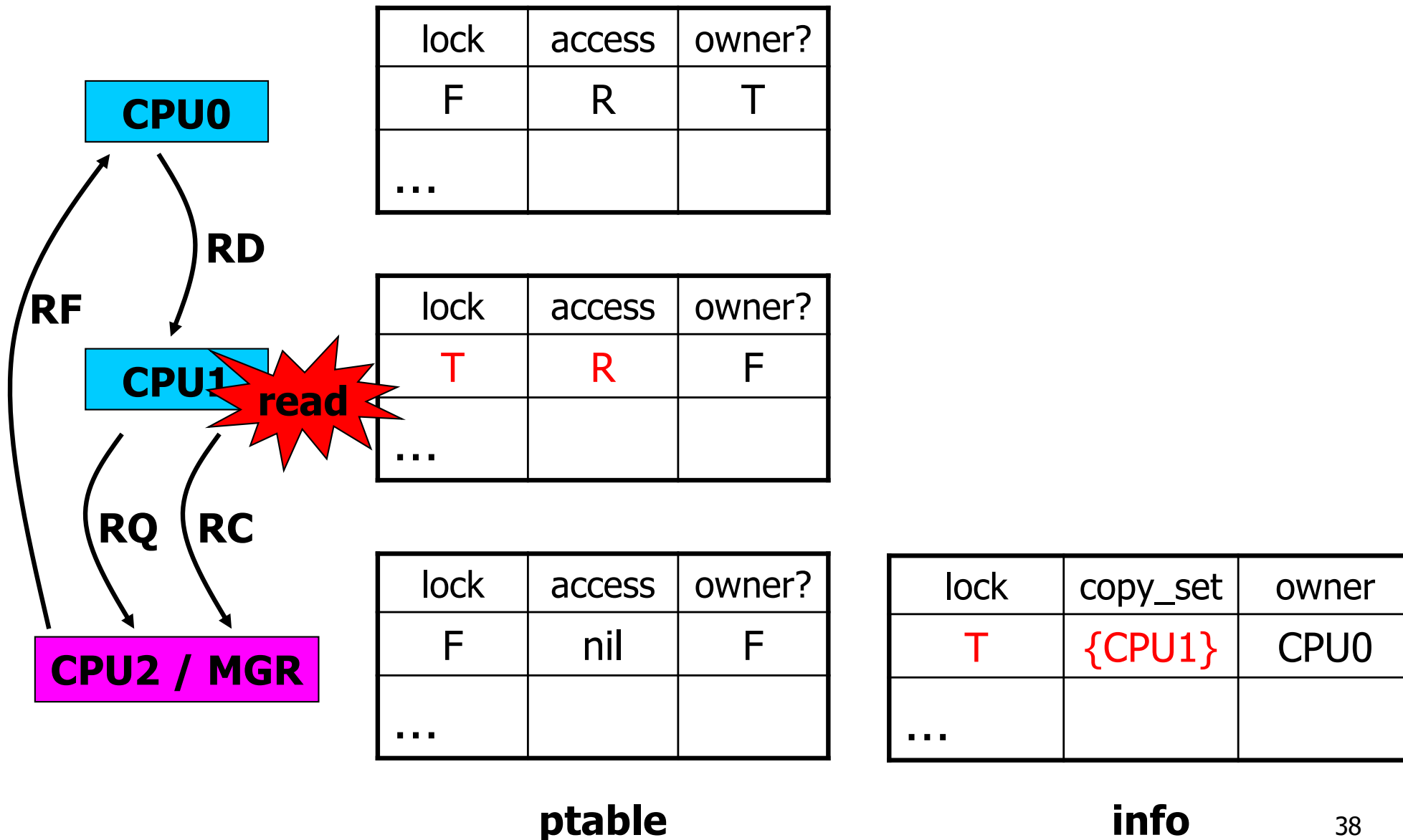
Centralized Manager Example 1: Owned by CPU0, CPU1 wants to read



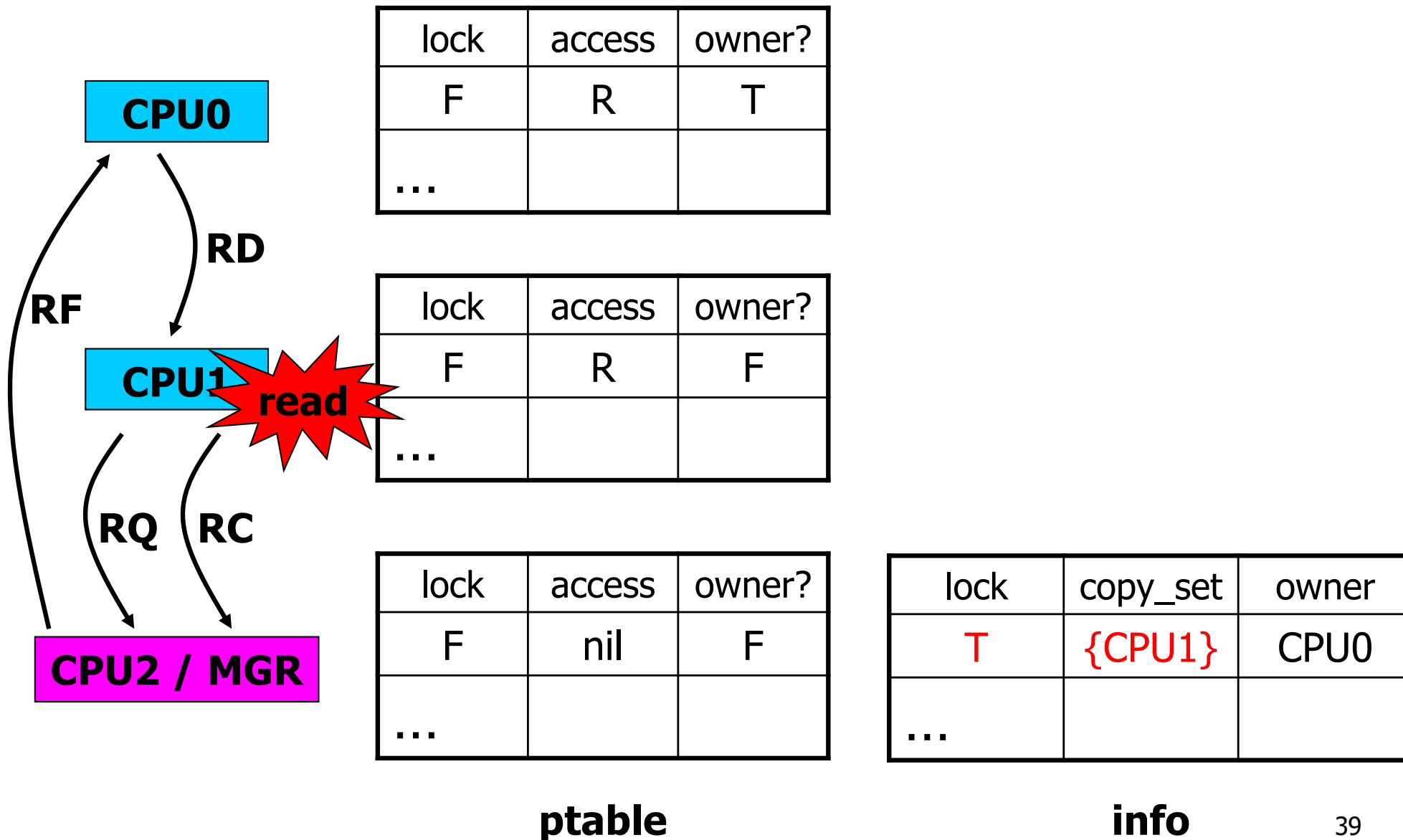
Centralized Manager Example 1: Owned by CPU0, CPU1 wants to read



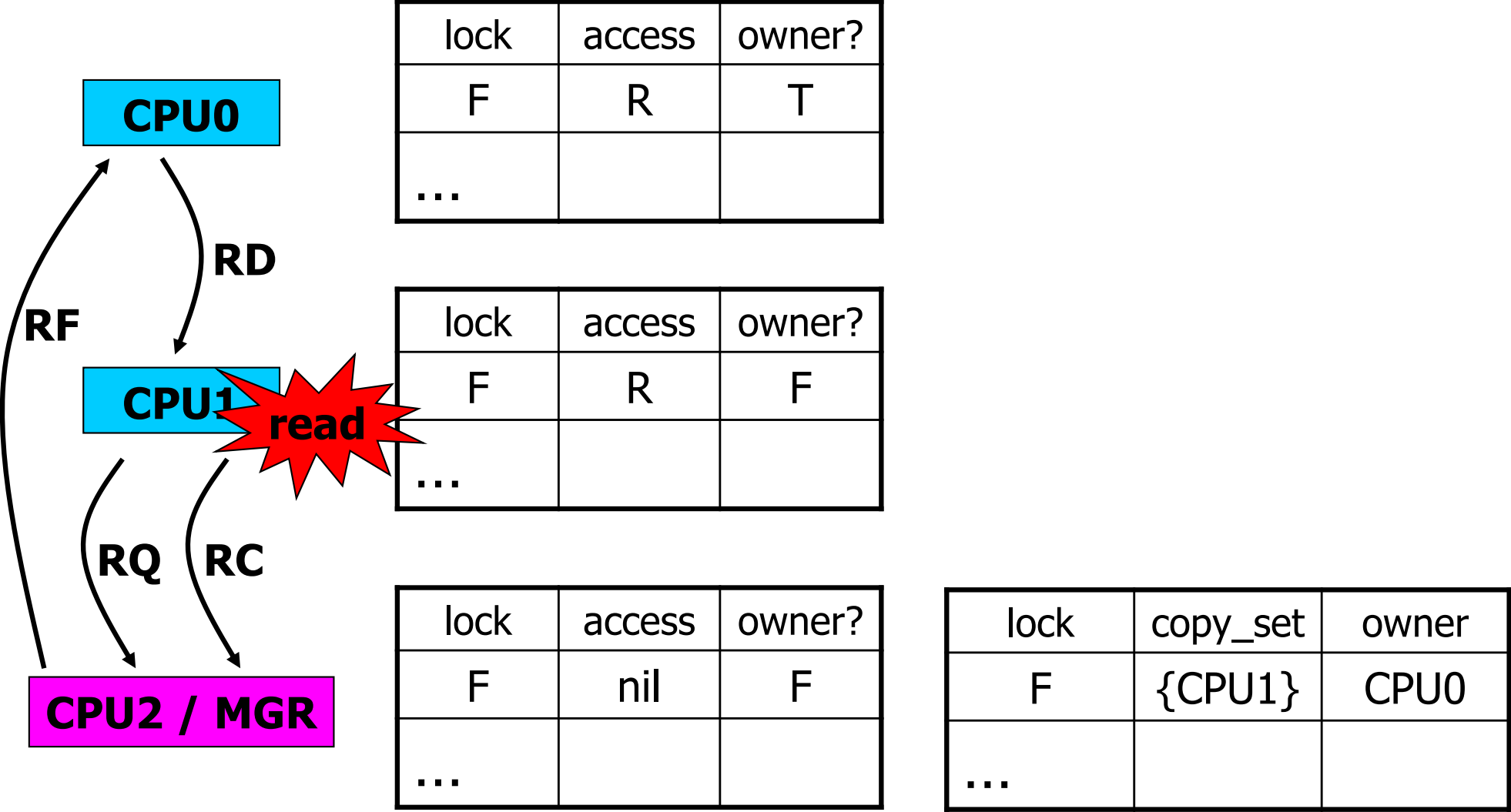
Centralized Manager Example 1: Owned by CPU0, CPU1 wants to read



Centralized Manager Example 1: Owned by CPU0, CPU1 wants to read



Centralized Manager Example 1: Owned by CPU0, CPU1 wants to read



ptable

info

Centralized Manager Example 2: Owned by CPU0, CPU2 wants to write

CPU0

lock	access	owner?
F	R	T
...		

CPU1

lock	access	owner?
F	R	F
...		

CPU2 / MGR



lock	access	owner?
F	nil	F
...		

ptable

lock	copy_set	owner
F	{CPU1}	CPU0
...		

info

Centralized Manager Example 2: Owned by CPU0, CPU2 wants to write

CPU0

lock	access	owner?
F	R	T
...		

CPU1

lock	access	owner?
F	R	F
...		

CPU2 / MGR



lock	access	owner?
T	nil	F
...		

lock	copy_set	owner
F	{CPU1}	CPU0
...		

ptable

info

Centralized Manager Example 2: Owned by CPU0, CPU2 wants to write

CPU0

lock	access	owner?
F	R	T
...		

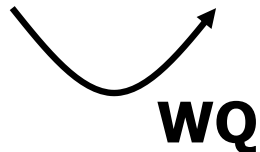
CPU1

lock	access	owner?
F	R	F
...		

CPU2 / MGR



lock	access	owner?
T	nil	F
...		



ptable

lock	copy_set	owner
F	{CPU1}	CPU0
...		

info

Centralized Manager Example 2: Owned by CPU0, CPU2 wants to write

CPU0

lock	access	owner?
F	R	T
...		

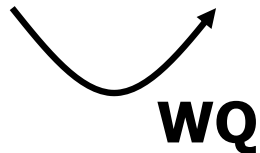
CPU1

lock	access	owner?
F	R	F
...		

CPU2 / MGR



lock	access	owner?
T	nil	F
...		

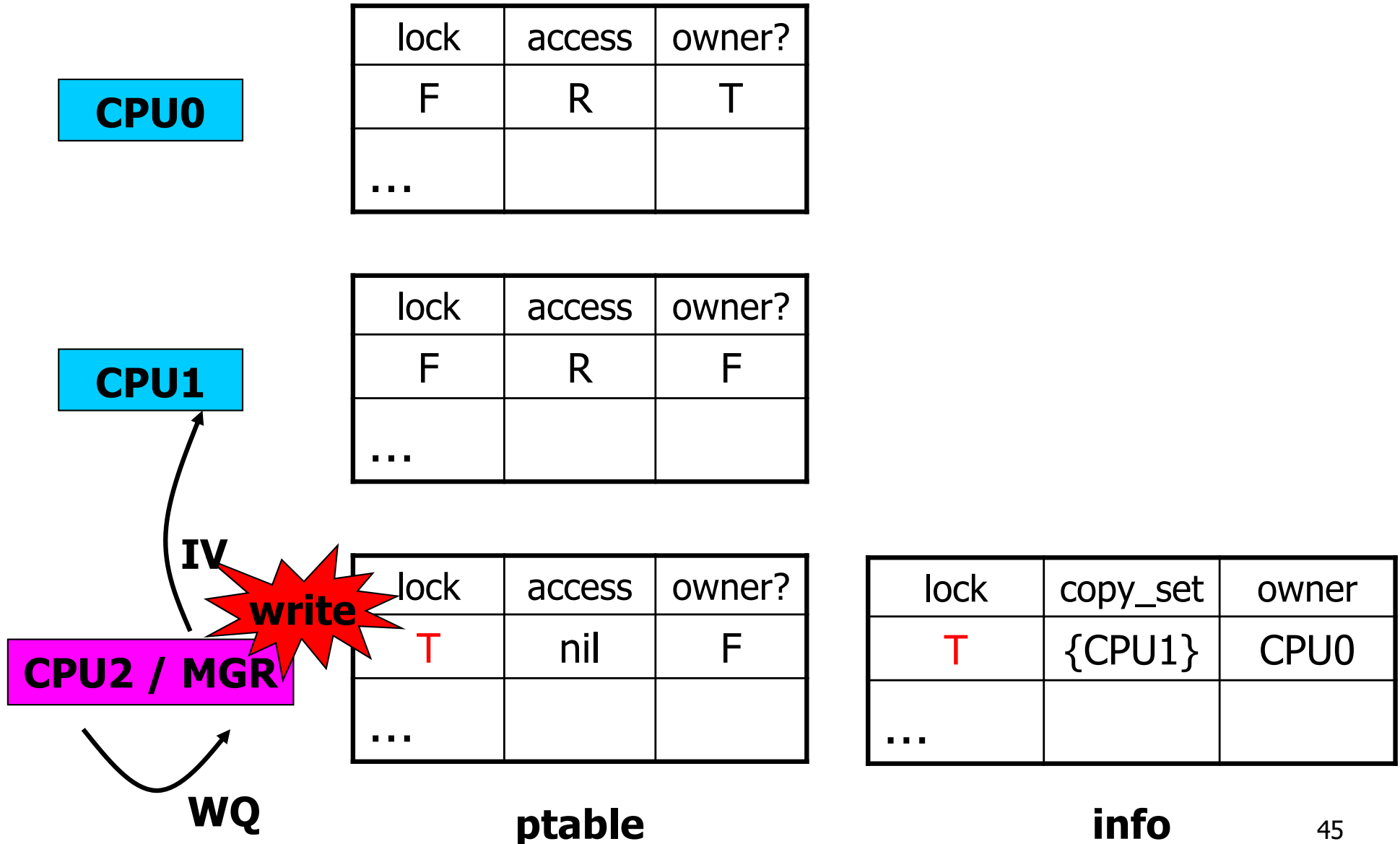


ptable

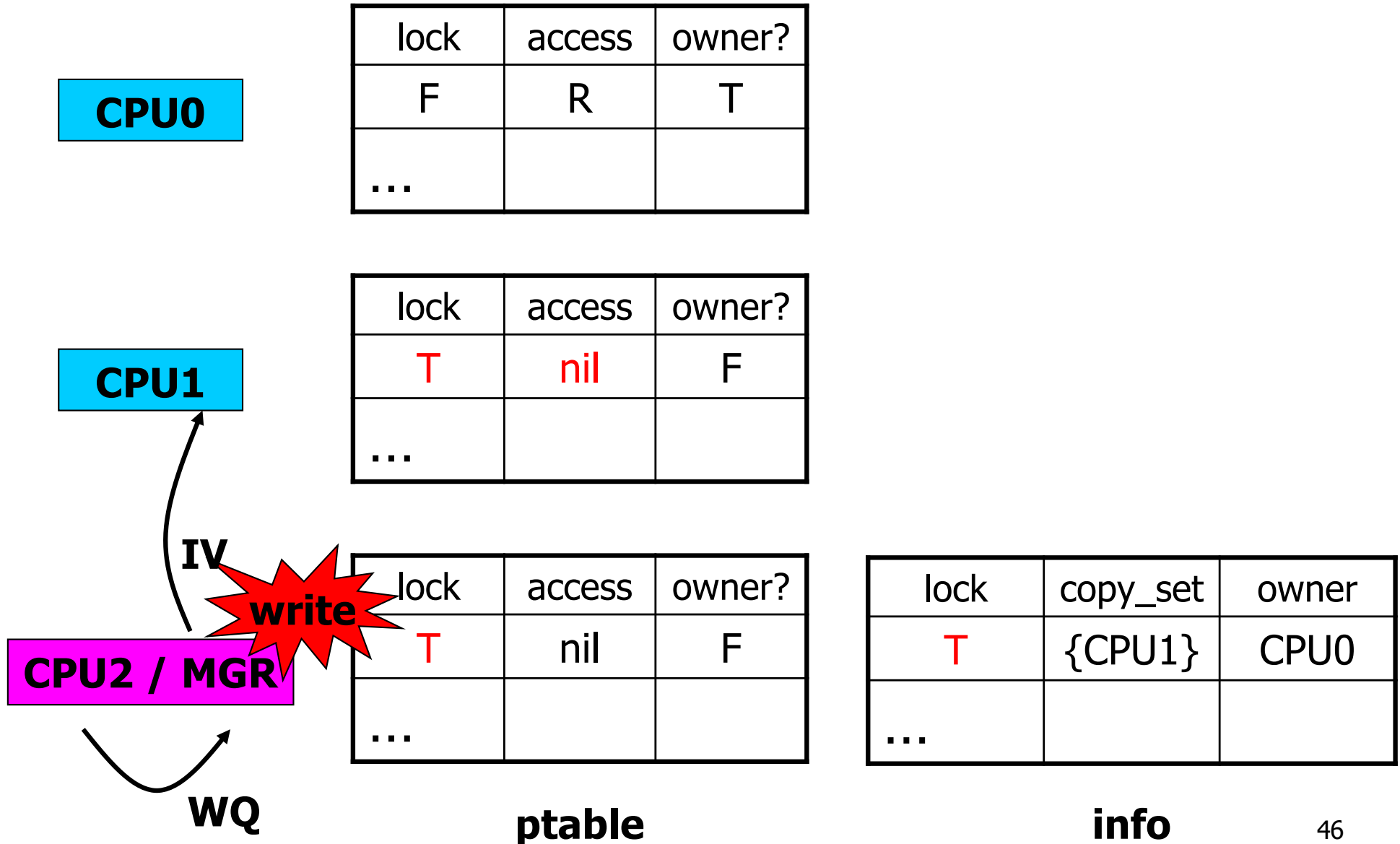
lock	copy_set	owner
T	{CPU1}	CPU0
...		

info

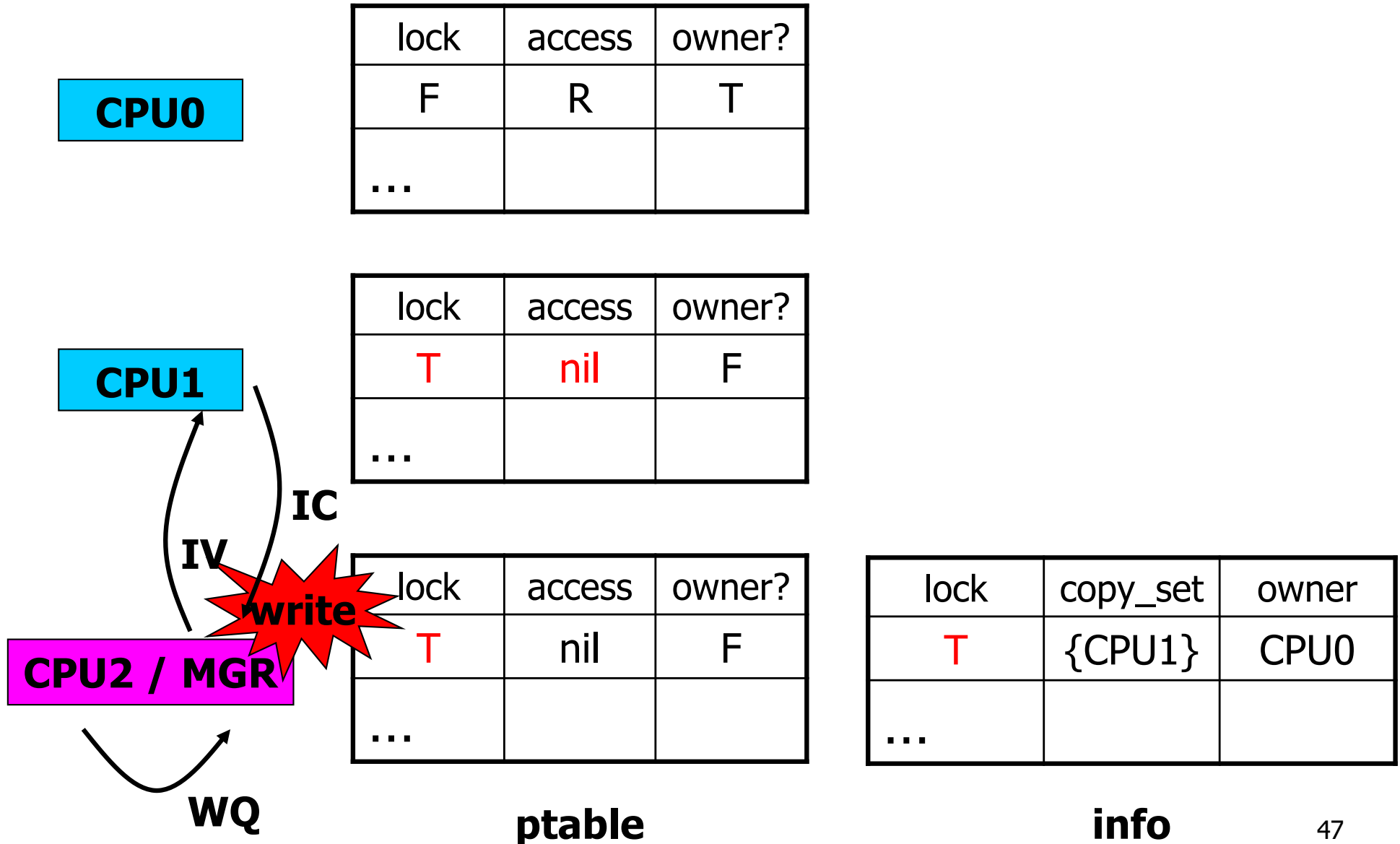
Centralized Manager Example 2: Owned by CPU0, CPU2 wants to write



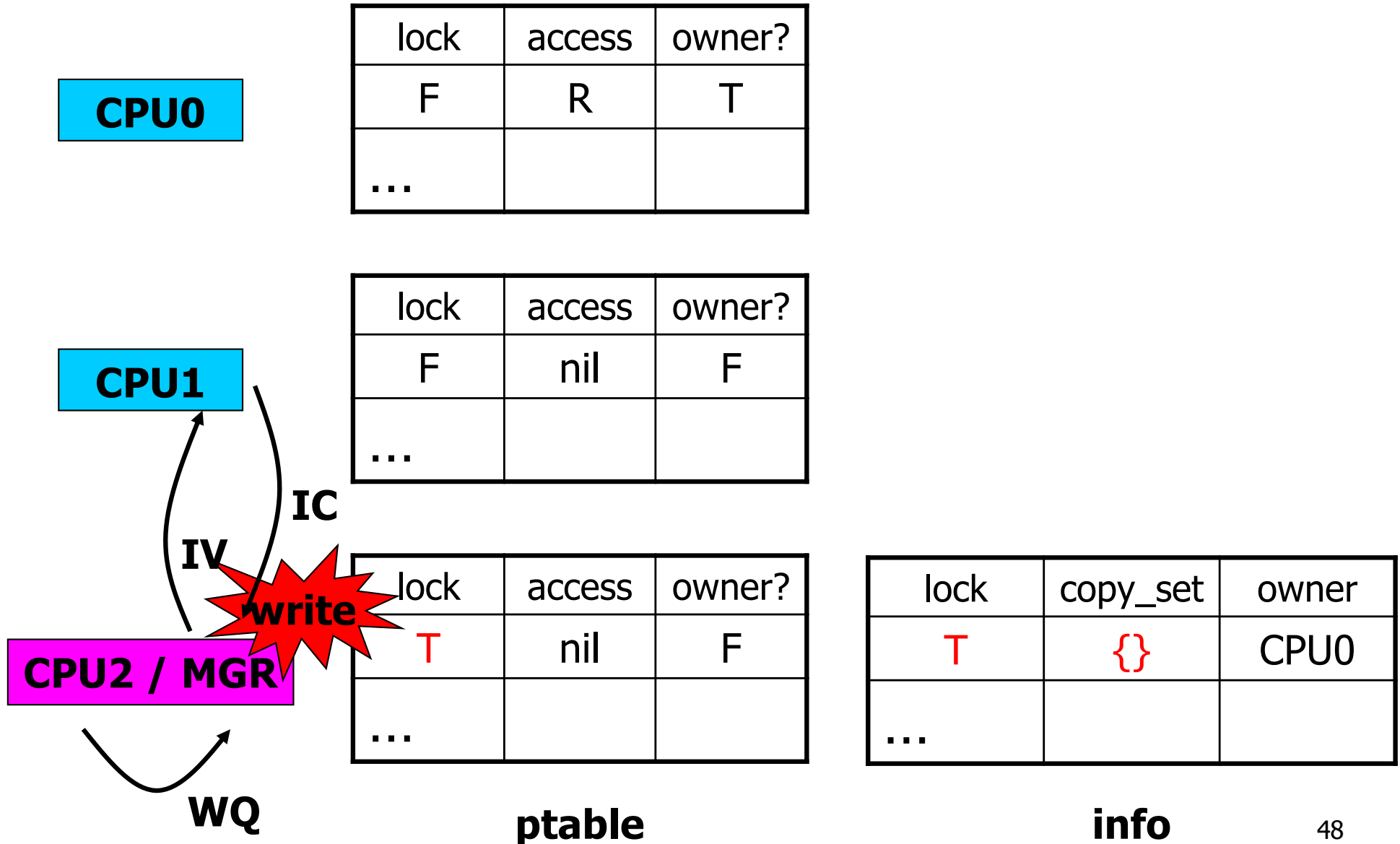
Centralized Manager Example 2: Owned by CPU0, CPU2 wants to write



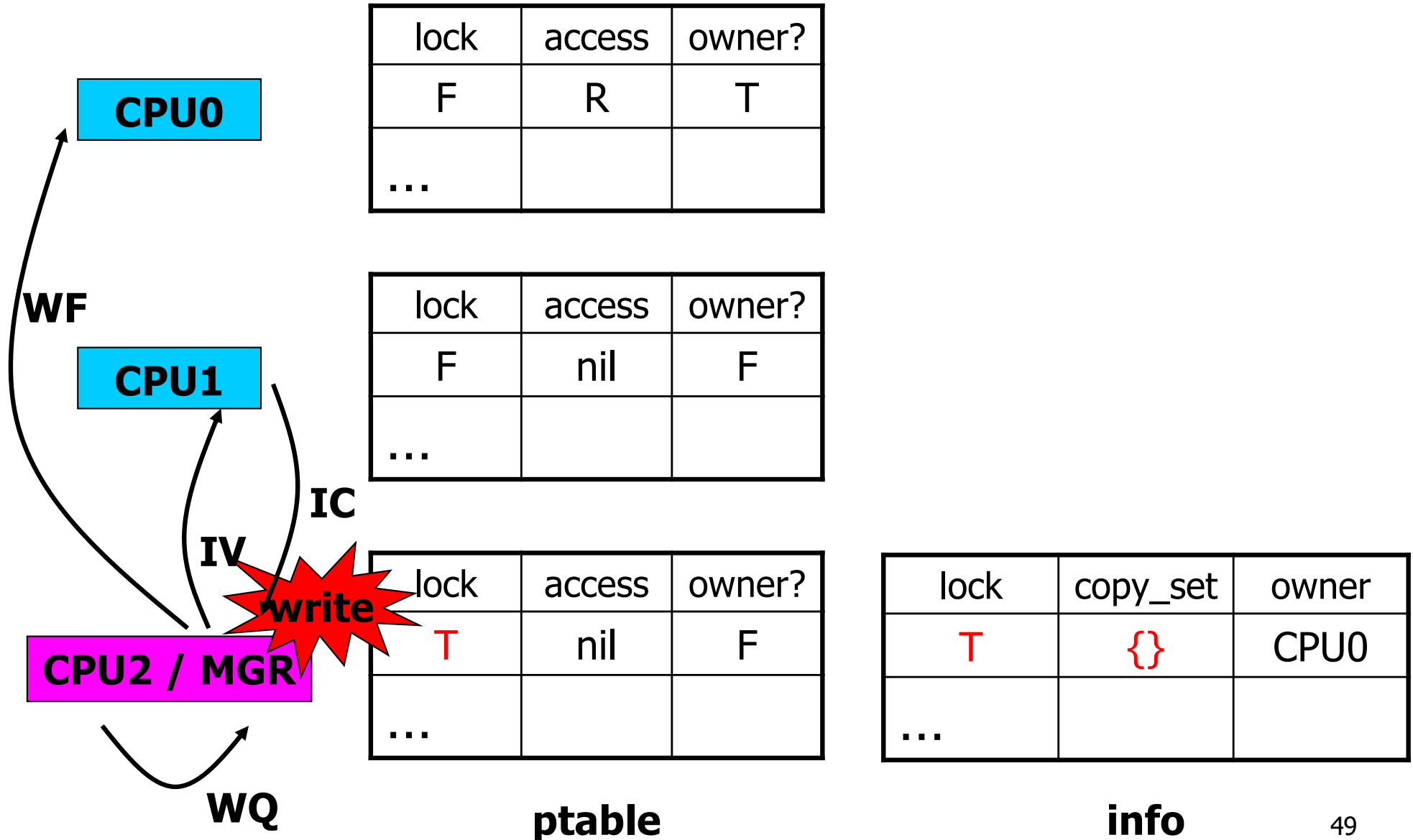
Centralized Manager Example 2: Owned by CPU0, CPU2 wants to write



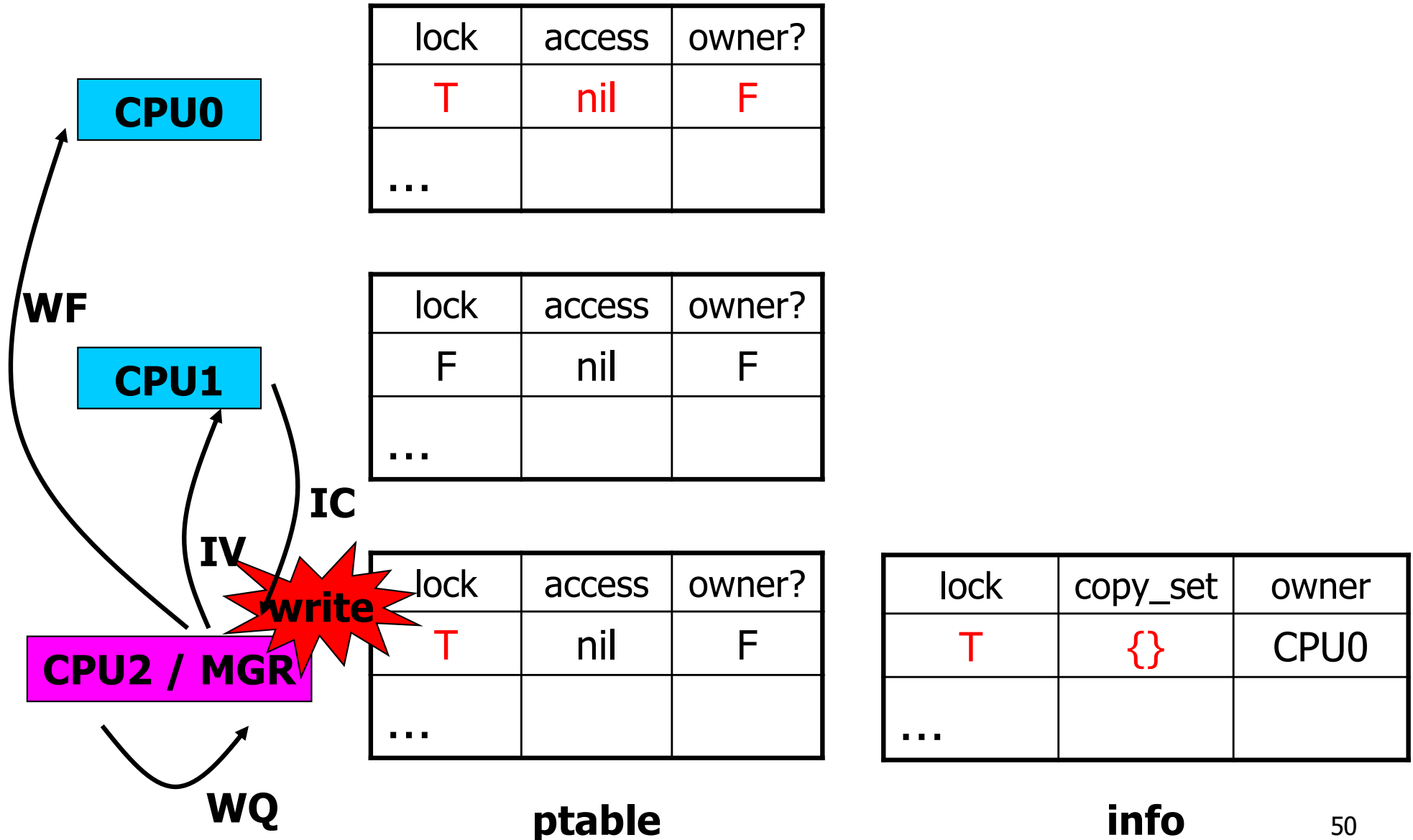
Centralized Manager Example 2: Owned by CPU0, CPU2 wants to write



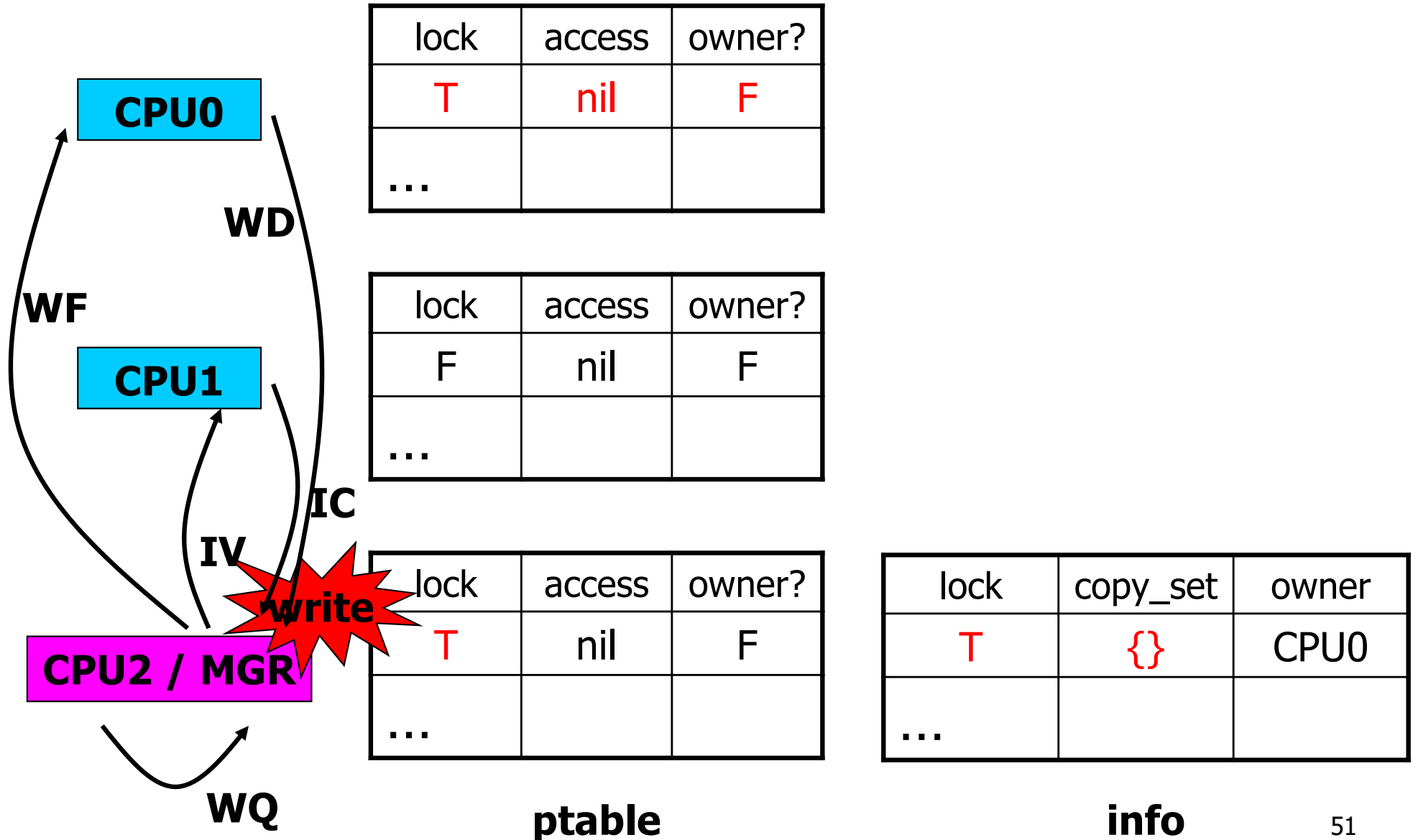
Centralized Manager Example 2: Owned by CPU0, CPU2 wants to write



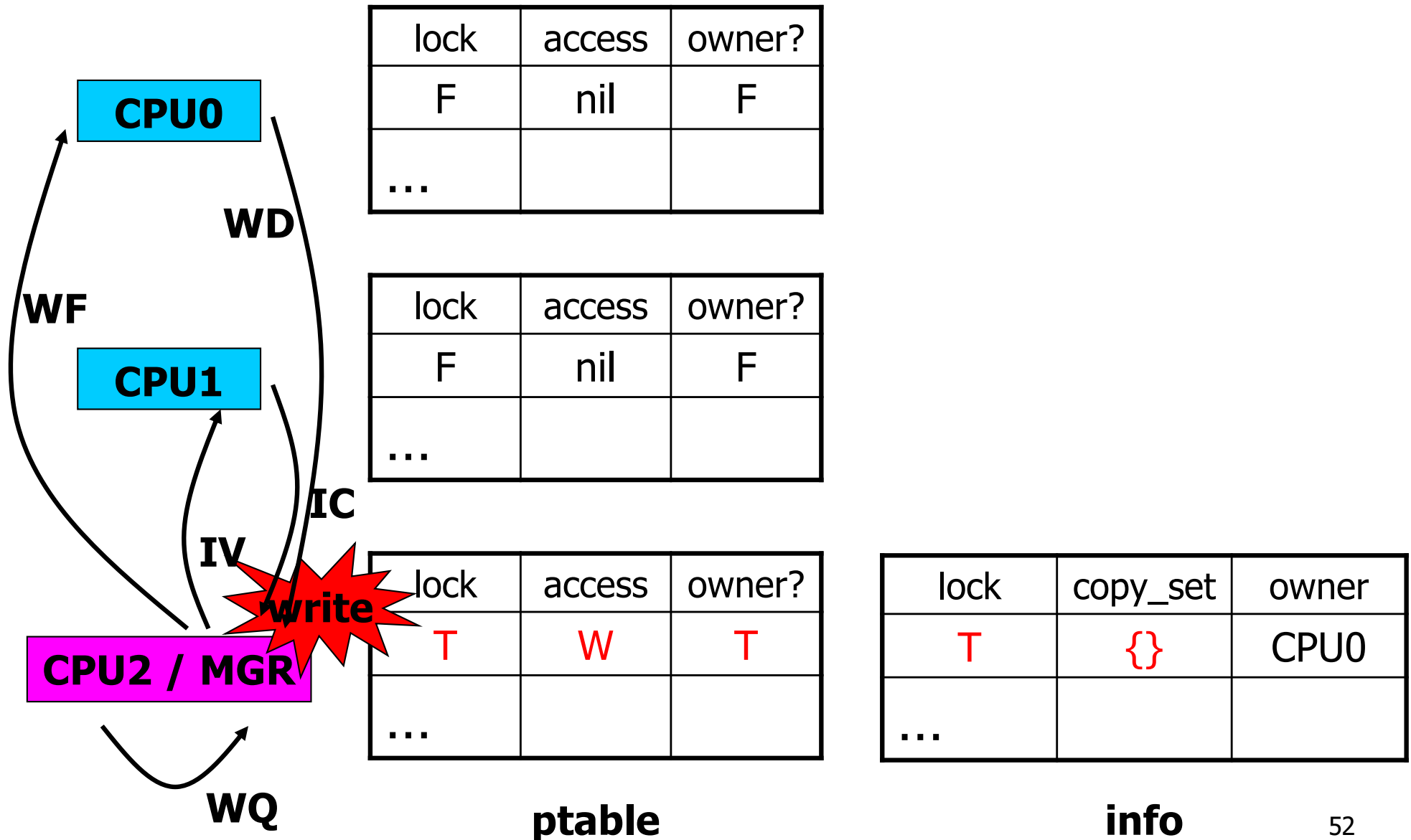
Centralized Manager Example 2: Owned by CPU0, CPU2 wants to write



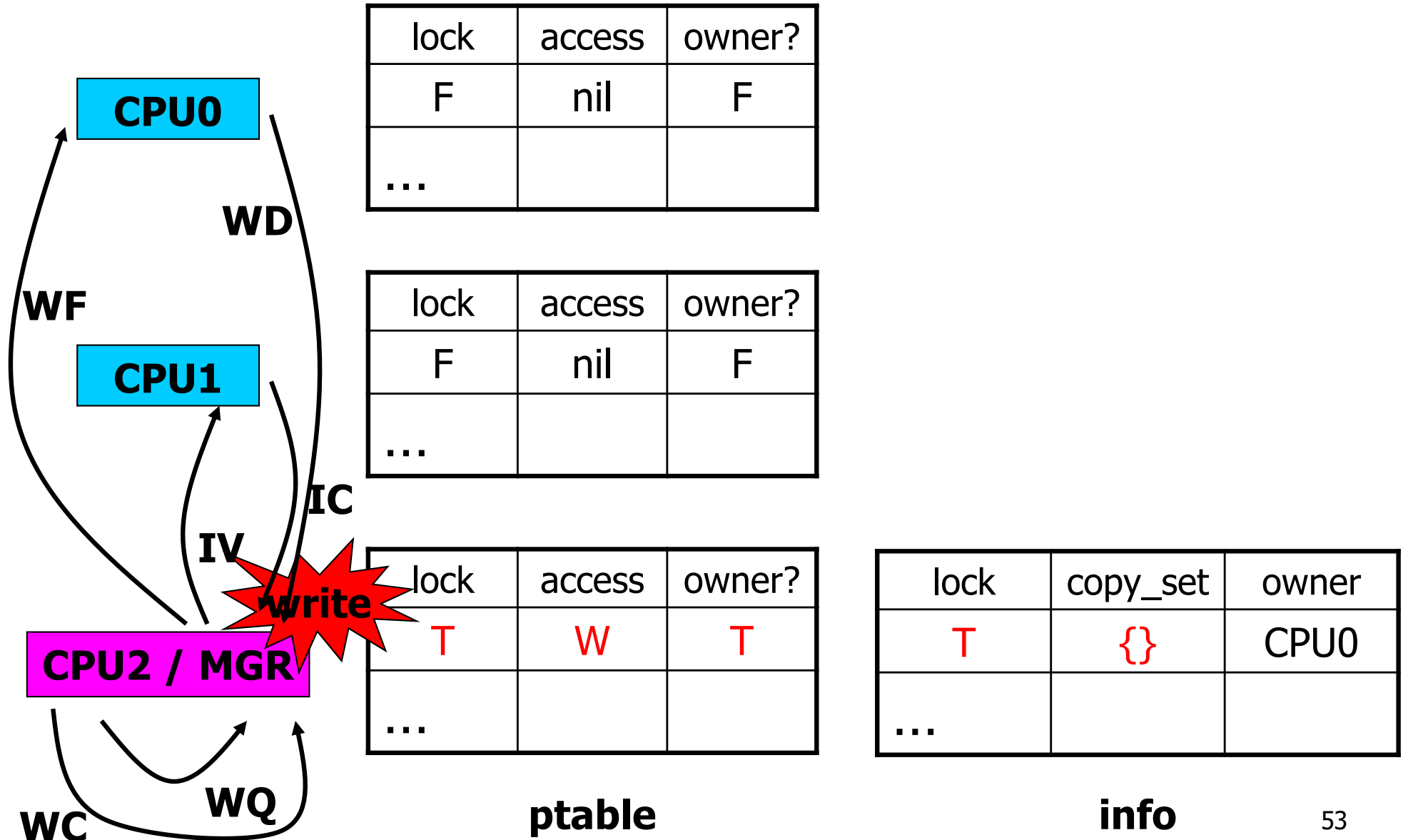
Centralized Manager Example 2: Owned by CPU0, CPU2 wants to write



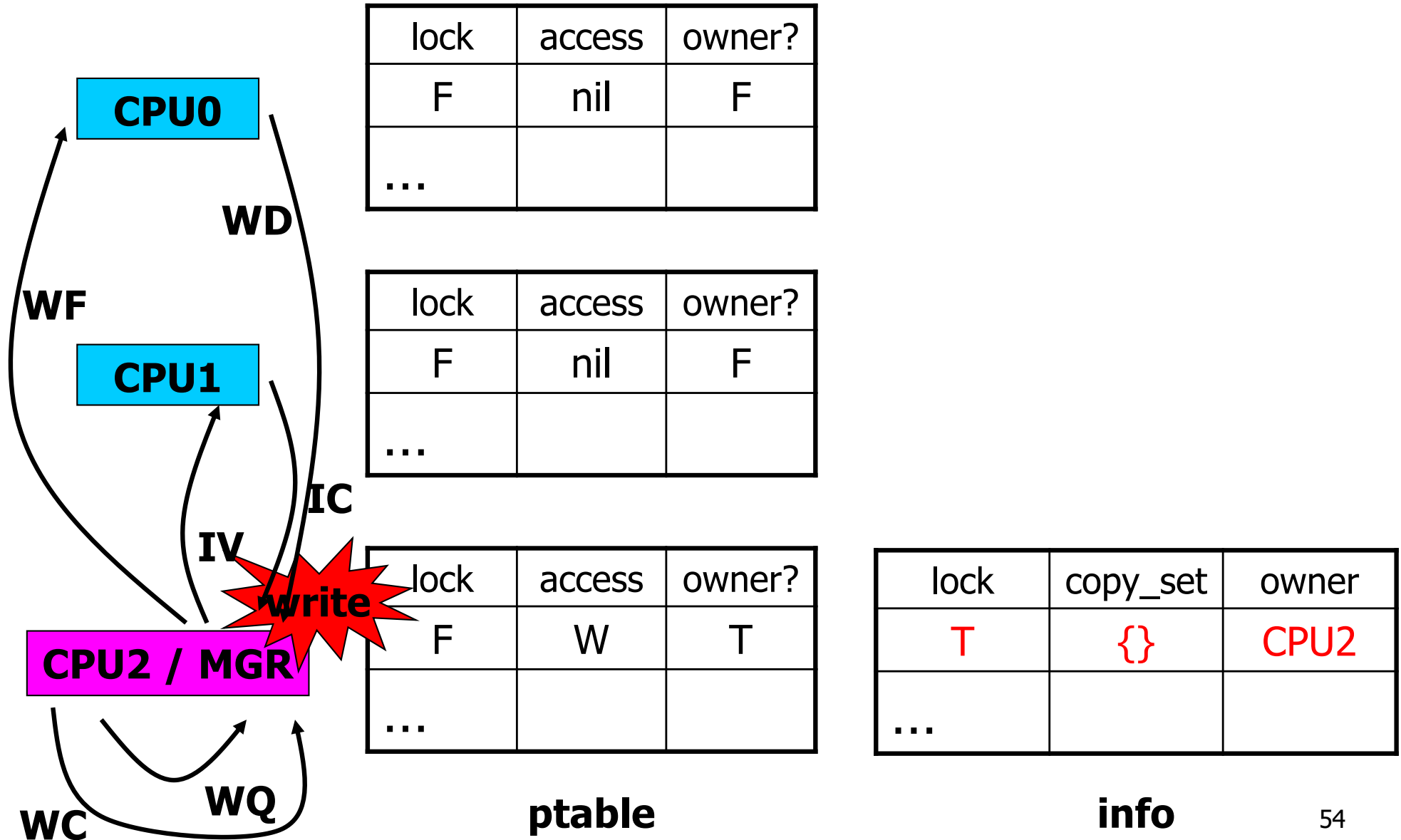
Centralized Manager Example 2: Owned by CPU0, CPU2 wants to write



Centralized Manager Example 2: Owned by CPU0, CPU2 wants to write



Centralized Manager Example 2: Owned by CPU0, CPU2 wants to write



What if Two CPUs Want to Write to Same Page at Same Time?

- Write has several steps, modifies multiple tables
- Invariants for tables:
 - MGR must agree with CPUs about single owner
 - MGR must agree with CPUs about copy_set
 - copy_set $\neq \{\}$ must agree with read-only for owner
- Write operation should thus be atomic!
- **What enforces atomicity?**

Sequential Consistency: Definition

- Must exist total order of operations such that:
 - All CPUs see results consistent with that total order (i.e., LDs see most recent ST in total order)
 - Each CPU's instructions appear in order in total order
- Two rules sufficient to implement sequential consistency [Lamport, 1979]:
 - Each CPU must execute reads and writes in program order, one at a time
 - Each memory location must execute reads and writes in arrival order, one at a time

Ivy and Consistency Models

- Consider done{0,1,2} example:
 - v0 = fn0(); done0 = true
 - **In Ivy, can other CPU see done0 == true, but still see old v0?**
- **Does Ivy obey sequential consistency?**
 - **Yes!**
 - **Each CPU does R/W in program order**
 - **Each memory location does R/W in arrival order**

Ivy: Evaluation

- Experiments include performance of PDE, matrix multiplication, and “block odd-even based merge-split algorithm”
- How to measure performance?
 - Speedup: x-axis is number of CPUs used, y-axis is how many times faster the program ran with that many CPUs
- **What’s the best speedup you should ever expect?**
 - **Linear**

Ivy: Evaluation

- Experiments include performance of PDE, matrix multiplication, and “block odd-even based merge-split algorithm”
- How to measure performance?
 - Speedup: x-axis is number of CPUs used, y-axis is how many times faster the program ran with that many CPUs
- **What’s the best speedup you should ever expect?**

When do you expect speedup to be linear?

What's "Block Odd-Even Based Merge-Split Algorithm?"

- Partition data to be sorted over N CPUs, held in one shared array
- Sort data in each CPU locally
- View CPUs as in a line, number 0 to $N-1$
- Repeat N times:
 - Even CPUs send to (higher) odd CPUs
 - Odd CPUs merge, send lower half back to even CPUs
 - Odd CPUs send to (higher) even CPUs
 - Even CPUs merge, send lower half back to odd CPUs
- "Send" just means "receiver reads from right place in shared memory"

Ivy's Speedup

- PDE and matrix multiplication: **linear**
- Sorting: **worse than linear, flattens significantly beyond 2 CPUs**

Ivy vs. RPC

- When would you prefer DSM to RPC?
 - More transparent
 - Easier to program for
- When would you prefer RPC to DSM?
 - Isolation
 - Control over communication
 - Latency-tolerance
 - Portability
- Could Ivy benefit from RPC?
 - Possibly for efficient blocking/unblocking

DSM: Successful Idea?

- Spreading a computation across workstations?
 - Yes! Google, Inktomi, Beowulf, ...
- Coherent access to shared memory?
 - Yes! Multi-CPU PCs use Ivy-like protocols for cache coherence between CPUs
- DSM as model for programming workstation cluster?
 - Little evidence of broad adoption
 - Too little control over communication, and communication dictates performance