**Operating System**: Layer of HW that provides application software access to hardware. Referree (resource allocation, isolation, commpunication) Illusionist (abstractions) Glue (storage, windows, networking, sharing, etc)

**Virtual machine**: software emulation of abstract machine (process/system).

**1 Thread**: Single unique execution context, has: PC, stack pointer, registers.

**2 Address space**: set of accessible addresses + state associated (code, static data, heap stack) Multiple processor illusion, All virtual CPU share non-CPU resources (I/O, memory)-can access other thread data, share instructions

**3 Process**: execution env with restricted rights, has address space with 1+ threads. Processes protected from each other, OS protected from them

**4 Dual Mode Operation:** user->kernel sets system mode, saves user pc.

**Process Control Block (PCB)** – represents process status, register state, PID, user, executable, priority, execution timer, memory space, translation… Scheduling algorithm selects next one to run.

**Simultaneous Multithreading/hyperthreading:** hardware technique, uses superscalar processors and duplicates register state to make more “threads” allowing instructions to run, but sublinear speedup and can only do small numbers of threads at a time

**Kernel stack**? 2 stack model: interrupt stack + user stack, Syscall handler: vector through well-defined entry points: locate args, copy args, validate args, copy results back

**Interrupt:** external signal. Handler invoked with interrupts disabled. OS kernel enables/ disables interrupts, some nonmaskable interrupts (kernel segfault)

**Take interrupts safely**: interrupt vector, kernel interrupt stack, interrupt masking, atomic transfer of control, transparent restartable execution

**Fork** returns copy or process, rv 0 = child, <0 = error, >0 = parent pid

**UNIX** Exec changes program being run, wait waits for process to finish, signal sends notification to other process. Shell: job control system allows user to manage programs

**Key I/O design**: uniformity, open before use, byte-oriented, kernel buffered reads/writes, explicit close

**File:** named collection of data, has data+metadata. Directory: folder containing files + directories. High level: file streams FILE\*, Low level: file descriptors: internal data structure describing file info.

**Device Driver:** device specific code in kernel that interacts directly with the device hardware. Standard internal interface. Top half: accessed from syscalls, bottom half: run as interrupt routine.

**User program -> I/O subsystem ->device driver top -> bottom -> device hardware**

**Socket:** abstraction of network IO queue, mechanism for communication, data transfer like files. (4.2 BSD)

**For protection:** separate fork for child. Parallelism: don’t wait for child.

Switch between processes: context switch. Lifecycle: new ready running waiting terminated

**Thread**: sequential execution stream within process (no protection), **Multithreading:** single program made of concurrent activities

**Thread state:** 1. shared by all threads (memory, I/O). 2. Private to each thread, TCB: CPU registers/PC, execution stack (parameters, temp variables, return PCs).

thread\_fork(func, args) create new thread to run func, thread\_yield relinquish processor, thread\_join waits for joined to exit, then return, thread\_exit quits and cleans up, wakes joiner if necessary.

**Switch:** save everything, can’t test exhaustively. Design for simplicity. Faster than process switching. Can happen by voluntary yield, IO, wait. Switch time increases with size of working set. What if thread never I/O, wait, yield? external events (interrupt/timer)

**Multithreaded:** PCB points to multiple TCB. Switching threads across blocks requires changes to memory, IO address tables

**Kernel threads:** a bit expensive -> user threads (user program provide scheduler+thread). But when one blocks on I/O, all block.

Option: scheduler activations, kernel inform user when thread blocks.
**Multiprocessing** = multiple CPU. **Multiprogramming** = multiple job/process. **Multithreading** = multiple thread per process

run threads “concurrently”: scheduler runs in any order + interleaving.

**Correctness and concurrent threads:** Independent = no state shared, deterministic, reproducible. Cooperating = shared state, nondeterministic, nonreproducible (heisenbug)

Why cooperating threads: share resources, speedup, modularity

**Thread pools:** if threads unbounded, throughput can sink with lots of use- instead allocate “pool” of worker threads, represenging max level of multiprogramming

**Thread tick:** thread updates counters, if quanta exhausted yield. Thread yield: Set cuttent to ready, put on ready list, call schedule. Schedule: select next thread, run

**Atomic operation:** operation that always runs to completion or not at all (mostly memory references and assignments or words, often not double precision FP store)

**Concurrent threads** allow transparent overlapping of computation and I/O, allow parallel processing

Threaded programs with shared data must work for all interleavings of thread instruction sequences (Therac 25 -> patient deaths, space shuttle)

**Synchronization:** using atomic operations to ensure cooperation

**Mutual Exclusion:** ensure only one thread does a particular thing at a time

**Critical section:** piece that only one thread can execute at once. Only one thread at a time will get this section.

**Lock:** Prevents someone from doing something (lock before entering critical section, before accessing shared data, unlock after), wait if locked

Need to disable+reenable interrupts during acquire and release. Since interrupts disabled when calling sleep, responsibility is on next thread to reenable. Problems: users cant use, doesn’t work on multiprocessor

**test&set:** if lock free, test&set reads 0 and sets value=1, so lock is busy, returns 0 so loop exits. If lock busy, test&set reads 1 and sets value=1. returns 1 so while continues. Problems: busywait, doesn’t take advantage of multiprocessor/multicore caches

test&test&set: in the big loop, add while(mylock) before while(test&set (&mylock)), waits until lock might be free. Still busywaits.

Can minimize busy waiting in test&set locks by going to sleep occationally

**Busywaiting:** thread consumes cycles while waiting. Problms: inefficient, waiting thread takes cycles from thread holding lock. For semaphores and monitors, waiting thread may wait for arbitrary amount of time

**Priority inversion:** If busy waiting thread has higher priority than thread holding lock, no progress (martian rover)

**Semaphores:** a kind of generalized loock (Dijkstra in late 60s, main synch primitive in original UNIX, has non-negative integer value, atomic operations:

P() (proberen-to test): wait for semaphore to be >0, then decrement by 1

V() (verhogen-to increment): increment by 1, waking up waiting P if necesary

Mutual exclusion(mutex): initial value = 1. Scheduling constraints: init = 0

**Producers and consumers with bounded buffer** (coke machine, GCC compiler): semaphore for each constraint (fullBuffer, emptyBuffer, mutex) Order of P() of mutex and butffers is important, could cause deadlock (need mutex inside buffers). Vs order isn’t important.

**Monitor:** a lock and 0 or more condition variable for managing concurrent access to shared data (Java provide natively, most other use locks and conds)

**Condition variable:** a queue of threads waiting for something inside a critical section- make it possible to go to sleep inside by atomically releasing lock when going to sleep. operations: wait(&lock), signal(), broadcast()

**Hoare:** signaler gices lock, CPU to waiter, waiter runs immediately. waiter gives lock, processor back to signaler when exiting critical or when wait again

**Mesa** (most real OS): signaler keeps lock and processor, waiter placed on ready queue with no priority. Need to check condition again after wait

**Scheduling assumptions:** 1 program per user, 1 thread per program, independent programs. Goals/criteria: minimize response time, maximize throughput (jobs/second), fairness.

**FCFS:** FIFO, simple, but short jobs can be stuck behind long ones.

**RR:** n process in ready queue and time quanta is q, each process get small unit of cpu time 1/n, q time units each. large q->FCFS, small q->interleaved

q must be large w/ respect to context switch, otherwise high overhead

RR is better for short jobs, fair. But context switching adds up for longer jobs

Bad when all jobs are same length: finish same as FCFS, worse response time

**Basic readers/writers:** readers wait until no writers, access data, checkout+wake up waiting writer. writers wait until no readers or writers, access database, checkout+backe up waiting readers or writers.

State vars: AR, WR, AW, WW, oktoread, oktowrite

**Monitors from semaphores:** Problems with condition vars: P V commutative, condition variables not commutative. not legal to look at contents of semaphore queue, race contion if signal between lock release and sema p.

**Basic structure of monitor:** **lock-> check/update state vars+wait-> unlock-> do something-> lock-> check/update state vars+signal-> unlock**

Shortest Job First/Shortest Time to Completion First(**SJF/STCF):** Run whatever job has least amount of computation first

Shortest Remaining Time First(**SRTF/SRTCF**): preemptive SJF: if job arrives w/ shorter time to completion than current, preempt CPU.

Idea: get short jobs out of system, best possible at minimizing average response time. If all jobs same length, becomes same as FCFS.

**Starvation:** SRTF can lead to starvation if many small jobs (unfair).

Need to predict future: some systems ask user but hard to predict. **Adaptive:** change policy based on past behavior (exponential averaging estimator function)

**Multi level feedback scheduler:** method for exploiting past behavior

Multiple queues, each with different priority, each with its own scheduling algorithm. Adjust: Job starts in highest priority, if timeout expires drop 1 level, if timeout doesn’t expire push up 1 level. Result approximates SRTF- CPU bound drop, I/O bound stay at top. Can fool by putting unnecessary I/O calls to keep burst time short.

Fairness tradeoff with average resonse time. How? could give each queue some fraction of CPU or increase priority of jobs that don’t get service.

**Lottery scheduling:** give each job some number of tickets, short jobs get more and long get fewer- behaves gracefully as load changes

**O(1) linux schedulier:** priority based, 140 priorities, 40 for user tasks. Two prioritiy queues “active” and “expired”. Timeslice depends on priorities,like multilevel with different timeslice at each level. User task priority adjusted based on heuristics, Real time always preempt non-RT

**Completely Fair Scheduler-** inspired by “Fair Queuing”, tried to give each process and equal fraction. Priorities reflected by weights such that increasing priority by 1 always gives same fractional increase in CPU time. Tracks “virtual time” received by each process, uses red-black tree to sort

**Real Time Scheduling-** Efficiency important but predictability essential. Hard real time: attempt to meet all deadlines(Earliest Deadline, Least Laxity, etc). **Soft real time:** attempt to meet deadlines with high probability, minimize miss ratio/maximize completion ratio-Constant Bandwidth Server

**Resources:** passive entities needed by threads to do their work

2 types- preemptable, can take it away (CPU, security chip) & non-preemptable, must leave with thread (disk space, virtual address space)

May require exclusive access or be shareable

**Starvation:** thread waits indefinitely. **Deadlock:** circular waiting for resources. Deadlock 🡪 starvation but not vice versa- can’t end w/o external interaction

**Conditions for deadlock:**

1. Mutual exclusion- only 1 thread a a time can use a resource
2. Hold & wait- thread holding at least 1 resources is waiting for additional resources held by other threads
3. No preemption- resources released voluntarily after thread is finished
4. Circular wait- exists a set of threads where 1 waiting for 2,etc.

**Resource allocation graph** request edge T🡪R, assignment edge R🡪T

Methods for handling deadlock: allow to enter then recover, ensure never enters deadlock, ignore the problem (most OS’s).

**Deadlock detection-** see if tasks can eventually terminate on ther own

What to do if detected? Terminate thread (not always possible eg mutex), preempt resources without killing (doesn’t always fit w/ semantics of computation), or roll back actions on deadlocked threads (common in databases, if don’t change anything will get deadlock again)

**Deadlock Prevention**: Infinite resources (not realistic), no sharing (not realistic), don’t allow waiting(phone companies, but inefficient), make threads request needs at beginning (prediction is hard), force threads to request in particular order(dimension), preventing cycles

**Banker’s Algorithm:** State maximum resources needs in advance, allow thread of proceed if (available - #requested) >= max remaining that might be needed by any thread. Allocates resources dynamically by pretending each request is granted, then run deadlock detection, grant if request is deadlock free. Keeps system in SAFE state, i.e. there is a sequence where everyone can finish. Allows sum of max resource needs to be > total resources.

**Memory Multiplexing**: Contolled overlap (separate states dont collide), ee (virtual-physical), Protection (no access other process’s memory)

**Address space:** all the addresses and state a process can touch.

**Base & Bound**- could use for dynamic address translation, illution of memory starting at 0. Fragmentation problem, missing support for space address space, hard to do inter-process sharing

**Segmentation-** multiple segments, each given region of contiguous memory. Segment number mapped to base/limit pair, with a valid bit, VA space has holes, need protection mode in segment table, context switch: segment table stored in CPU, might store all process memory to disk (Swap) Problems: fit variable sized chunks to physical memory, move processes multiple times to fit, limited options for swapping to disk, external fragmentation

**Fragmentation:** wasted space. External- free gaps between allocated chunks, internal- don’t need all memory in allocated chunks

More programs than fit in memory: swap? increases context-switching

**Paging:** allocates physical memory in fixed size chunks (pages), typically small -> multiple pages per segment. Page table resides in physical memory, contains physical page and permission (valid, read, write) for each virtual page. Virtual address mapping: offset from VA coped to PA, page size = 2^offset. VP# is all remaining bits, PP# copied from table to PA

**\*Check page table bounds and permissions!**

Context switch: need to switch page table pointer and limit.

pro: simple memory allocation, easy to share. con: lots of entries if sparse address space, if table is big not all pages are used all the time

Combine: Paging+segmentation=2 level paging. Tree of Page Tables

**Two level page table:** fixes sparse address space. Valid bits on PTEs, don’t need every 2nd level table and even when they exist, 2nd level tables can reside on disk if not in use.

Pro: only need to allocate as many PTE as need, easy memory allocation, easy sharing (segment or page level). Con: onepointer per page, page tables need to be contiguous,2 or more lookups per reference (expensive)

**Inveted Page Table:** use a hash table, size independent of VA space & directly related to amount of physical memory. attractive for 64bit address spaces. Con: complexity of managing hash changes (often in hardware)

**Page Table Entry:** PPN, permissions (Valid, writeable, user accessible, Dirty, accessed, ketnel only, etc)

Can a process modify own translation table? NO- otherwise could access all of physical memory.

**User🡪Kernel:** Syscall: I/O files, process, network. Not entirely consistent across OS. What happens: set system to kernel mode, handler address fetched from table, handler started. Argument passing: in registers (not much), copies into user mem/kernel mem- every argument must be checked

**Exceptions**: Synchronous Exceptions (Trap)- divide by 0, illegal instruction, bus error, segfault, page fault. Asynchronous exceptions: Interrupts (timer, disk ready, network, etc). Can be disabled, traps can’t.

On syscall/exception/interrupt- hardware enters kernel mode with interrupts disabled, save PC, jump to appropriate handler who saves registers/other CPU state, then switches to kernel stack

**Protection/dual-mode without hardware**? Doesn’t require but normally uses.

Protection via strong typing (LISP, Java), Protection via software fault isolation (compiler checks for “dangerous” operations. OR: use VM to guarantee safe behavior

Critical sections can still get context switched

Create thread: TCB, init registers, place on ready queue. Process: PCB, address space/page table, new thread (fork)

Processes can communicate by: shared memory, message passing, file system.

Interrupt controller provide control over hardware interrupt signal enable and priority

“Dump Core” dumps the core memory

Too many threads can be bad: lots of switching overhead,space from TLB, deadlock

BLOCK to RUNNING: illegal. RUNNING to BLOCK: legal. RUNNABLE to BLOCK: illegal.

Core memory: technology where each bit of memory was stored as a magnetic field in a round iron ring. looked like a woven tapestry made from these rings. each core had horizontal+vertical wires running through for read and write

Address translation protect processes: kernel make sure they don’t access same page

bits per page level = # bits to address pages[address bits-offset bits]/#levels

monitors can be constructed using semaphores.

MicroKernel isolates the components of the system (file system, network stack) into their own address spaces. causes resilience against bugs.

It’s possible for a single monitor to deadlock. (create 2 semaphores)

Largest page table: (1+#page table chunks) (PTE size) [in bytes]

Max memory: 2^address size

Copy on Write: make cheap copies of address space by creating duplicate page table that points to same physical pages as existing. Accomplished by setting PTE to read-only so the target page can be copied at first write.

wait { sema.p() } is bad because it will deadlock.

Therac 25 was a medical device for radiation therapy. Had race conditions.

Disabling interrupts in 1 CPU doesn’t affect other CPUs.

Registers are private to each thread and won’t be overwritten by others.

Segmentation without paging can fragment physical virtual address space

using CAS(&addr, expr1, expr2) : int return; do { expr1 = M[addr];} while (!CAS(addr, expr1, expr2); where we want to set M[addr] to expr2 and it was expr1 before.

If expected wait time of lock is very short, spinlock could be better than sleeping. expected wait time must be less than time to sleep+wake up.

Xerox Parc was first to develop mice & windows

direct map can have higher hit rate than fully associative

Conds of deadlock necessary, not sufficient- banker algorithm prevents not removes

Cache: a repository for copies that can be accessed more quickly than original- underlies many of the techniques used to make computer fast

Average access time = (hit rate x hit time) + (miss rate x miss time)

Cannot afford to translate on every access -> “TLB”: Translation Lookaside Buffer

Temporal Locality- keep recently accessed close, Spatial Locality- contiguous blocks to upper levels.

Cache Misses: Compulsory (first access), Capacity (cache full), Conflict (collision), Coherence (other process updates memory). idx identify set, Tag identify copy, Block is min quantim of caching.

Direct Mapped, Set Associative (N entry per cache intex), Fully Associative (any block can hold anything) What to replace: Random or LRU.

Write through: information written to both cache and mem

Write back: written only to cache, written to memory when block replaced

TLB Miss: Hardware PT: HW in MMU look at current PT to fill TLB. Could Page Fault. Software PT: TLB fault, fills TLB and return from fault. Most chipsets = hardware

Precise exception: state is preserved as if program executed up to offending instruction

Context Switch: TLB entries no longer valid. Could invaliate TLB (simple but expensive) or include ProcessID in TLB (needs hardware). If translation tables change, must invalidate TLB entry.

Size? Needs fast-> low associativity. But, need few conflicts. TLB Usually small, 128-512 entries-> support higher associativity. Usually organized as fully-associative, lookup = VA, return PA+other info. If Fully associative too slow, put small (4-16 entry) direct mapped cache in front (TLB slice)

Can reduce translation time further by overlap TLB lookup with cache access since they are serial. Offset in VA exactly covers the “cache index” and “byte select”. If they don’t overlap completely: need to co something else. Another option: virtual caches- tags in cache are virtual addresses, translation only happens on cache misses

Use of caching techniques: paged virtual memoty, TLB, file systems, DNS, web proxies

Impact: efftcts, process scheduling, thread scheduling on cach performance

Demand paging: programs require lots of memory, but not all the memory all the time-> use main memory as cache for disk.

disk larger then physical memory- transparent level of indirection (page table)- suppot flexible placement of physical data, variable location transparent to user program

block size? 1 page. organization? fully associative. finding page? TLB->page-table

miss? lower level (disk). write? write back=need dirty bit!

PTE: vaild=in memory, PTE points. not vaild = not in memory, use PTE to find on disk.

Reference page with invalid PTE? page fault!

Load exe into memory: .exe lives on disk, contain contents of code/data, relocation entries/symbols. OS loads to memory, initialize registers. program set up stack/heap

Utilized pages in virtual address space backed by disk page block called backing store

User page table maps entire VAS: all utilized regions backed on disk, swapped in&out of memory as needed, resident pages mapped to frame in memory they occupy, portion that HW needs must be resident in memory, for all other pages OS must record where to find them on disk.

Nonresident pages: FindBlock(PID, page#)->disk block. stored in memory (hash?)

Software loaded TLB use bit? options 1. hardware set use bit in TLB, when entry replaced software copies back to PT 2. software manage TLB entries as FIFO list, not in TLB is second chance LRU list

Core map: page tables map VP->PP, need a reverse mapping for clock algorithm

Effective Access Time = Hit Rate x Hit Time + Miss Rate x Miss Time

 EAT = Hit Time + Miss Rate x Miss penalty

Compulsory Miss: never loaded to memory before. Capacity: not enough memory. Conflict: don’t exist (fully associative). Policy: pages were kicked out b/c policy

Replacement policies: FIFO bad, throw out heavily used. MIN: throw out pages that won’t be used- ideal. RANDOM: unpredictable, LRU: approximation to MIN

Problem: many instruction for each hardware access. In practice, approximate LRU

Adding memory = miss rate down- only for LRU and MIN

Perfect LRU: timestamp on each reference, but too expensive

Clock Algorithm: arrange physical pages in circle with single clock hand- approximates LRU (replaces old page, not oldest. Hardware “use” bit per physical page (not require- can use valid to emulate) , sets bit on each reference (if not set = not referenced for a while). On page fault: advance clock hand, check use bit: 1 – used recently, clear & leave. 0- replace.

Nth chance: give page N changes. large N = better LRU approximation. small N = more efficient. extra overhead to replace dirty page-> common -> clean N=1, dirty N=2

Which are useful in PTE: use, modified, valid, readonly. Can emulate “modified” and “use” bits using readonly and valid bits and software.

Second Chance List (VAX/VMS): Split memoty in 2: active list(RW), SC list(invalid)

access pages in active list full speed, otherwise pagefault- move overflow page from end of active list to front of SC list and mark invalid. move desired page on SC list to front of active list, mark RW. How many pages: if 0, FIFO, if all, LRU but page fault always. pick intermediate value: few disk access, but incresed overhead trapping

Free List: keep set of free pages ready for use (filled in background by clock algorithm or other)- faster for page fault.

Allocation of page frames: global vs local replacement, equal allocation(per process) vs proportional allocation (size of process)vs priority allocation

Thrashing- process is busy swapping pages, if happens then suspend/swap out processes.

Working set- set of pages referenced recently. when swapping process back, use WS.

Clustering: on a pagefault, bring in multiple pages around faulting page

IO devices supported by IO controllers. but unreliable/unpredictable. operational parameters: byte/block, sequential/random, polling/interrupts.

Goal of IO subsystem: uniform interfaces, b/c device driver implements standard interface

block devices (drives, DVD)- access blocks of data: open(), read(), write()- raw IO/FS access

character devices (keyboard, mouse, serial, usb)- single character at time: get(), put()

network devices (ethernet, wireless)- socket: select(). pipe, fifo, stram, queue, mailbox

Timing: blocking interface: “wait”, nonblocking: “don’t wait”: return quick with count of successful transfer, Asynchronous: “tell me later”: take ptr, return, later kernel fills buffer and notifies user.

Processor talk to device: controller- set of registers to read/write, memory for request queues

Access registers in 2 way: IO instructions(in out)/memory-mapped IO(load store)

Transferring data: Programmed IO (transfer each byte via in/out/load/store. simple, but consumes cycles proportional to data size), Direct memory access (controller access memory bus)

interrupt: hendle unpredictable events, but high overhead. polling: low overhead, but waste cycles

virtual map: kernel mem not visible to user, every phys page described by “page” structure

one mechanism for allocating/requesting pages, also routines for freeing pages.

page frame reclaiming algorithm: low on memory reclaiming (flush dirty to disk), hibernation reclaiming(kernel suspend to disk), periodic reclaiming((two LRU lists)

slab allocator- objects segregated into “caches”, divided into “slabs”. avoid memory fragmentation

performance concepts: response time/latency: time for operation. bandwidth/ throughput: rate ops are performed, start up: time to init operation. Latency(n) = ovhd + n/bandwidth

half power pt: n=S\*B – when half the bandwidth is used

peak bw: device transfer bw, bus speed, bottleneck of path

storage devices: magnetic disks (rarely corrupt, large capacity, block level random access, better performance for straming access). flash memory(rarely corrupt, intermediate cost, good read perdormance, worse random write, erasure in large blocks, wear patterns)

Magnetic disk: unir of transfer- sector, ring of sectors = track, stack of tracks = cylinder, heads position on cylinders. tracks segregated by unused guard regions, length varies across disk, only outer half of radius is used

read/write: 3 stages- seek time (position over track), rotational latency (wait for sector to rotate), transfer time (transfer block of bits-sector-under the readwrite head)

Key: minimize seek and rotational delays.

Intelligence in controller- sector contains sophisticated error correcting (disk head magnet wider than track, hide corruptions), sector sparing (remap bad sector transparently), slip sparing (remap all sectors to preserve sequential), track skewing (sector # offset from one track to next)

SSD architecture reads- no seek/rotational delay, latency=queuing time+controller time+xfer time

highest bandwidth: sequential OR random reads

writing data is complex- can only write empty pages in a block (writes 10x reads, erasure 10x write)

pro: low latency, high throughput, no moving parts (lightweight, low power, shock insensive), fast

con: small storage, expensive, asymmetric block write performance, limited drive lifetime

startup cost: syscall overhead, OS processing, controller overhead, queuing

Littles Law: stable system N(ops) = B(ops/s)\*L(s), N=average#tasks, B=throughput, L=response time

M/M/1 queue: Tq=T¬ser x u/(1-u). M/G/1 queue: Tq=T¬ser x (1/2)(1+C) x u/(1-u)

λ=mean customers per s, Tser=mean time to serve customer, µ=service rate=1/Tser, u=server utilization = 1/µ, C=squared coefficient of variance, Tq=time spent in q, Lq=length of q = λ\*Tq

improve IO performance? queues absorb bursts, finite queues limits delays (but unfair/deadlock)

disk scheduling FIFO: long seeks, SSTF: shortest seek time first, but starvation, SCAN: closest request in direction of travel (elevator), C-SCAN: only goes in one direction- not biased toward middle tracks

kernel level driver: critical devices that must keep running, limited set of resources, avoid blocking

user drivers: non threatening, higher level primitives, called often- fast or background, can use threads/blocking

File system: layer of OS that transform block interface into files, directories, etc.

Components: disk management, naming, protection, reliability/durability

disk management: basic entities file/directory, access with 2 options: sector as vector [cylinder, surface, sector] (not used much), Logical Block Addressing- each sector has int address from 0 to max # sectors. track free disk blocks: bitmap, structure files: file headers.

user view of file: durable data structure. system view: bytes, OS view: blocks

open: name resolution (pathname to “file number”, makes file descriptor, retun handle

read, write, seek, sync on handle- map to desctiptor and blocks

directory: hierarchical structure, entry is collection of files, directories, each have name/attribute

file: named permanent storage, contains data, metadata (owner, size, last opened, access rights)

FAT (file allocation table)- linked 1-1 with disk blocks, file number is index of root of block list, file offset, follow list to get block number. grow file by allocating free blocks and linking in

but: no accress rights, no header in file blocks

UNIX Fast file system: file number index to inode arrays, inodes contain information, multi level index structure, scalable dir structure. freespace: bit vector with bit per storage block

FFS pros: efficient storage/locality for small & large, metadata & data; cons: inefficient for tiny files (inode + data), inefficient when file contiguous, reserve 10-20% free space- prevent fragmentation

More on directories: stored in files, can be read, but typically don’t – system calls to access, open/create traverse structure, mkdir/rmdir add/remove intries, link/unlink (DAG)

When can file be deleted: maintain ref-count of links to file, delete when last reference gone

Links: hard link sets another dir entry to contain file number, creates another name (path), each is “first class”. soft/symbolic link: directory entry contains name of file, map one name to another

New Technology File System (NTFS) variable length extent instead of fixed block, everything is sequence of <attribute:value>, mix direct/indirect freely, directories in B-tree structure, Master File Table – database w/ flexible 1KB entries for metadata/data, Extents- variable length contiguous regions, Journaling for reliability

In memory file system structures. Open resolves file name, finds file controle block (inode),, makes entries in tables, return index in table. Read/write use handle to locate inode, perform ops.

Authorization: Access Control Maxtix- resources across top (file, devices), domains in columns (user or group of users); Access Control List- store permissions with object, rwx for owner, group, world; Capability List- each process tracks which objects have permission to tough (out of favor)

Combination: users have capabilities called “groups” or “roles”, objects have ACLs (refer to users or groups), change object permissions by modifying ACL, change broad user permissions via changes in group membership

Revoke rights: ACL- remove entry from list; capability- in single machine keep all capability lists in well-known place, hard in distributed systems. Various approaches: expiration dates on capabilities, epoch numbers, back pointers to all that have been handed out, revokation list

Memory Mapped Files: traditional IO involves explicit transfers between buffers in process address space to regions of a file, instead “map” file directly to empty region of address space- executables

mmap sys call: map a specific region or let system find one, used for manipylating files & sharing

FS Caching key idea: exploit locality by caching data in memory. Buffer Cache: mem used to cache kernel resources, including disk blocks & name translations (can contain dirty). Replacement policy: LRU (but this fail when application scans through FS). Cache size: too big=few apps, too small=slow apps->adjust boundary dynamically. Read Ahead Prefetching: fetch sequential blocks early: exploit most common file access is sequential by prefetching subsequent disk blocks (elevator). Delayed writes: writes not immediately sent to disk- instead copy to kernel buffer, flushed periodically

Availability: probability that system can accept and process request (“nines”), failures independent

Durability: recover data despite faults (doesn’t necessarily imply availability)

Reliability: ability to perform required functions under stated conditions (availability,security,fault)

File system durable: disk blocks have reed-solomon error correcting, make sure writes survive short time (NVRAM, battery-backed), make sure data survives long term (replicate, independent fail), RAID: Redundant Arrays of Inexpensive Disks – data stored on multiple disks, SW or HW

RAID 1: disk mirroring, disk fully duplicated, B sacrificed on write, read may optimize

RAID 5+: high IO rate parity, data stripped across multiple disks, parity block constructed by XOR

Geographic Replication- highly durable, available for read, low availability for write

FS reliability: disk loses power/machine software crashes- raid doesn’t necessarily protect. approaches: careful sequencing of FS ops, copy on write, journaling, log structure

storage reliability: single logical file op can involve updates for multiple disk blocks, ata physical level, ops go 1 at a time

Reliability threats: interrupted operation, loss of stored data

Use of log: all changes = transactions, committed once written to log- data preserved in log

Log Structured FS: data stays in log form, Journaled: Log used for recovery- used to asynchronously update filesystem (removed after used). After crash: remaining transactions in log (“redo”)

General solutions: transactions or atomic updates, redundancy for media failures.

Transactions: closely related to critical sections, extend concept of atomic update, like flags

=atomic sequence of actions on a storage system that takes it from one consistent state to another

Typical structure- begin: get transaction id, do updates: if any fail/conflict roll back, commit

if started writing/crash recovery- redo, if uncommited was discarded- do again from scratch

Atomicity: all or no actions happen. Consistency: maintain data integrity (Ex. positive balance). Isolation: execution is isolated (no concurrency problem). Durabiltiy: if commit, effects persist

FFS create file: normal op- allocate/write data, allocate/write inode, update bitmap of free blocks, update directory with file name/number, update modify time. Recovery- scan inode table, if unlinked files delete, compre free blocks against inode trees, scan for missing update/access time

Application level: normal op- write name of open file to app folder, write changes to backup file, rename backup to be file, delete list in app folder on clean shutdown. recovery- on startup see if files left open, if so look for backup, ask user to compare versions

Copy on Write: way to copy large chunk between addr space. copied set of mem pgs put in the page table of dst process and left in the page table of the src process. Both set of PTE changed to “read-only”. if neither process tries to write to pages, result is like actual copy. If src/dst process attempts to write one of the pages, get a page fault, generate an actual copy page, clean PTE so they point at the two copies and marked writable, then allow the write to continue. pages only actually copy if will be modified. files grow incrementally as written, disk trends: huge/cheap, high startup, memory trends: reads from cache, buffer writes, application trends: make multiple changes and commit all or nothing. useful for unix fork

Emulating COW @ user level: transform file to new version, open/create new file, so updates based on old (reading/writing, copying unchanged) update linky

Creating new version: if file is tree of blocks, just need to update leading fringe

ZFS: variable size blocks 510b-128kb, symmetric tree, store version # with pointers, buffers collection of writes before creating new version, free space represented as tree of extents

Redo logging: prepare-write changes in transaction to log, commit- single disk write to make transaction durable, redo-copy changes to disk, garbage collection-reclaim space, recovery-read log, redo operations for committed, ignore uncommitted, garbage collect log

interleave transactions in log? if serializable

performance: log written sequentially, asynchronous write back, can process multipl transactions

isolation- prevent interleaving with locks- shared lock (mult concurrent transactions allowed), exclusive lock(only 1 transaction can operate on data at a time)

two phase locking (2PL): each transaction must obtain S or X before reading, X before writing, cannot request additional locks once it releases locks->guarantee dependency graph acyclic, conflict serializable. important variant strict 2PL, all locks released at end of transaction

serializability- with 2PL and rego locking, transactions occur in sequential order- other implemntations can also provide this

caveat: most FS implement transactional model internally, provide one for individual syscalls, but not for user data (likely historical artifact or unfamiliar model)

Centralized System: system in which major functions are performed by a single physical computer

Distributed System: physically separate comps working together on same task

DS motivation: cheaper/easier to build lots of simple, easier to add power incrementally, users complete control over some component, easier for users to collaborate. promise: higher availablility, better durability, more security. reality: worse availability (every machine up) reliability (lose data if crash) security (anyone can break in). coordination more difficult

DS goals/requirements: transparency- ability of system to mask complexity behind simple interface (location, migration, replication, concurrency, parallelism, fault tolerance)

Networking defs: network- physical connection that allows communication. packet- unit of transfer, sequence of bits carried over network. protocol- agrreement between 2 parties on how information is transmitted (syntax, semantics)

Namespace: hostname, IP address, port #

Client request, server provice, clent “sometimes on”, server “always on”. P2P: no server at center

Broadcast networks: shared medium (set of wires), delivery: put header on front of packet

Arbitration: act of negotiating use of shared medium- carrier sense, collision detection, mult access

Backoff scheme- adaptive/randomized

Point to point networks: network in which every physical wire has 2 computers. switch: bridge that transforms shared bus into point to point, router: device acts as junction between two network

IP packet: network packet, IP address: 32 bit, Internet Host: computer

Subnet: network connecting set of hosts with related destination, prefix of bits

Address ranges: Class A /8, Class B /16, Class C /24

LAN: designed to cover small geographical area, WAN: link geographically separated sites

routing: forwarding packets hop by hop to destination- routing tables destination->output link

set up: routing table has “cost”, neighbors periodically exchange (distance vector)

naming DNS: herarchical mechanism for naming- resolution, caches. not very secure

Layering: build complex services from simple ones

arbitrary sized methods: fragment, checksum. internet: “best effort”

IP packet: version, HL, size, ID, flags/fragmentation, TTL, protocol, checksum, src, dst- 20 byte

UDP: unreliable datagram- source port, dest port, length, checksum

ordered messages- queuing at destination- sequence numbers- acks

alternating bit protocol- one at a time with acks

windowing protocol- N packets at a time w/ sequence number, acks for reliability + ordering

TCP fragment into packets, then to Ip, window based protocol, auto retransmit, congestion ctrl

selective acknowledgement-includes which packets have been received

congestion control: slow start (+1 each ack), timeout->cut size in half (AIMD)

seqno initialization: random, epoch # identifies which set is being used (stored on disk)

socket: abstraction of network IO queue, one side of communication channel.

server: create socket, bind to protocol, local address, port, call listen(), accept()- return new socket

client: create socket, bind to protocol, remoe address, port, connect()

Distributed applications- how to program? interface mailbox, send, receive

messaging for producer-consumer-style: producer keeps sending, consumer keeps receiving

request response/client server: request, get, receive, send

Generals Paradox: messengers can be captured, need to coordinate attack- no way to be sure.

->2 phase commit: use persistent stable log to keep track of whether commit has happened. global coordinator. prepare: record promise to commit or rollback, else record to abort. commit: coordinator records commit, ask all to commit, write “got commit”

Distributed decsion making- fault tolerance. undsireable feature of 2phase commit- blocking

PAXOS: alternative that does not block. what if one is malicious?

byzantine general problem- n-1 liutenants, some numebr can be malicious, all loyal obey same

->impossibility: with f faults, need n > 2f to solve. various algoritms exist (bft).

Remote Procedure Call: raw messaging too low level for programming: remote procedure call(call procedure on remote machine). implementation: request-response messge, “stub” provide glue

Marshalling: convert to canonical form (network/host), serialize objects, copy pass by reference

RPC details: equivalence with regular proceture call(parameters-request msg, result-reply msg, name passed in request, return client mailbox), stub generator-compiler that geerates stubs (input interface definitions , outut stub code), cross platform issues-convert to same canonical form, which mbox client send to (need translate name of remote service to network endpt, binding: process of converting user-visible name into network endpoint), dynamic binding (access contol, fail-over), mult servers- flexibility at binding+ provide same mbox, mult clients-pointer to mbx

Problems with RPC: non atomic failure (different fail mode in distributed sys than single machine, different types of failures, one machine die while others go->inconsistent view, answer=byzantine commit), performance-cost of procedure call

Cross-domain communication/location reansparency- address spacees communicate (shared memoty, FS, pipes, RPC), RPC used to communicate b/w address spaces on diff or same machine

Microkernel OS: split kernel into separate domains (fault isolation, modular, location transparent)

3 way handshake: agree on set of parameters(start seqno)- connect() send SYN, server accept() SYN ACK, ACK. Add1 RTT delay. Close: FIN FINACK from both.

New API (NAPI): use polling to receive packets (only osme drivers), exit hard intr contxt asap

CAP: Consistency: changes appear to everyone in same order, Availability: result at any time, Partition-Tolerance-system works even if network partitioned. theorem-cant have all

Simple Distributed File System: Remote Disk-reads/writes forwarded to server, Advantage-consistent view of FS, Problem-Performance (slower than local mem)

Caching reduce network load: Advantage-fast, Problem-failure data not committed, consistency

Failures- Server crash->data can be lost. Stateless protocol: all info required to process request is passed with request. Client crash-data in caches

Network File System (NFS): 3 layers (UNIX FS interface, VFS layer-local from remote, NFS service layer-bottom layer, implement NFS protocol). NFS protocol: RPC for file ops on server. Write-through caching: modified data committed to servers disk before results returned. Servers stateless: each request procides all arguments needed. Idempotent: request multiple times has same result as once. Failure Model: transparent to client

NFS protocol: weak consistency- client polls server periodically, 2 clients writing can get either ver

Sequential ordering/cache coherence- if read >30sec acter write, new copy

Pros: simple, highly portable. Cons: sometimes, inconsistent, not scale up to large # clients

Andrew File System (AFS): Callbacks-server record who has copy of file, write through on close, everyone who has file open sees old version, data cached on local disk of client and memory, if server crash lose callback state. Pro: less server load than NFS, central server bottleneck for both

 Virtual Filesystem(VFS): virtual abstraction similar to local FS, allow same syscall interface for different FS types (in linux: Virtual Filesystem Switch).

Common File Model: 4 primary object types (superblock-specific mounted FS, inode-specific file, dentry-directory entry, file-open file assoc with process). no specific directory object(treat as file)

Linux VFS: operations object contained within each primary object type to set ops of specific filesystems (super\_operations, inode\_operations, dentry\_operations, file\_operations)

65535 ports are limitations of TCP/UDP, not IP (software, not hardware)

constant bit density = faster read on outer tracks

queuing equations work for steady state only

RAID 5 recovers 1 failure only

TLB can be done in parallel to cache lookup

bottom half of device driver responds to interrupts

DNS is a distributed service, not centralized.

minimum routing table: dest name, next hop. no cycles: if already in path, drop advertisement

contiguous allocation leads to external fragmentation (space that may not be allocatable)

buffer cache: temporal locality reads, smaller data than block, reorder writes (seek time), prefetch

buffer cache: reads can be from cache instead of disk, delayed writes to disk, cache kernel resources

rename() can be used to change location, more efficient than copying/deleting

when moving across disks, no more efficient way than to copy/delete.

nonblocking IO returns regardless of the state of operation (partial read/write), asynchronous notifies user later when operation has completed (signal)- will perform complete operation

inode keeps track of sectors allocated

Tread= controller+seek+rotational+xfer time, rotational time = time to rotate half of disk, xfer = bytes/BW

manufacturer measures seek time by avg time to seek from random track to another. real=locality

fully associative cache: no conflict misses, but bigger/slower than direct map

modern processor have reorder buffer that permit exec results be flushed back to exception point

two phase commit- all participants commit to op. no violate gen paradox b/c no guarantee when

Aloha sends without checking- retransmit after garbled packet- could interrupt in progress msg

mem mapped IO can be accessed by user level if PT has mapping for it (kernel can map to user space)

compulsory misses can be reduced by prefetching

memoryless prob distributions can approximate many independent sources

more than one computer can have same ip address

7 bit VPN-> size of page table = size of one mem page

address more phys bits – use 0 bits of PTE. TLB: valid, VPN, PTE

want precise exception after TLB fault because easier to implement the fault handler

spatial locality: access a location close to or next to recently accessed location. ex. sequential file access

disk scheduling algorithm minimizes time for moving disk arm (seek)

journaled performance/durability: smaller traffic to disk, log spatial locality, when fs update requests can be ordered for higher efficiency

Skip sector positioning: place successive sectors on every other sector. avoid situation in which the processor does work after reading sector, misses the next sector & has to wait for a complete revolution. Modern disk controllers have track buffers.

stateless: server can crash transparently, but can’t track if clients are caching data

if disk as fast as memory: write through instead of write back memory, polling instead of interrupt

max data: MTU – header. data bandwidth: data fraction \* bottleneck bandwidth

window size = latency \* bandwidth

memory mapped IO accessed with normal loads/stores

byzantine agreement algorithm needs 3f+1 nodes to tolerate f faults, single file server can’t tolerate

software TLB: same hardware can support forward and inverse PT

computer virus requires human interaction to spread, worm doesn’t

write-behind policy: more dirty blocks (better arm scheduling), tmp files can write+del, data may be lost

mitigate disadvantages: NVRAM

FAT faster for random access, linked allocation faster for sequential if unable to keep FAT in mem

max disk size = 2^(file block ptr bits)

FFS improvements: mostly sequential data, inodes in same group as data so successive chunks of files is fast, inodes/data for files in a directory in same group->dir operations fast

AFS: when file updated, tells users with read-only copies to invalidate

some of 2^32 ip addresses are “unrouteable” and “private”

correct thrashing by decreasing running threads (working sets don’t fit in memory)

complexity of software TLB increases with imprecise exceptions

byzantine agreement when distributed decision has malicious nodes

the UNIX BSD inode structure contains 10 direct pointers to blocks, 1 pointer to a block of pointers (indirect block), 1 pointer to a doubly-indirect block, and 1 pointer to a triply indirect block. Small files are supported efficiently with the direct pointers, while large files are supported through all the levels of indirection. Small files are handled more efficiently (can read blocks directly, given the inode structure).

primary access pattern sequential->link each file to the next

KV Storage: huge volumes of data, Simple Interface (“put”, “get”), simpler & more scalable “database”. Also called Distributed Hash Tables, partition set of key-values across many machines

Challenges: • Fault Tolerance: handle machine failures without losing data / degradation in performance • Scalability: – scale to 1000s of machines – need easy addition of new machines • Consistency: data consistency in face of failures and msg losses • Heterogeneity (if p2p): – Latency: 1ms to 1000ms – Bandwidth: 32Kb/s to 100Mb/s

Directory Based Architecture: node maintain mapping b/w keys & machines (nodes) that store the values associated with the keys

master relay requests = recursive query • iterative query – Return node to requester

• Recursive Query: – Advantages: » Faster,closer to nodes » Easier to maintain consistency, serialize puts()/gets() – Disadvantages: scalability bottleneck, as all “Values” go through master/directory • Iterative Query – Advantages: scalable – Disadvantages: slow, hard consistency

Fault Tolerance: • Replicate value on several nodes • Usually, different racks. can do recursive/iterative rep or recursive q, iterative rep

Scalability: Storage: use more nodes • Number of requests: – Can serve requests from all nodes on which value is stored in parallel – Master can replicate popular value on more nodes • Master/directory scalability: – Replicate it – Partition it, so different keys are served by different masters/directories

Load Balancing:

• Directory keeps track of the storage availability at each node – Preferentially insert new values on nodes with more storage available • What happens when a new node is added? – Cannot insert only new values on new node. Why? – Move values from the heavy loaded nodes to the new node • What happens when a node fails? – Need to replicate values from fail node to other nodes

Consistency: • Need to make sure that a value is replicated correctly • How do you know a value has been replicated on every node? – Wait for acknowledgements from every node • What happens if a node fails during replication? – Pick another node and try again • What happens if a node is slow? – Slow down the entire put()? Pick another node? • In general, with multiple replicas – Slow puts and fast gets. If concurrent updates (i.e., puts to same key) may need to make sure that updates happen in the same order. Large variety of consistency models: – Atomic consistency (linearizability): reads/writes (gets/puts) to replicas appear as if there was a single underlying replica (single system image), Eventual consistency: given enough time all updates will propagate through the system, causal consistency, sequential consistency, strong consistency

Quorum Consensus: • Improve put() and get() operation performance • Define a replica set of size N – put() waits for acknowledgements from at least W replicas – get() waits for responses from at least R replicas – W+R > N • Why does it work? – There is at least one node that contains the update • Why might you use W+R > N+1?

Scaling Up Directory: Challenge: – Directory contains a number of entries equal to number of (key, value) tuples in the system – Can be tens or hundreds of billions of entries in the system! • Solution: consistent hashing • Associate to each node a unique id in an unidimensional space 0..2m-1 – Partition this space across m machines – Assume keys are in same uni-dimensional space – Each (Key, Value) is stored at the node with the smallest ID larger than Key

Lookup in Chord-Like System: • Assign IDs to nodes Source – Map hash values to node with closest ID • Leaf set is successors and predecessors – All that’s needed for correctness • Routing table matches successively longer prefixes – Allows efficient lookups • Data Replication: – On leaf set

Security Today: Computing in the presence of an adversary! – Adversary is the security field’s defining characteristic • Reliability, robustness, and fault tolerance – Dealing with Mother Nature (random failures) • Security – Dealing with actions of a knowledgeable attacker dedicated to causing harm – Surviving malice, and not just mischance • Wherever there is an adversary, there is a computer security problem!

Protection/Security: • Protection: mechanisms for controlling access of programs, processes, or users to resources – Page table mechanism – Round-robin schedule – Data encryption • Security: use of protection mech. to prevent misuse of resources – Misuse defined with respect to policy » E.g.: prevent exposure of certain sensitive information » E.g.: prevent unauthorized modification/deletion of data – Need to consider external environment the system operates in » Most well-constructed system cannot protect information if user accidentally reveals password – social engineering challenge

Requirements: • Authentication – Ensures that a user is who is claiming to be • Data integrity – Ensure that data is not changed from source to destination or after being written on a storage device • Confidentiality – Ensures that data is read only by authorized users • Non-repudiation – Sender/client can’t later claim didn’t send/write data – Receiver/server can’t claim didn’t receive/write data

How to identify users to the system? – Passwords » Shared secret between two parties » Since only user knows password, someone types correct password ⇒ must be user typing it » Very common technique – Smart Cards » Electronics embedded in card capable of providing long passwords or satisfying challenge → response queries » May have display to allow reading of password » Or can be plugged in directly; several credit cards now in this category – Biometrics » Use of one or more intrinsic physical or behavioral traits to identify someone » Examples: fingerprint reader, palm reader, retinal scan » Becoming quite a bit more common

Cryptography: communication in the presence of adversaries • Studied for thousands of years – See the Simon Singh’s The Code Book for an excellent, highly readable history • Central goal: confidentiality – How to encode information so that an adversary can’t extract it, but a friend can • General premise: there is a key, possession of which allows decoding, but without which decoding is infeasible – Thus, key must be kept secret and not guessable

Symmetric Key: Same key for encryption and decryption • Achieves confidentiality • Vulnerable to tampering and replay attacks, can just XOR plaintext with key

Data Encryption Standard (DES) – Developed by IBM in 1970s, standardized by NBS/NIST – 56-bit key (decreased from 64 bits at NSA’s request) – Still fairly strong other than brute-forcing the key space » But custom hardware can crack a key in < 24 hours – Today many financial institutions use Triple DES » DES applied 3 times, with 3 keys totaling 168 bits • Advanced Encryption Standard (AES) – Replacement for DES standardized in 2002 – Key size: 128, 192 or 256 bits • How fundamentally strong are they? – No one knows (no proofs exist)

Asymmetric Encryption (Public Key) • Idea: use two different keys, one to encrypt (e) and one to decrypt (d) – A key pair • Crucial property: knowing e does not give away d • Therefore e can be public: everyone knows it!

Properties of RSA • Requires generating large, random prime numbers – Algorithms exist for quickly finding these (probabilistic!) • Requires exponentiating very large numbers – Again, fairly fast algorithms exist • Overall, much slower than symmetric key crypto – One general strategy: use public key crypto to exchange a (short) symmetric session key » Use that key then with AES or such • How difficult is recovering d, the private key? – Equivalent to finding prime factors of a large number

Digital Certificates • How do you know KE is Alice’s public key? • Trusted authority (e.g., Verisign) signs binding between Alice and KE with its private key KVprivate – C = E({Alice, KE}, KVprivate) – C: digital certificate • Alice: distribute her digital certificate, C • Anyone: use trusted authority’s KVpublic, to extract Alice’s public key from C

Summary of Our Crypto Toolkit • If we can securely distribute a key, then – Symmetric ciphers (e.g., AES) offer fast, presumably strong confidentiality • Public key cryptography does away with (potentially major) problem of secure key distribution – But: not as computationally efficient » Often addressed by using public key crypto to exchange a session key • Digital signature binds the public key to an entity

Datacenter/Cloud Operating System • Data sharing – Google File System, key/value stores – Apache project: Hadoop Distributed File System • Programming Abstractions – Google MapReduce – Apache projects: Hadoop, Pig, Hive, Spark • Multiplexing of resources

GFS/HDFS Insights • Petabyte storage – Files split into large blocks (128 MB) and replicated across several nodes – Big blocks allow high throughput sequential reads/writes • Data striped on hundreds/thousands of servers, • Failures will be the norm – Mean time between failures for 1 node = 3 years – Mean time between failures for 1000 nodes = 1 day • Use commodity hardware – Failures are the norm anyway, buy cheaper hardware • No complicated consistency models – Single writer, append-only data

MapReduce Pros • Distribution is completely transparent – Not a single line of distributed programming (ease, correctness) • Automatic fault-tolerance – Determinism enables running failed tasks somewhere else again – Saved intermediate data enables just re-running failed reducers • Automatic scaling – As operations as side-effect free, they can be distributed to any number of machines dynamically • Automatic load-balancing

MapReduce Cons • Restricted programming model – Not always natural to express problems in this model – Low-level coding necessary – Little support for iterative jobs (lots of disk access) – High-latency (batch processing) • Addressed by follow-up research and Apache projects