Advanced Systems Topics
Part I of III

Steven Hand

Lent Term 2005

6 lectures of 15 for CST II

Course Aims

This course aims to help students develop and understand complex systems and interactions, and to prepare them for emerging systems architectures.

It will cover a selection of topics including:

- operating systems,
- database systems,
- peer-to-peer systems, and
- parallel and distributed systems.

On completing the course, students should be able to

- describe three techniques supporting extensibility
- argue for or against distributed virtual memory
- describe how to build effective concurrency-control primitives for a modern computer
- compare and contrast various self-organising distributed lookup schemes
- architect a basic peer-to-peer application

Course Outline

- Part I: Advanced Operating Systems [SMH, 6L]
  - Distributed & Persistent Virtual Memory
  - Capability Systems & The CAP Computer
  - Microkernels & Virtual Machine Monitors
  - Extensibile Operating Systems
  - Database & Distributed Storage [2L]
- Part II: Scalable Synchronization [KAF, 4L]
  - Introduction (systems with 10K threads)
  - Architectures and Algorithms
  - Implementing Mutual Exclusion
  - Programming without Locks
- Part III: Peer-to-Peer Systems [JAC, 5L]
  - P2P Intro, Case Studies and Applications [3L]
  - Internet Coordinate Systems (Guest Lecture)
  - Structured v Unstructured P2P (Guest Lecture)

Recommended Reading

- Singhal M and Shivaratris N
  Advanced Concepts in Operating Systems
  McGraw-Hill, 1994
- Stonebraker M and Shivaratri N
  Readings in Database Systems
  Morgan Kaufmann (3rd ed.), 1998
- Wilkes M V and Needham R M
  The Cambridge CAP Computer and its Operating System
  North Holland, 1979
- Hennessy J and Patterson D
  Computer Architecture: a Quantitative Approach
  (Chapter 6 in particular)
  Morgan Kaufmann (3rd ed.), 2003
- Bacon J and Harris T
  Operating Systems, Addison Wesley, 2003
- Peer-to-Peer Systems and the Grid
  `www.cl.cam.ac.uk/~jac22/out/grid-p2p-paper.pdf`
- Additional links and papers (via course web page)
  `www.cl.cam.ac.uk/Teaching/current/AdvSysTop/`

Memory Models

Memory models for concurrent/**parallel** programs:

- Shared memory model:
  - collection of 'threads' sharing address space
  - reads/writes on memory locations implicitly and immediately globally visible
  - e.g. `x := x + 1`
- Message passing model:
  - collection of 'processes' (private address spaces)
  - explicit coordination through messages, e.g

Processor 1	Processor 2
send_message("fetch(x)")	receive message
	send_message("x")
tmp := recv_message(P2)	
tmp := tmp + 1	
send_message("tmp")	x: = recv_message(P1)

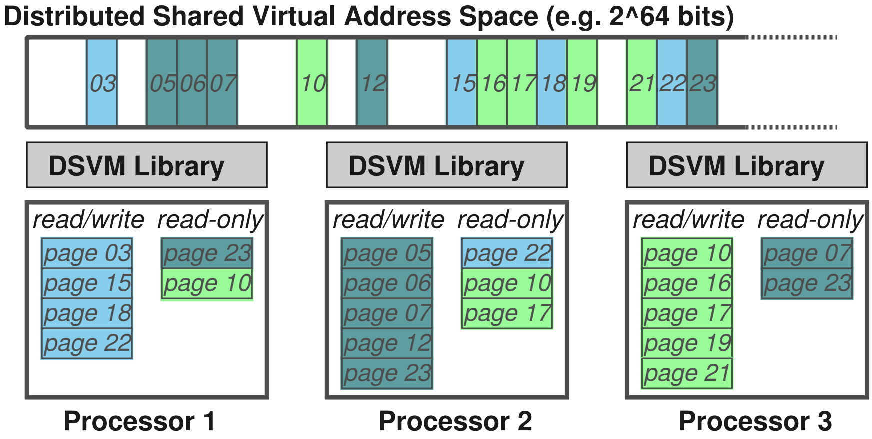
- Message passing: control, protection, performance
- Shared memory:
  - ease of use
  - transparency & scalability
  - but: race conditions, synchronisation, cost

Recap: Demand Paged Virtual Memory

- Run-time mapping from logical to physical addresses performed by special h/w (the MMU).
- Variants: segmentation, capabilities, **paging**.
- Typically use *demand paging*:
  - create process address space (setup page tables)
  - mark PTEs as either "invalid or "non-resident"
  - add PCB to scheduler.
- Then whenever we receive a *page fault*:
  1. check PTE to determine if "invalid" or not
  2. if an invalid reference ⇒ kill process;
  3. otherwise 'page in' the desired page:
     - find a free frame in memory (may require direct or asynchronous *page replacement*)
     - initiate disk I/O to read in the desired page
     - when I/O is finished modify the PTE for this page to show that it is now valid
     - restart the process at the faulting instruction

Seems fairly straightforward for *uniprocessors*. . .

Distributed Shared Virtual Memory



- Memory model typically dictated by hardware:
  - shared memory on *tightly-coupled* systems,
  - message passing on *loosely-coupled* systems
- Radical idea: provide shared memory on clusters!
  - each page has a "home" processor
  - can be mapped into remote address spaces
  - on read access, page in across network
  - on write acess, sort out ownership. . .
- OS/DSVM library responsible for:
  - tracking current ownership
  - copying data across network
  - setting access bits to ensure coherence

Implementing DSVM (1)

- Simple case: centralized page manager
  - runs on a single processor
  - maintains two data structures per-page:
    - owner(p) = the processor P that created or which last wrote to page p
    - copyset(p) = all processors with a copy of p
  - can store copyset as bitmap to save space
- Then on read fault need four messages:
  - contact manager; manager forwards to owner;
  - owner sends page; requester acks to manager;
- On write fault, need a bit more work:
  - contact manager; manager *invalidates* copyset;
  - manager conacts owner; owner relinquishes page;
  - requester acks to manager;
- Load-balance: manager(p) is (p % #processors)
- Reduce messages: manager(p) = owner(p):
  - broadcast to find manager(p) ?
  - or keep per-processor *hint*: probOwner(p) ?
  - update probOwner(p) on forwarding or invalidate

Implementing DSVM (2)

- Still potentially expensive, e.g. false-sharing:
  - $P1$ owns p, $P2$ just has read-access
  - $P1$ writes $p \Rightarrow$ copies to $P2$
  - but $P2$ doesn't care about this change
- Reduce traffic by using weaker memory consistency:
  - so far assumed sequential consistency:
    * every read sees latest write
    * easy to use, but expensive
  - instead can do e.g. release consistency:
    * reads and writes occur locally
    * explicit *acquire* & *release* for synch
    * analogy with memory barriers in MP
- Best performance by doing *type-specific* coherence:
  - private memory $\Rightarrow$ ignore
  - write-once $\Rightarrow$ just service read faults
  - read-mostly $\Rightarrow$ owner broadcasts updates
  - producer-consumer $\Rightarrow$ live at P, ship to C
  - write-many $\Rightarrow$ release consistency & buffering
  - synchronization $\Rightarrow$ strong consistency

DSVM: Evolution & Conclusions

- mid 1980's: IVY at Princeton (Li)
  - sequential consistency (used probOwner(), etc)
  - some nice results for parallel algorithms with large data sets
  - overall: too costly
- early 1990's: Munin at Rice (Carter)
  - type-specific coherence
  - release consistency (when appropriate)
  - allows optimistic multiple writers
  - almost as fast as hand-coded message passing
- mid 1990's: Treadmarks at Rice (Keleher)
  - introduced "lazy release consistency"
  - update not on release, but on next acquire
  - reduced messages, but higher complexity
- On clusters:
  - can always do better with explicit messages
  - complexity argument fails with complex DSVM
- On non-ccNUMA multiprocessors: sounds good!

Persistence

Why is *virtual* memory volatile?

- virtual memory means memory is (or at least may be) backed by non-volatile storage.
- why not make this the default case?
  - ⇒ no more distinction between files and memory
  - ⇒ programmatic access to file system or DB bases:
    - ∗ is easier (e.g. linked structures)
    - ∗ can benefit from type system

Some definitions:

- *Persistence* of data = length of time it exists
- *Orthogonal Persistence* = manner in which data is accessed is independent of how long it persists

Two main options for implementation:

- Functional/interpreted languages ⇒ can 'fake out' in language runtime.
- Imperative/compiled languages:
  - – prescribe way to access data (e.g. pure OO), or
  - – use the power of virtual memory. . .

The Multics Virtual Memory

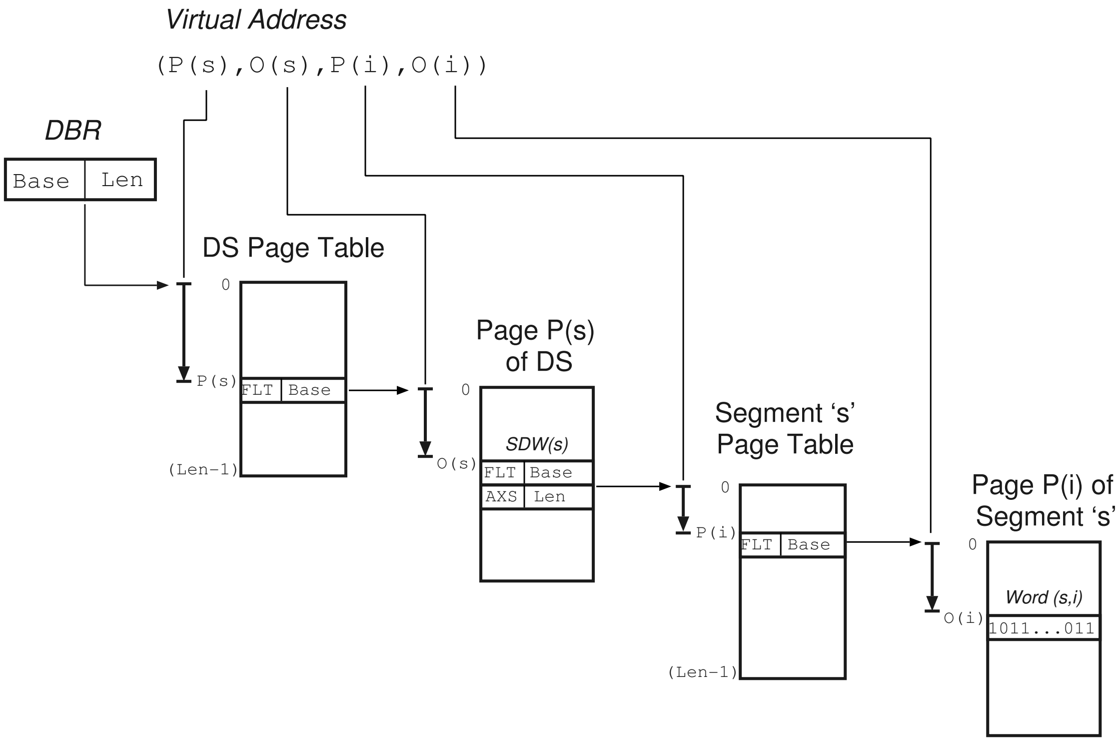
Unifying VM and non-volatile storage is *old*:

- Developed 1964– by MIT, GE and AT&T Bell Labs
- Many (8–64) concentric **rings** of privilege.
- No filesystem *per se*; user saw large number of orthogonal linear regions ("segments") of virtual address space, each backed by a secondary store.
- Segments were created and named by users, and remained available until explicitly deleted.
- Tree of directories and non-directories, a la Unix
- Directories contains a set of **branches** ($\sim$ inodes)
- A branch contains a set of attributes for a segment:
  - Unique Segment ID (assigned by FS)
  - An access control list (ACL) for each UID
  - A ring **bracket** and limit:
    * ACL applies only within bracket
    * bracket is a pair $(b1 \leq b2)$, limit is $l \geq b2$
  - A list of **gates** = procedure entry points
- Flexible security: processes "jump" through gates

Problems with Multics

A victim of complexity. . .

- e.g. GE 645 hardware for paged segments



- e.g. translating "names" to usable segments:
  - access via pathnames or reference names
  - separate "known" and "active" concepts:
    - ∗ per proc "known" segments (KST)
    - ∗ per system "active" segments (AST)
  - segment fault handlers, page fault handlers
- good security ⇒ harder to do stuff!

Persistent Virtual Memory

Increasing secondary storage $\Rightarrow$ persistence tricky:

- cannot safely *name* (refer to) all data
- e.g. consider limitations of `mmap()`
- possible soln via *pointer swizzling*, e.g. Texas
  - portable C++ library
  - can allocate objects on *persistent heap*
  - data in persistent pages canonically addressed by special 64-bit persistent pointers (PPtrs)
  - ensure PPtrs are *never* directly accessed:
    - mark any resident persistent page as invalid
    - trap on access and for every PPtr $\hat{p}$
      - allocate a new page P and mark it invalid
      - *swizzle* (rewrite) $\hat{p}$ to refer to P
      - unprotect original page and resume
- Recent 64-bit address spaces mean virtual addresses can directly serve as unique names:

$\Rightarrow$ can have a *single address space* OS (SASOS)

- many SASOSes (e.g. Opal, Mungi) have PVM
- can also combine with DSVM: smart?

Recoverable Virtual Memory

- RVM refers to a region of virtual memory on which *transactional* semantics apply.
- Building block for filesystems, DBs, applications.
- Best known work: *lightweight* RVM (Satya et al, SOSP '93, ACM TOCS '94)
  - full transaction semantics too expensive

  ⇒ just consider atomicity and [some] durability
  - processes map regions of external segments into their virtual address space and then:
    * Start with `t = begin_transaction(rmode)`
    * Invoke `set_range(t, base_addr, nbytes)`
      · normally LRVM *copies* range when notified and adds to undo log ⇒ on abort, can restore old values.
      · elide if `rmode` is "no-restore"
    * Finally `end_transaction(t, cmode)`:
      · LVRM synchronously writes all ranges to redo log.
      · (lazy write if `cmode` is "no-flush")
  - Redo log gets full ⇒ reflect log contents to external segments and truncate log.
- Can build full transaction semantics on top of LRVM (see paper for details).

Making RVM Faster

LVRM is faster than full transaction system but:

- up to three copies of data (undo, redo, truncate)
- expensive synchronous disk writes

Can we do better?

- Controversial *Rio Vista* proposed SOSP '97
- Uses Rio, a persistent (NVRAM) file cache:
  – on map, just `mmap` region of NVRAM
  – on `set_range()` copy to undo log in NVRAM
  – all updates immediately durable ⇒ no redo log.
- Authors report performance wins up to 2000x since:
  – no synchronous disk writes required
  – no redo log ⇒ avoid two copies
- So what if the machine crashes?
  – early in reboot, flush NVRAM contents to disk
  – on map, lazily undo any transactions which had not committed at time of crash.
- Q: performance if DB doesn't fit in NVRAM?

Capability-based addressing

A capability is a protected name for an object.

- possession is necessary and sufficient for access
- supplied by system and must be unforgeable
- can be manipulated in a defined and restricted set of ways (e.g. passed as param, refined, etc)

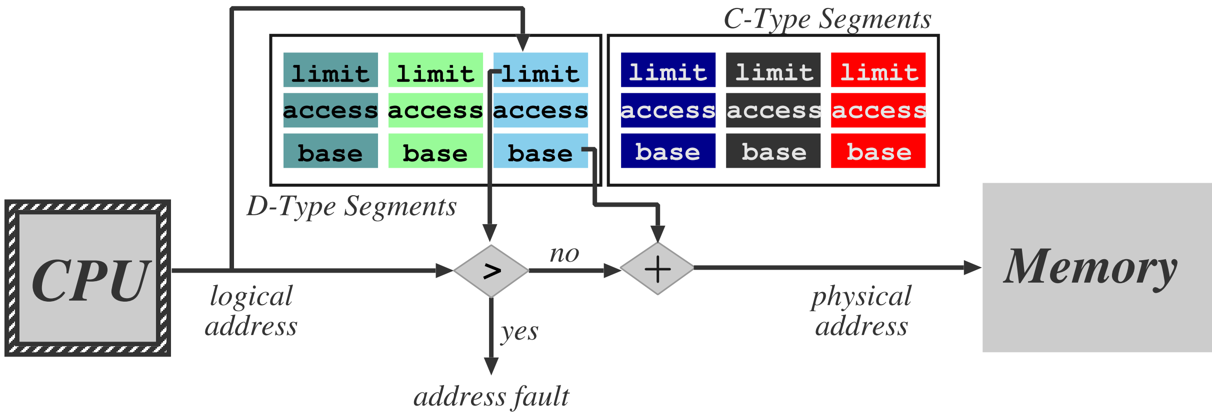
Can implement in software (crypto) or in *hardware*.

The Cambridge CAP Computer

Developed here starting in 1970 (Needham, Wilkes, Wheeler, Richards, Moody, Johnson, Herbert, . . .)

- Recognises need for hardware memory protection on a fine grained level.
- The CAP controlled *what* can be written to registers, not *who* can write to them.
- Base-limit registers and their contents become *capability registers* and *capabilities*
- A capability consists of three values:
  - **base, limit** and **access code**

Capability Architectures



- Protection relies on unforgeable capabilities
  - data (and code) are stored in different segment type from capabilities
  - D-type (data-type) segments: words may be transferred to/from arithmetic registers
  - C-type (capability-type) segments: words may be transferred to/from capability registers
- Need some highly trusted system procedure with both C- and D-type capability for same segment
- Also need way to load capabilities into registers:
  - e.g. Plessey system 250 had explicit instructions
  - by contrast, in CAP, loading is implicit whenever a capability is referred to (c/f TLB)

Control of Privilege in the CAP

- In conventional systems, all control lies with OS designer (i.e. coarse grained)
- Rings of protection: more flexible as long as OS remains at the centre of the set of rings
- CAP: no problem with giving access to facilities to a subsystem designer which are identical to those used by main system
- Nothing hierarchical about capabilities
- Note that hierarchies are useful in organisation of flow of **control**, but are unnecessarily restrictive for **protection**

Analogy with Structured Programming

- The CAP is to hardware what scoping is to programming
- Further advantages are being able to more easily debug programs and even to prove correctness!

Domains of Protection

- This is the set of capabilities to which a process has access (i.e. can cause to be loaded into the capability registers)
- Special instruction needed to change domain of protection (ENTER)
- Need to be careful when leaving a protection domain – cannot leave capabilities lying about in capability registers
- ENTER and RETURN give rise to a hierarchy of **control** but not of **protection**

Protection of Processes

- Necessary to support multiprogramming
- Also need to give one process privileges which differ from another – define a **protection environment** for process
- "Kernel" (co-ordinator) ENTERs user process; control RETURNs on process trap, or interrupt
- Requires specific hardware support (in microcode)

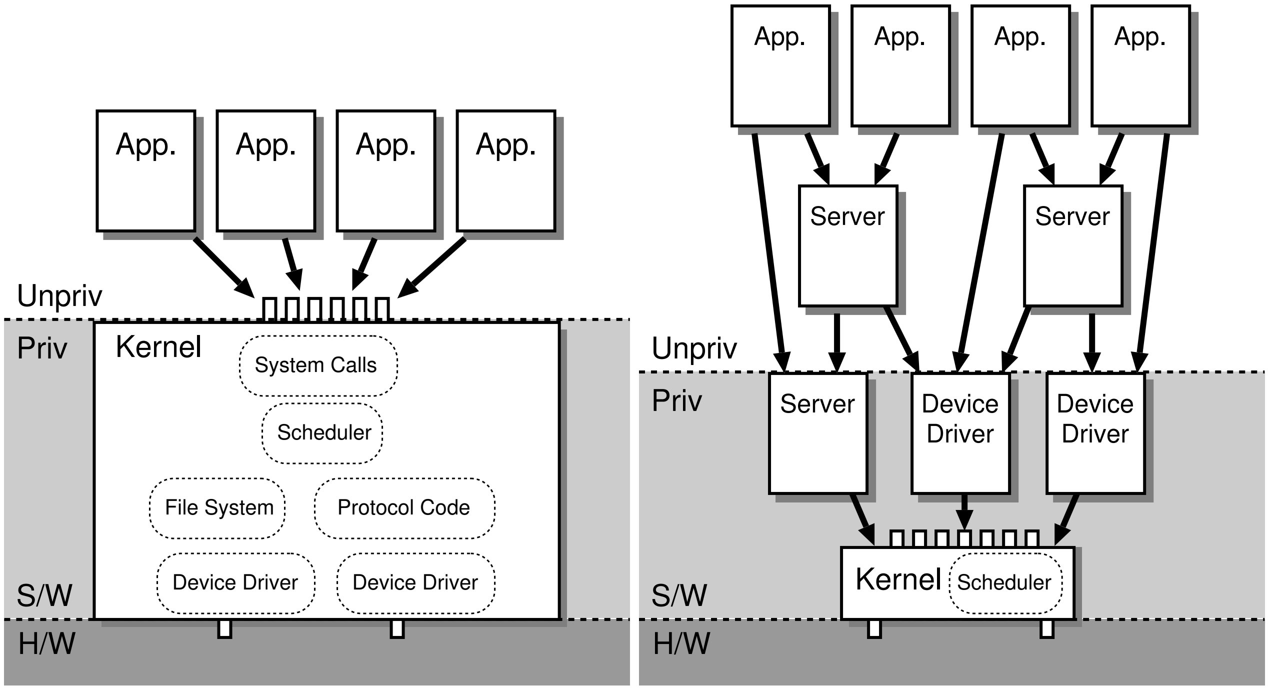
Relative Capabilities

- Capabilities defined previously have a segement base which is an absolute address in memory;
- i.e. a capability selects subset of entire memory
- *Relative Capabilites* allow the base to be relative to the base of some other segment
- Now capability is:
  - (base, limit, access code, reference)
  - reference is { capability | whole memory }
- This allows us to evaluate a chain of references
- Furthermore, a process can now 'hand on' a subset of memory access privileges to its sub-processes
- In the CAP operating system:
  - "kernel" (co-ordinator) capabilities live in a segment called the *master resource list* (MRL)
  - each process has *process resource list* (PRL) with capabilities relative to those in MRL
  - Can recurse. . .

Summary: The CAP & Capabilities

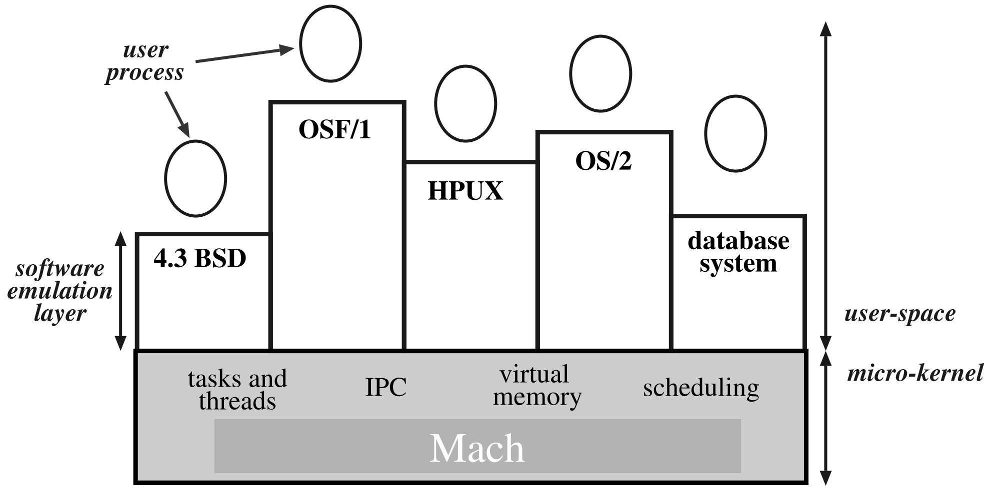
- The CAP was an architectural innovation and extremely successful – for details see the book
- Key features include:
  - Segment swapping.
  - Local naming.
  - Flexible control of sub-processes:
- CAP enforces high degree of modularity on programs $\Rightarrow$ easy to modify OS and programs
- Minimum privilege: each process runs with minimum degree of required privileges
- Similar ideas used in Hydra (CMU), IBM System 38, Intel i432, CAP II/III
- But:
  - hardware complex and expensive
  - systems often slow in practice
  - and the killer: security vs. usability
- So although technically cool, capabilities – and the CAP – didn't win (although see EROS later. . .)

Microkernel Operating Systems



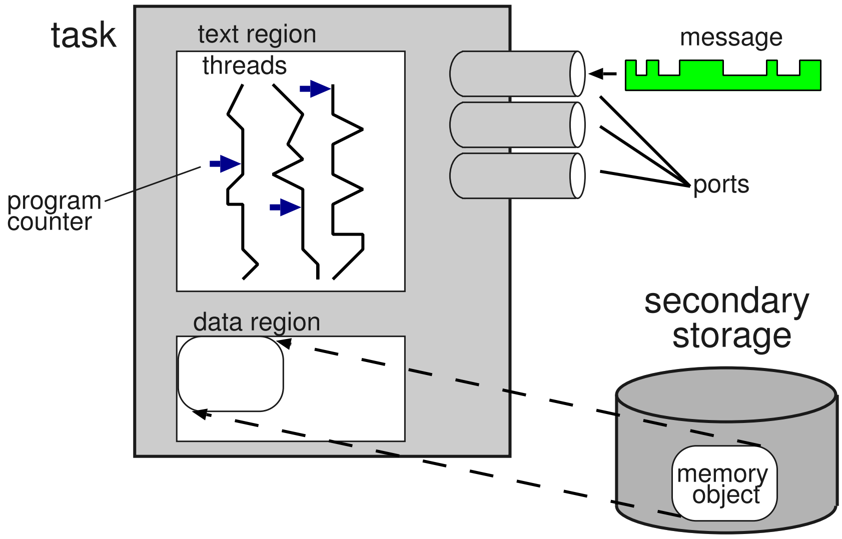
- New concept in early 1980's:
  - "kernel" scheme (*lhs*) considered complex
  - $\Rightarrow$ try to simplify kernel, build modular system
  - support multiprocessors, distributed computing
- Re-engineered OS structure (*rhs*)
  - move functions to user-space servers
  - access servers via some *interprocess communication* (IPC) system
  - increase modularity $\Rightarrow$ more robust, scalable. . .

The Mach Microkernel



- Mach developed at CMU (Rashid, Bershad, . . .)
- Evolved from BSD 4.2
- Provided compatibility with 4.3 BSD, OS/2, . . .
- Design goals:
  - support for diverse architectures, including multiprocessors (SMP, NUMA, NORMA)
  - scale across network speeds
  - distributed operation:
    - heterogeneous machine types
    - memory management & communications
- (NB: above diagram shows Mach 3.0)

Mach Abstractions



- Tasks & threads:
  - a task is an execution environment
  - a thread is the unit of execution
- IPC based on *ports* and *messages*:
  - port = generic reference to a 'resource'
  - implemented as buffered comms channels
  - messages are the unit of communication
    $\Rightarrow$ IPC is message passing between threads
  - also get *port sets* (share a message queue)
- Also get *memory objects*:
  - memory object is a 'source' of memory
  - e.g. memory manager, or a file on a file server

L3/L4: Making Microkernels Perform

- Perceived problems with microkernels:
  - many kernel crossings ⇒ expensive
  - e.g. Chen (SOSP'93) compared Mach to Ultrix:
    * worse locality (jumping in/out of Mach)
    * more large block copies
- Basic dilemma:
  - if too much in μ-kernel, lose benefits (and microkernels often "grow" quite a bit)
  - if too little in μ-kernel, too costly
- Liedtke (SOSP'95) claims that to fix you:
  1. minimise what should be in kernel
  2. make those primitives really fast.
- The L3 (and L4, SOSP'97) systems provided just:
  - recursive construction of address spaces
  - threads
  - IPC
  - unique identifier support
- (Cynical question: is this an operating system?)

L3/L4 Design and Implementation

- Address spaces support by three primitives:
  1. Grant: give pages to another address space
  2. Map: share pages with another address space
  3. Flush: take back mapped or granted pages
- Threads execute with address space:
  - characterised by set of registers
  - μ-kernel manages thread↔address space binding
- IPC is message passing between address spaces:
  - highly optimised for i486 (3μs vs Mach's 18μs)
  - interrupts handled as messages too
- Does it work? '97 paper `getpid()` comparison:

System	Time	Cycles
Linux	1.68μs	223
L^4Linux	3.95μs	526
MkLinux (Kernel)	15.41μs	2050
MkLinux (User)	110.60μs	14710

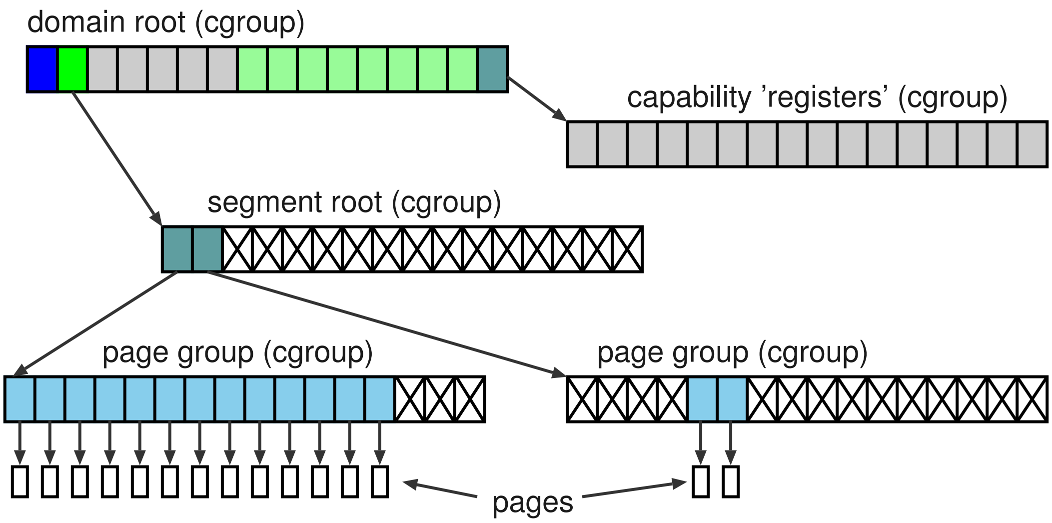
- Q: are these micro-benchmarks useful?
- Q: what about portability?

Extremely Reliable Operating System

EROS: a persistent software capability microkernel.

- Why revisit capabilities?
  - reliability requires system decomposition
  - decomposition $\rightarrow$ access delegation (flexibility)
  - ability to restrict information and access right transmission (security, confinement)
  - access policy is a run time problem
  - persistence simplifies applications, improves I/O
  - ‘active agent’ (applet/servlet/cgi) confinement
  - mutually suspicious users
- But surely:
  - capabilities are slow ?
  - microkernels are (must be?) slow ?
  - capabilities can’t support discretionary access control (just pass them on) ?
  - capability systems are complex ?
- EROS set out to challenge the above. . .

Software Capabilities in EROS



- Two disjoint “spaces” (as per CAP):
  1. data space
     - set of pages: each holds 4096 bytes
     - read and write data to/from data registers
  2. capability space:
     - set of *cgroups*: each holds 16 capabilities
     - read and write to/from capability registers
- Each capability is *(type, oid, authority)*:
  - basic types are *page*, *cgroup*, *number*, *schedule*
  - complex types include *segment* and *domain*
- Segments correspond to address spaces.
- Domains correspond to processes.

Making EROS Fast & Persistent

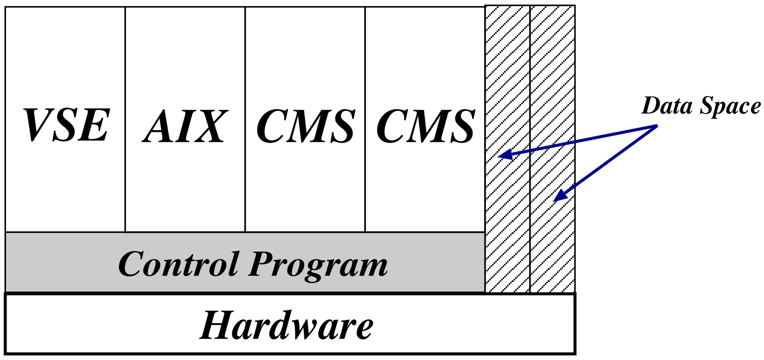
- Persistence achieved by flushing objects to disk:
  - circular log used for checkpointing
  - eventually log entries migrate to home location
- Before using capabilities, they must be *prepared*
  - if necessary bring object referred to into memory
  - modify capability to point to object table
  - mark capability as prepared
- Only unprepared capabilities written to disk.
- Get run-time speed by caching a page-table representation of segment tree:
  - update on any write to segment tree
  - update if capabilities or pages paged out
- Fast capability-based IPC scheme:
  - invocation names capability to be invoked, operation code, four capabilities, and some data
  - *call*, *return* and *send* operations
  - threads migrate with call & return
  - hand-coded for L4-style speed

Virtual Machine Monitors

Forget microkernels: take a different approach.

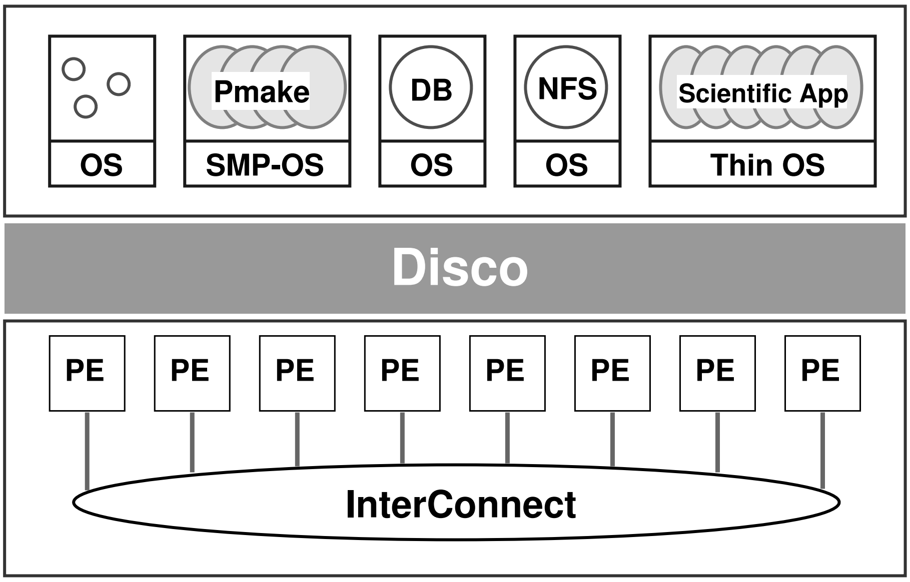
- Use a "hypervisor" (beyond supervisor, i.e. beyond a normal OS) to multiplex multiple OSes.
- (NB: hypervisor $\equiv$ virtual machine monitor)
- Made popular by IBM's VM/CMS (1970's)
- Idea regained popularity in mid 90's:
  - e.g. Disco uses a VMM to make it easier to write operating systems for ccNUMA machines.
  - e.g. VMWare allows you to run Windows on Linux, or vice versa.
  - e.g. Denali lets you run 10,000 web servers
  - e.g. XenoServers allow you to run whatever you want, wherever you want.
- Virtual Machine Monitors somewhat similar to but not the same as the JVM (Java Virtual Machine)

IBM's VM/CMS



- 60's: IBM researchers propose VM for System/360
- 70's: implemented on System/370
- 90's: VM/ESA for ES/9000
- Control program provides each OS with:
  - virtual console
  - virtual processor
  - virtual memory
  - virtual I/O devices
- Complete virtualisation: can even run another VM!
- Performance good since most instructions run direct on hardware.
- Success ascribed to extreme flexibility.

Disco (Stanford University)



- Motivation: run commodity OS on ccNUMA:
  - existing commodity OS do badly on NUMA
  - tricky to modify them successfully
  - writing from scratch a *lot* of work
- Also hope to get:
  - fault tolerance between operating systems
  - ability to run special-purpose OSes
  - reasonable sharing between OSes
- OSes mostly unaware of VMM:
  - CPU looks like real MIPS R10000: privileged insts (including TLB fill) trap and are emulated.

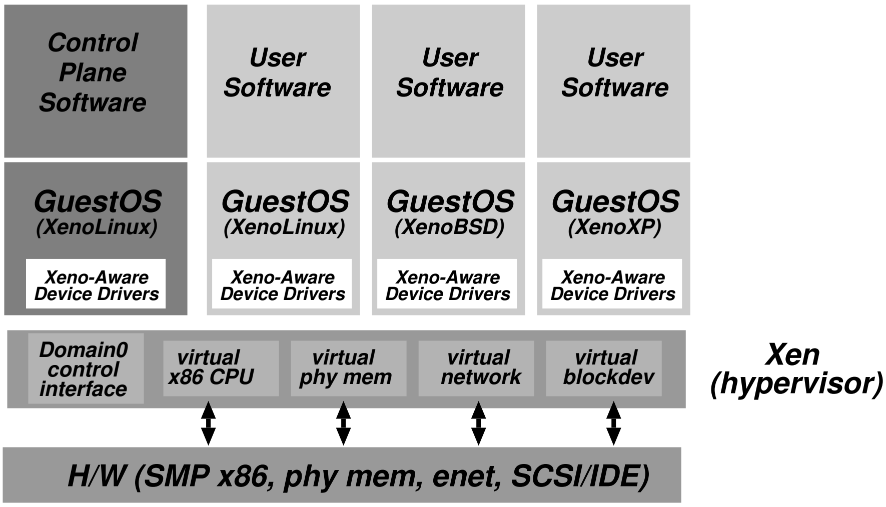
VMWare

- Startup founded 1998 by Stanford Disco dudes
- Basic idea: virtual machines for x86
- One major problem to overcome:
  - x86 not fully virtualizable: 17 instructions have different user/kernel semantics, but do not trap
  - $\Rightarrow$ cannot emulate them!
- VMWare solution: perform binary rewriting to manually insert traps (*extremely* hairy)
- (explains why only certain guest OSes supported)
- "Physical" to machine address mapping realized by using *shadow* page tables.
- Second big problem: performance
  - no longer research prototype $\Rightarrow$ must run at a reasonable speed
  - but no source code access to make small effective modifications (as with Disco)
- VMWare address this by writing special device drivers (e.g. display) and other low-level code

Denali (Univ. Washington)

- Motivation: new application domains:
  - pushing dynamic content code to caches, CDNs
  - application layer routing (or peer-to-peer)
  - deploying measurement infstructures
- Use VMM as an *isolation kernel*
  - security isolation: no sharing across VMs
  - performance isolation: VMM supports fairness mechanisms (e.g. fair queueing and LRP on network path), static memory allocation
- Overall performance by *para-virtualization*
  - full x86 virtualization needs gory tricks
  - instead invent "new" x86-like ISA
  - write/rewrite OS to deal with this
- Work in progress:
  - Yakima isolation kernel based on Flux OSKit
  - Ilwaco single-user guest OS comprises user-space TCP/IP stack plus user-level threads package
  - No SMP, no protection, no disk, no QoS

XenoServers (Cambridge)

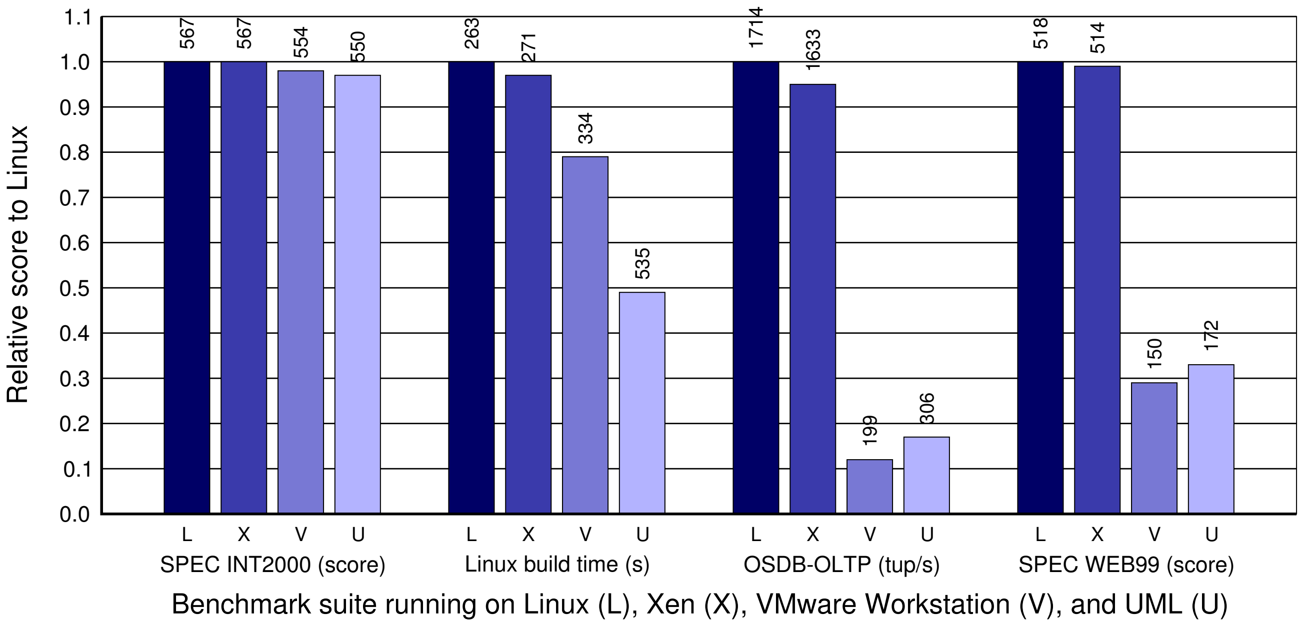


- Vision: XenoServers scattered across globe, usable by anyone to host services, applications, . . .
- Use Xen hypervisor to allow the running of arbitrary untrusted code (including OSes)
- Crucial insight:
  - use SRT techniques to guarantee resources in time and space, and then $charge$ for them.
  - share and protect CPU, memory, network, disks
- Sidestep Denali of Service (DOS:-)
- Use paravirtualization, but real operating systems

Xen Implementation

- Xen based on low-level parts of linux $\Rightarrow$ don't need to rewrite 16-bit startup code.
- Includes device drivers for timers (IOAPICs), network cards, IDE & SCSI.
- Special guest OS (Domain 0) started at boot time:
  - special interface to Xen
  - create, suspend, resume or kill other domains
- Physical memory allocated at start-of-day:
  - guest uses buffered page-table updates to make changes or create new address spaces
  - aware of 'real' addresses $\Rightarrow$ bit awkward
- Interrupts converted into $events$:
  - write to event queue in domain
  - domain 'sees' events only when activated
- Guest OSes run own scheduler off either virtual or real-time timer facility.
- Asynchronous queues used for network and disk

Xen: Comparative Performance



- Attempt to measure *overall* system performance:
  - SPEC INT2000: CPU intensive, no OS
    $\Rightarrow$ expect all systems to perform well
  - Linux build: more I/O $\Rightarrow$ potentially larger hit
  - Final pair (DB workload, Web workload) exercise all parts of OS $\Rightarrow$ can get huge overhead
- Ongoing work in migration, I/O, . . .
- More info from `http://www.cl.cam.ac.uk/xeno/xen`

VMMs: Conclusions

- Old technique having recent resurgence:
  - really just 1 VMM between 1970 and 1995
  - now at least 10 under development
- Why popular today?
  - OS static size small compared to memory
  - (sharing can reduce this anyhow)
  - security at OS level perceived to be weak
  - flexibility as desirable as ever
- Emerging applications:
  - Internet suspend-and-resume:
    * run all applications in virtual machine
    * at end of day, suspend VM to disk
    * copy to other site (e.g. conference) & resume
  - Multi-level secure systems
    * many people run VPN from home to work
    * but machine shared for personal use ⇒ risk of viruses, information leakage, etc
    * instead run VM with only VPN access
  - Data-center management & beyond. . .

Extensibility

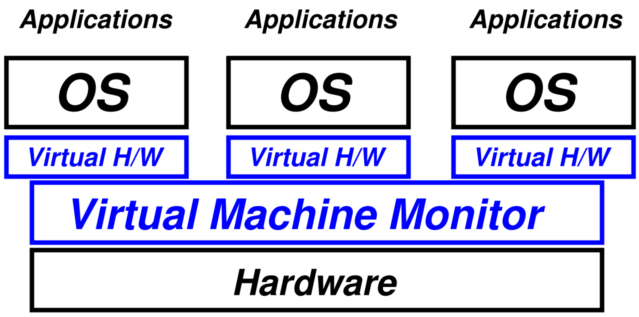
What's it about?

- Fixing mistakes.
- Supporting new features (or hardware).
- Efficiency, e.g.
  - packet filters
  - run-time specialisation
- Individualism, e.g.
  - per-process thread scheduling algorithms.
  - customizing replacement schemes.
  - avoiding "shadow paging" (DBMS).

How can we do it?

1. give everyone their own machine.
2. allow people to modify the OS.
3. allow some of the OS to run outside.
4. reify separation between protection and abstraction.

Low-Level Techniques



Have just seen one way to provide extensibility: give everyone their own [virtual] machine:

- Lowest level s/w provides
  a) virtual hardware, and
  b) some simple secure multiplexing.

  $\Rightarrow$ get N pieces of h/w from one.
- Then simply run OS on each of these N:
  - can pick and choose operating system.
  - users can even recompile and "reboot" OS without logging off!
  - Q: how big is a sensible value for N?
  - what about layer violations?
- Examples: VM, VMWare, Disco, XenoServers, . . .

Kernel-Level Schemes (1)

Often don't require entirely new OS:

- Just want to replace/modify some small part.
- Allow portions of OS to be dynamically [un]loaded.
- e.g. Linux kernel modules
  - requires dynamic relocation and linking.
  - once loaded must *register*.
  - support for [un]loading on demand.
- e.g. NT/2K/XP services and device drivers
  - well-defined entry / exit routines.
  - can control load time & behaviour.
- However there are some problems, e.g.
  - requires clean [stable?] interfaces
  - specificity: usually rather indiscriminate.
- . . . and the big one: security.
  - who can you trust?
  - who do you rate?

Kernel-Level Schemes (2)

Various schemes exist to avoid security problems:

- Various basic techniques:
  - Trusted compiler [or CA] + digital signature.
  - Proof carrying code
  - Sandboxing:
    - limit [absolute] memory references to per-module [software] segments.
    - use *trampolines* for other memory references.
    - may also check for certain instructions.
- e.g. *SPIN* (U. Washington)
  - based around Modula-3 & trusted compiler
  - allows “handlers” for any event.
- Still problems with dynamic behaviour (consider handler `while(1);`) $\Rightarrow$ need more.
- e.g. Vino (Harvard)
  - uses “grafts” = sandboxed C/C++ code.
  - timeouts protect CPU hoarding.
  - in addition supports per-graft resource limits and transactional “undo” facility.

Proof Carrying Code (PCC)

- Take code, *check it*, and run iff checker says it's ok.
- "Ok" means cannot read, write or execute outside some *logical fault domain* (subset of kernel VAS)
- Problem: how do we check the code?
  - generating proof on fly tricky + time-consuming.
  - and anyway termination not really provable
- So expect proof *supplied* and just check proof.
- Overall can get very complex, e.g. need:
  - formal specification language for safety policy
  - formal semantics of language for untrusted code
  - language for expressing proofs (e.g. LF)
  - algorithm for validating proofs
  - method for generating safety proofs
- Possible though, see e.g.
  - Necula & Lee, *Safe Kernel Extensions without Run-time Checking*, OSDI 1996
  - Necula, *Proof Carring Code*, PPOPL 1997
  - SafetyNet Project (Univ. Sussex)

Sandboxing

- PCC needs a lot of theory and a lot of work
- *Sandboxing* takes a more direct approach:
  - take untrusted code as input
  - transform it to make it safe
  - run transformed code
- E.g. *Software Fault Isolation* (SFI, Wahbe et al)
  - Assume logical fault domain once more
  - Scan code and look for memory accesses
  - Insert instructions to perform bounds checking:

```
                        cmp r1, $0x4000; blt fault;
ldr r0, [r1]    →       cmp r1, $0x5000; bgt fault;
                        ldr r0, [r1]
```

  - Better if restrict and align LFD:

```
                        bic r1, $0x03ff;
ldr r0, [r1]    →       cmp r1, $0x4000; bne fault;
                        ldr r0, [r1]
```

  - Can handle indirect jumps similarly.
- Problem: ret, int, variable length instructions, . . .
- Problem: code expansion
  - Trusted optimizing compiler?

The *SPIN* Operating System

- Allow extensions to be downloaded into kernel.
- Want performance comparable with procedure call $\Rightarrow$ use language level (compiler checked) safety:
- *SPIN* kernel written (mostly) in Modula-3
  - Type-safe, and supports strong interfaces & automatic memory managent.
  - (some low-level kernel stuff in C/assembly)
- Kernel resources referenced by *capabilities*
  - capability $\equiv$ unforgeable reference to a resource
  - in *SPIN*, capabilities are Modula-3 pointers
  - *protection domain* is enforced by language name space (not virtual addressing)
- Extensions somewhat ungeneral:
  - define *events* and *handlers*
  - applications register handlers for specific events
  - e.g. handler for "select a runnable thread"
  - what about unforseen needs?
- Problems: trusted compiler, locks, termination. . .

The VINO Operating System

Set out to overcome perceived problems with *SPIN*

- Download *grafts* into kernel.
- Grafts written in C or C++
  - free access to most kernel interfaces
  - safety acheived by SFI (sandboxing)
  - (must use trusted compiler)
- Prevent quantitative resource abuse (e.g. memory hogging) by resource quotas and accounting
- Prevent resource starvation by *timeouts*
  - grafts must be preemptible ⇒ kernel threads
  - decide "experimentally" how long graft can hold certain resources (locks, ipl (?), cpu (?))
  - if graft exceeds limits, terminate.
- Safe graft termination "assured" by transactions:
  - wrapper functions around grafts
  - all access to kernel data via accessors
  - two-phase locking + in-memory undo stack

User-Level Schemes

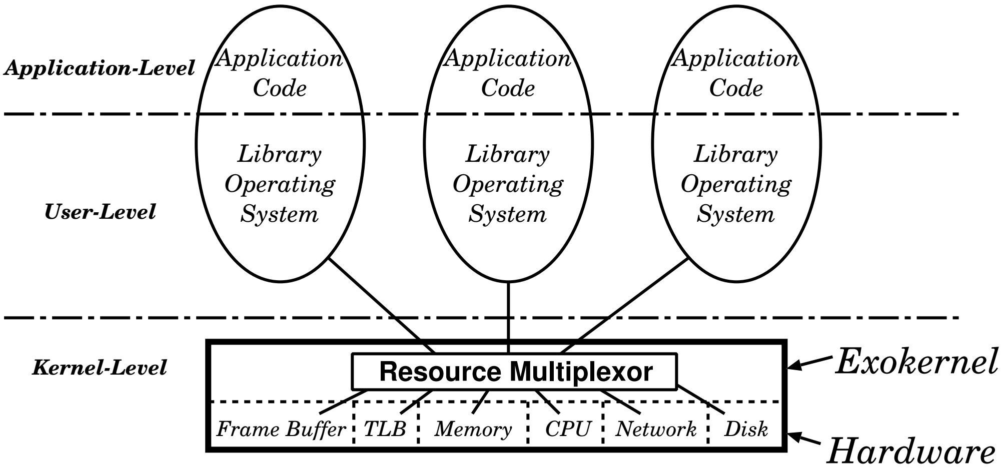
Kernel-level schemes can get very complex $\Rightarrow$ avoid complexity by putting extensions in user-space:

- e.g. μ-kernels + IDL (Mach, Spring)
- still need to handle timeouts / resource hoarding.

Alternatively reconsider split between *protection* and *abstraction* : only former need be trusted.

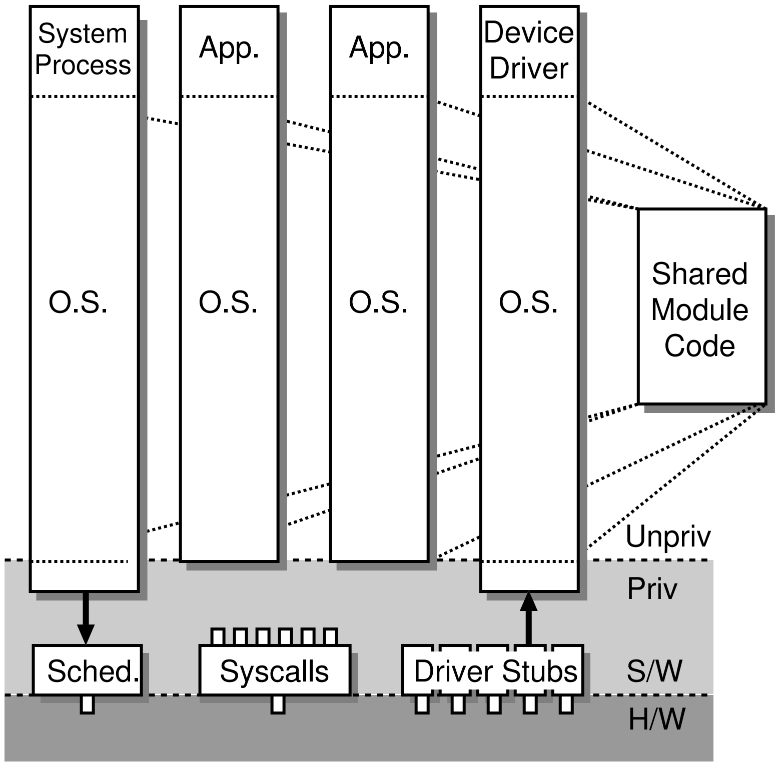
- e.g. Exokernel:
  - run most of OS in user-space library.
  - leverage DSL/packet filters for customization.
  - can get into a mess (e.g. UDFs).
- e.g. Nemesis:
  - guarantee each application share of *physical* resources in both space and time.
  - use IDL to allow user-space extensibility.
  - still requires careful design. . .
- Is this the ultimate solution?

The Exokernel



- Separate concepts of protection and abstraction ⇒ get extensibility, accountability & performance.
- Why are abstractions bad?
  - deny application-specific optimizations
  - discourage innovation
  - impose mandatory costs
- Still need some "downloading":
  - describe packets you wish to receive using DPF; exokernel compiles to fast, unsafe, machine code
  - Untrusted Deterministic Functions (UDFs) allow exokernel to sanity check block allocations.
- Lots of cheezy performance hacks (e.g. Cheetah)

The Nemesis Operating System



- Design to support soft real-time applications
  - isolation: explicit guarantees to applications
  - exposure: multiplex *real* resources
  - responsibility: applications must do data path
- Parallel development to exokernel:
  - similar overall structure (though leaner – no device drivers, DPFs, UDFs, etc, in NTSC)
  - but: strongly typed IDL, module name space
  - but: "temporal protection" built in

Extensibility: Conclusions

- Extensibility is a powerful tool.
- More than just a "performance hack"
  - Simplifies system monitoring.
  - Enables dynamic system tuning.
  - Provides potential for better system/application integration.
- Operating system extensibility is a good design paradigm for the future:
  - Allow extensible applications to take advantage
  - Do operating system modifications "on-the-fly"
- Lots of ways to achieve it:
  - virtual machine monitors (everyone gets own operating system)
  - downloading untrusted code (and checking it?)
  - punting things to user space (fingers crossed)
  - pushing protection boundary to rock bottom

Database Storage

- Recall relational databases from Part IB
- Why not just store relations and directories in ASCII format in standard files, e.g.

Store relation R1 in /usr/db/R1

```
Moody # 123 # CUCL
Kelly # 231 # DPMMS
Bacon # 432 # CUCL
.....
.....
.....
```

Store directory file in /usr/db/directory

```
R1 # Name # STR # Id # INT # Dept # STR
R2 # Id # INT # CRSId # STR
.....
.....
.....
.....
```

- To do `select * from R where` *`condition`*:
  - read directory to get `R` attributes
  - for each line in file containing `R`:
    - ∗ check condition
    - ∗ if OK, display line
- To do `select * from R,S where` *`condition`*:
  - read directory to get `R`, `S` attributes
  - read file containing `R`, for each line:
    - ∗ for each line in file containing `S`
      - · create join tuple
      - · check condition
      - · display if OK

What's Wrong with This?

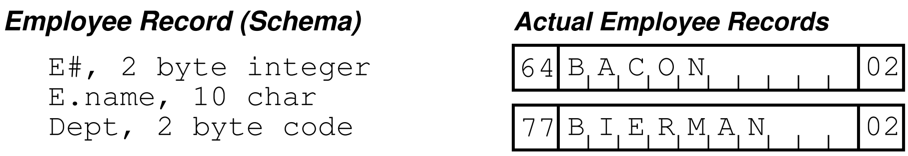
- Tuple layout on disk
  - change 'Bacon' to 'Ham' ⇒ must rewrite file
  - ASCII storage expensive
  - deletions expensive
- Search expensive – no indexes
  - cannot quickly find tuple with key
  - always have to read entire relation
- Brute force query processing
  - `select * from R,S where R.A = S.A and S.B > 10`
  - do select first? more efficient join?
- No reliability
  - can lose data
  - can leave operations half done
- No security
  - file-system is insecure
  - file-system security is coarse
- No buffer management, no concurrency control

Disk Storage Issues

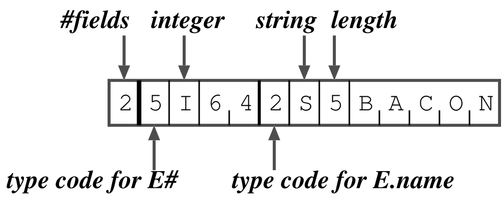
- What block size?
  - large blocks $\Rightarrow$ amortise I/O costs
  - but large blocks mean may read in more useless stuff, and read itself takes longer.
- Need efficient use of disk
  - e.g. sorting data on disk (external sorting)
  - I/O costs likely to dominate
    $\Rightarrow$ design algorithms to reduce I/O
- Need to maximise concurrency
  - e.g. use (at least) double buffering
  - more generally, use asynchronous I/O and a database-specific buffer manager
  - care needed with replacement strategy
- Need to improve reliability
  - need to deal with failures mid transaction
    $\Rightarrow$ use write-ahead log
  - recall transactions from Part IB CSAA

Representing Records

- Record = collection of related data ("fields"):
  - can be fixed or variable *format*
  - can be fixed or variable *length*
- Fixed format ⇒ use schema:
  - schema holds #fields, types, order, meaning
  - records interpretable only using schema
  - e.g. fixed format and length



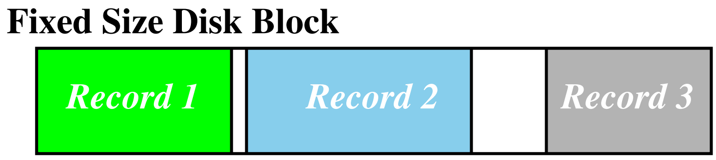
- Variable format ⇒ record "self describing".
  - e.g. variable format and length



- More generally get hybrid schemes
  - e.g. record header with schema id, length
  - e.g. fixed record with variable suffix

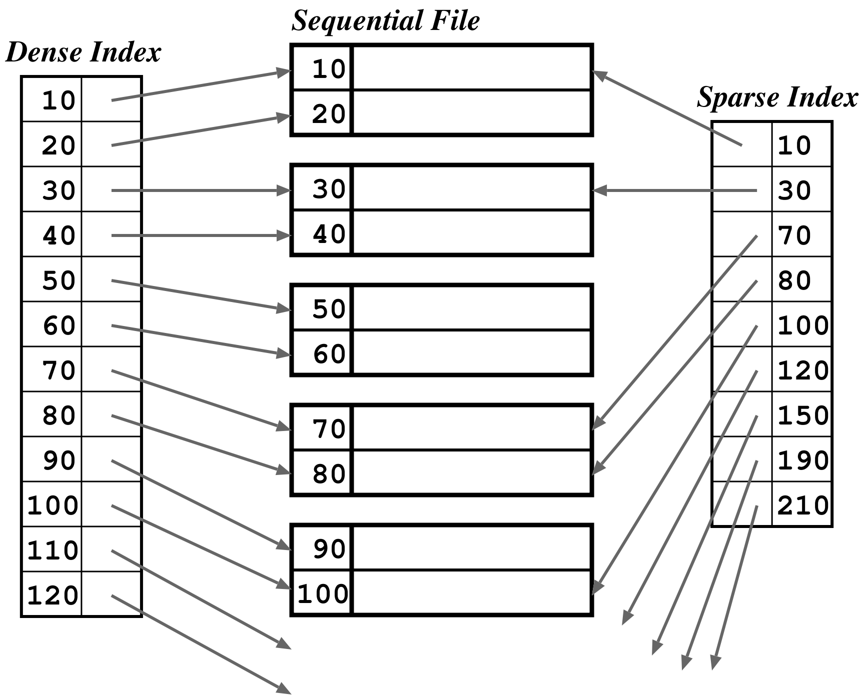
Storing Records in Blocks

- Ultimately storage device provided blocks
- Could store records directly in blocks:



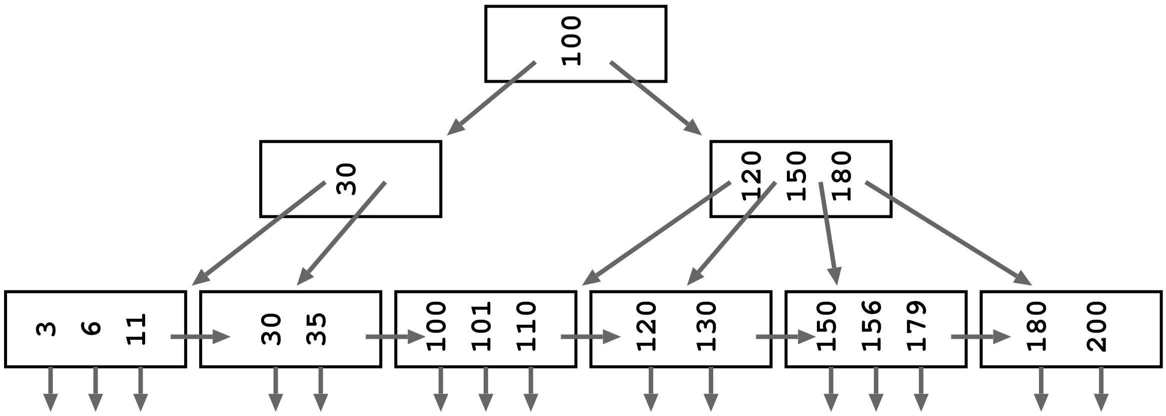
  - fixed size recs: no need to separate
  - variable length ⇒ separation marker?
  - better: offsets in block header
- What about *spanning* multiple blocks?
  - may need if variable length record grows
  - certainly need if |record| > |block|
  - can impl with pointers at end of blocks
- Should we mix record types within a block?
  - *clustering* benefit for related records
  - usually too messy ⇒ just co-locate
- In which order should we store records?
  - often want sequential within block (and 'file')
  - (makes e.g. merge-join easier)

Efficient Record Retrieval (1)



- Assume have sequential file ordered by key.
- Can build dense or sprase index:
  - sparse: smaller $\Rightarrow$ more of index in memory
  - dense: existence check without accessing fils
  - sparse better for inserts
  - dense needed for secondary indexes
  - multi-level sparse also possible
- Can use block pointers ($<$ record pointers)
- If file actually contiguous, can omit!
- But: insertions and deletions get messy. . .

Efficient Record Retrieval (2)



- Eschew sequentiality – focus on *balance*
- Good example is a *B+-Tree*
  - All nodes have n keys and $n+1$ pointers
  - In non-leaf nodes, each pointer points to nodes with key values $<$ right key, $\geq$ left key
  - In leaves, point direct to record (or across)
- *Balanced* tree (i.e. all leaves same depth):
  - keep $\geq \lceil (n+1)/2 \rceil$ in non-leaves
  - keep $\geq \lfloor (n+1)/2 \rfloor$ data pointers in leaves
- Search is easy and fast:
  - binary search at each level – $O(\log(n))$
  - with N records, height $\log_n N$

More on B+-Trees

- Insertion fairly straightforward:
  - space in leaf ⇒ sorted
  - if no space somewhere ⇒ split
  - if root split ⇒ new root (and new height)
- Deletion a bit hairy:
  - if min bounds not violated ⇒ easy
  - otherwise need to either:
    - redistribute keys (and propagate upward), or
    - coalesce siblings
  - many implementation don't coalesce. . .
- Buffering: is LRU a good idea?
  - No! Keep root (and higher levels) in memory
- Can we do better?
  - also get *B-Tree* : avoid key duplication
  - i.e. interior nodes also point to records
  - smaller, & faster lookup (at least in theory)
  - but: deletion even more difficult
  - but: leaf and non-leaf nodes different sizes

Aside: Spatial Indexes

- Spatial data pertains to the space occupied by objects (e.g. points, lines, surfaces, volumes)
- Real life: roads, cities, countries, internet
- Challenging for conventional DBMS:
  - inherently highly dimensional (and may be continuous) $\Rightarrow$ cannot simply store as relation.
  - want to express spatial queries (e.g. close to, encompasses, intersects)
- How can we index such data?
  - B-tree cannot handle high dimensionality
  - hashing cannot handle range queries
- Two main approaches:
  - balanced trees in spatial occupancy (R-trees)
  - multi-dimensional $\sim$hashing (grid files)
- Detailed discussion beyond scope of this course; see papers on web page for more info.

Postgres DBMS

- Postgres: developed at UCB between 1989 – 19991
- Postgres motivation:
  - Old DBMS *data management* only (fixed format records, traditional transactions & queries)
  - New need for 'object' management (bitmaps, vector graphics, free text, etc)
  - e.g. CAD, general knowledge management
- Postgres used set-oriented POSTQUEL:
  - small number of concepts ⇒ simple for users
  - embedded directly in programming language.

  ✔ variable persistence, standard control flow

  ✘ big memory footprint
- Handles base types, ADTs, composite types, complex objects, and path expressions
- Used some novel techniques in backend design and implementation.

Postgres Implementation

- Every previous system used a write-ahead log
- Postgres wanted to do something different:
  - "no-overwrite" storage manager
  - i.e. leave old version of record in data base
  - 'log' now just 2-bits per transaction stating if in progress, committed, or aborted
- Benefits of this approach:
  - abort is very cheap (nothing to undo)
  - recovery is very cheap (same reason)
  - "time-travel": support historic queries
- But there are a few (!) problems:
  - must flush new records to disk on commit
  - may need multiple indices (or R-trees?)
  - disk fills up $\Rightarrow$ flush to write-once media
  - but 'cleaner' didn't run under load :-(
  - time travel queries hard to express
- 1995 saw Postgresql (SQL version):
  - some improvements to storage manager
  - free and useful system for small databases

Operating Systems and Databases (1)

OSes not suited to DBMS (Stonebraker CACM '81):

- Extra data copies to/from disk
- Buffer replacement:
  - most OSes use LRU but DBMS accesses are:
    1. sequential access to blocks without re-reference (build hash table, sort)
    2. sequential access to blocks with cyclic reference (inner loop of nested join) – this *kills* LRU
    3. random access to blocks without re-reference (should discard immediately)
    4. random access to blocks with non-zero probability of re-reference
  - only case 4. works ok with LRU
  - but DBMS *knows* all of this
- (similar arguments apply to prefetching)
- No support for synchronous reorder barriers

⇒ DBMSes end up doing their own buffer pool management in user space

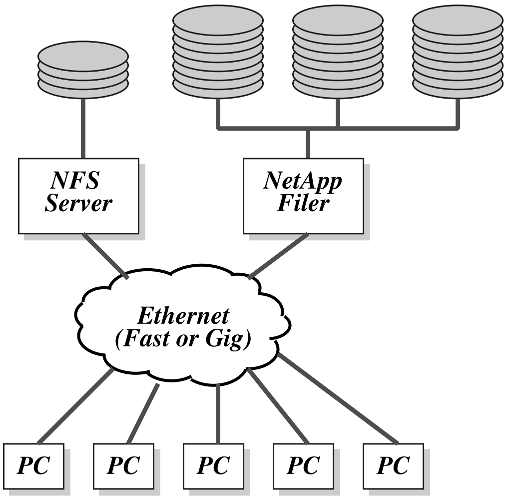
Operating Systems and Databases (2)

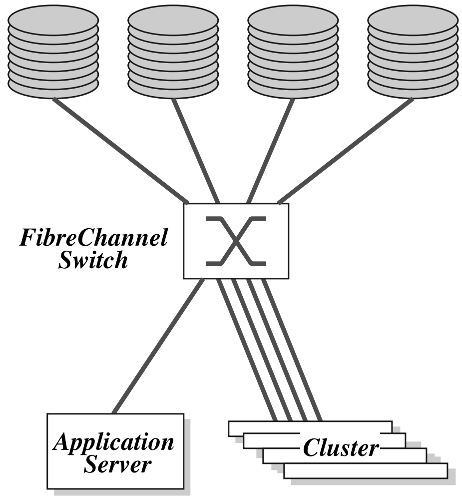
- Problems with user-space buffer management:
  - want extents, not variable byte-length files
  - buffer pool in virtual memory $\Rightarrow$ poss poor interaction with VM system ("double paging")
- Other problems noted in paper:
  - multiple trees (directory, inode, B-trees) wasteful
  - multi-process DBMS can suffer from priority inversion if have user-space locks in DBMS
  - but single process DBMS more complex – and cannot benefit from multiprocessors.
- Stonebraker suggests OS accept 'hint' from DBMS
- So where have we got in 20+ years?
  - Not very far. . .
  - Some OS give DBMS raw disk/partition access
  - NTFS directly supports [infinite] write-ahead log.
  - Some extensibility *research* in this area (e.g. page-replacement or buffer cache callbacks)
- Overall: still serious engineering effort to get good DBMS perf without tight OS integration.

Distributed Storage

- Filesystems/DBMS want big, fast, reliable disks.
- Cheaply achieve this using multiple disks, e.g.
  - RAID = Redundant Array of Inexpensive Disks
  - better performance through *striping*
  - more reliable via *redundancy*:
    - simple mirroring
    - generalised parity (Reed-Solomon)
    - variable length erasure codes (IDA)
  - key benefits: scalability, fault tolerance
- Even better: make storage *distributed* – i.e. separate data management from apps / servers
- Why is this a good idea?
  - centralised data management
  - even more scalability
  - location fault tolerance
  - mobility (for access)
- What are the options here?

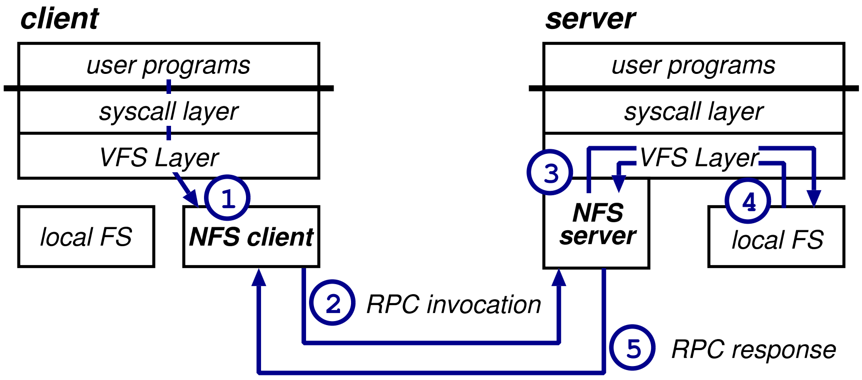
NAS versus SAN





- Two basic architectures:
  - *lhs*: *Network Attached Storage* (NAS)
  - *rhs*: *Storage Area Networks* (SANs)
- NAS distributes storage at the FS/DB level:
  - runs over TCP/IP (or NetBIOS) network
  - exports NFS, CIFS, SQL, . . .
- SAN distributes storage at the *block* level:
  - runs over fibre channel
  - accessed via encapsulated SCSI
  - filesystem/DBMS run on hosts
- NAS better general purpose, SAN more specialised

Network File-Systems



- NAS normally accessed by network file-system:
  - client-server (e.g. NFS, SMB/CIFS, etc)
  - mostly RPC-based at some level
- NFS originally (V2, 1989) designed to be stateless:
  - no record of clients or open files
  - no implicit arguments to requests
  - no write-back caching on server
  - requests idempotent where possible
  - only hard state is on [server] local filesystem
- Statelessness good for recovery, but:
  - synchronous disk write on server sucks
  - cannot help client caching
- More recent NFS versions are an improvement. . .

NFS Evolution

NFS V3 (1995) brings mostly minor enhancements:

- scalability:
  - remove limits on path and file name lengths,
  - support 64-bit file offsets
  - allow large (>8KB) transfer size negotiation
- explict asynchrony:
  - server can do asynchronous writes
  - client sends `commit` after some # of writes; must keep cached copy until successful commit
- enhanced operations (symlink, readdirplus)

NFS V4 (RFC3530, April 2003) a major rethink:

- single, **stateful** protocol (including mount, lock)
- TCP – or at least reliable transport – only
- explicit `open` and `close` operations
- *share reservations* (file level revocable 'locks')
- *delegation* (sever gives client revocable autonomy)
- arbitrary compound operations

Actual success yet to be seen. . .

AFS and Coda

- AFS (1983+) developed concurrently with NFS:
  - purely remote: even local access via client
  - persistent client caching with delegation on a directory basis through callbacks
  - increment file version # on every change
  - live replication and relocation of volumes
- Free 'OpenAFS' implementation available.
- Coda (1987+):
  - descendant of AFS (based on AFS2)
  - support for *disconnected operation*
  - Venus cache manager operates in 1 of 3 modes:
    * *hoarding*: when connected tries to cache copies of "working set" for user
    * *emulating*: when disconnected, services what it can and notes any updates in LRVM
    * *reintegrating*: when reconnected, performs (assisted) conflict resolution
- Extensions to Coda support *weakly connected* – e.g. wireless – operation (Satya et al, SOSP 95)

LBFS: An Alternative for Wide Area

LBFS (SOSP 01) addresses network file system access over *low-bandwidth* (wide area, wireless) networks.

- more and more people have laptops
- remote CIFS or NFS (even V4) pretty suckful
- key idea: *aggressive compression* on wire/air.

How can we compress an order of magnitude better?

- don't just look at individual blocks / files
- maintain persistent cache on client

⇒ huge amount of shared information!

In a bit more detail:

- server divides its files into chunks, and for each chunk computes a secure (SHA-1) hash
- client does the same
- if e.g. client wants to write a (portion of) a file, actually just transfers hashes for relevant chunks
- if server already has chunks, can avoid transfer; replies requesting missing chunks (if any)
- reads proceed in an similar fashion

LBFS: Exploiting Inter-File Similarities

- As described, LBFS clearly going to work, but performance gains may not be huge:
  - e.g. consider edits in place
  - e.g. consider fetch of file never seen before
- So here's the clever bit:
  - file chunk boundaries chosen based on *contents*
  - compute Rabin fingerprint over every overlapping 48-byte region of every file
  - if low-order 13 bits match a magic value, deem this to be a chunk boundary
  - assuming uniformity, expected chunk size is 8K
  - use low (2K) and high (64K) thresholds to deal with pathological cases
- Now we get 'hits' if contents appear *anywhere* in *any* file that we know about
- Tested on fairly realistic workloads:
  - up to 20% redundancy between unrelated files
  - LBFS 10x better than NFS / CIFS / AFS
- Q: what about hash collisions?

Serverless File-Systems

- Modern trend towards *serverless file-systems*:
  - no discrimnated "server"
  - all nodes hold some data
  - (think P2P in the local area)
- e.g. xFS (Berkeley):
  - have clients, cleaners, managers, storage servers
  - any machine can be [almost] any subset of above
  - to read file:
    * lookup manager in globally-replicated map
    * contact manager with request
    * manager redirects to cache or disk (imap)
  - to write file:
    * obtain write token from manager
    * append all changes to *log*
    * when hit threshhold, flush to *stripe group*
- xFS approx 10x better than NFS. Why?
  - co-operative caching
  - parallelism via software RAID (striping)
  - avoid read-modify-write by using log-structure
  - managers replicated for fault tolerance

JetFile: Serverless Internet Storage

- The Jetfile distributed FS (OSDI 99) aimed to:
  - support shared personal file access in heterogeneous Internet (LAN and WAN)
  - provide approx performance of local file system
- Basic implementation:
  - uses scalable reliable multicast (SRM):
    * receiver driven: multicast request for data, receive (hopefully!) 1 or few responses.
    * version numbers on all data ⇒ receiver can retry to guarantee eventual delivery
  - files named by `(org, vol, fileID, version)`:
    * hash first three to get *file address*
    * to retrieve file first multicast data_request
    * pick reponse and use unicast for remainder.
  - updates handled by *versioning*
    * write-on-close semantics (bumps version)
    * client now 'server' for new version
    * explicit version requests, plus "current table" multicast over per-volume channel
- Overall: approx local FS (if warm cache. . .)

File-systems for SANs

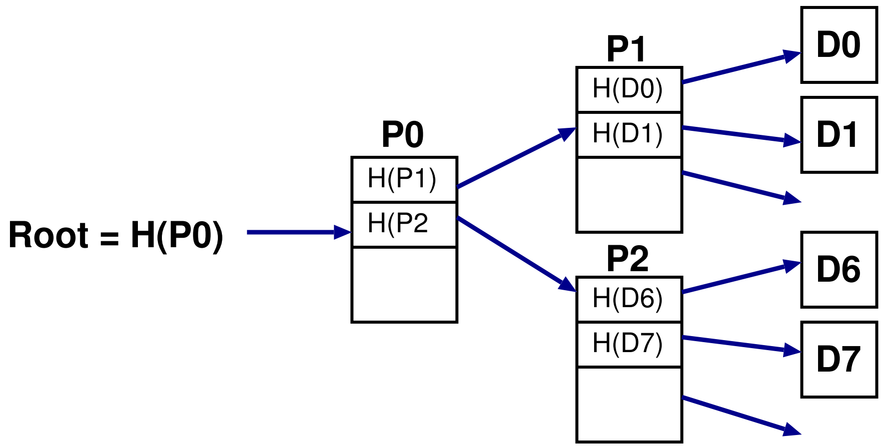
- Recall: SAN has pool of disks accessed via iSCSI
- With multiple clients $\Rightarrow$ need coordination
- Two ways to build a shared disk file system (SDFS)
  1. asymmetric: add a metadata manager:
     - exclusive access to metadata disk[s]
     - clients access data disks directly
  2. symmetric:
     - clients access data and metadata directly
     - distributed locking used for synchronisation
- Asymmetric simpler, but less scalable/fault-tolerant
- Symmetric systems becoming mature:
  - e.g. GFS, open source project for linux
  - bottom half: network storage pool driver:
    * combines all disks into single 'address space'
    * supports striping for performance/reliability
  - top half: file-system
    * almost standard unix structure (inodes, etc)
    * device locks and global locks for synch
- NASD work (CMU) tries to make SDFS easier. . .

Network Attached Secure Disks

- Basic idea: a less stupid SAN.
- Still have network attached disks, but:
  - disks export variable-length *object* interface:
    * create, read, write, etc
    * can use to store e.g. files or DBMS tables
    * disk manages block allocation and layout
  - integrity/privacy available on transfers
  - access checks on disk:
    * disk and file manager share secret key
    * after file-system-specific checks, file manager issues derived *capabilities* to clients

  ⇒ clients can securely access disk directly.
- Middle ground between regular 'dumb' network attached disks and network file systems
- Advantages:
  - data path operations fast and secure
  - offloads work from file manager: e.g. NFS server using NASD requires 10x CPU cycles.
  - multiple NASDs can be accessed in parallel
- But: less allocation control, key mgt, `enoexist`

Venti: Distributed Archival Storage



- Archival storage typically on tape / optical
- Venti (FAST 02) considers using magnetic disk:
  - cheap, ubiquitous, fast
  - but: subject to overwrite. . .
- So use software to provide *immutable* storage
  - every file update a new version
  - c/f Plan-9 in lb, Elephant (SOSP 99)
  - but: what about directories?
- Key idea: use *content hashes* to access everything
  - file contents are a set of blocks
  - inode is table/tree of block content hashes
  - directories map names to inode content hashes

Venti (Continued)

- Gives you some nice properties:
  - if e.g. a file changes in leaf directory, then some or all its block hashes change ⇒ inode changes
  - hence inode content hash changes, and so some directory block(s) change
  - and so on all the way up the tree
- So root hash uniquely captures filesystem snapshot!
- Same applies to arbitrary subdir: can build a very cheap 'tar' replacement (just store hash in text file)
- Not without design/implementation challenges:
  - all addressing now via hashes ⇒ need some way to map from hash to actual block location
  - Venti uses log-based (append-only) storage:
    - ∗ store blocks sequentially in an arena, each with a header including its fingerprint (hash)
    - ∗ trailer on arena points back to blocks
    - ∗ simple, but doesn't help with hash search
    - ⇒ build index to map hashes to block headers
  - index can be a bottleneck, so also add a index cache plus a regular block cache

Summary & Outlook

We've seen a selection of systems topics:

- Distributed and persistent virtual memory
- Capability systems
- Microkernels (Mach, L3/L4, [EROS])
- Virtual machine monitors (VM/CVS, Disco, VMWare, Denali, Xen)
- Extensible operating systems (SPIN, Vino, Exokernel, Nemesis)
- Database storage & retrieval (issues, consistency, records, blocks, indices, Postgres, OS issues)
- Distributed storage and filesystems (NAS, SANs, NFS, LBFS, xFS, NASD, etc)

Lots more research ongoing in most of above areas.

Next section of course: **scalable synchronization**.