Distributed Computing
CS 111
Operating Systems
Peter Reiher

Outline

- Goals and vision of distributed computing
- Basic architectures
  - Symmetric multiprocessors
  - Single system image distributed systems
  - Cloud computing systems
  - User-level distributed computing

Goals of Distributed Computing

- Better services
  - Scalability
    - Some applications require more resources than one computer has
    - Should be able to grow system capacity to meet growing demand
  - Availability
    - Disks, computers, and software fail, but services should be 24x7!
  - Improved ease of use, with reduced operating expenses
    - Ensuring correct configuration of all services on all systems
- New services
  - Applications that span multiple system boundaries
  - Global resource domains, services decoupled from systems
  - Complete location transparency

Important Characteristics of Distributed Systems

- Performance
  - Overhead, scalability, availability
- Functionality
  - Adequacy and abstraction for target applications
- Transparency
  - Compatibility with previous platforms
  - Scope and degree of location independence
- Degree of coupling
  - How many things do distinct systems agree on?
  - How is that agreement achieved?

Loosely and Tightly Coupled Systems

- Tightly coupled systems
  - Share a global pool of resources
  - Agree on their state, coordinate their actions
- Loosely coupled systems
  - Have independent resources
  - Only coordinate actions in special circumstances
- Degree of coupling
  - Tight coupling: global coherent view, seamless fail-over
    - But very difficult to do right
  - Loose coupling: simple and highly scalable
    - But a less pleasant system model

Globally Coherent Views

- Everyone sees the same thing
- Usually the case on single machines
- Harder to achieve in distributed systems
- How to achieve it?
  - Have only one copy of things that need single view
    - Limits the benefits of the distributed system
    - And exaggerates some of their costs
  - Ensure multiple copies are consistent
    - Requiring complex and expensive consensus protocols
- Not much of a choice

Major Classes of Distributed Systems

- Symmetric Multi-Processors (SMP)
  - Multiple CPUs, sharing memory and I/O devices
- Single-System Image (SSI) & Cluster Computing
  - A group of computers, acting like a single computer
- Loosely coupled, horizontally scalable systems
  - Coordinated, but relatively independent systems
  - Cloud computing is the most widely used version
- Application level distributed computing
  - Application level protocols
  - Distributed middle-ware platforms

Symmetric Multiprocessors (SMP)

- What are they and what are their goals?
- SMP price/performance
- OS design for SMP systems
- SMP parallelism
  - The memory bandwidth problem
- Non-Uniform Memory Architectures (NUMA)

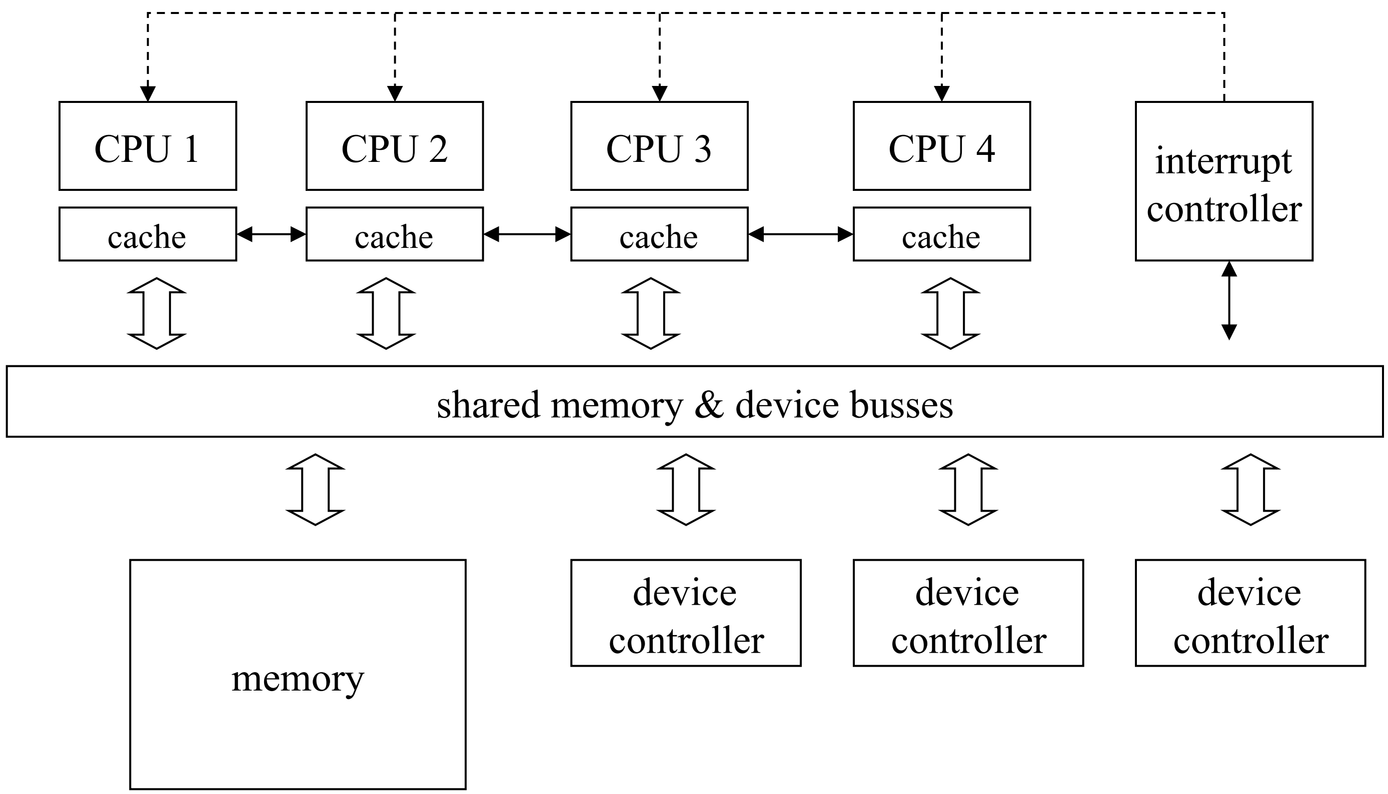
SMP Systems

- Computers composed of multiple identical compute engines
  - Each computer in SMP system usually called a node
- Sharing memories and devices
- Could run same or different code on all nodes
  - Each node runs at its own pace
  - Though resource contention can cause nodes to block
- Examples:
  - BBN Butterfly parallel processor
  - More recently, multi-way Intel servers

SMP Goals

- Price performance
  - Lower price per MIP than single machine
- Scalability
  - Economical way to build huge systems
  - Possibility of increasing machine's power just by adding more nodes
- Perfect application transparency
  - Runs the same on 16 nodes as on one
  - Except faster

A Typical SMP Architecture



The SMP Price/Performance Argument

- A computer is much more than a CPU
  - Mother-board, disks, controllers, power supplies, case
  - CPU might cost 10-15% of the cost of the computer
- Adding CPUs to a computer is very cost-effective
  - A second CPU yields cost of 1.1x, performance 1.9x
  - A third CPU yields cost of 1.2x, performance 2.7x
- Same argument also applies at the chip level
  - Making a machine twice as fast is ever more difficult
  - Adding more cores to the chip gets ever easier
- Massive multi-processors are an obvious direction

SMP Operating Systems

- One processor boots with power on
  - It controls the starting of all other processors
- Same OS code runs in all processors
  - One physical copy in memory, shared by all CPUs
- Each CPU has its own registers, cache, MMU
  - They cooperatively share memory and devices
- ALL kernel operations must be Multi-Thread-Safe
  - Protected by appropriate locks/semaphores
  - Very fine grained locking to avoid contention

Handling Kernel Synchronization

- Multiple processors are sharing one OS copy
- What needs to be synchronized?
  - Every potentially sharable OS data structure
    - Process descriptors, file descriptors, data buffers, message queues, etc.
    - All of the devices
- Could we just lock the entire kernel, instead?
  - Yes, but it would be a bottleneck
  - Remember lock contention?
  - Avoidable by not using coarse-grained locking

SMP Parallelism

- Scheduling and load sharing
  - Each CPU can be running a different process
  - Just take the next ready process off the run-queue
  - Processes run in parallel
  - Most processes don't interact (other than inside kernel)
    - If they do, poor performance caused by excessive synchronization
- Serialization
  - Mutual exclusion achieved by locks in shared memory
  - Locks can be maintained with atomic instructions
  - Spin locks acceptable for VERY short critical sections
  - If a process blocks, that CPU finds next ready process

The Challenge of SMP Performance

- Scalability depends on memory contention
  - Memory bandwidth is limited, can't handle all CPUs
  - Most references better be satisfied from per-CPU cache
  - If too many requests go to memory, CPUs slow down
- Scalability depends on lock contention
  - Waiting for spin-locks wastes time
  - Context switches waiting for kernel locks waste time
- This contention wastes cycles, reduces throughput
  - 2 CPUs might deliver only 1.9x performance
  - 3 CPUs might deliver only 2.7x performance

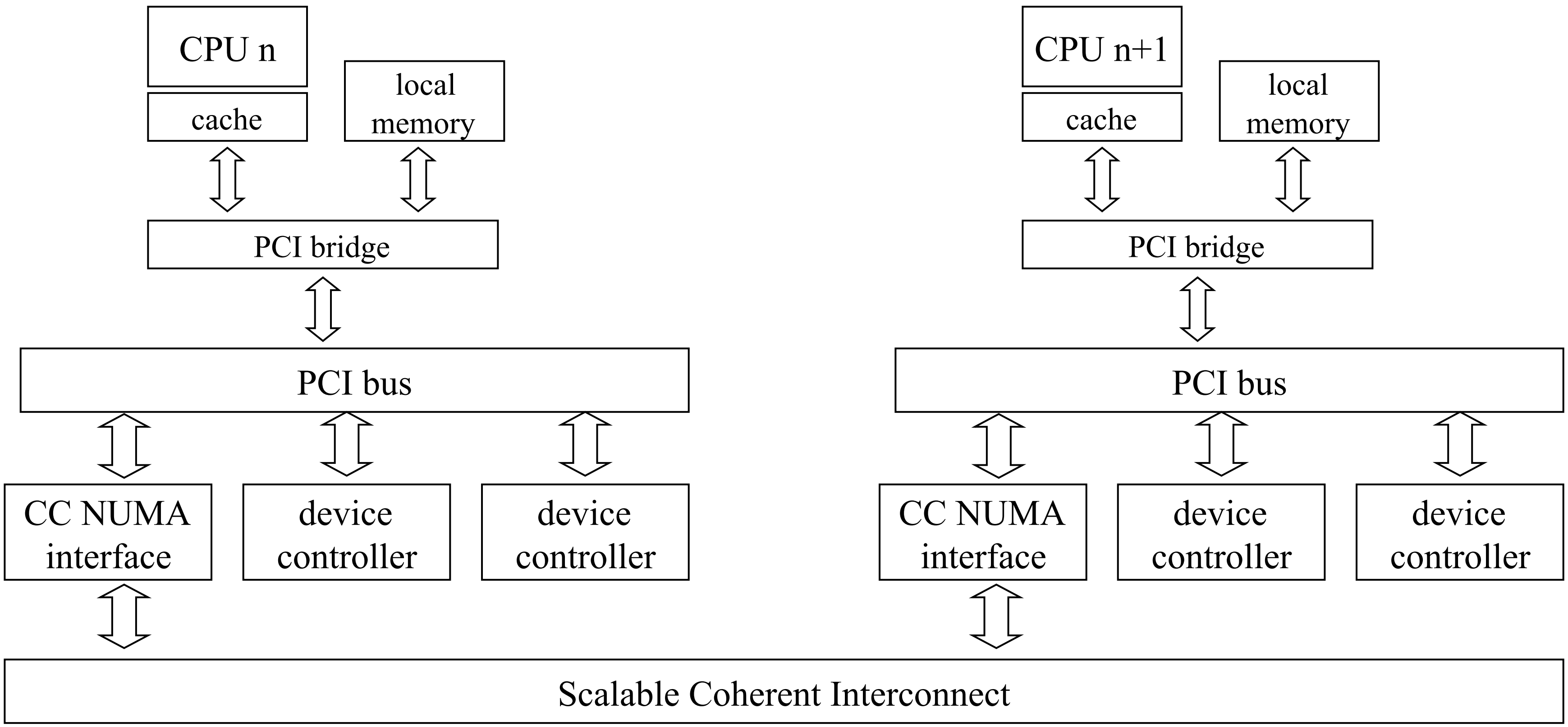
Managing Memory Contention

- Each processor has its own cache
  - Cache reads don't cause memory contention
  - Writes are more problematic
- Locality of reference often solves the problems
  - Different processes write to different places
- Keeping everything coherent still requires a smart memory controller
- Fast n-way memory controllers are <u>very</u> expensive
  - Without them, memory contention taxes performance
  - Cost/complexity limits how many CPUs we can add

NUMA

- Non-Uniform Memory Architectures
- Another approach to handling memory in SMPs
- Each CPU gets its own memory, which is on the bus
  – Each CPU has fast path to its own memory
- Connected by a Scalable Coherent Interconnect
  – A <u>very fast</u>, <u>very local</u> network between memories
  – Accessing memory over the SCI may be 3-20x slower
- These interconnects can be highly scalable

A Sample NUMA SMP Architecture



OS Design for NUMA Systems

- All about local memory hit rates
  - Each processor must use local memory almost exclusively
  - Every outside reference costs us 3-20x performance
  - We need 75-95% hit rate just to break even
- How can the OS ensure high hit-rates?
  - Replicate shared code pages in each CPU's memory
  - Assign processes to CPUs, allocate all memory there
  - Migrate processes to achieve load balancing
  - Spread kernel resources among all the CPUs
  - Attempt to preferentially allocate local resources
  - Migrate resource ownership to CPU that is using it

The Key SMP Scaling Problem

- True shared memory is expensive for large numbers of processors
- NUMA systems require a high degree of system complexity to perform well
  - Otherwise, they're always accessing remote memory at very high costs
- So there is a limit to the technology for both approaches
- Which explains why SMP is not ubiquitous

Single System Image Approaches

- Built a distributed system out of many more-or-less traditional computers
  - Each with typical independent resources
  - Each running its own copy of the same OS
  - Usually a fixed, known pool of machines
- Connect them with a good local area network
- Use software techniques to allow them to work cooperatively
  - Often while still offering many benefits of independent machines to the local users

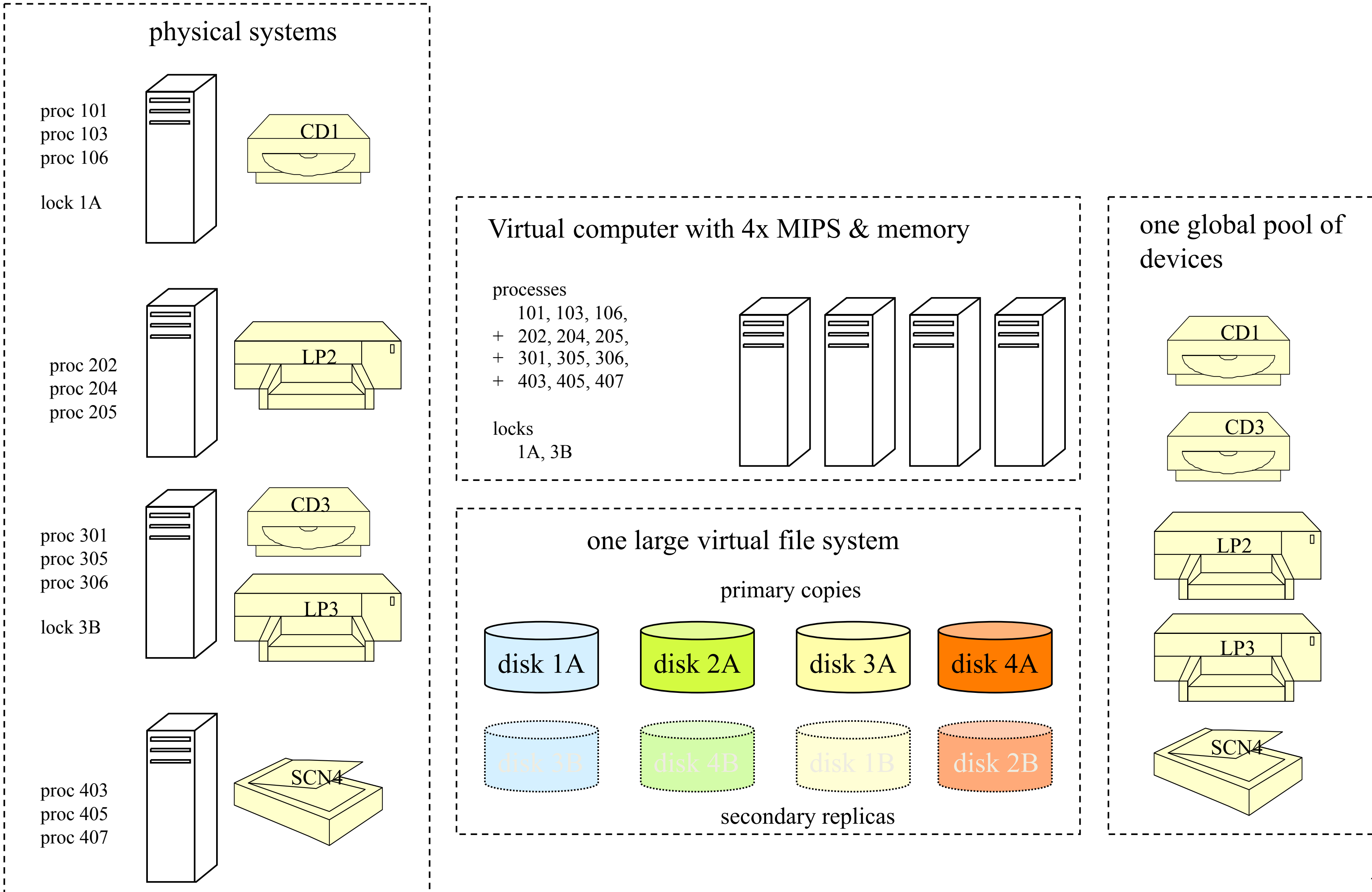
Motivations for Single System Image Computing

- High availability, service survives node/link failures
- Scalable capacity (overcome SMP contention problems)
  - You're connecting with a LAN, not a special hardware switch
  - LANs can host hundreds of nodes
- Good application transparency
- Examples:
  - Locus, Sun Clusters, MicroSoft Wolf-Pack, OpenSSI
  - Enterprise database servers

Why Did This Sound Like a Good Idea?

- Programs don't run on hardware, they run on top of an operating system
- All the resources that processes see are already virtualized
- Don't just virtualize a single system's resources, virtualize many systems' resources
- Applications that run in such a cluster are (automatically and transparently) distributed

The SSI Vision



OS Design for SSI Clusters

- All nodes agree on the state of all OS resources
  - File systems, processes, devices, locks, IPC ports
  - Any process can operate on any object, transparently
- They achieve this by exchanging messages
  - Advising one another of all changes to resources
    - Each OS's internal state mirrors the global state
  - To execute node-specific requests
    - Node-specific requests automatically forwarded to right node
- The implementation is large, complex, and difficult
- The exchange of messages can be very expensive

SSI Performance

- Clever implementation can minimize overhead
  - 10-20% overall is not uncommon, can be much worse
- Complete transparency
  - Even very complex applications “just work”
  - They do not have to be made “network aware”
- Good robustness
  - When one node fails, others notice and take-over
  - Often, applications won't even notice the failure
  - Each node hardware-independent
    - Failures of one node don’t affect others, unlike some SMP failures
- Very nice for application developers and customers
  - But they are complex, and not particularly scalable

An Example of SSI Complexity

- Keeping track of which nodes are up
- Done in the Locus Operating System through "topology change"
- Need to ensure that all nodes know of the identity of all nodes that are up
- By running a process to figure it out
- Complications:
  - Who runs the process? What if he's down himself?
  - Who do they tell the results to?
  - What happens if things change while you're running it?
  - What if the system is partitioned?

Is It Really That Bad?

- Nodes fail and recovery rarely
- So something like topology change doesn't run that often
- But consider a more common situation
- Two processes have the same file open
  – What if they're on different machines?
  – What if they are parent and child, and share a file pointer?
- Basic read operations require distributed agreement
  – Or, alternately, we compromise the single image
  – Which was the whole point of the architecture

Scaling and SSI

- Scaling limits proved not to be hardware driven
  - Unlike SMP machines
- Instead, driven by algorithm complexity
  - Consensus algorithms, for example
- Design philosophy essentially requires distributed cooperation
  - So this factor limits scalability

Lessons Learned From SSI

- Consensus protocols are expensive
  - They converge slowly and scale poorly
- Systems have a great many resources
  - Resource change notifications are expensive
- Location transparency encouraged non-locality
  - Remote resource use is much more expensive
- A very complicated operating system design
  - Distributed objects are much more complex to manage
  - Complex optimizations to reduce the added overheads
  - New modes of failure with complex recovery procedures