Advanced Systems Topics
Part I of III

Steven Hand

Lent Term 2003

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

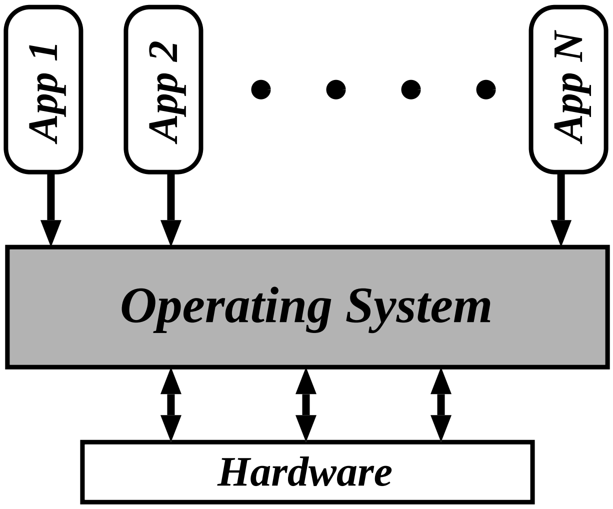
Course Outline

- Part I: Advanced Operating Systems [SMH, 6L]
  - Local & Distributed Virtual Memory
  - Capability Systems and Microkernels
  - Virtual Machine Monitors
  - Extensibile Operating Systems
  - Filesystem & Database Storage
- Part II: Scalable Synchronization [TLH, 4L]
  - Introduction (systems with 10K threads)
  - Architectures and Algorithms
  - Implementing Mutual Exclusion
  - Programming without Locks
- Part III: Peer-to-Peer Systems [JAC, 5L]
  - Client-server versus P2P
  - Case studies (including napster, gnutella, RON, Freenet, Publius)
  - Middleware (including Chord, CAN, Tapestry, JXTA, ESM, Overcast)
  - Applications (Storage, conferencing).

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/2002/AdvSysTopics/`

Operating System Overview



An operating system is a collection of software which:

- *securely multiplexes resources*, i.e.
  - protects applications from each other, yet
  - shares physical resources between them.
- provides an abstract *virtual machine*, e.g.
  - time-shares CPU to provide virtual processors,
  - allocates and protects memory to provide per-process virtual address spaces,
  - presents h/w independent virtual devices.
  - divides up storage space by using filing systems.

And ideally it does all this *efficiently* and *robustly*.

Hardware Support for Operating Systems

Recall that OS should *securely* multiplex resources.
⇒ we need to ensure that an application cannot:

- compromise the operating system.
- compromise other applications.
- deny others service (e.g. abuse resources)

To achieve this efficiently and flexibly, we need hardware support for (at least) *dual-mode operation*.

Then we can:

- add memory protection hardware
  ⇒ applications confined to subset of memory;
- make I/O instructions privileged
  ⇒ applications cannot directly access devices;
- use a *timer* to force execution interruption
  ⇒ OS cannot be starved of CPU.

Most modern hardware provides protection using these techniques (c/f Computer Design course).

Hardware / Software Co-Design

Operating Systems are key candidates for *co-design*:

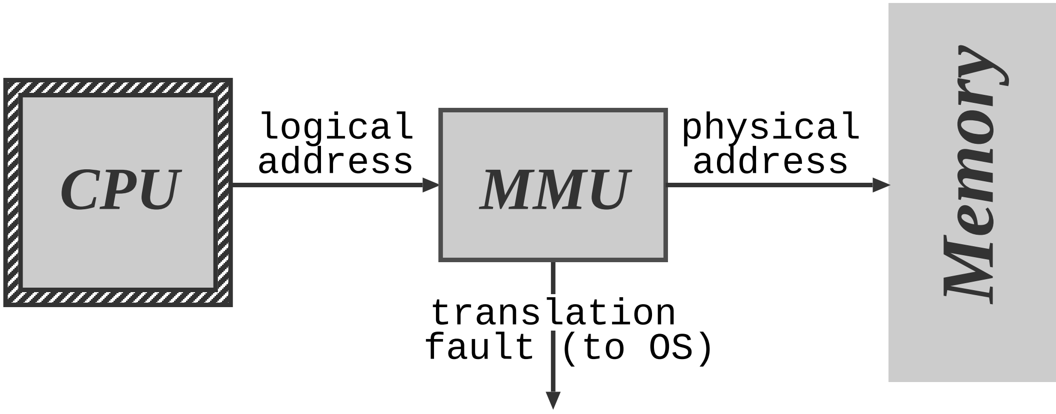
- see what hardware support is available
- build OS as best you can
- push back for new hardware features

Happens seldom nowadays, but was more prevalent.

Coming up in next few lectures

- Local and distributed virtual memory
- Systems with funky hardware:
  - Multics: complex mother of Unix
  - The CAP: capability system
- The microkernel revolution(s)
  - Mach: flexible and slow
  - L3/L4: re-engineering the microkernel
  - EROS: software capabilities
- Resurgence of virtual machine monitors
- The quest for extensibility

Logical and Physical Addresses



Recall from IA:

- Run-time mapping from logical to physical addresses performed by special h/w (the MMU).
- If we make this mapping a *per process* thing then:
  - Each process has own *address space*.
  - Allocation problem split:
    - ∗ virtual address allocation easy.
    - ∗ allocate physical memory 'behind the scenes'.
  - Address binding solved:
    - ∗ bind to logical addresses at compile-time.
    - ∗ bind to real addresses at load time/run time.
- Variants: segmentation, capabilities, **paging**.

Virtual Addresses allow Demand Paging

When loading a new process for execution:

- create its address space (e.g. page tables, etc)
- 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
   - 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

Scheme described above is *pure* demand paging:

- never brings in a page until required ⇒ get lots of page faults and I/O when process begins.
- hence many real systems explicitly load some core parts of the process first

Page Replacement

- When paging in from disk, we need a free frame of physical memory to hold the data we're reading in.
- In reality, size of physical memory is limited $\Rightarrow$
  - need to discard unused pages if total demand for pages exceeds physical memory size
  - (alternatively could swap out a whole process to free some frames)
- Modified algorithm: on a page fault we
  1. locate the desired replacement page on disk
  2. to select a free frame for the incoming page:
     (a) if there is a free frame use it
     (b) otherwise select a *victim page* to free,
     (c) write the victim page back to disk, and
     (d) mark it as invalid in its process page tables
  3. read desired page into freed frame
  4. restart the faulting process
- Can reduce overhead by adding a 'dirty' bit to PTEs (can potentially omit step 2c above)
- Question: how do we choose our victim page?

Page Replacement Algorithms

- First-In First-Out (FIFO)
  - keep a queue of pages, discard from head
  - performance difficult to predict: no idea whether page replaced will be used again or not
  - discard is independent of page use frequency
  - in general: pretty bad, although very simple.
- Optimal Algorithm (OPT)
  - replace the page which will not be used again for longest period of time
  - can only be done with an oracle, or in hindsight
  - serves as a good comparison for other algorithms
- Least Recently Used (LRU)
  - LRU replaces the page which has not been used for the longest amount of time
  - (i.e. LRU is OPT with -ve time)
  - assumes past is a good predictor of the future
  - Q: how do we determine the LRU ordering?

Implementing LRU

- Could try using *counters*
  - give each page table entry a time-of-use field and give CPU a logical clock (counter)
  - whenever a page is referenced, its PTE is updated to clock value
  - replace page with smallest time value
  - problem: requires a search to find min value
  - problem: adds a write to memory (PTE) on every memory reference
  - problem: clock overflow
- Or a *page stack*:
  - maintain a stack of pages (doubly linked list) with most-recently used (MRU) page on top
  - discard from bottom of stack
  - requires changing 6 pointers per [new] reference
  - very slow without extensive hardware support
- Neither scheme seems practical on a standard processor $\Rightarrow$ need another way.

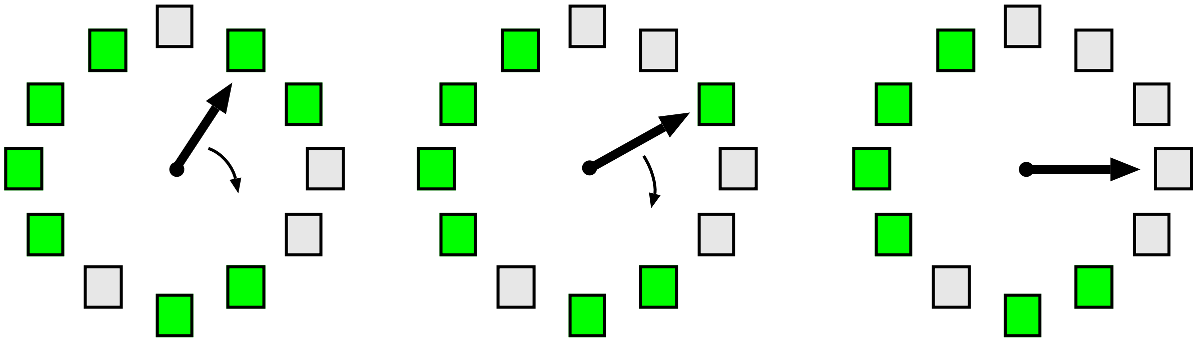
Approximating LRU (1)

- Many systems have a *reference bit* in the PTE which is set by h/w whenever the page is touched
- This allows *not recently used* (NRU) replacement:
  - periodically (e.g. 20ms) clear all reference bits
  - when choosing a victim to replace, prefer pages with clear reference bits
  - if also have a *modified bit* (or *dirty bit*) in the PTE, can extend MRU to use that too:

Ref?	Dirty?	Comment
no	no	best type of page to replace
no	yes	next best (requires writeback)
yes	no	probably code in use
yes	yes	bad choice for replacement

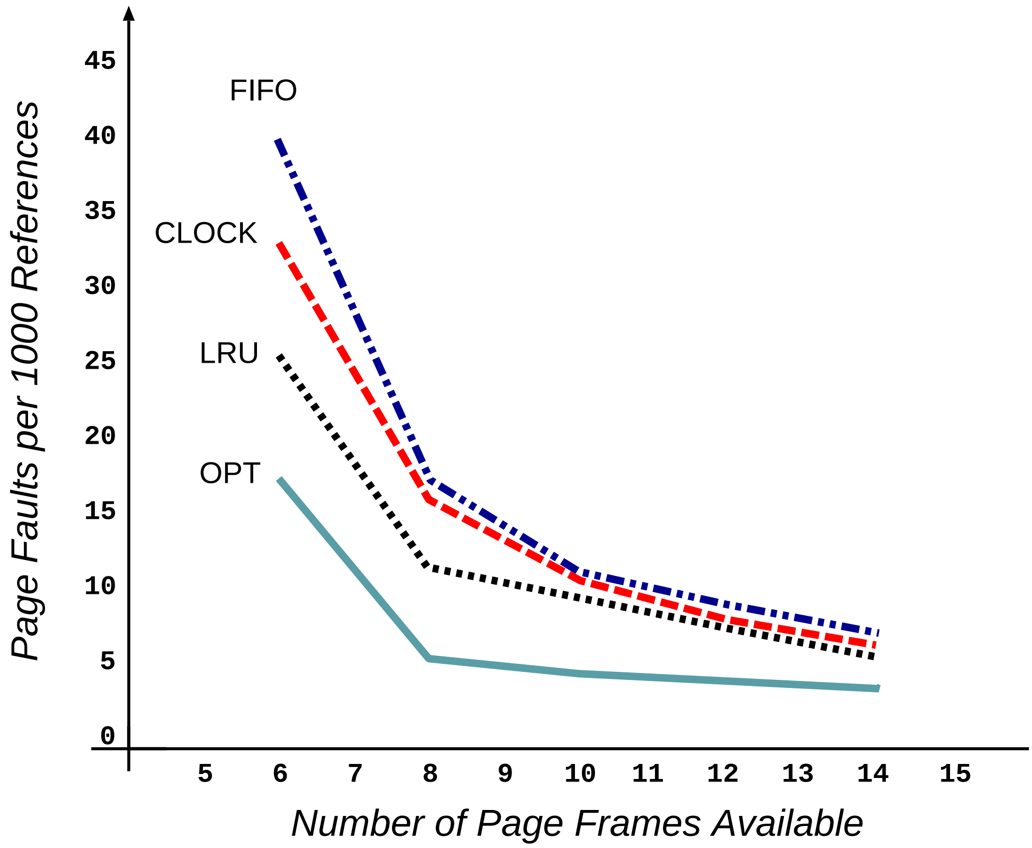
- Or can extend by maintaining more history, e.g.
  - for each page, the operating system maintains an 8-bit value, initialized to zero
  - periodically (e.g. 20ms) shift reference bit onto high order bit of the byte, and clear reference bit
  - select lowest value page (or one of) to replace

Approximating LRU (2)



- Popular NRU scheme: *second-chance FIFO*
  - store pages in queue as per FIFO
  - before discarding head, check its reference bit
  - if reference bit is 0, discard, otherwise:
    * reset reference bit, and
    * add page to tail of queue
    * i.e. give it "a second chance"
- Often implemented with a circular queue and a current pointer; in this case usually called *clock*.
- If no h/w provided reference bit can emulate:
  - to clear "reference bit", mark page no access
  - if referenced ⇒ trap, update PTE, and resume
  - to check if referenced, check permissions
  - can use similar scheme to emulate modified bit

Performance Comparison



Graph plots page-fault rate against number of physical frames for a pseudo-local reference string.

- want to minimise area under curve
- FIFO can exhibit Belady's anomaly (although it doesn't in this case)
- getting frame allocation right has major impact. . .

Shared Virtual Memory

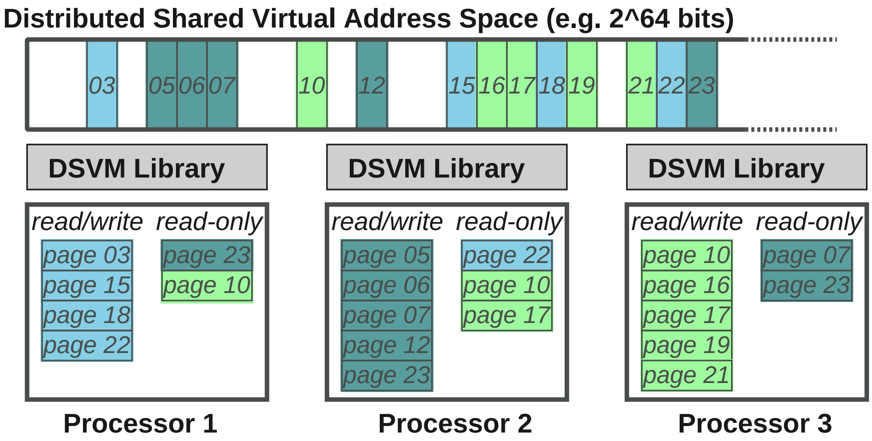
Memory models for parallel programming:

- 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

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)

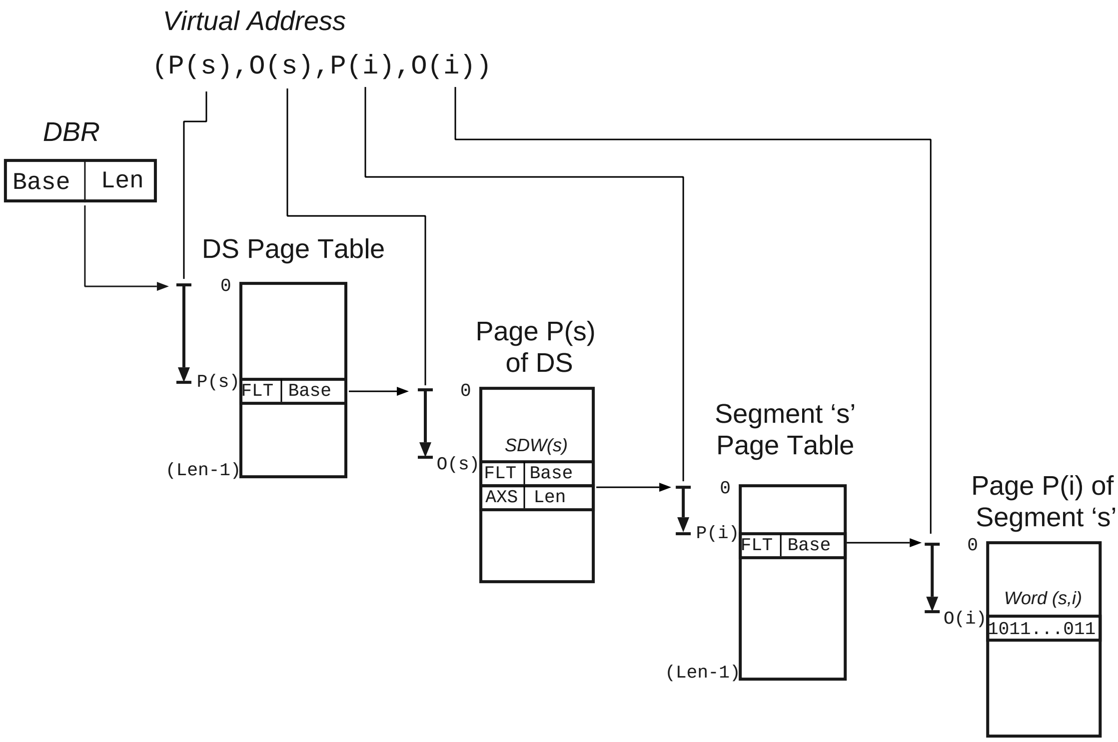
- Still potentially expensive, e.g. false-sharing:
  - $P1$ owns p, $P2$ just has read-access
  - $P1$ writes $p \Rightarrow$ copies to $P2$
  - but $P2$ doesn't care about this change
- Reduce traffic by using weaker memory consistency:
  - so far assumed sequential consistency:
    * every read sees 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!

MULTICS

- Developed 1964– by MIT, GE and AT&T Bell Labs
- Based on the GE 645 hardware:



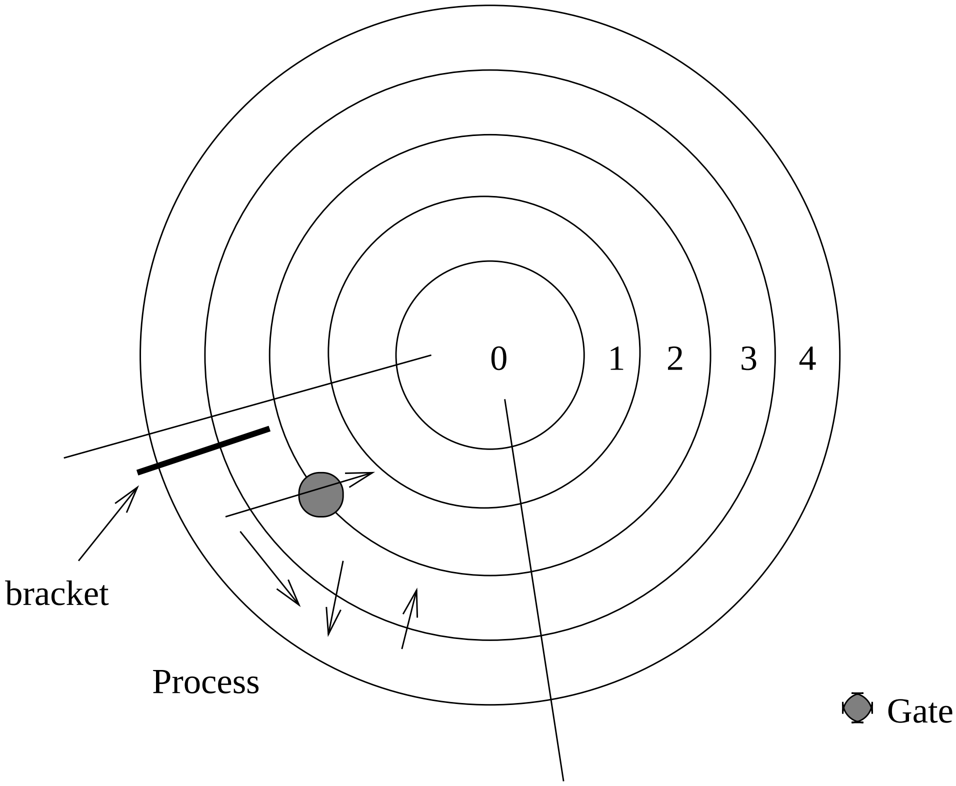
  - extensive support for memory management
  - Multics was first system to use paged segments

- Key contribution of Multics: system security
- Offered protection in two ways:
  1. Per Segment
  2. Concentric Rings of privilege

Multics “File System”

- No filesystem *per se*; user saw large number of orthogonal linear regions of the virtual address space, each backed by a secondary store.
- Segments created by users, and remained available until explicitly deleted.
- Have a tree of directories and non-directories, similar to Unix
- Each directory contains a set of **branches** – points to and describes a ‘file’
- (a file is equivalent to a memory **segment**)
- A branch contains a set of attributes for a segment:
  - Unique Segment ID (assigned by FS)
  - An access control list (ACL), stored as a linked list of entries, one per UID
  - UID structured as $< user >< project >< \ldots >$
  - A ring **bracket** and limit:
    - ACL applies only within bracket
    - bracket is a pair $(b1, b2)$, limit is a ring value l
  - A list of gates – procedure entry points

Rings and Protection



- Initially MULTICS provided for up to 32 rings of system privilege and 32 user privilege
- In practice only 8 ever implemented
- To minimize ring crossings, also had *access brackets* = band of rings
- Reference by procedure in bracket to segment within bracket causes no ring crossing faults
- Allow useful procedures to be invoked from a range of rings (since [r,e], procedures are ‘safe’)

Ring Assignment

- On the 'need to know' principle
- On basis of the degree of likely damage
- Three rules:
  1. Process residing in ring j should be able to call procedures in rings $j + 1 \ldots n$
  2. Process in ring j should have limited access to procedures in rings $i < j$
  3. Process in ring j should have NO access to data segments in rings $i < j$
- To implement the protection need to detect ring crossings (both call and return)
- Achieved by making the process 'address fault' when it calls a procedure at a different ring level

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
  - passed as a parameter
  - transferred to grant access
  - loaded into special registers
  - refined
- the object name may include a type
  - rights, etc., may be interpreted by type manager

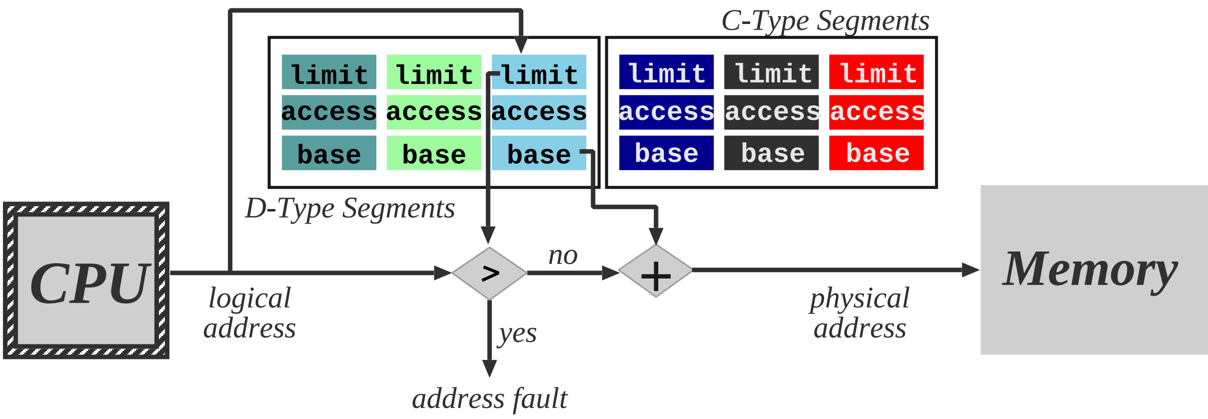
What is an object name?

- path name in file store? 64-bit unique identifier?
- not much use for hardware implementation
- hence often use aliases for real capabilities and an associative store for fast matching.

The Cambridge CAP Computer

- Developed at Cambridge starting in 1970
- Major designers were Prof R.M. Needham, Prof M.V Wilkes and Prof D.J. Wheeler
- Recognises need for hardware memory protection on a fine grained level:
  - e.g. CTSS: separate memory for supervisor code; could only be accessed when in privileged mode.
  - more flexibility required
- The CAP (and similar systems) are different – no control over **who** writes into the regsiters, but tight control over **what** can be written 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 a segment out of the 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 reference
- Process can now 'hand on' a selection of memory access privileges to its sub-processes
- Small number of absolute capabilities makes management easier

Resource Lists (I)

- The set of co-ordinator capabilities is located in a segment called the **master resource list** MRL
- The set of capabilities for each process is located in its **process resource list** PRL
- PRLs contain relative capabilities which refer to absolute capabilities in the MRL
- All absolute capabilities are located in the MRL
- Capabilities directly available to a process at any time are contained in a variety of capability segments including:

  G Global capabilities – remains unchanged through life of process

  P Capabilities for code (Procedure) segments

  I Capabilities for stack segments

  R Capabilities for data segments associated with procedure

Resource Lists (II)

- Further capability segments support ENTER:

N,A Top two capabilities on the C-stack:

1. ENTER pushes the stack down: old N capability segment becomes new A capability segment
2. MAKEIND instruction creates the next N capability segment on top of stack, ready for next call
3. RETURN pops the stack and reverses the effect of ENTER

- Address format of a word in memory is thus a tuple (CS, CS-OFF, SEG-OFF)
- CS – the capability segment containing the capability for the segment concerned
- CS-OFF – offset in the capability segment of the capability for the segment required
- SEG-OFF – offset in the segment concerned of the word being addressed
- The tuple (CS, CS-OFF) is the capability specifier

Setting up Sub-processes

- Co-ordinator must acquire and set up a segment which will become PRL of new sub-process
- Must set up the appropriate capabilities needed by the sub-process
- Co-ordinator then executes ENTER SUBPROCESS (ESP) instruction with argument a capability for the newly created PRL
- Microcode saves state of co-ordinator and activates sub-process
- Return to co-ordinator is via the ENTER COORDINATOR (EC) instruction
- A user process can act as a co-ordinator for its own sub-processes, and so on...

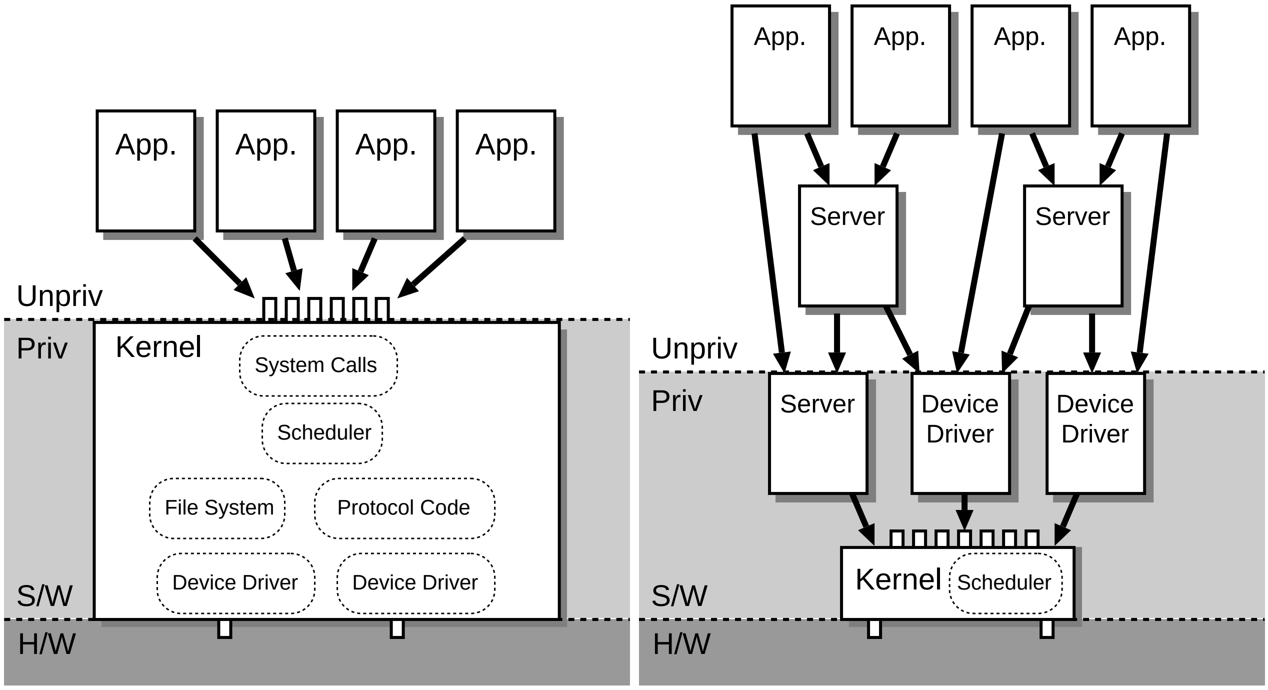
Capability Loading

- CAP has 64 capability registers in capability store
- If a non-resident capability is referenced in an instruction, the microcode loads it into the store
  1. read capability from the capability seg where it resides – this is a *relative* capability (ref to PRL)
  2. read the capability from the PRL – this has a reference to a capability in the superior process
  3. algorithm proceeds down chain until it reaches the absolute capability in the MRL
  4. the final evaluated capability is loaded into the capability store
  5. the instruction is then restarted and execution continues or a trap occurs
- When overwriting entries the capabilities for the PRL and process base are never overwritten
- Must also keep evaluated capabilities for MRL and process base of co-ordinator
- Various things can go wrong during evaluation – all handled in software by the process co-ordinator

The CAP: Summary of Features

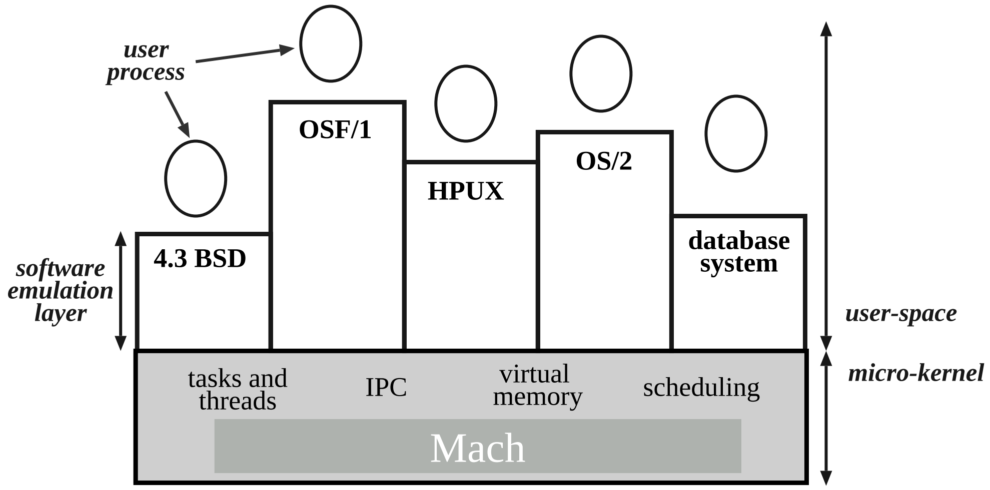
- The CAP was an architectural innovation and extremely successful – for details see the book
- Segment swapping:
  1. separate capability and data segs leads to large number of segments
  2. segmentation allows protection of arbitrarily sized objects
- Local Naming:
  1. capabilities are relative to PRL of process (i.e. its capability segs)
  2. thus every instance of a protected procedure has its own address space
- Control of Sub-processes:
  - can create an infinite hierarchy of co-ordinators
  - this is something we'd *love* today
- 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

Microkernel Operating Systems



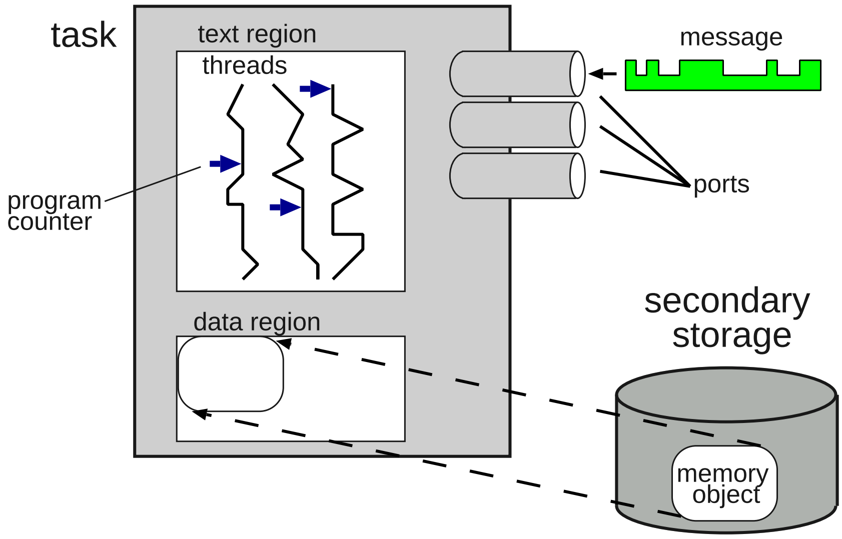
- New concept in early 1980's:
  - "kernel" scheme (*lhs*) considered complex
  - ⇒ try to simplify kernel, build modular system
  - support multiprocessors, distributed computing
- Re-engineered OS strucuted (*rhs*)
  - move functions to user-space servers
  - access servers via some *interprocess communication*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

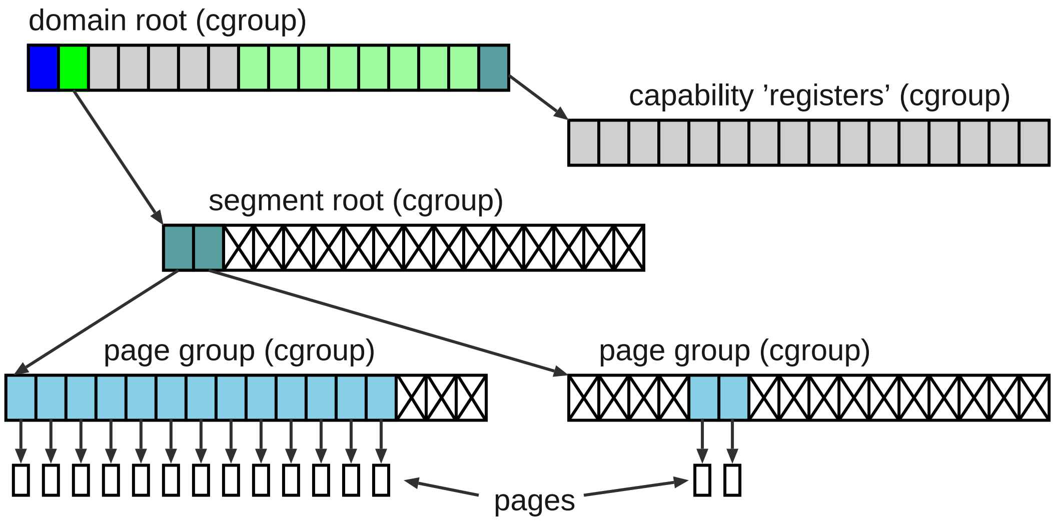
- Q: are these micro-benchmarks useful?
- Q: what about portability?

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

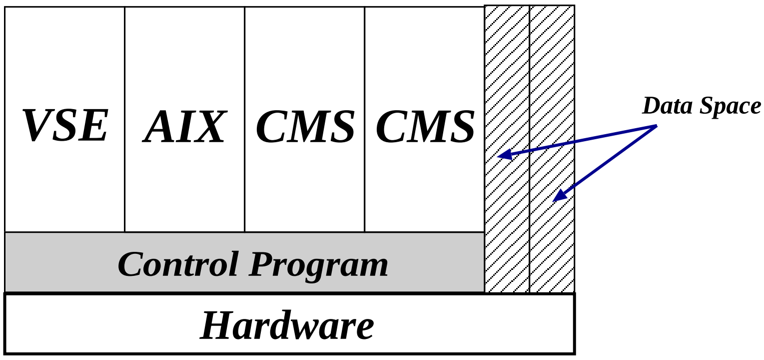
- Persistence achieved by flushing objects to disk:
  - circular log used for checkpointing
  - eventually log entries migrate to home location
- Before using capabilities, they must be *prepared*
  - if necessary bring object referred to into memory
  - modify capability to point to object table
  - 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.

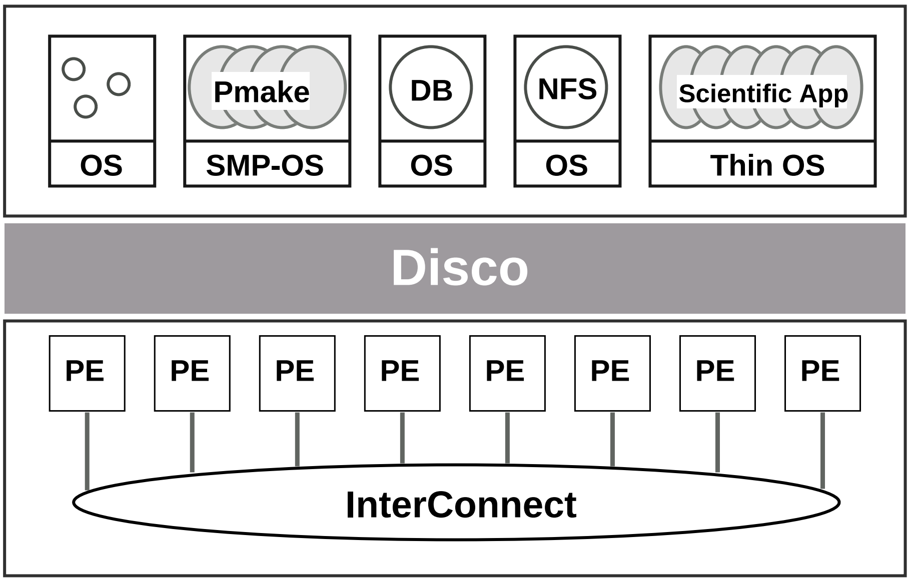
- Use a “hypervisor” (beyond supervisor, i.e. beyond a normal OS) to multiplex multiple OSes.
- (NB: hypervisor $\equiv$ virtual machine monitor)
- Made popular by IBM’s VM/CMS (1970’s)
- Idea regained popularity in mid 90’s:
  - e.g. Disco uses a VMM to make it easier to write operating systems for 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.

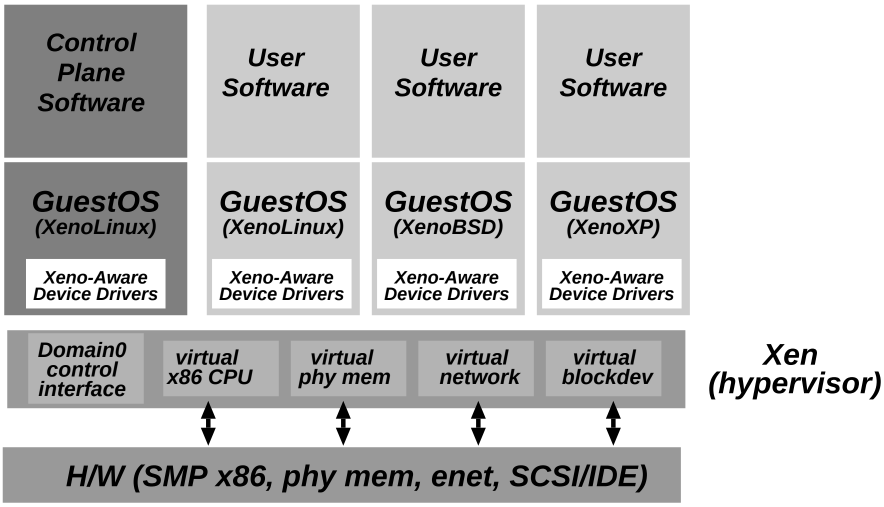
VMWare

- 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

XenoServer 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

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
  - Machine-level firewalling
    - 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
    - NSA/VMWare actively pursuing this

Extensibility

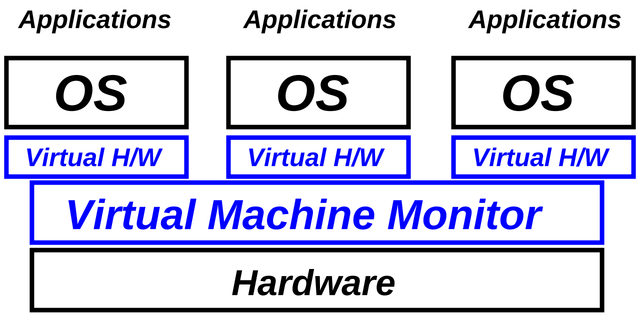
What's it about?

- 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.

  ⇒ 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:

```
                         and 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

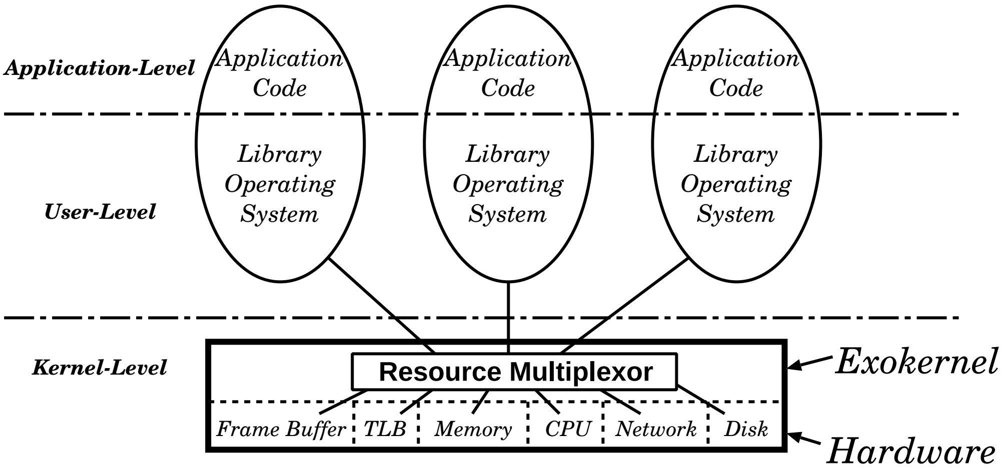
Kernel-level schemes can get very complex $\Rightarrow$ avoid complexity by putting extensions in user-space:

- 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.

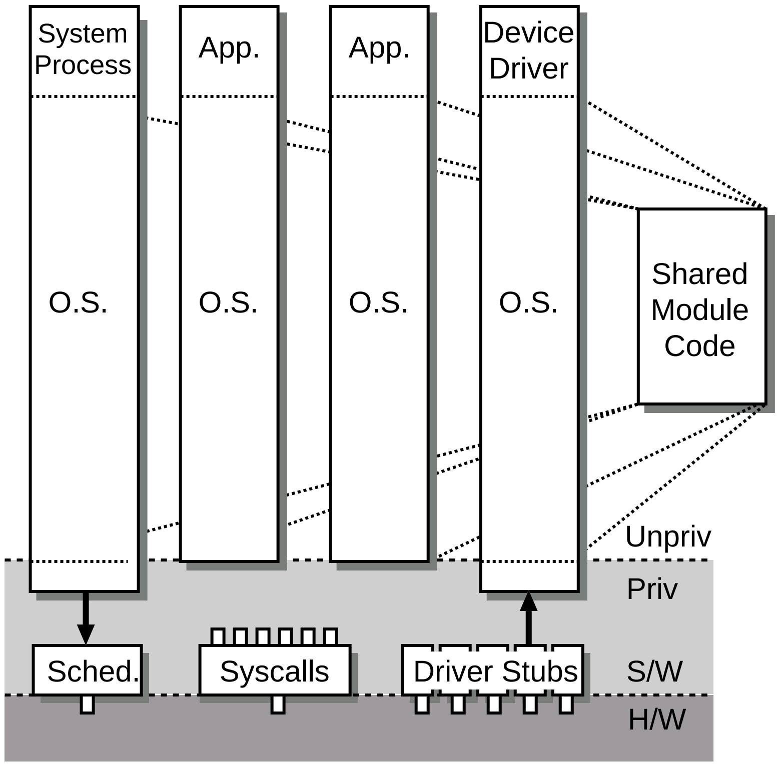
- e.g. Exokernel:
  - run most of OS in user-space library.
  - leverage DSL/packet filters for customization.
  - can get into a mess (e.g. UDFs).
- e.g. Nemesis:
  - guarantee each application share of *physical* resources in both space and time.
  - use IDL to allow user-space extensibility.
  - still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

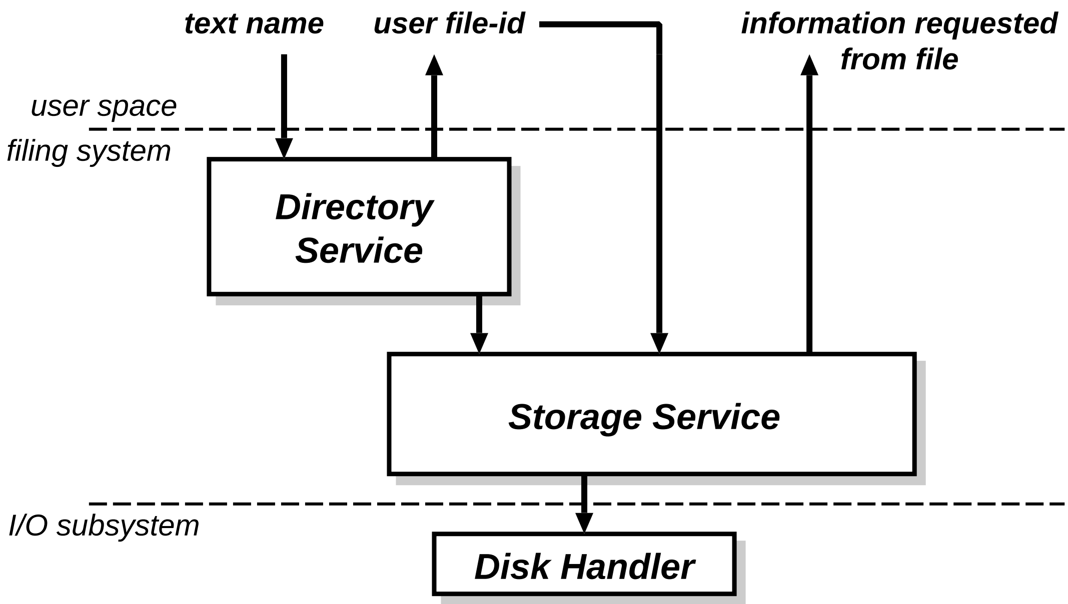


- Design to support soft real-time applications
  - isolation: explicit guarantees to applications
  - exposure: multiplex *real* resources
  - responsibility: applications must do data path
- Parallel development to exokernel:
  - similar overall structure (though leaner – no device drivers, DPFs, UDFs, 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

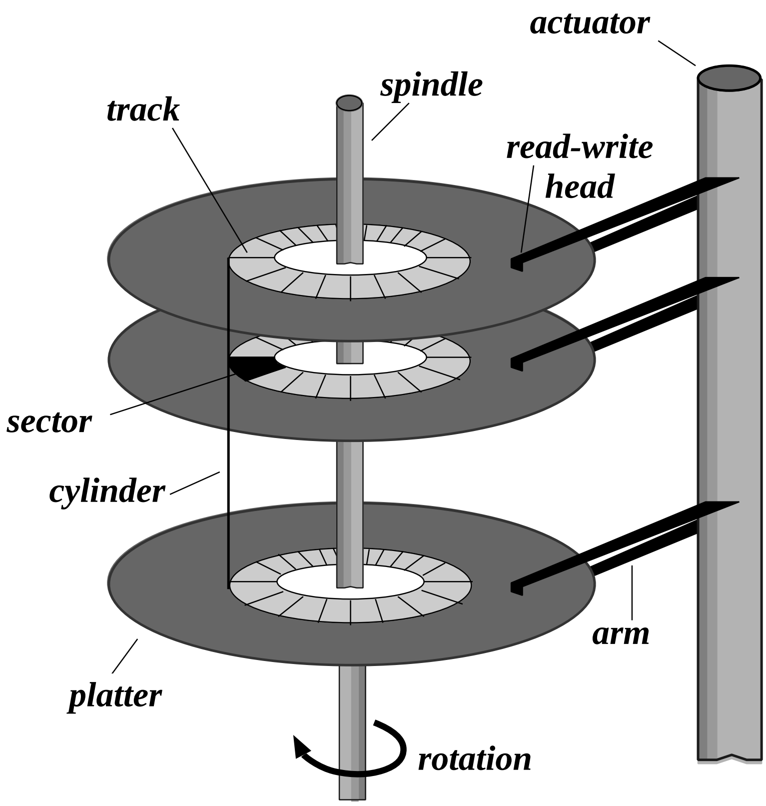
Filesystem and Database Storage



- So far (OS IA) saw high-level view of how file-systems work.
- In this section we will:
  1. Examine disk structure & scheduling
  2. See how some example file systems work
  3. Investigate on-disk storage of database records
  4. Look at ways of efficiently retrieving records
  5. Briefly case study the Postgres DBMS
  6. Learn a little bit about SANs, NAS, and distributed file-systems

Disk I/O

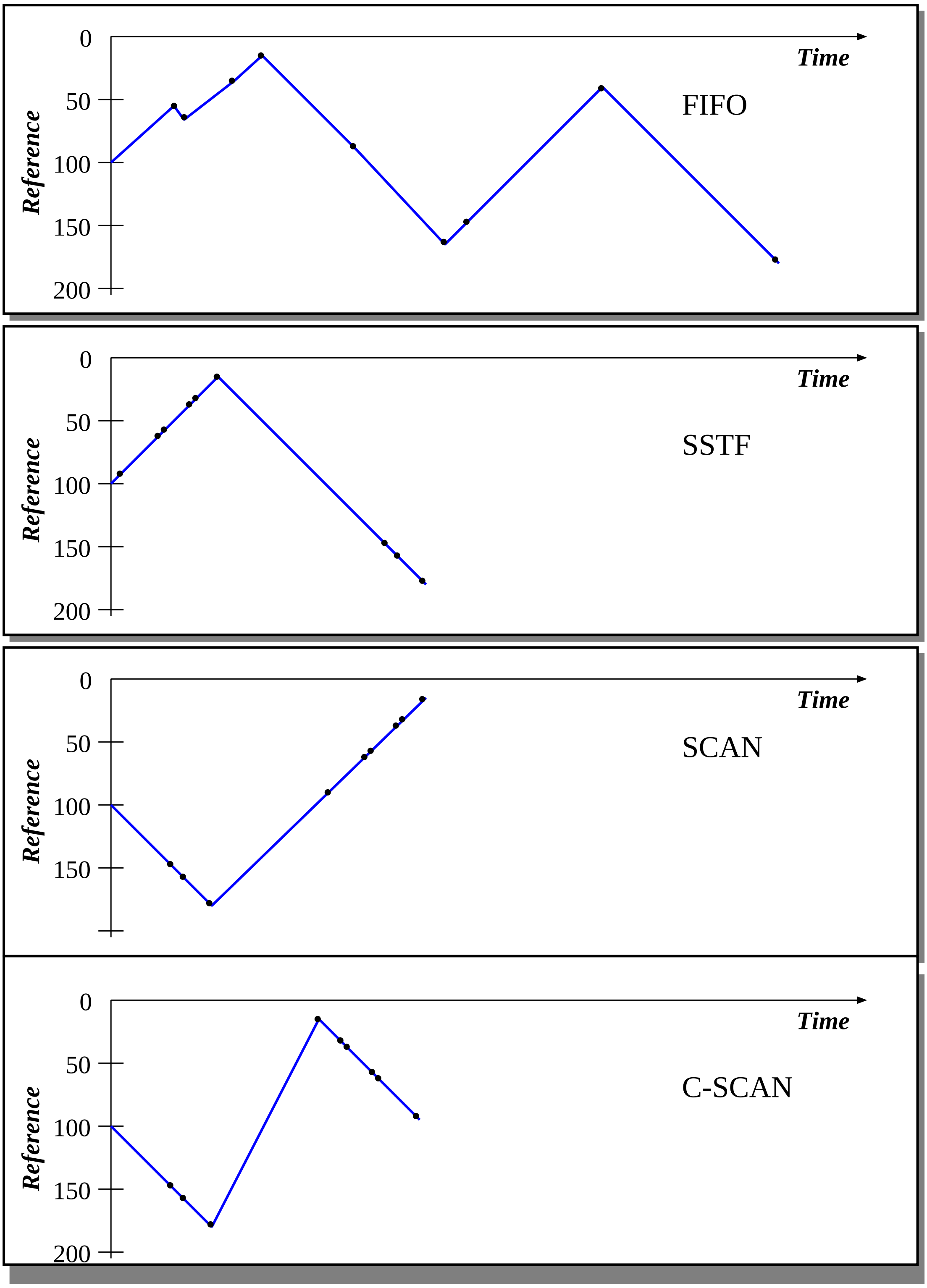
- Performance of disk I/O is crucial to virtual memory, file system and database operation
- Key parameters:
  1. wait for controller and disk.
  2. seek to appropriate disk cylinder
  3. wait for desired block to come under the head
  4. transfer data to/from disk
- Performance depends on *how the disk is organised*



Disk Scheduling

- In a typical multiprogramming environment have multiple users queueing for access to disk
- Also have VM system requests to load/swap/page processes/pages
- We want to provide best performance to all users — specifically reducing seek time component
- Several policies for scheduling a set of disk requests onto the device, e.g.
  1. FIFO: perform requests in their arrival order
  2. SSTF: if the disk controller knows where the head is (hope so!) then it can schedule the request with the shortest seek from the current position
  3. SCAN ("elevator algorithm"): relieves problem that an unlucky request could receive bad performance due to queue position
  4. C-SCAN: scan in one direction only
  5. N-step-SCAN and FSCAN: ensure that the disk head always moves

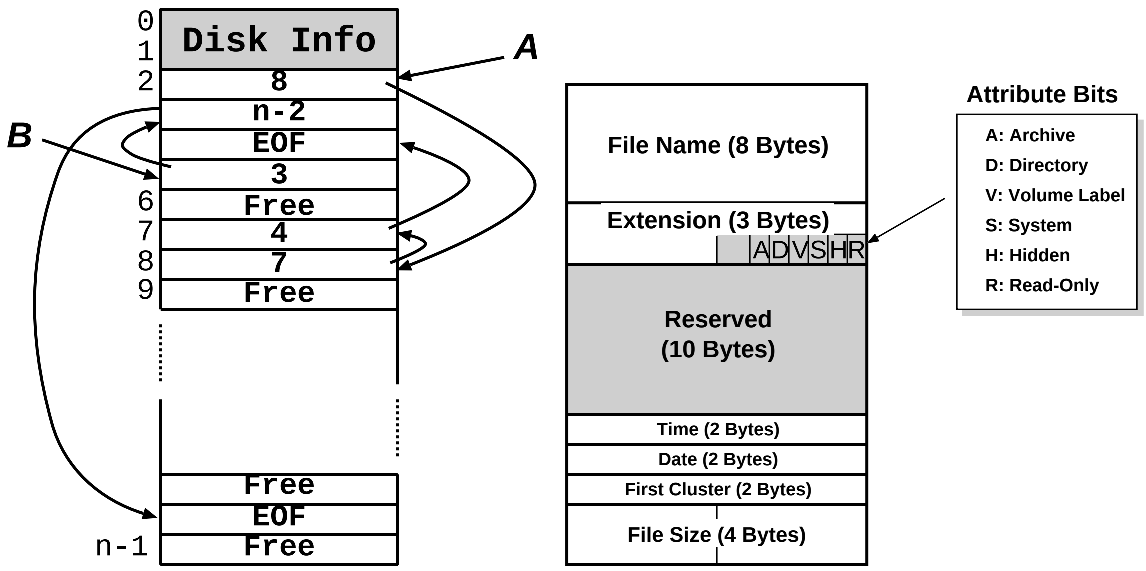
Reference String = `55,58,39,18,90,160,150,38,184`



Other Disk Scheduling Issues

- Priority: usually beyond disk controller's control.
  - system decides to prioritise, for example by ensuring that swaps get done before I/O.
  - alternatively interactive processes might get greater priority over batch processes.
  - or perhaps short requests given preference over larger ones (avoid "convoy effect")
- SRT disk scheduling (e.g. Cello, USD):
  - per client/process scheduling parameters.
  - two stage: admission, then queue.
  - problem: overall performance?
- 2-D Scheduling (e.g. SPTF).
  - try to reduce rotational latency.
  - typically require h/w support.
- Bad blocks remapping:
  - typically transparent ⇒ can potentially undo scheduling benefits.
  - some SCSI disks let OS into bad-block story

Case Study 1: FAT16/32



- A file is a linked list of *clusters*: a cluster is a set of 2^n contiguous disk blocks, $n \geq 0$.
- Each entry in the `FAT` contains either:
  - the index of another entry within the `FAT`, or
  - a special value `EOF` meaning "end of file", or
  - a special value `Free` meaning "free".
- Directory entries contain index into the `FAT`
- `FAT16` could only handle partitions up to $(2^{16} \times c)$ bytes ⇒ max 2Gb partition with 32K clusters.
- (and big cluster size is *bad*)

Extending FAT16 to FAT32

- Obvious extetension: instead of using 2 bytes per entry, `FAT32` uses 4 bytes per entry

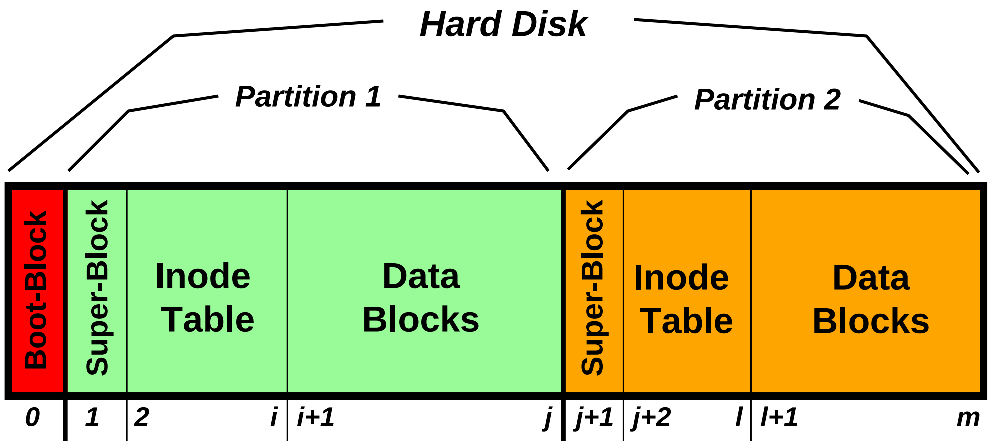
$\Rightarrow$ can support e.g. 8Gb partition with 4K clusters

- Further enhancements with `FAT32` include:
  - can locate the root directory anywhere on the partition (in `FAT16`, the root directory had to immediately follow the `FAT`(s)).
  - can use the backup copy of the `FAT` instead of the default (more fault tolerant)
  - improved support for demand paged executables (consider the 4K default cluster size . . .).
- `VFAT` on top of `FAT32` does long name support: unicode strings of up to 256 characters.
  - want to keep same directory entry structure for compatibility with e.g. DOS

  $\Rightarrow$ use *multiple* directory entries to contain successive parts of name.
  - abuse `V` attribute to avoid listing these

Case Study 2: BSD FFS

The original Unix file system: simple, elegant, slow.



The *fast file-system* (FFS) was develped in the hope of overcoming the following shortcomings:

1. Poor data/metadata layout:
   - widely separating data and metadata ⇒ almost guaranteed long seeks
   - head crash near start of partition disastrous.
   - consecutive file blocks not close together.
2. Data blocks too small:
   - 512 byte allocation size good to reduce internal fragementation (median file size ~ 2K)
   - but poor performance for somewhat larger files.

FFS: Improving Performance

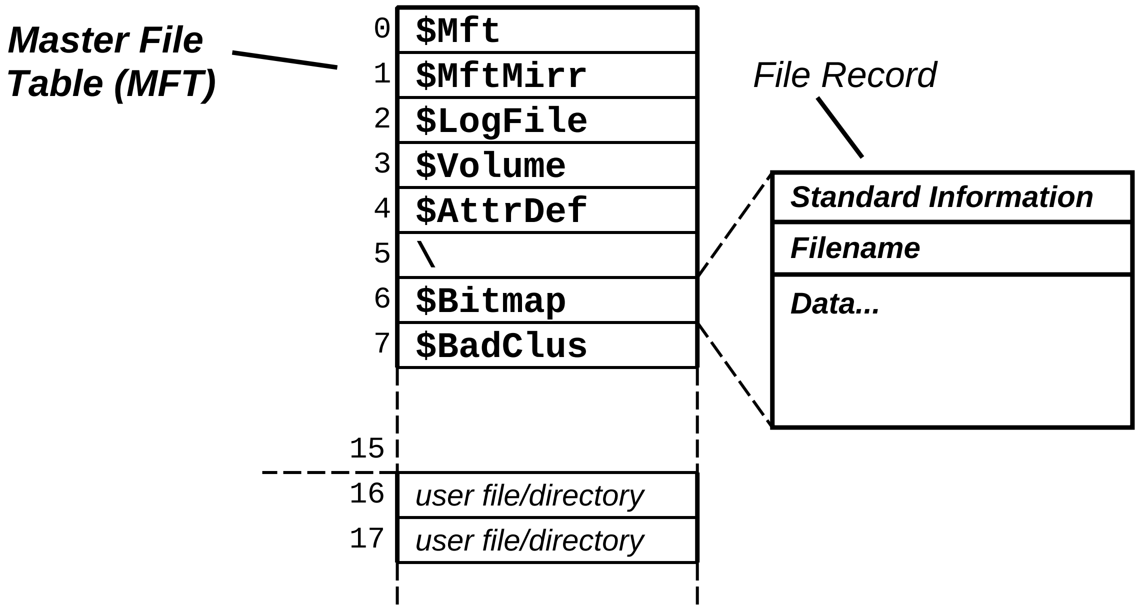
The FFS set out to address these issues:

- Block size problem:
  - use larger block size (e.g. 4096 or 8192 bytes)
  - but: last block in a file *may* be split into fragments of e.g. 512 bytes.
- Random allocation problem:
  - ditch free list in favour of bitmap $\Rightarrow$ since easier to find contiguous free blocks (e.g. `01110000011101`)
  - divide disk into *cylinder groups* containing:
    * a superblock (replica),
    * a set of inodes,
    * a bitmap of free blocks, and
    * usage summary information.
  - (cylinder group $\simeq$ little Unix file system)
- Cylinder groups used to:
  - keep inodes near their data blocks
  - keep inodes of a directory together
  - increase fault tolerance

FFS: Locality and Allocation

- Locality key to achieving high performance
- To achieve locality:
  1. don't let disk fill up ⇒ can find space nearby
  2. spread unrelated things far apart.
- e.g. the BSD allocator tries to keep files in a directory in the same cylinder group, but spread directories out among the various cylinder groups
- similarly allocates runs of blocks within a cylinder group, but switches to a different one after 48K
- So does all this make any difference?
  – yes! about 10x–20x original FS performance
  – get up to 40% of disk bandwidth on large files
  – and *much* beter small file performance.
- Problems?
  – block-based scheme limits throughput ⇒ need decent clustering, or skip-sector allocation
  – crash recovery not particularly fast
  – rather tied to disk geometry. . .

Case Study 3: NTFS



- Fundamental structure of NTFS is a *volume*:
  - based on a logical disk partition
  - may occupy a portion of a disk, and entire disk, or span across several disks.
- An array of file records is stored in a special file called the Master File Table (MFT).
- The MFT is indexed by a *file reference* (a 64-bit unique identifier for a file)
- A file itself is a structured object consisting of set of attribute/value pairs of variable length. . .

NTFS: Recovery

- To aid recovery, all file system data structure updates are performed inside *transactions*:
  - before a data structure is altered, the transaction writes a log record that contains redo and undo information.
  - after the data structure has been changed, a commit record is written to the log to signify that the transaction succeeded.
  - after a crash, the file system can be restored to a consistent state by processing the log records.
- Does not guarantee that all the user file data can be recovered after a crash — just that metadata files will reflect some prior consistent state.
- The log is stored in the third metadata file at the beginning of the volume (`$Logfile`) :
- Logging functionality not part of NTFS itself:
  - NT has a generic *log file service*

  $\Rightarrow$ could in principle be used by e.g. database
- Overall makes for far quicker recovery after crash

NTFS: Other Features

- Security:
  - each file object has a *security descriptor attribute* stored in its MFT record.
  - this atrribute contains the access token of the owner of the file plus an access control list
- Fault Tolerance:
  - `FtDisk` driver allows multiple partitions be combined into a logical volume (RAID 0, 1, 5)
  - `FtDisk` can also handle *sector sparing* where the underlying SCSI disk supports it
  - NTFS supports software *cluster remapping*.
- Compression:
  - NTFS can divide a file's data into *compression units* (blocks of 16 contiguous clusters)
  - NTFS also has support for *sparse files*
    * clusters with all zeros not allocated or stored
    * instead, gaps are left in the sequences of VCNs kept in the file record
    * when reading a file, gaps cause NTFS to zero-fill that portion of the caller's buffer.

Case Study 4: LFS (Sprite)

LFS is a *log-structured file system* — a radically different file system design:

- Premise 1: CPUs getting faster faster than disks.
- Premise 2: memory cheap ⇒ large disk caches
- Premise 3: large cache ⇒ most disk reads "free".

⇒ performance bottleneck is writing & seeking.

Basic idea: solve write/seek problems by using a *log*:

- log is [logically] an append-only piece of storage comprising a set of *records*.
- all data & meta-data updates written to log.
- periodically flush entire log to disk in a single contiguous transfer:
  - high bandwidth transfer.
  - can make blocks of a file contiguous on disk.
- have two logs ⇒ one in use, one being written.

What are the problems here?

LFS: Implementation Issues

1. How do we find data in the log?
   - can keep basic UNIX structure (directories, inodes, indirect blocks, etc)
   - then just need to find inodes ⇒ use *inode map*
   - find inode maps by looking at a checkpoint
   - checkpoints live in fixed region on disk.
2. What do we do when the disk is full?
   - need asynchronous *scavenger* to run over old logs and free up some space.
   - two basic alternatives:
     1. compact live information to free up space.
     2. thread log through free space.
   - neither great ⇒ use *segmented log*:
     - divide disk into large fixed-size segments.
     - compact within a segment, thread between segments.
     - when writing use only clean segments
     - occasionally clean segments
     - choosing segments to clean is hard. . .

Subject of ongoing debate in the OS community. . .

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

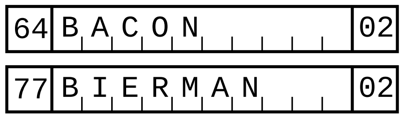
Representing Records

- 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

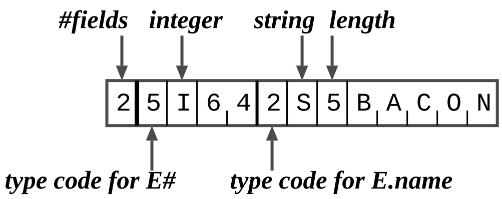
Employee Record (Schema)

```
E#, 2 byte integer
E.name, 10 char
Dept, 2 byte code
```

Actual Employee Records



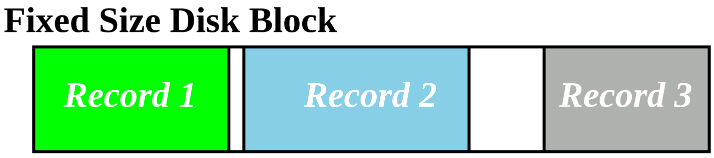
- Variable format ⇒ record "self describing".
  - e.g. variable format and length



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

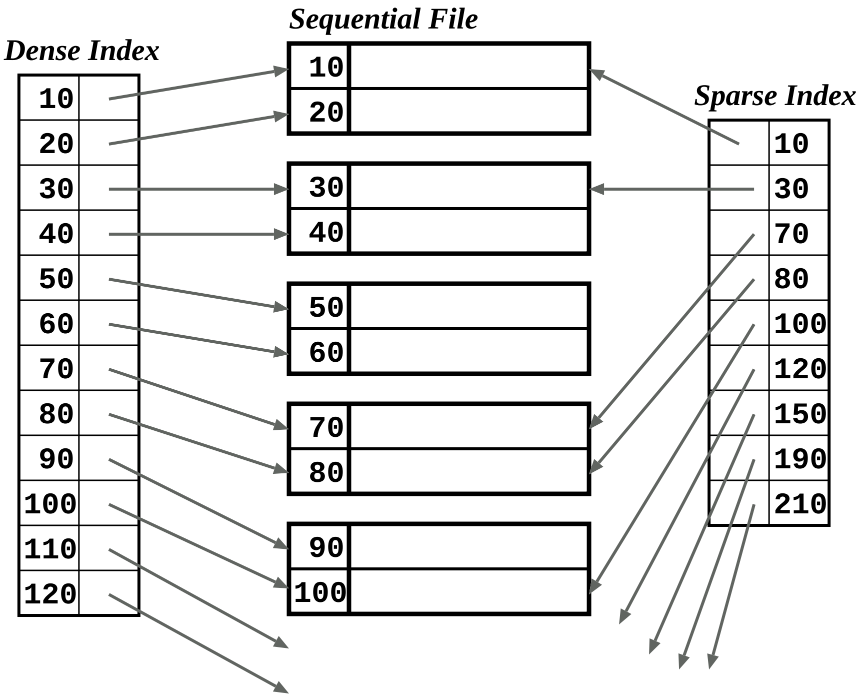
Storing Records in Blocks

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



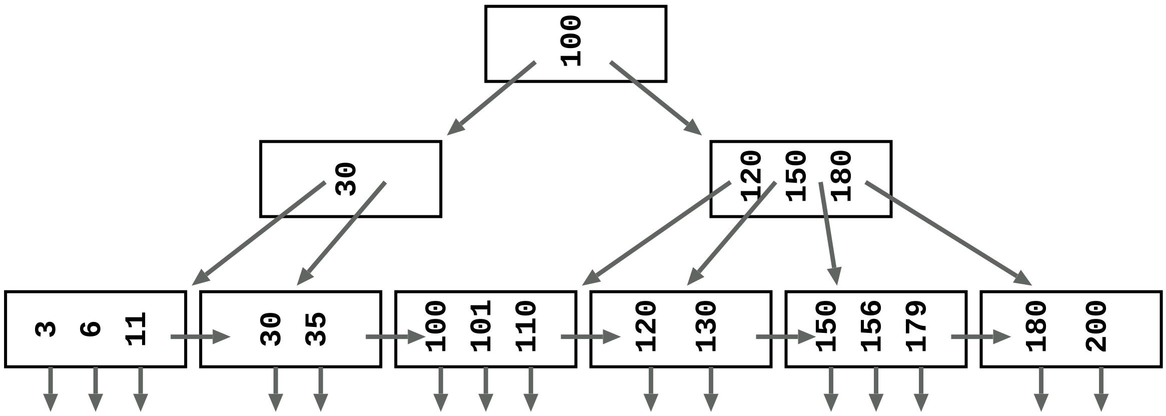
  - fixed size recs: no need to separate
  - variable length ⇒ separation marker?
  - better: offsets in block header
- What about *spanning* multiple blocks?
  - may need if variable length record grows
  - certainly need if |record| > |block|
  - can impl with pointers at end of blocks
- Should we mix 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

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.

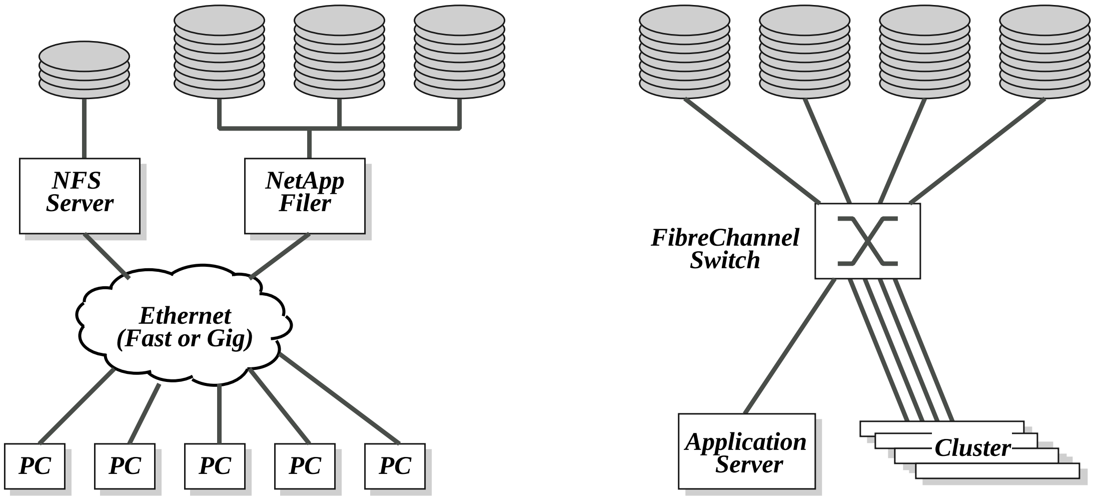
Postgres Implementation

- 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

Distributed Storage

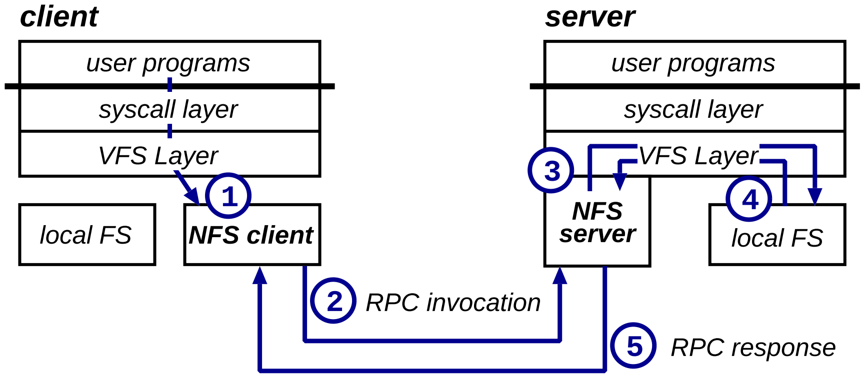
- Filesystems/DBMS want big, fast, reliable disks.
- Cheaply achieve this using multiple disks, e.g.
  - RAID = Redundant Array of Inexpensive Disks
  - better performance through *striping*
  - more reliable via *redundancy*:
    - simple mirroring
    - generalised parity (Reed-Solomon)
    - variable 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 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
- NFSv3 provides some support for this.

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

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 SSDFS easier.

Summary & Outlook

We've seen a selection of systems topics:

- Local and distributed virtual memory
- Systems with funky hardware (Multics, the CAP)
- Microkernels (Mach, L3/L4, EROS)
- Virtual machine monitors (VM/CVS, Disco, VMWare, Denali, XenoServers)
- Extensible operating systems (SPIN, Vino, Exokernel, Nemesis)
- Disk scheduling and management
- Local Filesystems (FAT, FFS, NTFS, LFS)
- Database storage & retrieval (issues, consistency, records, blocks, indices, Postgres)
- Distributed storage and filesystems (NAS, SANs, NFS, xFS)

Lots more research ongoing in most of above areas.

Next section of course: scalable synchronization.