I. EXOKERNEL

A. problems of traditional OS abstraction

1) denies domain-specific optimizations.
2) discourages changes to implementations of existing abstractions.
3) restricts the flexibility.

(general-purpose implementations of abstractions force applications that do not need a given feature to pay substantial overhead costs.)

B. microkernel

benefits: easier to extend a microkernel; easier to port the operating system to new architectures; more reliable (less code is running in kernel mode); more secure. *detriments*: performance overhead of user space to kernel space communication

C. comparing with microkernel

- m do not allow untrusted application software to define specialized IPC primitives because virtual memory and message passing services are implemented by the kernel and trusted servers.
- many abstractions, such as page-table structures and process abstractions, cannot be modified in microkernels.
- many of the hardware resources in microkernel systems, such as the network, screen, and disk, are encapsulated in heavyweight servers that cannot be bypassed or tailored to application-specific needs.

D. methods

secure binding, visible revocation, abort protocol

why not virtual machine? severe performance penalty

E. secure binding

decouples authorization from the actual use of a resource (seperate protection from management)

1) implementations: 1, hardware mechanisims; 2, software caching; 3, downloading application code. (*benefits of downloading code*: 1, eliminate kernel crossings; 2, execution time of downloaded code can be readily bounded (no need to be scheduled))

II. FLASHVM

A. problems with flash

- write amplification (rewrite multiple blocks). *solution*: finer granularity write-back, stride prefetching, etc.
- low reliability (a finite number of times to be written). *solution*: page sampling, zero page sharing.
- aging (fewer clean blocks). *solution*: merged discard, dummy discard.

B. performance

write locality because random reads are inexpensive; more aggressive precleaning; clustering; much finer granularity page scanning; prefetch a full set of valid pages; stride prefetching for temporal locality.

III. NON-SCALABLE LOCK

A. model

- a: avg lock acq time on single core.
- $a_k = (n-k)/a$: arrival rate.
- s: time in serial section.
- c: time for home dir to respond to a cache line req.
- $s_k = 1/(s + ck/2)$: service rate.

If a large number of waiters, hard to go back. s_k rapidly decays as k grows, for short serial section. (ans to why so rapidly)

IV. LEARNING FROM MISTAKES

A. bug pattern

1, Dead lock. 2, Atomicity violation. 3, Order violation. 4, Others.

B. fixing strategies

- non-deadlock: condition check(while-flag; consistency check); code switch; design change; lock (add/change; adjust cric sec regions); others.
- deadlock: give up resource; split resource; change acq order.

C. lockset algorithm

1) Let $\text{locks}_{\text{held}}(t)$ be the set of locks held by thread t.
2) For each v, initialize $C(v)$ to the set of all locks
3) On each access to v by thread t, set $C(v) := C(v)/caplocks_held(t)$. if $C(v) = /emptyset$, then issue a warning

pros: more efficient way to detect data race. predict data race that have not manifest. *cons*: exists report false positive (e.g. memory reuse), lLimits the synchronization method to lock.

V. A FILE IS NOT A FILE

A. findings

- **a file is not a file**: file -> small FS containing subfiles.
- **sequential access is not sequantial**: *pure* is rare (substantial according to 4.2.2). more 95% sequential.
- **auxiliary files dominate**: helper files
- **writes are often forced**: most explicitly synced (some frequently).
- **renaming is popular**: atomic operations are common, generally *rename*.
- **multiple threads perform I/O**: virtually all from a number of threads (to hide long-latency operations from interactive users).
- **frameworks influence I/O**: the behavior of the framework, not just the application, determines I/O patterns.
- wide variety of file types, mostly multimedia files.
- apps tend to open many very small files, while most of the bytes accessed are in large files.
- preallocation is used rarely, or in useless way...

VI. LFS

A. structures

1) inode map: Current location of each inode. Blocks are written to log; addresses of blocks in checkpoint region. Almost always cached in main memory.
2) segment usage table: 1, the number of live bytes in the seg. 2, most recent modified time of any block in the seg. Used by cleaner. Blocks are written to log, addresses of blocks in checkpoint region.
3) checkpoints: Special fixed position on disk. Addresses of all the blocks in inode map, seg usage table, current time, pointer to last seg written. Two checkpoint regions, operations alternate between them. Time: perriodically, when FS unmounted, system shut down.
4) directory operation log: Operation code, location of dir entry (inum and pos within dir), contents (name and inum), new ref count. In log, before corresponding dir block or inode.

VII. FRANGIPANI

A. logging and recovery

- write-ahead redo logging for metadata; user data is not logged.
- only after a log record is written to Petal does the server modify the actual metadata.
- a write lock that covers dirty data can change owners only after the data has been written back to Petal.
- version number for each log.
- no guarantee that FS state is consistent after a failure.

B. synchronization

- multiple-reader/single-writer lock.
- write lock holder must write dirty data to disk before releasing or downgrading.
- on-disk structures -> logic segments; locks for each segment.
- per-file lock granularity.
- ordering locks and acquiring in two phases.

VIII. DEVICE DRIVER

A. redundancy

many opportunities. *methods*: 1. procedural abstractions; 2. better multiple chipset support; 3. table driven programming.

IX. X86 VIRTUALIZATION

A. shadow page table

gVAs -> gPAs -> hPAs. shadow page table stores the composite mappings. *problems*:

- hidden page faults on first access. *solution*: eager validate.
- context switching flush the TLB. *solution*: make copies, traces (provide notifications upon access to pages of interest).
- traces overheads. *solution*: not tracing.

must balance among these three demands. but it is difficult and varies from workload to workload.

B. address space

methods:

1) page permission. *pros*: works well with trap-and-emulate; *cons*: not well for running translated guest kernel code.
2) bounds check on every guest memory access. *cons*: significant overhead.
3) segmentation.

problem of segmentation: must also emulate the guest's use of the same segmentation functionality. *solution*: place the VMM at top of the address space so that flat segments can be precisely "truncated" to prevent access to the VMM while allowing access to all remaining virtual addresses.

C. virtualizing 64 bit x86

problems:

1) all segment registers but *%fs* and *%gs* were flattened: limit checks were removed -> could no longer use segmentation.
2) the *lahf* and *sahf* instructions were removed from long mode (faster way to save and restore flags).

D. IS virtualization with hardware support: VT-x and AMD-V

- virtual machine control block (VMCB): an in-memory data structure.
- guest mode: a new, less priviledged execution mode.

vmrun -> do something -> exit. *optimizations*: buffered exit.

E. memory virtualization with hardware support: RVI and EPT

- nested page table: VMM, gPAs -> hPAs.
- in guest mode, TLB: gVAs -> hPAs.

pros: no exits. *cons*: cost of a TLB miss will be higher with nested page

X. DYNAMO

A. partitioning algorithm

consistent hashing: the output range of a hash function is treated as a fixed circular space or "ring". each node in the system is assigned a random value within this space which represents its "position" on the ring. *problems*: 1, non-uniform; 2, oblivious to heterogeneity. *solution*: vritual nodes.

B. replication

each data item is replicated at N hosts. each key k is assigned to a coordinator node. **preference list**: the list of nodes that is responsible for storing a particular key.

C. data versioning

vector clock. *problems*: size, but writes are usually handled by one of the top N nodes, not likely.

D. quorum

R and W. $R + W > N$. W is the minimum number of nodes that must participate in a successful write operation. (high availability: $W = 1$)

E. sloppy quorum

all read and write operations are performed on the first N *healthy* nodes from the preference list.

F. Merkle tree

a Merkle tree is a hash tree where leaves are hashes of the values of individual keys. parent nodes higher in the tree are hashes of their respective children. two nodes exchange the root of Merkle tree corresponding to the key ranges that they host in common. subsequently, using the tree traversal above the nodes determine if they have any differences and perform appropriate synchronization action.

G. partition strategies

1) T random tokens per node and partition by token value. *problems*: 1, scan on the backgrpound; 2, Merkle tree changes; 3, snapshot due to randomness in key ranges, complicated archival.
2) T random tokens per node and equal sized partitions.
3) Q/S tokens per node, equal-sized partitions. *pros*: 1, efficiency; 2, faster bootstraping/recovery; 3, ease of archival.

XI. DEMAND-BASED CO-SCHEDULING

A. uncoordinated vs. coordinated scheduling

uncoordinated scheduling, also called local scheduling, allows each per-CPU scheduler to make its own decision on time-sharing among its assigned threads without any coordination with threads on other CPUs. *pros*: high throughput, low overheads. *cons*: ineffective for communicating workloads.

coscheduling is a representative scheme of coordinated scheduling that allows cooperative threads to be synchronously scheduled and descheduled. *cons*: CPU fragmentation, since cooperative threads cannot be scheduled until thier required CPUs are all available. *ineffective* utilization with sequential workloads.

demand-based coscheduling dynamically initiates coscheduling only for communicating threads, whereas non-communicating ones are managed in an uncoordinated fashion.

B. demand-based coscheduling

TLB shootdown is a kernel-level operation for TLB synchronization via inter-CPU (inter-vCPU) communication. OSes use an IPI to notify a remote CPU of TLB invalidation. a busy-waiting vCPU could consume excessive CPU cycles if one of the recipient vCPUs is not immediately scheduled. -> a TLB shootdown IPI is regarded as a performance-critical signal of inter-vCPU communication that needs to be urgently handled.

excessive lock spinning: a vCPU that is holding a spinlock is involuntarily descheduled before releasing it. happens in the workloads with a large traffic of inter-vCPU communication, especially reschedule IPIs. **a reschedule IPI** is used to notify a remote CPU of the availability of a thread newly awakened by a local CPU. the hypervisor can delay the preemption of a vCPU that initiates a reschedule IPI when another vCPU makes a preemption attempt.

user-level synchronization typically employs block or spin-then-block based primitives, communication between threads can be recognized as reschedule IPIs by the hypervisor. its recipient vCPU can be coscheduled to alle- viate inefficient or unnecessary user-level contention.

urgent request: **urgent queue**. a corresponding vCPU can request to enter urgent state in two ways: 1) event-based and 2) time-based requests. **the event-based request** is used for a vCPU to be retained in urgent state until pending urgent events are all acknowledged. used for TLB shootdown. **the time-based request** allows an IPI to specify a time during which a corresponding vCPU can run in urgent state. used for reschedule IPI.

C. load-conscious balance scheduling

uncoordinated: evenly. lazy algorithms to balance global loads to avoid ineffecient use of hardware. **coscheduling**: assign sibling vCPUs onto different pCPUs in order to prevent them from time-sharing a pCPU. could degrade synchroniza- tion latency if pCPU loads are imbalanced at the moment of assignment. **vCPU stacking**: the time-sharing of sibling vCPUs. *solution*: selectively allows vCPU stacking in the case where the balance scheduling can aggravate load imbalance. by checking if the load of each pCPU is higher than the average load of all pCPUs.

XII. CLOUD SECURITY

A. probing

A probe is **external** when it originates from a system outside EC2 and has destination an EC2 instance. A probe is **internal** if it originates from an EC2 instance and has destination another EC2 instance. **co-residence**: instances that are running on the same physical machine as being co-resident. instances are likely co-resident if they have:

1) matching Dom0 IP address,
2) small packet round-trip times, or
3) numerically close internal IP addresses (e.g. within 7).

B. solution to placement:

let users request placement of their VMs on machines that can only be populated by VMs from their (or other trusted) accounts. In exchange, the users can pay the opportunity cost of leaving some of these machines under-utilized.

C. side-channel attack

Prime+Trigger+Probe measurement: the probing instance first allocates a contiguous buffer B of b bytes (b should be large enough that a significant portion of the cache is filled by B). let s be the cache line size, in bytes.

1) **Prime**: Read B at s-byte offsets in order to ensure it is cached.
2) **Trigger**: Busy-loop until the CPU's cycle counter jumps by a large value. (This means our VM was preempted by

the Xen scheduler, hopefully in favor of the sender VM.)

1) **Probe**: Measure the time it takes to again read B at s-byte offsets.

the time of the final step's read is the load sample, measured in number of CPU cycles. these load samples will be strongly correlated with use of the cache during the trigger step, since that usage will evict some portion of the buffer B and thereby drive up the read time during the probe phase.

covert channel:

- a: larger than the attacked cache level (e.g., $a = 2^{21}$ to attack the EC2's Opteron L2 cache).
- b: slightly smaller than the attacked cache level (here, $b = 2^{19}$),
- d: the cache line size times a power of 2.
- even addresses: (resp. odd addresses) those that are equal to 0 mod $2d$ (resp. d mod $2d$)
- the class of even cache sets (resp. odd cache sets): cache sets to which even (resp. odd) addresses are mapped.

steps:

1) allocate a contiguous buffer B of b bytes
2) sleep briefly (to build up credit with Xen's scheduler).
3) prime: read all of B to make sure it's fully cached
4) trigger: busy-loop until the CPU's cycle counter jumps by a large value. (This means our VM was preempted by the Xen scheduler, hopefully in favor of the sender VM.)
5) probe: measure the time it takes to read all even ad- dresses in B, likewise for the odd addresses. Decide "0" iff the difference is positive.