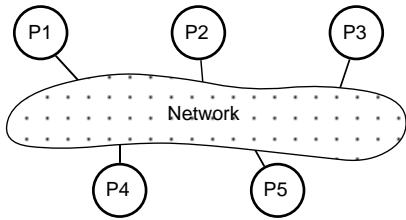## What is a Distributed System?

- A *distributed system* is a set of physically separate processors connected by one or more communication links



  - Workstation = computer = machine = processor = host = site = node

- Is every system with >2 computers a distributed system??

  - File server, printer server, web server

  - Beowulf-style cluster of workstations

  - 16-processor Cray SV1 at OSC

  - How does a distributed system differ from a parallel system?

---

## Two Taxonomies for Classifying Computer Systems

- Michael Flynn (1966)

  - SISD — single instruction, single data

  - SIMD — single instruction, multiple data

  - MISD — multiple instruction, single data

  - MIMD — multiple instruction, multiple data
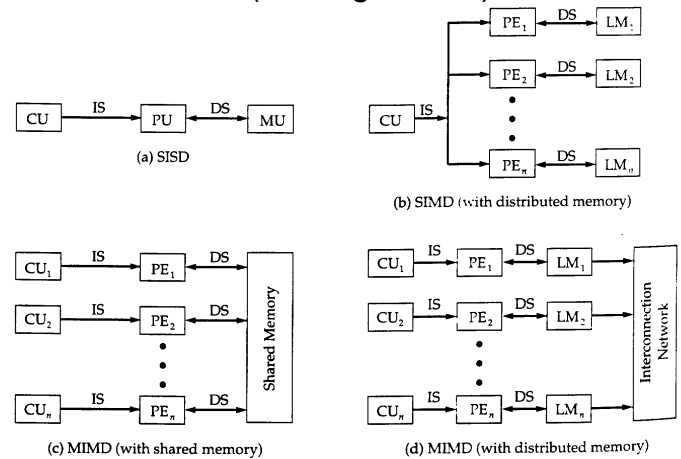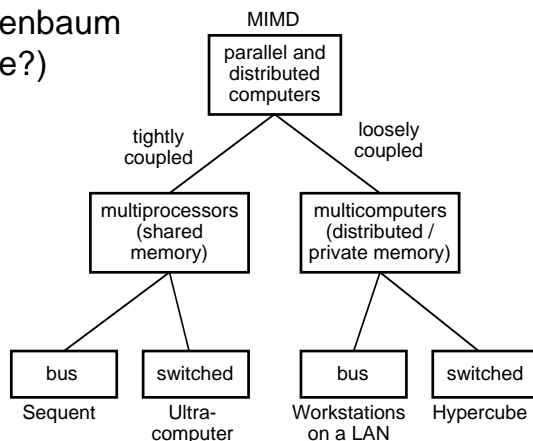
- More recent (Stallings, 1993)



**FIGURE 16.16. Alternative Computer Organizations**

---

## Classification of MIMD Architectures

- Tanenbaum (date?)



- Tightly coupled ≈ *parallel processing*

  - Processors share clock and memory, run one OS, communicate frequently

- Loosely coupled ≈ *distributed computing*

  - Each processor has its own memory, runs its own OS (?), communicates infrequently

---

## Distributed Operating System

- Loosely-coupled hardware

  - No shared memory, but provides the "feel" of a single memory

- Tightly-coupled software

  - One single OS, or at least the feel of one

- Machines are somewhat, but not completely, autonomous
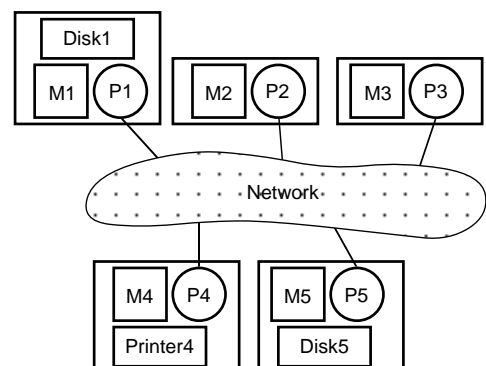
## Why Use Distributed Systems? What are the Advantages?

- Natural programming model
  - Some applications (database in large company) are inherently distributed

- Resource sharing
  - Expensive (scarce) resources need not be replicated for each processor

- Price / performance
  - Network of workstations provides more MIPS for less $ than a mainframe does

- Reliability
  - Replication of processors and resources yields fault tolerance

- Scalability
  - Modular structure makes it easier to add or replace processors and resources

## Central Coordinator

- To enter the critical section, a thread sends a *request* message to the central coordinator, and waits for a reply

- When the coordinator receives a request:
  - If **no** other thread is in the critical section, it sends back a *reply* message
  - If another thread **is** in the critical section, the coordinator adds the request to the tail of its queue, and does not respond

- When the requesting thread receives the *reply* message from the coordinator, it enters the critical section
  - When it leaves the critical section, it sends a *release* message to coordinator
  - When the coordinator receives a *release* message, it removes the request from the head of the queue, and sends a *reply* message to that thread

## Central Coordinator (cont.)
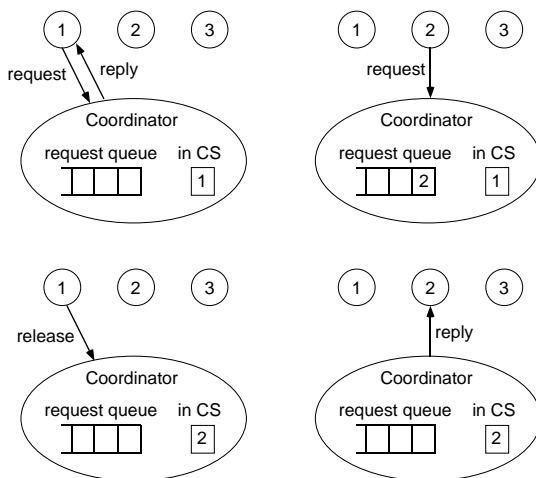


- Evaluation:
  - 3 messages required to enter CS
    - release, request, reply
  - ✗ Coordinator is a performance bottleneck
  - ✗ Coordinator is a single point of failure
  - ✗ Delay is unconstrained

## Lamport's Algorithm (1978)

- Each process maintains a request queue, ordered by timestamp value

- Requesting the critical section (CS):
  - When a thread wants to enter the CS, it:
    - Adds the request to its own request queue
    - Sends a timestamped *request* message to all threads in that CS's request set
  - When a thread receives a *request* message, it:
    - Adds the request to its own request queue
    - Returns a timestamped *reply* message

- Executing the CS:
  - A thread enters the CS when **both**:
    - Its own request is at the top of its own request queue (its request is earliest)
    - It has received a *reply* message with a timestamp larger than its request from all other threads in the request set

## Lamport's Algorithm (cont.)

■ Releasing the CS:

- ● When a thread leaves the CS, it:
  - ■ Removes its own (satisfied) request from the top of its own request queue
  - ■ Sends a timestamped *release* message to all threads in the request set

- ● When a thread receives a *release* message, it:
  - ■ Removes the (satisfied) request from its own request queue
  - ■ (Perhaps raising its own message to the top of the queue, enabling it to finally enter the CS)
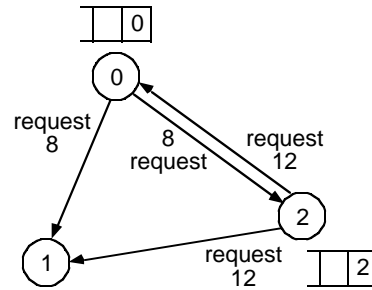
■ Evaluation:

- ● 3(N–1) messages required to enter CS
  - ■ (N–1) release, (N–1) request, (N–1) reply

- ✗ Later…
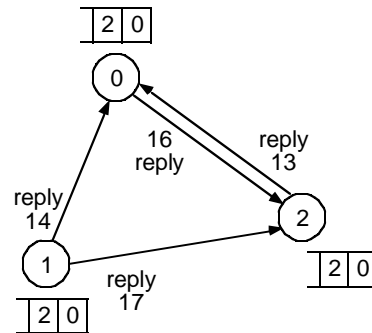
## Lamport's Algorithm (cont.)

■ Both threads 0 and 2 request the CS:



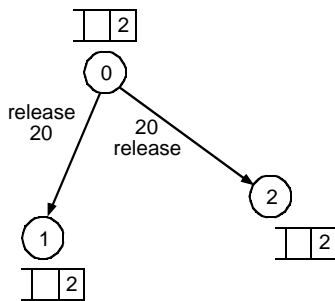■ Everyone replies, thread 0 enters the CS since its request was first:

## Lamport's Algorithm (cont.)

■ Thread 0 releases the CS, thread 2 enters it: