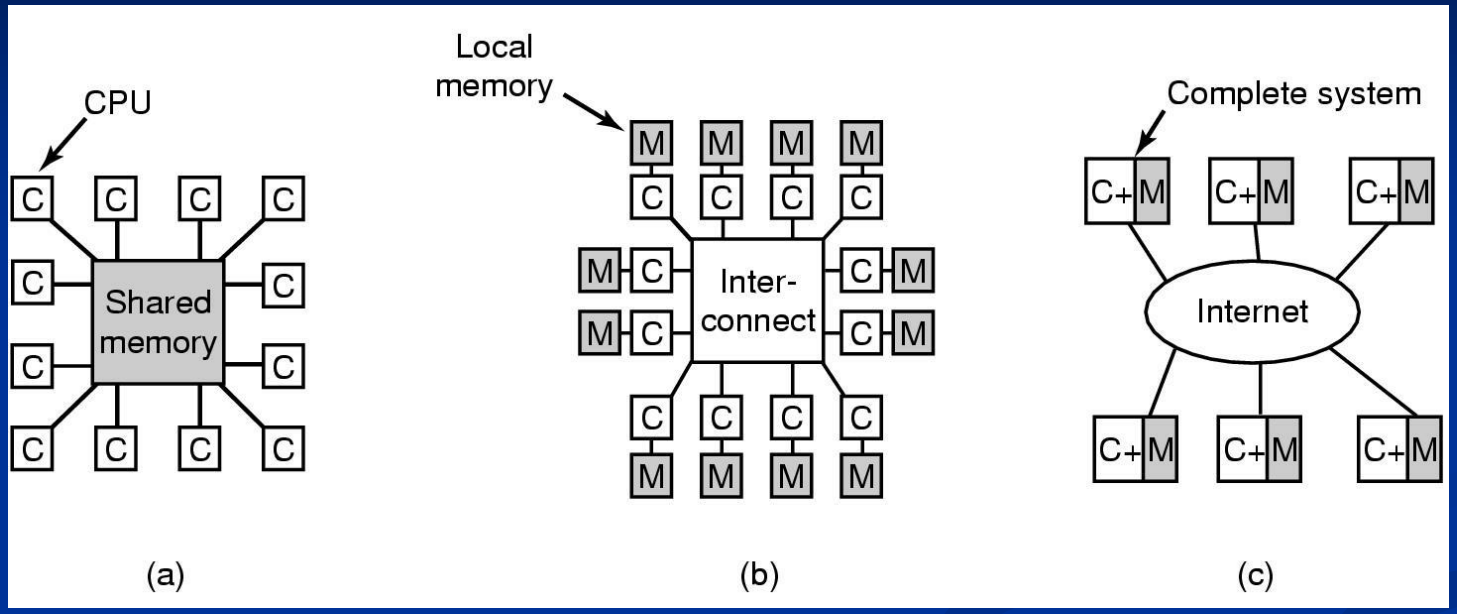
Multiple processor systems

Multiprocessor Systems



- Continuous need for faster computers
  - Multiprocessors:
    - shared memory model, access time nanosec (ns)
  - Multicomputers:
    - message passing multiprocessor, access time microsec (μs)
  - Distributed systems:
    - wide area distributed system, access time millisec (ms)

Multiprocessors

Definition:

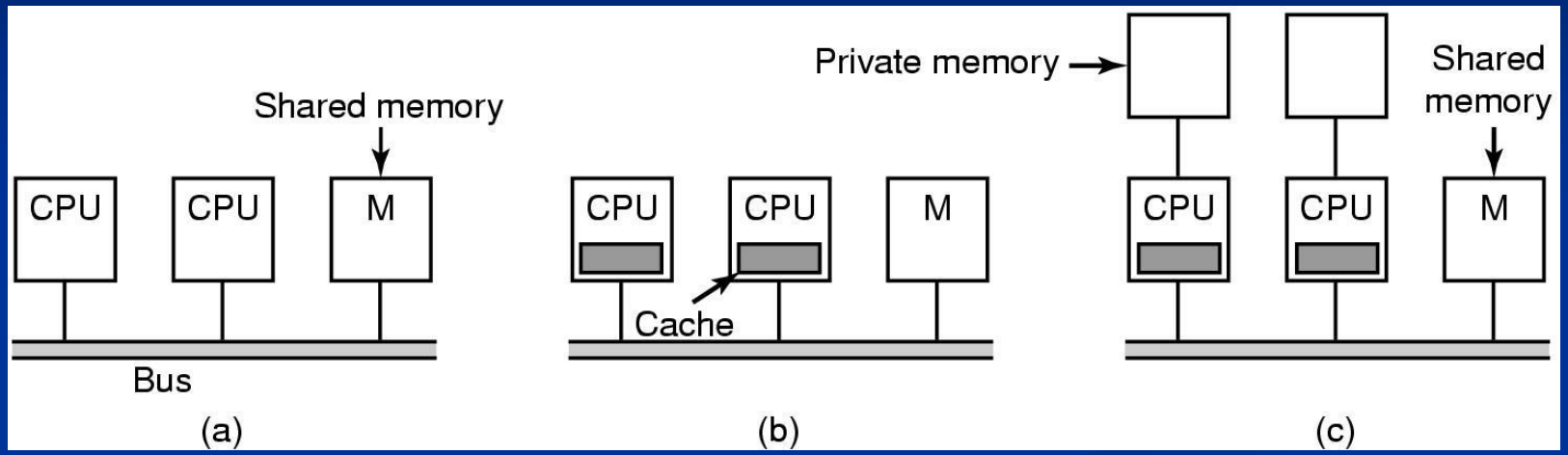
A computer system in which two or more CPUs share full access to a common RAM

Memory access:

UMA – Uniformed Memory Access

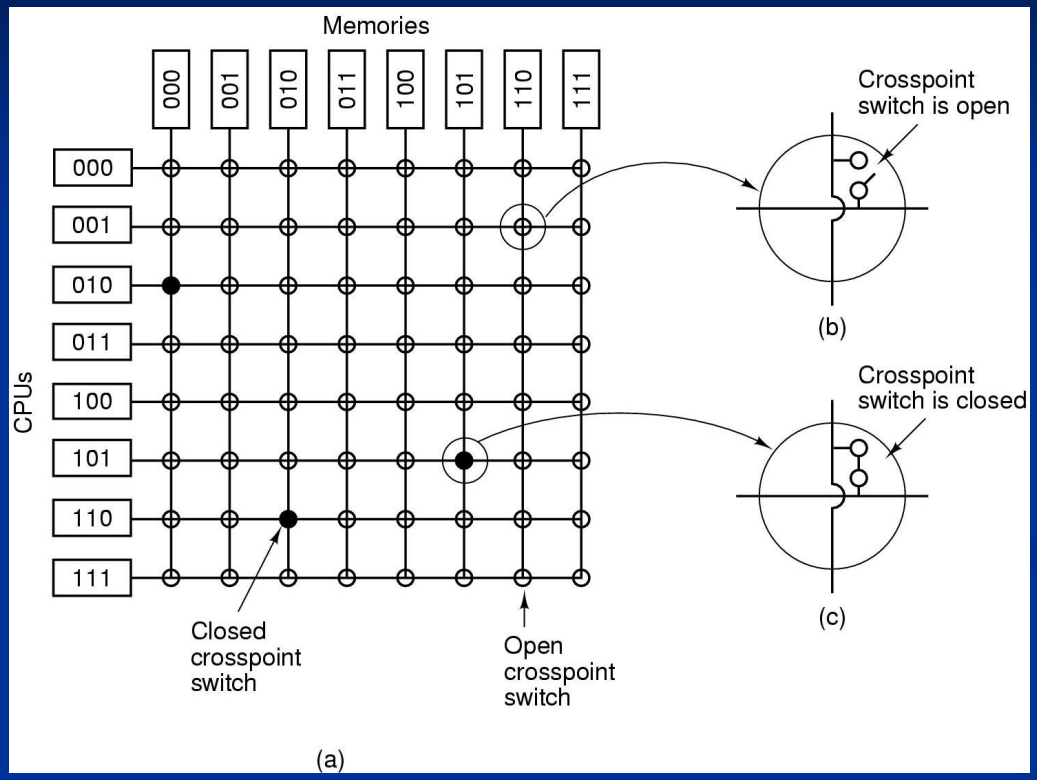
NUMA – Nonuniform Memory Access

Bus-Based UMA Multiprocessors



- Contention for the bus
  - Only one CPU can access the memory at any time
  - If bus is busy, requesting CPU must wait
  - Limits the number of CPU
- Internal cache reduces the memory accesses
  - Need: cache-coherence protocol (several copies exists)

UMA Multiprocessor using a crossbar switch

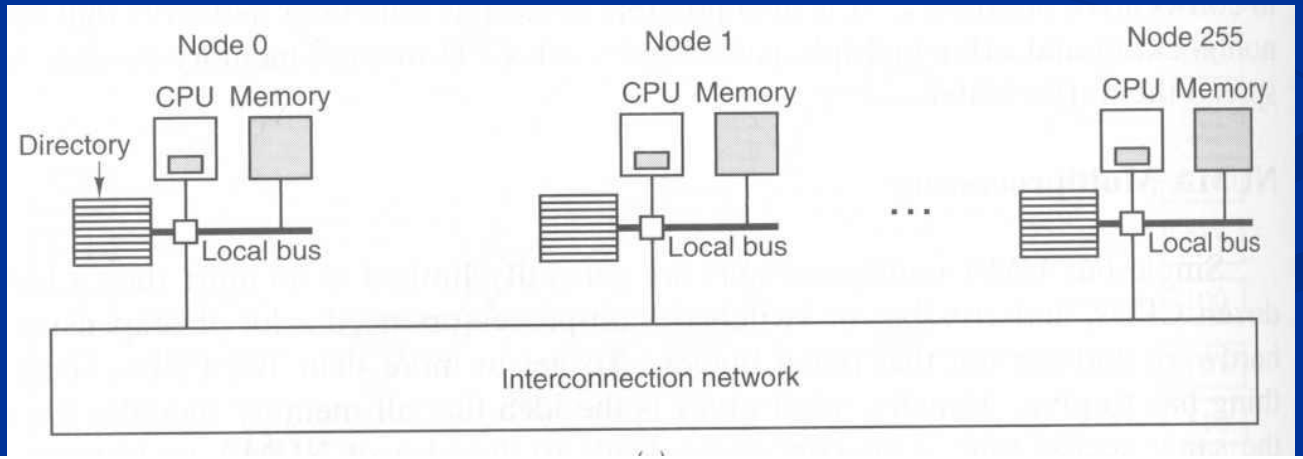


- Allows n CPUs to connect with k memories using n*k crosspoints
- Nonblocking network
  - CPU can always connect, assuming that memory is available

NUMA Multiprocessor

Characteristics

1. Single address space visible to all CPUs
2. Access to remote memory via commands
   - LOAD
   - STORE
3. Access to remote memory slower than to local



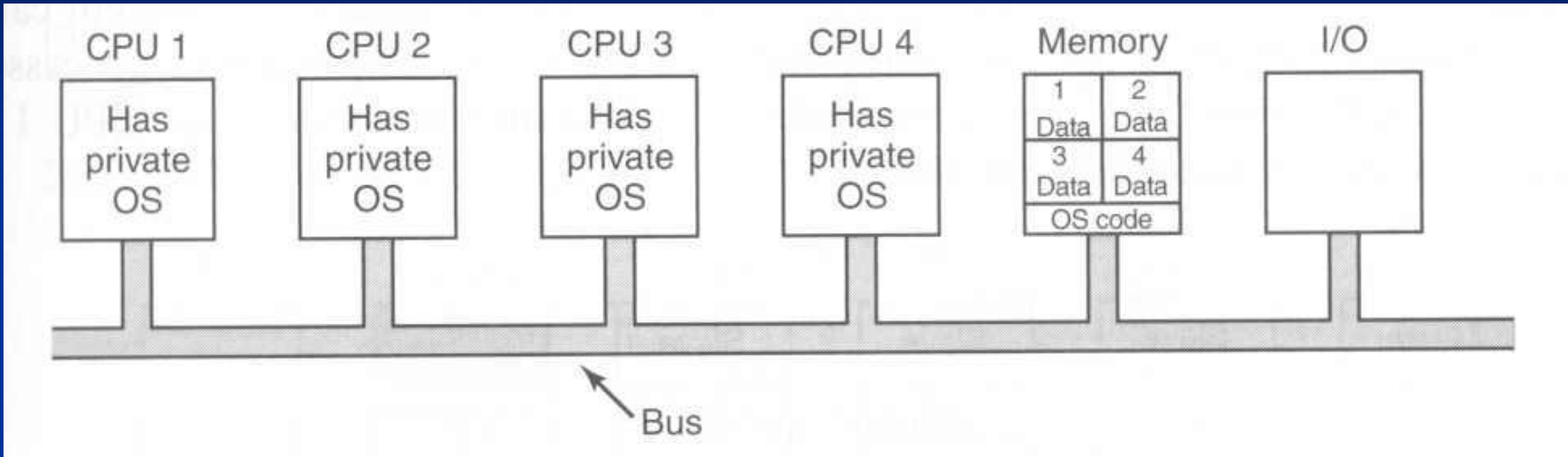
Multicore Cips

- ”What to do with all the transistors on the chip?”
  - Add cache -> after a point does not improve performance
  - Add a CPU, called core -> multicore chip
- May share cache or each has its own (or both)
- Failure in shared component can bring down more cores
- System on a ship –design has also special-purpose cores (for video, audio, crypto, network) with the CPUs
- For the software symmetric multicore chips are similar than UMA multiprocessors
- ”How to use them with the software?”
  - Parallel coding, lack of algorithms
  - Synchronization, race conditions, deadlocks
  - Benefits

Multiprocessor OS Types

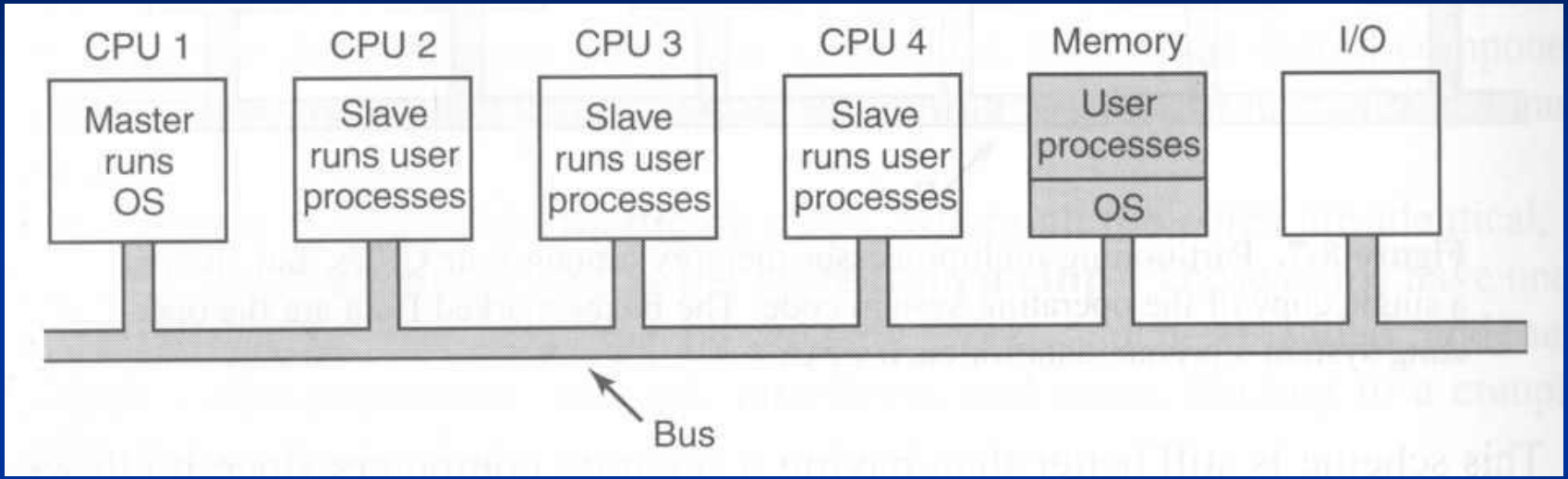
- Each CPU has its own Operating System
  - Partition the memory for private use
  - Each CPU and its OS operate independently
- Master-Slave Multiprocessors
  - One CPU is the master and others are slaves
  - Only master may run Operating System
- Symmetric Multiprocessors (SMP)
  - One copy of the OS and any CPU may run it
  - Balances processes and memory dynamically
  - Synchronization issues in the OS code itself
    - Critical regions must be protected (disabling interrupts is not enough)

Each CPU has its own OS



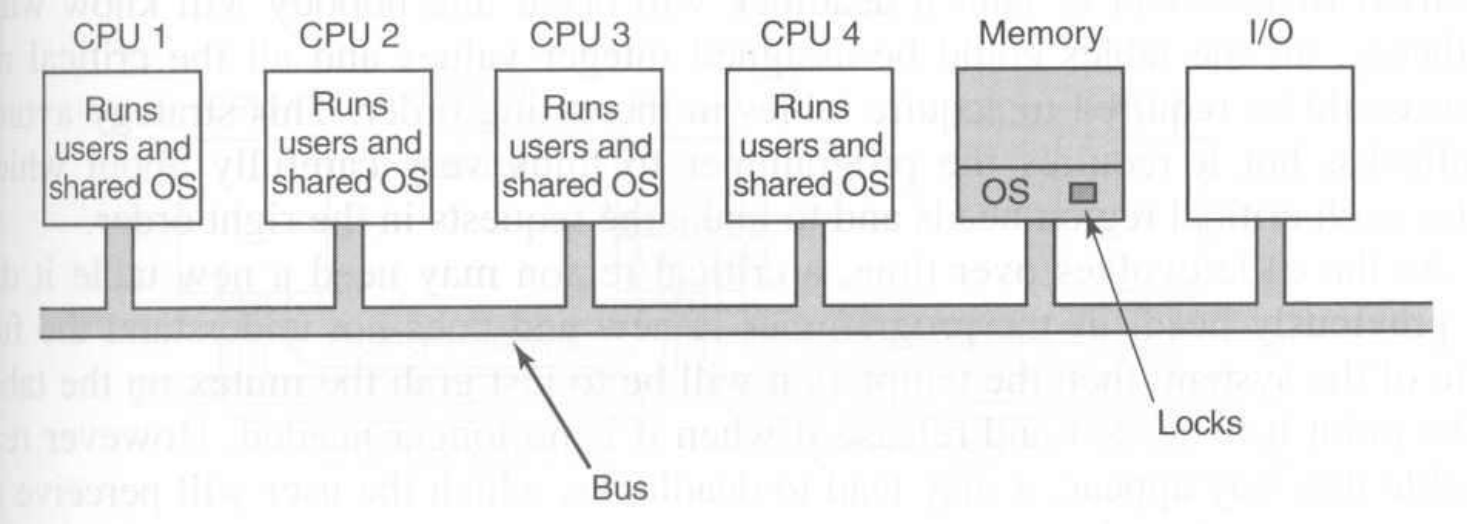
- System call handled by the private OS using its own private data structures
- No sharing of processes, no load balancing
- No sharing of memory pages, no reallocation of free pages
- No buffer cache for shared file systems to avoid inconsistencies
- No used any more

Master-Slave multiprocessors



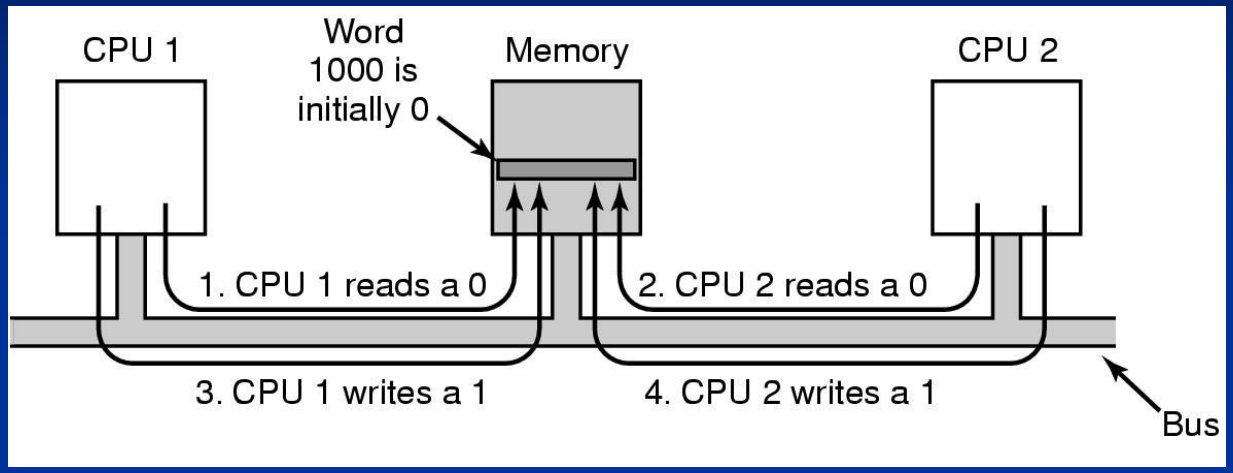
- Only one instance of OS and its data structures
- Master allocates load (processes/threads) to other CPUs
- Shared memory, pages can be allowed for all processes
- Shared buffer cache, inconsistencies do not occur
- Master may become bottleneck!
  - It executes all system calls for all processes

Symmetric Multiprocessors, SMP



- One copy of the OS, any CPU can run it
  - All CPUs can execute system calls
  - Access to critical regions must be controlled
    - Mutexes to control access to multiple independent critical regions
    - Mutexes to cotrol access to critical tables (used in several critical regions)
    - Only one CPU can access a particular part of OS at any time!
    - Deadlocks might freeze the system

Multiprocessor Synchronization



Test and Set Lock (TSL)
-instruction can fail if bus cannot be locked

- Correct mutex implementation is not simple!
  - Disabled interrupts do not work with multiple processors
  - Test and Set Lock without locking the bus does not work (see fig above)
  - Test and Set with logging the bus before command and releasing it after works
  - The waiting CPU spins (loops in testing) fast waiting for the spin lock
  - Instead of spinning the CPU could switch to another thread

Spinning versus switching

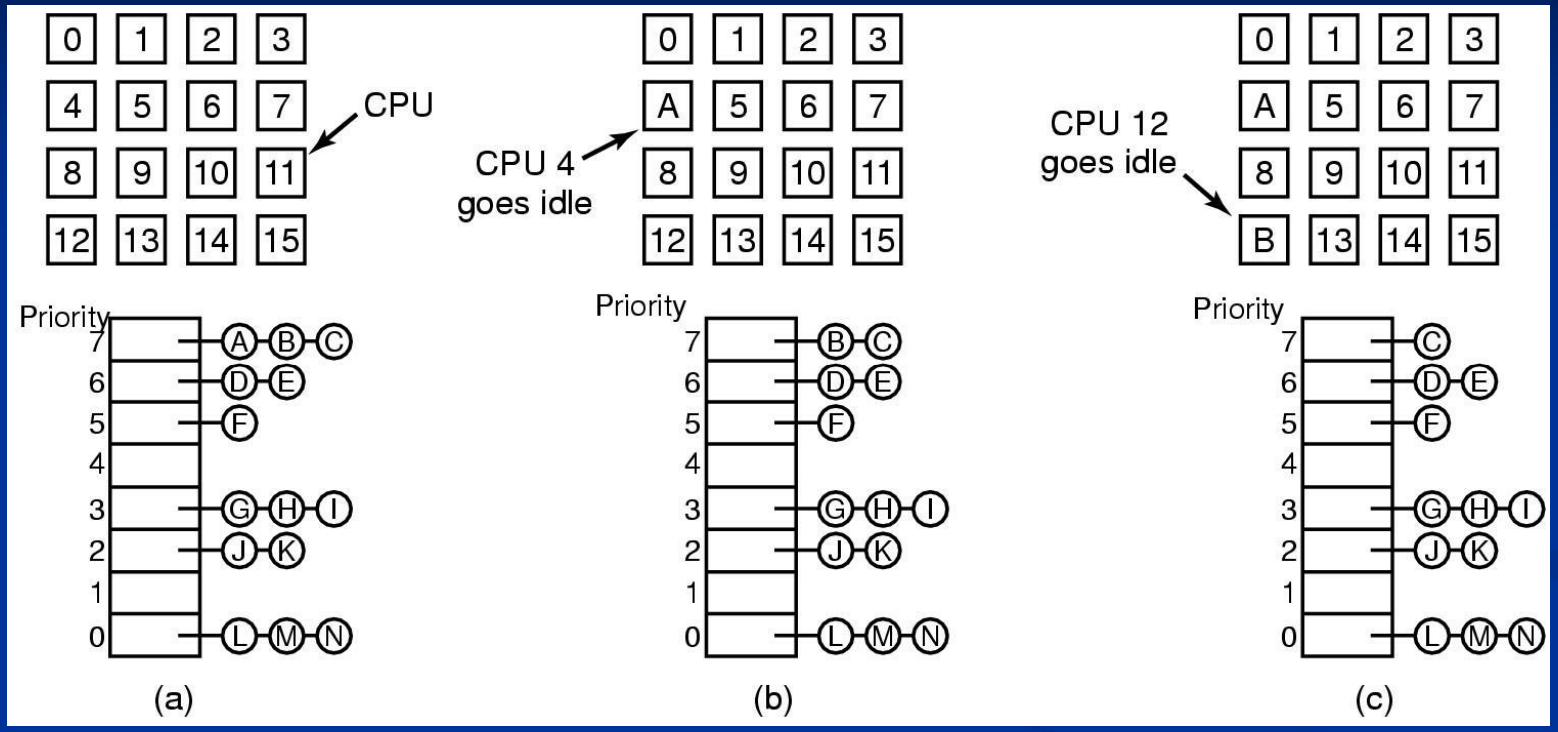
- In some cases CPU must wait
  - waits to acquire ready list
- In other cases a choice exists
  - spinning wastes CPU cycles
  - switching uses up CPU cycles also
  - could be possible to make separate decision each time locked mutex encountered

Multiprocessor scheduling

- What is scheduled?
  - User-level threads → OS schedules processes
  - Kernel-level threads → OS schedules threads
- Where to run it?
  - Which CPU
- Schedules thread independently or in groups?

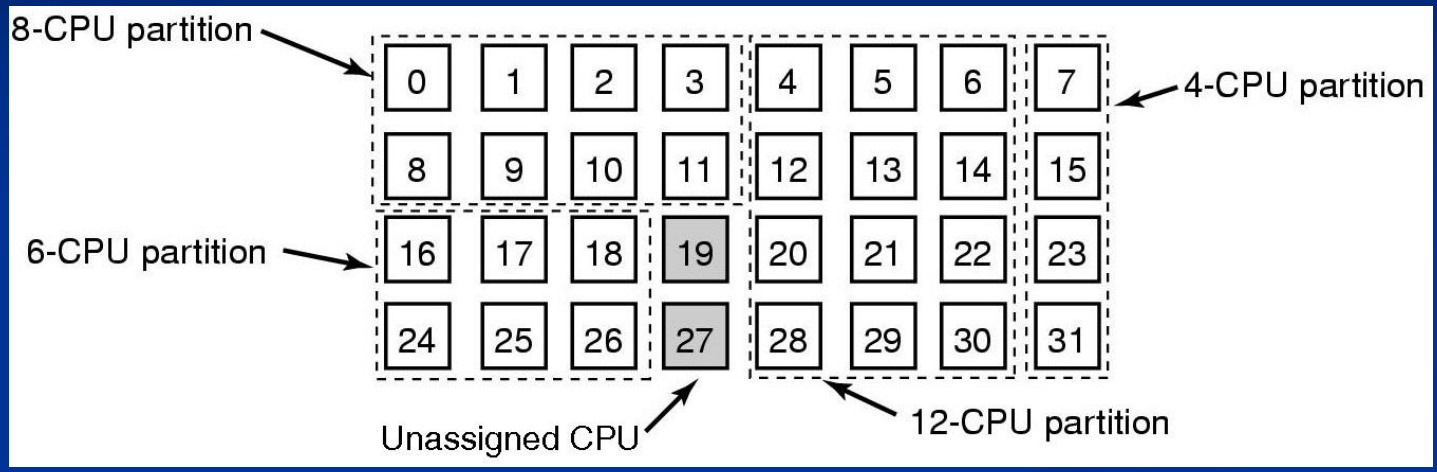
- Timesharing
- Space sharing
- Gang scheduling

Timesharing



- Single scheduling data structure for CPUs
- Automatic load balancing
- Smart scheduling : thread in critical region is not switched off
- Affinity scheduling: make an effort to have thread run on the same CPU again

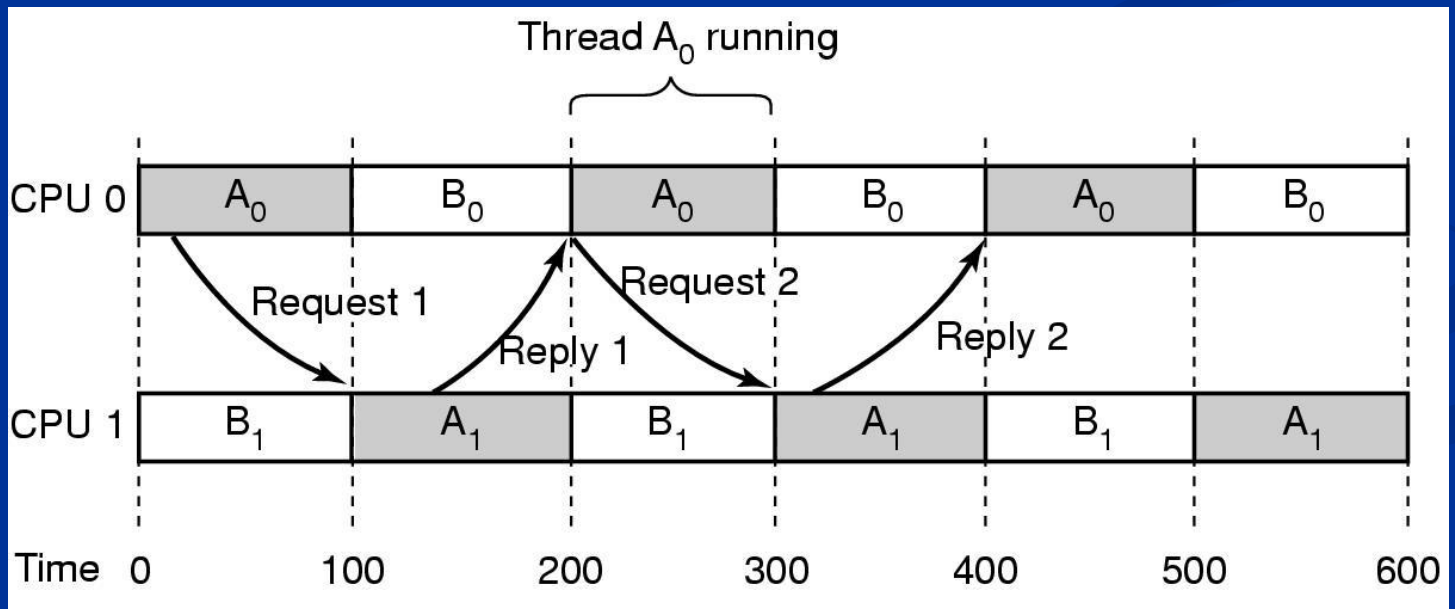
Space sharing



- Multiple related threads at same time across multiple CPUs
- Simple model:
  - All threads are allocated CPUs at the same time
  - They hold on the CPU even while waiting for I/O
  - Release only when thread is finished
- More complex alternatives exist

Multiprocessor Scheduling (4)

- Problem with threads in figure:
  - Belong to same thread A, need to communicate,
  - but run in different phases, makes A slow!
- Solution: Gang Scheduling



Gang Scheduling

Time slot	CPU 0	1	2	3	4	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

A's threads
B & C
D, E0
Rest of E
repeat

- Groups of related threads scheduled as a unit (a gang)
- All members of gang run simultaneously on diff. timeshared CPU
- All gang members start and end time slices together

Multicomputers

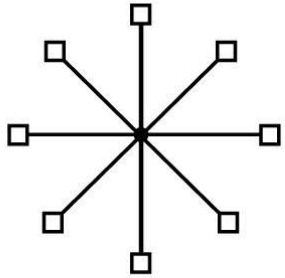
- Definition:
  Tightly-coupled CPUs that do not share memory

- Also known as
  - cluster computers
  - clusters of workstations (COWs)

Multicomputers

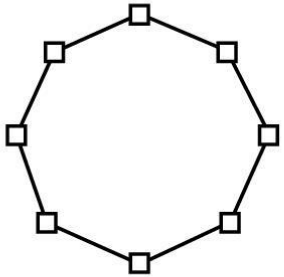
- Hardware
  - Topology
  - Communication
  - Network interfaces
- Communication software
  - Low-level:from memory to network
  - User-level: send & receive, blocking vs. Nonblocking
  - Remote-Procedure Call, RPC
  - Distributed shared memory
- Scheduling & load balancing

Interconnection Topology



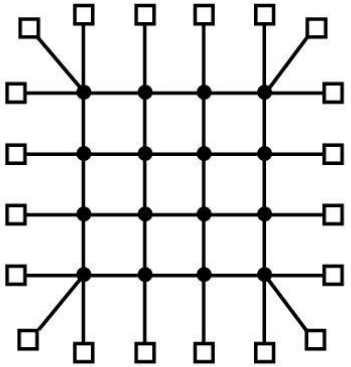
(a)

Single switch



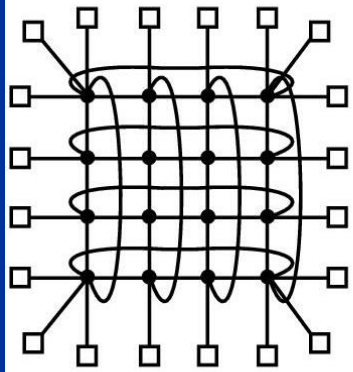
(b)

Ring



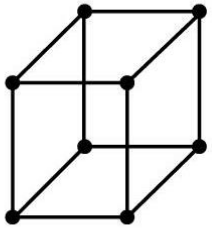
(c)

Grid



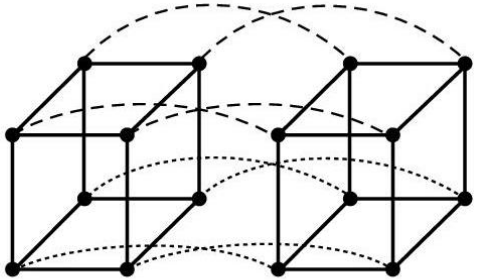
(d)

Double torus



(e)

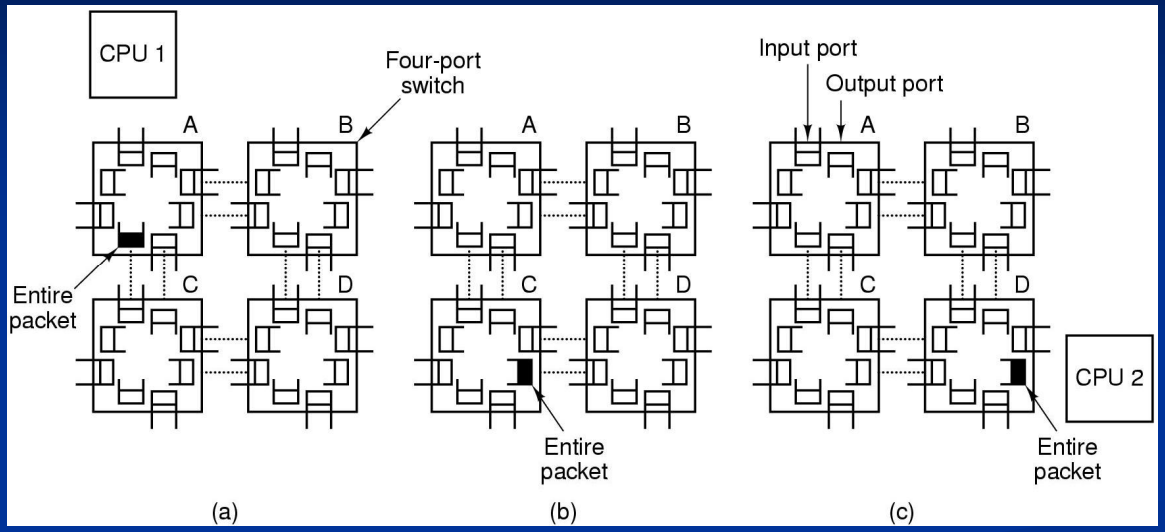
Cube



(f)

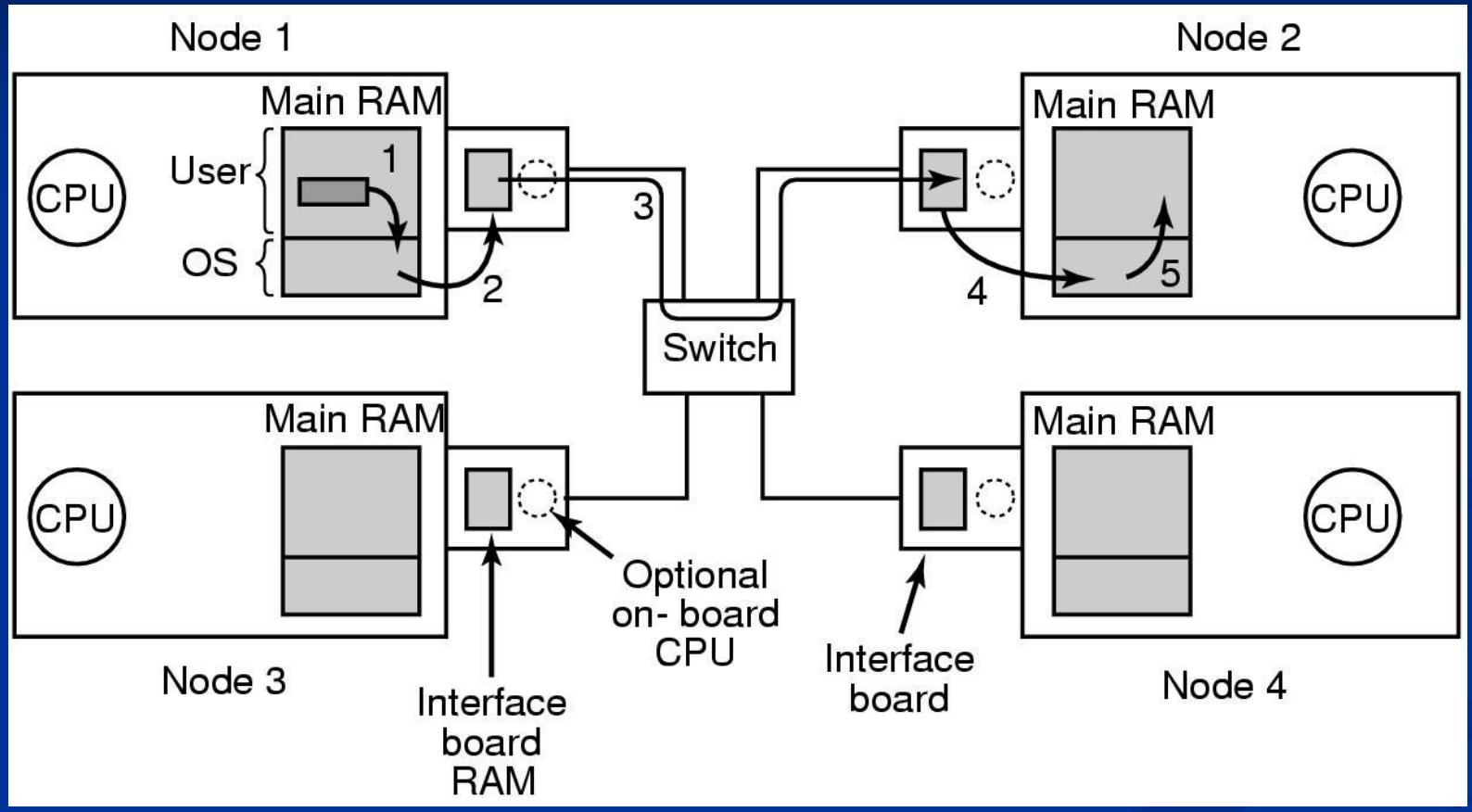
4D hypercude

Communication



- Store-and-forward
  - Each switch that receives the packet, forwards to next one
  - Flexible and efficient, Latency grows with size
- Circuit switching
  - Path established first during setup phase
  - All packets travel the same path

Low-level communication

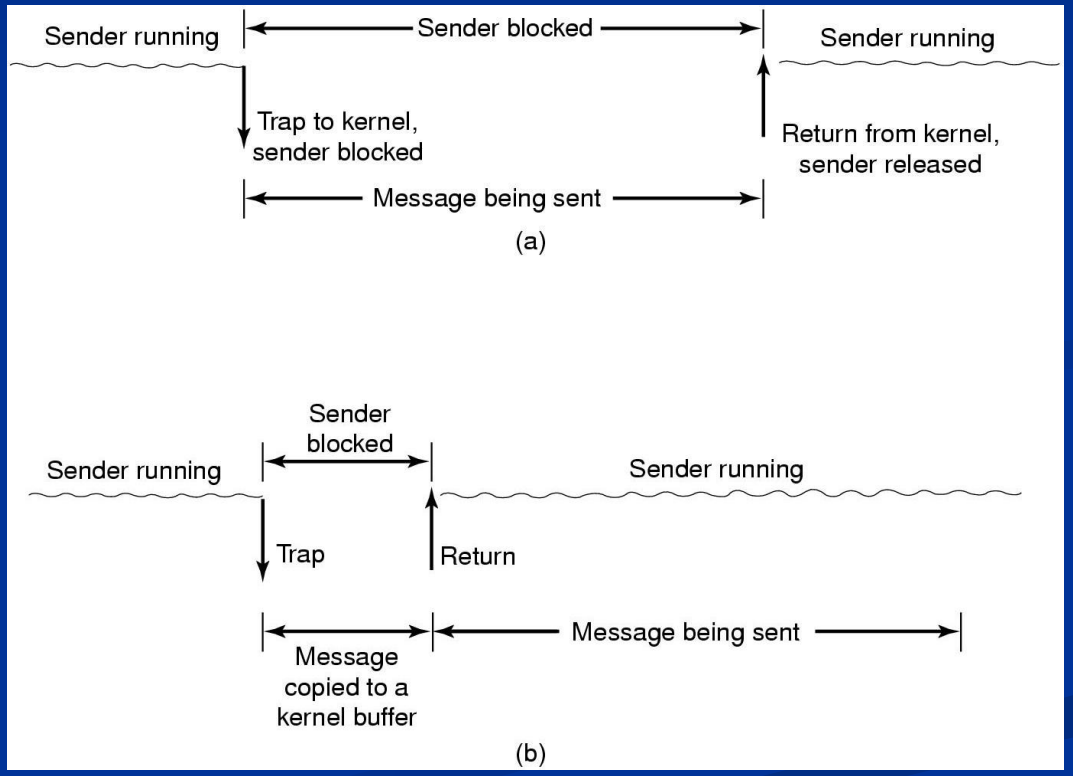


Packets copied several times during transmission

User Level Communication Software

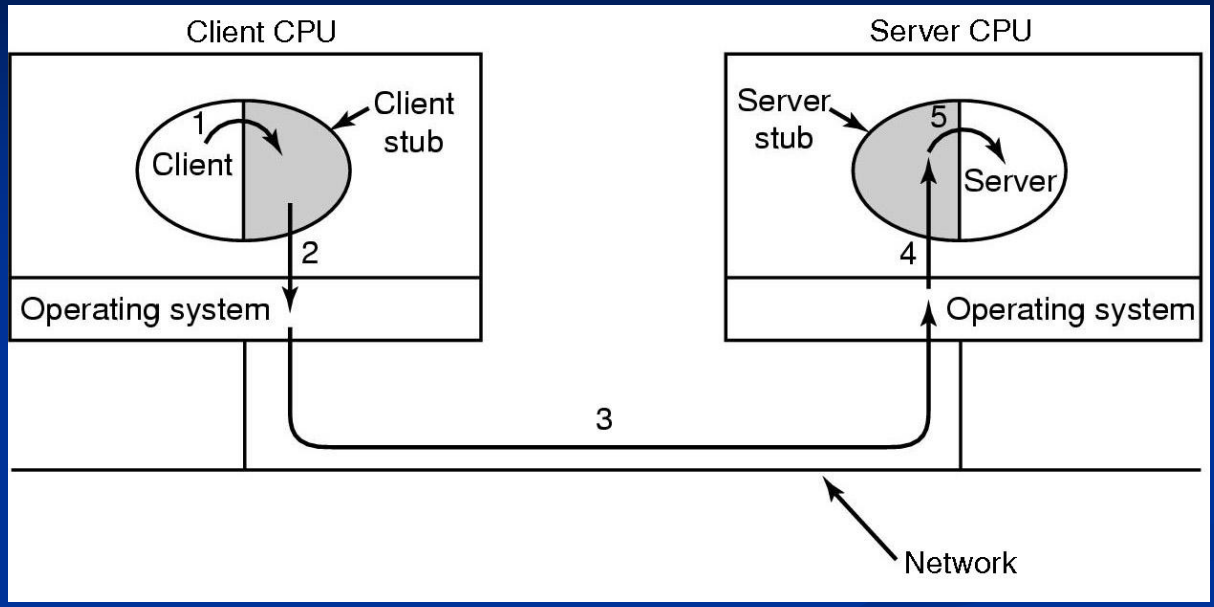
- Minimum services provided
  - send and receive commands
- These are blocking (synchronous) calls

(a) Blocking send call



(b) Nonblocking send call

Remote Procedure Call (1)



- Client makes (normal) procedure call
- Client stub marchalls the parameters for network
- Network passes the request to server
- Server stub unmarhalls the parameters
- and makes a normal procedure call on server

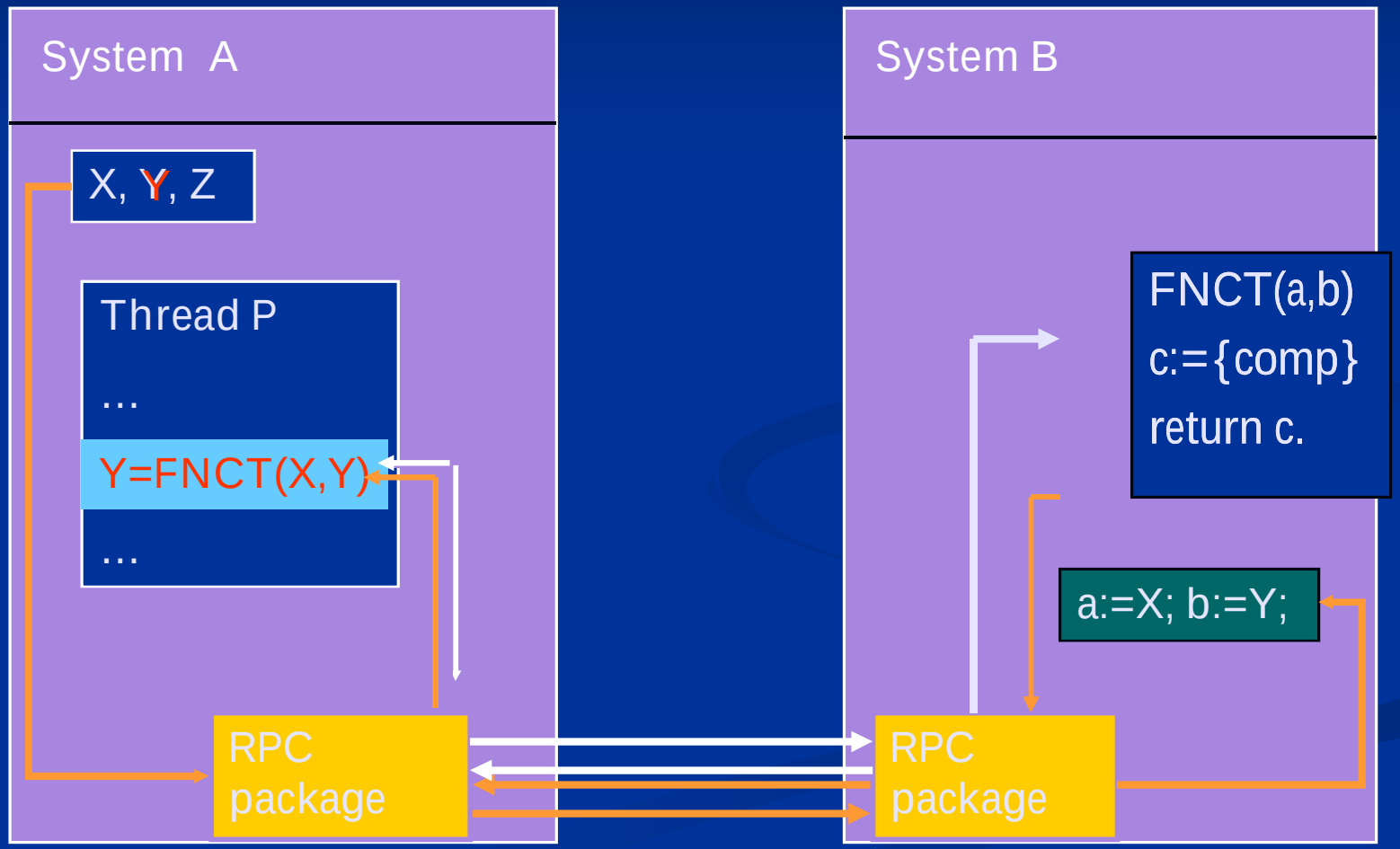
Remote Procedure Call (2)

Implementation Issues

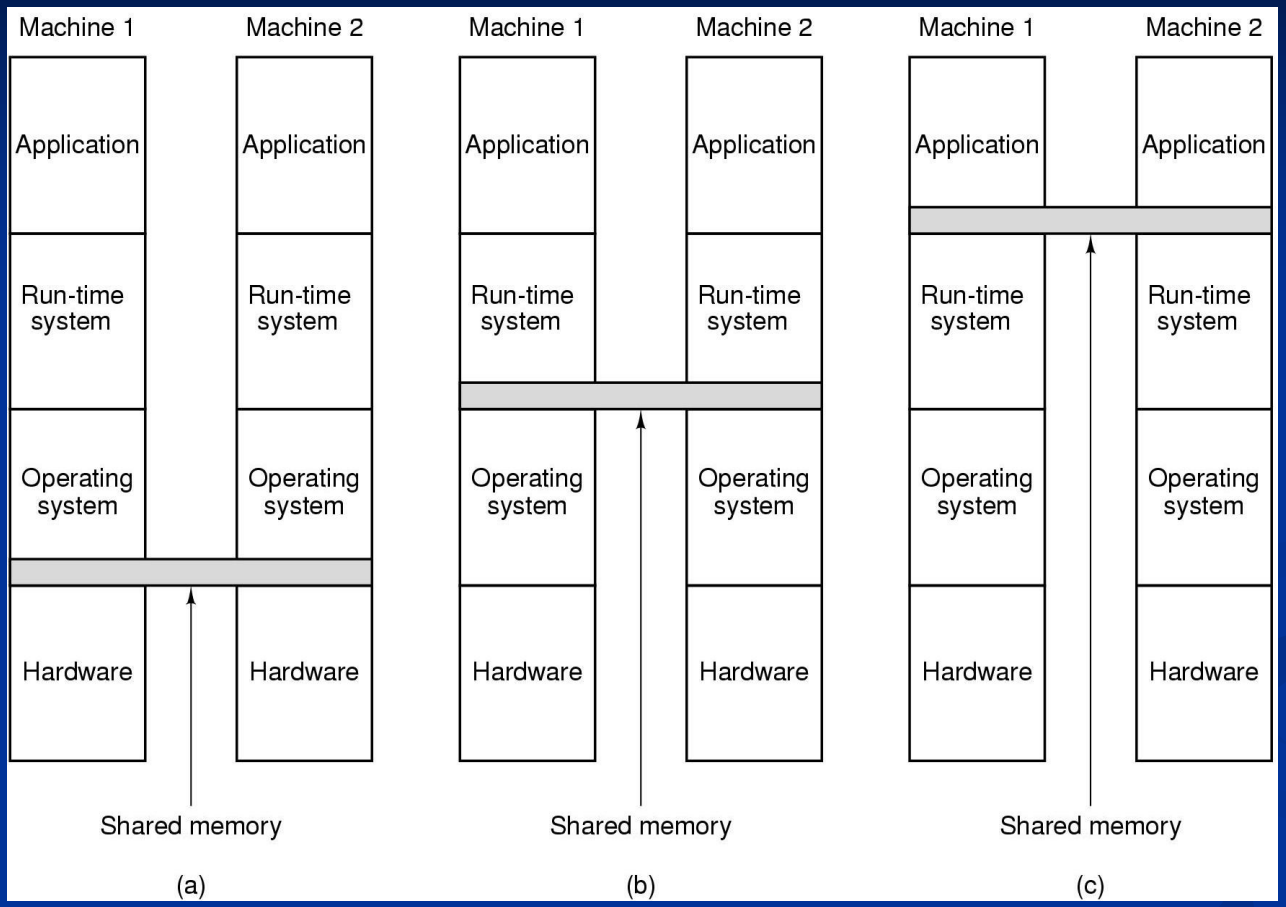
- Cannot pass pointers
  - call by reference becomes copy-restore (but might fail)
- Weakly typed languages
  - client stub cannot determine size
- Not always possible to determine parameter types
- Cannot use global variables
  - may get moved to remote machine

- RFC 1057
- RPC v2 : RFC 1831

RPC: a Schematic View



Distributed Shared Memory

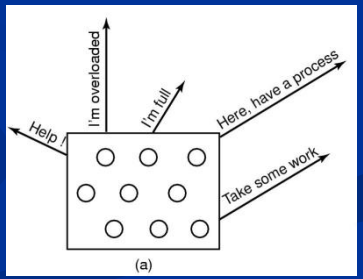


- Can be implemented on
  - hardware
  - operating system
  - user-level software

Scheduling & load balancing

- Each node has its own set of processes
  - Local scheduling decisions
  - Global decision on allocation of processes
- Gang scheduling over multicomputers is possible

- Processor allocation algorithms (for load balancing)
  - A graph-theoretic deterministic algorithm
  - A sender-initiated distributed heuristic algorithm
    - Overloaded node
  - A receiver-initiated distributed heuristic algorithm
    - Underloaded node



Virtualization

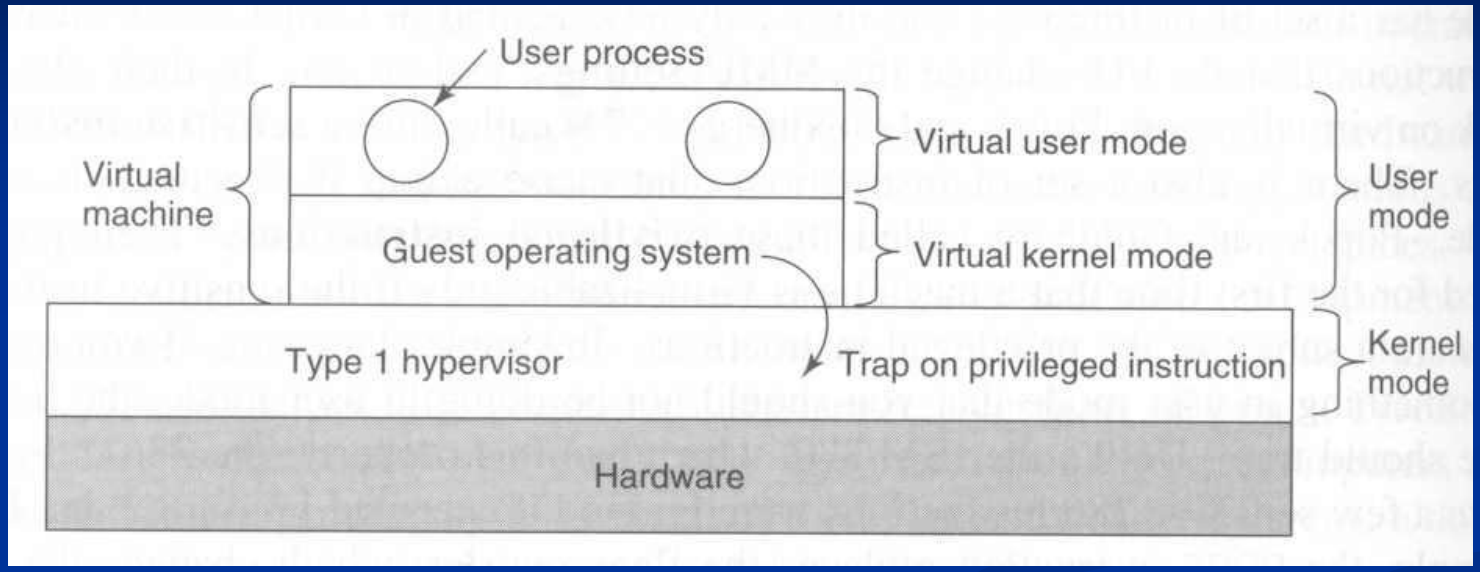
Background and motivation

- Consolidating servers
  - Each service in its own virtual machine
    - Still not dependent on others
    - Each has its own OS, libraries, configuration files
    - Can fail independently, no effect to other virtual machines
  - Fewer physical machines
    - Reduced hardware and energy costs
    - Hardware (or hypervisor) failure fails all services on that server
- Other benefits:
  - Checkpointing and migrating straighforward (memory image)
  - Running lecacy systems (no hardware available any more)
  - Software development (testing on several OS)

Requirements

- Virtual machine must act just like the real hardware
  - Booting the machine, installing operating system, etc.
  - Hypervisor provides this illusion
  - Hypervisor emulates the hardware by ”interpreting” the machine code instructions
- Hardware support necessary for type 1 hypervisors
  - Priviledges instructions (trap if run on user mode)
  - Sensitive instructions (executed only in kernel mode)
  - Virtualizable only wen sensitive instr. are a subset of priv.
  - Intel 386 not virtualizable:
    - Some sensitive instructions ignored in user mode
    - Some instructions can read sensitive data without trap

Type 1 hypervisor



- Execution of priviledged instructions
  - Hardware detects!
  - Trap to hypervisor
  - Check whether the instruction for guest OS or user applic.
    - Execute the instruction from OS
    - Emulate the actual behavior for the user applications

Type 2 hypervisor

- No hardware support needed
- Hypervisor itself runs in user mode
- Executing program (or OS) in virtual machine in done using binary translation
  - First, hypervisor scans the code for basic blocks without any jumps etc changing the program counter
  - Replace each sensitive instruction and the last instructionn of each block with call to the hypervisor's own procedure
  - Modified block cached within the hypervisor and executed
- Binary translation can be used on type 1 hypervisors also to avoid traps

Paravirtualization

- Use only modified guest operating system
  - Do the modifications (done in binary translation) already to the code of the guest os
  - The modified OS cannot be anymore run directly on the hardware on its own, it needs the hypervisor
- Create a special Virtual Machine Interface
  - The implementation under the interface can change (VMILinux using VMIL)

VMI – Virtual Machine Interface

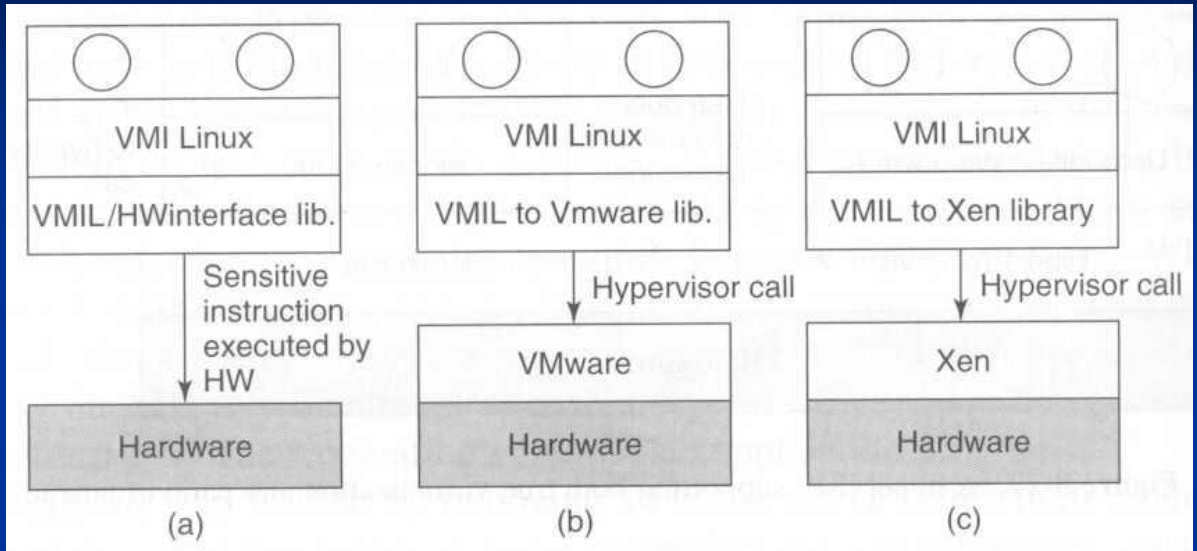


Figure 8-28. VMI Linux running on (a) the bare hardware (b) VMware (c) Xen.

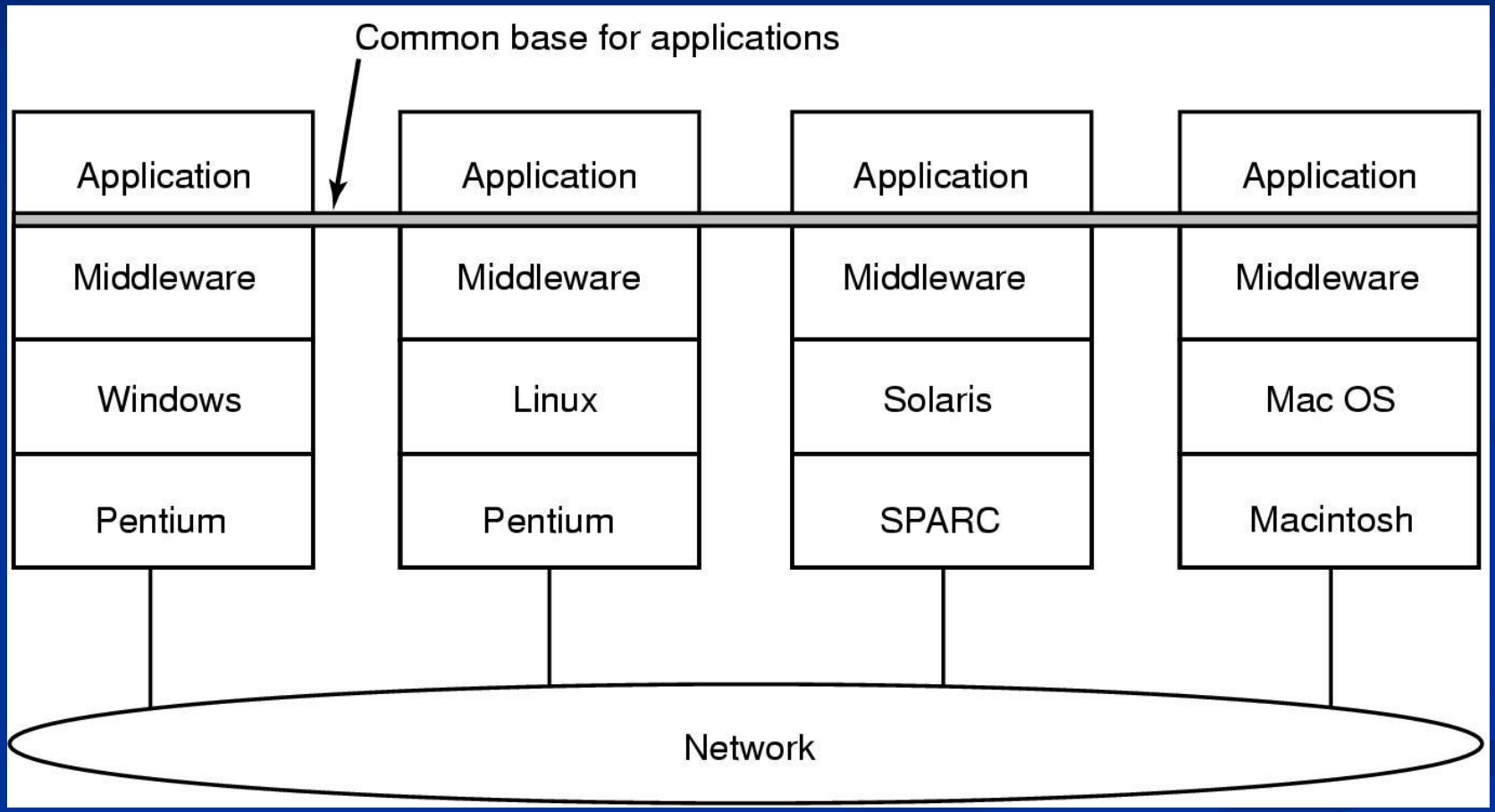
- Modified kernel calls procedures of VMI for any sensitive operations
- The VMI form a low-level layer interface with hardware or hypervisor (just change the procedure implementations)

Distributed Systems

Item	Multiprocessor	Multicomputer	Distributed System
Node configuration	CPU	CPU, RAM, net interface	Complete computer
Node peripherals	All shared	Shared exc. maybe disk	Full set per node
Location	Same rack	Same room	Possibly worldwide
Internode communication	Shared RAM	Dedicated interconnect	Traditional network
Operating systems	One, shared	Multiple, same	Possibly all different
File systems	One, shared	One, shared	Each node has own
Administration	One organization	One organization	Many organizations

- The difference between multicomputer and distributed system depends on the viewpoint
- Distributed systems are loosely coupled
- Covered in our Distributed Systems -course

Distributed Systems



Achieving uniformity with middleware

The Internet