Advanced Architectures

Goals of Distributed Computing

- better services
  - scalability
    - apps too big to run on a single computer
    - grow system capacity to meet growing demand
  - improved reliability and availability
  - improved ease of use, reduced CapEx/OpEx
- new services
  - applications that span multiple system boundaries
  - global resource domains, services (vs. systems)
  - complete location transparency

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
- application level distributed computing
  - peer-to-peer, application level protocols
  - distributed middle-ware platforms

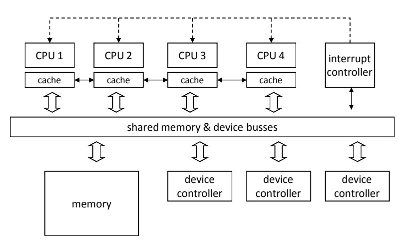
Evaluating Distributed Systems

- 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
  - on how many things do distinct systems agree
  - how is that agreement achieved

SMP systems and goals

- Characterization:
  - multiple CPUs sharing memory and devices
- Motivations:
  - price performance (lower price per MIP)
  - scalability (economical way to build huge systems)
  - perfect application transparency
- Example:
  - multi-core Intel CPUs
  - multi-socket mother boards

Symmetric Multi-Processors



SMP Price/Performance

- 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 obvious direction

SMP Operating System Design

- 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 must 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

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 in kernel)
- 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

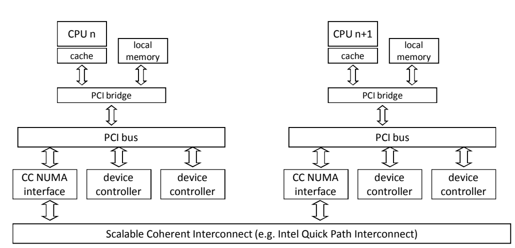
The Challenge of SMP Performance

- scalability depends on memory contention
  - memory bandwidth is limited, can't handle all CPUs
  - most references satisfied from per-core 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
- contention wastes cycles, reduces throughput
  - 2 CPUs might deliver only 1.9x performance
  - 3 CPUs might deliver only 2.7x performance

Managing Memory Contention

- Fast n-way memory is <u>very</u> expensive
  - without it, memory contention taxes performance
  - cost/complexity limits how many CPUs we can add
- Non-Uniform Memory Architectures (NUMA)
  - each CPU has its own memory
    - 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

Non-Uniform Memory Architecture Symmetric Multi-Processors



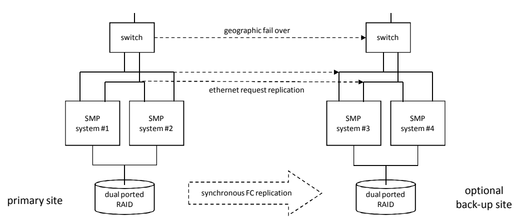
OS design for NUMA systems

- it is all about local memory hit rates
  - 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

Single System Image (SSI) Clusters

- Characterization:
  - a group of seemingly independent computers collaborating to provide SMP-like transparency
- Motivation:
  - higher reliability, availability than SMP/NUMA
  - more scalable than SMP/NUMA
  - excellent application transparency
- Examples:
  - Locus, MicroSoft Wolf-Pack, OpenSSI
  - Oracle Parallel Server

Modern Clustered Architecture



Active systems service independent requests in parallel. They cooperate to maintain shared global locks, and are prepared to take over partner's work in case of failure. State replication to a back-up site is handled by external mechanisms.

OS design for SSI clustering

- 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
  - request execution of node-specific requests
    - node-specific requests are forwarded to owning node
- implementation is large, complex, difficult
- the exchange of messages can be very expensive

SSI Clustered 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
- nice for application developers and customers
  - but they are complex, and not particularly scalable

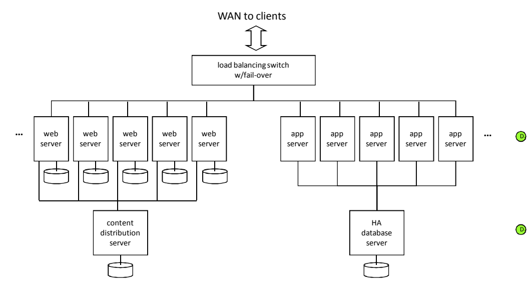
Lessons Learned

- 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 greatly complicated operating system
  - distributed objects are more complex to manage
  - complex optimizations to reduce the added overheads
  - new modes of failure w/complex recovery procedures
- Bottom Line: Deutsch was right!

Loosely Coupled Systems

- Characterization:
  - a parallel group of independent computers
  - serving similar but independent requests
  - minimal coordination and cooperation required
- Motivation:
  - scalability and price performance
  - availability – if protocol permits stateless servers
  - ease of management, reconfigurable capacity
- Examples:
  - web servers, Google search farm, Hadoop

Horizontal Scalability w/HA



(elements of architecture)

- farm of independent servers
  - servers run same software, serve different requests
  - may share a common back-end database
- front-ending switch
  - distributes incoming requests among available servers
  - can do both load balancing and fail-over
- service protocol
  - stateless servers and idempotent operations
  - successive requests may be sent to different servers

Horizontally scaled performance

- individual servers are very inexpensive
  - blade servers may be only \$100-\$200 each
- scalability is excellent
  - 100 servers deliver approximately 100x performance
- service availability is excellent
  - front-end automatically bypasses failed servers
  - stateless servers and client retries fail-over easily
- the challenge is managing thousands of servers
  - automated installation, global configuration services
  - self monitoring, self-healing systems

Limited Transparency Clusters

- Single System Image Clusters had problems
  - all nodes had to agree on state of all objects
  - lots of messages, lots of complexity, poor scalability
- What if they only had to agree on a few objects
  - like cluster membership and global locks
  - fewer objects, fewer operations, much less traffic
  - objects could be designed for distributed use
    - leases, commitment transactions, dynamic server binding
- Simpler, better performance, better scalability
  - combines best features of SSI and Horizontally scaled

Limited LocationTransparency

- what things look the same as local?
  - remote file systems
  - remote terminal sessions, X sessions
  - remote procedure calls
- what things don't look the same as local?
  - primitive synchronization (e.g. mutexes)
  - basic Inter-Process Communication (e.g. signals)
  - process create, destroy, status, authorization
  - Accessing devices (e.g. tape drives)

Clouds: Applied Horizontal Scalability

- Many servers, continuous change
  - dramatic fluctuations in load volume and types
  - continuous node additions for increased load
  - nodes and devices are failing continuously
  - continuous and progressive s/w updates
- Most services delivered via switched HTTP
  - clients/server communication is over WAN links
  - large (whole file) transfers to optimize throughput
  - switches route requests to appropriate servers
  - heavy reliance on edge caching

Geographic Disaster Recovery

- Cloud reliability/availability are key
  - one data center serves many (10^3-10^7) clients
- Local redundancy can only provide 4-5 nines
  - fires, power and communications disruptions
  - regional scale (e.g. flood, earthquake) disasters
- Data Centers in distant Availability Zones
  - may be running active/active or active/stand-by
  - key data is replicated to multiple data centers
  - traffic can be redirected if a primary site fails

WAN-Scale Replication

- WAN-scale mirroring is slow and expensive
  - much slower than local RAID or network mirroring
- Synchronous Mirroring
  - each write must be ACKed by remote servers
- Asynchronous Mirroring
  - write locally, queue for remote replication
- Mirrored Snapshots
  - writes are local, snapshots are mirrored
- Fundamental tradeoff: reliability vs. latency

WAN-Scale Consistency

- CAP theorem - it is not possible to assure:
  - Consistency (all readers see the same result)
  - Availability (bounded response time)
  - Partition Tolerance (survive node failures)
- ACID databases sacrifice *partition tolerance*
- BASE semantics sacrifice *consistency*
  - Basic Availability (most services most of the time)
  - Soft state (there is no global consistent state)
  - Eventual consistency (changes propagate, slowly)

Dealing with Eventual Consistency

- distributed system has no single, global state
  - state updates are not globally serialized events
  - different nodes may have different opinions
- expose the inconsistencies to the applications
  - ask the cloud, receive multiple answers
  - let each application reconcile the inconsistencies
- BASE semantics are neither simple nor pretty
  - they embrace parallelism and independence
  - they reflect the complexity of distributed systems

Distributed Computing Reformation

- systems must be more loosely coupled
  - tight coupling is complex, slow, and error-prone
  - move towards coordinated independent systems
- move away from old single system APIs
  - local objects and services don't generalize
  - services are obtained through messages (or RPCs)
  - in-memory objects, local calls are a special case
- embrace the brave new (distributed) world
  - topology and partnerships are ever-changing
  - failure-aware services (commits, leases, rebinds)
  - accept distributed (e.g. BASE) semantics

How to Exploit a Cloud

- Replace physical machines w/virtual machines
  - cloud provides inexpensive elastic resources
- Run massively parallel applications
  - requiring huge numbers of computers
- Massively distributed systems are very difficult
  - to design, build, maintain and manage
- How can we make exploiting parallelism easy?
  - new, tool supported, programming models
  - encapsulate complexity of distributed systems

Distributed Middle-Ware

- API adapters
  - e.g. HIVE (SQL bridge)
- complexity hiding
  - e.g. remote procedure calls, distributed objects
- restricted programming platforms
  - e.g. Java Applets, Erlang, state machines
- new programming models
  - e.g. publish-subscribe, MapReduce, Key-Value stores
- powerful distributed applications
  - e.g. search engines, Watson, Deep Learning

Distributed Systems – Summary

- different degrees of transparency
  - do applications see a network or single system image
- different degrees of coupling
  - making multiple computers cooperate is difficult
  - doing it without shared memory is even worse
- performance vs. independence vs. robustness
  - cooperating redundant nodes offer higher availability
  - communication and coordination are expensive
  - mutual-dependency creates more modes of failure

Changing Architectural Paradigms

- a “System” is a collection of services
  - interacting via stable and standardized protocols
  - implemented by app software deployed on nodes
- Operating Systems
  - manage the hardware on which the apps run
  - implement the services/ABIs the apps need
- The operating system is a platform
  - upon which higher level software can be built
  - goodness is measured by how well it does that job

What Operating Systems Do

- Originally (and at the start of this course)
  - abstract heterogeneous hardware into useful services
  - manage system resources for user-mode processes
  - ensure resource integrity and trusted resource sharing
  - provide a powerful platform for developers
- None of this has changed, but …
  - notion of a self-contained system becoming obsolete
  - hardware and OS heterogeneity is a given
  - most important interfaces are higher level protocols
- Operating Systems continue to evolve as
  - new applications demand new services
  - new hardware must be integrated and exploited

What Operating Systems Do

Could you and I with Him conspire,
to grasp this sorry scheme of things entire,
Would we not shatter it to bits – and then
re-mould it nearer to the heart's desire?

Omar Khayyam
The Rubaiyat, XCIX

Was uns nicht umbringt macht uns nur stärker!

(That which does not kill us only makes us stronger!)

Friedrich Wilhelm Nietzsche
Man and Superman

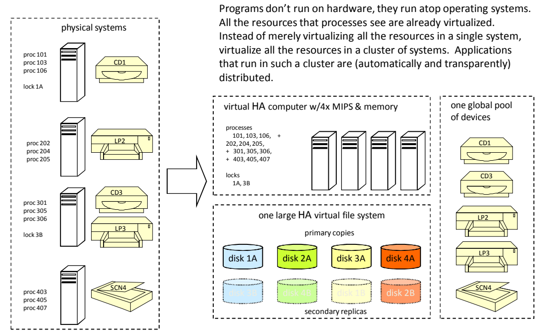
Final Exam

- Tue 6/13, Boelter 3400
- Part I: 08:00-09:30
  - 10 questions, similar to mid-term
  - closed book
  - covering reading and lectures since mid-term
- Part II: 09:30-11:00
  - 4/6 questions, similar to mid-term XC
  - closed book
  - covering entire course

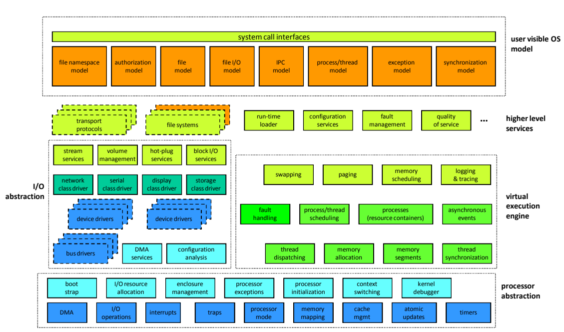
Supplementary Slides

The Dream

Programs don’t run on hardware, they run atop operating systems. All the resources that processes see are already virtualized. Instead of merely virtualizing all the resources in a single system, virtualize all the resources in a cluster of systems. Applications that run in such a cluster are (automatically and transparently) distributed.



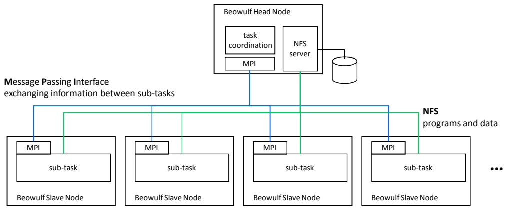
Structure of a Modern OS



Transparency

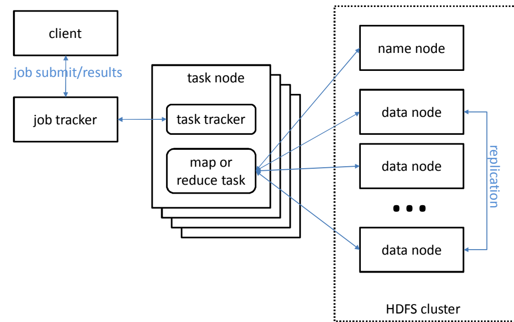
- Ideally, a distributed system would be just like a single machine system
- But better
  - More resources
  - More reliable
  - Faster
- *Transparent* distributed systems look as much like single machine systems as possible

Loosely Coupled Scalability
(Beowulf High Performance Computing Cluster)



There is no effort at transparency here. Applications are specifically written for a parallel execution platform and use a Message Passing Interface to mediate exchanges between cooperating computations.

Example: A Hadoop Cluster



(Hadoop Distributed Middleware)

- Client
  - up-loads data into an HFDS cluster
  - creates map/reduce data analysis program
  - submits job to a Hadoop Job Tracker
- Job Tracker
  - sends sub-tasks to task trackers on many nodes
- Task Trackers
  - spawn/monitor map/reduce tasks, collect status
- Map/Reduce Tasks
  - run analysis program on a defined data sub-set
  - reading data from/writing results to HDFS cluster

Embarrassingly Parallel Jobs

- Problems where it’s really, really easy to parallelize them
- Probably because the data sets are easily divisible
- And exactly the same things are done on each piece
- So you just parcel them out among the nodes and let each go independently
- Everyone finishes at more or less same time

MapReduce

- Perhaps the most common cloud computing software tool/technique
- A method of dividing large problems into compartmentalized pieces
- Each of which can be performed on a separate node
- With an eventual combined set of results

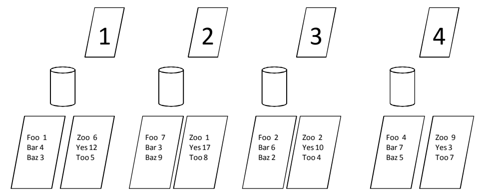
The Idea Behind MapReduce

- There is a single function you want to perform on a lot of data
  - Such as searching it for a string
- Divide the data into disjoint pieces
- Perform the function on each piece on a separate node (*map*)
- Combine the results to obtain output (*reduce*)

An Example

- We have 64 megabytes of text data
- Count how many times each word occurs in the text
- Divide it into 4 chunks of 16 Mbytes
- Assign each chunk to one processor
- Perform the map function of "count words" on each

The Example Continued

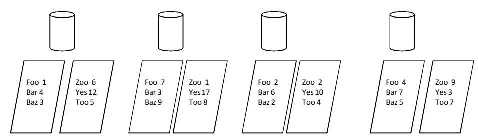


That's the map stage

On To Reduce

- We might have two more nodes assigned to doing the reduce operation
- They will each receive a share of data from a map node
- The reduce node performs a reduce operation to "combine" the shares
- Outputting its own result

Continuing the Example



The Reduce Nodes Do Their Job

Write out the results to files

And MapReduce is done!

But I Wanted A Combined List

- No problem
- Run another (slightly different) MapReduce on the outputs
- Have one reduce node that combines everything

Synchronization in MapReduce

- Each map node produces an output file for each reduce node
- It is produced atomically
- The reduce node can’t work on this data until the whole file is written
- Forcing a synchronization point between the map and reduce phases