OS Cheatsheet
Color encoded: unique pro con meta hw additioanl

THE: multiprogramming, batch. Layered. IPC: smem, semaphore.
Hierarchy, layered OS structure: process control, mem mgt, msg interpreter, peripheral, user-level, user
Layer: modular, testable debug, easier to develop and verify (proof); hard to define layers, circular dependency, not efficient (go thru layers), low flexibility
Semaphore for sync: mutual exclusion (directly use shared mem) or private semaphore (record status, while conditional variable is not remembered)
Harmonious cooperation (no deadlock): hierarchical call
Sharing: shared mem. efficient, easy to code; fault intolerant (corruption), race condition.
Nucleus: extensibility. Fork, proc hier tree.
Motivation: make things general and flexible. **Structure**: two layers, OS and user program
3 parts of nucleus: 1. processes (internal & external → based on device), comm., scheduling (quit, stop,...); 2. peripheral: a device; 3. document: data or file
Msging: sent/wait msg & answer: assume a server/client model (need to wait for ack.)
IPC: msg (indpt, isolated, scalable, safer, explicit coor, easy, extra space as buffer, buffer limit, copy, queue)
TENEX: **BBN pgr**= translation + rights checking, pg share: indirect ptr, CoW.
Backward comp: routines trap n translate instructions. library in user space (old instruction set) (another way: emulate: 32bits on 64-bit)
Scheduler: WWS (aim no starvation, interactive higher priority, maximize CPU → minimize pg fault): working set (pg fault interval → too coarse)
FS: device/directory, 5 level tree, **protection**: usr/grp/others, versioning. (obsolete)
Specialized hw a) **Instance**: The BBN Pgr. The BBN Pgr implemented VAddr translation; also performing rights checking.
Hw supporting the Multics descriptor-based VAddr spaces is another example. Similar to the BBN Pgr, on each instruction and mem reference the Honeywell hw would translate VAddres to physical addresses and perform rights checks on the accesses.
b) **Pro**: The BBN Pgr made pgd VMem perform well (greatly reduced overhead of VMem), and it provided useful features for VMem: shared mem, pg permissions, etc.
In addition to similar advantages of the BBN Pgr, the Multics hw supported more sophisticated VAddr spaces: segments (which worked well with mem mapped files) and rings (generalization of the usr-K mode into 8 protection levels). As a result, usrs of Multics could implement their own protected subsyss, not just the OS.
c) **Con**: sys becomes tied that hw platform, making it unportable.
d) **Use today**: The BBN Pgr is essentially an early form of MMUs that we use today for demand-pgd VMem. Multics descriptor checking in hw is also essentially done today by MMUs in general (descriptors == PTEs at a very high level).
Hydra: ¬ layer: Proc + LNS + Obj. Right amp. Capability, flexibility vs. usability.
Goal: extensive, extensible, modular, secure → reject hierarchy (no ownership), separate mechanism (protection) and policy (security)
Hydra: capability-based protection, a micro-K design (flexible to construct OS)
Capability: reference to an object, access rights, manipulated by K (can't be forged); unique name, type, representation
- proc (reference to objects, static, template → check expected parameter)
- Local Name Space (domain): runtime env, unique for each run, capability list
- Process: stack of LNS: IPC: msg buffering, semaphore
Protection (a) **Protection dom**: Proc (b) **Entry**: Proc call (c) **Rights**: Capabilities
(d) **Amplification**: Procs enhance rights of capabilities using rights defined by proc templates
"If the caller's rights are adequate, a capability is constructed in the (new) LNS which references the object passed by the caller and which contains rights formed by merging the caller's rights with the rights specified in the template." "(A template has a "regular" rights field distinct from the check-rights which specifies the rights which the callee (i.e. the proc) will need to operate on the actual parameter.) This implies that a callee may have greater freedom to operate on an object than the caller who passed it as a parameter."
(e) **Allow**: Rights on proc capability and argument capabilities have to match rights specified in proc definition. Specifically, there are two kinds of rights: the "type" right and "check-rights."
"If the types agree, the rights are then checked, using a special field present only in templates called the "check rights field." The rights contained in the actual parameter capability must include the rights specified in the check-rights field of the template otherwise, a protection failure occurs and the call is not permitted."

Protection: **Protection**: access control from program to resrc
Patch program: modular programming (gain confidence in dev), proprietary program (can call not read)
Protection domain: protection env or context, entities have access to objects
Msg sys: isolation, every process is a domain (guarded at call and return); flexible and general; impossible to control runway process (resrc waste), hard to cooperate
Object (global id) and access matrix (sparse): **capability list**: attached to process/domain, what objects a domain can access (a column of access matrix); **ACL**: attached to object, recording domains (using access key) who have accesses (a row of access matrix) → access proc or a list, group objects into directories to save mem; **mixed**: Cap+ACL=Hybrid, use ACL to establish cap 1st time access, then cap as subsequent access. Capability vs. ACL: setup, lookup, transfer, revoke, store.
Each domain has its own address space → enforce protection; domains are objects, objects do not 'live in' or 'belong to' domains.
Fine-grained protection domains with very little overhead due to hw support: It supports **flexible** sharing of data between apps and simplifies the construction of protected subsystems. Compared with previous protection domain approaches, we remove the restriction that permissions are managed at pg **granularity**. Compared to capabilities, we provide a more flexible permissions model, fast rights **revocation**, and compatibility with current linearly addressed ISAs. Our feasibility study indicates that the space and **runtime overhead** of providing this fine-grain protection is small and scales with the degree to which fine-grain protection is used. Our zero-copy networking example shows how our new facilities can be used to implement efficient apps.
Multics: ACL rwxc, mmap, mem seg descriptor (ACL derived cap), protected subsys, ring (right amp). **Protection principles**: permission ¬ exclusion; check ∀ access; ¬ obscure; least privilege; usability.
Goals: decentralize setting of protection specification, customizable protection scheme → basic and extendable
Protection: first → ACL; second → hw (descriptor, check every access); all info mapped into vm, supervisor is a subsys and operates with same addressing and code; descriptor has least privilege
Segmented mem: hw to check segment (cataloging unit, data and code → data and execution), segment descriptor (modern PTE), every segment has a descriptor (protected sys, permission to call → easier to program and extend); FS is integrated with the mem access sys so that programs access files by making mem references
Access format: individual user. projects. Compartments rw/(none) → * as don't care
Sharing: physical segment mapped to multiple domains
ACL (on segment): easier to store, modify, and understand; slower to check (directory...)
Capabilities: easier to check and transfer; hard to setup and revoke
"open file" → check ACL → return file descriptor(key or capability) → descriptor can pass to child process (no need to check the access all the time)
each computation has its own private descriptor segment
during mapping, Multics creates descriptors that encode rights derived from ACLs **Protected subsys**: collections procs and data can be used via designated entry (gates) **Ring**: right amplification, checked by hw with descriptors, call same or bigger number **Authentication of users**: unique, password, time-out, logging and penetration detection **Con**: program modules can compromise protection sys, user interface complex, others along the whole sys links
Ref: Multics implemented a single-level store for data access, discarding the clear distinction between files (called segments in Multics) and process mem. The mem of a process consisted solely of segments that were mapped into its address space.
Protection (a) **Protection dom**: Rings -- executing at ring level N gives a process privilege to access code and data at level N and lower protection levels (higher ring numbers). Note that this access applies to segments mapped into your address space; you cannot access another process's segment arbitrarily even if it is at the same protection level you are running at. You would have to map that segment first, and mapping it would require checking the ACL for the segment to see if you can get access to it.
or: protected subsyss -- a slightly narrower definition than the first one is the set of all segments at level N and lower privilege...that you have mapped into your address space.
(b) **Entry**: protected procs via call gates. Note that ACLs are not quite the right answer here; they are involved in mapping a segment, but dom crossings happen after a segment has been mapped.
(c) **Rights**: ACLs, descriptors, ring # in descriptor (d) **Amplification**: Change ring level
(e) **Allow**: Ultimately a process wants to call into a segment with higher privilege. A process does this by calling into a gate into that segment. This step is equivalent to a sys call. From this perspective, once a segment is mapped all calls are allowed (one answer). Now, just like sys calls, protected procs will check the parameters to see if the call is permissible.
First, though, we need to get the segment mapped into the address space. mapping a segment requires checking the ACL on that segment for the process (another answer). Given the importance of checking ACLs for segment mapping, this answer captures an important aspect of how Multics works.
UNIX: FS: ∀ f (device abstraction), f: a linear sequence of bytes. FS handler not MMU, inode, mnt, link. File exists indptly of directory (hard links, i-node link), i-list for quick consistency check
Portable.
Protection: ACL, 7bits rwx, owner/other, no grp yet. **set-usr-id**: assign user id of file owner to execute, right amp. **Proc**: fork, CoW, exec, wait, exit, & mnemonic (run in background), IPC by pipes.
Protection (a) **Protection dom**: Address space -- address spaces are protected from each other (usr-level vs. usr-level, usr vs. K), going from one to the other requires OS support. This is also the most precise definition.
or: Usr-level vs K-level -- but note that just being a process at usr level does not mean that you can access everything (other processes) at usr level
or: Everything accessible by a given usr – an acceptable answer, but does not reflect anything about aparticular sys (the same statement applies to all).
(b) **Entry**: Sys call (preferred), or: setuid (also acceptable)
(c) **Rights**: ACLs (FS), PTs (when executing in mem), CPU mode (usr/K) or: if setuid answered above, then the consistent answer is usr id.
(d) **Amplification**: Change CPU mode, or: if setuid answered above, then amplification is represented by running as the usr id of the owner of the file (rather than the usr id of the process opening the file).
(e) **Allow**: Use descriptors (read and write check fd for validity), usr id (is the process running as root?), or ACLs (open checks file ACL) to decide whether to perform sys call once in K.
or: anything allowed, can invoke any sys call (inside the sys call the OS will then determine whether to allow further execution)
Multi-usr OS. 1) **Multi account**. We may still want to provide the ability to have multiple usr accounts for different usrs.
2) **Protection**. To adhere to principle of least privilege, by having the ability to create such accounts, we can use the protection sys designed for multiple usrs to provide protection among all of these services and roles in the os to provide privilege specifically and only when and where needed. In Unix, the sys itself has a number of usr accounts (virtual usrs) for running the sys: "root", for complete control; "nobody", for an account with minimal access (e.g., for running a Web server); "daemon" for sys processes; accounts associated with particular services (ftp, nfs, etc.).
3) **Security**. Nowadays, security is a critical concern for all usrs connected to the Internet. It is very difficult to achieve security either with no protection, or with only defensive protection. You get protection doms, principals (usrs), ACLs, capabilities, etc. These mechanisms and the ability to express policies are naturally provided by multi-usr OSes.
Proc debug: A debugger provides an environment in which the user can freeze and inspect the state of his executing program. However, in order for the user to have access to such an environment, three things must happen :
1. A complete **isolation** from other programs (otherwise, otherwise other programs may interfere with the user's inspection).
2. A way to **freeze** the execution of a process.
3. The debugger (i.e. another proc) must be able to inspect the data of the debugged program, thus **access mem** of the executing proc.
All three of these actions required operating system support. As such, debugging would not be possible without help from the OS.
Comm channel: In order to provide the aforementioned **isolation from other procs**, the debugger must stop contact with the regular means of communication and coordination. The only communication allowed is one that is controlled and monitored by the debugger, thus it has to happen through specialized communication channels.
Java: Java do not need OS support, debug tools provided in JVM.
Plan 9: Network extension Unix. f server
Reject time-sharing on personal machine → nodes specialized (file, cpu, routers, terminal → dkless, no persistent state, personalized on demand)
File abstraction: file (∀, and implements operations), file servers (all service, exports file hierarchies),
Namespace: local namespace (follow convention → familiarity: dev/cons/, dev/mouse)
File server: mem, dk, WORM; one main server holds all sys files (centralized mgt, yet no superuser)
∪ **op**: combining directories from many namespaces
9P: protocol, implements RPCs for all file "methods", resolve paths, access file by bytes
Storage hierarchy: WORM, no backup, daily snapshot (incremental, monthly full backup) → need checkpoints → not today, we have more storage
Contributions: UTF-8, rfork (specify what to share), /n/dump (snapshot).
Storage and bandwidth: "Technology has created storage faster than we can use it" The statement (as quoted) is still true, but the problematic surrounding it has evolved. In the days of plan 9, all computers (personal computers, mainframes, computer labs...) their data was stored locally. The data was either stored in a central location, or was distr over a local area network.
Nowadays in the age of the Internet, the statement is no longer true. Home users use cloud based web services. Data is global, and jumps from one spot to another before reaching your smartphone's screen.
storage has been cheap and easy to find. However, delivery is another matter. The quoted sentence could be modernized as:
"Technology has created storage faster than we can use it, but not bandwidth." An average internet user does not expect a web pg to load in more than a second. There is a lot of travelling and processing for just one second of expected latency, and bottlenecks can arise for a multitude of reasons (limited bandwidth, limited dk access, request saturation, missing/outdated data...). Caches can be used at multiple levels to alleviate some of the incurred latency cost, but it also constrains how much data can be accessed at once. This is why internet requests usually only involve small data loads, as to not constrain servers or create bottlenecks in the described process.
Medusa: Distr: utility. Centralized vs. full replica vs. utility: consistency, admin ez, SPF, redundancy, reliability, perf, space. Descriptor→K obj,
Goal: build modular, robust, efficient distr sys (match hw of cm*) → measure success by evloving
Compare 3 distr OS designs: centralize in one processor (easy maintain; single point of failure, bad perf.), full replica at each machine (waste mem → not today), utilities (redundancy)
Structure: Cm, KMaps (inter-cluster), SLocal → local hit ratio
Utility: single OS module, communicate with others via msg (isolation), redundancy for load balancing and reliability; sys call as remote invocation
Activity: process to run utility (communicate via pipes as "trap", descriptor to access object)
Descriptors: access to objects is obtained through descriptors. UDL (utility entry points), PDL (ref private pgs, pipes), SDL (ref shared pgs, task force), XDL (For mapping a local descriptor onto a remote PDL/SDL descriptor (stub; PROXY))
failover: (un)sealing callee (utility) state in caller, change remote XDL. **Unsealing**: When callee fails, Caller keeps the object accesses state; keep the object in its own space and protect from other proc (seal). A FS needs to maintain file objs and keep track to of state of opening files. store file object in the app's address space. No need to move files at fail-over. Need to protect that file object from the app itself. K objects stored in the app's address space are sealed to the app; thus the app cannot directly manipulate it. When an activity in a utility needs to manipulate the K object, the object is remote, so it needs to map it as an external obj XDL and thereby unseal it; then manipulate it; then seal it again (remove from its XDL).
Co-scheduling: A task force is said to be coscheduled if all of its runnable activities are simultaneously scheduled for execution on their respective processors. activities (schedule unit), schedule in sync, similar to spin-wait (wait a while before swapped out); coroutine and multi-eventing (use one pipe to receive). spin-wait: quick acquire lock, waste CPU.
Task force (**concurrent**): A task force is defined to be a collection of concurrent activities that cooperate closely in the execution of a single logical task. for FS, throughput, for compiler and program, speed up single request
Deadlock free: single utility provides all services of a class, no cycle in class dependencies
Exception handler: remote processor to handle exception, emulating as other OS (in Pilot, "world swap", save everything to dk and load debugger)
vs Plan9: (1) medusa into utilities, plan 9 join hierarchies into a single private hierarchical file name space (2) plan 9 res are named and accessed as files, Medusa is not
Pilot: Mesa, coupled. Hint not coerce. Single usr PC, protection defensive not absolute. Flat FS. Mem: Linear addr space partitioned into hier *space*, f mmap. Socket. Stream (FS) vs. mmap (MMU): init OH, seq/rand access pattern, multi/single-pass, sharing.
Assumption: PC → single-language, single user, single address space (no process scheduling), defensive protection (error than malicious); maliciousness is less serious (not true today)
A K on which additional feature can be layered (no sys per se) → provides mechanism
File data access: mmap, go thru VMem (does permission check, read/write in mem easy to implement); global naming (64 bit): has machine number; no directory; Permanent (after sys restart) and immutable file (protection)
VMem: space and subspace (continuous) for allocation/ mapping/ swapping → hierarchy
Circular dependency: K/manager breaks circularity → interlocking structure
Comm: 64 bits address, TCP over network (in v.m.), IPC on the same machine, via socket; transducer(device to stream), filter.
Mesa: Benefit: type-safe language, garbage collection, mem reference safe. Drawbacks: performance: slow (garbage collection is low efficient, type check before accessing mem)
Protection (a) **Protection dom**: everything is in a single dom (b) **Entry**: No doms to cross
(c) **Rights**: Given that everything is in a single address space, in a certain sense there are on rights to represent (one answer). However, we do still value defensive mem protection (e.g., read-only code segments). so Pilot does have rights associated with spaces (read, write, execute), and these define access rights during execution (another answer, more precise). Note that there is not much of a discussion on the FS, so it's hard to talk about ACLs, etc.
(d) **Amplification**: None
(e) **Allow**: Nothing checked (or type checking)
mmap vs. stream: Use mem maps if you access data randomly [random access], keep it around for a long time [**intensive usage**], or if you know you can share it with other processes [**sharing**].
Read files normally if you access data sequentially [**sequential usage**] or **discard** it after reading. And if either method makes your program less complex, do that.

Monitor: higher level wrap up of lock, K just uses monitors to manage resrcs (just critical resrc, limited concurrency, mutex is more free mechanism)
Hoare: Signal: force. bare wake, not flexible, lock on all procs, module granularity, suggest priorities inside monitor;
I{b.wait}I&B; I&B{b.signal}I
Mesa: Monitor, monitor vs. lock: coarse-grain →perf. Signal: hint vs. force, perf, scheduling responsibility.
Hint: wake up is a hint, no require for OS scheduling (no context switch) → waiter check condition, lock on entry, in monitor class/record granularity (fine grained control), priorities on processes → priority inversion problem (temporarily rise up to monitor's priority, so won't be preempted)
Java: lock on synchronized, each object has a conditional variable (implicitly, easier to program), thread.stop → deprecated ∵ causing inconsistent state
I&!B{b.wait}I; {b.signal} For wait, I has to hold before and after calling wait ∵ the monitor lock will be released and then acquired again. B does not hold before wait by definition. The state of B is unknown after the wait - it could be valid (if we are the first to wake up after a signal), or it could be invalid (another thread woke up and invalidated the condition).
Signal does not release the monitor lock, technically you do not need to have either the invariant (∵ the lock is not released) or condition (no thread can run yet to see if the condition holds yet) hold.

```
...
[1]
while ( ... ) {
    [2]
    ...
    [3] b.wait [4]
    ...
}
[5]
...
```

[1] assume nothing at this point. [2] would not be inside the while loop unless B = !hold. [3] this is just before calling wait. B does not hold ∵ we are inside the loop. I has to hold ∵ we are releasing the lock. [4] this is just after the wait. I holds ∵ we just acquired the lock. B is unknown ∵ we do not know if we are the first thread to enter the monitor after a signal. [5] we can assume B holds ∵ the while loop predicate tells us that it does.

VAX/VMS: **Challenges**: mechanism and policies to support a wide range of apps, time-sharing, real-time, batch sys.
(Stack heap,) x (usr, K). **Address space**: P0 (data), P1(control), sys(OS), res (reserved); OS (VMS) in every process address space (no context switch), single-threaded, apps cannot change OS part. K stack with P to avoid TLB flush when context switch.
VMem: logical pgs. Continuous PT, PT: usr PT in pgd K addr space, K PT in phy mem.
Local pg replacement: RSS, FIFO. Pg caching: 2^{nd} chance list r w caches. Clustering: for dk IO.
Optimization: local pg replacement (app replace from its own resident set → isolation, upper bound on resrc), pg caching (free & modified pg list), clustering + delayed writing (I/O in large chunks → improve both read and write; FIFO, low overhead, pg caching to make up perf.); swapping also provides isolation → enhance efficiency
Not every mem reference goes through OS → only pg fault
Why clustering: most minicomputers don't have separate paging device, additional I/O caused by paging incurs overhead → reduce this overhead
(a) [K Stack] Allocating K stacks in the usr address space more closely associates the resrc used by the process (mem for K stack) with the process itself. Doing so further simplifies resrc accounting of physical pgs assigned to each process. It also simplifies swapping ∵ the K stack pgs are just another set of pgs in the usr address space. Otherwise, the OS has to specifically track pgs allocated to the process in the K address space. It also slightly reduces pressure on sys TLBs (∵ usr TLBs would be used for the stack when running in the K).
(b) [Safety] It is safe ∵ VAX/VMS implements a single-threaded process model. Whenever a process executes in the K, it ignores what was on the K stack and starts using it from the bottom of the stack. So even if a process write malicious values to the K stack before executing a sys call, for instance, it cannot affect K behavior. While a process executes in the K, the K stack remains accessible to the process -- but no usr-level code in the process can execute. ∵ it is single-threaded, and it is executing in the K, it does not have the opportunity to execute usr code.
(c) [Modern OS] When an OS supports K threads, as in modern OSes, then it would be possible for a second thread to modify the state of an active K stack being used for a first thread as that first thread executes in the K.
Pg reference bit: In VMem, demand paging is a type of swapping in which pgs of data are not copied from dk to RAM until they are needed. Pg reference bit is useful for LRU / Referenced -> write / reference vs dirty.
Lazy: Zeroing pgs. An example of lazy evaluation in the VAX/VMS system is lazy zeroing of pgs. When a process requires a fresh pg of mem, than the OS maps a new pg to the process address space, pretending that the pg is available in ready, when in fact all it does is add an entry to the process pg table (PT) marked as inaccessible. When the user attempts to r/w that pg, an OS trap takes place. It is only then that the OS maps a physical mem pg to that table entry and fills it with zeros.
Use case. This optimization in cases where processes often demand more mem than is necessary, and end up with pgs they don't read. If all pgs requested by processes were actually read/written to, then lazy zeroing would be a poor choice of optimization.
Mach: Hw Portable - pmap. **PT**: Mem obj, addr map (doubly linked list). **Pg sharing**: Fork inheritance, CoW, Shadow obj, Sharing map (additional map as indirection); long-chain of shadow obj, complexity, gc.
Pg replace: global FIFO pool, on pg fault, external pgr – tuned perf, consistency in multiprocessors, extendable to distri sys, upcall, context switch.
Main idea: maintain VM state in a machine indpt module, hw PT /TLB as cache of machine-indpt info
difference with VMS → allocate any region in virtual address space, children can inherit regions (copy-on-write), mem mapped files, user-level pgr and backing store
Key data structure: resident PT (track every physical pg, used by K → fast lookup), address maps (virtual range to mem object region), mem objects (mem abstraction - backing store, decides sharing property, inheritance, protection, and pgr), pmap (machine-dpt data and code)
Mem mapped file: create a mem object, pgr is the FS, pgs always shared
Inheritance: child process inherits (shared, copied → Unix fork, or ignored) a copy of data segment of parent; copy-on-write: after a write, create a shadow object (mem object, only holds modified pgs that belonged to another object, reference original object for remainder)
Replacement policy: global FIFO pool replacing resident sets → easier to tune, can pin some pgs (device pin received data), can use external pgrs
Pg fault: fault handler in K as external pgr to make replacement decision (via upcall, msg), external pgr reads the pg from dk and hand it to fault handler, then chan MMU handler
External pgr: default pg handler is provided at user-handler; customization for app need, good for maintaining consistency, allow for expansion of VM over distr network; upcall from K (maybe no response), lots of context switch
Efficiency: vm. mgt a part of to comm. facility → send whole address space in one msg, simple mem remapping.
Lazy: CoW using shadow obj.
Shared object. When two tasks share an object in mem, instead of providing each task with a copy of the object, Mach allows both tasks to point to the same location in mem. As a long as no tasks reads this object, no other action is necessary (this is the lazy part).
Shadow object. If the pg is written to by either task, even then a fully copy of the object isn't created. Instead, a shadow object is created for that task. Shadow objects yet are another form of lazy evaluation. A shadow object is task-specific and a reference to all pgs which that task's version of the object. For all pgs which were never modified, the shadow object points to the original shared mem object as described earlier. For each pg that has been written to by the task, it points to the modified version of that pg which is specific to that task.
Pro & Con. Shadow objects are an excellent way of saving mem space when multiple tasks point to the same object for workloads where writes mostly affect a small subset of the pgs of that object. When a shared object is being intensively written to by multiple tasks and the subset of pgs written to is large, then the shadow object becomes an overhead in terms of maintenance and indirection.

VK: dkless terminal, IPC msg: sync, small msg sz, control/data msg, last ack. *replyWithSegment*, *sendWithSegment*.
OS structured around msg-based IPC for dkless workstations and large file servers (single machine IPC to distr env) → to replace special access protocols
IPC: address concern of efficiency → sync request/response msgs, small fixed size msg (32bytes, inefficient use of bandwidth), separation of control msg and data msg (WithSegement() call to improve → reduce number of msg)
easier programming, separate control and data, sync → easy buffering + smaller K, direct copy between user spaces
smaller packet → padding, waste bandwidth, sync → reduce parallelism, separate control and data (increase number of operations)
Implt: remote control implemented in K, use raw ethernet packet, stop-and-wait as reliable services, single ACK for data movement, WithSegmnt to reduce packet number
Measurement: network penalty (minimum time to transfer), remove file access adds dk I/O overhead
Streaming: network, dk read-ahead/write-behind; is necessary → large data transfer is streamed, only last packet ACKed
No dkless terminal today: privacy issue, not always online, V-K uses local network → V-K is less relevant today, IPC can be useful in K design
Increase block size of FS: Faster (dk seek less, network transfer less (one block))
Sprite: Name n location transparency, process migration, caching network FS.
Motivations: networks, large memories, multiprocessors
Distri transparent FS. **Extend Unix**: ∀ f [single uniform namespace for files and devices], shared mem, proc migration.
Distr: RPC, Flat naming with prefix table (DNS), server and client cache – consistency ver#, disable caching.
Namespace → prefix table: predecessor of cloud sys, how they find things, broadcast configuration; simple administration, efficient, FS as collection of domains, domain on different machines
Semantics of shared mem: share as possible, share heap → data all or nothing
K: support for multiprocessor, RPC (like V K, ACK once, implicit ACK, fragmentation → only retransmit missed data).
Caching: both client and server (absorb read), delayed write, consistency → seq write-sharing (versioning), concurrent write-sharing (disable cache); ensure consistency but not synchronization.
Vmem: use file as backing store → uniform naming, facilitate process migration, aggregate multiple backing files, can cache user backing store pgs in server cache; sticky segments
Process migration: pre-copying, lazy approach to copying (home node → handle machine dpt code)
Contribution: file caching protocol, dynamic balance between VM and file buffer cache (negotiation), discuss on large client and server caches
Sharing: process share mem, machines share file, share processing power via migration
Flexibility: prefix table, backing files
Perf: RPC between Ks, physical mem as cache for program and files
Dynamic cache alloc: FS and VMem in mem, but double caching. Pg backing store as original f not separate partition as Unix.
Why not proliferate: network speed << CPU speed / dk speed.
Grapevine: **Goal**: distribute msg and registration service (interdpt), mail service, naming and access control → trade-off between transparency and ease of design/implt
Msg service: deliver, buffering, accept any msg (SMTP or POP)
Registration: user mgt, authentication (password), distribution list; naming → tables mapping user, machine, service (registries); access control: ACLs in registration (NFS with file)
Scaling: adding more servers of fixed power (cost doesn't grow as a function of sys size)
Scaling issue: distribution list grow (to maintain and deliver)
Scale: problem – with distr list, sz and add/delete/update ops grow with usr population; solution – sublist, hierarchical & another level of indirection; thus fine-grained sharding unit. Availability vs. reliability. **Transparency**: issues – delayed propagation, inconsistency of changes, duplicate msg delivery, reliability of distr exposed; reliability of networks exposed. Total transparency? complete transparency is neither possible nor good, ∵ need failure feedback, design decisions on which to hide.
GMS: A cluster of machines connected by high-speed LAN (idle nodes) → use idle mem resrc, minimize average data reference time.
Why GMS: 1) feasibility: underutilization of res (idle mem) 2) benefits: growing disparity between processor speed and dk speed, and decreasing comm latency (fast network)
Pg fault: local mem - global mem - dk; **Global prob LRU**: epoch initiator node [summarize ages, send to initiator, response w_i fraction of old pgs & *MinAge*, if pg age < *MinAge*, evict pg to node i prob $\propto w_i$]. Locate pg: POD, GCD, PFD.
Hide & Expose 1) GMS provides another layer in the caching hierarchy to reduce the cost of mem operation. GMS hides the implt -- rmt mem is transparent to app and other parts of the OS.
2) rmt mem was hidden. GMS provides two operations, getpg and putpg, for satisfying rmt pg faults and performing rmt pg evictions. It uses three data structures (pdf, gcd, pod) to track the locations and ownership of pgs throughout the cluster. It uses an epoch-based distr algorithm for approximating a global LRU eviction policy.
3) When an app on an active node filled up its local mem with data and continued to access additional data, GMS would automatically evict pgs into the idle mem of rmt machines (according to global LRU). If the app touches those pgs again, GMS will transfer pgs from the rmt machine into the local mem of the app.
4) The central problem is global LRU policy. GMS uses a probabilistic distr algorithm. It divides time into epoch. At the start of eachepoch, one node collects the ages of all pgs on all other nodes in the sys. That node determines the fraction (weight) of the M oldest pgs each node contains, and broadcasts this information to all other nodes. During the epoch, when a node needs to evict a pg, it randly chooses to evict to another node with globally old pgs. This rand choice is weighted according to the distribution of old pgs across nodes. Across the full epoch, the M evictions will probabilistically achieve the global LRU goal.
5) The authors sought to hide implt from apps (and other parts of the sys). GMS did achieve these goals. apps did not need to know where their mem was being evicted to. If it was rmt mem, a subsequent pg fault was faster. If it was dk, there was no difference. GMS ensured correctness as well. If a node fails, the pgs it contains will have been written to dk as well and could always have been accessed from dk if necessary.
Dirty pg: dirty pgs must be written to dk before they can participate in the GMS pool; thus 1) Crash-resilient 2) Data consistency.
Trust: expect nodes to follow the GMS algorithm. Otherwise, for example, if a node decides to lie on its age of its pgs when it sends that information to the initiator node, it could force all evictions to occur on its own node, slowing the system excessively. A malicious node could also corrupt its global mem, making the corrupt pgs unusable for nodes in the system which need them.
Ensure trust: Encryption. Encourage to evict more pgs. Penalty to evict less pgs.
Reliability, Scale Bottleneck, Performance
Reliability: **Sprite**: ∵ of the extensive use of file caches to avoid overhead of reading from dk, and the **write-back** mechanism, file data may be lost when a mem chip failed. ∵ Sprite is caching pg data in mem, if the mem error corrupts data (e.g., no ECC on the mem) then it could lead to Sprite serving/storing corrupted file data from mem.
GMS: ∵ GMS is caching pg data in **mem**, if the mem error corrupts data then it could lead to GMS serving corrupted data from global mem (in addition to local mem). If the initiator node is down, nodes will be stuck with the stale weights, which may lead to bad paging decisions. (For example, a node whose weight is high has to pg out frequently even if it is no longer idle.) These situations do hold if the initiator fails due to mem problems, but they will be transient (until nodes timeout and select another initiator). – reselection.
Scale Bottleneck: **Sprite**: **Broadcasting**. more broadcasting requests to file servers will be sent, which may lead to more network congestion. **Multi-access**. it is more likely that multiple clients access the same file at the same time and one of the client is a writer to the file, which will have local file cache disabled and lead to more network traffic and substantially slower file access. **Global namespace**. Managing the single global namespace with the prefix table will be more expensive. More broadcasting requests may lead to more network congestion.
GMS: **Bookkeeping**. The size of the global-cache-directory (GCD) may increase ∵ more mem is available in the cluster and a node may have more global pgs. The size of the pg-ownership-directory (POD) will increase if the number of shared pgs increases. **Single Node**. The mechanism for calculating weights for nodes will be impacted the most ∵ it is done on a single initiator node. All nodes will send a summary to the node and it needs to calculate a weight for each node based on the summaries.
Performance: **Sprite**: **Network speed**: Sprite is a network system that relies on network to make RPC and access file data. **Mem sz**: Sprite uses mem to cache file data. More mem will allow the client to cache more data locally and to avoid accessing data from a remote file server.
GMS: **Network speed**: Faster network means short time to access global pgs, which can improve the performance. **Mem sz**: Larger mem mem the possibility to access the second storage (local dk or NSF), which can improve the performance. **CPU**: If CPUs are faster, there are more idle nodes, thus more global mem can be exploited.
Single master: Problem – SPF, scalability bottleneck. Solution – shadow master, min master involvement, replica.
Node failure: detect thru heartbeat, re-assignment, update metadata.
GFS: **Assumptions**: failures are norm (monitoring, error detection, fault tolerance, automatic recovery), files are huge, most files updated appending new data, co-design app API
File chunk 64MB: plain linux file (lazy allocation), global unique chunk handle, large size → less comm. with master, less metadata; small file can be hotspot (increase replica, read from other clients); with version number.
Design: mainly concurrent append (atomic), streaming/seq reads, co-design with app level. **Architecture**: Master (metadata, delegate write thru lease mechanism, gc, migrate); Chunk server (store f chunk, handle rw req, chunk replica); Client (metadata req, data req, cache metadata, r/w); no cache data ∵ usage pattern.
Goals: perf, scalability, reliability, availability (fault tolerant on cluster of commodity machine)
Single master: file and chunk namespace (prefix tree, a lookup table from full pathname to metadata, each node has read-write lock) & mapping from files to chunks → in mem, persistent in operation log (all client request, central, define operation order), location of chunk replica via heartbeat; master controls: chunk lease (pick primary, control/data separation), garbage collection (deleted file to hidden name; simple, reliable, merged into background task, when master is free; not good when space is tight), chunk collection, replica placement (reliability, availability, bandwidth use, spread across racks), creation, re-replication (favor more lost, long live, blocking; limit cloning traffic), rebalancing (to equalize dk space usage), stale replica detection (version number)
Neither client (cache metadata) nor the chunk server cache (linux caches) file.
Read: Client sends master: read (file name, chunk index)
Master's reply: chunk ID, chunk version number, locations of replicas
Client sends "closest" chunkserver w/replica: read (chunk ID, byte range); "Closest" determined by IP address on simple rack-based network topology
Chunkserver replies with data.
Write: Primary chunk, some chunkserver is primary for each chunk: Master grants lease to primary (typically for 60 sec.); leases renewed using periodic heartbeat messages between master and chunkservers
Client asks master for primary and secondary replicas for each chunk
Client sends data to replicas in daisy chain - pipelined: each replica forwards as it receives; takes advantage of full-duplex Ethernet links.
Response: All replicas acknowledge data write to client
Client sends write request to primary
Primary assigns serial number to write request, providing ordering
Primary forwards write request with same serial number to secondaries
Secondaries all reply to primary after completing write
Primary replies to client
Append: Google uses large files as queues between multiple producers and consumers. Same control flow as for writes, except…
Client pushes data to replicas of last chunk of file
Client sends request to primary

Common case: request fits in current last chunk: Primary appends data to own replica; Primary tells secondaries to do same at same byte offset in theirs; Primary replies with success to client
Consistency model: consistent, defined (not concurrent write), defined not consistent (append)
Atomic record appends: master guarantee at least once (unique record ID for filter), master chooses offset
Snapshot: copy-on-write (lease revoked thus further comm. goes via master)
Conclusion: Fault tolerance → constant monitoring & replicating crucial data (operation log, checkpoints, shadow master) & fast and automatic recovery, online repair, checksum (64kb block) to detect corruption, high aggregate throughput → separating FS control from data transfer, master involvement is minimized → large chunk & chunk lease
Limitation: single master, bad latency, not too many files or small files, relaxed consistency
Lease & Renew: Master grants lease to primary (typically for 60 sec.). Leases renewed using periodic heartbeat messages between master and chunkservers.
Version number: Chunks have version numbers. Stored on disk at master and chunkservers. Each time master grants new lease, increments version, informs all replicas.
Master Recover: checkpoints & roll-forward for master in-mem metadata.
Node recovery: notice via heartbeat, decrement chunk count, re-replicate
Limitations: Single master. Bad latency. Cannot support too many files or small files. Relaxed consistency
Big table: distr multi-level sorted map.
Main point: Distr storage sys for managing structured data, various requirements, flexible and high perf. solution, resemble database (not fully relational, client dynamic control, data as uninterpreted strings) → sparse, distr, persistent, multidimensional sorted map
no SQL → less constrained consistency → simple design, horizontal scaling, control over availability
Data model: (row, col:quantifier, timestamp)→ cell, rows [ordered, atomic], cols [two-level, ACL], timestamp [ver, lookup methods, dec sorted]. Tablet: row range, fine-grained LB. API: change metadata, manipulate data in complex way (single row transaction, cell as counter, Sawzall script). **Architecture**: Master [assign tablet, detect changes, monitor health of tablet servers thru Chubby dir, LB, gc, schema migration], Tablet server [handle rw req, re-split], Chubby [distr lock service, provide namespace with directory and small files, lock as node registration with lock lease, meta dir, locate a tablet], GFS [SSTable file] Chubby (highly-available and persistent). Tablet server registers itself by getting a lock in a specific directory chubby. Chubby gives "lease" on lock, must be renewed periodically. Server loses lock if it gets disconnected.
Metadata: Client do not rely on the master for tablet location information, thru Chubby instead
Tablet: contiguous range of column families, 100-200MB, fast recovery(each server picks 1 tablet), fine-grained load balancing
Tablet location: chubby file → root tablet → other METADATA → usertable (3 trips if empty, 6 if stale).
Tablet assignment: tabletserver holds a lock in chubby (master monitors the directory), master restart using chubby(lock, server directory, add root tablet) Tablet split change METADATA and notify master
Tablet serving: tablet log (commit) → memtable, read merge memtable and SSTable, read/write authorization from Chubby file, compaction when memtable is full (major compaction: no deleted data → timely);
Session Lease: each clients has a session with Chubby. The session expires if it is unable to renew its session lease within the lease expiration time.
Refinement: locality groups (Can group multiple column families into a locality group. Segregating columns families that are not typically accessed together enables more efficient reads.), compression (long common string, repetition in small window), 2-level caching (key value pair, SSTable blocks), bloom filters (SStable contains the data?), commit-log, tablet recovery with minor compression, exploit immutability (SStable, split share parent data)
MapReduce: **Goal**: lib to abstract parallelization, tolerate fault, distr data, LB. Programming model (restricted → easier to implement): high perf. on clusters of commodity PC.
File based, M map task partitions, R reduce task partitions [hash(key)%R].
Locality: GFS provide redundancy, master schedule task on close machine (important, network bandwidth is scarce).
Node fail tolerant: heartbeat detect; re-execute completed & in-process map() and in-process reduce(); completion committed thru master. **Master fail tolerant**: checkpoint.
Perf: file replica, backup task mechanism (to avoid last few steps take too long → execution redundancy).
Refinement: partition function (same host together), ordering guarantee of files (fast random access), combiner function (output intermediate file for other task), custom input type (reader interface), side-effects (auxiliary files), skip bad records (master notice), local execution (debug), status info, counters
File locations: a. Map input: from distr FS
b. Map output: to local FS at Map node; c. Reduce input: from multiple remote FS; uses local FS d. Reduce output: to distr FS • Local FS: Linux FS • Distr: GFS, HDFS
Evaluation: easy to use, problems expressible in MapReduce, scale well
HayStack: **Previous**: POSIX (directory and file metadata), NFS-based (dk operations are bottleneck, caching not helping due to long-tail content). **Goal**: high throughput and low latency, fault tolerant, cost-effective, simple. **Assumption**: long tail, recently uploaded most reads and likely deleted, rare modification and deletion. **HayStack**: reduce dk IO, simplified metadata as kv., single photo severing and storage layer.
Architecture: 1. H dir [in-mem mapping logical vol to phy, LB, determine cache or CDN, identify read-only]. 2. H cache [distr hash table, handle req, fetch from dk, add to cache if not from CDN and write-enabled store]. 3. H store [manage multiple phy vol on diff machines, multiple photos as a single large f to reduce meta, logical vol id+offset→photo]
H obj: H [superblock+needles, append-only (update version by offset)], H needle [read guard by cookie, delete flag], H index file [store machine uses as checkpoints for fast boosting (asynchronous update)]. Verifies cookies and data integrity.
XFS benefits: 1. the blockmaps for several contiguous large f can be small enough to be stored in main mem. 2. provides efficient file preallocation, mitigat ing fragmentation and reining in how large block maps can grow.
recovery: pitch-fork (to detect) and bulk sync (to repair)
Operation: r [(id, key, cookie) {dir} → (vol, offset) {cache/store}, seek, verify, return]; w [(vol id, key, cookie, data), sync append the needle, update in-mem map in dir] d [delete flag, reclaim thru compaction]
Optimization: compaction (copy and swap), save more in mem (offset 0 as deleted, no cookie), batch upload as possible
Evaluation: dramatically low cost, higher throughput, incrementally scalable.
Pro: Reduced disk I/O, metadata easily cached in mem. Simplified metadata – kv instead, no directory structures/file names, results in easier lookups. Single photo serving and storage layer, direct I/O path between client and storage, results in higher bandwidth

---

MicroK: **Traditionally**: problem with monolithic K: hard to extend, heavyweight, complex and huge, hard to maintain
MicroK exports and implements only basic critical resrc abstractions: only IPC (e.g. msg), VMem (addr space), Threads Scheduling. **Pro**: Multiple OS "personalities" – run Unix and NT on same machine at the same time; Specialization – specialized implt of general functionalities (e.g. different pipes); Extensibility – extend features not provided by base implt (e.g. usr-level pgr/VMem, cache partitioning).
L4Linux: **IO**: devices mapped into addr space; interrupts wrapped in msg; **Binary comp**: shared lib [Emulation layer between Linux API and Linux server] trampoline [syscalls in statically linked unmodified Linux libraries are reflected back to emulation layer]. **Tagged TLB**: problem – TLB flush when context switch, ∵ cache is local for proc, when proc changes, cache should be invalidated; solution – Tag proc id on TLBs to prevent flush.
ExoK: **Design**: LibOS [∀usr-level, OS functionality, untrusted]. K no param check; thus faster. Malicious behavior only affects the single app itself that using libOS rather than entire machine. No share: ExoK hard to share FS, each app has own implt of FS and lib call; thus hard to share. No protection: FS mem errors by app can potentially corrupt on-dk data structures. **ExoK**: multiplex & protection; resrcs multiplexing [track res ownership, multiplex hw via allocation and revocation], protection – secure binding [hw: protected data sharing via VMem hw. Caching: sw TLB cache on top of hw TLB. Downloaded code: e.g. packet filters, ASHs]. **ASH Packet filters**: ASH (
App-specific Safe Handlers) is downloaded into K, to do packet filter. A packet comes in, ExoK analyzes it, and sends it out to the right App. This is more efficient than the ExoK querying all the server processes on every incoming packet to determine who the packet was for.
FS in ExoK: apps know better than OS the goal of resrc mgt thus apps make decisions. FS done in LibOS inside app's address space. safety concern (ASH), limited sharing (app development is not, ∵ OS interface the same)
Hide & Expose 1) ExoK exposes the implt. expose the mgt of hw resrcs to apps (LibOSes) to maximize their perf.
2) ExoK exposed all hw resrcs to usr-level processes. ExoK supported a streamlined sys call interface by which apps could request resrcs (mem) and express mgt policy (how to map virtual pgs to physical pgs). It also supported callbacks to usr-level to delegate mgt decisions (pg faults) and to allow processes to express preferences for revocation decisions.
3) One concrete example is virtual mem. ExoK controlled which processes were allocated which physical pgs, but those processes determined which of their virtual pgs mapped to the physical pgs that they were given. On a pg fault, for example, ExoK would invoke a callback to usr-level asking which physical pg to map to the undefined virtual pg mapping (TLB translation). The usr-level process would then be able to specify any of its physical pgs to map (i.e., it would decide its eviction policy). ExoK would validate the mapping (i.e., ensure the process owned the physical pg) and then install the TLB entry to provide the mapping.
4) Problem1 that ExoK experienced is that calling back to the usr-level process on each pg fault was expensive (crossing the protection boundary, deciding which pg to evict). ExoK solved this problem by implementing a software TLB -- a cache of translations provided by usr-level processes. This cache dramatically reduced the number of usr-level callbacks. Problem2 was demultiplexing (dispatching) incoming network packets to their respective usr-level processes. Upon receipt of a packet, ExoK would not know which process to deliver the packet to (e.g., without asking the processes). ExoK solved this problem by supporting packet filters that expressed usr-level intent (this packet is for me) while running with low overhead inside the K. Problem3 was resrc revocation. When ExoK needed to remove physical pgs from a process (e.g., to provide them to a new process starting up), it would ask processes which physical pgs they wanted to release (delegating mgt policy to usr level). Processes could not be fully trusted, however, and ExoK still needed to provide a mechanism to take pgs from uncooperative processes
5) The authors sought to expose mgt of hw as much as possible to usr-level processes while minimizing functionality inside the K. They achieved this goal for the most part. Usr-level processes had far more control over hw than other syss (even microK structures). The purity of the approach was somewhat ruffled, however, by having to introduce support for perf optimizations inside the K (software TLB, packet filters, ASTs).
¬ **Lazy**: **Secure binding**. An example of eager evaluation in the ExoK operating system is the secure bindings feature. ExoK allows library OSes to manage resources, while the K is responsible for handling protection between mutually untrusted apps. **Protection** is an expensive operation, but once an app has been **authorized** to use a given resource, **access checks are no longer required**. Therefore, the secure bindings separates binding a resource to an app from accessing that resource :
By isolating the need to understand these semantics to bind time, the K can efficiently implement access checks at access time without understanding them.
Use case. This type of operation is best when an app continuously uses a resource it has requested authorization for. If an app decides to **continuously ask** for different resources, more binding checks will be required which removes the benefit of the secure bindings.

VM370: **CP**: Control Program, computing machine to simulate multiple copies of the machine. CP is a general multiprogramming sys that uses virtual machines to organize indpt job streams. Each virtual machine is identified by an entry in the CP directory. interface to virtual machine, allocation and mgt of rest of machine **CMS**: Conversational Monitor Sys, Interactive use of sys by one person. CMS is a dk-file-oriented OS to support the personal use of a dedicated computer. data sharing (safe) is enabled via links made by CP (shared mem of IO channel) **RSCS**: Remote Spooling and Comms Subsys, Information transfer among machines linked with comm facilities. RSCS is an interrupt-driven, multi-tasking sys that uses a dedicated computer with attached comms equipment for the support of data file transfer among computers and remote work stations.
Xen: **Paravirtualization**: (VM abstraction that is similar but not identical to the underlying hw, modify the OS to make it VM-aware and perf tuned), VMMonitor exposes hw interface [runs on raw hw vs. runs on host OS]. **Usage**: OS dev & debugging, sw testing, utilize hw, isolation, virtual infra for IoT [scalability, security & isolation, developer control, low cost].
Explicit schedule among guest OSes; devices are virtual by VMM;
Fast handler (hypervisor not involved): pg fault and sys call; can register handler with VMM.
Privilege instruction: VMM tests usr/K mode, emulates VM's Syscall [Privileged instruction → Interrupt (trap) → Trap handler → Emulate], Protection doms ring 0, 1, 2, 3. – Xen modifies the guestOS to run on ring 1.
Hypercall: extend machine interface with hypervisor calls. Control transfer: hypercall, event (asynchronous notification from Xen to domains); Xen in top of every address space.
TLB flush: 2 level PTE (difficulty in x86, hw manages PT, software not involved in TLB miss, TLB not tagged). Problem – x86 does not have a software managed TLB. Its TLB is not tagged, which means that the TLB must be flushed on a context switch. Solution – a. guest OSes are responsible for allocating and managing the hw PT, with minimal involvement from Xen to ensure safety and isolation; b. Xen exists in a section at the top of every address space, thus avoiding a TLB flush of Xen when entering and leaving the hypervisor.
Mem: guest OS allocates mem pg, registers with Xen, call Hypersor for write req, validated by Xen, guest batches the update requests.
Hypervisor Mgr: Dom0.
Lazy: An example of lazy evaluation in the Xen operating system is GuestOS **"fast" exception handler registration**. Xen operates on the x86 architecture on ring 0 (highest instruction privilege). Guest OSes which Xen virtualizes operate on ring 1, but their code has been written to function on the highest privilege. As such, Xen has to simulate some registers which are only available on ring 0 to handle exceptions such as segmentation faults and allow Guest OSes to register exception handlers with the hw.
Xen tries to make the exception registration process as fast as possible. Therefore, exception handling validation does not do safety checks such as verifying that the handler code is present in mem (which could be expensive). As such, it could be the case that a guestOS invokes an exception handler which is not in mem, this is called a "**double fault**" (the program supposed to handle the exception faults as well). In such cases, Xen is capable of fetching the appropriate handler for the guestOS.
This kind of optimization can hardly go wrong with OSes the authors expect to handle, but one could imagine a malicious OS which would register an unusually high number of exception handlers which would cause double faults to occur much too often.
ExoK, MicroK, Xen. 1) [Protection dom] ExoK: The protection doms are the exoK and the Web server process. LibOS does not form a protection dom (it is a part of the Web server process). Only the exoK is privileged. L4: The protection doms are the L4 microK, the L4Linux server, and the Web server process. Only L4 is privileged. Xen: The protection doms are the Xen hypervisor, Dom 0, and a Dom to run the Web server which has two protection doms, the Linux OS and the Web server process. Only Xen is privileged; Dom 0 is _trusted_, but not privileged. 2) [web packet] ExoK: The incoming packet steps are exoK → [upcall] → Web server process. L4: The incoming packet steps are L4 → [IPC] → Linux → [IPC] → L4 → [IPC] → server process. Xen (orig): Xen → [virtual interrupt/event channel] → Linux DomU → [read syscall] → server process. 3) [perf] ExoK is likely to perform the best. There is only one dom crossing, packet data does not have to be copied, and the networking stack can be customized to the needs of a Web server. 4) [Pg Fault] ExoK: The CPU raises an interrupt, halts the Web server, vectors to the exoK. The exoK performs an upcall to the Web server process, which decides which of its physical pgs to map to the faulting virtual pg. The Web server process returns to the exoK with the chosen virtual → physical mapping, the exoK installs the mapping in the TLB and restarts the Web server process.
- The LibOS in the Web server process "handles" the pg fault ∵ it decides the mapping. It uses its pool of physical mem given it by the exoK.
L4: The CPU raises an interrupt, halts the Web server, vectors to the L4 microK. L4 sends an IPCto L4Linux, which acts as a pgr for the address space of the Web server process. The L4Linux pg fault handler allocates a physical pg from the physical pgs that L4Linux manages, updates the PTE for the Web server address space in its shadow PTs. L4Linux returns from the IPC from L4, and L4 updates the true hw PTs to reflect the changes L4Linux made to the shadow PTs. L4 can then insert the true PTE into the TLB and resume the Web server process.
- L4Linux "handles" the pg fault ∵ it decides the mapping. It uses the pool of physical mem delegated to it by L4.
Xen: The CPU raises an interrupt, halts the Web server, vectors to the Xen hypervisor. The Xen hypervisor upcalls to the OS in the DomU running the Web server process. The DomU OS pg fault handler allocates a physical pg from the physical pgs that it manages (as provided by Xen when the dom was created), updates the PTE for the Web server address space in its PTs, and performs a hypervisor call (syscall) back into Xen with the PTE with the new virtual → physical mapping. Xen then installs this PTE into the TLB and restarts the Web server process.
- The DomU OS "handles" the pg fault ∵ it decides the mapping. It uses the pool of physical mem delegated to it by Xen.

Cells: supports multiple VPs on one phy phone. Main points: lightweight virtualization architecture for smartphones. Foreground-background model allows for low overhead virtualization for resrc constrained phone platform. Supports shared device functionality with minimal modification to usr and K space.
Design: Virtualize OS interface, multiplex device namespace. **Usage model**: Foreground and background VPs. **Device access**: access restriction (exclusive, shared, no), foreground full access (direct hw access, native machine perf), virtualization (device driver wrapper, isolated device namespaces, device subsys namespace aware). **Scalability**: Shared read-only FS saves resrcs between VPs. Union each VPs writeable files with read only FS. Low mem killer- destroys background processes, keeps solution lightweight.
Device namespace (K level abstraction, transparency and perf.): mux_fb (for framebuffer, foreground exclusive, background VP writes to backing buffer), GPU (accelerated graphics), power mgt (bearly suspend with namespace aware, wake lock)
Cells proxy libraries (user-level, portability and transparency): closed and proprietary device infrastructure (telephony radio stack) → VP RilD and Cells RIL.
VP has complete use of standard cellular phone network: VoIP cloud services Novel FS layout based on unioning (and KSM find same pg, kill app → scalability)
Virtualization methods: device driver wrapper, device subsys aware of device namespace, device driver aware of device namespace
Root namespace(user level): minimal init env (CellD: start and switch VP, RIL: telephony, NAT: network)
modest time and mem overhead, apps run at native speed
TaintDroid: **Main**: Sys wide, dynamic, taint tracking and analyzing, simultaneously tracking multiple sensitive data → try to identify an appropriate trade-off between perf and accuracy.
Taint source, data-flow track, taint sink. Tracking: variable-level, method-level (native method, interface exposed), msg-level (binder IPC), file-level. **Variable-level**: Modified the Dalvik VM interpreter to store and propagate taint tags (a taint bit-vector) on variables. **Method-level**: Rules: 1) all accessed external variables (e.g. class fields) are assigned taint tags. 2) the return values is assigned a taint tag. JNI Methods: JNI methods are invoked through the JNI call bridge. TaintDroid uses a combination of heuristics and method profiles to patch VM tracking state. **Msg-level**: IPC, TaintDroid uses msg-level taint tracking. A msg taint tag represents the upper bound of taint markings assigned to variables contained in the msg. (coarse-granularity, increase FP).
File-level: Persistent storage tracked at the file level. The design stores one taint tag per file. The taint tag is updated on file write and propagated to data on file read. (coarse-granularity, increase FP). **Privacy hook**: taint source, Low-bandwidth Sensors – place hooks in Android's LocationManager and SensorManager apps. High-bandwidth Sensors – place hooks for both data buffer and file tainting for tracking microphone and camera information. Information Databases – address book, SMS storage. Device Identifiers – the phone number, SIM card identifiers (IMSI, ICC-ID), and device identifier (IMEI). Network Taint Sink – identifies when tainted information transmits out the network interface.

Scheduler Activations: **Usr vs. K threads**: perf on context switch, flexible customized, Poor integration: K threads block, resume and are preempted without notification to the usr level; K threads scheduled oblivious wrt the usr-level state; when usr thread blocks, K thread also blocks. **Solution**: - a new K interface (straightforward modification) and user-level thread package (programmer sees no difference) → functionality, perf., flexibility - Share necessary control and scheduling information between K and app address space
Scheduler activation: 1) Vessel for executing a usr thread context. 2) Mechanism for notifying usr-level of K events (upcall, downcall). 3) Data structure for storing usr thread state in the K. **Virtual Processor**: a. Each process supplied with a virtual multiprocessor; b. K allocates processors to address spaces; c. Usr level threads sys (ULTS) has complete control over scheduling; d. K *upcalls* ULTS whenever it changes the number of processors, or a usr thread blocks or unblocks; e. ULTS *downcalls* K when app needs more or fewer process. Both downcalls and upcalls are just hints.
To notify an event: K create a new scheduler activation, assign processor to it, upcall
An activation blocked: one upcall for block, one upcall for ready, user-level scheduler decides
The stopped thread is never directly resumed by the K → potential deadlock (preempted thread holds the ready queue lock)
Example: 1. IO blocks thread in K; K upcall in the context of a fresh SA; UL scheduler schedules another thread. 2. IO completes. K preempts processor, takes the preempted SA to form a new fresh SA as the context of upcall.
Critical section: Problem - Deadlock (blocked thread put in ready list has ready list lock, scheduling is managed in usr-level rather than K-level, thus usr cannot acquire the lock on the ready list, resulting deadlock). Solution – on thread preemption or unblock, check if in critical section, continue run till end of critical section.
Usr-level scheduling. SA have to worry about this deadlock situation ∵ a thread can be interrupted, suspended, and preempted at any time during its execution. And the interruption and suspension are done within the K, yet scheduling it again has to be done at usr level.
Standard K threads do not require this kind of cooperation between the K and usr level. All thread operations are implemented and complete their execution while within the K.
(Optional) what about usr level threads running on K threads? It is the case that the K decides when a K thread (and hence the usr-level thread running on it) gets scheduled and gets suspended, e.g., when its time slice is up. But eventually that K thread will execute again, effectively resuming the usr-level thread again. And that usr-level thread can only be descheduled -- context switched to another usr-level thread -- while the K thread is executing.
Lottery Scheduling: **Main point**: modular resrc mgt, express relative rates, probabilistic mechanism/fair, efficient (implt), flexible and responsive control. Prob based. **Ticket**: priority is thread's own tickets relative percentage of all the tickets whose owners compete for a particular resrc. Scheduler picks the wining ticket randly and grants the resrc.
flexibility through transfer, inflation, currencies, compensation.
transfer → to handle priority inversion problems of client/server model
inflation → to manage resrc allocation
currencies → to hierarchical allocation
compensation → to compensate threads that relinquish CPU due to IO etc.
Implt: random numbers, sort threads by decreasing order of # tickets (or tree)
Generalized to diverse resrc: synchronizing(transfer ticket to lock owner), space-shared resrc(mem, inverse lottery), multiple resrcs(manager)
Issue: 1) Cannot express response time differently from share. 2) Cannot instant wake up.

FFS: **Cylinder groups**: a cylinder group is comprised of one or more consecutive cylinders on a dk. **Optimize storage utilization**: Increase block size; Introduce fragments. **Layout policy**: global policy to select cylinder group, local policy to select data block, to localize related data and scatter unrelated. **Hw parameterized FS**.
Structure:
Partitions → FS → Directory/inode → file.
Dk → Cylinder Group → Cylinder → Block → fragment.
Layout policy: related data locality. Inodes data locality. Larger block size: high bw utilization, larger max f sz. Fragments within block: reduce wasted space. Replicated superblock: recovery. Hw param: device aware.
Throughput rate: is tied to the amount of free space maintained, utilization rate μ; ∵ if free space is low, blocks have to be scattered across many cyl groups, not localized. **Write**: In old FS writes are async; thus host can reorder requests in cache to min total seek time on flushing. In FFS blocks are placed on dk in a much better layout according to layout policy; writes have higher OH ∵ of OH of allocating new blocks.
Obsolete: device parameter (cylinder diff radius, host doesn't know cylinder and track boundary), rotation allocation (blocks allocated seq, do read ahead in seq cmd)
Current UFS (and NT), extent-based (allocate consecutive blocks, r/w more blocks a time) with journaling
Perf. In the old FS, over time file blocks for the same file become scattered randly across the dk. As a result, reading the same-sized file will take longer on an aged FS ∵ therewill be long seeks between accessing each file block. The new FFS largely avoids long seeks between blocks ∵ it tries to achieve seq/rotationally optimal placement (best case), or at least it can place blocks for the same file in the same cylinder group (resulting in head switches and short seeks). The ability for the new FFS to achieve "optimal" placement depends upon the flexibility of finding free blocks that provide the optimal placement. The more free space in the FS → the greater the flexibility (better the chance) that optimal free blocks can be found → the better the transfer rates.
LFS: **Assumption**: dk latency is bottleneck, large main mem (absorb reads), workloads (small file access); flash mem → random access, **LFS for Flash mem**: limited number of erases can be performed; LFS write seqly (append) than in-place, amortizing the wearing evenly. **Log**: ∀ info written to dk is appended to log: data, inodes, direntry, imap (opp in-place write). **Write-back cache**. **GC**: main perf OH drawback, inter-segment threads (frag) + within-segment copy & compact (copy still append, OH). Segment > Block. **File locating**: file# ↦ inode location ↦ inode ↦ data (opp. direntry ↦ inode ↦ data). **Recovery**: checkpoints & roll-forward (write all modified info to the log, checkpoint region → a fixed position on dk), roll-forward (redo) **LFS vs. FFS rw**: LFS < FFS on rand w and seq r – ∵ FFS organizes file blocks structurally seqly, no matter how written; LFS organizes file blocks in order w (append to log).
Sprite: LFS dramatically increases the perf of file **writes** at the expense of **read** perf. Under Sprite, this tradeoff is particularly useful ∵ most read access was **cached** anyway—that is, Sprite syss would typically perform fewer reads than a normal Unix sys. LFS-like syss also allow for much easier crash recovery, which became a major focus of the project during this period.
Eager write. If you assume that the eager dk will seek to the next extent that fits the entire LFS segment on awrite, then eager writing will make a little, but not substantial difference. When writing LFS segments, the dominant cost is transfer time rather than seek time.
Assumption & impliciation. **Assumption**: 1) Dk - faster sequential write 2) Large RAM, cache read 3) Large RAM buffer write 4) Hot f cold f access pattern.
Implication of assumptions: **Write mechanism** - sequential write. The writing mechanism for LFS is behind the fundamental idea of a log structured file system. Given the hard dk assumption, it is inconvenient to write to random locations on the file system. Instead, LFS chooses to write all of the incoming changes sequentially, even if they correspond to unrelated files. Moreover, all of the changes are done in one write, which is possible given the assumed large size of the write buffer.
Read mechanism - inode, imap. The reading mechanism for LFS is more similar to that of traditional file systems (such as FFS). An inode data structure contains pointers to the file's dk blocks. However, inodes need to be retrieved somehow, ∵ that how no longer stored on a fix location (as in FFS), but written like everything else at the end of the log. Therefore, an additional layer of indirection (given by an inode map) is required. This makes reads slightly slower or just as good as FFS, which is not such a big problem given the workload isn't expected to be read heavy.
Clean mechanism - The cleaning mechanism for LFS is required to guarantee free space to be constantly available in the dk and to avoid fragmentation of the log. This is done by loading into mem (fragmented) segments of data and write them (sequentially) back to dk. The read segments are now marked as free. This is made possible by the large file cache assumption, allowing to load the segments in mem. The authors thread very carefully with the loaded segments ∵ some of the data could be "live" (about to be edited). Given the hot/cold assumption, by focus on moving around cold files, the problem of having updates on segments being cleaned happens less often.
Non-volatile RAM: The seek time for data in NVRAM is no longer limited by the movement of a mechanical arm. This makes random accesses much faster and reading sequentially less important. The LFS gets its write performance from the fact that all of its writes are sequential. LFS's write performance dominance will be crippled ∵ its performance will not increase as much as that of other file systems such as FFS. The drawback of having to clean segments, however, will remain. As a result, the design is unlikely to become attractive in such an environment. **Utilize NVRAM**: Managing metadata in LFS is currently cumbersome. An extra layer of indirection (the inode map) is required. Whenever a file is updated, this map must be update as well. The inode is not appended to the log like other writes, ∵ it must be easy to find. Having the inode map located in NVRAM would benefit both read and write performance. Another use of the NVRAM could be read caching.
Reading data is where LFS does not perform well, an extra layer of caching could only be beneficial to the overall performance.
Soft updates: **Reliability**: can waste some inode or blocks, but data pointed cannot be corrupted or uninitialized.
Soft updates: write-back caching of metadata, avoid reliability-induced write → perf (metadata write overhead), integrity (handle possible corruption), fast recovery (instant mounting, ∵ no log replay) → app view is up-to-date (in mem), dk view is consistent (dependency is resolved). **Operands**: direntry, inode. **Order dependencies**: 1. Never point to a structure before it has been initialized (e.g., an inode must be initialized before a directory entry references it) 2. Never reuse a resrc before nullifying all previous pointers to it (e.g., an inode's pointer to a data block must be nullified before that dk block may be reallocated for a new inode). 3. Never reset the last pointer to a live resrc before a new pointer has been set (e.g., when renaming a file, do not remove the old name for an inode until after the new name has been written). To leave the FS in a consistent state [¬ point to unintended inode, okay for dangling inodes].
Undo/redo operations: soft updates tracks dependencies on a per-pointer basis and allows blocks to be written in any order. To solve circular dependency, any still-dpt updates in a metadata block are rolled-back before the block is written and rolled-forward afterward.
Recovery: fsck (storage leak) to reclaim unreferenced resrc and correct link counts
Semantic: dk data can be trusted, no more data can be lost in crash
Rio: Write-back perf with write-through reliability. **Reliability**: Rio protects file cache mem thru disabling write-permission bit in PT && modifying to ∀ access thru TLB during normal operation. Restores file cache contents on warm reboot thru registry. **Warm reboot**: power is not lost, RAM not reset. **Write-protection**: 1) mapping all addresses go through TLB, write-permission bit disabled 2) binary rewriting (code patching, slower), check every instruction it doesn't modify buffer cache. **Registry**: keep track of cache pgs, used in reboot, stored in protected, recovered area → on reboot, K vm dumped to swap file, a user-level process does recovery.
Effect on FS: 1. Eliminate reliability-induced writes; 2. Metadata updates need to be ordered in mem; 3. Easy to do atomic updates by shadowing copy.
Corruption: direct-corruption thru sys non-IO proc, indirection-corruption thru sys-IO with wrong param. **Detect corruption**: checksum, memTest. checksum mem pgs, update checksum on every write(only direct corruption, no I/O, works only in mem), memTest (direct & indirect corruption, has I/O call, repeatable series of file/dir ops, go back and check mem) → app level check.
Protection: Rio is useful in: protection domain separated from address space, buffer cache separated into protected module
Reliability. 1) [dk access] Assuming no data block allocation is needed for the directory or new file, the three dk operations are a write to the inode freelist, the inode dk block, and the directory data block.
2) Unix FFS will issue all metadata and directory block writes synchronously, although it may not protect the inode freelist (as one can rebuild these with a recovery utility like fsck). Therefore, at least the inode block and directory block writes will be issued synchronously. The reason you do synchronous writes for metadata (and directory data) is that you want to leave the FS structure consistent if a failure occurs between any two operations. In this particular case, there are two dependencies. The inode freelist must be written before the directory data block is written (if the inode freelist is being protected) and the inode dk block must be written before the directory data block is written. Hence, if the sys crashes between two write operations, the worst that will happen is that the inode will be allocated but unlinked, or initialized but unlinked, respectively. Note that the last write in such a sequence may be issued asynchronously, depending on the implt.
3) Soft Updates caches all metadata updates for write-back at some later time. Their method for making this approach safe is, at a high-level, by keeping a dependency graph for all the dk blocks and only writing back in a "safe" order. The difficulty with using this technique is that, ∵ a single data block can have multiple units of metadata, there can be cyclic dependencies that result from false sharing between two dk blocks. Their solution for breaking these cycles is to track dependencies at the level of a unit of metadata, rather than a dk block (note that the dependency graph over metadata is a DAG). By consulting this graph, they can build an ordering among dk blocks such that the sys remains in a consistent state by allowing a block to appear multiple times with some previous changes undone or redone.
4) LFS uses a cache which writes for reliability after a usr-identified duration. As a result, no writes are issued synchronously. However, this cache writes to a journal, effectively relocating data and metadata blocks each time a write is issued. That is, although the cache is write-back, it does not write each block in the cache to a pre-allocated location on dk, but instead dumps the cache into a contiguous block of free space on dk. The reason why this is still clean from a reliability perspective is that if we guarantee that the sys is consistent at the end of every write-back, and we only write to the bookkeeping section that tells us where the end of the log is when we finish a write-back, we can treat all updates done in a single journal write as atomic, ∵ writing to the journal does not erase anything (if anything was removed, say an old log segment was cleaned, we can modify that process to maintain global consistency). So, if the log write fails we always have a consistent view to fall back on.
5) Rio treats mem as non-volatile (persistent as in dk), supported by a battery backup, or some other kind of persistent (but fast) mem device. The consequence of this assumption is that mem is not so much a cache or buffer as it is a functional equivalent to dk. Then, writes are done synchronously to mem (as always) and write-back is only done if the buffer fills. In the case of power being removed from the machine, the data is assumed to be more or less retained (there's some mem hacker that tends to need to happen to keep inconvenient writes from corrupting the buffer cache), so as long as the writes are issued in the correct order the data will be consistent. Note that Soft Updates only needs to be aware of the dependency graph, and can otherwise receive writes in any order.
a. Read-dominated workload (100x reads per write)
b. Latency improves by 10x, bandwidth improves by 10x ("SSD")
c. Latency degrades by 10x, bandwidth degrades by 10x ("Internet Cloud storage")
FFS: Previous system: old unix FS. The major optimizations: bigger blocks; arrange related data closely. Major improvements: faster reads
a.Since the optimization basically accelerates reads by sacrificing the performance of writes a little bit, a read-dominated workload will make FFS outperform the old UNIX file system even more significantly.
b. Interpretation 1; Since the underlying storage device is an SSD, and an SSD does not have much of a seek penalty (composition changes), then the optimization of arranging related data close together will not benefit as much. Although the larger blocks optimization will still help FFS, FFS probably will not outperform the old UNIX file system as much as before.
Interpretation 2: Composition. If latency and bandwidth both improves by 10x, then the bottleneck of the dk performance will still be latency. Since under this interpretation the composition of latency does not change, the optimizations of FFS will still be as helpful as before, thus FFS should still outperform the old UNIX file system significantly. (Amdahl's law)
c. Interpretation 1: If latency and bandwidth both degrades by 10x, then the dk performance tends to be limited by latency. However, the optimization of arranging related data closely improve a small part of this latency (mainly seek time) while the majority of the latency that comes from network (bottleneck, composition changes) communication does not change. Although the larger blocks optimization will still help FFS, FFS probably will not outperform the old UNIX file system as much as before.
Interpretation 2: composition of latency does not change (c.f.)
LFS: Previous system: FFS
The major optimizations: turning random writes into sequential writes; aggressive caching of both data and metadata.
Major improvements: fewer writes, faster small writes (especially small-file writes); higher write throughput.
LFS transforms random writes → sequential writes.
Write buffer caching
a. If the workload is read-dominated, then the optimization that mainly improves writes will not be very helpful. However caching will make reads much faster than in FFS, therefore LFS will still outperform FFS significantly.
b. Interpretation 1 seek: Since the underlying storage device is an SSD, and an SSD does not have much of a seek penalty, then the optimization of turning random writes into sequential writes will not be very helpful. However, aggressive caching of both data and metadata will still help avoid many writes, which will probably help LFS outperform FFS significantly.
Interpretation 2 bandwidth: composition of latency does not change (c.f.)
c. Interpretation 1: In this scenario, the majority of the latency comes from network communication rather than seek time, so turning random writes into sequential writes is not as helpful as before.
On the other hand, latency is even worse, so the aggressive caching of data and metadata will avoid many synchronous writes which FFS will suffer from, and the large segment writes will both reduce the number of writes overall and still greatly outperform the small writes that FFS does. Therefore, LFS is still going to do significantly well relative to FFS in this scenario.
Interpretation 2: composition of latency does not change (c.f.)
Soft updates: Previous system: Systems that update metadata synchronously
The major optimizations: caching metadata updates with respect to the dependencies among them.
Major improvements: fewer updates to metadata; better performance; better availability.
a. Although a read also updates metadata (the last access field in a inode), there will be much fewer metadata updates in this scenario, thus the performance improvement will be less significant.
b.In this scenario, grouping (write-back cache) multiple updates together or even avoiding updates completely will still be very helpful, thus soft updates will still outperform previous systems significantly.
c. In this scenario, avoiding multiples updates to a single metadata component (e.g. delete a recently created file) (write-back cache) will be more beneficial since multiple expensive remote dk accesses can be avoided, thus the performance improvement should hold (maybe even more significant).