Distributed Operating Systems

Distributed Operating Systems

- Not all operating systems are on a single CPU
- The nature of the distribution varies widely
- Thus, so do the possible solutions
- Let's look at such computers, and in particular what they do to OS design

Types of Distributed Computes

- Multiprocessors
- Multicomputers
- Distributed systems (and the Global Grid)

Multiprocessors

- We've been encountering them all semester
- Multiple CPUs on a single bus
- Current trend in chip and system design
- Cause of great complexity all throughout the system
- Primary effect: true concurrency; need Test and Set Lock instruction

Memory Architecture

- Primarily shared memory — low-latency (nanoseconds) access to all of RAM from all CPUs
- But — limit is probably about 128 CPUs, due to bus contention (yes, that number will go up. . .)
- Solutions: caching and private memory
- Access to private memory doesn't cause bus contention
- But — what do you put there?

Non-Uniform Memory Architecture

- Linux supports multiple types of memory
- Good OS, compiler, and application design can use this well
- Example: put stack and program in private memory; heap can be split

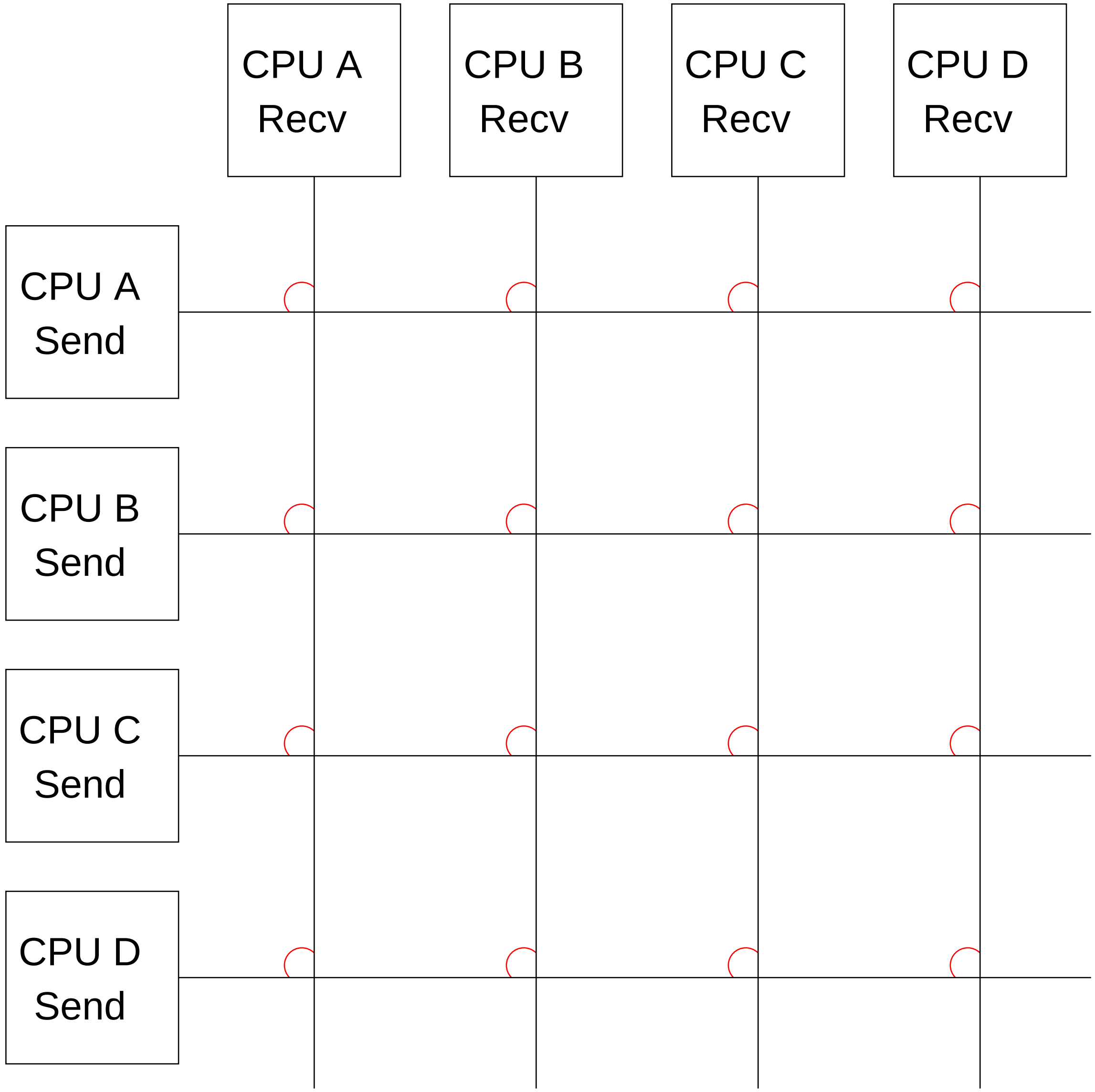
Threads and Multiprocessors

- Should threads from the same process use the same CPU?
- No perfect answer!
- Yes — avoid cache and TLB flushes
- No — avoid latency after messages (or equivalent) from one thread to another

Multicomputers

- Many independent computers connected by a *switching fabric*
- Memory is not shared
- No bus contention
- Contention for switching fabric

A Crossbar Switch



Crossbar Switches

- Switch for every input/output pair
- Non-blocking — every possible conversation can happy simultaneously
- Needs n^2 interconections — only scales to a certain point
- (Classic telephone switch design)

Other Types of Switch Fabrics

- Many other types of switching fabrics
- Goals include lower cost, more scalability, etc.
- Some have contention — see below
- Basic goal: communication time on the order of microseconds

Implications of a Multicomputer

- Operating system code must be replicated
- No shared memory between CPUs for data structures or locks
- No shared memory between CPUs for threads
- Conclusion: threads live on a single CPU (well, maybe)
- Hard to move processes

Distributed Shared Memory

- We don't have shared memory, but we can fake it with *Distributed Shared Memory*
- Make shared pages write-protected on each CPU
- When a process (or the OS) tries to write to such a page, a protection fault occurs
- Tell the other CPUs the page is locked for write, and make it writable
- Later, copy that page to the other CPUs

We’ve Seen This Before

- This is the same sort of copy-on-write that we use after a `fork()`
- Also similar to some caches
- Other CPUs can make the page unreadable until they get a new copy
- Alternatively, leave it readable elsewhere — no guarantees of synchronization without locks

What is Being Locked?

- This scheme presumes locking on a *page* basis
- Application programs normally lock based on their own data structures
- What if these structures are much bigger? Or much smaller?
- What if there are several independently-locked structures in a single page?
- Must make this visible to the user (or at least to the compiler

Network I/O

- What are the properties of this network?
- How fast is it? How fast is it relative to a disk?
- What is the overhead for starting a transmission?
- Is contention possible? Who handles it?
- If contention is possible, do we need some sort of fair scheduling algorithm?
- Which CPU decides?

Data Copies on the Network

Consider a normal network transmission:

1. User to kernel
2. Kernel to interface
3. Interface to interface
4. Interface to kernel
5. Kernel to user

Five copies, four involving RAM!

Can we do direct I/O to user space? Possible, but it's not easy

Direct Network I/O

- All the usual problems of direct I/O: DMA to virtual addresses, locking pages in memory, etc.
- More complex here — data can arrive asynchronously, too
- How does user program start a transmission? Realize one has finished? System calls and I/O interrupts are expensive

Ring Buffers

- User process specifies a receiver and allocate two memory areas, one for input and one for output
- Each area contains several buffers, arranged in a ring, plus a control section
- To write a messaege, copy it to a free buffer and set its "buffer busy" flag
- The network interface then transmits it; when through, it clears the "busy" flag
- The same thing happens on receive

Onboard Buffers?

- In that scheme, the data is generally copied from the CPU's memory to the board's memory
- Why not have board transmit/receive from CPU memory?
- Bus contention — need buffering anyway, in case the board can't get to RAM
- Why not map board memory directly to user space?
- Often possible, but might tie up board's memory bus

An Issue for Multicomputers

- All network I/O — and for that matter, all high-speed I/O of any type — has such issues
- Why do we focus on it here?
- Much higher bandwidth interface; besides, we're trying to run it like a single computer

Remote Procedure Calls

- Direct I/O is still I/O
- That is, the programmer has to treat it as I/O
- Can we avoid that?
- Yes — with *remote procedure calls* (RPC)

Remote Procedure Calls

- Appears to the programmer as an ordinary function call
- Under the hood, the arguments are copied over the network
- Results are copied back to the caller
- It looks exactly like an ordinary procedure call, only slower
- Well, not really. . .

Copying Arguments

- Procedure arguments from the caller have to be *marshaled*
- Marshaling converts the arguments to a linear format, perhaps with type information, for transmission across the network
- The same is done with any results
- Pointers are more difficult. The marshaling routine dereferences the pointer, sends that value across the network, and on return copies the new value into the pointed-to variable
- But what if it's pointing to a complex data structure?

Under the Hood

- The programmer has to specify which routines are remote
- A preprocessor generates stub routines on each side — on the caller's side, they're just subroutines that do network I/O; on the procedure side, they're network listeners that call the actual procedures
- Somewhere, network addressing information has to be supplied

Distributed Systems

- We're not restricted to a bus or a limited area, dedicated switch
- We can build a distributed system on any networking technology at all
- That includes the Internet

The Start of Sun Microsystems

- "Sun" stood for *Stanford University Network*
- Typical deployments involved a group of diskless workstations connected to a disk server via an Ethernet
- Each machine ran a separate copy of Unix
- But in many ways, the network was designed to act as a single distributed computer

Challenges

- Latency
- Network reliability
- Locking
- Bandwidth
- Security

Latency

- Latency is much higher – hundreds of microseconds
- (Early networks had millisecond latency)
- Distributed shared memory performs more poorly
- Effect of higher latency is pervasive

Network Reliability

- Is the network functioning?
- Will all messages be delivered?
- Generally must assume that the network is not reliable
- Any desired reliability must be provided by the host — and the OS
- For that matter, are remote computers reliable?
- What if they crash or are rebooted?

Design for Unreliability

- Every distributed system operation can fail
- Every distributed system operation can take a long time
- Every distributed system operation requires a timeout or other "liveness" check
- The distribution is visible to the application, whether it's explicit I/O, remote procedure calls, or distributed shared memory
- Applications must be aware of these issues and be prepared to cope

Locking

- How do you lock a resource globally?
- Is one machine a lock manager? Which one?
- What if the lock manager crashes?
- Principle: the machine that owns the resource owns the locks for it
- (What about dual-ported disks?)

Bandwidth

- A LAN isn't as fast as a small-scale switch
- It can't be, because of the *inherently* higher latency
- The speed of light in fiber is 20 cm/nanosecond
- The *TCP throughput equation* shows that maximum bandwidth is inversely proportional to latency

$$B \leq C \cdot \frac{S}{R\sqrt{p}}$$

where B is bandwidth, C is a constant, S is packet size, R is round trip time, and p is packet loss probability

Effects of Bandwidth Limits

- Networked disk I/O speed is limited
- Another reason why distributed shared memory doesn't work well — too much latency on certain memory references (design principle: actions that are expensive should appear different to the programmer)

Security

- How do we trust machines across a network?
- How do we enforce file permissions?
- How do we identify users?
- Use cryptography

Cryptography and the Distributed OS

- Cryptography can be used for confidentiality, which is good
- More important, cryptography can be used for *integrity* and *authenticity*, which are often more important
- Suppose root on machine A has a key K_A
- A message integrity-protected with K_A could only have come from root on A

Non-Root Permissions

- Let each machine's root attach the actual userid to a message
- Prepare a message "Root says that `smb` says ..."
- Integrity-protect it with K_A; the receiving machine can believe it, and apply `smb`'s permissions
- (Cryptographic reality is far more complex)

Capabilities

- Instead of passing around userids, use *capabilities*
- The OS prepares a list of access rights, cryptographically seals it, and gives it to the user process
- The user process can employ it locally or send it across the network
- File permission-checking can be much simpler — is access to that file in the user's capability set?
- (But how are capabilities revoked?)

Distributed File Systems

- How do we build a distributed file system?
- Naming
- Security
- Performance
- Consistency

Naming

- Must have a uniform naming convention
- Does the name include the location of the file? If not, how do we find it?
- Must have a (distributed) name service

Performance

- Especially if a file is far away, don't want to retrieve each block from the network one at a time
- Cache it — make a local copy
- Especially good for things like shared executables

Consistency

- Suppose that machines A and B open a file simultaneously
- If A writes to the file, does B see the change? If so, when?
- What if A has a cached copy?
- Usual answer is *session semantics*: changes are only pushed out when the file is closed