Chapter 12: Multiprocessor Systems

Operating Systems

UMA: bus, crossbar, omega

NUMA and directory-based coherence

MESI and false sharing

Multicore, SMT, GPUs

Multiprocessor OS types

Synchronization: TSL, ticket, MCS

Scheduling: time, space, gang

Multicomputer topologies

Message passing and RPC

DSM and load balancing

Virtualization and containers

Distributed systems and middleware

Cloud and clusters

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides: https://sercankulcu.github.io/files/os/slides/Chapter_12_Multiprocessor_Systems.html

Learning Outcomes

Hardware

Compare UMA (bus, crossbar, omega) and NUMA architectures; trace directory-based coherence.

Cache coherence

Apply MESI state transitions to a sequence of reads and writes; find and fix false sharing.

OS and locks

Evaluate separate-OS, master-slave and SMP models; TSL, ticket and MCS locks; spinning versus switching.

Scheduling

Analyze a shared ready queue, space sharing and gang scheduling with an example.

Multicomputers

Compute topology diameters; explain message passing, RPC, DSM and load-balancing algorithms.

Virtualization and distributed systems

Recognize hypervisor types, VT-x/EPT and containers; middleware types and the cloud.

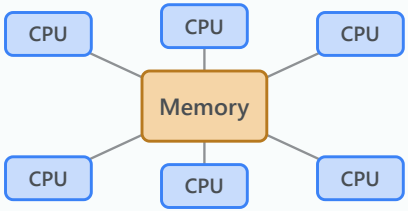
Why Multiple Processors?

Symmetric (SMP)
All processors have equal access to memory and I/O; kernel code can run on any of them.

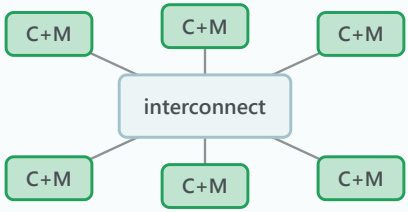
Asymmetric (AMP)
Processors are not equal: one runs the OS, or each runs separate software.

Cluster
Independent computers on a network act as one system; each node has its own OS.

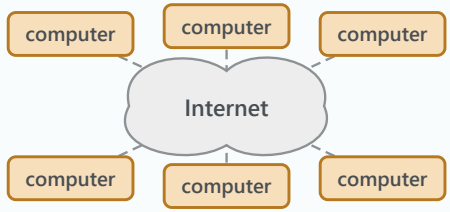
- Clock rates have been stuck at 3–5 GHz since ≈ 2005 (the end of Dennard scaling); the transistor budget now goes to **more cores**.
- The OS handles processor allocation, synchronization, interprocess communication and the **consistency of shared data**.



(a) shared memory



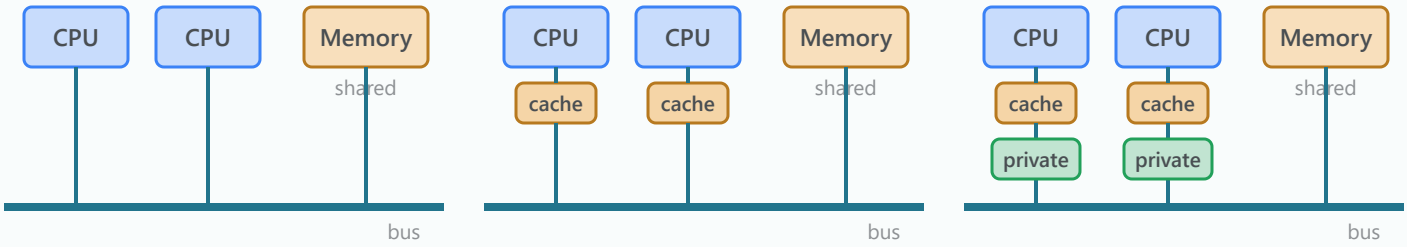
(b) message-passing multicomputer



(c) wide-area distributed system

Amdahl: with parallel fraction p, the speedup on n processors is 1 / ((1 − p) + p/n). For p = 0.95: 5.93 on 8 processors, 15.42 on 64, with a limit of 20. “More processors, more performance” holds only when the serial part is small.

UMA: Bus-Based Architecture

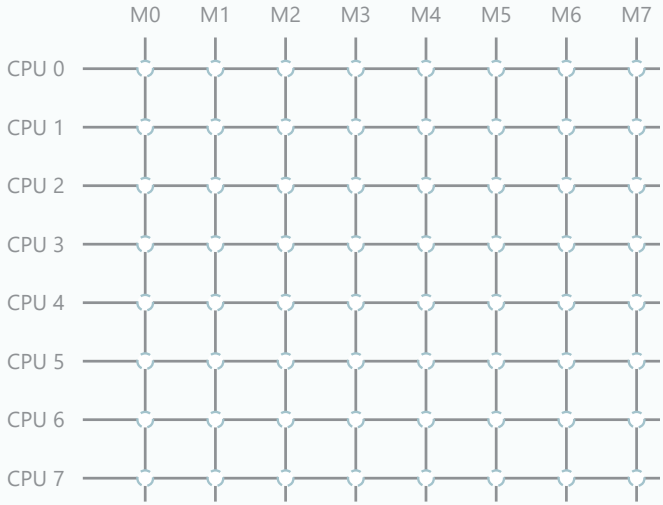


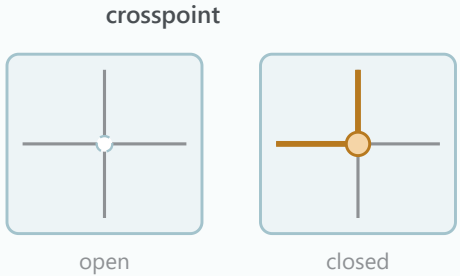
(a) no cache (b) a cache per CPU (c) cache + private memory

- **UMA** (*Uniform* Memory Access): every CPU reaches every memory word in **the same time**, which makes parallel programming easier.
- (a) Single bus: as CPUs are added the bus saturates; beyond 2–4 processors the CPUs spend most of their time waiting.
- (b) Caches serve most reads locally and cut traffic, but copies of the same line must be kept **coherent** (MESI).
- (c) Private memory holds program code, constants, stacks and locals; shared memory is used only for shared variables.

UMA: *Uniform* Memory Access. Today an on-chip ring or mesh replaces the single shared bus (Intel mesh, AMD Infinity Fabric).

Crossbar Switch





n × n switch → n^2 crosspoints

8 → 64 · 64 → 4.096

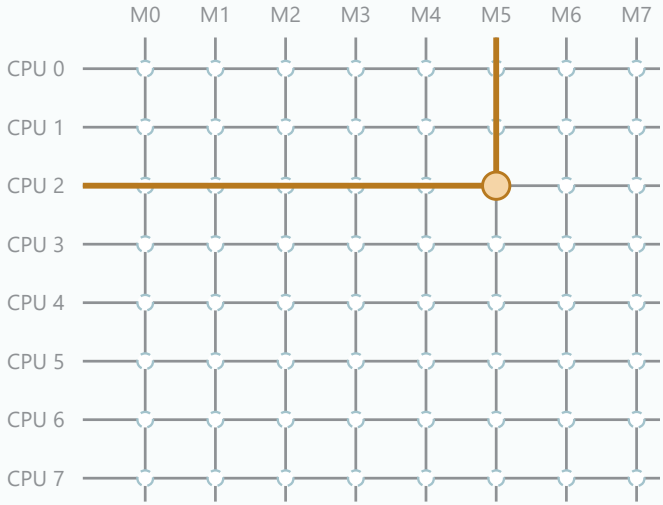
1,024 → 1,048,576

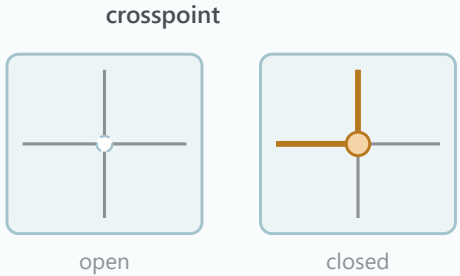
8 CPUs (rows) and 8 memory modules (columns); each crosspoint is a small electronic switch. All are **open**.

Step 1 / 4

Example: the old Sun Fire E25K used an 18 × 18 crossbar; today small crossbars between cores and L3 slices are common on chip.

Crossbar Switch





n × n switch → n^2 crosspoints

8 → 64 · 64 → 4.096

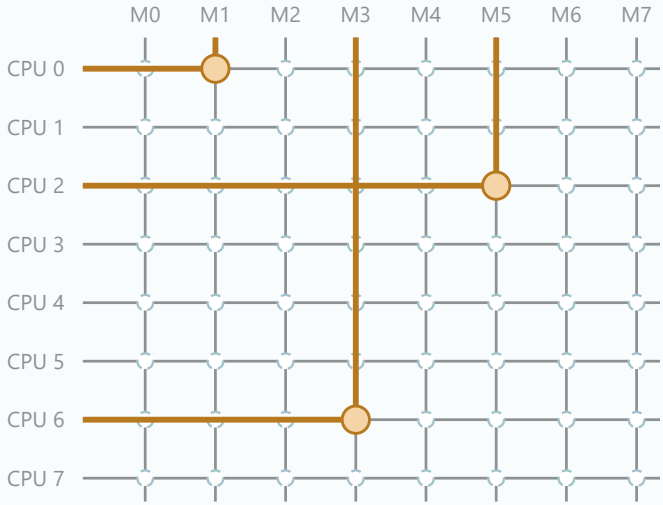
1,024 → 1,048,576

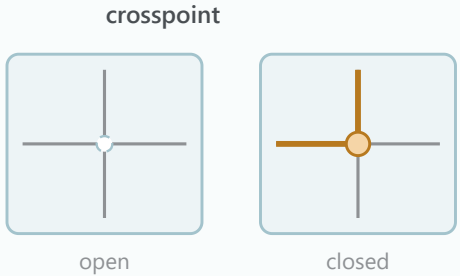
CPU 2 wants M5: crosspoint (2, 5) **closes**, creating a direct path.

Step 2 / 4

Example: the old Sun Fire E25K used an 18 × 18 crossbar; today small crossbars between cores and L3 slices are common on chip.

Crossbar Switch





n × n switch → n^2 crosspoints

8 → 64 · 64 → 4.096

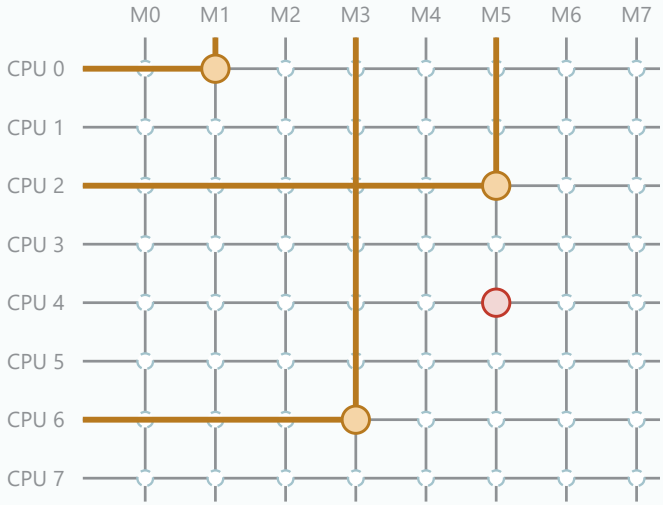
1,024 → 1,048,576

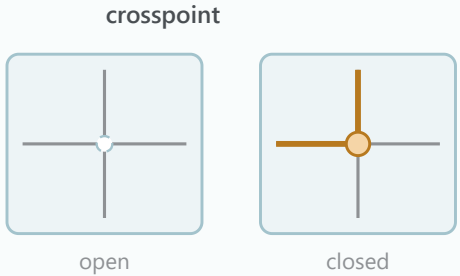
3 transfers at once

CPU 0 → M1 and CPU 6 → M3 are set up at the same time. A crossbar is **nonblocking**: requests to different modules never wait for each other.

Example: the old Sun Fire E25K used an 18 × 18 crossbar; today small crossbars between cores and L3 slices are common on chip.

Crossbar Switch





n × n switch → n^2 crosspoints

8 → 64 · 64 → 4.096

1,024 → 1,048,576

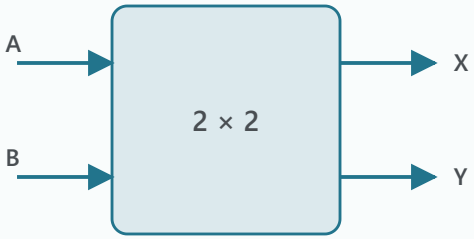
M5 busy → CPU 4 waits

CPU 4 also wants M5: the network has a path, but the **module is busy**; the only contention is at the memory module. The cost: n^2 crosspoints.

Step 4 / 4

Example: the old Sun Fire E25K used an 18 × 18 crossbar; today small crossbars between cores and L3 slices are common on chip.

Routing in an Omega Network



any input can connect to any output

bit = 0 → upper output (X)

bit = 1 → lower output (Y)

message format



Module: which memory module (routing tag)

Address: word within the module

Opcode: READ / WRITE

Value: word to write (optional)

n CPUs · n memories → $\log_2 n$ stages × n/2 switches

n = 8: 3 × 4 = 12 switches (a crossbar needs 64)

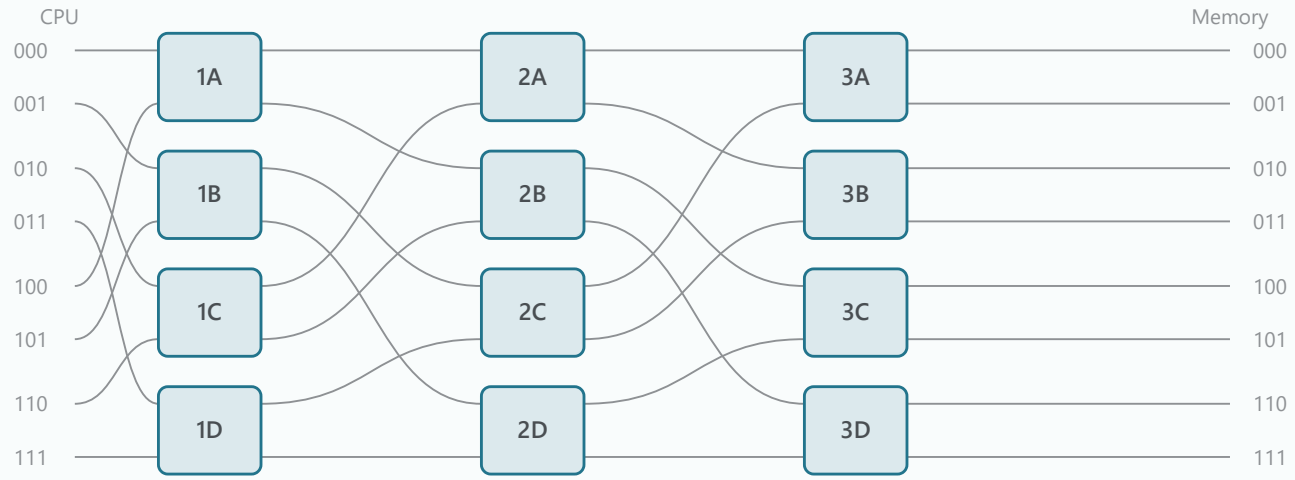
n = 1,024: 10 × 512 = 5,120 switches (1,048,576)

The building block is a **2 × 2 switch**: two inputs, two outputs. Switches look only at the message's **Module** field, the target memory module.

Step 1 / 5

An omega network carries many transfers *concurrently* (not “synchronously”), making conflicting ones wait; with $(n/2)\cdot\log_2 n$ switches it needs **far less** hardware and power than a crossbar.

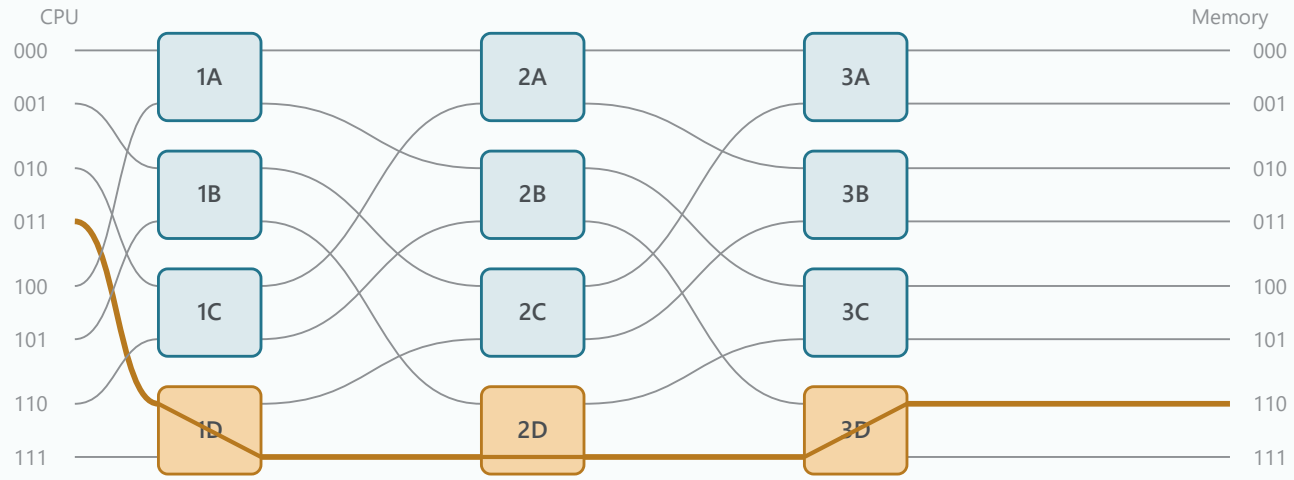
Routing in an Omega Network



8 CPUs, 8 memories, **3 stages × 4 switches**. The wiring between stages is a **perfect shuffle** (address bits rotate left by one).

An omega network carries many transfers *concurrently* (not "synchronously"), making conflicting ones wait; with $(n/2)\cdot\log_2 n$ switches it needs **far less** hardware and power than a crossbar.

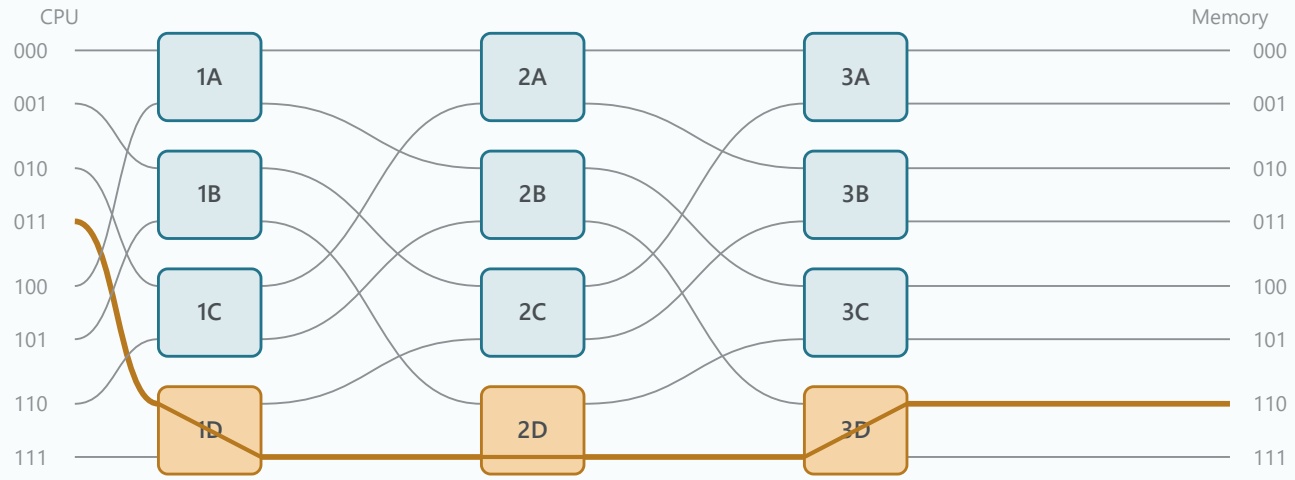
Routing in an Omega Network



CPU 011 reads memory **110**. Stage 1 (1D) checks the first target bit: **1** → lower output; 2D, second bit **1** → lower; 3D, third bit **0** → upper. Arrives at 110.

An omega network carries many transfers *concurrently* (not "synchronously"), making conflicting ones wait; with $(n/2)\cdot\log_2 n$ switches it needs **far less** hardware and power than a crossbar.

Routing in an Omega Network

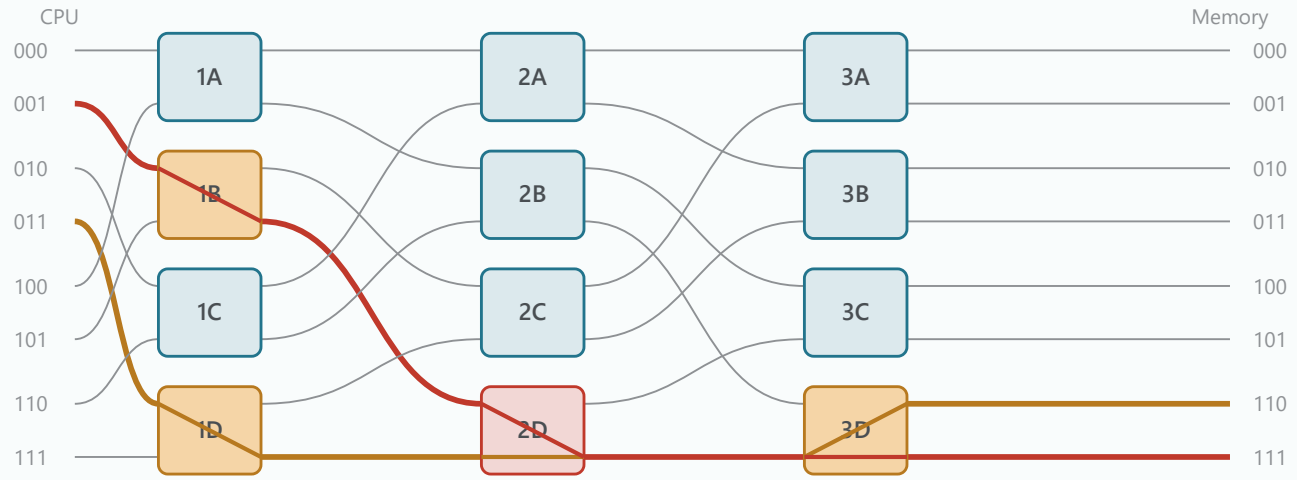


reply: the source (011) is written into Module and the path is retraced

The reply returns through the same network: each switch writes the source bit into the Module field, and memory uses that tag to reply to 011.

An omega network carries many transfers *concurrently* (not "synchronously"), making conflicting ones wait; with $(n/2)\cdot\log_2 n$ switches it needs **far less** hardware and power than a crossbar.

Routing in an Omega Network

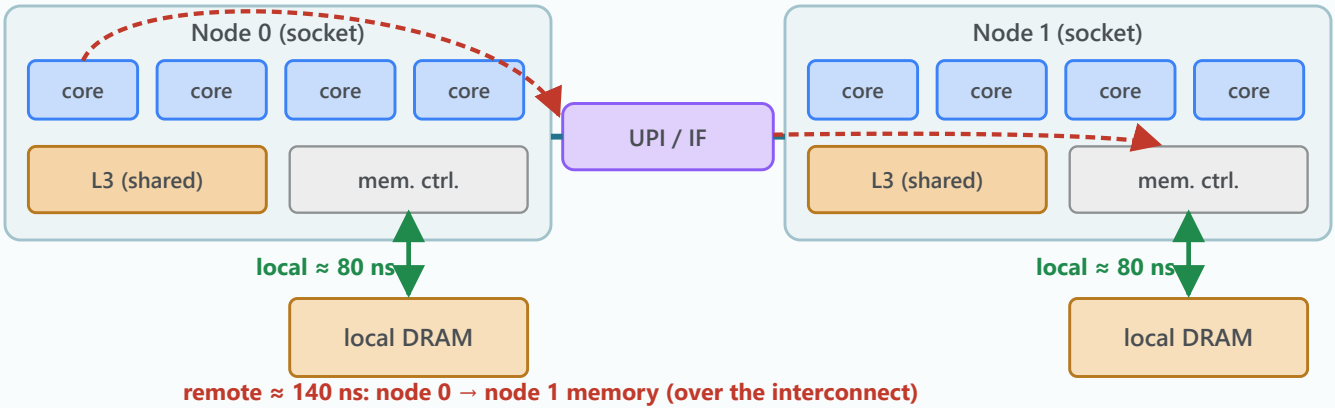


conflict: two requests want 2D's lower output

Meanwhile CPU 001 wants memory **111**; both requests need **2D**'s lower output → one waits. Omega is a **blocking** network.

An omega network carries many transfers *concurrently* (not "synchronously"), making conflicting ones wait; with $(n/2)\cdot\log_2 n$ switches it needs **far less** hardware and power than a crossbar.

NUMA Multiprocessors

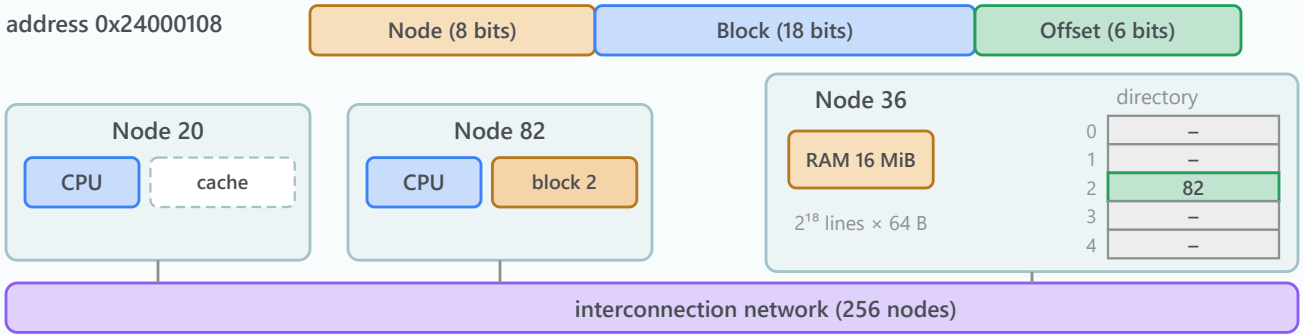


- **NUMA** (*Non-Uniform* Memory Access): every CPU sees **a single address space**; remote memory is also reached with ordinary LOAD/STORE.
- Remote access is slower than local. **NC-NUMA**: no cache coherence; **CC-NUMA**: directory-based coherence (every server today).
- The OS keeps a process and its pages **on the same node** (first-touch policy, Linux automatic NUMA balancing, `numactl`).

Average access = $y \cdot t_{local} + (1 - y) \cdot t_{remote}$
$y = 0.9 \rightarrow 0.9 \cdot 80 + 0.1 \cdot 140 =$ **86 ns**
$y = 0.5 \rightarrow$ **110 ns** (28% slower)
latencies are sample values

The term is *Non-Uniform* Memory Access. Even single-socket chips can be NUMA inside (NPS setting on AMD EPYC, Intel SNC).

Directory-Based CC-NUMA



32-bit address = 8-bit node + 18-bit block + 6-bit offset

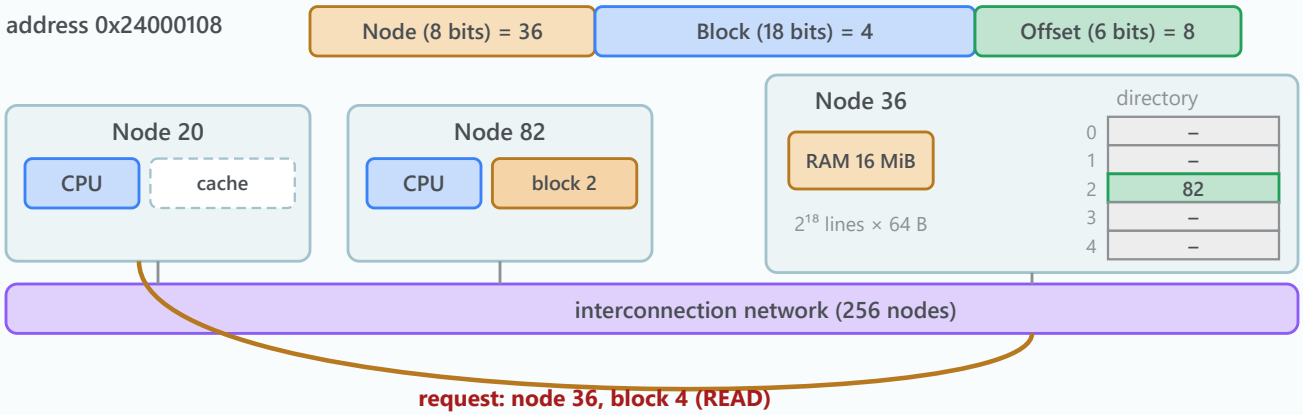
256 nodes × 16 MiB = 4 GiB · each node keeps the directory for its own blocks

Tanenbaum's example: 256 nodes, 16 MiB per node, 64-byte lines. Each node keeps the directory for the lines in **its own** memory.

Step 1 / 6

In this simple design a line can be cached at only one node. Real systems track the set of copies with a bit vector or a limited pointer list; on multicore chips a “snoop filter” does the same job.

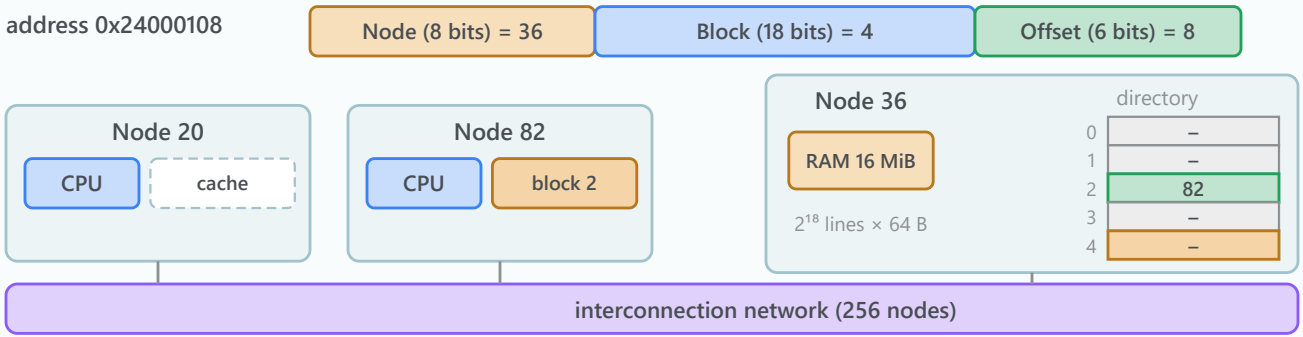
Directory-Based CC-NUMA



A CPU on node 20 executes `LOAD 0x24000108`: node = 0x24 = **36**, block = **4**, offset = **8**. Not local → the request goes to 36.

In this simple design a line can be cached at only one node. Real systems track the set of copies with a bit vector or a limited pointer list; on multicore chips a “snoop filter” does the same job.

Directory-Based CC-NUMA

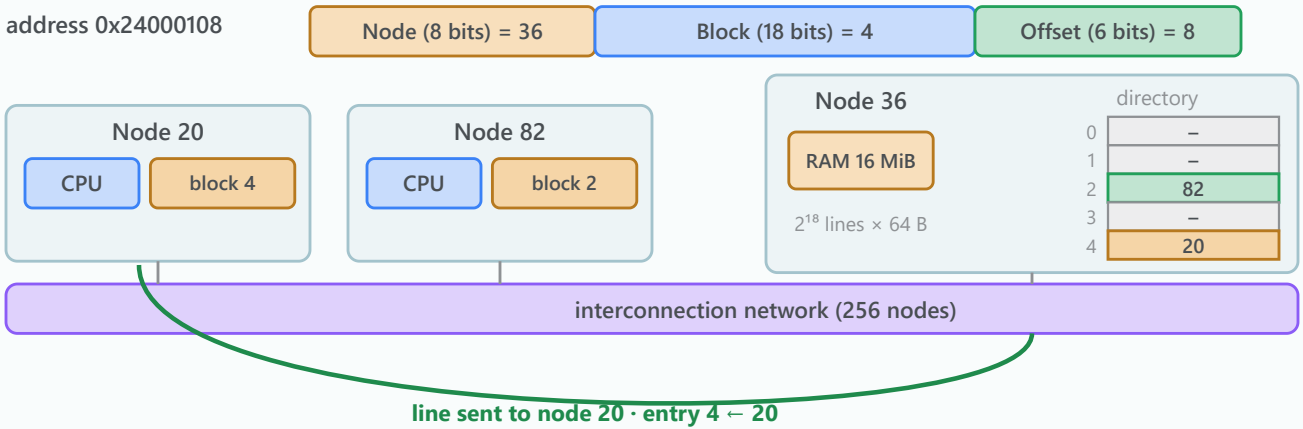


directory entry 4 empty → line not cached anywhere

Node 36 checks its directory: entry 4 is empty, so the line is in no cache. The current copy is in its own RAM.

In this simple design a line can be cached at only one node. Real systems track the set of copies with a bit vector or a limited pointer list; on multicore chips a “snoop filter” does the same job.

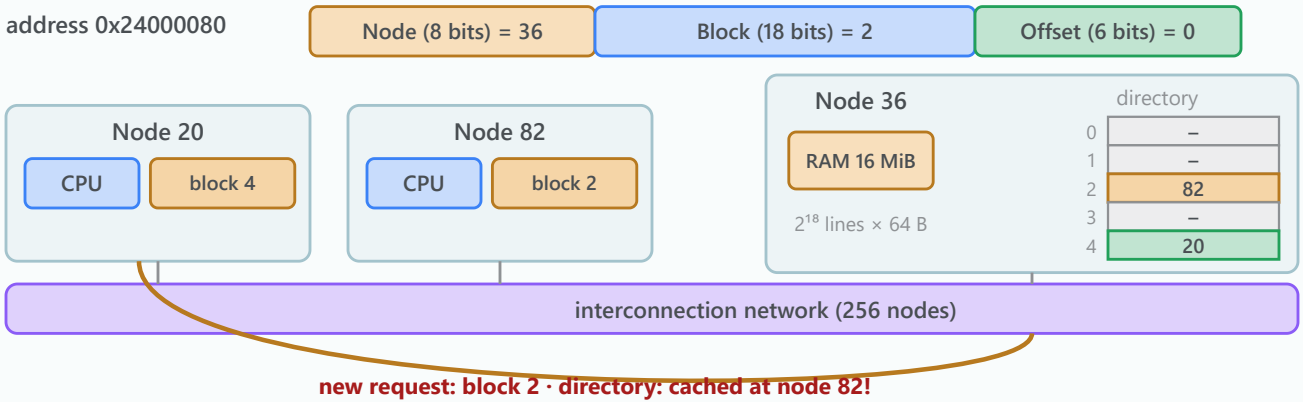
Directory-Based CC-NUMA



The line goes to node 20 and directory entry 4 is set to **20**: “a copy of this line is at 20”.

In this simple design a line can be cached at only one node. Real systems track the set of copies with a bit vector or a limited pointer list; on multicore chips a “snoop filter” does the same job.

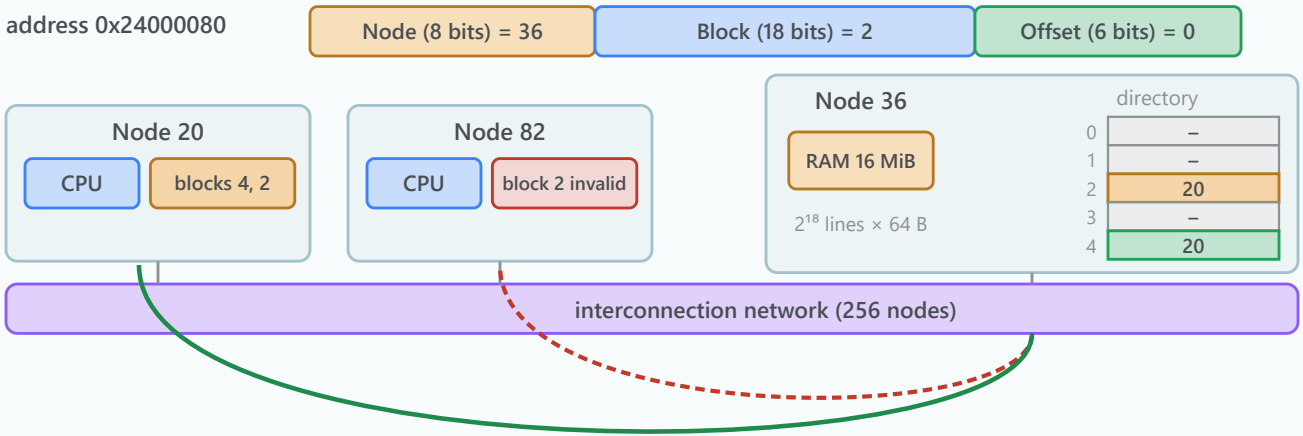
Directory-Based CC-NUMA



Now node 20 wants block **2** (0x24000080). The directory says the line is cached at **node 82**.

In this simple design a line can be cached at only one node. Real systems track the set of copies with a bit vector or a limited pointer list; on multicore chips a “snoop filter” does the same job.

Directory-Based CC-NUMA

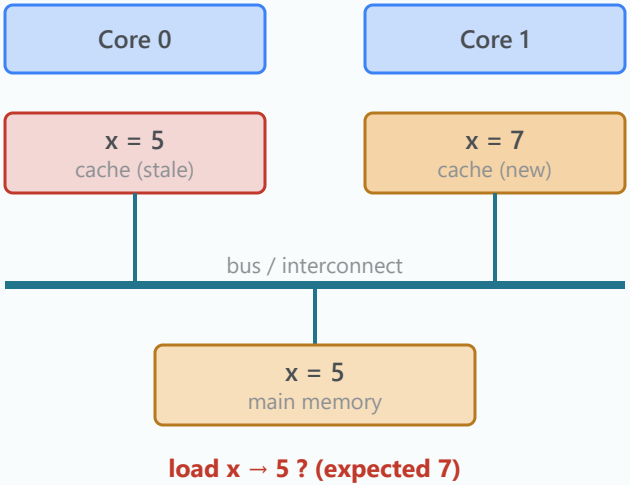


invalidate at 82 → then send the line to 20 · entry 2 ← 20

Node 36 sends 82 an “invalidate” (with write-back if needed), then sends the line to 20; entry 2 ← 20. **Point-to-point** messages replace bus snooping.

In this simple design a line can be cached at only one node. Real systems track the set of copies with a bit vector or a limited pointer list; on multicore chips a “snoop filter” does the same job.

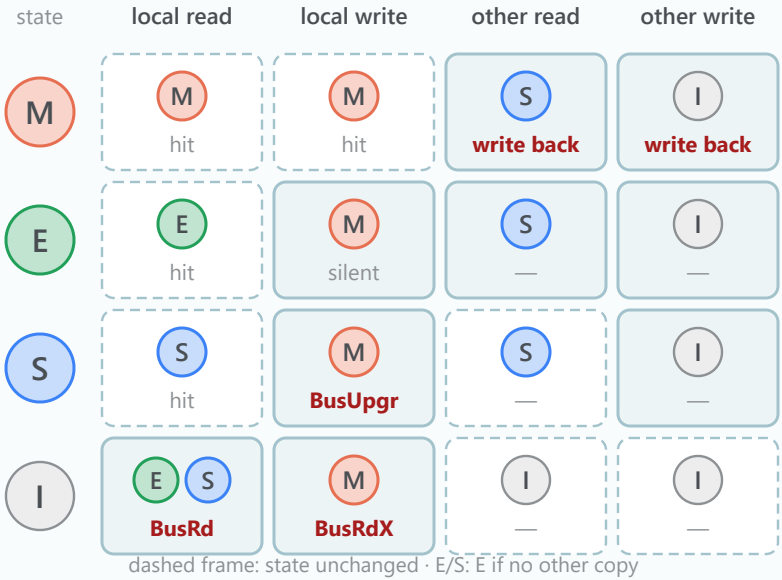
The Cache Coherence Problem



- Each core's private cache holds **copies** of the same memory. When one writes, the other's copy goes stale.
- **Coherence**: all cores see writes to the same address in the same order, and a read returns the latest written value.
- **Snooping**: every cache controller listens on a shared bus; fine for a few cores.
- **Directory**: a list of copies is kept and messages go only to the nodes involved; scales to many cores and NUMA.
- Two basic policies: **write-invalidate** (common) and **write-update**.

Coherence concerns a single address; the order of different addresses is the business of the *memory consistency model* (x86-TSO, ARM's weak model), which is why locks include memory barriers.

The MESI Protocol



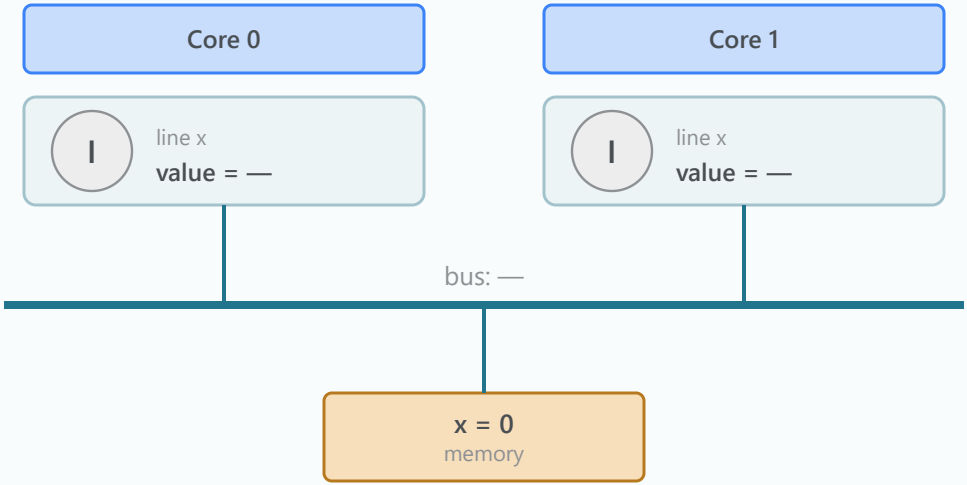
State	Meaning
M Modified	Only copy, **dirty**; memory is stale. Written back on eviction.
E Exclusive	Only copy, clean. A write turns it into M without touching the bus.
S Shared	There may be several clean copies. To write, invalidate the others first.
I Invalid	The copy is invalid; an access is a cache miss.

Intel MESIF (F: the copy that responds) and AMD MOESI (O: dirty but shared) are variants of this protocol.

MESI Step by Step

operation sequence

1. C0: read x
2. C1: read x
3. C1: x = 1
4. C0: read x
5. C0: x = 2
6. C0: x = 3

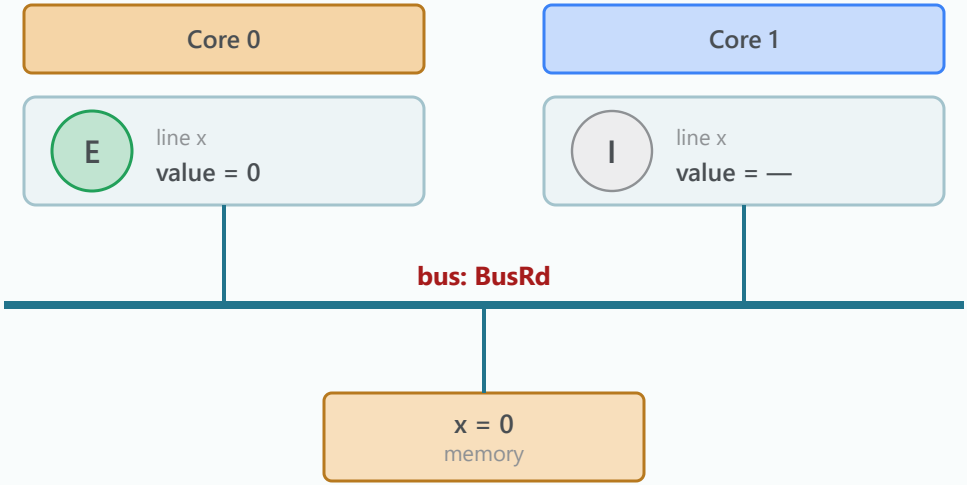


Start: x is only in memory (0). The line is **I** in both caches.

MESI Step by Step

operation sequence

1. C0: read x
2. C1: read x
3. C1: x = 1
4. C0: read x
5. C0: x = 2
6. C0: x = 3

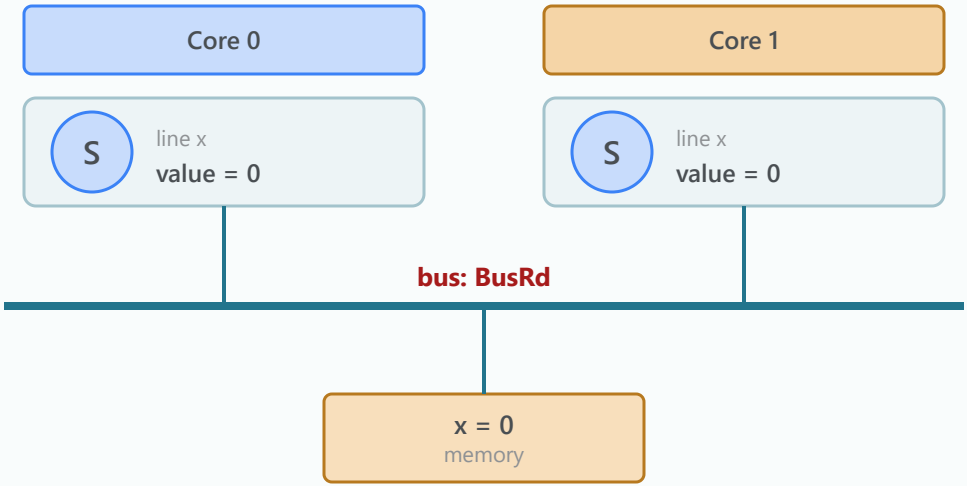


C0 reads: BusRd, no other copy → **E** (only and clean).

MESI Step by Step

operation sequence

1. C0: read x
2. C1: read x
3. C1: x = 1
4. C0: read x
5. C0: x = 2
6. C0: x = 3

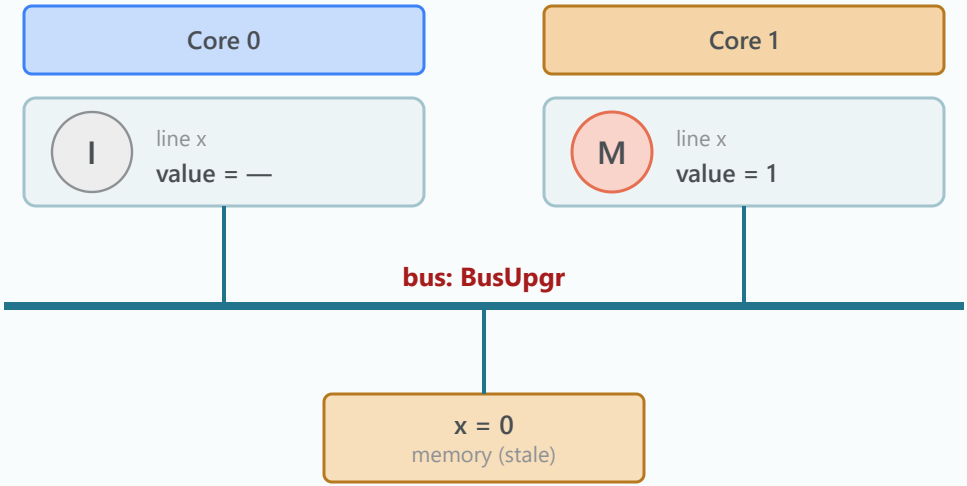


C1 reads: C0 snoops the request and says “I have it” → both **S**. Memory is still current.

MESI Step by Step

operation sequence

1. C0: read x
2. C1: read x
3. C1: x = 1
4. C0: read x
5. C0: x = 2
6. C0: x = 3

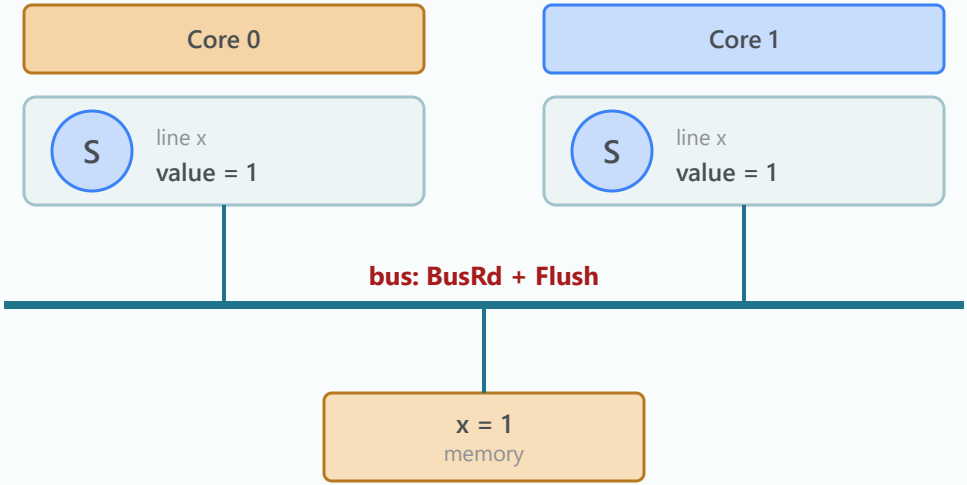


C1 writes: to write in state S it broadcasts **BusUpgr** (invalidate) → C0 **I**, C1 **M**; memory is now stale.

MESI Step by Step

operation sequence

1. C0: read x
2. C1: read x
3. C1: x = 1
4. C0: read x
5. C0: x = 2
6. C0: x = 3

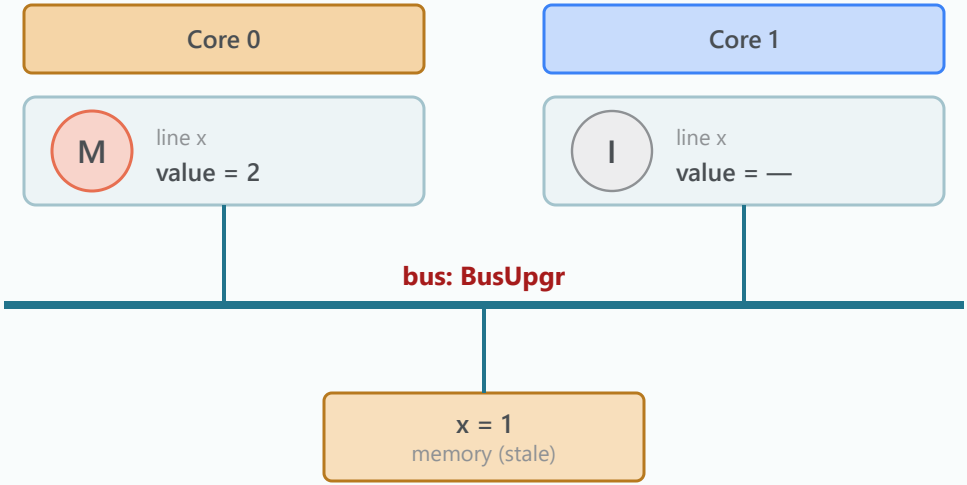


C0 reads: a miss. C1 **flushes** the dirty line (data to C0 and memory); both **S**, memory = 1.

MESI Step by Step

operation sequence

1. C0: read x
2. C1: read x
3. C1: x = 1
4. C0: read x
5. C0: x = 2
6. C0: x = 3

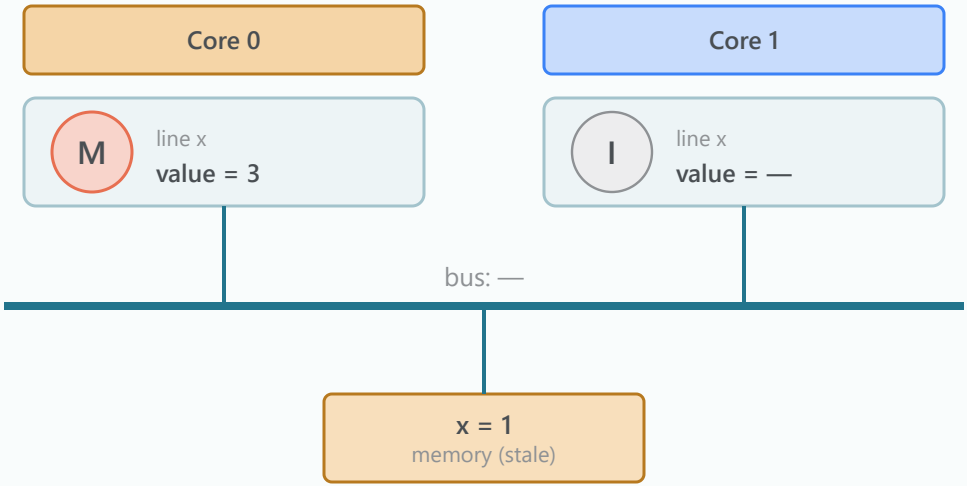


C0 writes: another invalidation → C0 **M**, C1 **I**.

MESI Step by Step

operation sequence

1. C0: read x
2. C1: read x
3. C1: x = 1
4. C0: read x
5. C0: x = 2
6. C0: x = 3



C0 writes again: already M, so **no bus traffic**. That is why write-heavy private data is cheap.

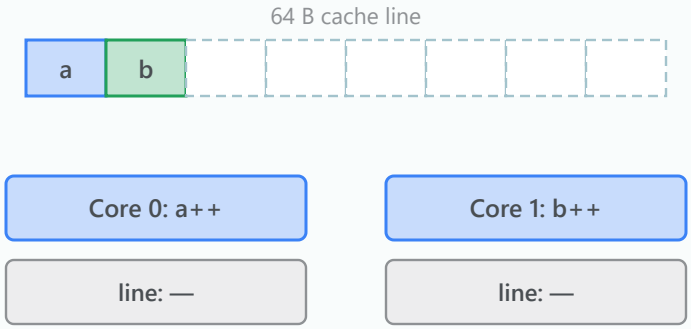
False Sharing

```
struct counters {       /* bad */
  long a;   /* offset 0 */
  long b;   /* offset 8 */
};

struct padded {         /* good */
  _Alignas(64) long a;
  _Alignas(64) long b;
};

/* thread 0: for(..) s.a++;
   thread 1: for(..) s.b++; */
```

- The unit of coherence is the **line** (64 B), not the variable.
- Java: `@Contended`; Linux: `____cacheline_aligned`.
- DSM has the same problem at **page** granularity: if two independent variables share a page, the page bounces between machines.



no sharing in the logic, sharing in hardware

`a` and `b` are independent counters but sit **in the same line**.

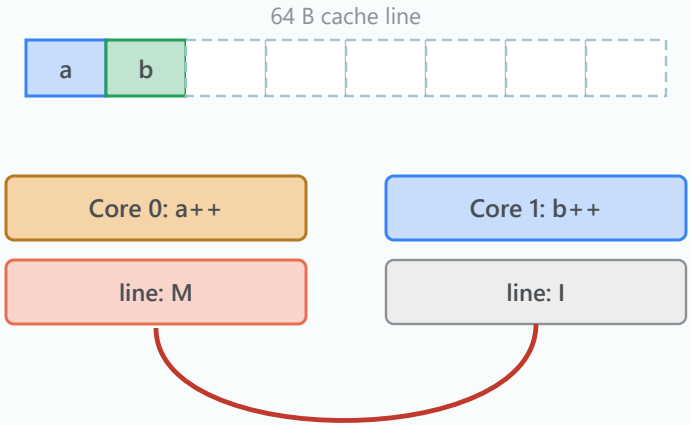
False Sharing

```
struct counters {       /* bad */
  long a;   /* offset 0 */
  long b;   /* offset 8 */
};

struct padded {         /* good */
  _Alignas(64) long a;
  _Alignas(64) long b;
};

/* thread 0: for(..) s.a++;
   thread 1: for(..) s.b++; */
```

- The unit of coherence is the **line** (64 B), not the variable.
- Java: `@Contended`; Linux: `____cacheline_aligned`.
- DSM has the same problem at **page** granularity: if two independent variables share a page, the page bounces between machines.



the line moves across: ~50–150 ns

C0 `a++`: takes the line in M; C1's copy is invalidated.

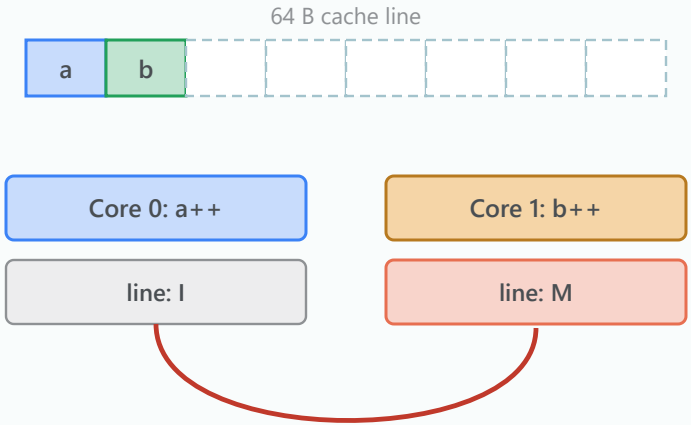
False Sharing

```
struct counters {       /* bad */
  long a;   /* offset 0 */
  long b;   /* offset 8 */
};

struct padded {         /* good */
  _Alignas(64) long a;
  _Alignas(64) long b;
};

/* thread 0: for(..) s.a++;
   thread 1: for(..) s.b++; */
```

- The unit of coherence is the **line** (64 B), not the variable.
- Java: `@Contended`; Linux: `____cacheline_aligned`.
- DSM has the same problem at **page** granularity: if two independent variables share a page, the page bounces between machines.



the line moves across: ~50–150 ns

C1 `b++`: the line moves to C1 (M); C0's copy becomes invalid.

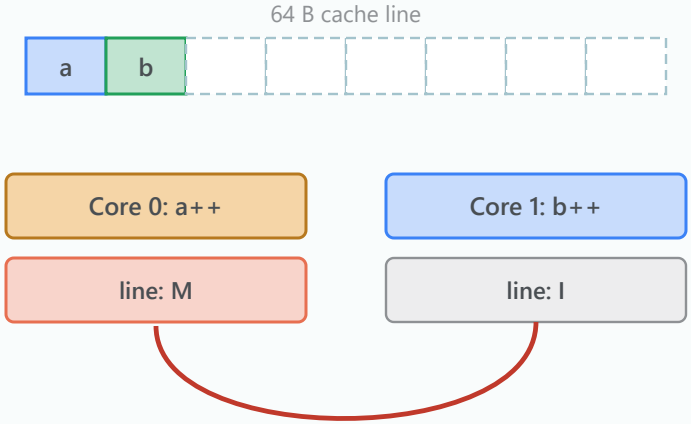
False Sharing

```
struct counters {       /* bad */
  long a;   /* offset 0 */
  long b;   /* offset 8 */
};

struct padded {         /* good */
  _Alignas(64) long a;
  _Alignas(64) long b;
};

/* thread 0: for(..) s.a++;
   thread 1: for(..) s.b++; */
```

- The unit of coherence is the **line** (64 B), not the variable.
- Java: `@Contended`; Linux: `____cacheline_aligned`.
- DSM has the same problem at **page** granularity: if two independent variables share a page, the page bounces between machines.



the line moves across: ~50–150 ns

When two cores write in quick alternation, the line moves every time: “ping-pong”. Two threads can run slower than one.

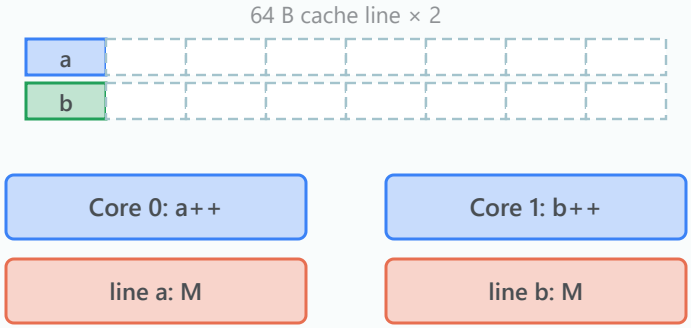
False Sharing

```
struct counters {       /* bad */
  long a;   /* offset 0 */
  long b;   /* offset 8 */
};

struct padded {         /* good */
  _Alignas(64) long a;
  _Alignas(64) long b;
};

/* thread 0: for(..) s.a++;
   thread 1: for(..) s.b++; */
```

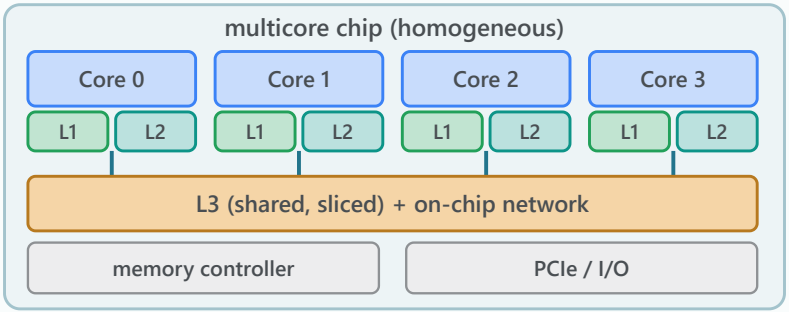
- The unit of coherence is the **line** (64 B), not the variable.
- Java: `@Contended`; Linux: `____cacheline_aligned`.
- DSM has the same problem at **page** granularity: if two independent variables share a page, the page bounces between machines.

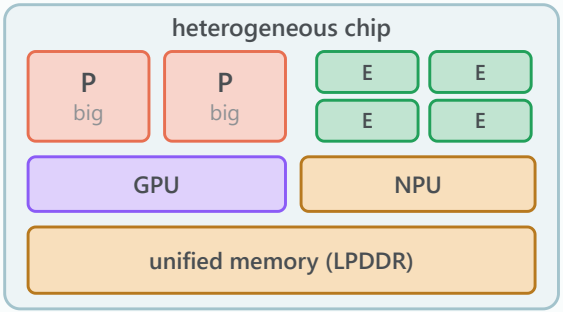


separate lines: both stay M
no coherence traffic

Fix: align or pad so each counter gets **its own line**. The memory cost is 56 B per counter; the gain is often a manyfold speedup.

Multicore and Heterogeneous Chips





Multicore (CMP)

Private L1/L2, shared L3. A server socket: 192 cores (AMD EPYC 9005), 288 E-cores (Xeon 6900E).

Chiplets

Groups of cores sit on separate dies; even within one socket, L3 and memory distances differ.

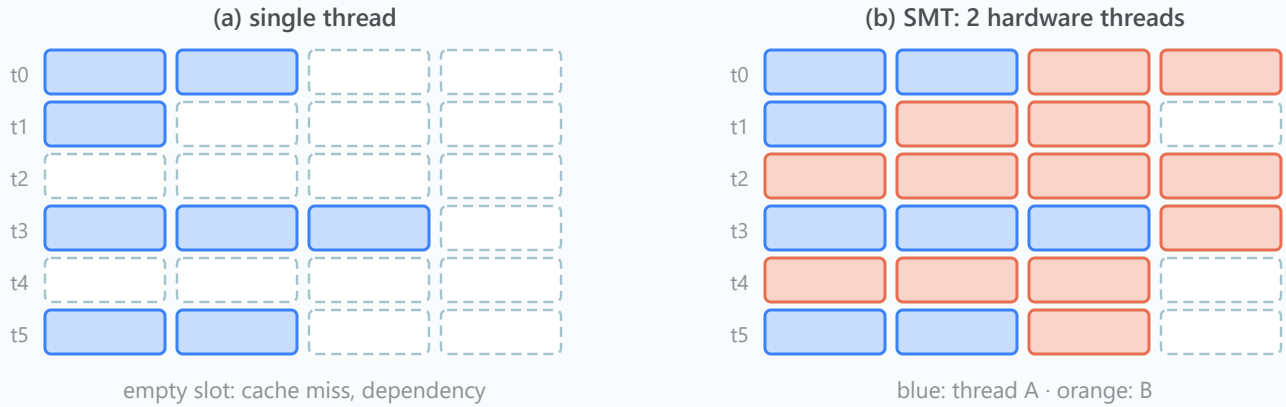
Big–little cores

big.LITTLE, Apple and Intel P/E: the OS places work by energy (Linux EAS, Thread Director).

Accelerators

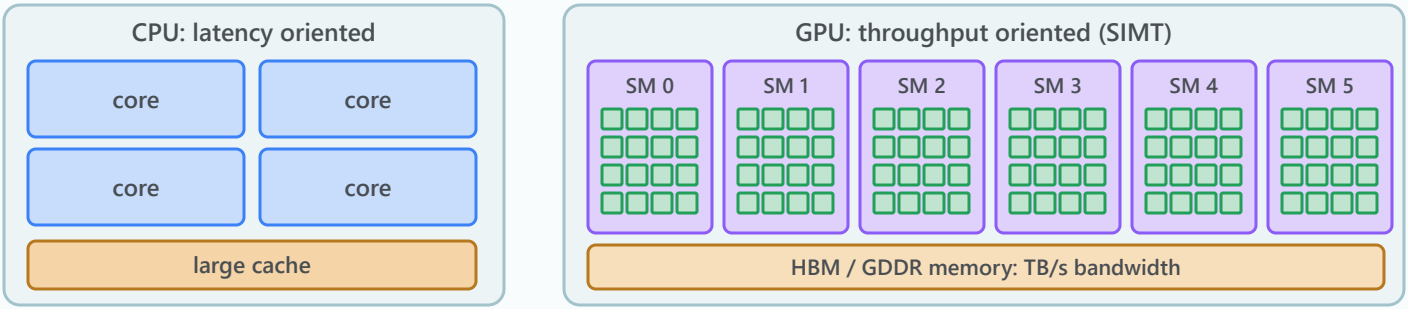
GPUs and NPUs on the same chip share the same memory; they are scheduled through the driver.

Simultaneous Multithreading (SMT)



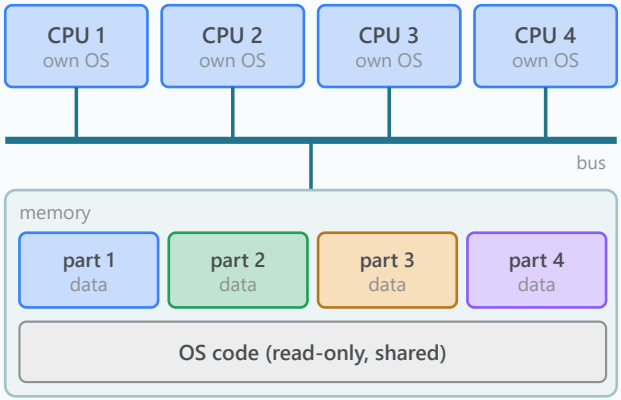
- One physical core appears as **two logical CPUs** (Intel Hyper-Threading, AMD SMT): two register sets, shared execution units and L1/L2.
- The gain is typically 10–30%, and zero when both threads want the same unit. The scheduler should fill **idle physical cores** first.
- Security: a shared cache opens side channels (L1TF, MDS) → Linux *core scheduling* places only mutually trusted threads on one core. Intel Lunar Lake (2024) dropped SMT on its P-cores.

GPUs and Heterogeneous Computing



- A GPU runs thousands of simple lanes in **groups of 32** (warps/wavefronts) on the same instruction: data-parallel work (graphics, AI, scientific computing).
- Programming: CUDA, ROCm/HIP, OpenCL, SYCL; the CPU launches a kernel, moves data and waits for the result.
- The OS and driver manage GPU memory, time-share contexts, and provide unified/shared virtual memory (IOMMU, page faults).
- Clusters put 4–8 GPUs per node linked by NVLink; nodes talk via RDMA over InfiniBand/RoCE.

Each CPU Has Its Own OS



- Memory is split into one partition per CPU; each CPU runs **an independent copy of the OS** in its partition. The OS code can be shared; the data is separate.
- A system call is caught and handled on the local CPU.

Pros

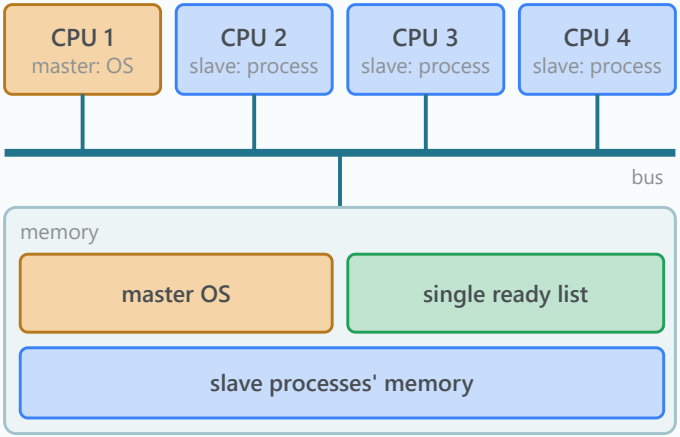
+ Simple: a uniprocessor OS runs almost unchanged.
+ No locks between partitions.

Cons

– Processes are not shared: one CPU may be overloaded while another idles.
– Pages are not shared; each OS keeps its own buffer cache → inconsistent disk blocks.

Today's equivalent: partitioned systems and separate VMs on a hypervisor; in research, the Barrelfish “multikernel” treats each core as a separate node that communicates by messages.

The Master-Slave Model



- The only copy of the OS and its tables lives **on the master CPU only**; every system call is redirected to the master.
- Slaves run user processes handed out by the master; with a single ready list, the load stays balanced.

Pros

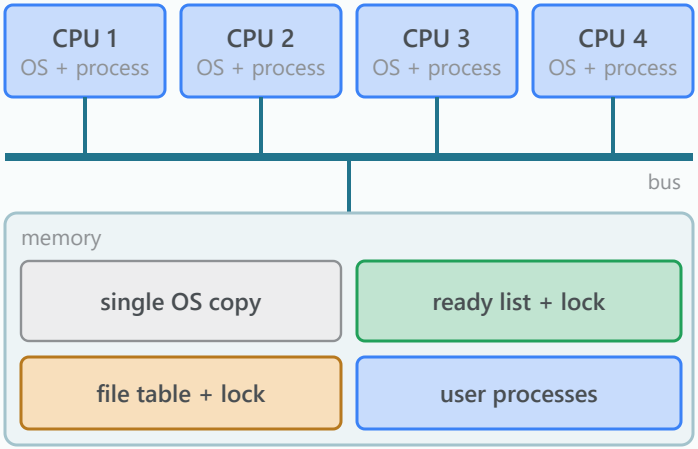
+ Simple and consistent: only one CPU touches kernel data.
+ Load balancing comes for free.

Cons

– The master becomes a **bottleneck**: with system-call-heavy work it does not scale past a few CPUs.
– If the master crashes, the whole system stops.

This is the classic form of asymmetric multiprocessing (AMP). Asymmetry lives on in embedded chips today: a Cortex-M core running an RTOS beside Cortex-A cores running Linux (OpenAMP, remoteproc).

Symmetric Multiprocessors (SMP)



- There is **a single copy of the OS** in memory; any CPU can run kernel code. A system call is handled on the CPU that made it.
- Problem: two CPUs may touch the same kernel table at once → **locks**.
- The first step was one big kernel lock (Linux BKL, 2.0), removed completely in 2.6.39 (2011). Today: thousands of fine-grained locks, RCU and per-CPU data.

Pros

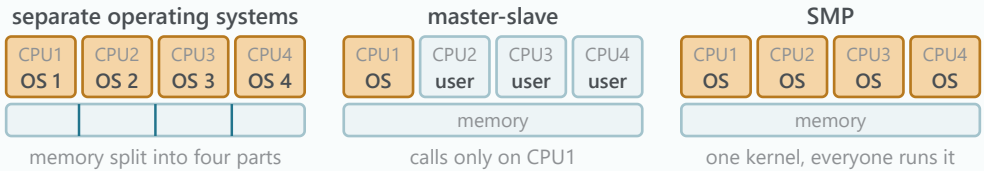
+ No bottleneck; every CPU runs the kernel.
+ Memory and processes are balanced dynamically.

Cons

– Kernel tables must be split into critical sections.
– A lock-ordering bug → deadlock.

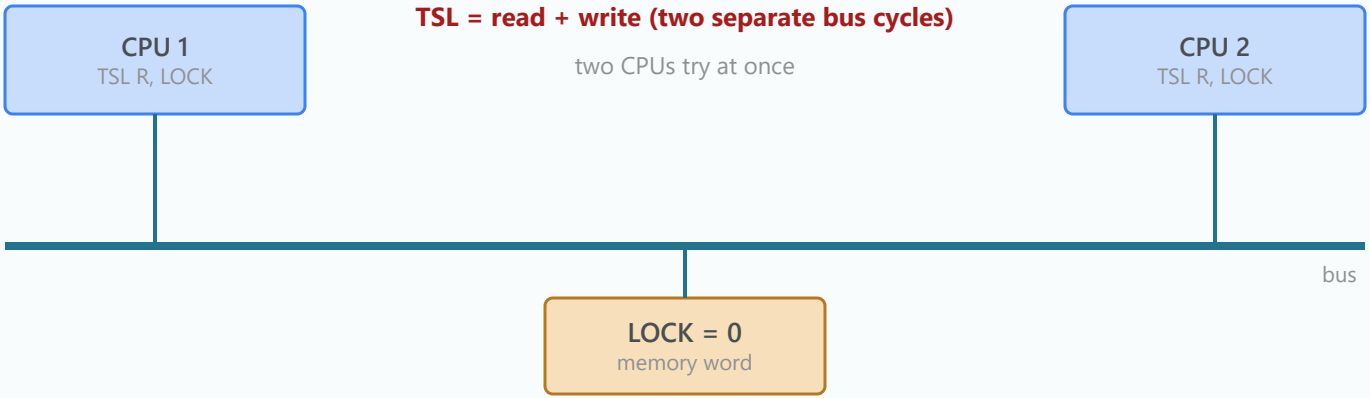
Comparing the Three Models

	Separate OS	Master-slave	SMP
OS copies	one per CPU	one (on the master)	one (everyone runs it)
Where are system calls handled?	local CPU	master CPU	local CPU
Ready list	per CPU	single, on the master	single, or per CPU + balancing
Load balancing	none	good	good
Scalability	good, but no sharing	master is a bottleneck	depends on lock design; the best
Complexity	lowest	low	highest
Today	VMs, partitioning	embedded AMP, legacy systems	Linux, Windows, macOS, BSD



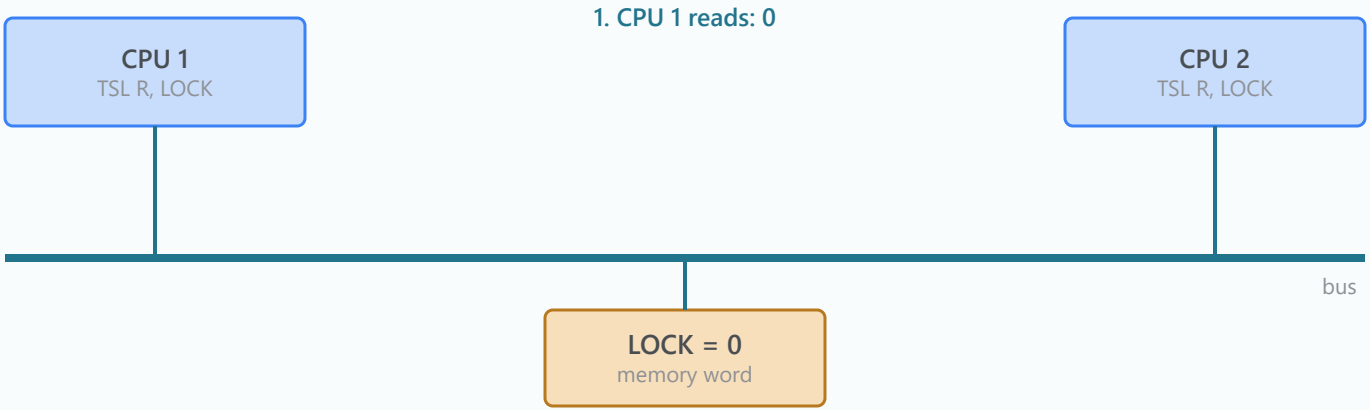
A multiprocessor OS differs from a multicomputer OS: in a multiprocessor all CPUs see **the same physical memory**; in a multicomputer each node has its own memory and communicates only by **messages**.

TSL and Bus Locking



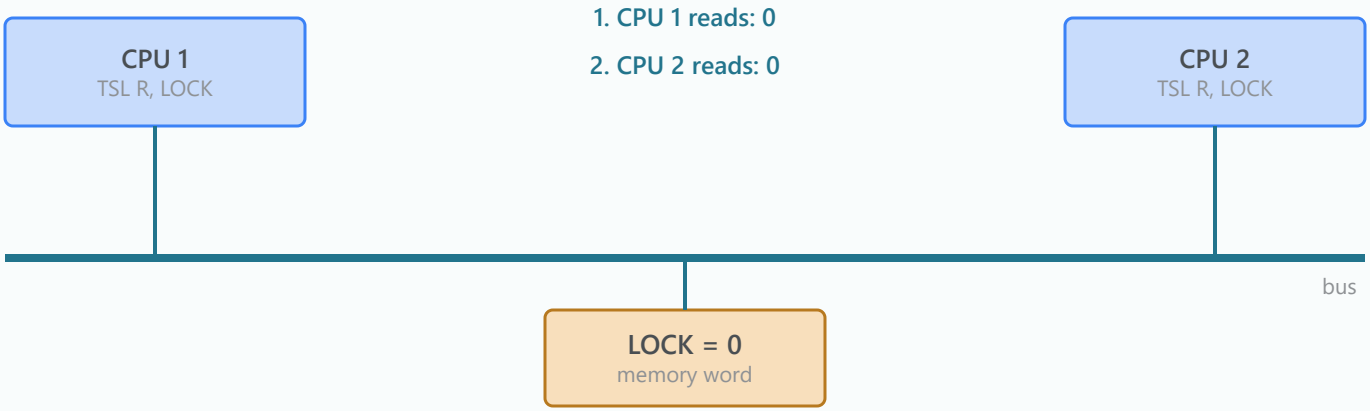
On a uniprocessor TSL (test and set lock) is an indivisible instruction. On a multiprocessor, **reading** and **writing** the memory word are separate bus cycles.

TSL and Bus Locking



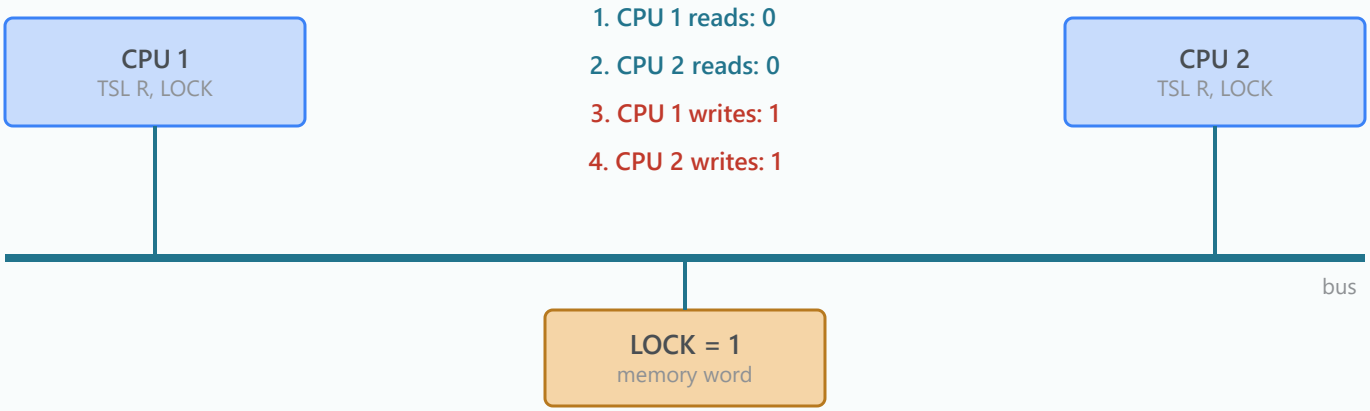
CPU 1 reads the lock: 0 (free).

TSL and Bus Locking



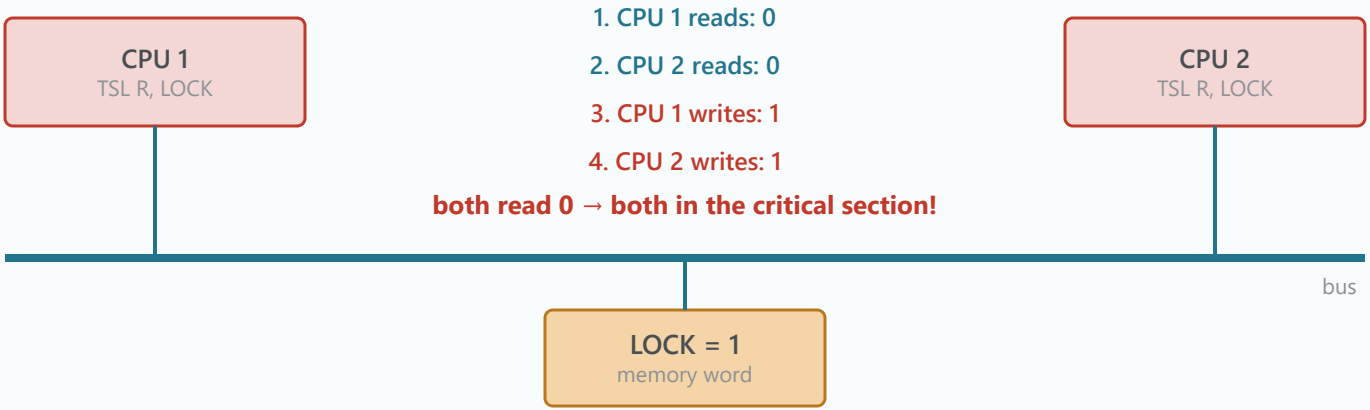
Before CPU 1 writes, CPU 2 reads too: still 0.

TSL and Bus Locking



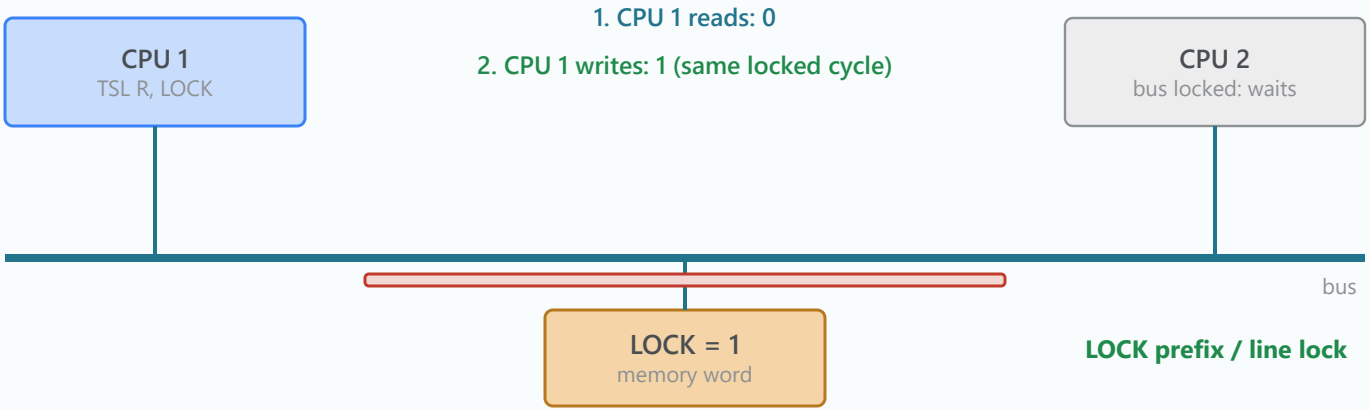
CPU 1 writes 1, and CPU 2 writes 1.

TSL and Bus Locking



Result: both think “the lock is mine” and mutual exclusion breaks. Disabling interrupts does not help either: it stops only **its own** CPU.

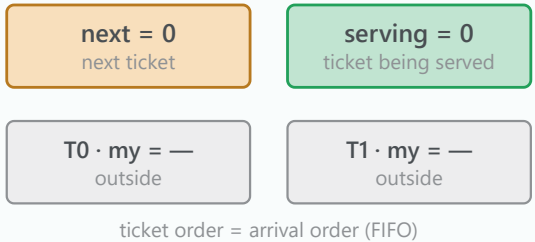
TSL and Bus Locking



Fix: TSL **locks the bus** for the whole read–write. Today the x86 `LOCK` prefix usually just holds the **cache line** in state M; ARM uses `LDXR/STXR` or LSE `CAS`.

Ticket Lock

```
1  import java.util.concurrent.atomic.*;
2  public class Ticket {
▶    static AtomicInteger next = new AtomicInteger();
▶    static volatile int serving = 0;
5    static int counter = 0;
6    static void lock() {
7      int my = next.getAndIncrement();
8      while (serving != my) Thread.onSpinWait();
9    }
10   static void unlock() { serving++; }
11   public static void main(String[] a)
12       throws Exception {
13     Runnable job = () -> {
14       for (int i = 0; i < 100_000; i++) {
15         lock(); counter++; unlock(); } };
16     Thread t0 = new Thread(job);
17     Thread t1 = new Thread(job);
18     t0.start(); t1.start(); t0.join(); t1.join();
19     System.out.println("counter = " + counter);
20   }
21 }
```



Two counters: `next` hands out tickets, `serving` shows whose turn it is.

Step 1 / 8

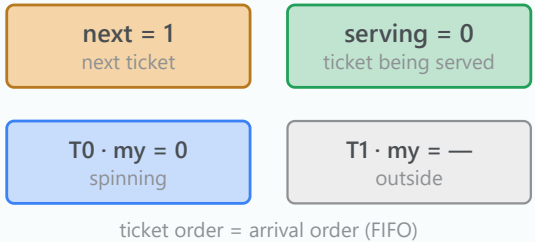
VARIABLES

`next = 0` `serving = 0` `T0.my = –` `T1.my = –` `counter = 0`

OUTPUT

Ticket Lock

```
1  import java.util.concurrent.atomic.*;
2  public class Ticket {
3    static AtomicInteger next = new AtomicInteger();
4    static volatile int serving = 0;
5    static int counter = 0;
6    static void lock() {
▶      int my = next.getAndIncrement();
8      while (serving != my) Thread.onSpinWait();
9    }
10   static void unlock() { serving++; }
11   public static void main(String[] a)
12       throws Exception {
13     Runnable job = () -> {
14       for (int i = 0; i < 100_000; i++) {
15         lock(); counter++; unlock(); } };
16     Thread t0 = new Thread(job);
17     Thread t1 = new Thread(job);
18     t0.start(); t1.start(); t0.join(); t1.join();
19     System.out.println("counter = " + counter);
20   }
21 }
```



T0 takes a ticket: `getAndIncrement` is atomic (x86 `LOCK XADD`) → my = 0.

Step 2 / 8

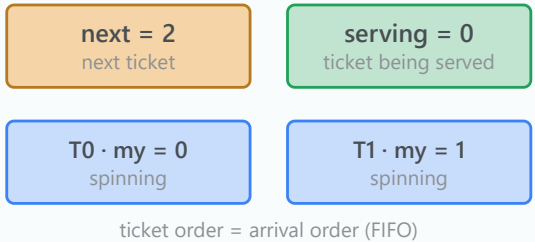
VARIABLES

`next = 1` `serving = 0` `T0.my = 0` `T1.my = –` `counter = 0`

OUTPUT

Ticket Lock

```
1  import java.util.concurrent.atomic.*;
2  public class Ticket {
3    static AtomicInteger next = new AtomicInteger();
4    static volatile int serving = 0;
5    static int counter = 0;
6    static void lock() {
▶      int my = next.getAndIncrement();
8      while (serving != my) Thread.onSpinWait();
9    }
10   static void unlock() { serving++; }
11   public static void main(String[] a)
12       throws Exception {
13     Runnable job = () -> {
14       for (int i = 0; i < 100_000; i++) {
15         lock(); counter++; unlock(); } };
16     Thread t0 = new Thread(job);
17     Thread t1 = new Thread(job);
18     t0.start(); t1.start(); t0.join(); t1.join();
19     System.out.println("counter = " + counter);
20   }
21 }
```



T1 takes a ticket too: my = 1. Even if two CPUs call at the same moment, the tickets differ.

Step 3 / 8

VARIABLES

`next = 2` `serving = 0` `T0.my = 0` `T1.my = 1` `counter = 0`

OUTPUT

Ticket Lock

```
1  import java.util.concurrent.atomic.*;
2  public class Ticket {
3    static AtomicInteger next = new AtomicInteger();
4    static volatile int serving = 0;
5    static int counter = 0;
6    static void lock() {
7      int my = next.getAndIncrement();
▶      while (serving != my) Thread.onSpinWait();
9    }
10   static void unlock() { serving++; }
11   public static void main(String[] a)
12       throws Exception {
13     Runnable job = () -> {
14       for (int i = 0; i < 100_000; i++) {
▶         lock(); counter++; unlock(); } };
16     Thread t0 = new Thread(job);
17     Thread t1 = new Thread(job);
18     t0.start(); t1.start(); t0.join(); t1.join();
19     System.out.println("counter = " + counter);
20   }
21 }
```



T0: serving == 0 → leaves the loop and does `counter++` .

Step 4 / 8

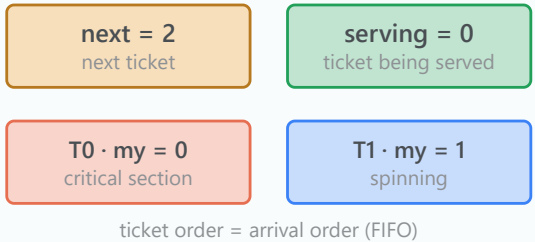
VARIABLES

`next = 2` `serving = 0` `T0.my = 0` `T1.my = 1` `counter = 1`

OUTPUT

Ticket Lock

```
1  import java.util.concurrent.atomic.*;
2  public class Ticket {
3    static AtomicInteger next = new AtomicInteger();
4    static volatile int serving = 0;
5    static int counter = 0;
6    static void lock() {
7      int my = next.getAndIncrement();
▶      while (serving != my) Thread.onSpinWait();
9    }
10   static void unlock() { serving++; }
11   public static void main(String[] a)
12       throws Exception {
13     Runnable job = () -> {
14       for (int i = 0; i < 100_000; i++) {
15         lock(); counter++; unlock(); } };
16     Thread t0 = new Thread(job);
17     Thread t1 = new Thread(job);
18     t0.start(); t1.start(); t0.join(); t1.join();
19     System.out.println("counter = " + counter);
20   }
21 }
```



T1 spins: it reads `serving` from the **S** copy in its own cache without loading the bus; `onSpinWait` = x86 `PAUSE`.

Step 5 / 8

VARIABLES

`next = 2` `serving = 0` `T0.my = 0` `T1.my = 1` `counter = 1`

OUTPUT

Ticket Lock

```
1  import java.util.concurrent.atomic.*;
2  public class Ticket {
3    static AtomicInteger next = new AtomicInteger();
4    static volatile int serving = 0;
5    static int counter = 0;
6    static void lock() {
7      int my = next.getAndIncrement();
8      while (serving != my) Thread.onSpinWait();
9    }
▶    static void unlock() { serving++; }
11   public static void main(String[] a)
12       throws Exception {
13     Runnable job = () -> {
14       for (int i = 0; i < 100_000; i++) {
15         lock(); counter++; unlock(); } };
16     Thread t0 = new Thread(job);
17     Thread t1 = new Thread(job);
18     t0.start(); t1.start(); t0.join(); t1.join();
19     System.out.println("counter = " + counter);
20   }
21 }
```



T0 releases: `serving++`. The write invalidates T1's copy (a single coherence message).

Step 6 / 8

VARIABLES

`next = 2` `serving = 1` `T0.my = 0` `T1.my = 1` `counter = 1`

OUTPUT

Ticket Lock

```
1  import java.util.concurrent.atomic.*;
2  public class Ticket {
3    static AtomicInteger next = new AtomicInteger();
4    static volatile int serving = 0;
5    static int counter = 0;
6    static void lock() {
7      int my = next.getAndIncrement();
▶      while (serving != my) Thread.onSpinWait();
9    }
10   static void unlock() { serving++; }
11   public static void main(String[] a)
12       throws Exception {
13     Runnable job = () -> {
14       for (int i = 0; i < 100_000; i++) {
▶         lock(); counter++; unlock(); } };
16     Thread t0 = new Thread(job);
17     Thread t1 = new Thread(job);
18     t0.start(); t1.start(); t0.join(); t1.join();
19     System.out.println("counter = " + counter);
20   }
21 }
```



T1 reads the new value: 1 == my → it enters. The lock is **fair**: nobody starves.

Step 7 / 8

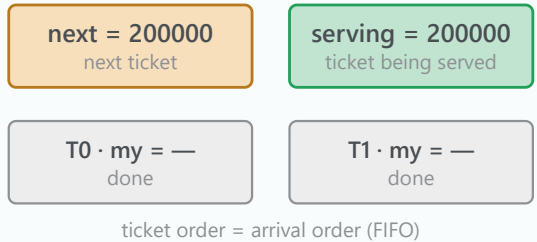
VARIABLES

`next = 2` `serving = 1` `T0.my = 0` `T1.my = 1` `counter = 2`

OUTPUT

Ticket Lock

```
1  import java.util.concurrent.atomic.*;
2  public class Ticket {
3    static AtomicInteger next = new AtomicInteger();
4    static volatile int serving = 0;
5    static int counter = 0;
6    static void lock() {
7      int my = next.getAndIncrement();
8      while (serving != my) Thread.onSpinWait();
9    }
10   static void unlock() { serving++; }
11   public static void main(String[] a)
12       throws Exception {
13     Runnable job = () -> {
14       for (int i = 0; i < 100_000; i++) {
15         lock(); counter++; unlock(); } };
16     Thread t0 = new Thread(job);
17     Thread t1 = new Thread(job);
18     t0.start(); t1.start(); t0.join(); t1.join();
▶      System.out.println("counter = " + counter);
20   }
21 }
```



2 × 100,000 increments, none lost. With an unlocked `counter++`, most runs end below 200,000.

Step 8 / 8

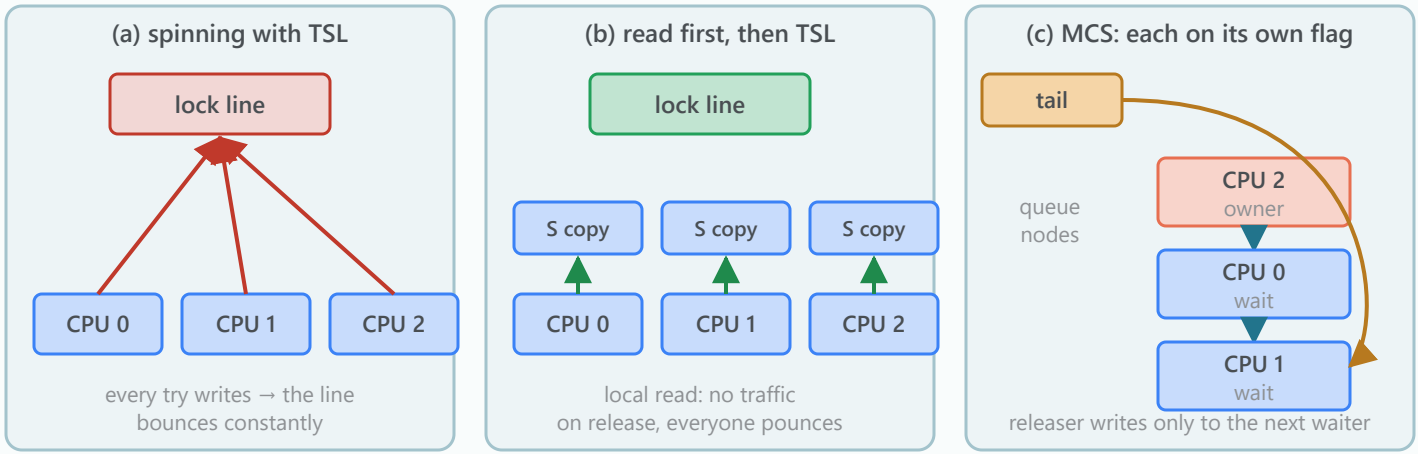
VARIABLES

`next = 200000` `serving = 200000` `T0.my = –` `T1.my = –` `counter = 200000`

OUTPUT

```
counter = 200000
```

Spinlocks and Cache Traffic

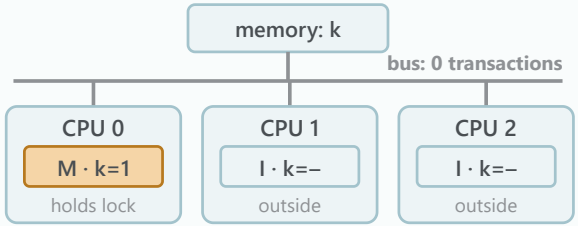


- (a) A plain TSL loop **writes** on every try: the lock line moves between CPUs nonstop, slowing the owner's work too.
- (b) **Test-and-test-and-set** (TTAS): spin on the local copy; try TSL only when the lock looks free. **Exponential backoff** further reduces collisions.
- (c) **MCS** (Mellor-Crummey & Scott, 1991): each waiter spins on its own flag in a separate line; a release is a single remote write. Linux `qspinlock` builds on this idea.

These locks do not prevent data inconsistency; they prevent coherence **traffic** (cache thrashing).

TTAS Lock: Code Trace

```
▶ void lock(int *k) {
2   for (;;) {
3     while (*k == 1)   /* spin locally */
4       ;
5     if (tsl(k) == 0)  /* looked free */
6       return;         /* acquired */
7   }
8 }
9 void unlock(int *k) {
10  *k = 0;
11 }
```



cache line state: M, E, S, I

CPU 0 holds the lock (k = 1, the line is **M** in its cache). CPU 1 and CPU 2 call `lock`.

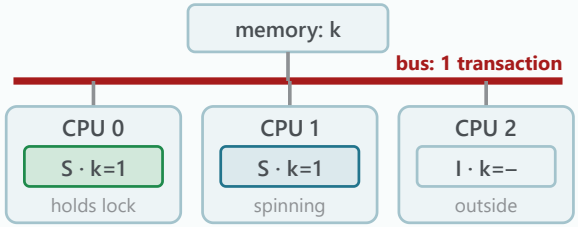
Step 1 / 10

VARIABLES

`k = 1` `CPU1 = I` `CPU2 = I` `bus = 0`

TTAS Lock: Code Trace

```
1  void lock(int *k) {
2    for (;;) {
▶      while (*k == 1)   /* spin locally */
4        ;
5      if (tsl(k) == 0)  /* looked free */
6        return;         /* acquired */
7    }
8  }
9  void unlock(int *k) {
10   *k = 0;
11 }
```



cache line state: M, E, S, I

CPU 1 reads `*k` : a miss, fetched over the bus. CPU 0's M copy is shared; both are **S**.

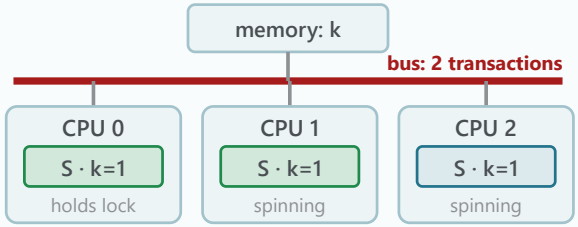
Step 2 / 10

VARIABLES

`k = 1` `CPU1 = S` `CPU2 = I` `bus = 1`

TTAS Lock: Code Trace

```
1  void lock(int *k) {
2    for (;;) {
▶      while (*k == 1)   /* spin locally */
4        ;
5      if (tsl(k) == 0)  /* looked free */
6        return;         /* acquired */
7    }
8  }
9  void unlock(int *k) {
10   *k = 0;
11 }
```



cache line state: M, E, S, I

CPU 2 reads too: three S copies. 2 bus transactions so far.

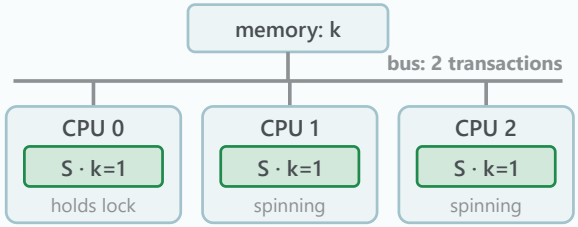
Step 3 / 10

VARIABLES

`k = 1` `CPU1 = S` `CPU2 = S` `bus = 2`

TTAS Lock: Code Trace

```
1  void lock(int *k) {
2    for (;;) {
▶      while (*k == 1)   /* spin locally */
▶        ;
5      if (tsl(k) == 0)  /* looked free */
6        return;         /* acquired */
7    }
8  }
9  void unlock(int *k) {
10   *k = 0;
11 }
```



cache line state: M, E, S, I

As long as they see k = 1 they spin **in their own caches**: the bus counter does not move. A plain TSL loop would pull the line for writing on every try.

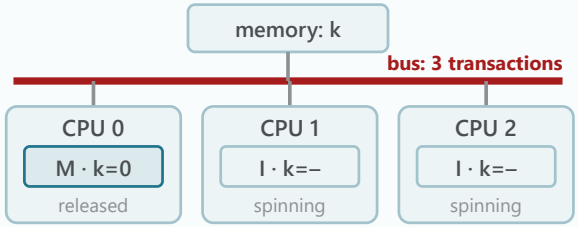
Step 4 / 10

VARIABLES

`k = 1` `CPU1 = S` `CPU2 = S` `bus = 2`

TTAS Lock: Code Trace

```
1  void lock(int *k) {
2    for (;;) {
3      while (*k == 1)   /* spin locally */
4        ;
5      if (tsl(k) == 0)  /* looked free */
6        return;         /* acquired */
7    }
8  }
9  void unlock(int *k) {
▶    *k = 0;
11 }
```



cache line state: M, E, S, I

CPU 0 calls `unlock` : writing k = 0 **invalidates** the other copies (I); the line is M on CPU 0.

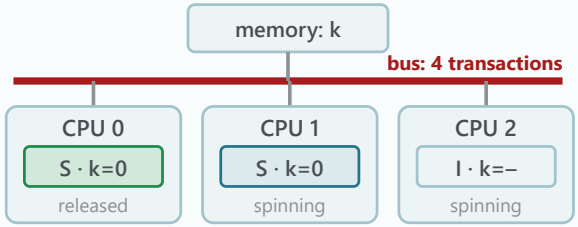
Step 5 / 10

VARIABLES

`k = 0` `CPU1 = I` `CPU2 = I` `bus = 3`

TTAS Lock: Code Trace

```
1  void lock(int *k) {
2    for (;;) {
▶      while (*k == 1)   /* spin locally */
4        ;
5      if (tsl(k) == 0)  /* looked free */
6        return;         /* acquired */
7    }
8  }
9  void unlock(int *k) {
10   *k = 0;
11 }
```



cache line state: M, E, S, I

CPU 1's copy is invalid: it rereads (a miss), sees k = 0 and leaves the while.

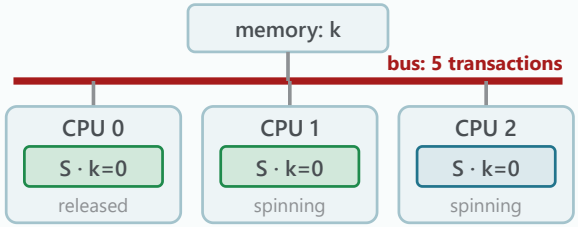
Step 6 / 10

VARIABLES

`k = 0` `CPU1 = S` `CPU2 = I` `bus = 4`

TTAS Lock: Code Trace

```
1  void lock(int *k) {
2    for (;;) {
▶      while (*k == 1)   /* spin locally */
4        ;
5      if (tsl(k) == 0)  /* looked free */
6        return;         /* acquired */
7    }
8  }
9  void unlock(int *k) {
10   *k = 0;
11 }
```



cache line state: M, E, S, I

CPU 2 also misses and sees k = 0. Both think the lock is “free” at the same time.

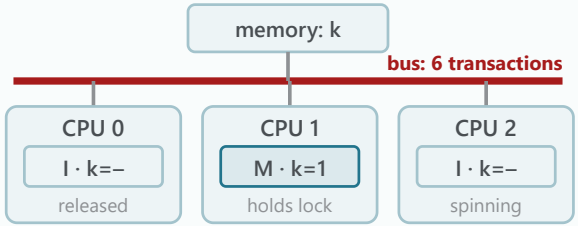
Step 7 / 10

VARIABLES

`k = 0` `CPU1 = S` `CPU2 = S` `bus = 5`

TTAS Lock: Code Trace

```
1  void lock(int *k) {
2    for (;;) {
3      while (*k == 1)   /* spin locally */
4        ;
▶      if (tsl(k) == 0)  /* looked free */
▶        return;         /* acquired */
7    }
8  }
9  void unlock(int *k) {
10   *k = 0;
11 }
```



cache line state: M, E, S, I

CPU 1 does TSL first: it takes the line for writing (M), old value 0 → **lock acquired**.

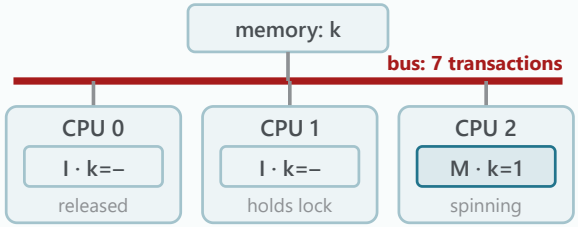
Step 8 / 10

VARIABLES

`k = 1` `CPU1 = M` `CPU2 = I` `bus = 6`

TTAS Lock: Code Trace

```
1  void lock(int *k) {
2    for (;;) {
3      while (*k == 1)   /* spin locally */
4        ;
▶      if (tsl(k) == 0)  /* looked free */
6        return;         /* acquired */
7    }
8  }
9  void unlock(int *k) {
10   *k = 0;
11 }
```



cache line state: M, E, S, I

CPU 2's TSL also pulls the line (CPU 1's copy goes I), but the old value is 1: it fails. This race costs one more bus transaction.

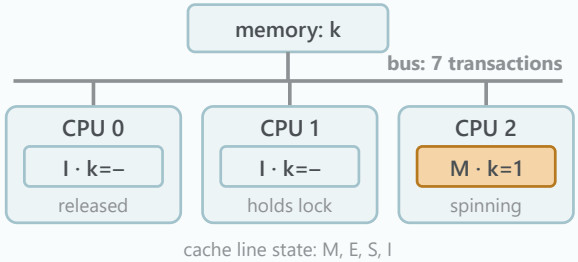
Step 9 / 10

VARIABLES

`k = 1` `CPU1 = I` `CPU2 = M` `bus = 7`

TTAS Lock: Code Trace

```
1  void lock(int *k) {
2    for (;;) {
▶      while (*k == 1)   /* spin locally */
▶        ;
5      if (tsl(k) == 0)  /* looked free */
6        return;         /* acquired */
7    }
8  }
9  void unlock(int *k) {
10   *k = 0;
11 }
```



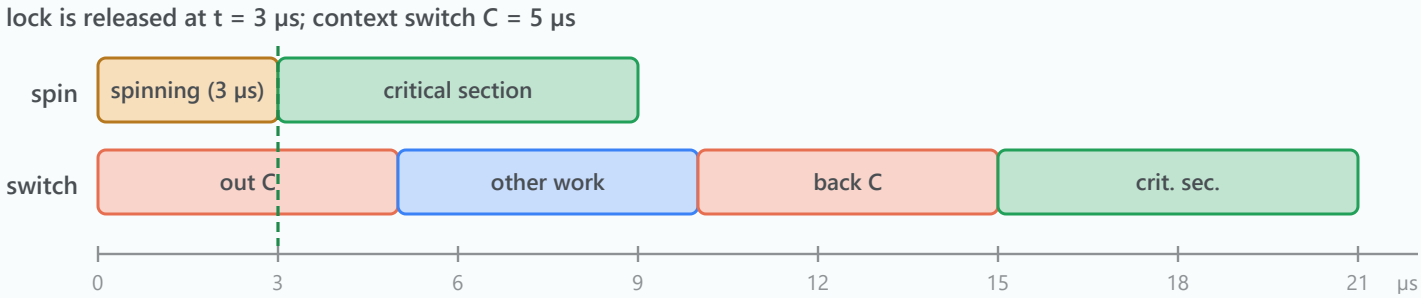
CPU 2 spins on its local copy again. 7 bus transactions in all, and the count stays put however long the wait.

Step 10 / 10

VARIABLES

`k = 1` `CPU1 = I` `CPU2 = M` `bus = 7`

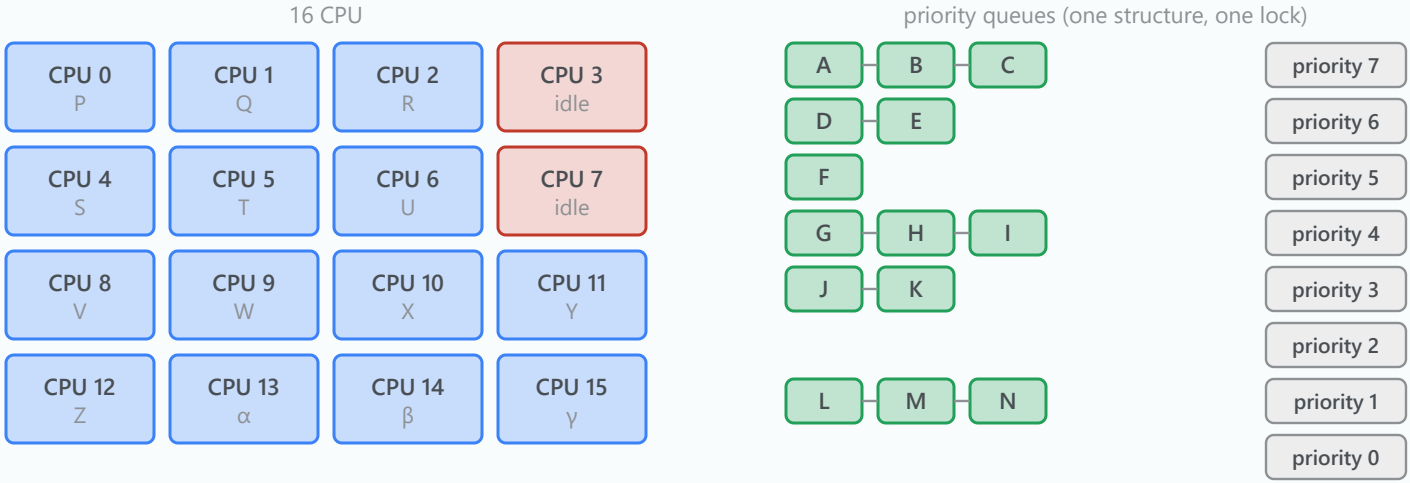
Spin or Switch?



- **Spinning**: the CPU is wasted, but work resumes the moment the lock frees. For short critical sections.
- **Switching** (blocking): the CPU is given to other work; the cost is two context switches + a cold cache.
- Rule: if the expected wait is shorter than 2C, spin; otherwise sleep.

- **Competitive** strategy: spin for as long as a switch costs (2C), then sleep → at worst twice the optimal cost.
- In practice: Linux `futex` + adaptive mutexes (spin while the owner runs), Windows `CRITICAL_SECTION` spin counts, lock inflation in Java.
- In the kernel, only spinlocks may be used where interrupts are disabled.

Time Sharing: A Shared Ready Queue



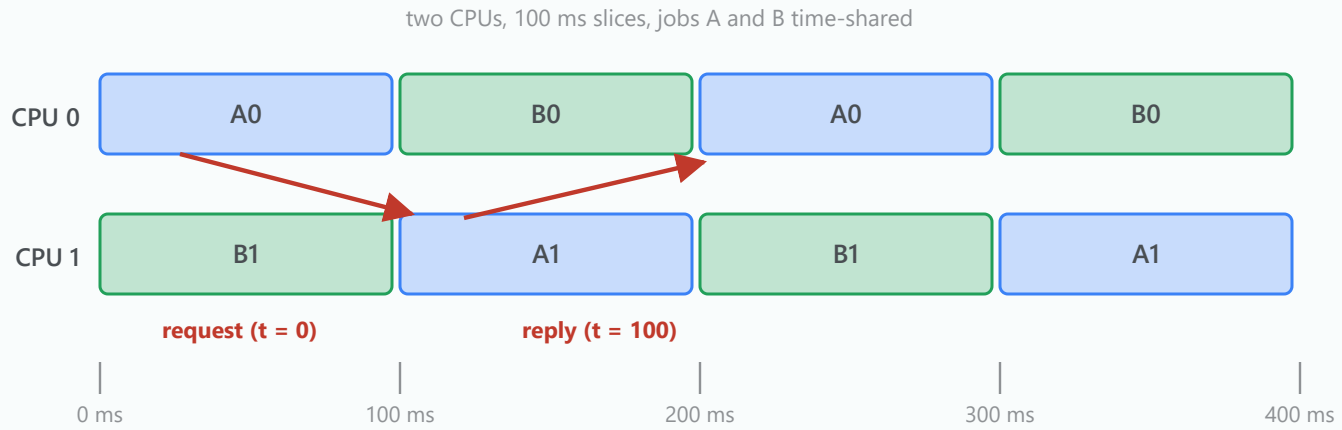
- With independent processes, all CPUs take work from one priority ready queue: the load balances itself, but the single lock becomes a bottleneck and a process may land on a CPU with a cold cache.
- If a thread holding a spinlock runs out of quantum, the other CPUs spin in vain. **Smart scheduling**: the lock holder sets a flag, and the scheduler does not preempt a flagged thread but gives it a little extra time to finish its critical section.

Space Sharing



- Related threads run **at the same time** on separate CPUs: when a job starts it gets as many CPUs as it asks for and keeps them until it finishes.
- No multiprogramming per CPU → no context switches. If there are not enough CPUs, the job waits until enough are free (simplest: FCFS).
- Drawback: when a thread blocks on I/O its CPU sits **idle**. HPC cluster job schedulers (Slurm, PBS) partition nodes this way today.

Gang Scheduling

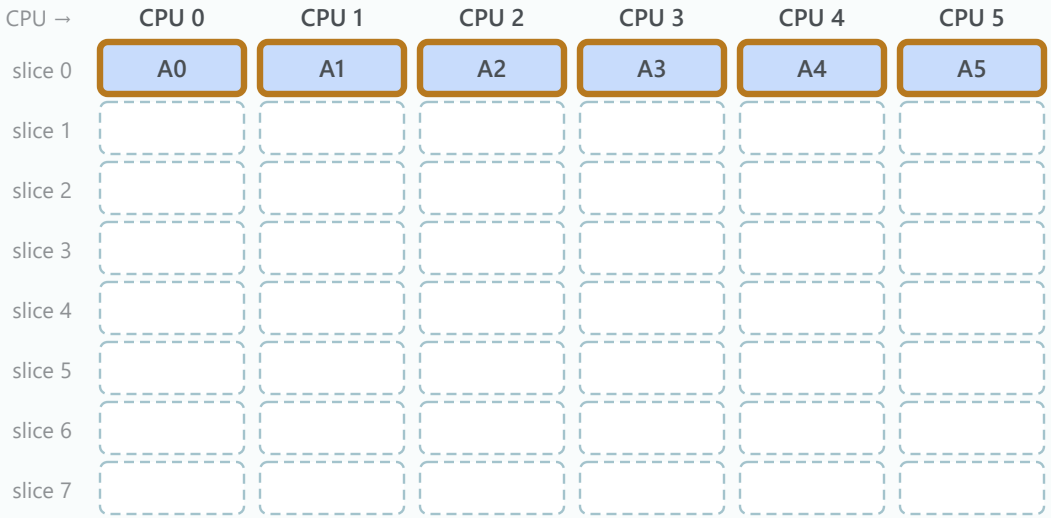


A1 sees the request only at t = 100; A0 gets the reply at t = 200

if communicating threads run at different times, every message waits a slice

The problem: A0 and A1 communicate, but their slices are staggered. A1 sees A0's request from t = 0 only 100 ms later, and the reply waits 100 ms too.

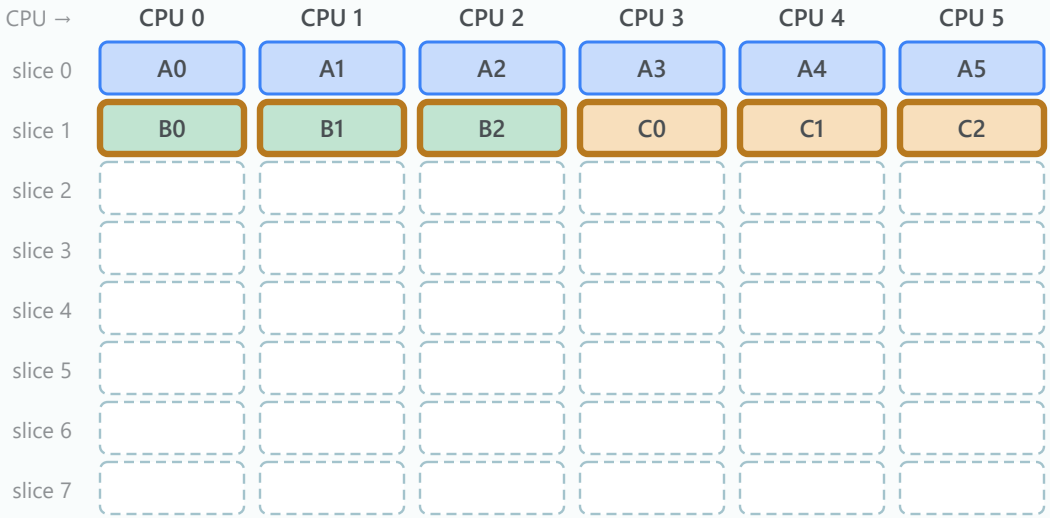
Gang Scheduling



A: 6 threads
B, C: 3 each
D: 5 · E: 7

In gang scheduling, related threads form a **gang**. Slice 0: all 6 threads of A at once, on 6 CPUs.

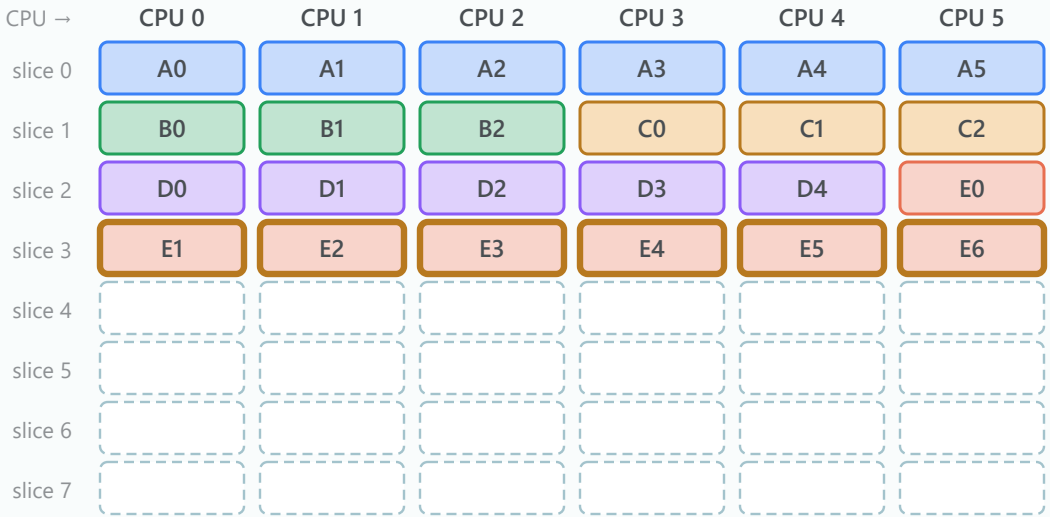
Gang Scheduling



A: 6 threads
B, C: 3 each
D: 5 · E: 7

Slice 1: B (3) and C (3) together; all CPUs start and end slices **in sync** (synchronized time slicing).

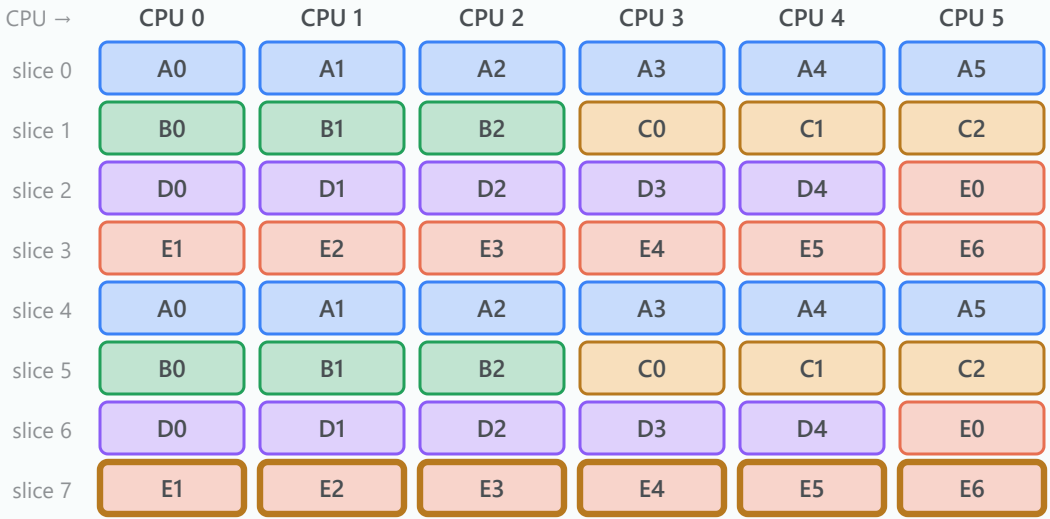
Gang Scheduling



A: 6 threads
B, C: 3 each
D: 5 · E: 7

Slices 2–3: D (5) + E0, then E's remaining 6 threads. All members of a gang run in the same slice, so messages get immediate replies.

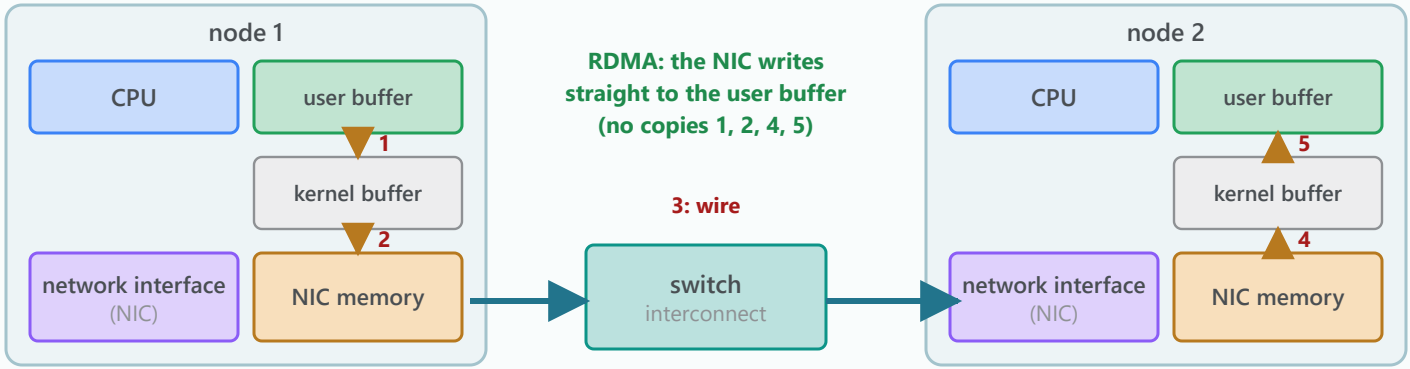
Gang Scheduling



A: 6 threads
B, C: 3 each
D: 5 · E: 7

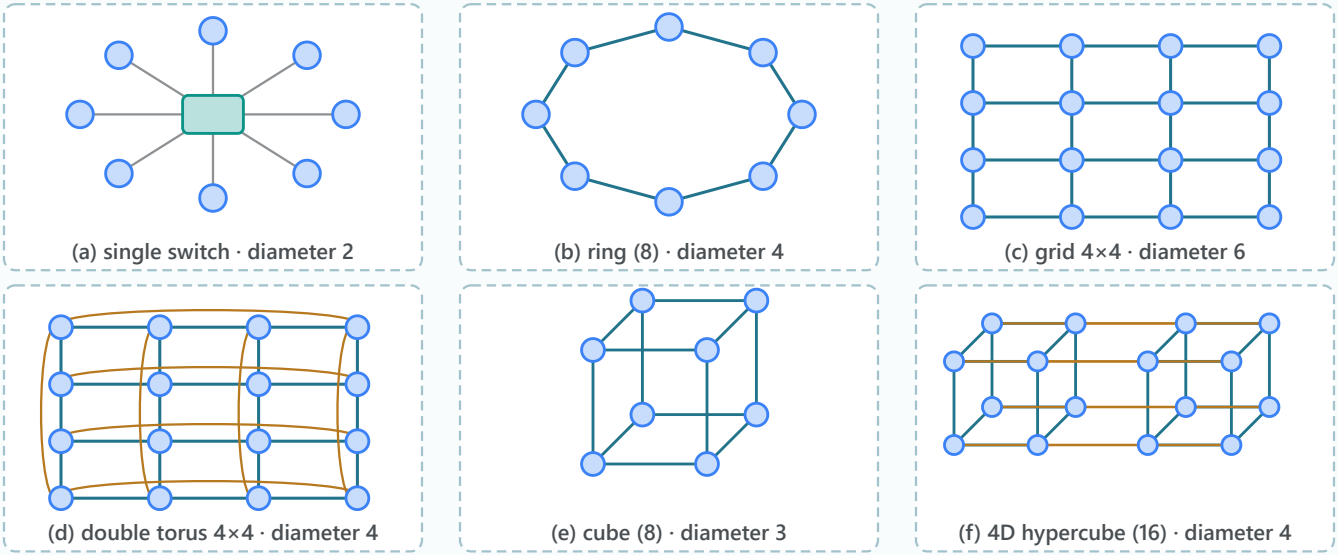
The cycle repeats. Two conditions: gang members start **at the same time** and all sit on different CPUs. Co-scheduling the vCPUs of a VM today is the same idea.

Multicomputers and Network Interfaces



- A **multicomputer** (cluster): each node is CPU + RAM + network interface; memory is not shared and each node runs its own OS copy. Communication is by **messages** only.
- In the worst case a message is copied 5 times, each copy costing microseconds. The fix: map NIC memory into user space and use **DMA** and **RDMA** (InfiniBand, RoCE), so the kernel never steps in.
- Uses: large-scale simulation, HPC, AI training; TOP500 systems are multicomputers with tens of thousands of nodes.

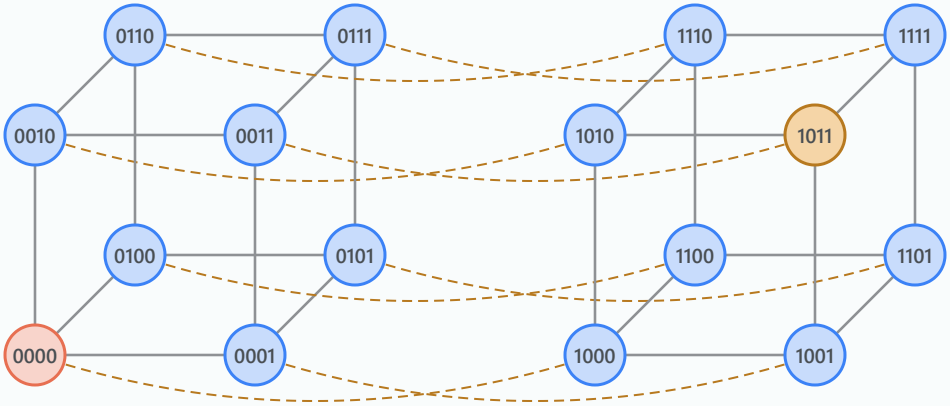
Interconnection Topologies



n = 1,024 nodes	ring	grid 32×32	torus 32×32	hypercube (10D)
diameter (two farthest nodes)	512	62	32	10
number of links	1.024	1.984	2.048	5.120

Diameter = worst-case hop count; $\log_2 n$ for a hypercube. Today's supercomputers mostly use *fat-tree* and *dragonfly* topologies; Fugaku has a 6D torus (Tofu) network.

Routing in a Hypercube



target 1011

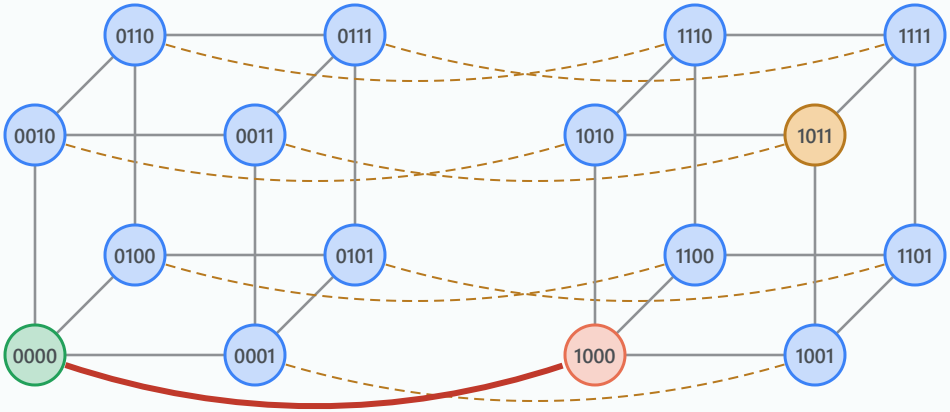
hops: 0

lower cube: first bit 0

upper cube: first bit 1

A 4D hypercube: 16 nodes, each linked to 4 neighbors that differ in exactly one bit. A message goes from 0000 to 1011.

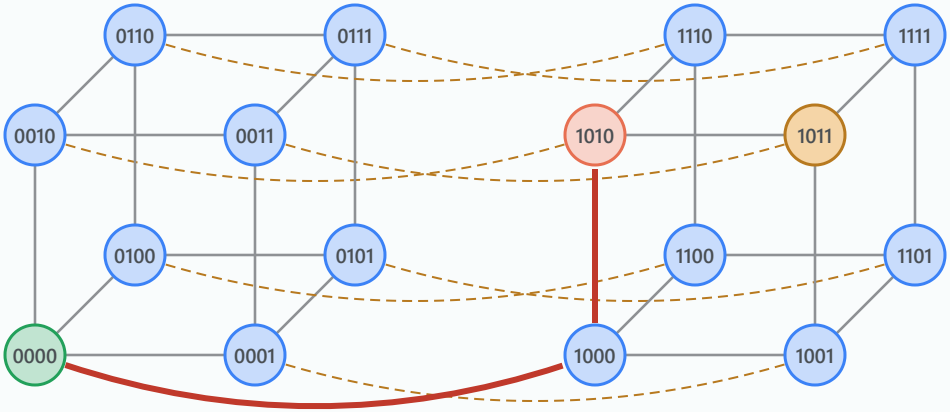
Routing in a Hypercube



target 1011
hops: 1

Source ⊕ target = 1011: three bits differ. The leftmost differing bit is fixed first → **1000** (from the lower to the upper cube).

Routing in a Hypercube



lower cube: first bit 0

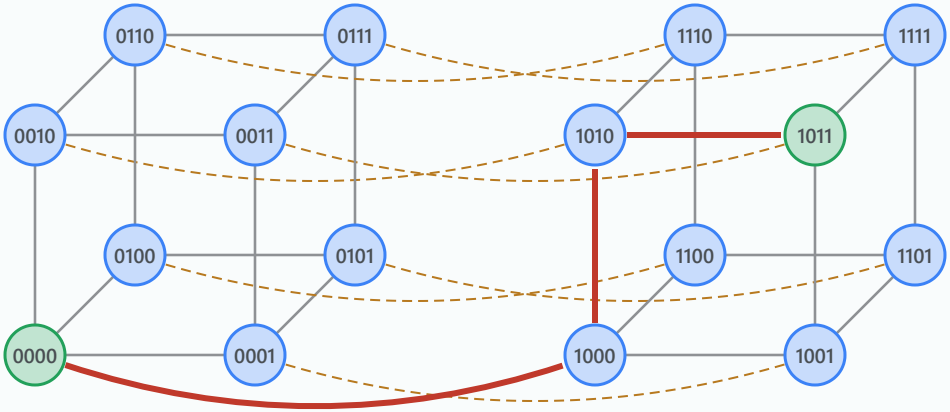
upper cube: first bit 1

target 1011

hops: 2

Next differing bit: **1010**.

Routing in a Hypercube



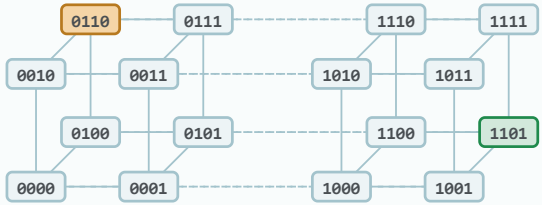
target 1011

hops: 3

Last bit: **1011**. Hops = number of differing bits (Hamming distance) = 3; the worst case is $d = \log_2 16 = 4$. The rule is local, so no routing table is needed.

E-Cube Routing: Code Trace

```
1  void route(int src, int dst, int d) {
2    int u = src;
3    for (int b = d - 1; b >= 0; b--) {
4      int m = 1 << b;
5      if ((u ^ dst) & m) {  /* bit differs */
6        u ^= m;             /* go to neighbor */
7        printf("%d ", u);
8      }
9    }
10 }
```



u = 0110 · u ⊕ dst = 1011

A message from 0110 (6) to 1101 (13). u ⊕ dst = 1011: three bits differ, so the shortest path is 3 hops. Dashed edges are the 4th dimension (between the two 3-cubes).

Step 1 / 6

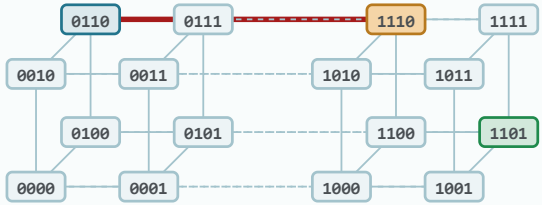
VARIABLES

`u = 0110` `dst = 1101` `b = –` `m = –`

OUTPUT

E-Cube Routing: Code Trace

```
1  void route(int src, int dst, int d) {
2    int u = src;
▶    for (int b = d - 1; b >= 0; b--) {
▶      int m = 1 << b;
▶      if ((u ^ dst) & m) {  /* bit differs */
▶        u ^= m;             /* go to neighbor */
▶        printf("%d ", u);
8      }
9    }
10 }
```



u = 1110 · u ⊕ dst = 0011 · bit 3

b = 3: the bit differs → flip bit 3 of u; the message moves to neighbor **1110** (14).

Step 2 / 6

VARIABLES

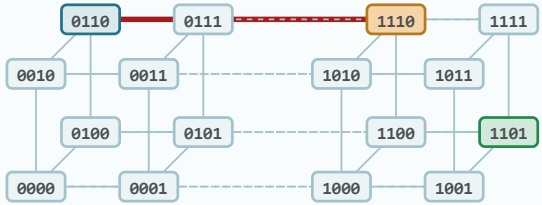
u = 1110 | dst = 1101 | b = 3 | m = 1000

OUTPUT

```
14
```

E-Cube Routing: Code Trace

```
1  void route(int src, int dst, int d) {
2    int u = src;
▶    for (int b = d - 1; b >= 0; b--) {
▶      int m = 1 << b;
▶      if ((u ^ dst) & m) {  /* bit differs */
6        u ^= m;             /* go to neighbor */
7        printf("%d ", u);
8      }
9    }
10 }
```



u = 1110 · u ⊕ dst = 0011 · bit 2

b = 2: this bit is the same (1) in u and dst; no hop.

Step 3 / 6

VARIABLES

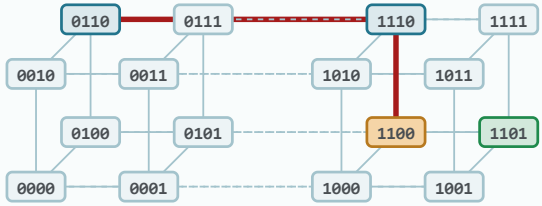
`u = 1110` `dst = 1101` `b = 2` `m = 0100`

OUTPUT

```
14
```

E-Cube Routing: Code Trace

```
1  void route(int src, int dst, int d) {
2    int u = src;
▶    for (int b = d - 1; b >= 0; b--) {
▶      int m = 1 << b;
▶      if ((u ^ dst) & m) {  /* bit differs */
▶        u ^= m;             /* go to neighbor */
▶        printf("%d ", u);
8      }
9    }
10 }
```



u = 1100 · u ⊕ dst = 0001 · bit 1

b = 1: the bit differs → flip bit 1 of u; the message moves to neighbor **1100** (12).

Step 4 / 6

VARIABLES

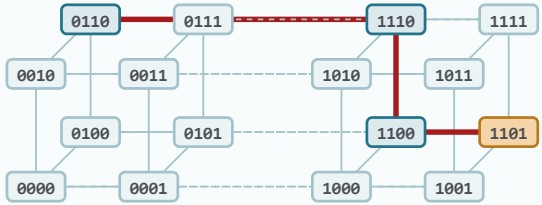
u = 1100 | dst = 1101 | b = 1 | m = 0010

OUTPUT

```
14 12
```

E-Cube Routing: Code Trace

```
1  void route(int src, int dst, int d) {
2    int u = src;
▶    for (int b = d - 1; b >= 0; b--) {
▶      int m = 1 << b;
▶      if ((u ^ dst) & m) {  /* bit differs */
▶        u ^= m;             /* go to neighbor */
▶        printf("%d ", u);
8      }
9    }
10 }
```



u = 1101 · u ⊕ dst = 0000 · bit 0

b = 0: the bit differs → flip bit 0 of u; the message moves to neighbor **1101** (13).

Step 5 / 6

VARIABLES

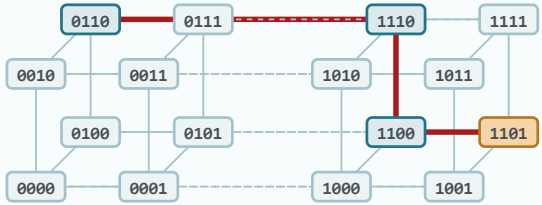
u = 1101 | dst = 1101 | b = 0 | m = 0001

OUTPUT

```
14 12 13
```

E-Cube Routing: Code Trace

```
1 void route(int src, int dst, int d) {
2   int u = src;
3   for (int b = d - 1; b >= 0; b--) {
4     int m = 1 << b;
5     if ((u ^ dst) & m) {  /* bit differs */
6       u ^= m;             /* go to neighbor */
7       printf("%d ", u);
8     }
9   }
▶ }
```



u = 1101 · u ⊕ dst = 0000

The target is reached in 3 hops: 0110 → 1110 → 1100 → 1101. Each node decides from its own address and the target alone; since bits are always fixed left to right, the path has no cycles.

Step 6 / 6

VARIABLES

`u = 1101` `dst = 1101` `b = -1` `m = –`

OUTPUT

```
14 12 13
```

Store-and-Forward or Cut-Through



store-and-forward: each switch receives the whole packet, then forwards it

latency ≈ hops × (packet / bandwidth)

t = 0 µs

Example: a 4 KiB packet over 10 Gb/s links; crossing one link takes $4096 \times 8 / 10^{10} \approx$ **3.28 µs**.

Store-and-Forward or Cut-Through



store-and-forward: each switch receives the whole packet, then forwards it

latency ≈ hops × (packet / bandwidth)

t = 3.28 µs

Store-and-forward: switch 1 buffers the **whole** packet, checks it, then forwards it.

Store-and-Forward or Cut-Through



store-and-forward: each switch receives the whole packet, then forwards it

latency ≈ hops × (packet / bandwidth)

t = 9.83 µs

Three links → 3 × 3.28 = **9.83 µs**. Latency grows linearly with the hop count.

Store-and-Forward or Cut-Through



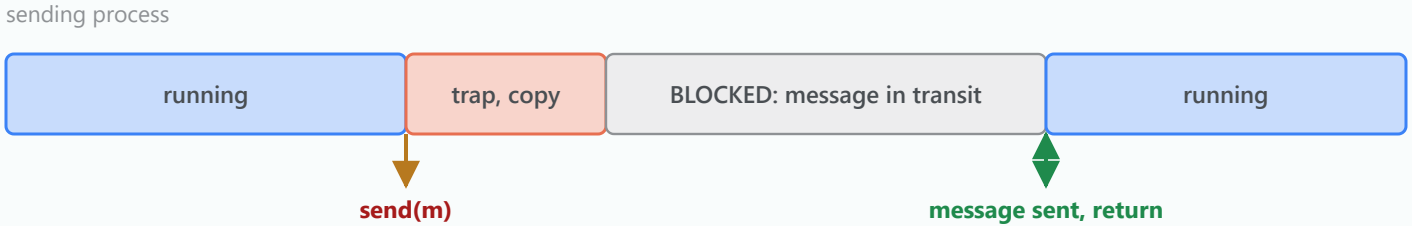
cut-through / wormhole: forwarded as soon as the header is read

pieces of the packet (flits) on several links at once

t ≈ 3.38 µs (one packet time + small header delays)

Cut-through and wormhole routing: the path is set up when the header arrives and the packet flows like a pipe → ≈ **3.38 µs**. HPC networks and data-center switches work this way.

Blocking and Nonblocking Send



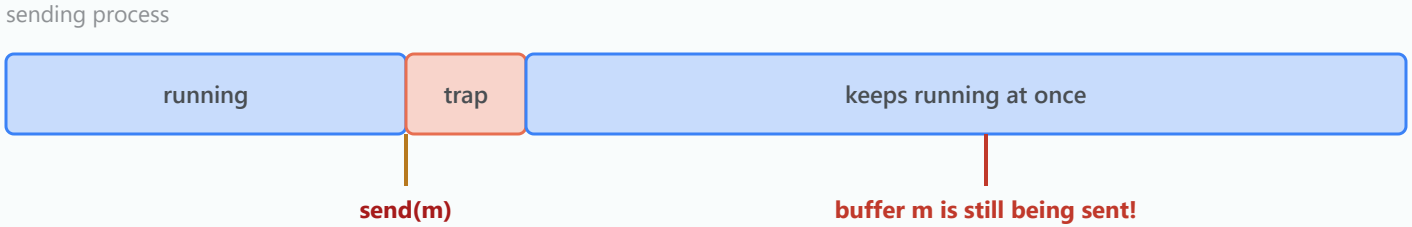
simple and safe, but the CPU idles for the whole transmission

(a) **Blocking** (synchronous) `send` : the call does not return until the message has been transmitted.

Step 1 / 3

In MPI, `MPI_Send` / `MPI_Recv` are blocking and `MPI_Isend` / `MPI_Irecv` + `MPI_Wait` are nonblocking; on Linux, `io_uring` and zero-copy `MSG_ZEROCOPY` are modern answers to the same problem.

Blocking and Nonblocking Send



we get parallelism, but what if the sender changes m before transmission ends?

(b) **Nonblocking** (asynchronous) `send` : returns at once, but the buffer is off-limits until transmission ends.

Step 2 / 3

In MPI, `MPI_Send` / `MPI_Recv` are blocking and `MPI_Isend` / `MPI_Irecv` + `MPI_Wait` are nonblocking; on Linux, `io_uring` and zero-copy `MSG_ZEROCOPY` are modern answers to the same problem.

Blocking and Nonblocking Send

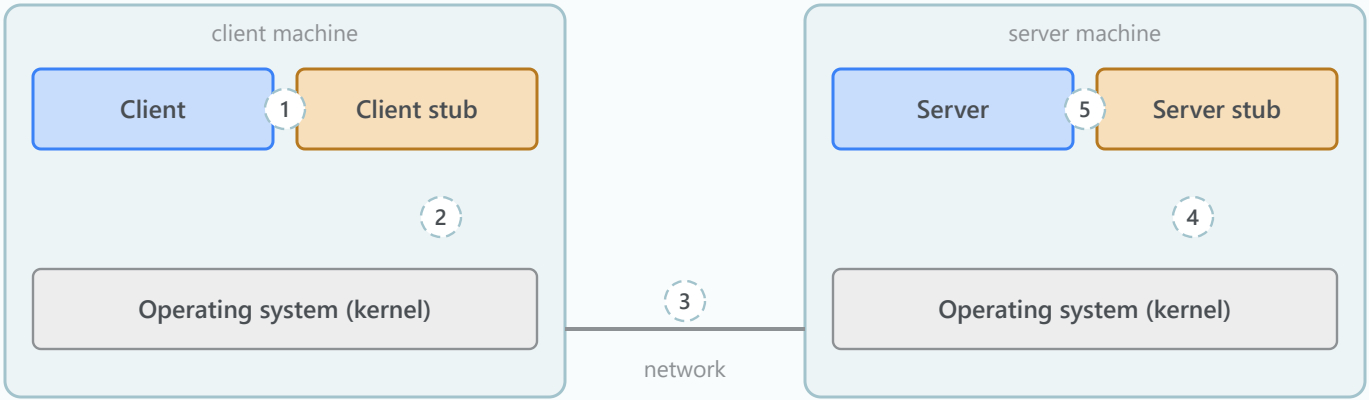
Option	Cost
Blocking send	CPU idles during transmission
Nonblocking + copy to kernel	CPU time for the extra copy
Nonblocking + interrupt when done	hard to program (race conditions)
Copy on write (CoW)	an extra copy anyway if the buffer is written

Tanenbaum's four options. The receiver side is similar: a blocking `receive`, polling, or a pop-up thread / active message.

Step 3 / 3

In MPI, `MPI_Send` / `MPI_Recv` are blocking and `MPI_Isend` / `MPI_Irecv` + `MPI_Wait` are nonblocking; on Linux, `io_uring` and zero-copy `MSG_ZEROCOPY` are modern answers to the same problem.

Remote Procedure Call (RPC)



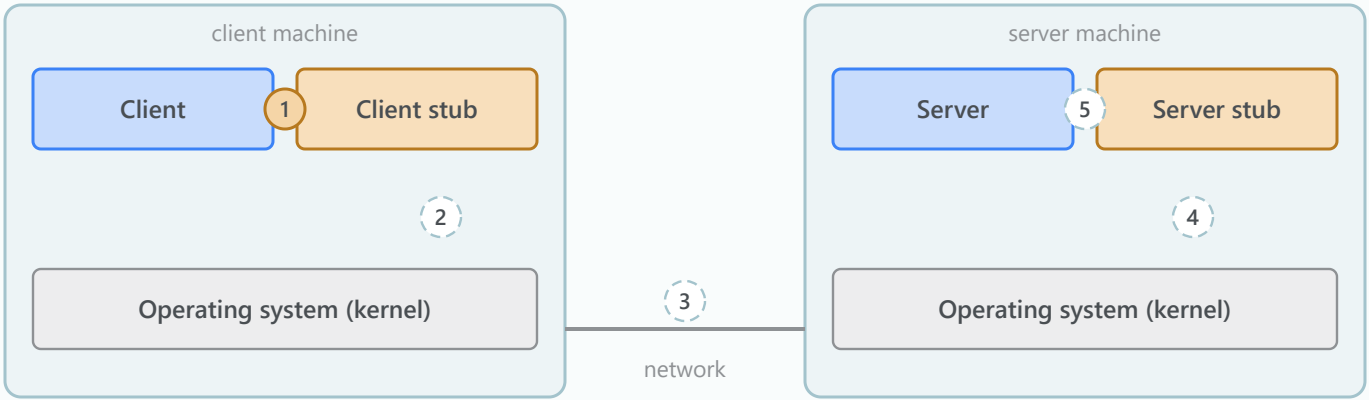
goal: make a remote call look like a local one (Birrell and Nelson, 1984)

The client says `result = add(3, 4)`; it looks like an ordinary local call.

Step 1 / 6

Difficulties: pointers (another address space → copy/restore), arrays of unknown length, global variables, partial failures (at least once or at most once?). Stubs are generated automatically from an interface definition; today gRPC + Protocol Buffers, Thrift, Java RMI.

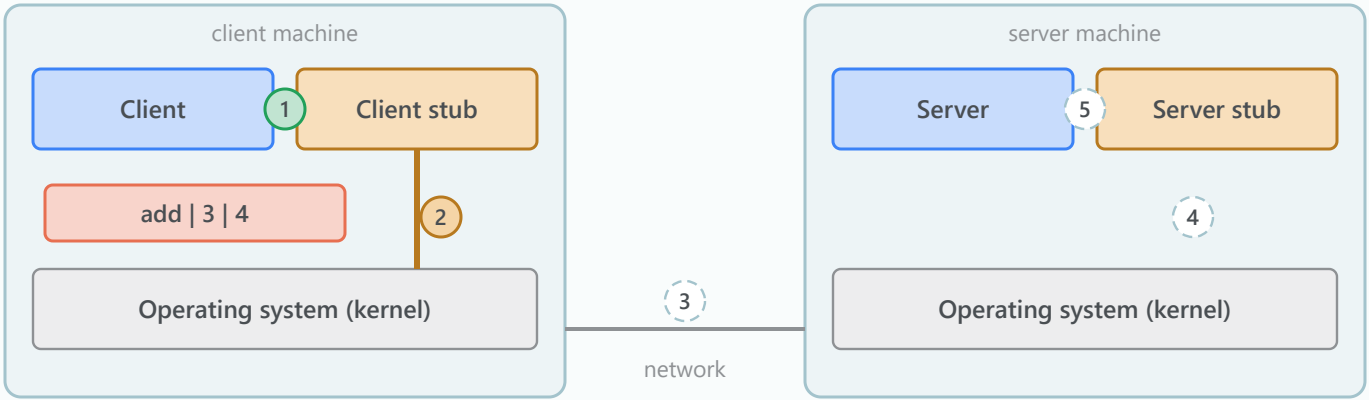
Remote Procedure Call (RPC)



1: The client calls the **client stub** as if it were a local procedure.

Difficulties: pointers (another address space → copy/restore), arrays of unknown length, global variables, partial failures (at least once or at most once?). Stubs are generated automatically from an interface definition; today gRPC + Protocol Buffers, Thrift, Java RMI.

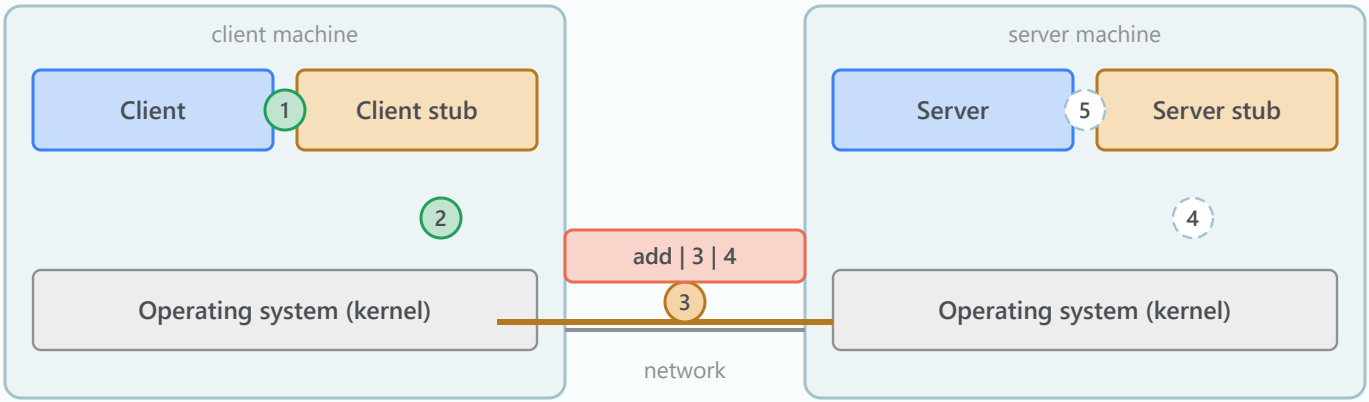
Remote Procedure Call (RPC)



2: The stub **marshals** the parameters into a message and makes a system call into the kernel.

Difficulties: pointers (another address space → copy/restore), arrays of unknown length, global variables, partial failures (at least once or at most once?). Stubs are generated automatically from an interface definition; today gRPC + Protocol Buffers, Thrift, Java RMI.

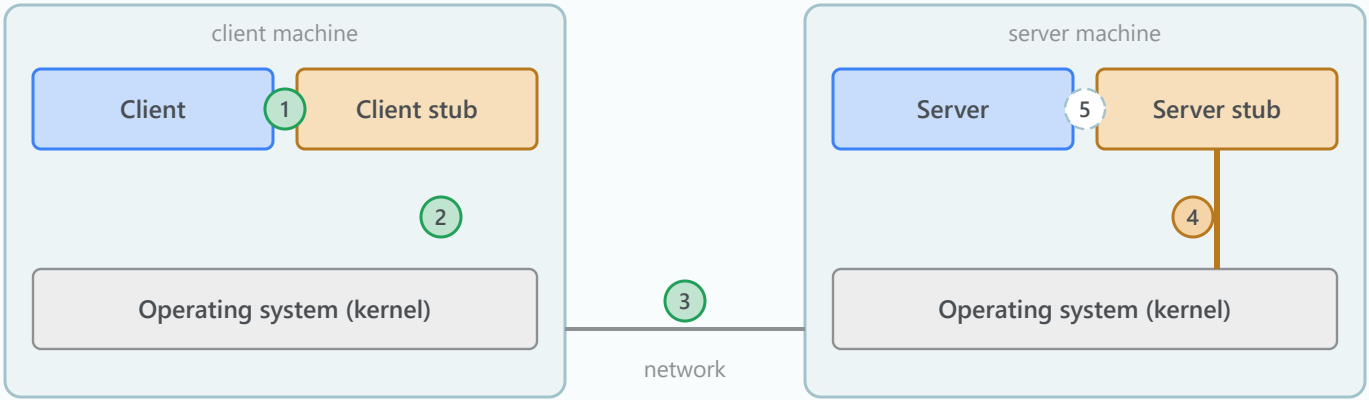
Remote Procedure Call (RPC)



3: The kernel sends the message over the network to the server machine.

Difficulties: pointers (another address space → copy/restore), arrays of unknown length, global variables, partial failures (at least once or at most once?). Stubs are generated automatically from an interface definition; today gRPC + Protocol Buffers, Thrift, Java RMI.

Remote Procedure Call (RPC)

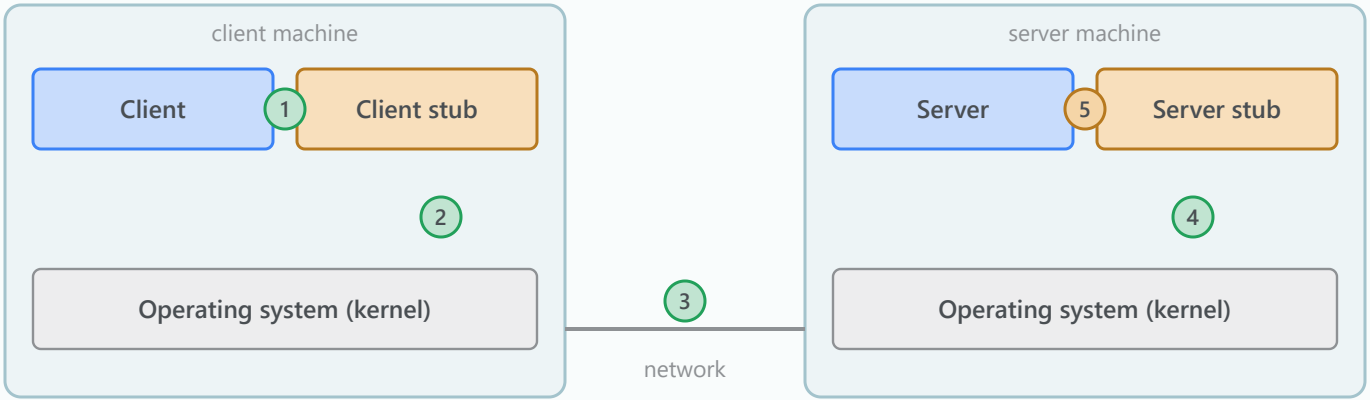


4: The server kernel hands the message to the **server stub**.

Step 5 / 6

Difficulties: pointers (another address space → copy/restore), arrays of unknown length, global variables, partial failures (at least once or at most once?). Stubs are generated automatically from an interface definition; today gRPC + Protocol Buffers, Thrift, Java RMI.

Remote Procedure Call (RPC)

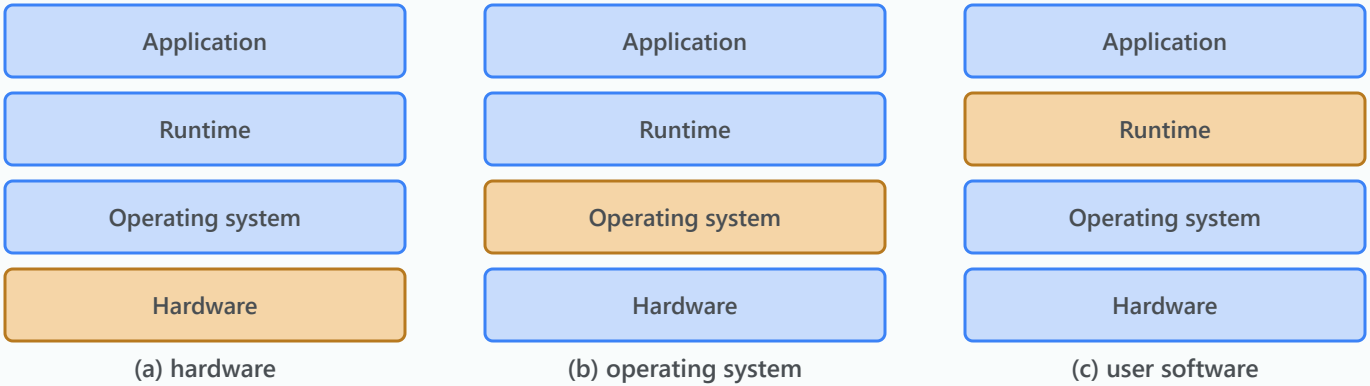


reply (7) takes the reverse path: server → stub → kernel → network → client

5: The stub unpacks the parameters and calls the server procedure locally; the reply returns the same way.

Difficulties: pointers (another address space → copy/restore), arrays of unknown length, global variables, partial failures (at least once or at most once?). Stubs are generated automatically from an interface definition; today gRPC + Protocol Buffers, Thrift, Java RMI.

Distributed Shared Memory (DSM)



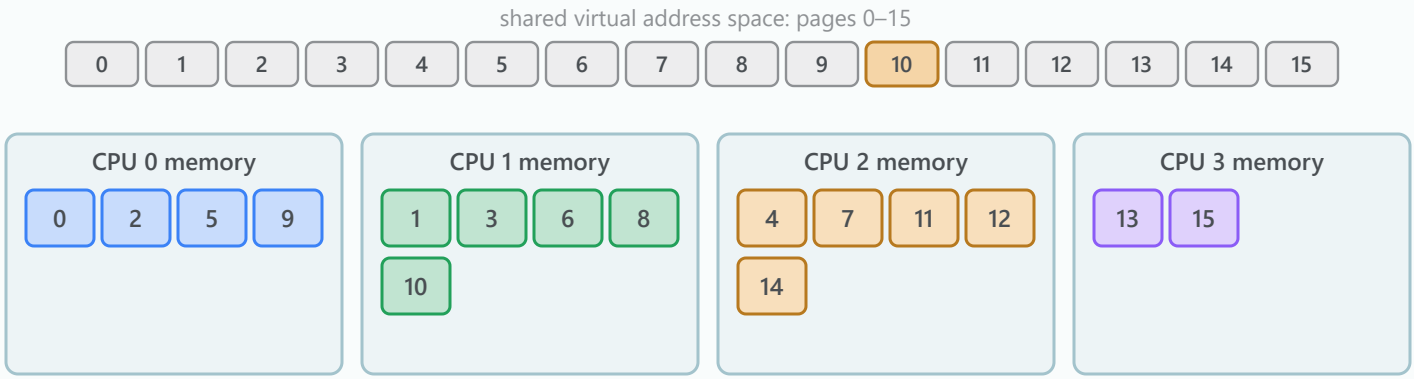
the orange layer creates the illusion of a shared address space

DSM gives machines that share no memory the **illusion** of shared memory. The layer can be hardware (NUMA), the OS (page-based, Li and Hudak 1989) or a user library.

Step 1 / 4

A 4 KiB page is expensive and coarse-grained on a network, so DSM gave way to message passing (MPI) and PGAS languages. Its modern counterpart is cross-machine memory pooling with CXL.

Distributed Shared Memory (DSM)



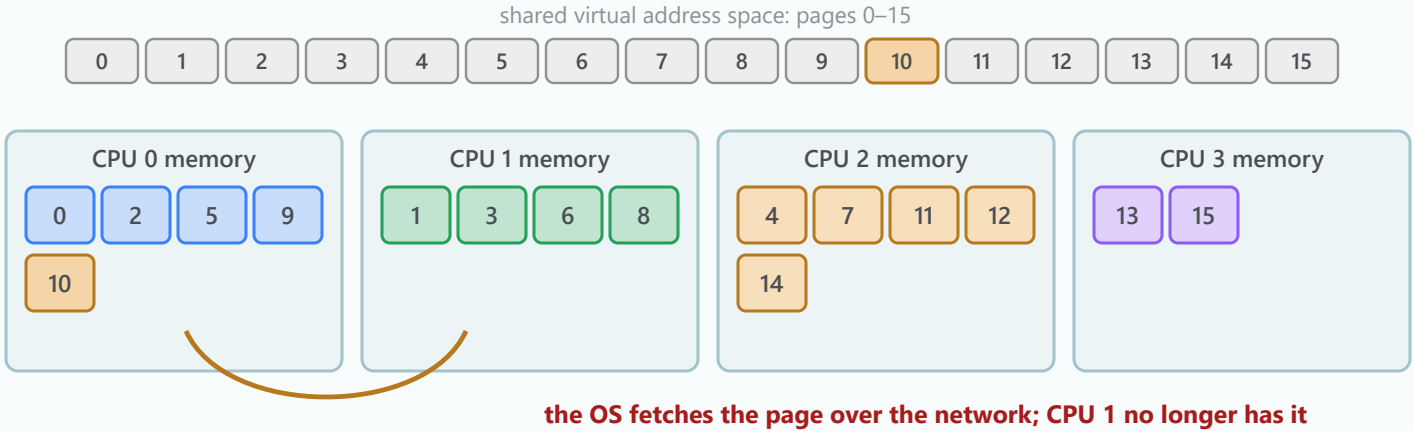
CPU 0 touches page 10 → page fault (not on CPU 0)

Pages are spread across the machines. When CPU 0 touches page 10, the MMU raises a page fault.

Step 2 / 4

A 4 KiB page is expensive and coarse-grained on a network, so DSM gave way to message passing (MPI) and PGAS languages. Its modern counterpart is cross-machine memory pooling with CXL.

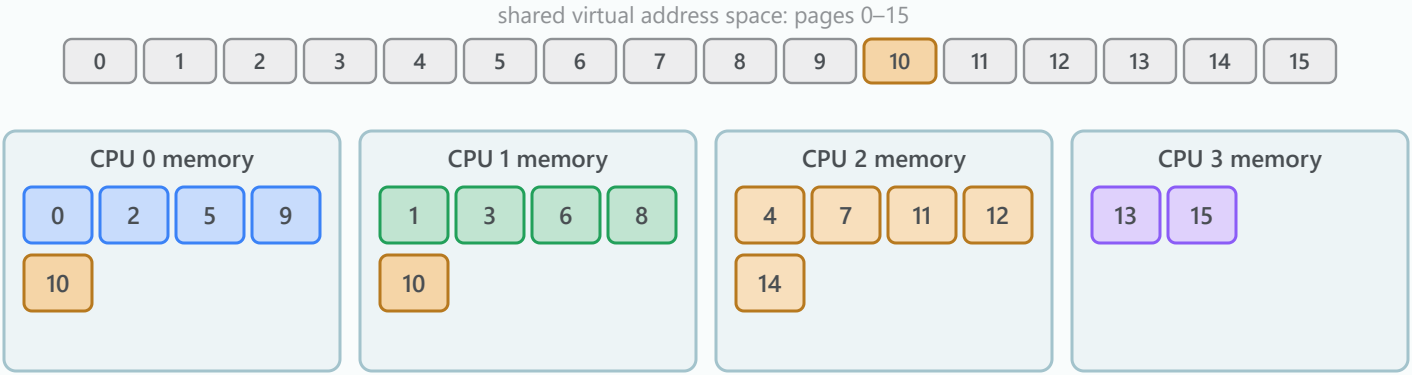
Distributed Shared Memory (DSM)



The OS finds who has the page and requests it; the page **moves** to CPU 0 and the instruction is restarted.

A 4 KiB page is expensive and coarse-grained on a network, so DSM gave way to message passing (MPI) and PGAS languages. Its modern counterpart is cross-machine memory pooling with CXL.

Distributed Shared Memory (DSM)

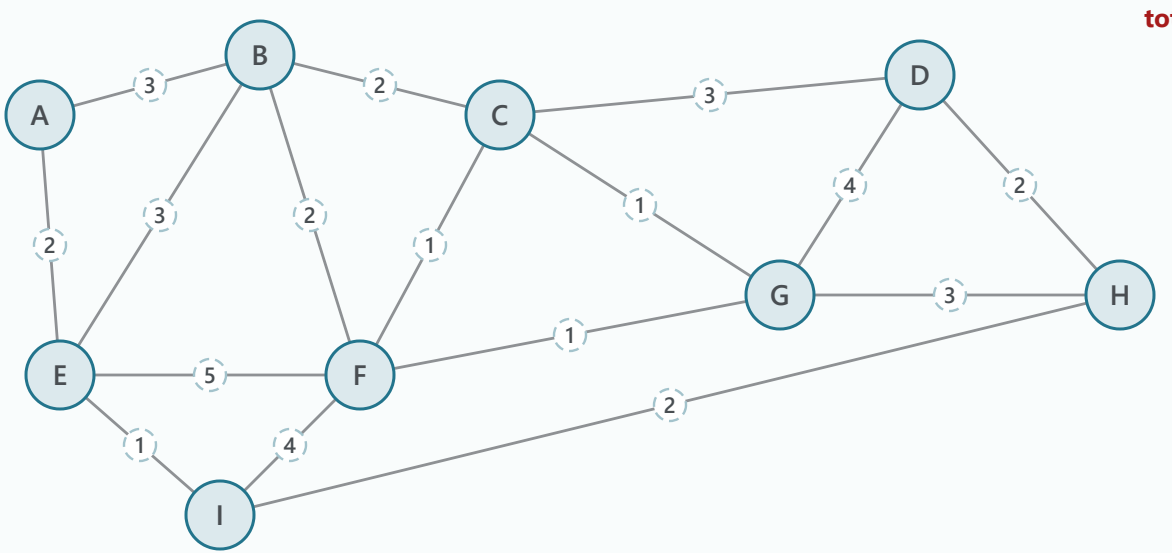


read-only page replicated: copies on CPU 0 and CPU 1; a write invalidates them

A read-only page is **replicated**: both CPUs read locally. On a write the other copies are invalidated; independent variables on the same page cause **false sharing**.

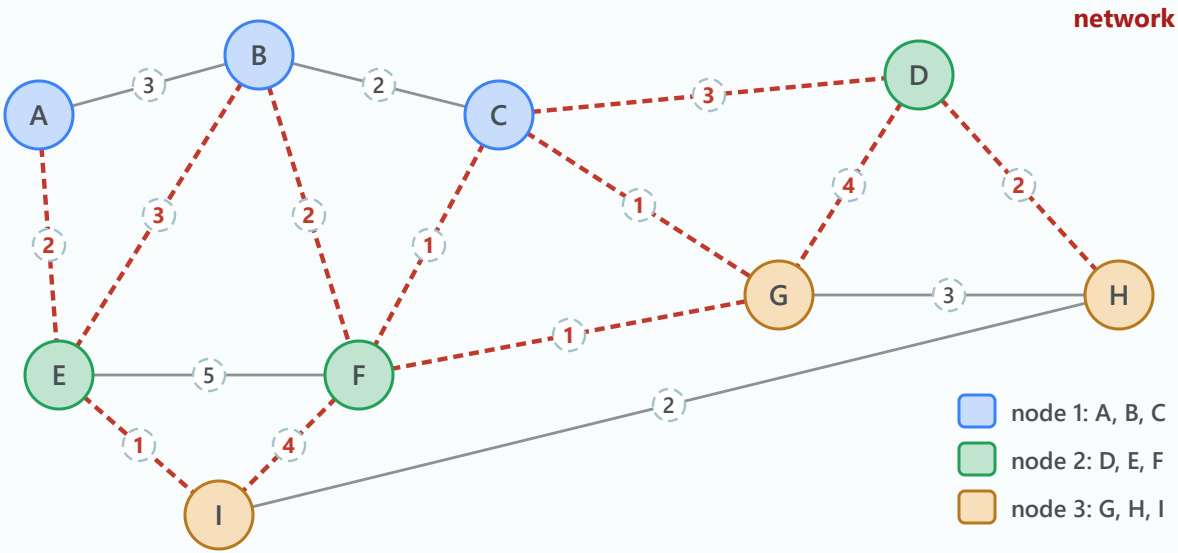
A 4 KiB page is expensive and coarse-grained on a network, so DSM gave way to message passing (MPI) and PGAS languages. Its modern counterpart is cross-machine memory pooling with CXL.

Graph-Theoretic Deterministic Allocation



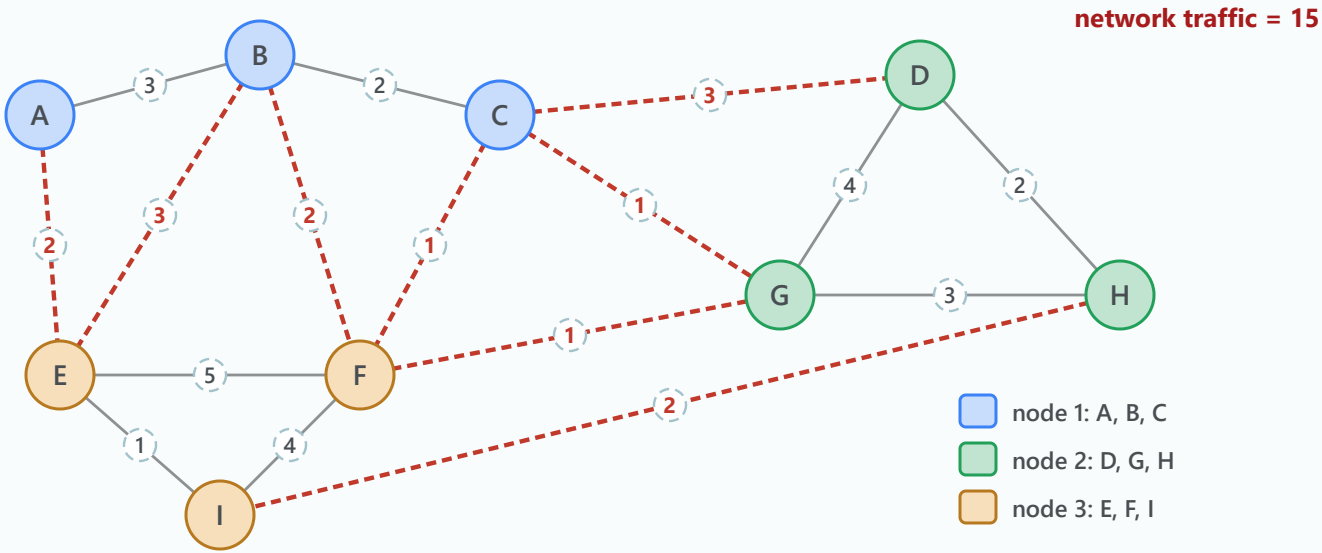
9 processes (vertices) and the message traffic between them (edge weights). Assign 3 processes to each of 3 machines; goal: minimize **network traffic**.

Graph-Theoretic Deterministic Allocation



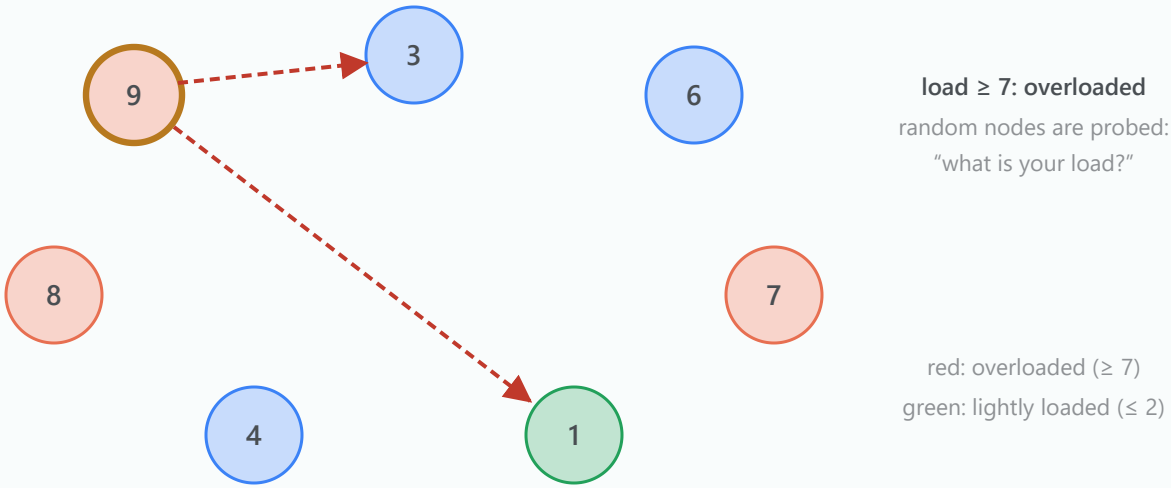
Alphabetical assignment: the cut (intermachine) edges total **24**.

Graph-Theoretic Deterministic Allocation



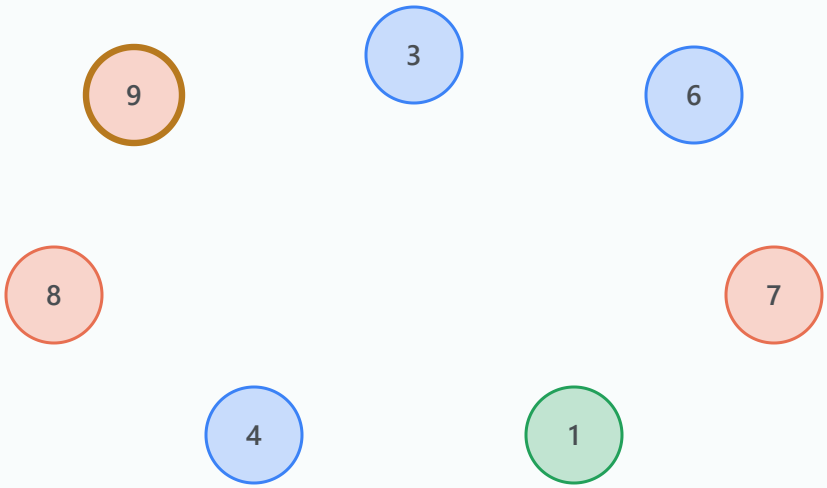
The best partition (all 280 balanced options tried): **15**. Chatty processes share a machine; the problem is NP-hard in general and is solved with heuristics.

Distributed Heuristic Load Balancing



Sender-initiated: when a new job arrives, a node over its threshold probes random nodes (Eager et al., 1986).

Distributed Heuristic Load Balancing



probed: 3 and 1

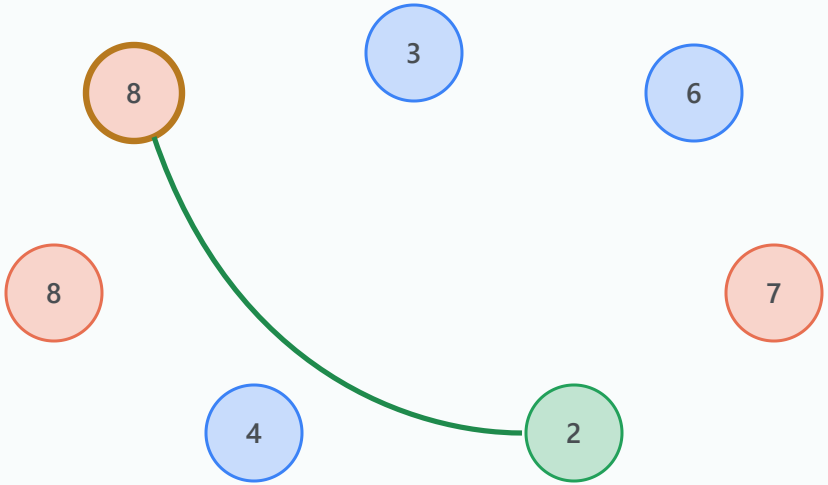
one below the threshold is found

red: overloaded (≥ 7)

green: lightly loaded (≤ 2)

Probing is limited (e.g. N = 3); if a node below the threshold turns up, the job goes there, otherwise it runs locally.

Distributed Heuristic Load Balancing

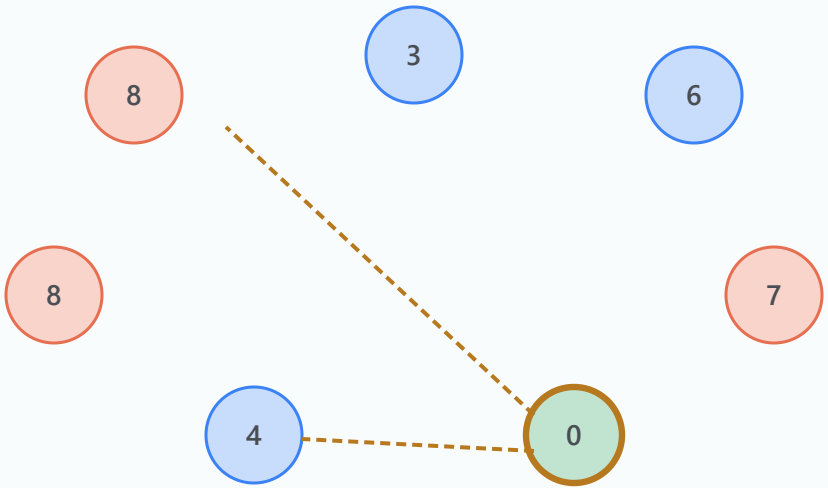


sender-initiated:
job sent to the node with load 1

red: overloaded (≥ 7)
green: lightly loaded (≤ 2)

The job has moved. Drawback: when the system is **fully loaded**, every probe is wasted and adds load.

Distributed Heuristic Load Balancing

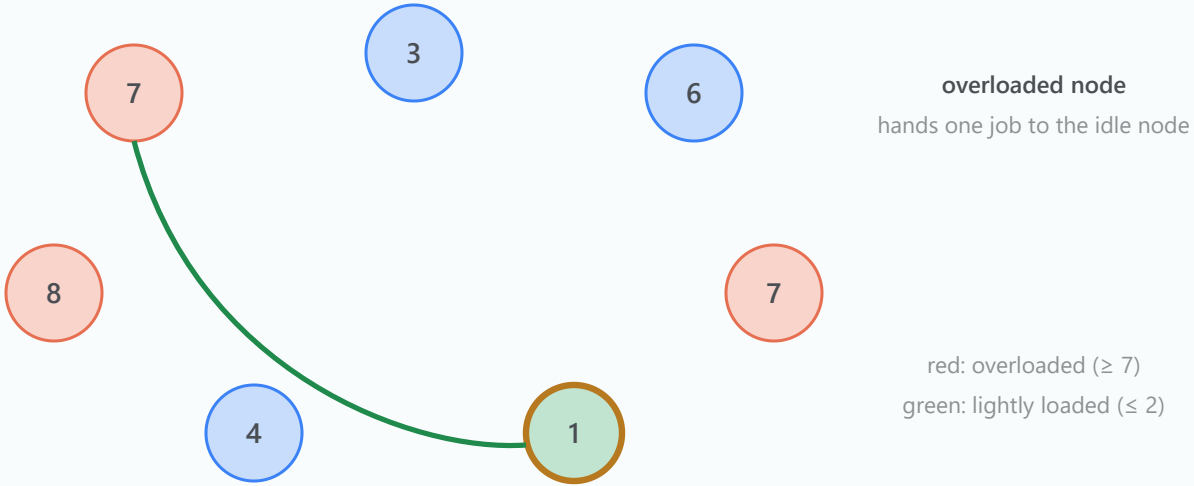


receiver-initiated:
an idle node looks for work

red: overloaded (≥ 7)
green: lightly loaded (≤ 2)

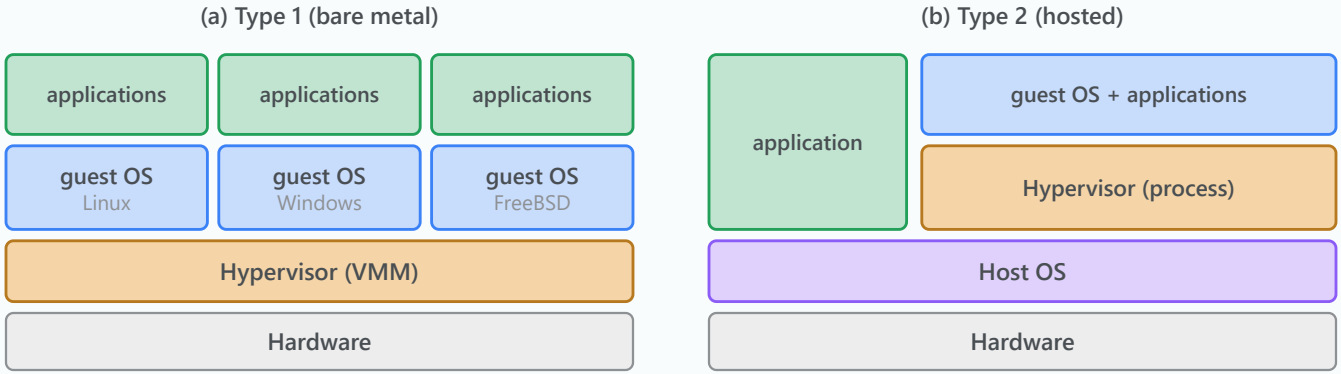
Receiver-initiated: a node that finishes a job and goes idle looks for work itself; the probing cost falls on the idle node.

Distributed Heuristic Load Balancing



A loaded node handed off a job. Works well under high load. Its modern form is **work stealing**: Cilk, Java ForkJoinPool, the Go and Tokio runtimes.

Hypervisors: Type 1 and Type 2

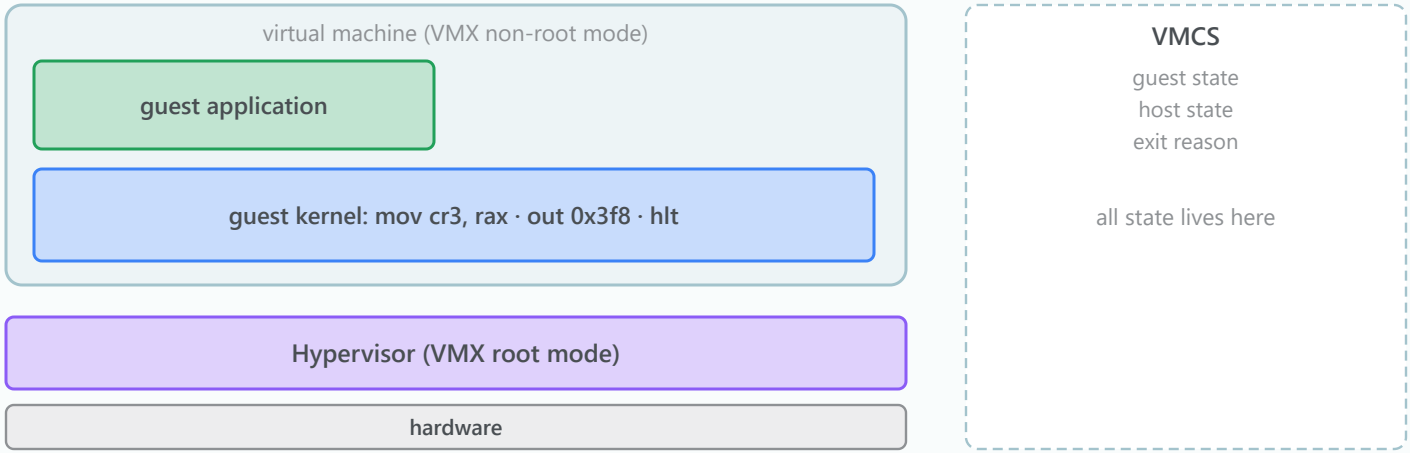


- **Type 1**: VMware ESXi, Xen (XCP-ng), Microsoft Hyper-V; Linux **KVM** makes the kernel a hypervisor (most clouds: AWS Nitro, Google). With Hyper-V on, Windows itself runs in a **root partition**.
- **Type 2**: VirtualBox, VMware Workstation/Fusion, Parallels, UTM. A **pure** type 2 interprets or binary-translates (QEMU without KVM); a **practical** one uses VT-x/AMD-V via a kernel module.

- Popek–Goldberg (1974): a machine is virtualizable if all **sensitive** instructions are privileged (trap in user mode).
- Classic x86 failed this test (e.g. `POPF` fails silently in user mode) → binary translation, then VT-x/AMD-V.

Uses: server consolidation, moving a running VM to another host (live migration), snapshots; the foundation of the cloud. In type 2, I/O goes through the host OS, so overhead is higher.

Trap and Emulate: VT-x / AMD-V

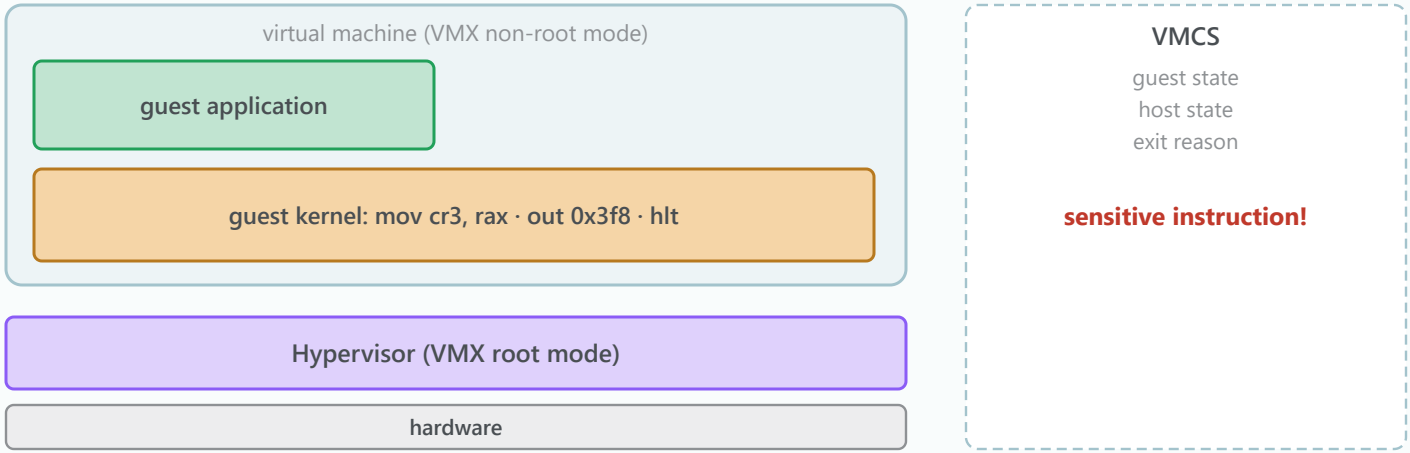


The guest OS believes it runs in kernel mode, but the hypervisor is really in control. Intel VT-x / AMD-V (2005–06) add a new mode: **root** (hypervisor) and **non-root** (guest).

Step 1 / 4

Before hardware support, VMware (1999) replaced sensitive instructions at run time through **binary translation**. Only the guest OS's *sensitive* instructions trap; the rest run directly on the hardware.

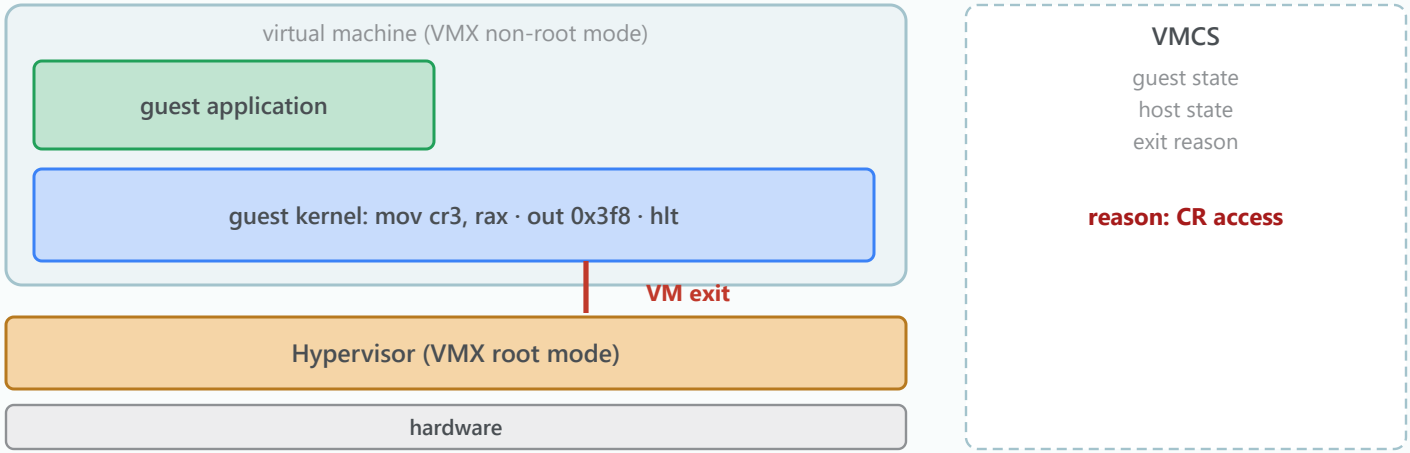
Trap and Emulate: VT-x / AMD-V



The guest kernel executes a sensitive instruction: changing the page table (CR3), an I/O port, `HLT` …

Before hardware support, VMware (1999) replaced sensitive instructions at run time through **binary translation**. Only the guest OS's *sensitive* instructions trap; the rest run directly on the hardware.

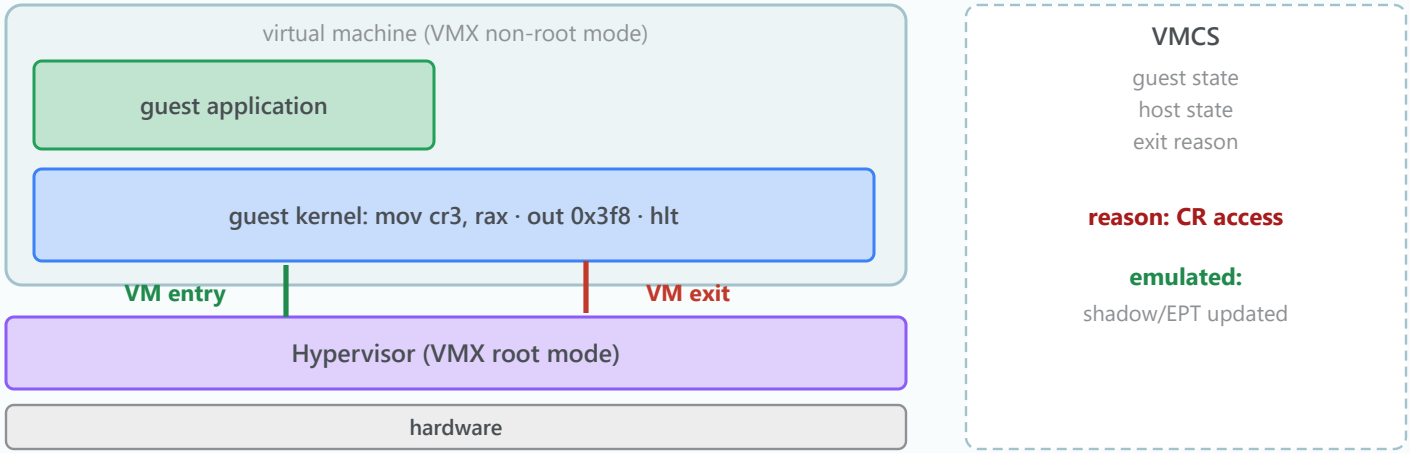
Trap and Emulate: VT-x / AMD-V



The hardware does not execute it: a **VM exit** traps to the hypervisor, and the reason is written to the VMCS.

Before hardware support, VMware (1999) replaced sensitive instructions at run time through **binary translation**. Only the guest OS's *sensitive* instructions trap; the rest run directly on the hardware.

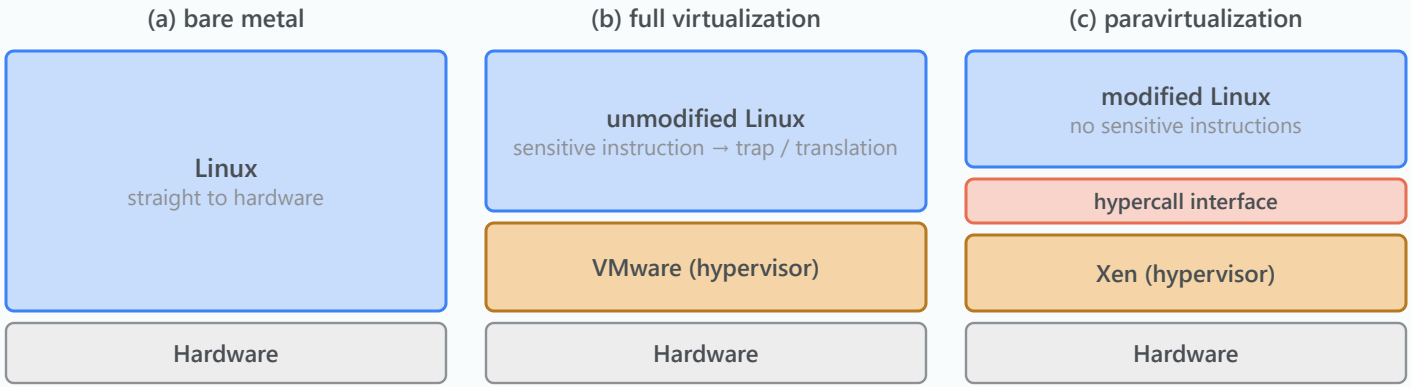
Trap and Emulate: VT-x / AMD-V



The hypervisor **emulates** the instruction's effect, updates the virtual hardware state and returns to the guest with a **VM entry**. An exit costs ~1 µs; frequent exits hurt performance.

Before hardware support, VMware (1999) replaced sensitive instructions at run time through **binary translation**. Only the guest OS's *sensitive* instructions trap; the rest run directly on the hardware.

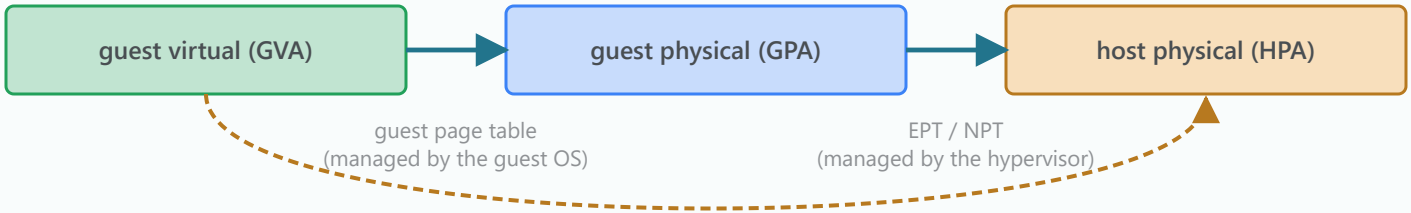
Paravirtualization



- The guest OS is **modified**: instead of sensitive instructions it makes **hypercalls** straight to the hypervisor — the one-layer-down analogue of a system call.
- No trap-and-emulate cost, so it is faster; but it needs an OS whose source can be changed (Xen 2003).
- VMI (Virtual Machine Interface) was proposed so one kernel could run on bare metal and on different hypervisors; it was removed from Linux in 2011. Paravirtualization lives on in **drivers** today: `virtio` disk/network, paravirtual clocks, Hyper-V enlightenments.

In today's clouds, CPU and memory are fully virtualized with hardware support; I/O uses virtio or direct device assignment with SR-IOV.

Memory Virtualization: Shadow Tables and EPT

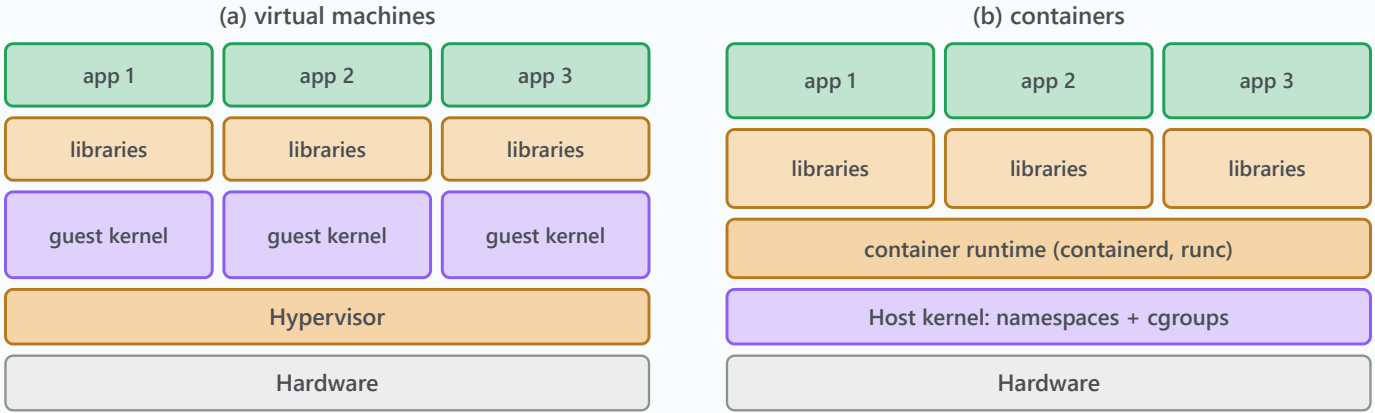


shadow page table: GVA → HPA directly (kept current in software)

2D walk on a TLB miss: 4 levels → $(4+1)^2 - 1 = 24$ memory accesses · 5 levels → 35

- **Shadow page tables**: the hypervisor makes the guest's tables read-only, traps every change and updates its GVA → HPA copy. Lots of VM exits.
- **EPT** (Intel, 2008) / **NPT** (AMD): the hardware walks both tables together; the guest changes its own table without traps.
- TLB misses get more expensive → huge pages (2 MiB / 1 GiB) and VM-tagged TLBs (VPID/ASID) are used.
- Overcommit: a **balloon driver** reclaims memory from the guest; **KSM** merges identical pages (copy on write).

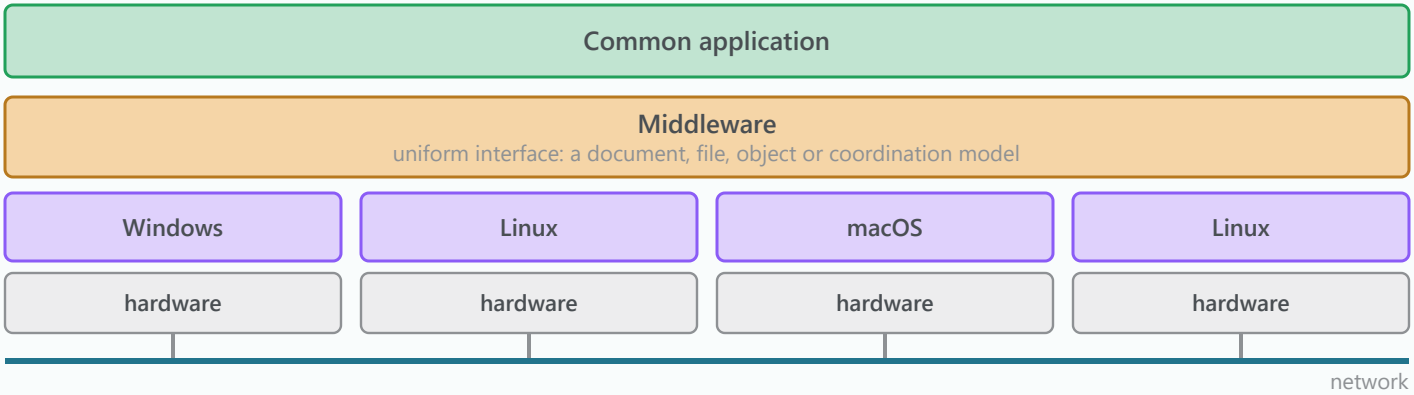
Containers and Virtual Machines



	Virtual machine	Container
Isolation boundary	hardware level, separate kernel	same kernel: namespaces, cgroups, seccomp
Startup / overhead	seconds, GBs of memory	milliseconds, MBs of memory
Different OS	yes (Windows on Linux)	no (same kernel family)
Example	KVM, Hyper-V, ESXi	Docker, Podman, Kubernetes pods

In between: Firecracker micro-VMs that boot in milliseconds (AWS Lambda), gVisor and Kata Containers combine container convenience with VM isolation. WSL 2 also runs a real Linux kernel in a lightweight Hyper-V VM; Linux containers on Windows run in that VM.

Distributed Systems and Middleware

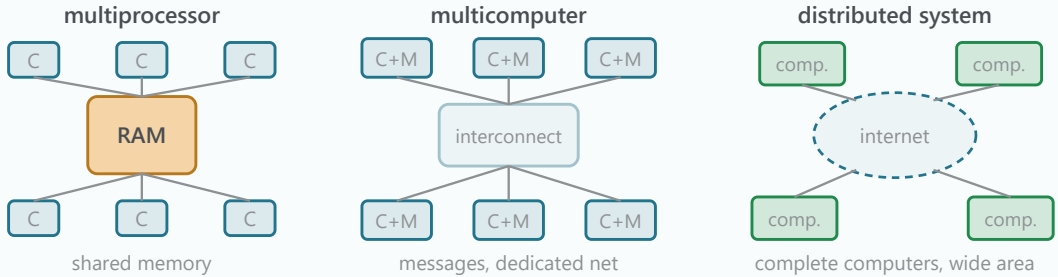


- **Distributed system**: autonomous computers communicating over a network; each manages its own OS, memory and resources, yet the whole tries to look like one coherent system to the user.
- Nodes may run different hardware and OSes, belong to different organizations and span the globe. The common ground is the **middleware** layer.
- Four paradigms: **document** (the Web), **file** (NFS, SMB), **object** (CORBA, RMI, gRPC), **coordination** (Linda, publish-subscribe).

True distributed operating systems (Amoeba, Plan 9) stayed in research; today each node runs its own OS and distribution is handled by middleware and cluster managers (Kubernetes).

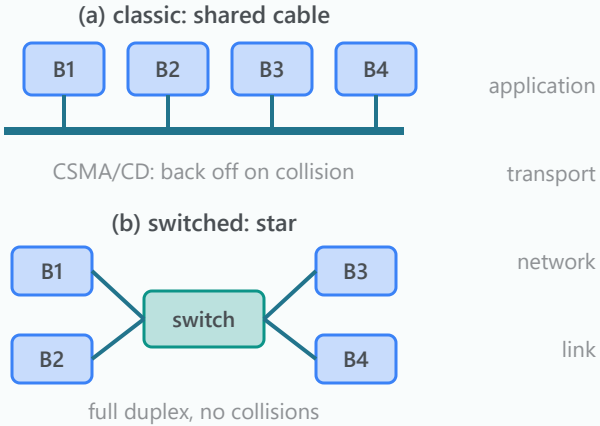
Comparing Three Kinds of Systems

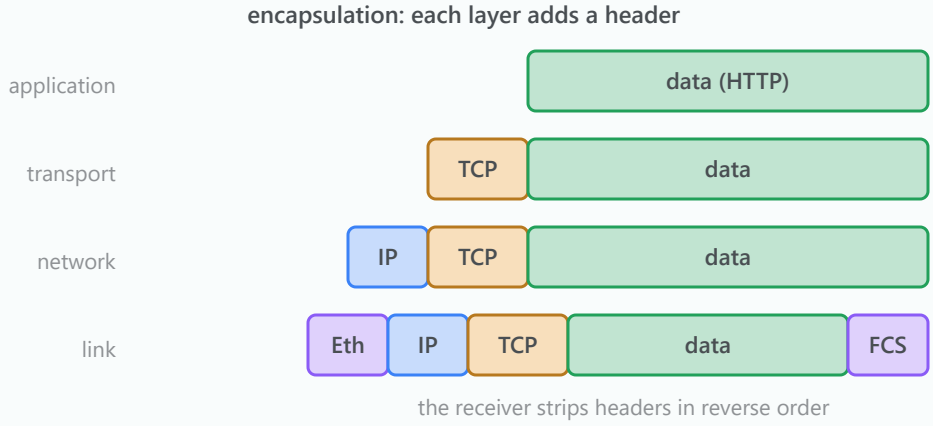
Item	Multiprocessor	Multicomputer	Distributed system
Node configuration	CPU	CPU, RAM, net interface	complete computer
Peripherals	all shared	shared (maybe except disk)	full set per node
Location	same rack	same room	possibly worldwide
Internode communication	shared RAM	dedicated interconnect	traditional network
Operating system	one, shared	many, the same	many, possibly different
File system	one, shared	one, shared	each node has its own
Administration	one organization	one organization	many organizations



Tanenbaum (MOS 4e, Fig. 8-29). In a distributed system peripherals are not shared: each node has **its own full set**. In a multicomputer, the disk is not always shared either.

Network Hardware and Encapsulation





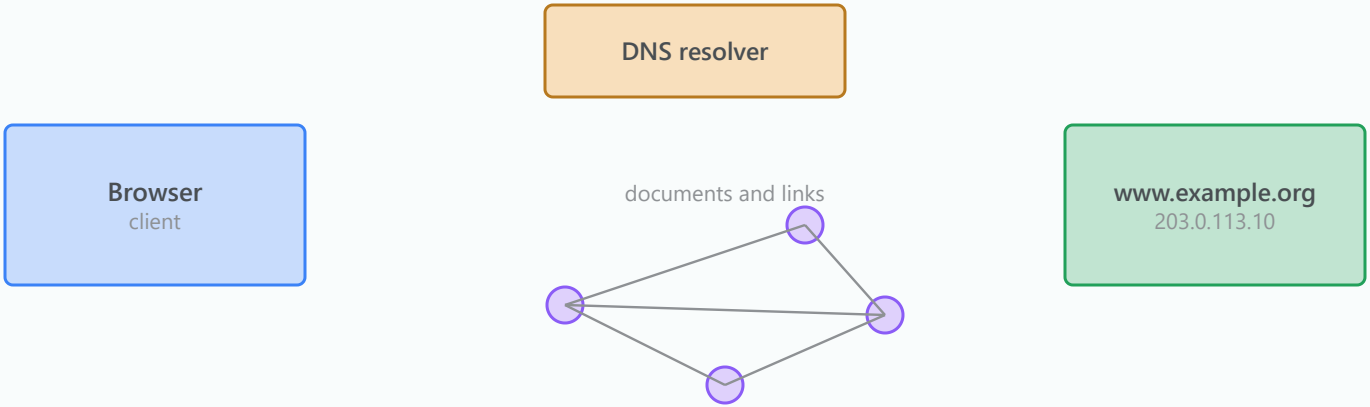
- Classic Ethernet (1976) shared one cable; today every computer connects to a **switch**, with full-duplex links of 1–400 Gb/s.
- Routers join LANs to each other, and local networks connect to ISPs and the **Internet backbone** (high-speed fiber, exchange points).

Network Protocols and Service Types

Protocol	Layer	Service	Note
IP (v4/v6)	network	connectionless, unreliable	addressing and routing
TCP	transport	connection-oriented, reliable byte stream	acks, retransmission, flow and congestion control
UDP	transport	connectionless, unreliable datagrams	has a checksum, no retransmission. DNS, games, voice
QUIC	transport (over UDP)	connection-oriented, reliable, encrypted	used by HTTP/3 (2022)
DNS	application	name → IP address	hierarchical, distributed, cached
HTTP/HTTPS	application	Web documents	HTTPS = HTTP + TLS, port 443
SMTP	application	mail transfer	server to server; reading via IMAP
FTP → SFTP	application	file transfer	FTP is unencrypted; replaced by SFTP/HTTPS
SNMP	application	network device monitoring	v3 adds authentication and encryption

Reliable service: guaranteed delivery and order, with acknowledgments (ACK). Connection-oriented: set up, then data, then teardown; connectionless service sends each packet independently with no setup cost. UDP **detects** a corrupted packet with its checksum but neither fixes nor resends it.

Document-Based Middleware: The Web

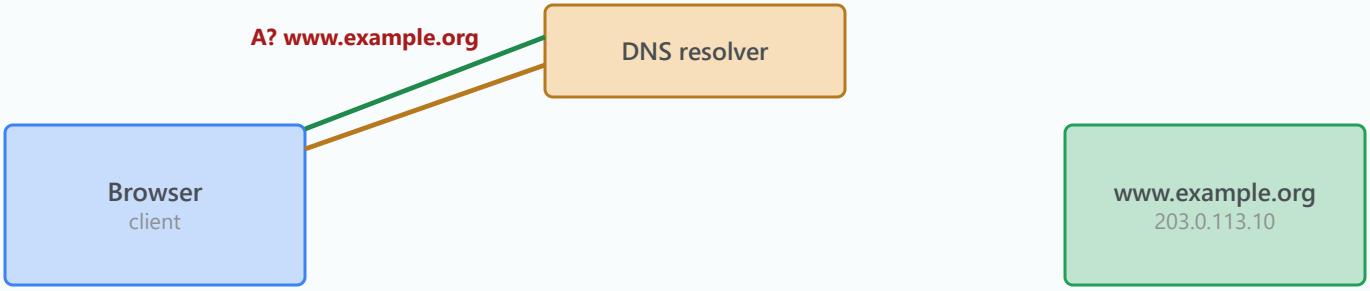


The user types `www.example.org/index.html`. The Web is a huge **directed graph** of linked documents.

Step 1 / 5

Port 80 and a new TCP connection per object belong to the HTTP/1.0 era; today the defaults are HTTPS (443), persistent connections and HTTP/2–3 multiplexing.

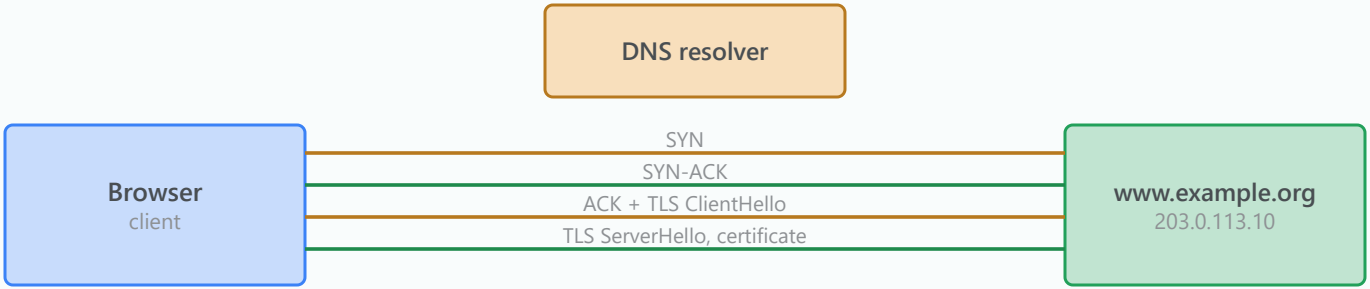
Document-Based Middleware: The Web



1–2: The browser (through the OS resolver) asks **DNS** for www.example.org's address; answer: 203.0.113.10 (cached).

Port 80 and a new TCP connection per object belong to the HTTP/1.0 era; today the defaults are HTTPS (443), persistent connections and HTTP/2–3 multiplexing.

Document-Based Middleware: The Web



3: A TCP connection to the server's port **443** and the **TLS handshake** (one step in HTTP/3, with QUIC).

Step 3 / 5

Port 80 and a new TCP connection per object belong to the HTTP/1.0 era; today the defaults are HTTPS (443), persistent connections and HTTP/2–3 multiplexing.

Document-Based Middleware: The Web



4–5: A `GET /index.html` request; the server returns the HTML and the browser starts rendering text.

Port 80 and a new TCP connection per object belong to the HTTP/1.0 era; today the defaults are HTTPS (443), persistent connections and HTTP/2–3 multiplexing.

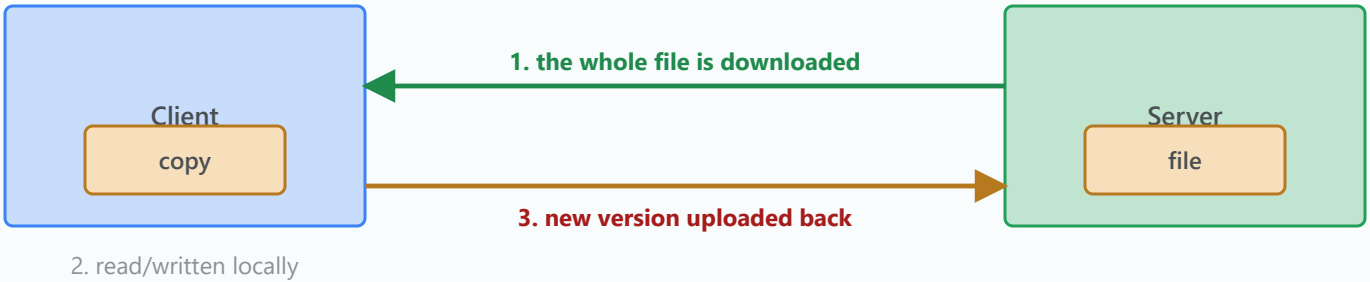
Document-Based Middleware: The Web



6: Images, CSS and scripts in the HTML are fetched over **the same connection**, as parallel streams in HTTP/2–3; the connection stays open for a while.

Port 80 and a new TCP connection per object belong to the HTTP/1.0 era; today the defaults are HTTPS (443), persistent connections and HTTP/2–3 multiplexing.

File-System-Based Middleware

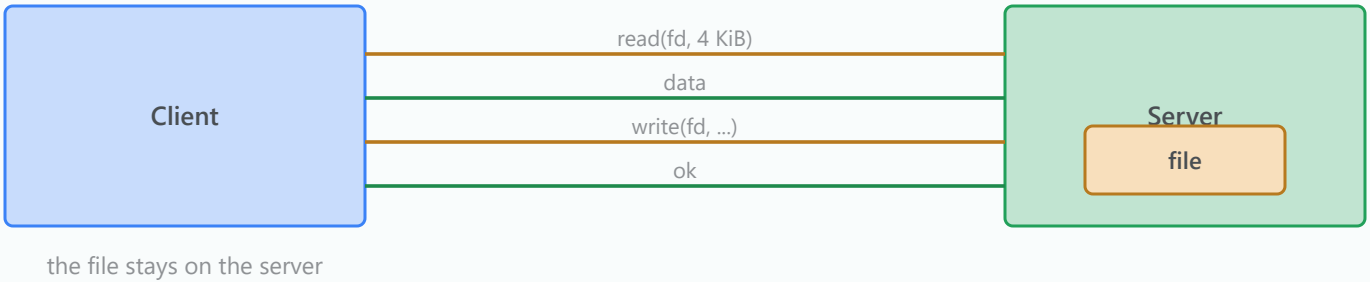


Upload/download model: the whole file is copied to the client, used and sent back (FTP-like; early AFS). Simple, but costly for large files.

Step 1 / 5

Naming: `machine:/path` (not location transparent) → mounting remote systems into the local tree → one namespace identical on all machines (location independence). Today: NFSv4, SMB 3, object storage (S3).

File-System-Based Middleware



Remote access model: the file stays on the server; the client sends `read` / `write` requests over the network (NFS, SMB).

Naming: `machine:/path` (not location transparent) → mounting remote systems into the local tree → one namespace identical on all machines (location independence). Today: NFSv4, SMB 3, object storage (S3).

File-System-Based Middleware

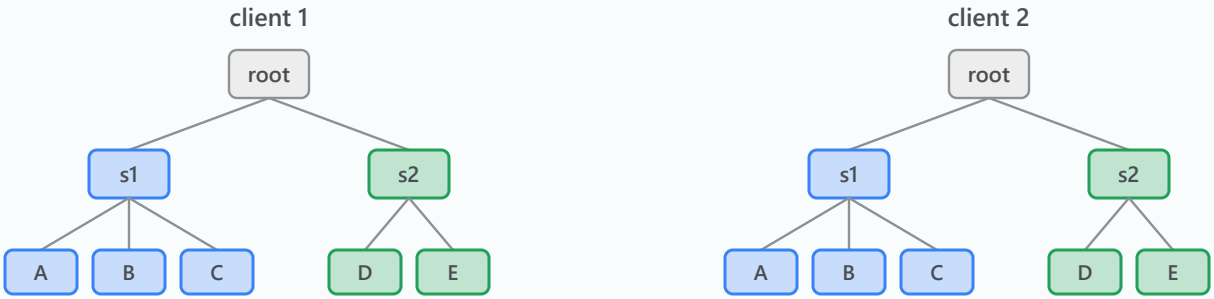


square: directory · circle: file

Two file servers, each with its own directory tree.

Naming: `machine:/path` (not location transparent) → mounting remote systems into the local tree → one namespace identical on all machines (location independence). Today: NFSv4, SMB 3, object storage (S3).

File-System-Based Middleware



same namespace: /s1/A is the same file on every client

All clients mount the servers in the same place → **a single shared namespace**.

Naming: `machine:/path` (not location transparent) → mounting remote systems into the local tree → one namespace identical on all machines (location independence). Today: NFSv4, SMB 3, object storage (S3).

File-System-Based Middleware

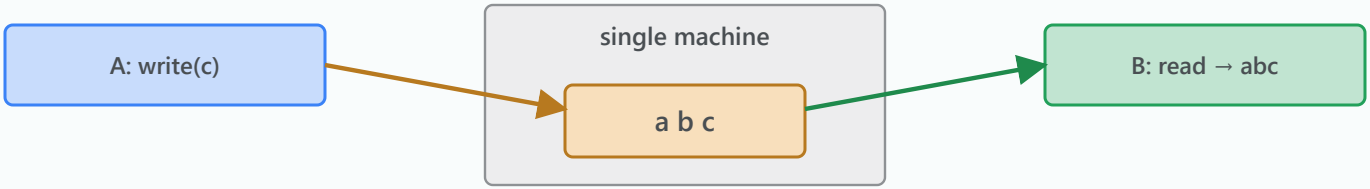


client 2 has no /s1, while /s2/D is the same on both

If clients mount different things, each sees a different view; the same path may name different files on different clients.

Naming: `machine:/path` (not location transparent) → mounting remote systems into the local tree → one namespace identical on all machines (location independence). Today: NFSv4, SMB 3, object storage (S3).

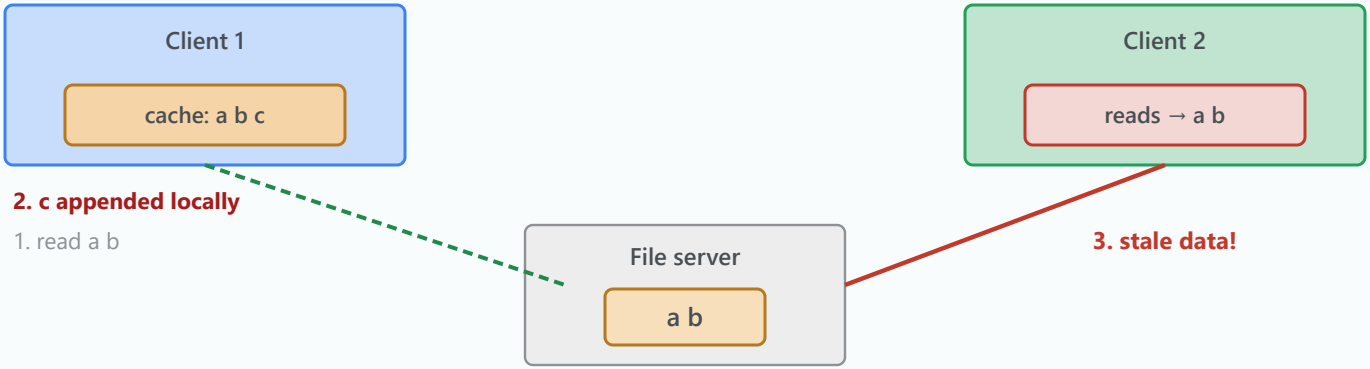
Semantics of File Sharing



every read sees the latest write: UNIX semantics

On one processor operations are **sequential**: A appends “c”, then B reads and sees “abc” (sequential consistency).

Semantics of File Sharing



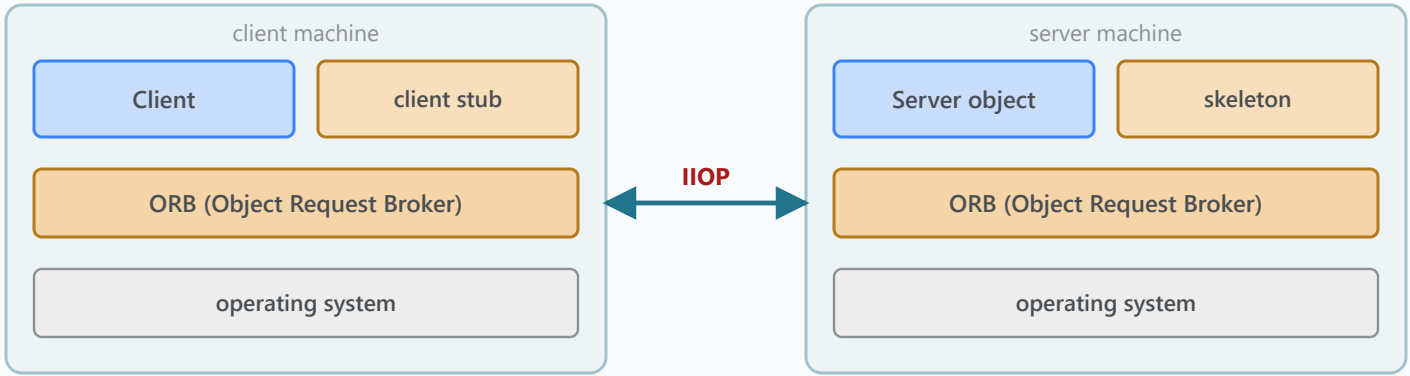
In a distributed system client 1 changes the file in its cache; the server does not know yet. Client 2 reads from the server and sees the **stale** “ab”.

Semantics of File Sharing

Approach	Rule	Example
UNIX semantics	every write is visible to all at once	single machine; remote access without caching
Session semantics	changes become visible when the file is closed	AFS; NFS close-to-open consistency
Immutable files	files cannot be written; a new one is created	object storage, Git objects
Transactions	all or nothing, serializable	databases

Each solution trades consistency against performance.

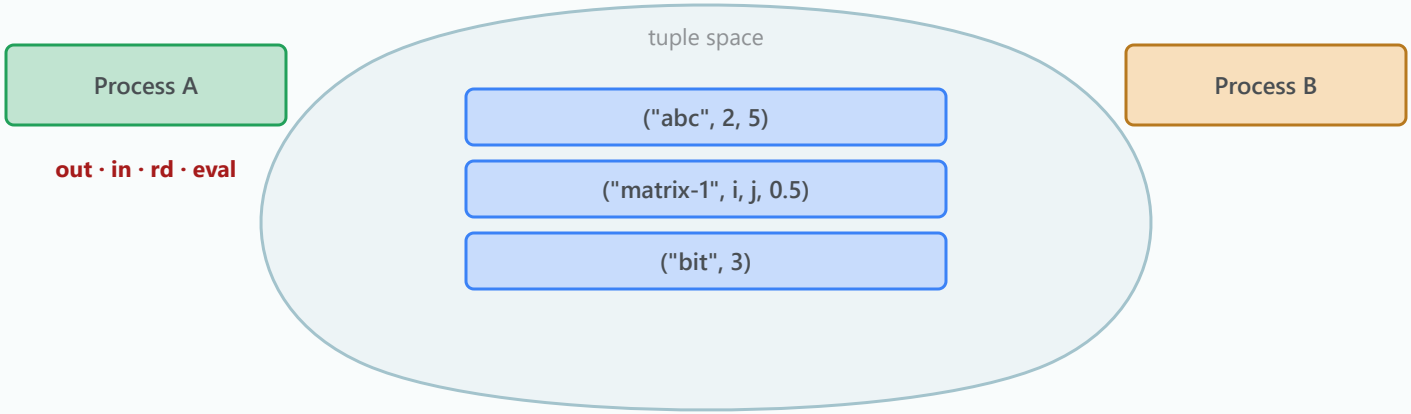
Object-Based Middleware: CORBA



- **IDL**: interfaces are defined independent of language; the compiler generates the client stub and the server skeleton.
- **ORB**: locates the object, marshals the call and sends it over IIOP: **network transparency**.
- **SII** (static invocation): through stubs generated from IDL at compile time; fast and type-checked.
- **DII** (dynamic invocation): the interface is discovered at run time (interface repository).

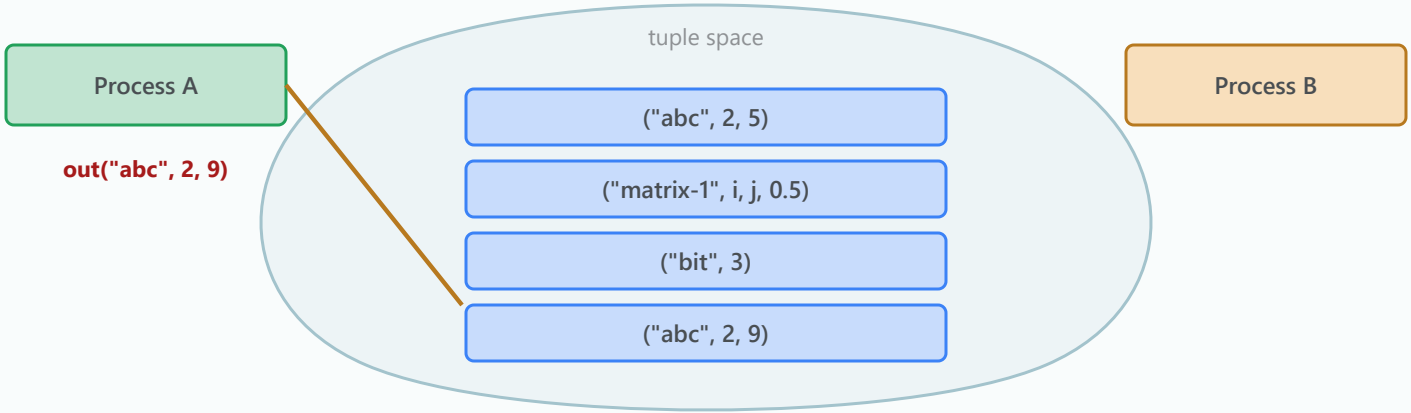
SII and DII are separate mechanisms; SII is not a compiled form of DII. CORBA (OMG, 1991) survives in legacy systems; new systems prefer gRPC/Protobuf, REST and Java RMI.

Coordination-Based Middleware: Linda



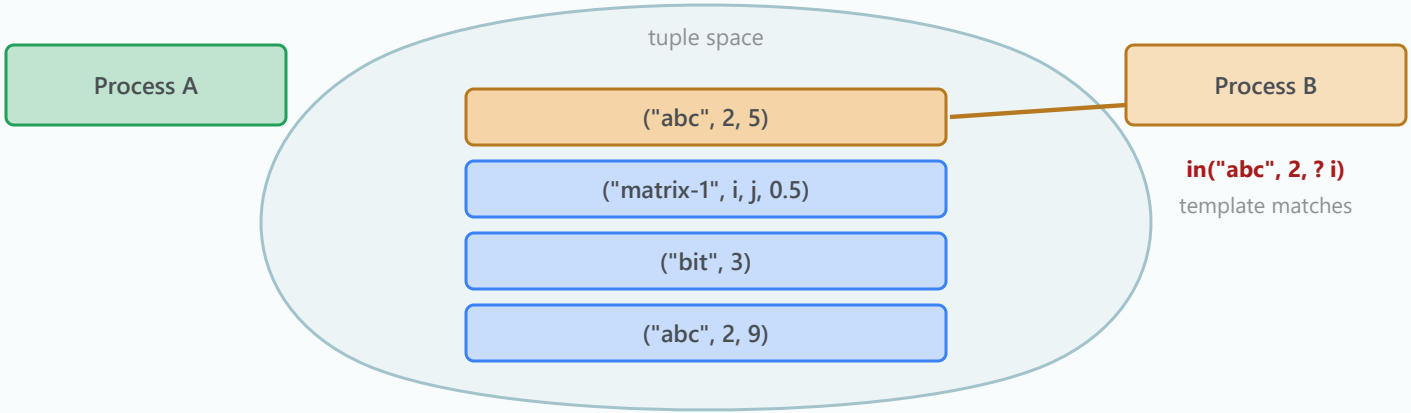
Linda (Gelernter, 1985): processes do not know each other; they communicate through a shared, persistent **tuple space**. A tuple is a record of typed fields.

Coordination-Based Middleware: Linda



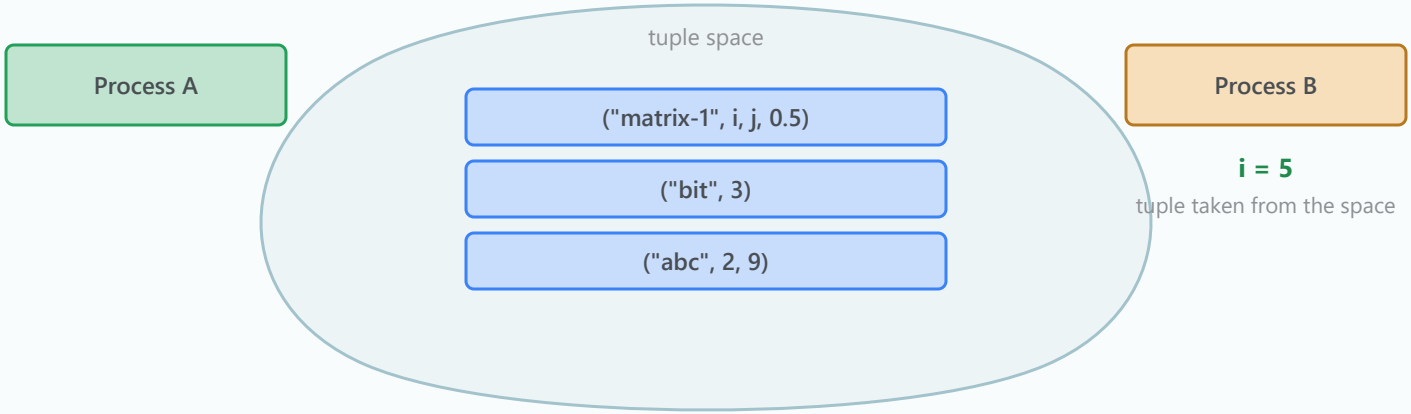
`out` puts a tuple into the space (nonblocking).

Coordination-Based Middleware: Linda



`in` searches with a **template**: same number of fields, matching types field by field, equal constant fields. `? i` is a formal parameter. With several matches, one is chosen arbitrarily.

Coordination-Based Middleware: Linda



The matching tuple is **removed** from the space and `i = 5` is assigned. With no match, `in` blocks. `rd` reads without removing; `eval` starts a new process. Synchronization comes for free.

Linda in(): Code Trace

```
▶ Tuple *in_find(Template *s) {
▶   for (Tuple *d = space; d; d = d->next) {
3     if (d->n != s->n) continue;
4     int ok = 1;
5     for (int k = 0; ok && k < s->n; k++)
6       if (d->type[k] != s->type[k] ||
7           (s->fixed[k] && !equal(d, s, k)))
8         ok = 0;          /* field mismatch */
9     if (ok) { remove(d); return d; }
10  }
11  return NULL;           /* block */
12 }
```

template: | "abc" | 2 | ? int i |

tuple space:

d1	"abc"	2	
d2	"xyz"	2	7
d3	"abc"	1	4
d4	"abc"	2	5
d5	"abc"	2	9

The space holds 5 tuples. A process calls `in("abc", 2, ? int i)` : 3 fields, the first two constant, the third a formal parameter of type int.

Step 1 / 11

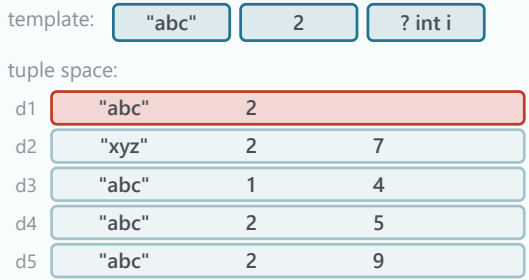
VARIABLES

`d = d1` `k = –` `ok = –` `i = –`

OUTPUT

Linda in(): Code Trace

```
1  Tuple *in_find(Template *s) {
2    for (Tuple *d = space; d; d = d->next) {
▶      if (d->n != s->n) continue;
4      int ok = 1;
5      for (int k = 0; ok && k < s->n; k++)
6        if (d->type[k] != s->type[k] ||
7            (s->fixed[k] && !equal(d, s, k)))
8          ok = 0;          /* field mismatch */
9      if (ok) { remove(d); return d; }
10   }
11   return NULL;           /* block */
12 }
```



d1: 2 fields, but the template has 3 → skipped at once (`continue`).

Step 2 / 11

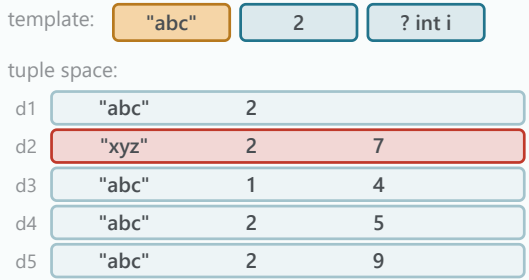
VARIABLES

`d = d1` `k = –` `ok = –` `i = –`

OUTPUT

Linda in(): Code Trace

```
1  Tuple *in_find(Template *s) {
2    for (Tuple *d = space; d; d = d->next) {
3      if (d->n != s->n) continue;
4      int ok = 1;
▶      for (int k = 0; ok && k < s->n; k++)
▶        if (d->type[k] != s->type[k] ||
▶            (s->fixed[k] && !equal(d, s, k)))
▶          ok = 0;          /* field mismatch */
9      if (ok) { remove(d); return d; }
10   }
11   return NULL;           /* block */
12 }
```



d2, field 0: "xyz" ≠ "abc" → ok = 0, move to the next tuple.

Step 3 / 11

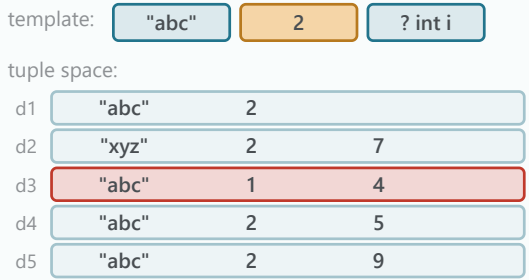
VARIABLES

`d = d2` `k = 0` `ok = 0` `i = –`

OUTPUT

Linda in(): Code Trace

```
1  Tuple *in_find(Template *s) {
2    for (Tuple *d = space; d; d = d->next) {
3      if (d->n != s->n) continue;
4      int ok = 1;
▶      for (int k = 0; ok && k < s->n; k++)
▶        if (d->type[k] != s->type[k] ||
▶            (s->fixed[k] && !equal(d, s, k)))
▶          ok = 0;          /* field mismatch */
9      if (ok) { remove(d); return d; }
10   }
11   return NULL;           /* block */
12 }
```



d3, field 1: 1 ≠ 2 → ok = 0, move to the next tuple.

Step 4 / 11

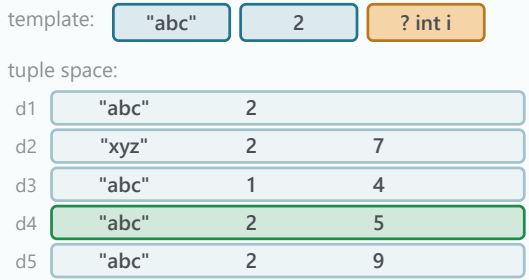
VARIABLES

`d = d3` `k = 1` `ok = 0` `i = –`

OUTPUT

Linda in(): Code Trace

```
1  Tuple *in_find(Template *s) {
2    for (Tuple *d = space; d; d = d->next) {
3      if (d->n != s->n) continue;
4      int ok = 1;
▶      for (int k = 0; ok && k < s->n; k++)
▶        if (d->type[k] != s->type[k] ||
▶            (s->fixed[k] && !equal(d, s, k)))
8          ok = 0;          /* field mismatch */
▶      if (ok) { remove(d); return d; }
10   }
11   return NULL;           /* block */
12 }
```



d4: field count and types fit, constants are equal; the formal field i gets **5**. The tuple is **removed** from the space.

Step 5 / 11

VARIABLES

`d = d4` `k = 3` `ok = 1` `i = 5`

OUTPUT

```
1. in: i = 5
```

Linda in(): Code Trace

```
1  Tuple *in_find(Template *s) {
2    for (Tuple *d = space; d; d = d->next) {
3      if (d->n != s->n) continue;
4      int ok = 1;
5      for (int k = 0; ok && k < s->n; k++)
6        if (d->type[k] != s->type[k] ||
7            (s->fixed[k] && !equal(d, s, k)))
8          ok = 0;          /* field mismatch */
▶      if (ok) { remove(d); return d; }
10   }
11   return NULL;           /* block */
12 }
```

template: | "abc" | 2 | ? int i |

tuple space:

d1	"abc"	2	
d2	"xyz"	2	7
d3	"abc"	1	4
d5	"abc"	2	9

d4 is gone from the space; d5 would also have matched, but the first one found was taken (Linda does not define which is chosen). The same call is made again.

Step 6 / 11

VARIABLES

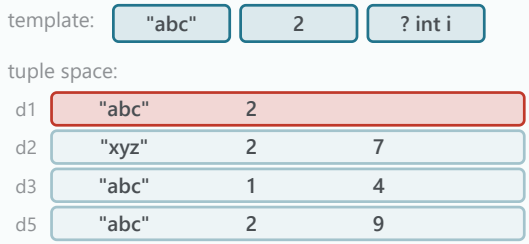
`d = d4` `k = 3` `ok = 1` `i = 5`

OUTPUT

```
1. in: i = 5
```

Linda in(): Code Trace

```
1  Tuple *in_find(Template *s) {
2    for (Tuple *d = space; d; d = d->next) {
▶      if (d->n != s->n) continue;
4      int ok = 1;
5      for (int k = 0; ok && k < s->n; k++)
6        if (d->type[k] != s->type[k] ||
7            (s->fixed[k] && !equal(d, s, k)))
8          ok = 0;          /* field mismatch */
9      if (ok) { remove(d); return d; }
10   }
11   return NULL;           /* block */
12 }
```



d1: 2 fields, but the template has 3 → skipped at once (`continue`).

Step 7 / 11

VARIABLES

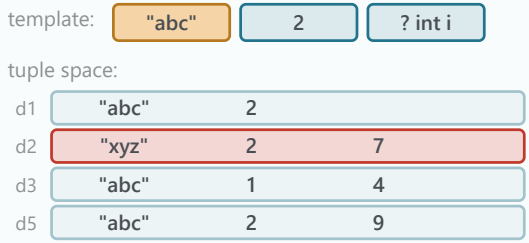
`d = d1` `k = –` `ok = –` `i = –`

OUTPUT

```
1. in: i = 5
```

Linda in(): Code Trace

```
1  Tuple *in_find(Template *s) {
2    for (Tuple *d = space; d; d = d->next) {
3      if (d->n != s->n) continue;
4      int ok = 1;
▶      for (int k = 0; ok && k < s->n; k++)
▶        if (d->type[k] != s->type[k] ||
▶            (s->fixed[k] && !equal(d, s, k)))
▶          ok = 0;          /* field mismatch */
9      if (ok) { remove(d); return d; }
10   }
11   return NULL;           /* block */
12 }
```



d2, field 0: "xyz" ≠ "abc" → ok = 0, move to the next tuple.

Step 8 / 11

VARIABLES

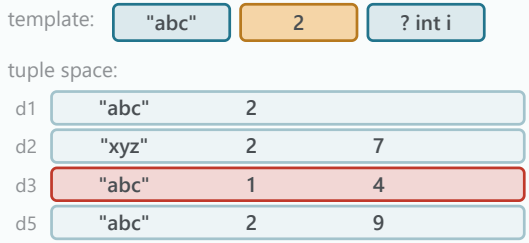
`d = d2` `k = 0` `ok = 0` `i = –`

OUTPUT

```
1. in: i = 5
```

Linda in(): Code Trace

```
1  Tuple *in_find(Template *s) {
2    for (Tuple *d = space; d; d = d->next) {
3      if (d->n != s->n) continue;
4      int ok = 1;
▶      for (int k = 0; ok && k < s->n; k++)
▶        if (d->type[k] != s->type[k] ||
▶            (s->fixed[k] && !equal(d, s, k)))
▶          ok = 0;          /* field mismatch */
9      if (ok) { remove(d); return d; }
10   }
11   return NULL;           /* block */
12 }
```



d3, field 1: 1 ≠ 2 → ok = 0, move to the next tuple.

Step 9 / 11

VARIABLES

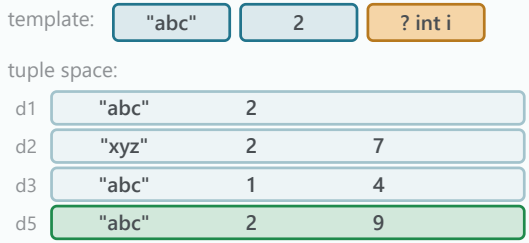
`d = d3` `k = 1` `ok = 0` `i = –`

OUTPUT

```
1. in: i = 5
```

Linda in(): Code Trace

```
1  Tuple *in_find(Template *s) {
2    for (Tuple *d = space; d; d = d->next) {
3      if (d->n != s->n) continue;
4      int ok = 1;
▶      for (int k = 0; ok && k < s->n; k++)
▶        if (d->type[k] != s->type[k] ||
▶            (s->fixed[k] && !equal(d, s, k)))
8          ok = 0;          /* field mismatch */
▶      if (ok) { remove(d); return d; }
10   }
11   return NULL;           /* block */
12 }
```



d5: field count and types fit, constants are equal; the formal field i gets **9**. The tuple is **removed** from the space.

Step 10 / 11

VARIABLES

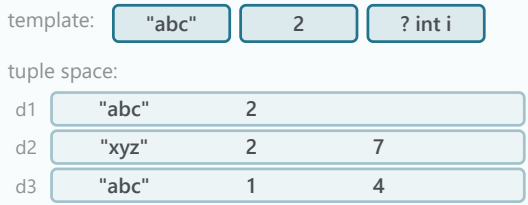
`d = d5` `k = 3` `ok = 1` `i = 9`

OUTPUT

```
1. in: i = 5
2. in: i = 9
```

Linda in(): Code Trace

```
1  Tuple *in_find(Template *s) {
2    for (Tuple *d = space; d; d = d->next) {
3      if (d->n != s->n) continue;
4      int ok = 1;
5      for (int k = 0; ok && k < s->n; k++)
6        if (d->type[k] != s->type[k] ||
7            (s->fixed[k] && !equal(d, s, k)))
8          ok = 0;          /* field mismatch */
9      if (ok) { remove(d); return d; }
10   }
▶    return NULL;           /* block */
12 }
```



A third `in("abc", 2, ? int i)` finds no match and returns NULL: the process **blocks** until someone `out` s a matching tuple.

Step 11 / 11

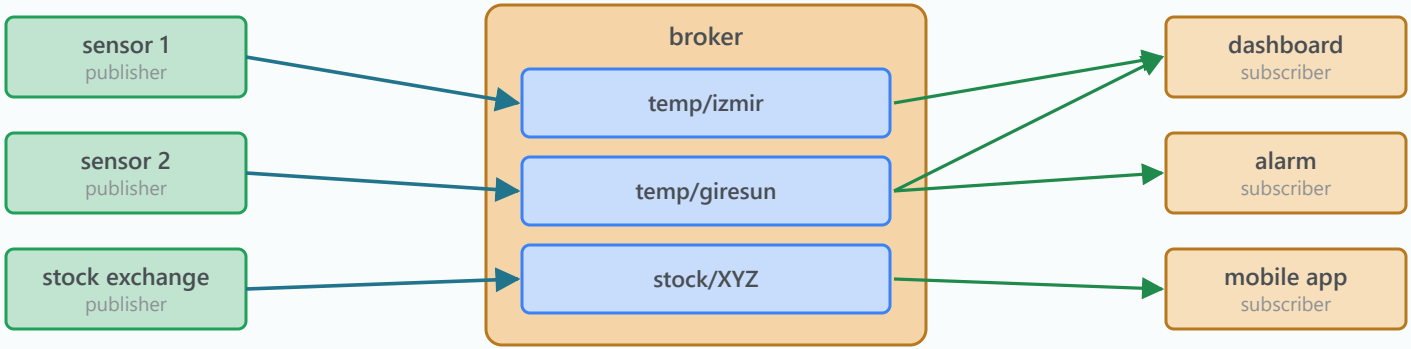
VARIABLES

`d = –` `k = –` `ok = –` `i = 9`

OUTPUT

```
1. in: i = 5
2. in: i = 9
```

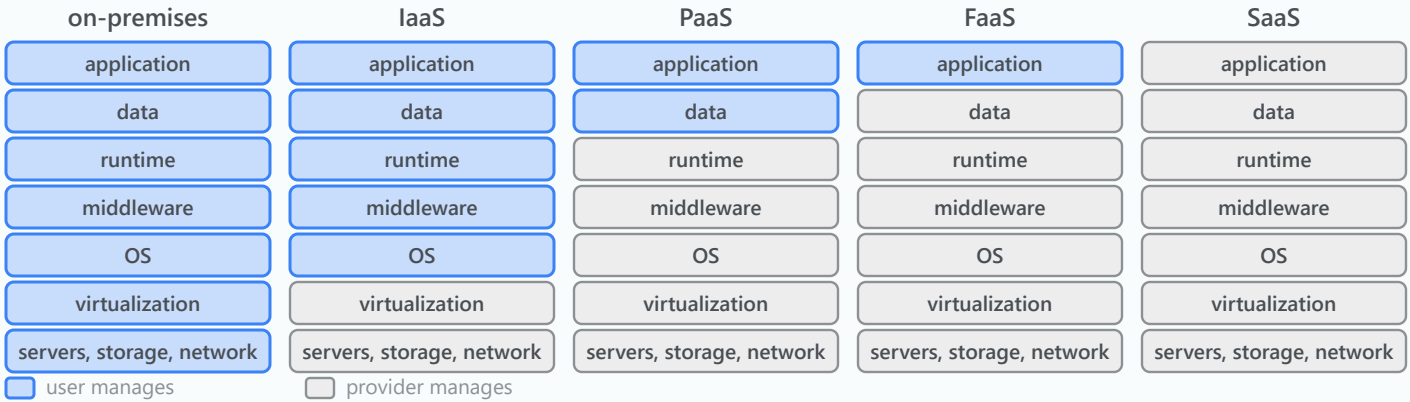
Publish-Subscribe and Jini



- A publisher publishes on a **topic** and a subscriber subscribes to topics it cares about; neither knows the other's identity: **decoupled in time and space**.
- Subscribers and topics are added dynamically → scalable and flexible. Delivery guarantees depend on broker settings (at most once / at least once / durable queue).
- Today: MQTT (IoT), Apache Kafka, RabbitMQ, cloud queues.

- **Jini** (Sun, 1998): Java devices **discover** each other on a network without prior configuration, register with a lookup service and use each other via Java RMI; leases make registrations expire on their own.
- It continued as Apache River but is no longer active; its ideas live on in service discovery (mDNS/DNS-SD, Consul, Kubernetes services).

Cloud and Clusters



- **Cluster**: servers linked by a fast network under one administration (HPC, data centers). **Cloud**: those resources offered over the network on demand, pay as you go (AWS, Azure, Google Cloud).
- Infrastructure: KVM/Nitro virtualization, containers and **Kubernetes** (schedules pods onto nodes and restarts failed ones: today's counterpart of a distributed OS).
- Every topic in this chapter comes together here: NUMA servers, virtual machines, RDMA networks, load balancing, publish-subscribe queues.

Summary

Hardware

UMA (bus, crossbar n^2, omega $(n/2)\log_2 n$) and NUMA; snooping or directories for coherence.

Caches

MESI transitions; writes generate invalidations; false sharing happens at line granularity and is fixed by padding.

OS and locks

Separate OS, master-slave, SMP; bus/line locking for TSL, TTAS, ticket and MCS; spin for short waits, sleep for long ones.

Scheduling

A shared queue means one lock and cold caches; space sharing and gang scheduling for related threads.

Multicomputers

Topology diameter, store-and-forward and cut-through, blocking/nonblocking send, RPC, DSM, load balancing.

Virtualization and distribution

Type 1/2, VT-x, EPT, containers; middleware: document, file, object, coordination; the cloud.

SUMMARY

Review Questions

1. How many crosspoints does a 64 × 64 crossbar have, and how many 2 × 2 switches does a 64-input omega network have?
2. In the omega network, which switches and outputs does a CPU 011 → memory 110 request pass through?
3. Local 80 ns, remote 140 ns, 75% of accesses local: what is the average access time?
4. In MESI, what is the bus-level difference between writing a line in state E and in state S?
5. Why can two threads' separate counters slow each other down? What is the fix?
6. In which two ways is a ticket lock better than a plain TSL loop?
7. What are the two conditions of gang scheduling, and what problem does it solve?
8. What is the diameter of a 256-node hypercube? Of a 16 × 16 torus?
9. With 4-level tables and EPT, how many memory accesses does a TLB miss need in the worst case?
10. Which of the tuples `("abc", 2, 5)` and `("abc", 2.0, 5)` does `in("abc", ? x, 5)` (x int) match?

▶ **Answers**

References

- A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, 4th ed. (2014) / 5th ed. (2023), Pearson — Chapter 8: Multiple Processor Systems; Chapter 7: Virtualization and the Cloud.
- A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10th ed., Wiley, 2018 — Section 5.5 (Multi-Processor Scheduling), Chapters 18 (Virtual Machines) and 19 (Networks and Distributed Systems).
- R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces* (OSTEP), ostep.org — “Multiprocessor Scheduling”, “Locks”, “Distributed Systems”, “Sun’s NFS”, “AFS”, Appendix: “Virtual Machine Monitors”.
- J. L. Hennessy, D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 6th ed., Morgan Kaufmann, 2017 — Chapter 5: Thread-Level Parallelism (coherence, MESI, directory protocols).
- J. M. Mellor-Crummey, M. L. Scott, “Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors”, ACM TOCS, 1991; A. D. Birrell, B. J. Nelson, “Implementing Remote Procedure Calls”, ACM TOCS, 1984; G. J. Popek, R. P. Goldberg, CACM, 1974.
- Linux kernel documentation: “CFS/EEVDF Scheduler”, “Scheduler Domains”, “NUMA Memory Policy”, “Core Scheduling”; Intel SDM Vol. 3 (VMX, EPT).

The End

Chapter 12: Multiprocessor Systems

Previous: Chapter 11: Input/Output · Next: Chapter 13: Security