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. methods

secure binding, visible revocation, abort protocol

why not virtual machine? severe performance penalty

C. 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 pre-cleaning; 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.

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. DEVICE DRIVER

A. redundacy

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