Multiple Processor Systems

Single CPU Computers

- the CPU can execute only one instruction at a time
- program execution is purely sequential
- multiprogramming is possible thanks to time division
- increasing performance means making the clock faster
- fundamental limit #1: $c \approx 20\,\mathrm{cm/ns}$ in wire or fiber
  - $10\,\mathrm{GHz}$ system must be smaller than $2\,\mathrm{cm}$
- fundamental limit #2: heat dissipation
  - the smaller the system the more heat it generates

Solution: Parallelization

- many CPUs running at “normal” speed, for some definition of “normal”
- speed up computations
  - at least those that can be parallelized
- deal with heavier loads
  - different CPUs deal with different transactions, users
- enormous range of systems:
  - single servers with 2, 4, 8, 16, and more CPUs
  - supercomputers and clusters ($10 \div 10^5$ CPUs)
  - internet-wide computations (e.g. SETI@home)
  - grid computing

Locality of Reference

a concept related to accessing a resource multiple times

- locality comes in flavours:
  - temporal: a resource referenced at one point will be referenced again in the near future
  - spatial: a resource is more likely to be referenced if a nearby resource has been referenced recently
  - sequential: memory is accessed sequentially
- reason: related data are stored sequentially in memory
  - structures, arrays, etc
- related data items are often accessed one after another
  - loops

Locality of Reference II

useful for performance optimization

- caching is based on temporal locality
- caching also uses spatial locality
  - data are brought into cache in cache lines
  - nearby data will be brought into cache with the referenced item
- paging benefits from spatial locality
- data that are referenced often can be kept in CPU registers
  - in C we can declare variables as register (a suggestion to the compiler)

Multiple Processor Systems

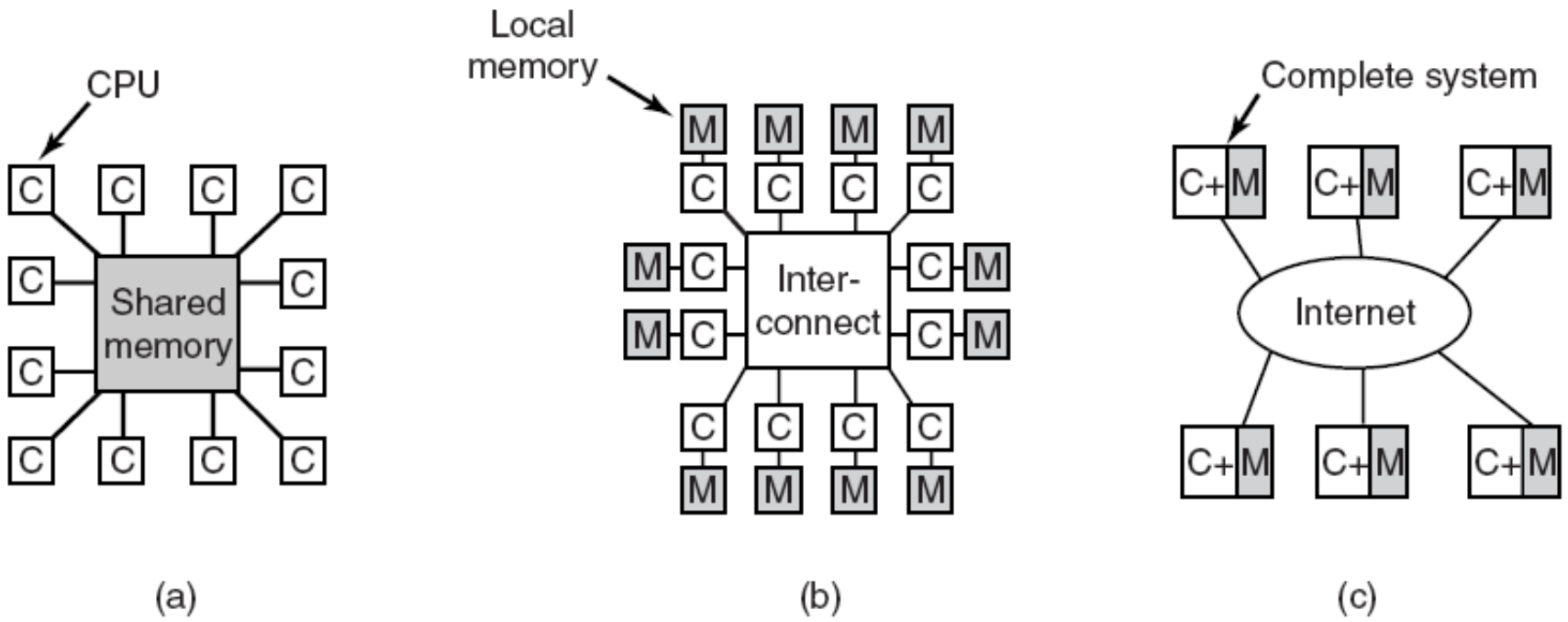


Figure 8-1. (a) A shared-memory multiprocessor. (b) A message-passing multicomputer. (c) A wide area distributed system.

(a) is a Uniform Memory Access (UMA) architecture, while (b) and (c) are Non-Uniform Memory Access (NUMA) architectures

UMA Multiprocessors with Bus-Based Architectures

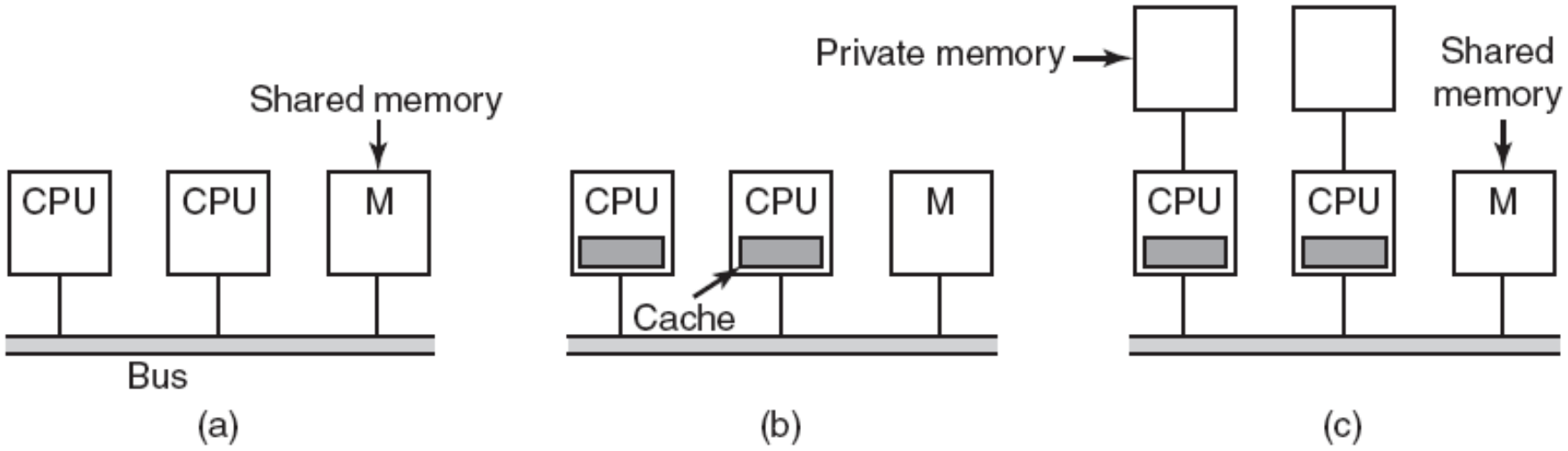


Figure 8-2. Three bus-based multiprocessors. (a) Without caching. (b) With caching. (c) With caching and private memories.

UMA Multiprocessors Using Crossbar Switches

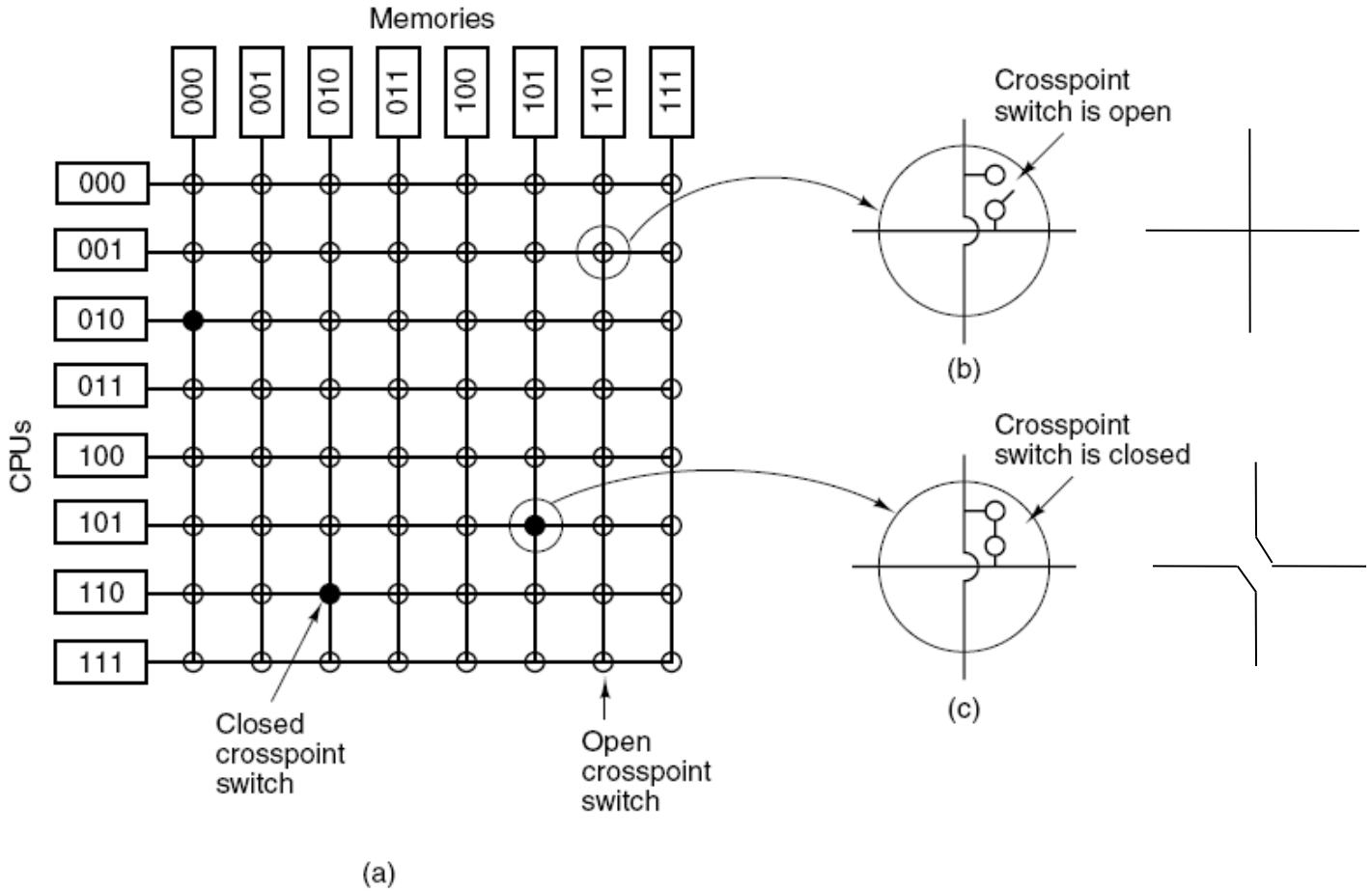


Figure 8-3. The "Dance Hall" approach: (a) An 8 × 8 crossbar switch. (b) An open crosspoint. (c) A closed crosspoint.

Interconnection Technology (1)

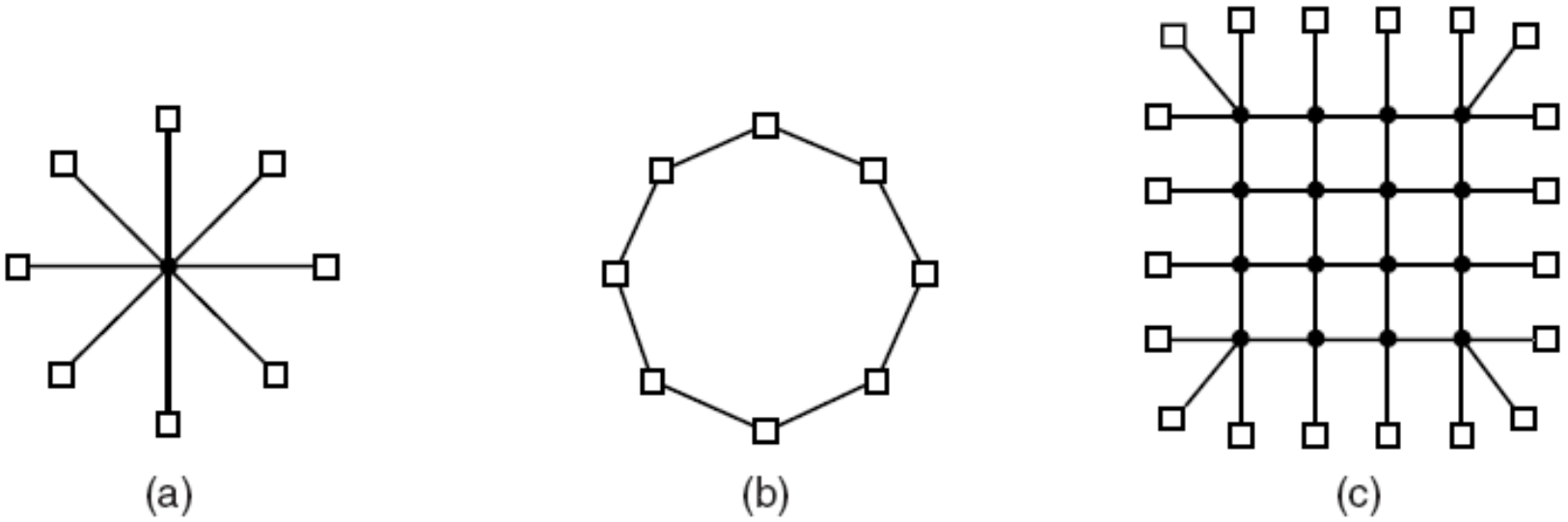


Figure 8-16. Various interconnect topologies. (a) A single switch. (b) A ring. (c) A grid.

Interconnection Technology (2)

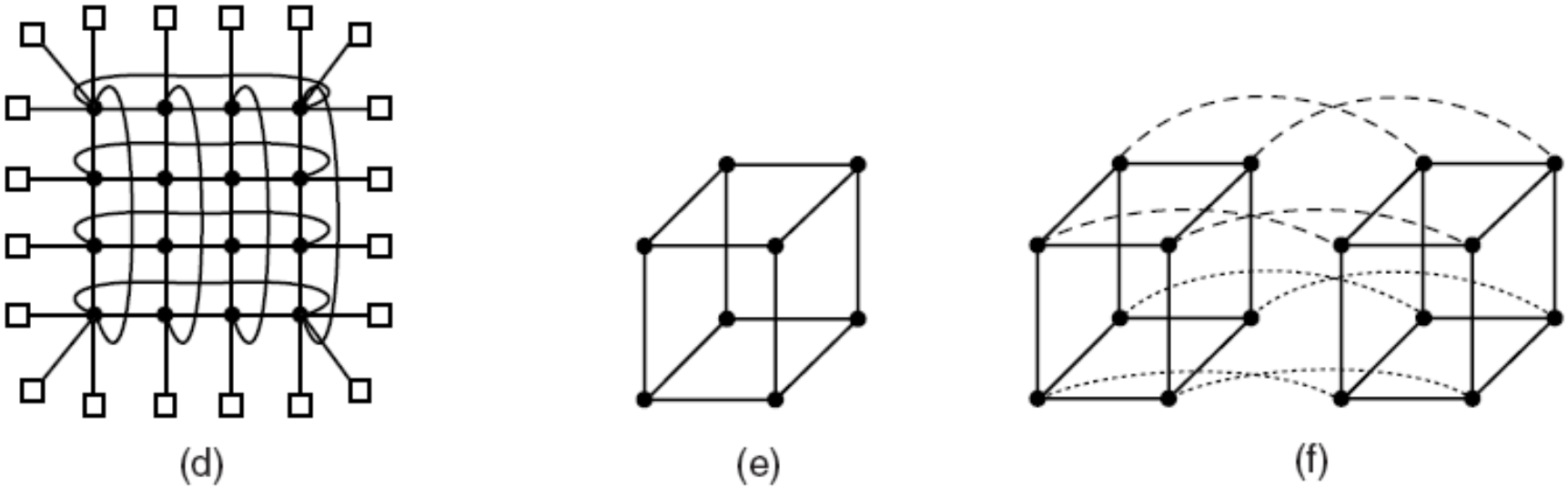


Figure 8-16. Various interconnect topologies.
(d) A double torus. (e) A cube. (f) A 4D hypercube.

UMA Multiprocessors Using Multistage Switching Networks (1)

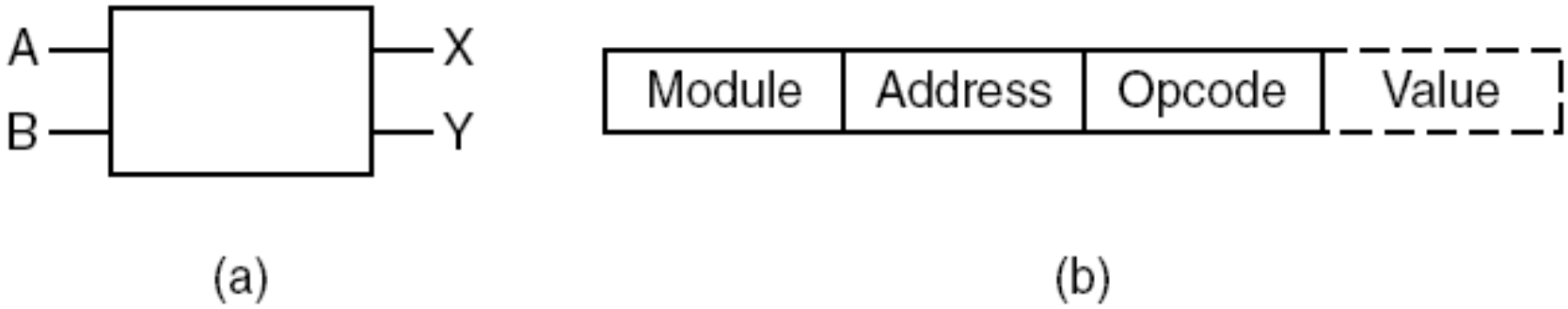


Figure 8-4. (a) A 2 × 2 switch with two input lines, A and B, and two output lines, X and Y. (b) A message format.

UMA Multiprocessors Using Multistage Switching Networks (2)

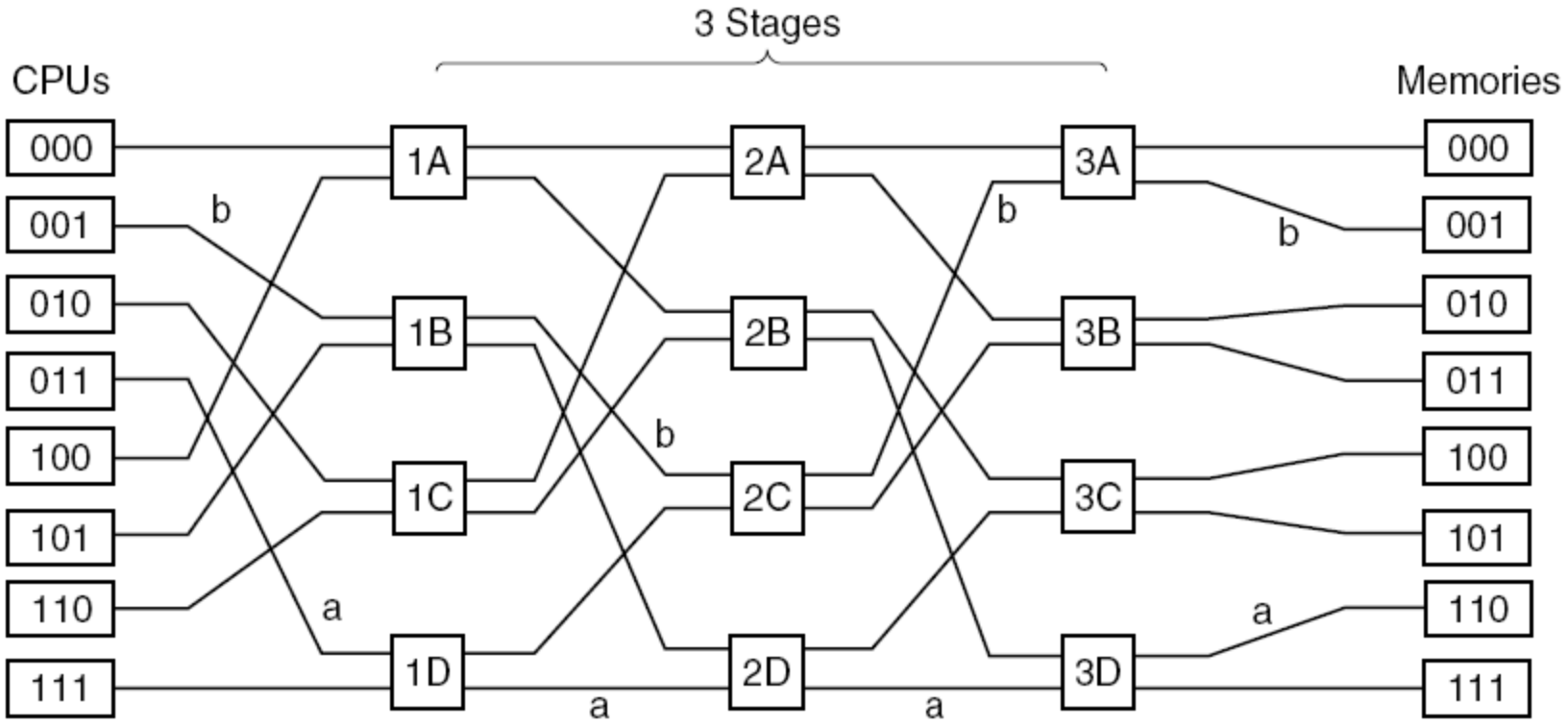


Figure 8-5. An omega switching network.

NUMA Multiprocessors (1)

Characteristics of NUMA machines:

1. There is a single address space visible to all CPUs.
2. Access to remote memory is via LOAD and STORE instructions.
3. Access to remote memory is slower than access to local memory.
4. Tow Types

   CC_NUMA (Cache Coherence NUMA)

   NC_NUMA

NUMA Multiprocessors (1)

CC_NUMA (Cache Coherence NUMA)

NC_NUMA

NUMA Multiprocessors (2)

Directory based method is a famous model for CC_NUMA

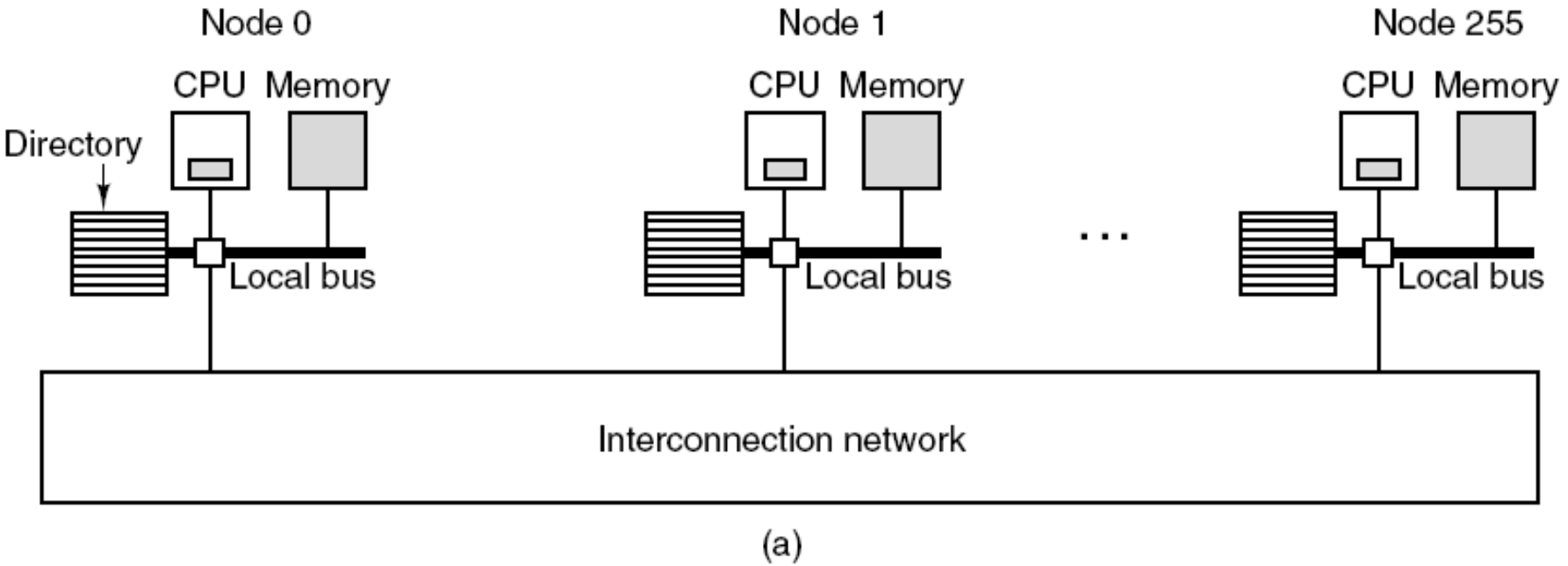
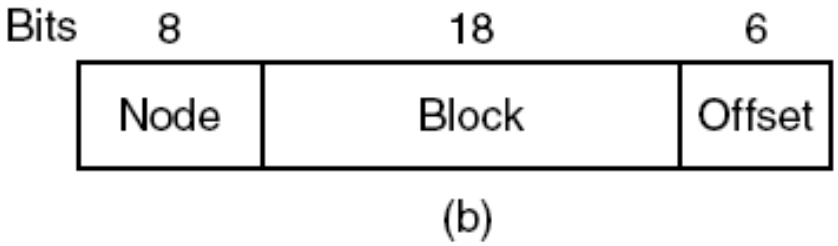


Figure 8-6. (a) A 256-node directory-based multiprocessor.

NUMA Multiprocessors (3)



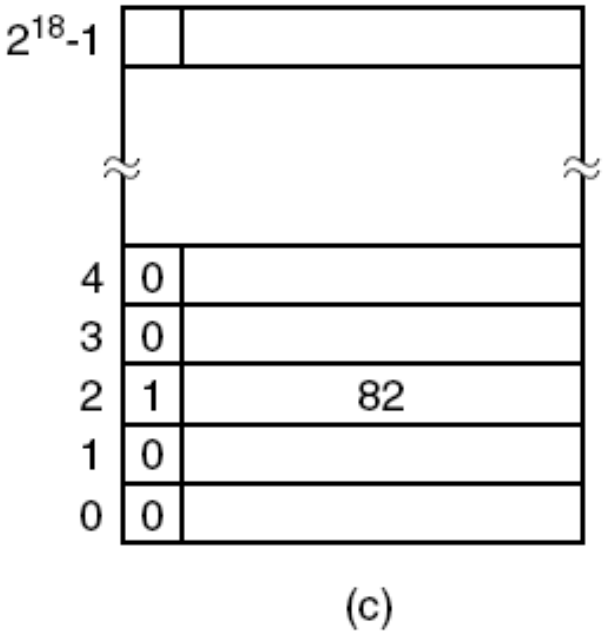


Figure 8-6. (b) Division of a 32-bit memory address into fields. (c) The directory at node 36.

Each CPU Has Its Own Operating System

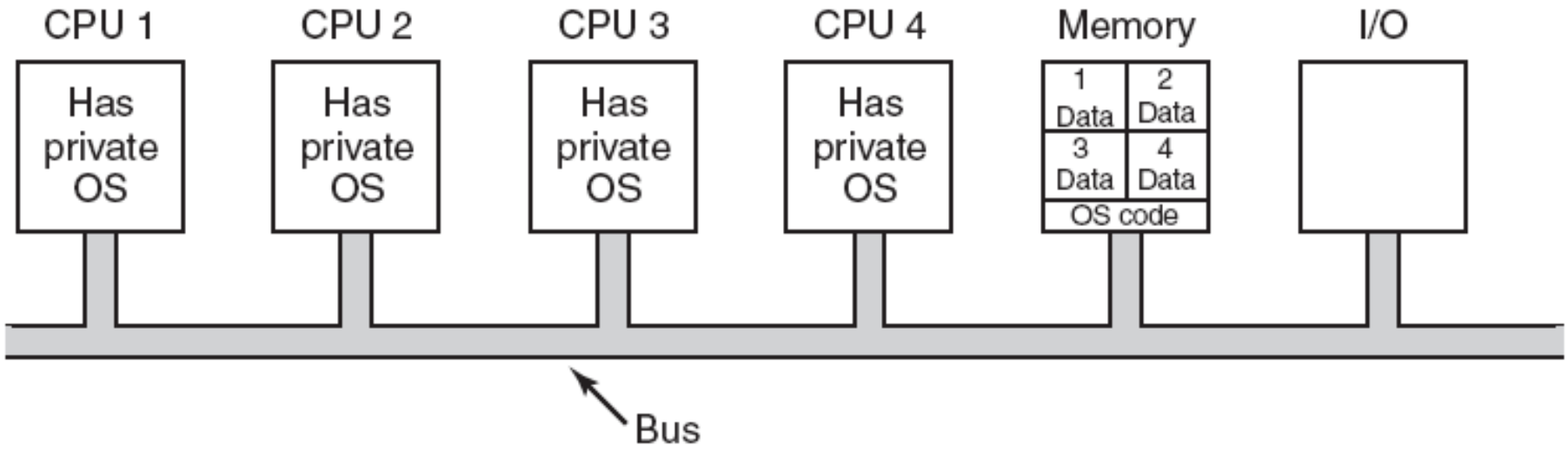


Figure 8-7. Partitioning multiprocessor memory among four CPUs, but sharing a single copy of the operating system code. The boxes marked Data are the operating system's private data for each CPU.

Multiprocessors with Private OS II

- better than n independent computers
  - shared I/O
  - flexible memory allocation
  - effective inter-processor communication
- system calls are handled locally — private tables etc
- no process sharing: CPU 1 idle while CPU 2 overloaded
- no page sharing: CPUs cannot borrow/loan pages
- local buffer caches (of recently used disk blocks)
  - if a block is present and dirty in multiple buffer caches the system is in inconsistent state
  - eliminating buffer caches hurts performance

Master-Slave Multiprocessors

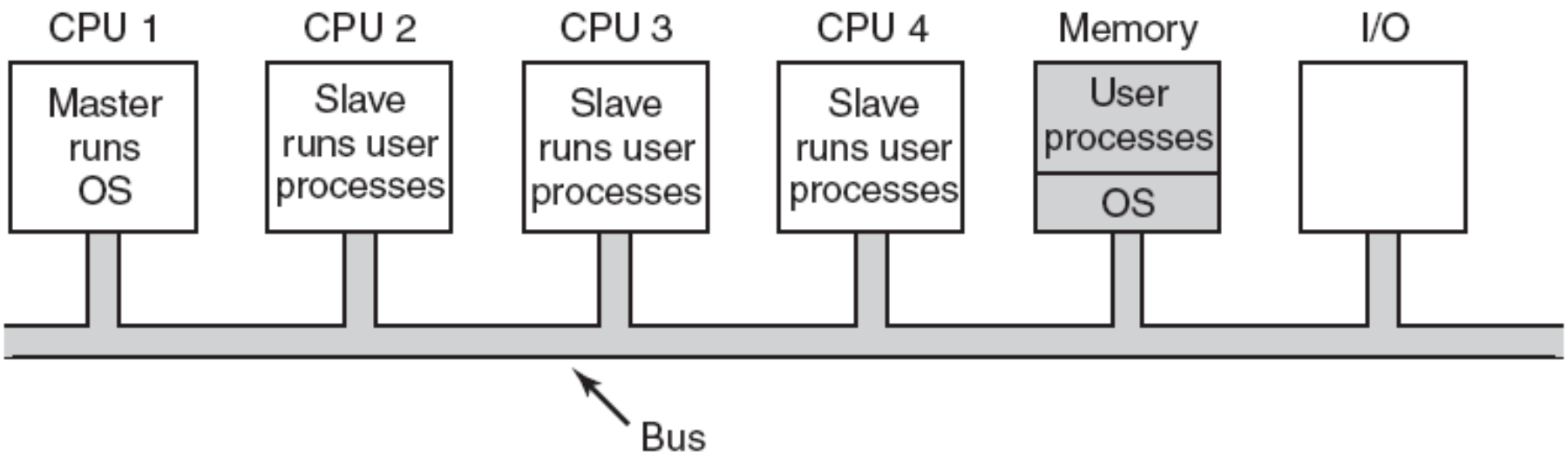


Figure 8-8. A master-slave multiprocessor model.

Master-Slave Multiprocessors II

- solves most of the problems of the private OS scheme
  - there is a single set of OS data structures
  - a CPU will never stay idle when another is overloaded
  - pages can be allocated among all the processes dynamically
  - there is one buffer cache, so no inconsistencies will occur
- problem: the master CPU is a bottleneck
  - must handle all the system calls from all the slaves
  - example: if 10% of the time is spent in system calls, the master will be saturated by 10 CPUs

Symmetric Multiprocessors

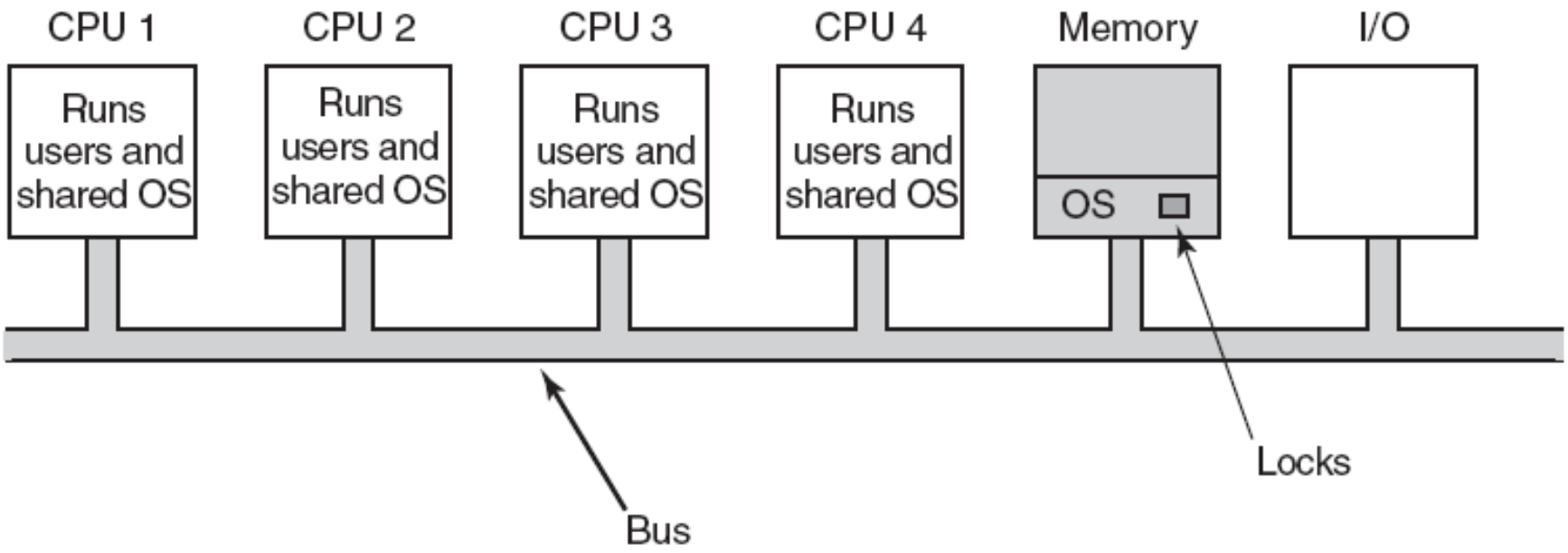


Figure 8-9. The SMP multiprocessor model.

Symmetric Multiprocessors II

- balances processes and memory dynamically
- there is only one set of OS tables
- eliminates the master CPU bottleneck
- problem: need to synchronize the CPUs
  - imagine 2 CPUs scheduling the same process to run
  - or claiming the same free memory page
- solution: protect the OS with a mutex
  - any CPU can run the OS, but only one at a time can do it
  - almost as bad as master-slave: CPUs will queue to get the OS

SMP Synchronization

- solution: split the OS into independent critical regions, protect each with its own mutex
- some tables may be used by multiple critical sections
  - e.g. process table is used by
    - scheduler
    - fork()
    - signal handling
  - such tables need their own mutexes
- such organization is hard to design...
- ... and is even harder to program

Multiprocessor Synchronization (1)

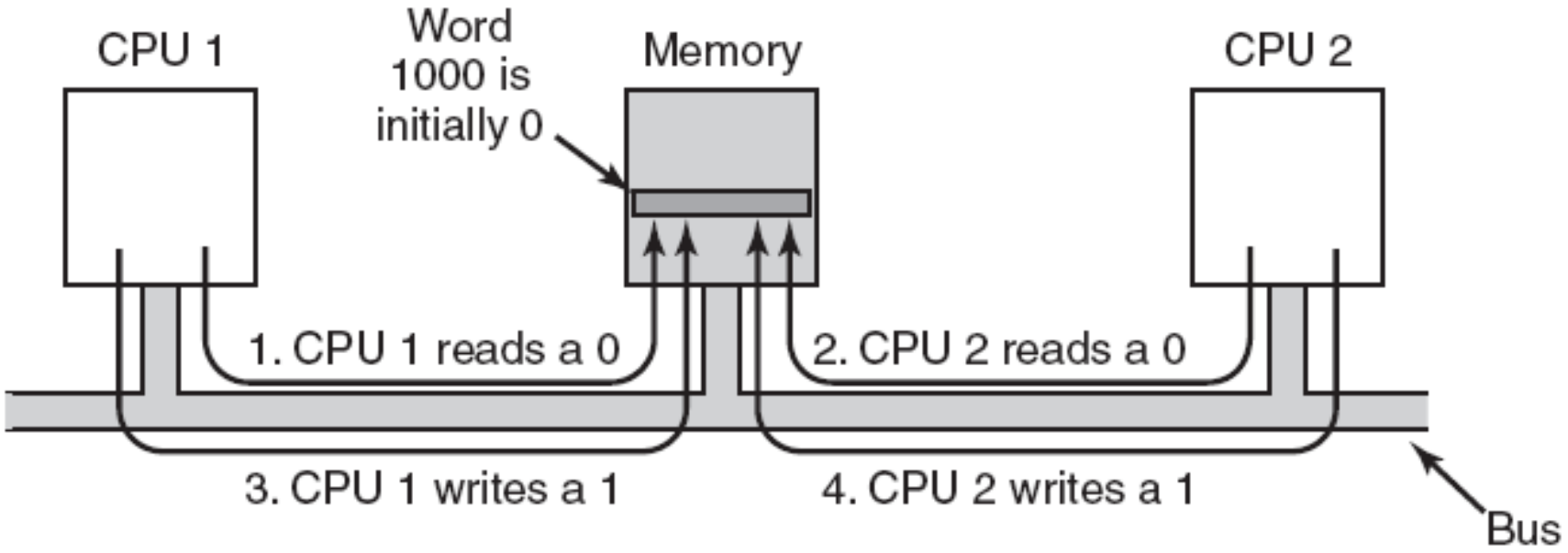


Figure 8-10. The TSL instruction can fail if the bus cannot be locked. These four steps show a sequence of events where the failure is demonstrated.

TSL solution for multi-processors

TSL involves testing and setting memory, this can require 2 memory accesses

- Not a problem to implement this in single-processor system

Now, bus must be locked to avoid split transaction

- Bus provides a special line for locking

A process that fails to acquire lock checks repeatedly issuing more TSL instructions

- Requires Exclusive access to memory block
- Cache coherence protocol would generate lots of traffic

Goal: To reduce number of checks

1. Exponential back-off: instead of constant polling, check only after delaying (1, 2, 4, 8 instructions)
2. Maintain a list of processes waiting to acquire lock.

Busy-Waiting vs Process switch

In single-processors, if a process is waiting to acquire lock, OS schedules another ready process

OS must decide whether to switch (choice between spinning and switching)

- spinning wastes CPU cycles
- switching uses up CPU cycles also
- possible to make separate decision each time locked mutex encountered

Multiprocessor Synchronization (2)

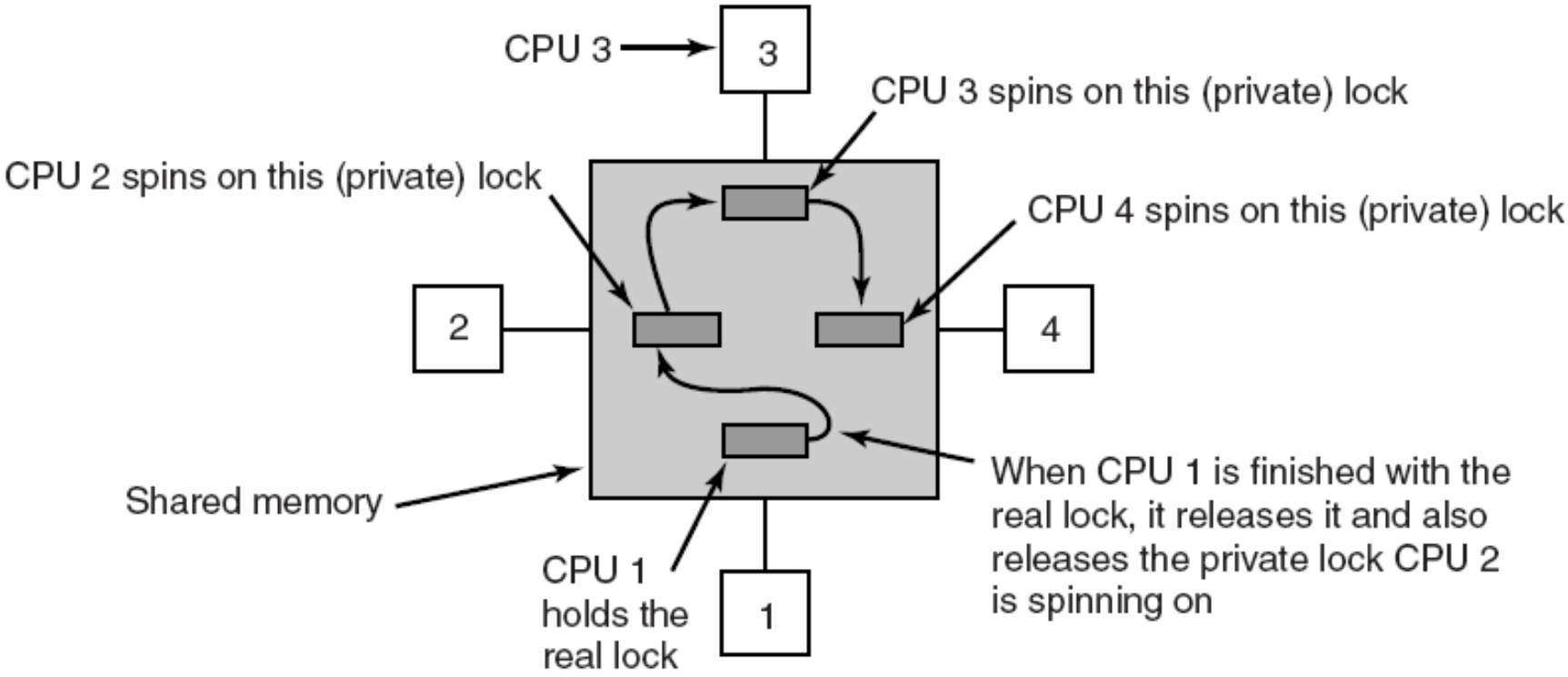


Figure 8-11. Use of multiple locks to avoid cache thrashing.

Timesharing

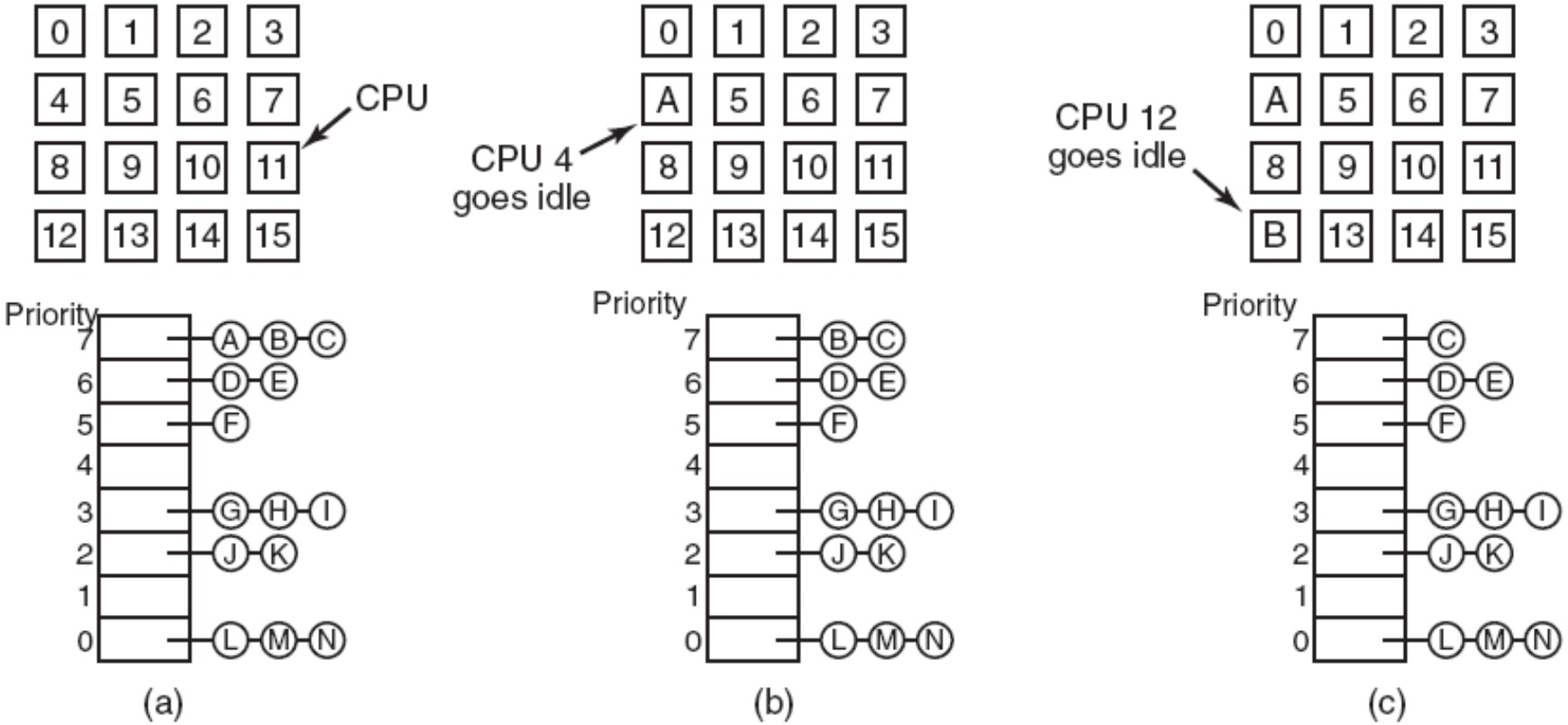


Figure 8-12. Using a single data structure for scheduling a multiprocessor.

Space Sharing

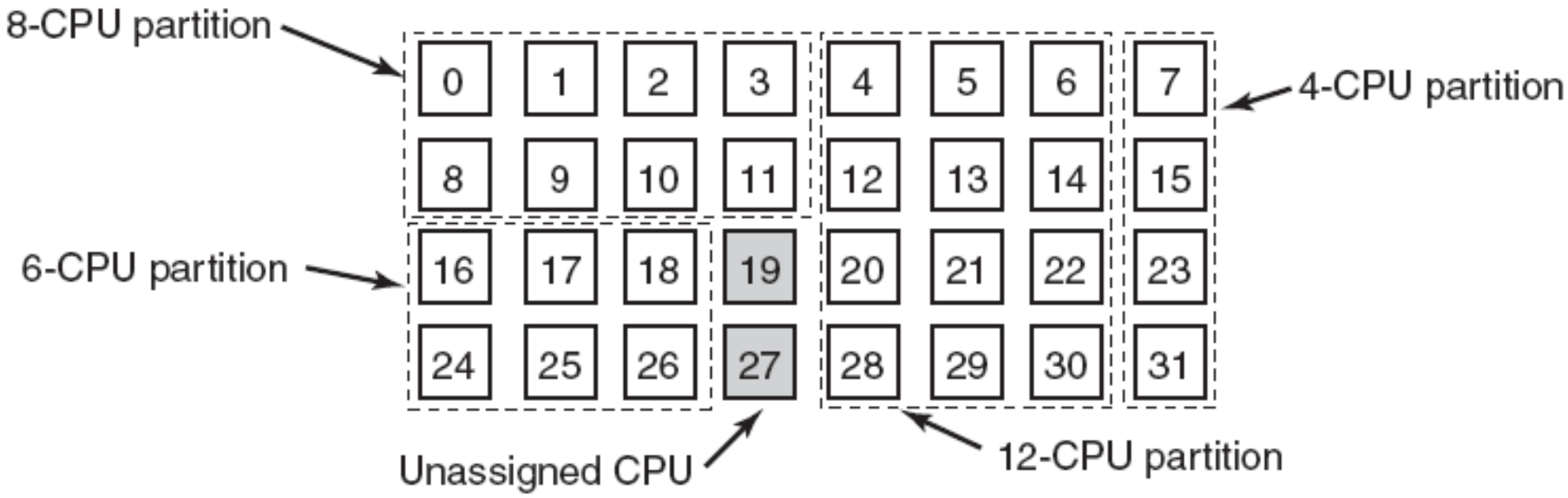


Figure 8-13. A set of 32 CPUs split into four partitions, with two CPUs available.

Gang Scheduling (1)

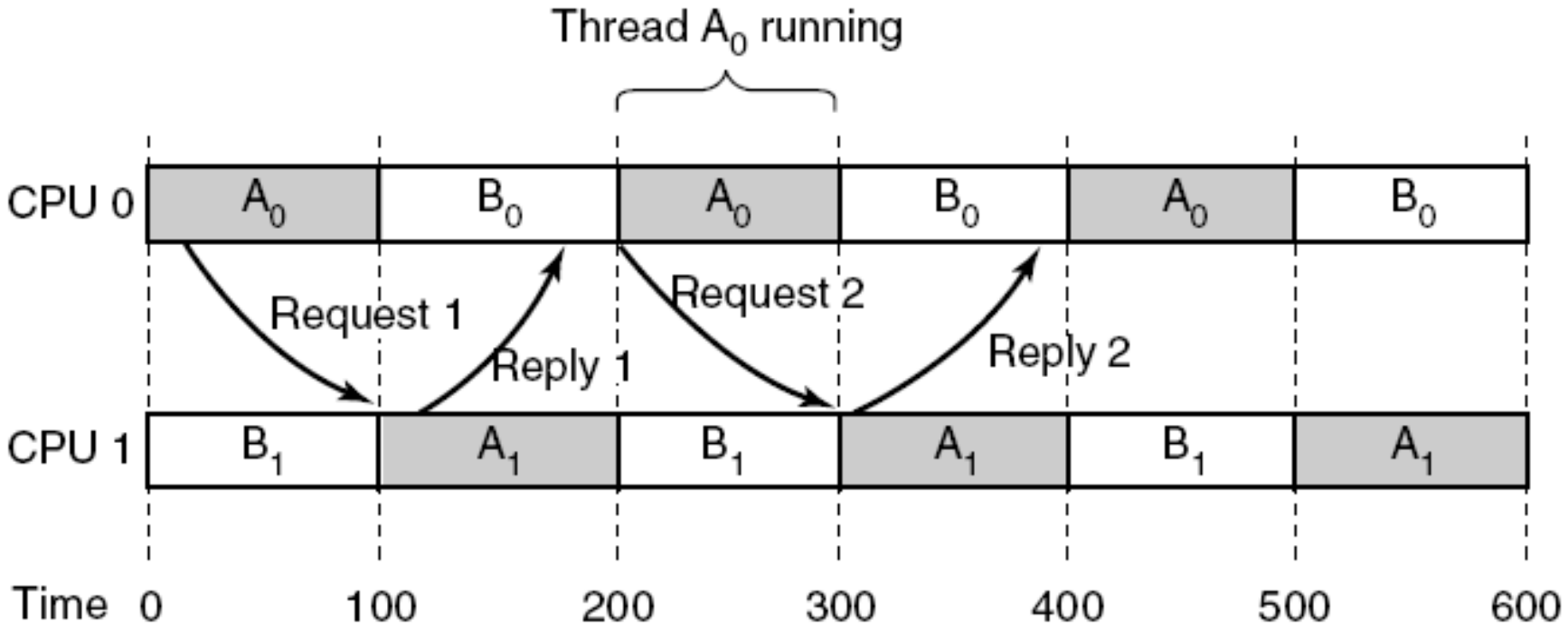


Figure 8-14. Communication between two threads belonging to thread A that are running out of phase.

Gang Scheduling (2)

The three parts of gang scheduling:

1. Groups of related threads are scheduled as a unit, a gang.
2. All members of a gang run simultaneously, on different timeshared CPUs.
3. All gang members start and end their time slices together.

Gang Scheduling (3)

Time slot	CPU 0	CPU 1	CPU 2	CPU 3	CPU 4	CPU 5
0	A_0	A_1	A_2	A_3	A_4	A_5
1	B_0	B_1	B_2	C_0	C_1	C_2
2	D_0	D_1	D_2	D_3	D_4	E_0
3	E_1	E_2	E_3	E_4	E_5	E_6
4	A_0	A_1	A_2	A_3	A_4	A_5
5	B_0	B_1	B_2	C_0	C_1	C_2
6	D_0	D_1	D_2	D_3	D_4	E_0
7	E_1	E_2	E_3	E_4	E_5	E_6

Figure 8-15. Gang scheduling.

Interconnection Technology (3)

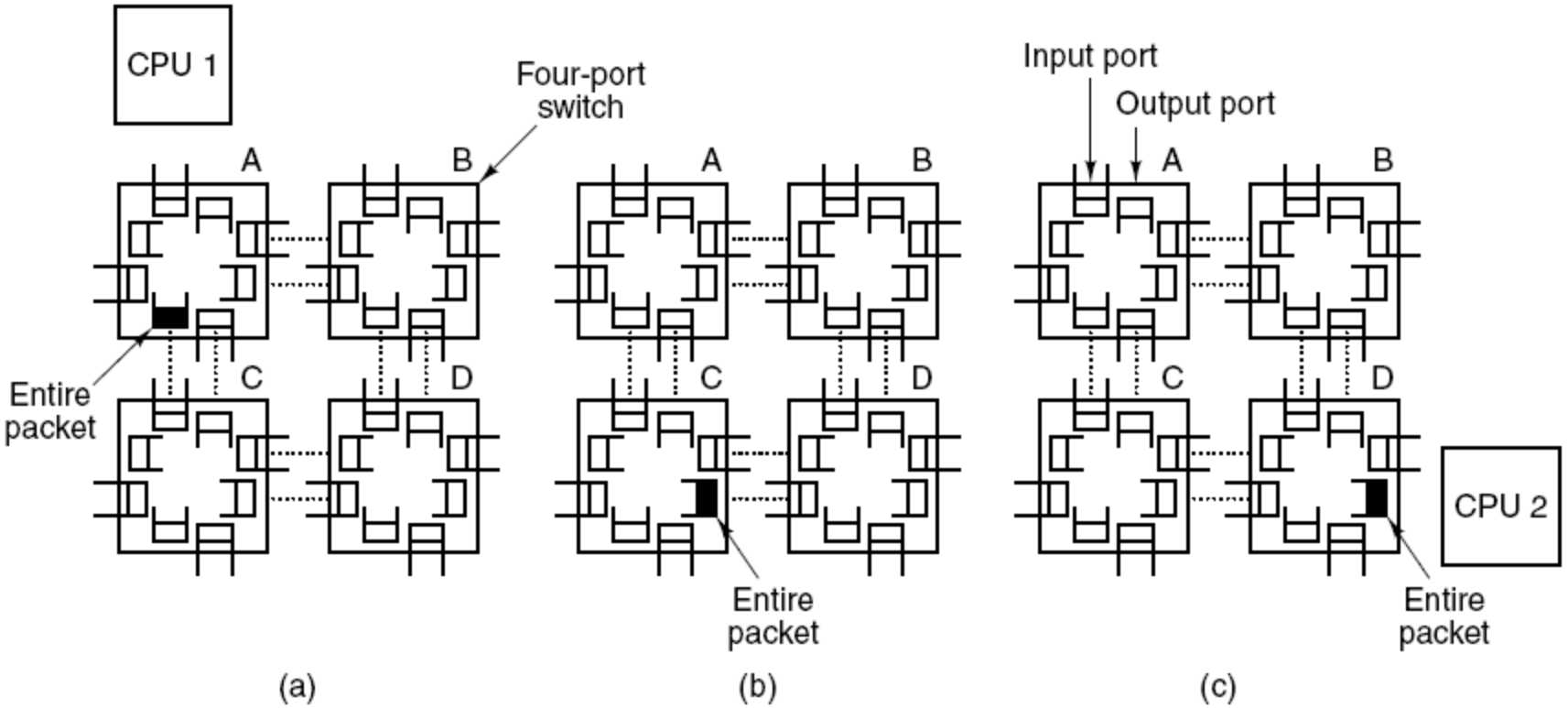


Figure 8-17. Store-and-forward packet switching.

Network Interfaces

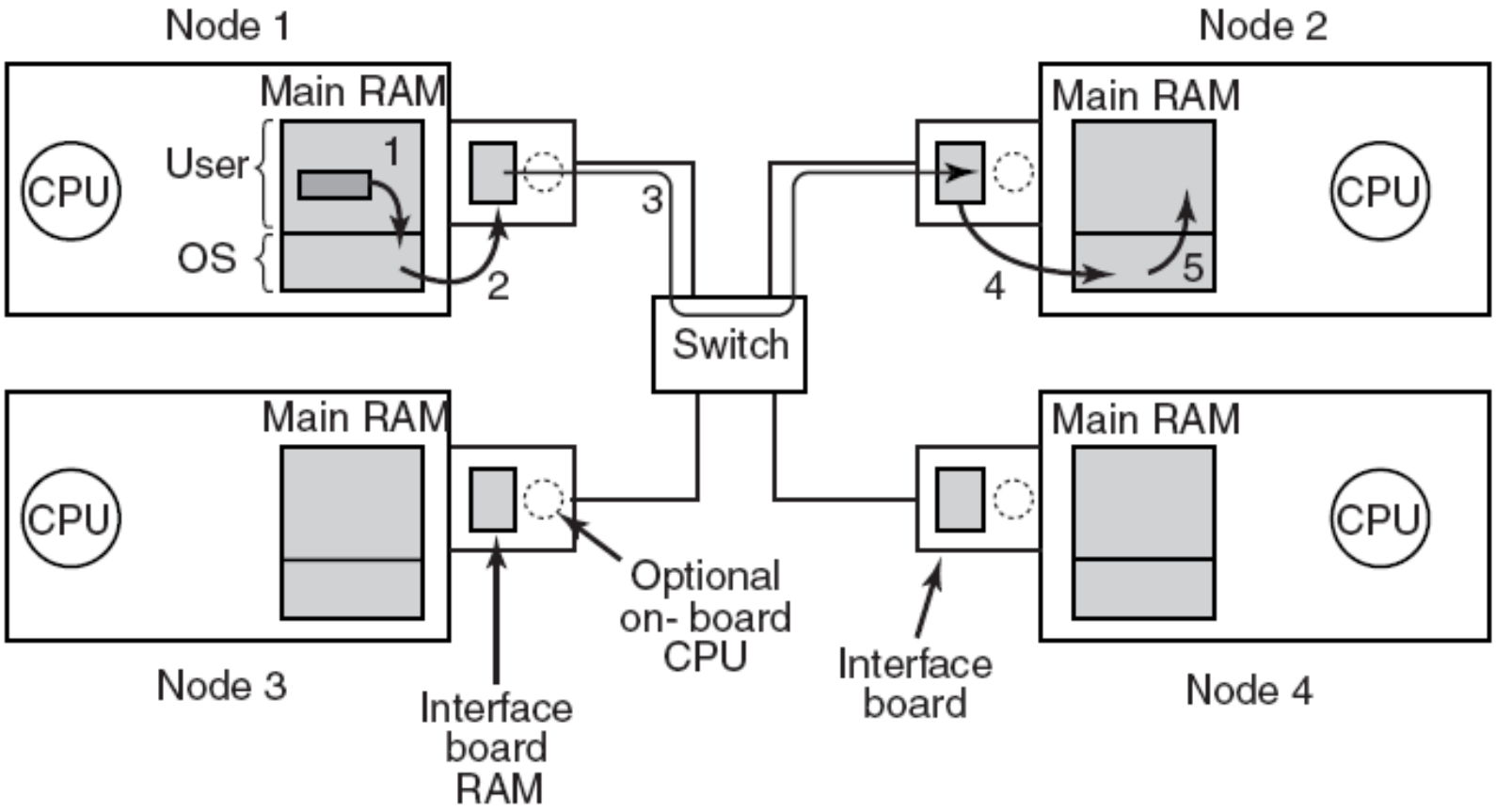


Figure 8-18. Position of the network interface boards in a multicomputer.

Blocking versus Nonblocking Calls (1)

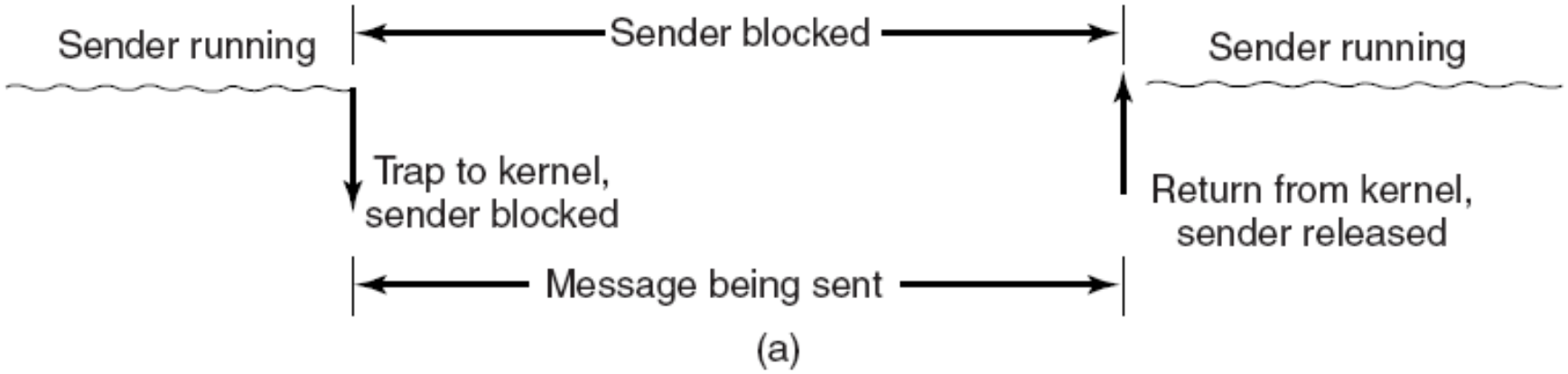


Figure 8-19. (a) A blocking send call.

Blocking versus Nonblocking Calls (2)

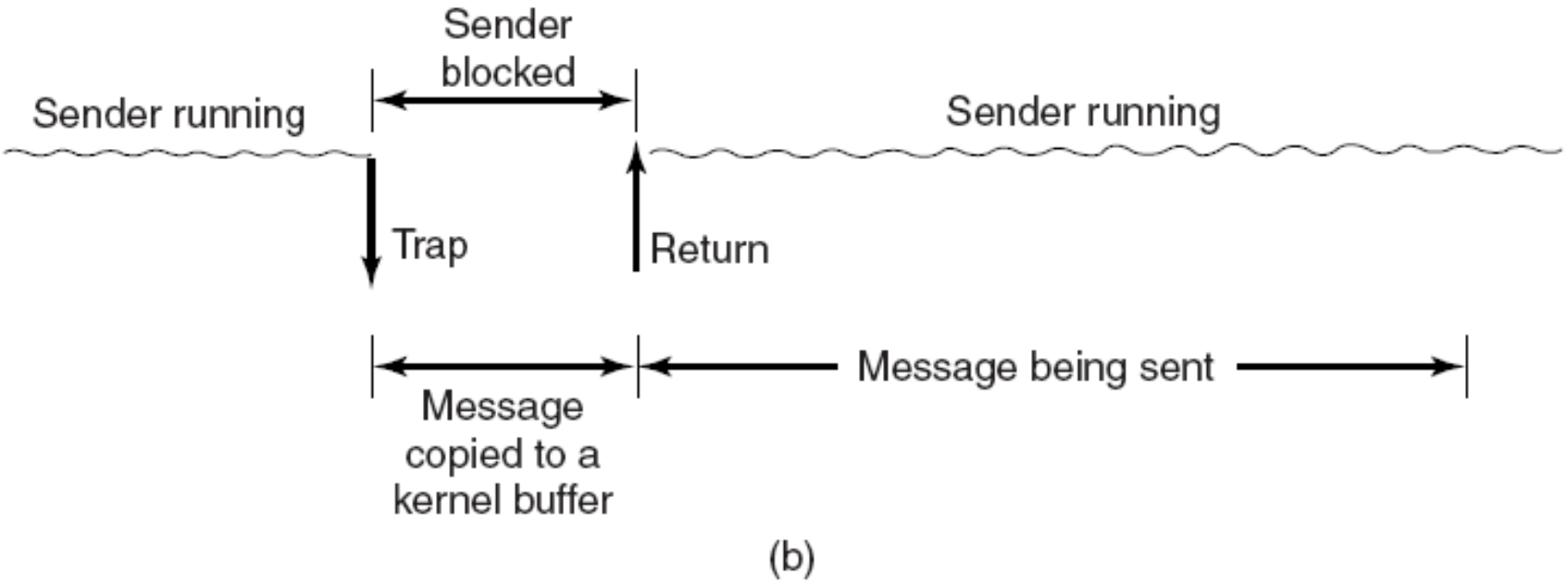


Figure 8-19. (b) A nonblocking send call.

Blocking versus Nonblocking Calls (3)

Choices on the sending side:

1. Blocking send (CPU idle during message transmission).
2. Nonblocking send with copy (CPU time wasted for the extra copy).
3. Nonblocking send with interrupt (makes programming difficult).
4. Copy on write (extra copy probably needed eventually).

Remote Procedure Call

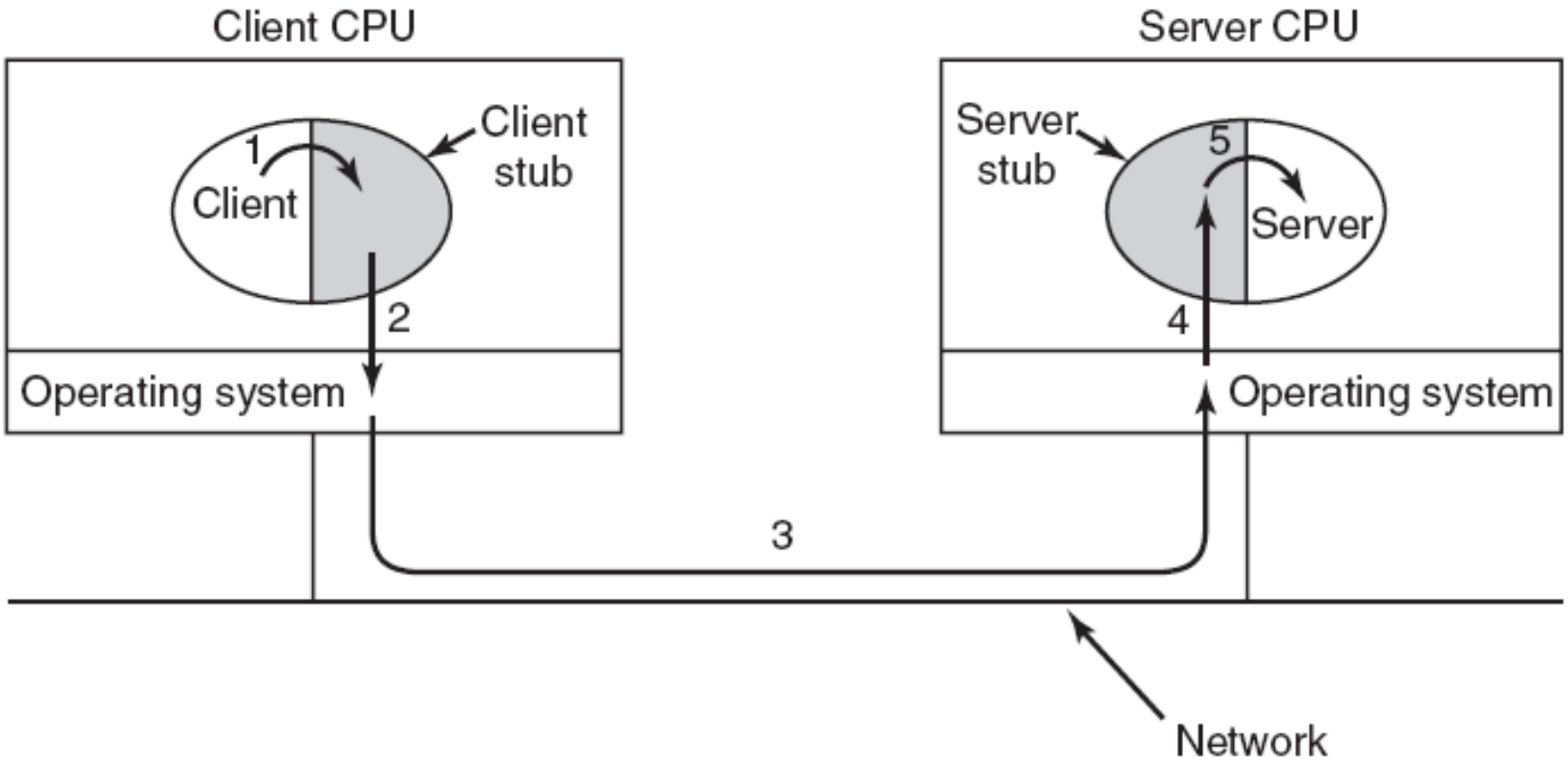
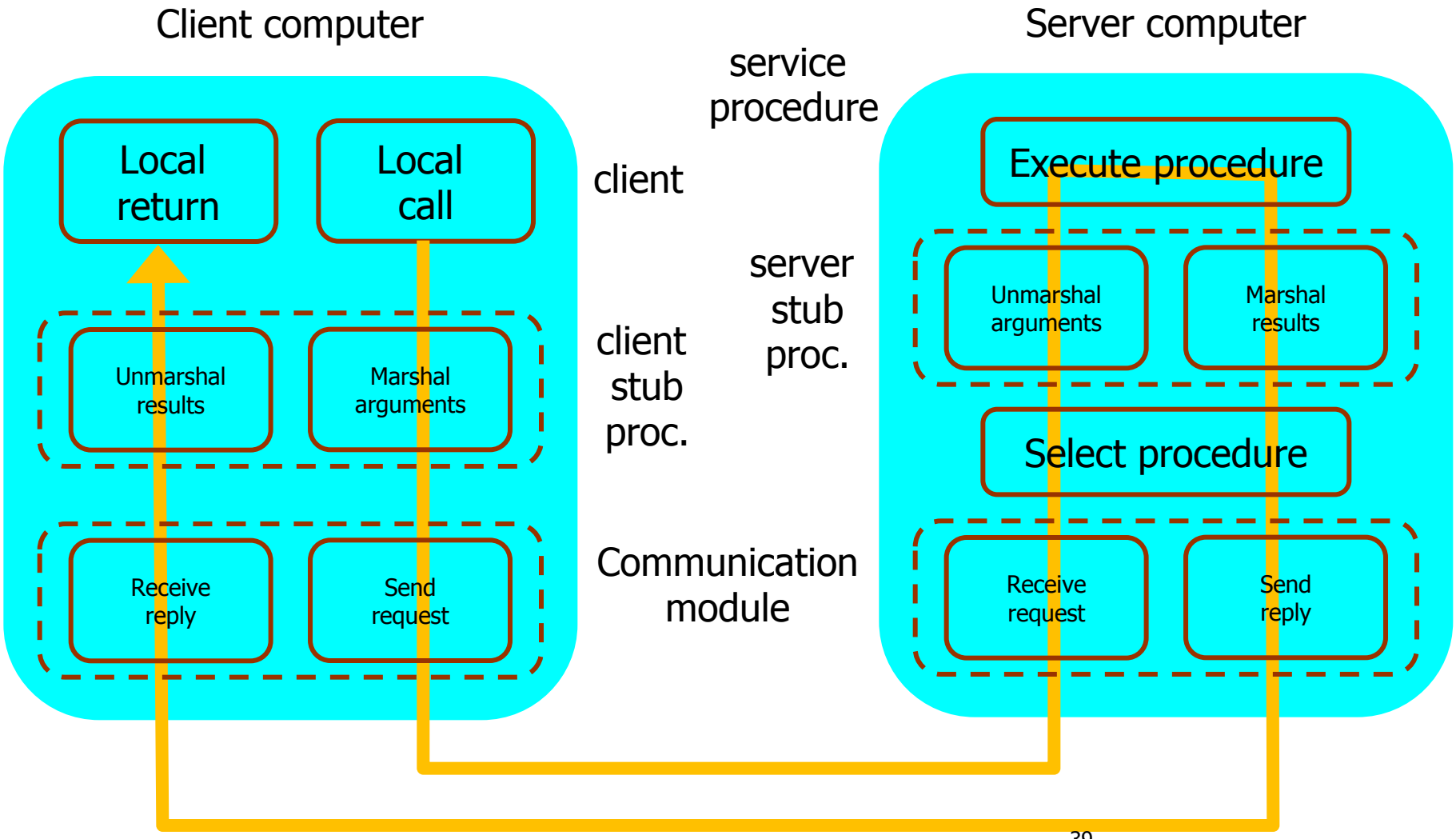


Figure 8-20. Steps in making a remote procedure call. The stubs are shaded gray.

RPC Mechanism



Distributed Shared Memory (1)

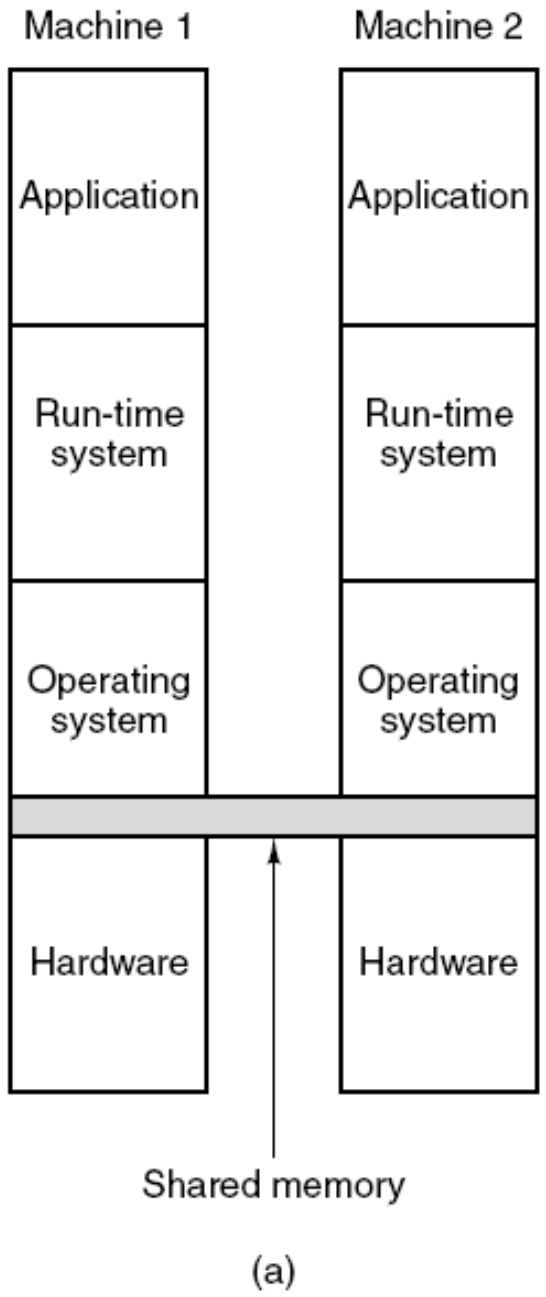


Figure 8-21. Various layers where shared memory can be implemented. (a) The hardware.

Distributed Shared Memory (2)

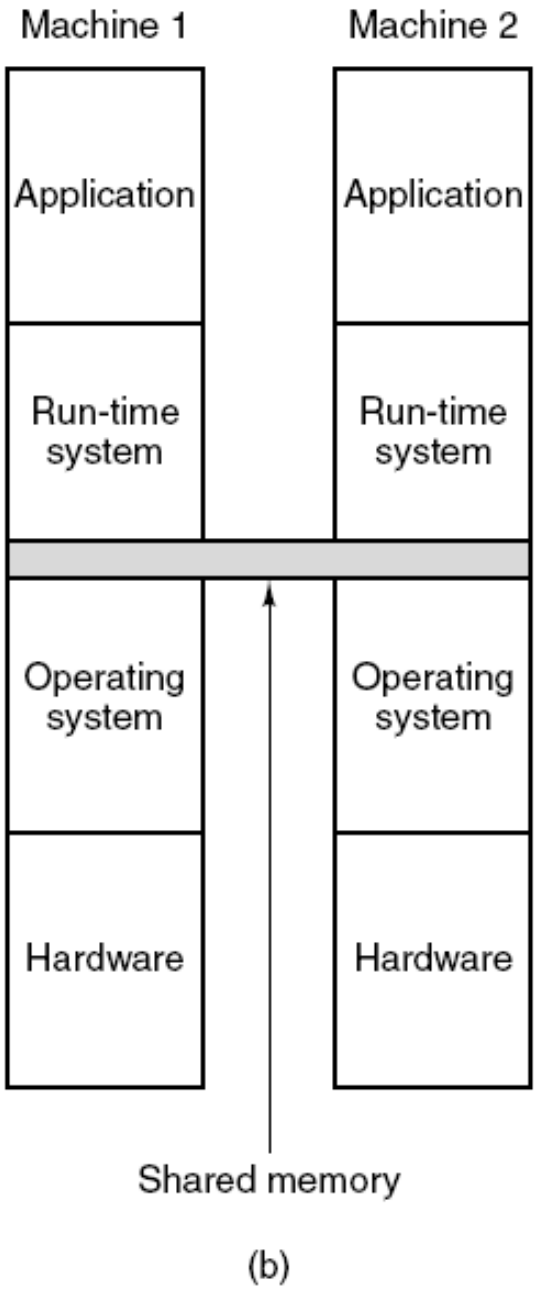


Figure 8-21. Various layers where shared memory can be implemented. (b) The operating system.

Distributed Shared Memory (3)

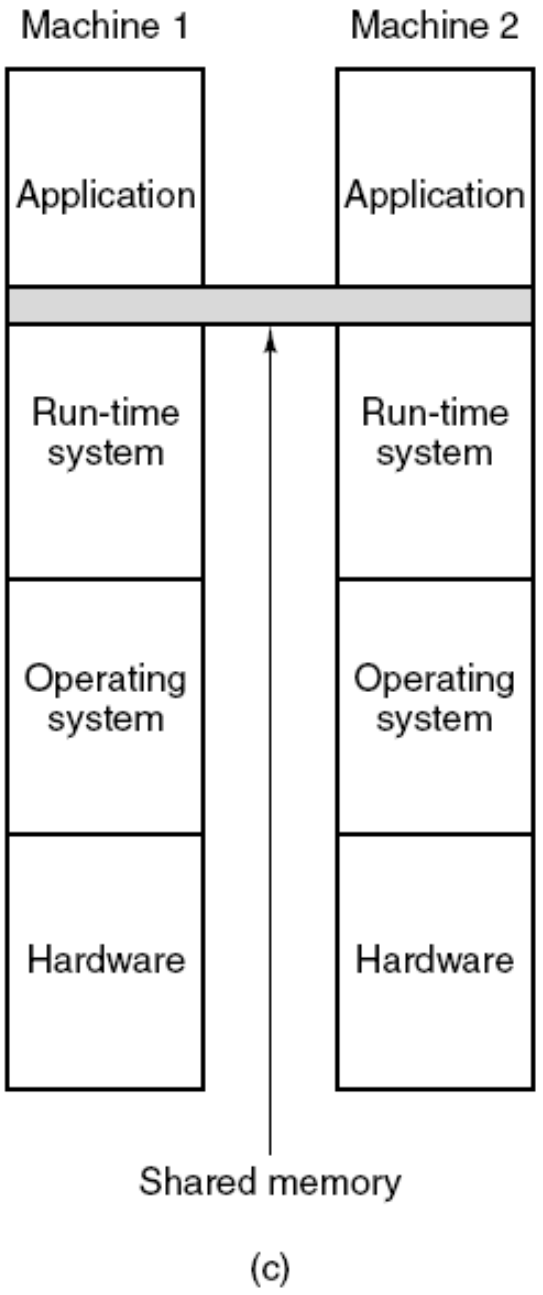


Figure 8-21. Various layers where shared memory can be implemented. (c) User-level software.

Distributed Shared Memory (4)

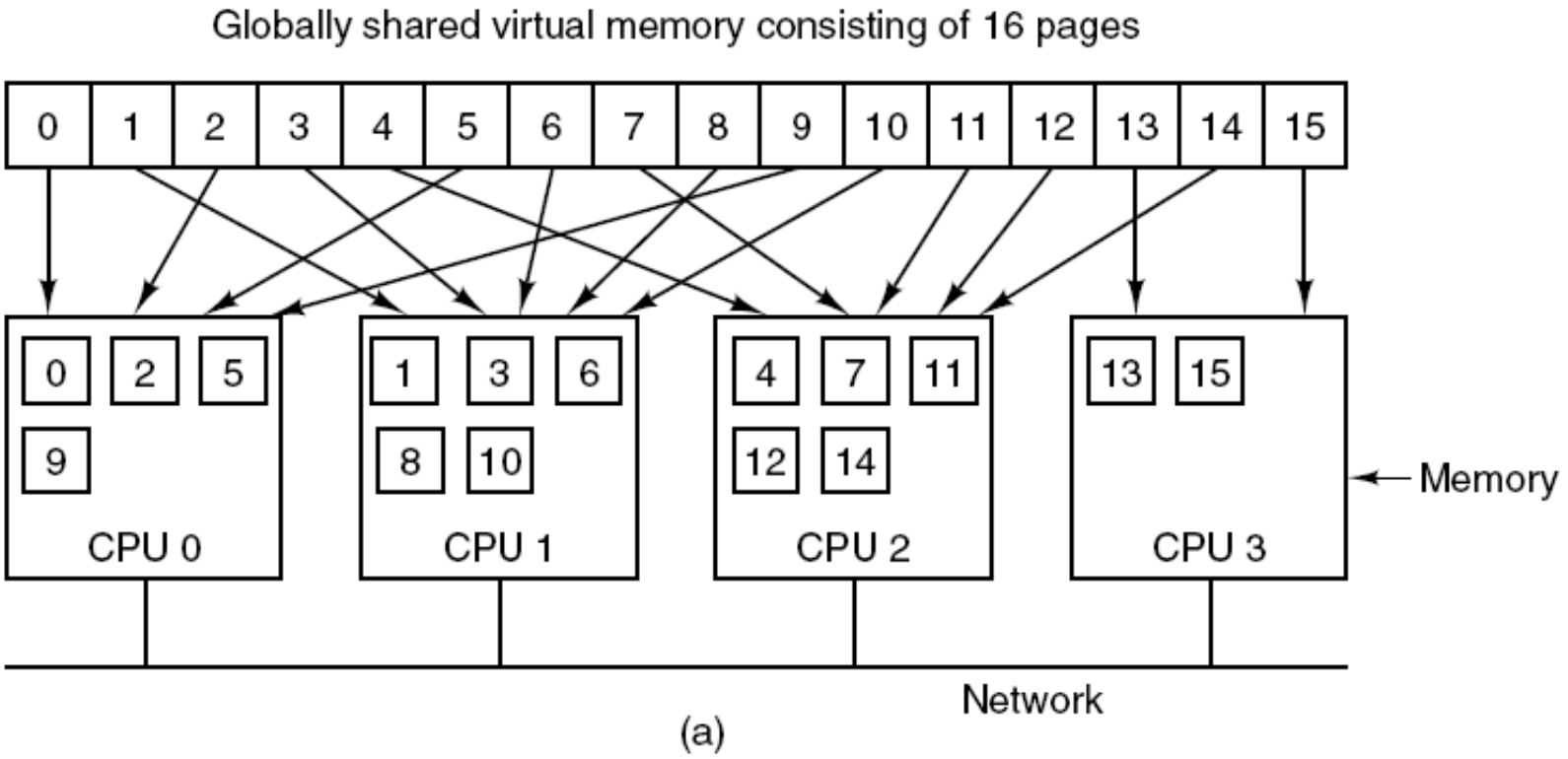


Figure 8-22. (a) Pages of the address space distributed among four machines.

Distributed Shared Memory (5)

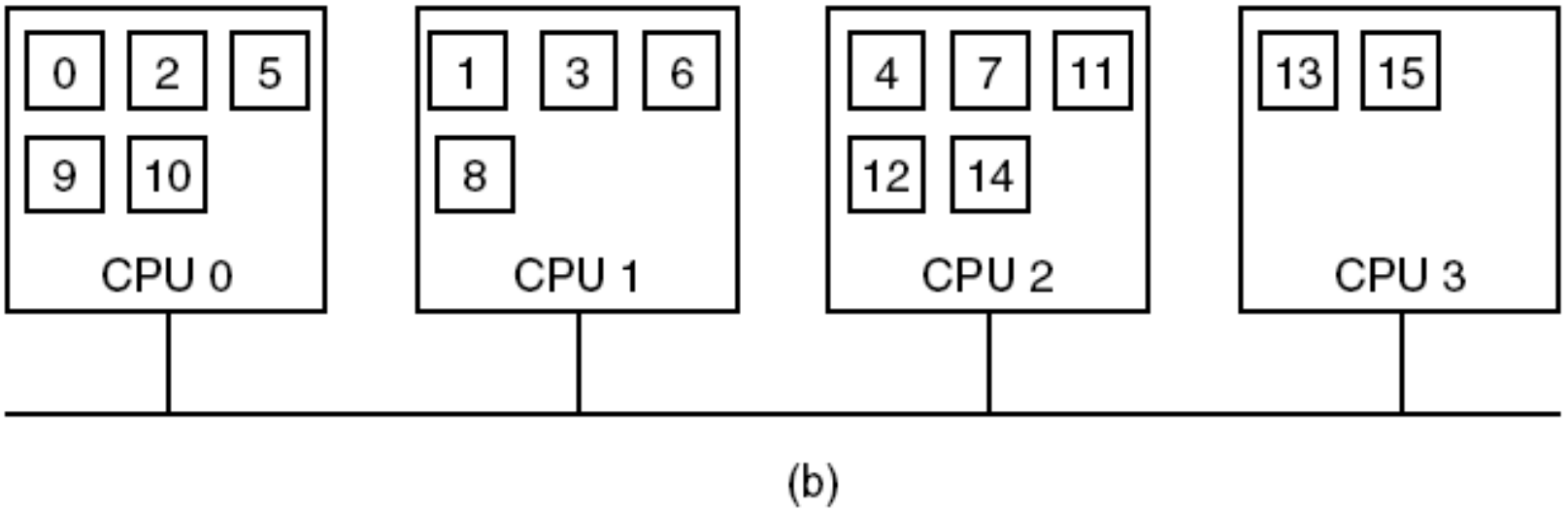


Figure 8-22. (b) Situation after CPU 1 references page 10 and the page is moved there.

Distributed Shared Memory (6)

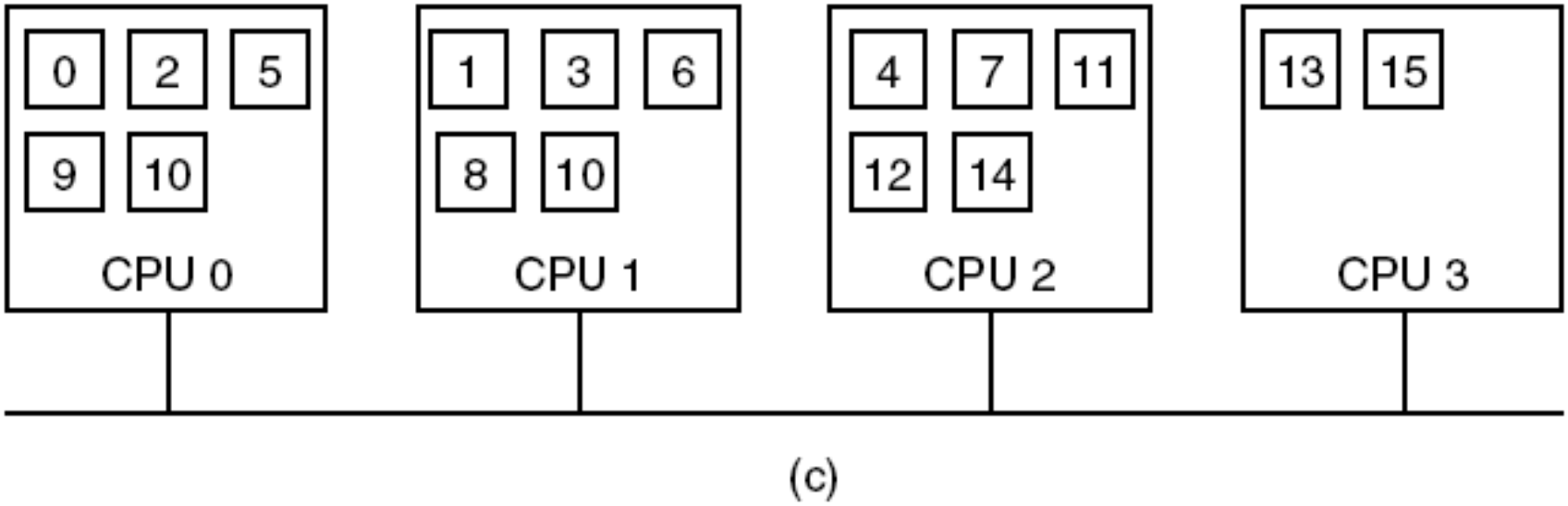


Figure 8-22. (c) Situation if page 10 is read only and replication is used.

False Sharing

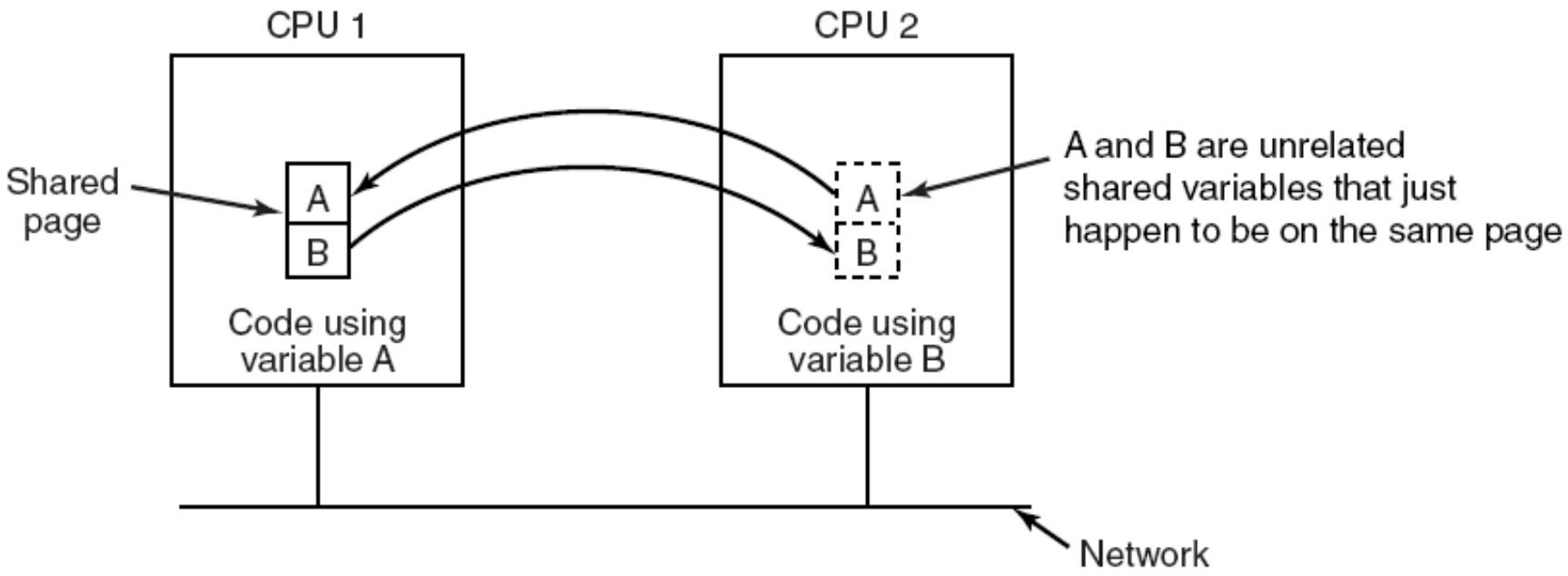


Figure 8-23. False sharing of a page containing two unrelated variables.

A Graph-Theoretic Deterministic Algorithm

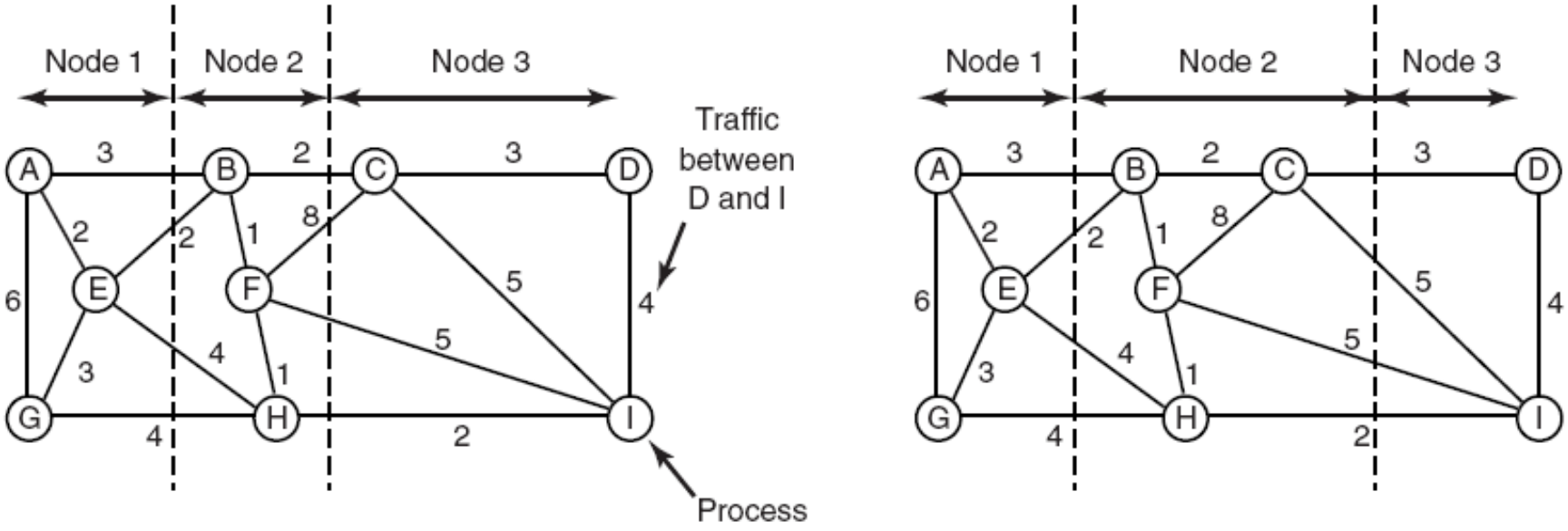


Figure 8-24. Two ways of allocating nine processes to three nodes.

A Sender-Initiated Distributed Heuristic Algorithm

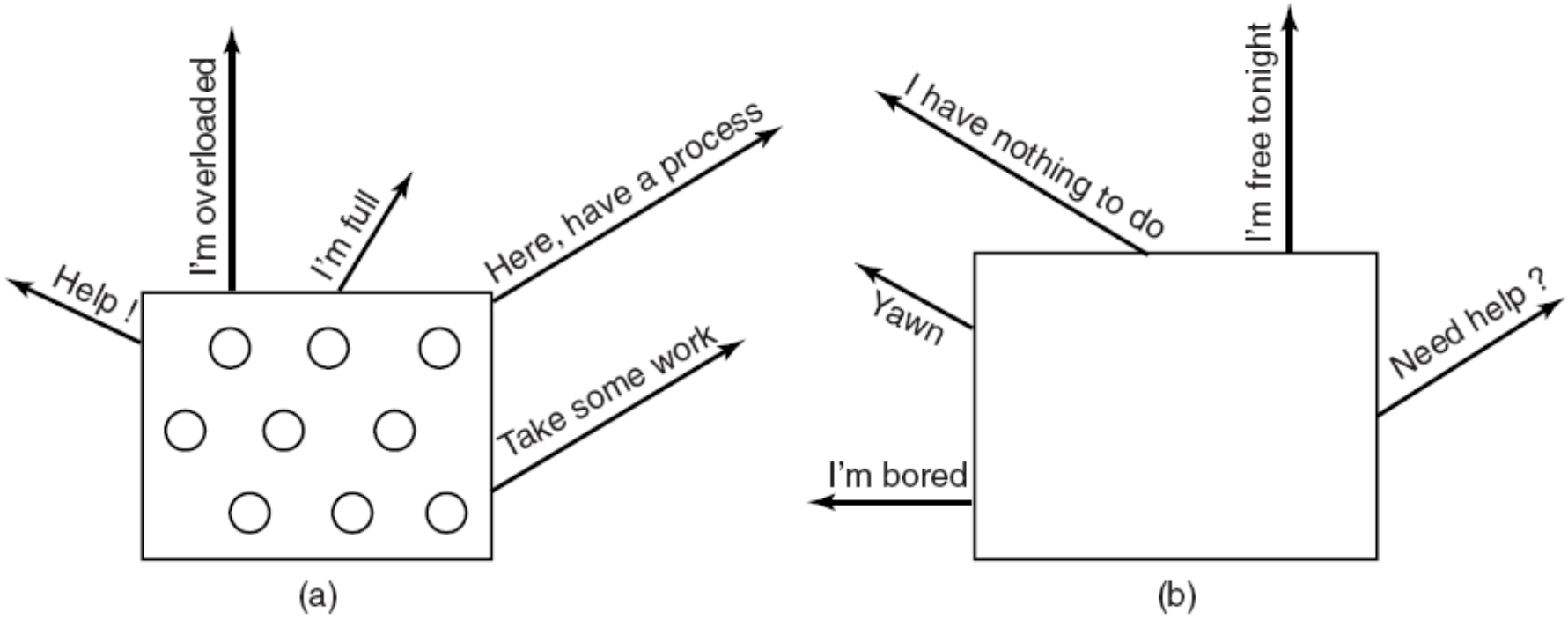


Figure 8-25. (a) An overloaded node looking for a lightly loaded node to hand off processes to. (b) An empty node looking for work to do.