**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**Context Switch**: TLB entries no longer valid.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=T­ser x u/(1-u). M/G/1 queue: Tq=T­ser 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)

**FA**T (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