Advanced Operating Systems

> Author: Guanzhou (Jose) Hu 胡冠洲 @ UW-Madison
>
> Teacher: [Prof. Andrea Arpaci-Dusseau]

This note lists knowledge fragments and great ideas shown by systems research papers on the reading list.

Storage: Local HDD FS

FFS

Link: https://dsf.berkeley.edu/cs262/FFS-annotated.pdf

Motivation

- HDD structure; Low throughput by long seeks on random accesses
- Traditional UNIX FS does not do well:
    - No locality in block allocation
        - Separation between metadata and data - long seek on file access
        - Do not group inodes in the same directory together - poor `ls`
    - Blocks too small
        - Pay fixed seeking costs per disk transfer
        - More indirect blocks for a file
    - Poor freelist organization, scattering

Contribution

- Larger block sizes (Sec 3.1):
    - Fewer seeks between block transfers
    - Fewer cases going to indirect blocks
    - One block can be further divided into *fragments* to reduce internal space fragmentation of small files; A file consists of normal blocks + possibly some fragments for the tail only
- Parameterize underlying HDDs (Sec 3.2):
    - Use *bitmaps* instead of linked-freelist
    - *Cylinder groups* & rotation sector skipping calculation
    - For each cylinder group, store superblock at different offset to avoid all superblocks on the top platter
- Better layout policy (Sec 3.3):
    - Put inodes of files in the same directory in the same cylinder group
    - New directory prefers mostly-free cynlinder group - load balances the groups
    - First data block allocated near the inode block
    - Subsequent data blocks at rotationally optimal positions of previous block in a cylinder group
    - Large file after 48KB, then every 1MB: go to a new group

LFS

Link: https://dl.acm.org/doi/10.1145/146941.146943

Motivation

- HDD characteristics; Sequential writes are much better
- Technology:
    - Processors faster, so more pressure I/O
    - Seek time not improving
    - Main memory faster, disks will be dominated by writes as reads are mostly cached in memory
- Workload:
    - Many small files pattern

Contribution

- Whole on-disk structure is a write-forward copy-on-write *log*
    - Inodes are scattered across the log as well

        - Keep a inode map from inode number $\rightarrow$ inode physical address
        - Hopefully, most of the active inode map will be cached in memory
    - In-place updates will become writing a new log entry + a redirection, resulting in invalidating an old block
    - Free space management: *threading* vs. *copying* (Sec 3.2); LFS uses a hybrid way: disk partitioned into *segments*
- Garbage collection by *segment cleaners* (Sec 3.3, 3.4):
    - Will garbage collect segments and write out compact, clean segments
    - Using *segment summary block* for quickly identifying garbage blocks; It lists inode + data offset for each data block in segment, i.e., reverse pointers from data blocks to its inode
    - Introduces the concept of *write cost* to compare performance in Sec 3.4
- Crash recovery by *checkpointing* + *roll-forward* (Sec 4)
    - Checkpoint often: more random I/O vs. checkpoint less: slower recovery

Drawbacks

- GC overhead measurement
- Sequential re-read - poor performance compared to in-place FFS

Storage: Disk Failures

RAID

Link: https://www.cs.cmu.edu/~garth/RAIDpaper/Patterson88.pdf

Motivation

- CPU & main memory faster, HDD I/O performance bound
- Inexpensive disks almost as good as expensive large disks

Contribution

- *Redundant* array of inexpensive disks to overcome the reliability issue of grouping many disks together
- RAID 0: striping, no redundancy
    - Capacity: $N * C$
    - How many can fail: 0
    - Latency: D
    - Random throughput: $N * R$
    - Sequential throughput: $N * S$
- RAID 1: simple mirroring; RAID 01: mirror of stripes vs. RAID 10: stripe of mirrors (assume RAID 10 here)
    - Capacity: $\frac{N}{2} * C$
    - How many can fail: 1
    - Latency: D
    - Random read throughput: $N * R$ (assume lots of overwhelming requests)
    - Random write throughput: $\frac{N}{2} * R$
    - Sequential read throughput: $\frac{N}{2} * S$ or $N * S$
    - Sequential write throughput: $\frac{N}{2} * S$
- RAID 4: striping + single parity ECC disk (assume *subtractive* parity update policy, 1R+1W on data block and 1R+1W on parity block for each write)
    - Capacity: $(N-1) * C$
    - How many can fail: 1
    - Read latency: D
    - Write latency: $2 * D$
    - Random read throughput: $(N-1) * R$
    - Random write throughput: $\frac{R}{2}$ (parity disk needs to do 1R+1W for every write request)
    - Sequential read throughput: $(N-1) * S$
    - Sequential write throughput: $(N-1) * S$ (parity calculated for full stripe)
- RAID 5: striping + rotate parity ECC disk for each stripe (assume *left-symmetric* block layout)
    - Capacity: $(N-1) * C$
    - How many can fail: 1
    - Read latency: D
    - Write latency: $2 * D$
    - Random read throughput: $N * R$

- Random write throughput: $N * \frac{R}{4}$ (every disk is responsible for 1R+1W on one data block and 1R+1W on its parity block at any given time point)
- Sequential read throughput: $(N-1) * S$
- Sequential write throughput: $(N-1) * S$ (parity calculated for full stripe)

> Metrics used:
>
> - $N =$ number of disks
> - $C =$ capacity of 1 disk
> - $S =$ sequential throughput of 1 disk
> - $R =$ random throughput of 1 disk
> - $D =$ latency of 1 small disk transfer

Drawbacks

- Tradeoffs between capacity, reliability, and performance
- RAID 0 is actually preferred RAID 5 for database workloads since it is much better at random throughput

RDP

Link: https://www.usenix.org/conference/fast-04/row-diagonal-parity-double-disk-failure-correction

Motivation

- More inexpensive disks used as RAID lead to more frequent disk failures, so protecting against double disk failure is worthwhile doing
  - RAID reconstruction can reveal/trigger a second failure
  - Failures are not independent if we choose disks from the same make, model, etc.
- Desired property of a dual-failure protection algorithm:
  - Stores data in clear, unencoded
  - Uses simple XOR operations on parity

Contribution

- Two types of disk failures
  - Whole-disk: the disk goes bad, cannot be used anymore
  - Media failure: individual sector errors/corruption on a request
- Row-diagonal parity algorithm to handle double disk failures at the same time
  - Two parity disks: row + diagonal
  - On two disk failures, there will always be one "diagonal stripe" which only misses one member block; Start from there, we can do cascading reconstruction to recover

Drawbacks

- Group size - Best-case failure tolerance tradeoff
  - Fewer larger groups saves capacity (fewer parity disks)
  - But smaller groups have better fault-tolerance

Storage: User File Benchmarking

iBench

Link: https://research.cs.wisc.edu/adsl/Publications/ibench-sosp11.pdf

Motivation

- Home-user applications I/O behaviors have not been carefully studied

Contribution

- Observations from the test suite:
  - A file is not a file: it can maintain its internal database structure
  - Pure sequential access is very rare
  - Small, auxiliary files dominate
  - Most use `fsync()` explicitly to force writes; Renaming and staging is also popular
  - Frameworks influence how applications do I/O

Storage: Archival Storage

SnapMirror

Link: https://www.usenix.org/conference/fast-02/snapmirror-file-system-based-asynchronous-mirroring-disaster-recovery

Motivation

- Archival data backup is important
  - Asynchronous backup offers inadequate protection
  - Synchronous backup significantly hurts performance
- WAFL file system log-structured nature; Compared to LFS:
  - Only one `fsinfo` pointer can be overwritten
  - Has local snapshotting feature
  - Active map + individual block granularity

Contribution

- Do asynchronous periodic updates with bounded frequency
  - Reduces the need to transfer unnecessary data overwritten in the same update window (Fig 3)
  - User can specify a proper update frequency to tradeoff between performance and protection
- When a mirror schedules an update, it tells the source to make an *incremental reference snapshot*; The source will check both sides' WAFL *active map*
  - If a block is not allocated on both sides, the block is unused - will not be transfered
  - If a block is active on both sides, the block is unchanged since latest snapshot - will not be transfered
  - If a block is only on the base active map, the block is deleted - will not be transfered
  - If a block is only on the new active map, the block is newly-added - IS transfered
- The FS superblock `fsinfo` block will not be updated until all blocks in an update have been finished
- Mirror maintains exactly the same logical block layout as the source - no extra indexing or mapping needed
  - Allows efficiently calculating difference with active map
  - Sequential layouts remain sequential on the mirror, which is not the case for other archival storage systems

Drawbacks

- On larger systems or systems with very frequent changes, SnapMirror snapshots are gonna be big and slow

Venti

Link: https://www.usenix.org/conference/fast-02/venti-new-approach-archival-data-storage

Motivation

- Archival storage systems impose a *write-once* policy and keeps written archives forever
- Cryptographically-safe hashing as *fingerprints* for static blocks
  - Collision resistent
  - Almost impossible to revert

Contribution

- Venti layer: Use hashing to mark fingerprints of blocks
  - *Deduplicates* blocks with exactly the same content and can quickly identify it
  - Helps security guarantees - can check content against fingerprint
- Fingerprints lead to no locality on both index and data disks, so they keep an index cache and a data cache
- Application is responsible for mapping namespace $\rightarrow$ Venti fingerprints

Drawbacks

- Performance is ridiculously bad, even with cache

Data Domain

Link: https://www.usenix.org/conference/fast-08/avoiding-disk-bottleneck-data-domain-deduplication-file-system

Motivation

- Same as Venti, but is a significantly improved data-deduplication system over Venti

Contribution

- Layered deduplication system structure:
  - *Content store*: object byte range
  - *Segment store + Index*: segment descriptors; 3 essential performance techniques:
    - *Summary vector*: bloom filter per segment for quick checking fingerprint definitely not in the segment
    - *Stream informed segment layout*: if we have seen fingerprint sequence `f1, f2, f3`, then when we see `f1` again, we are probably going to see `f2, f3`; So, pack these blocks in a *container*
    - *Locality preserved caching*: cache prefetches the container for better locality
  - *Container manager*: actual data
- Steps on a segment write (Sec 4.4)
- Shows great I/O reduction by the 3 techniques (Table 4)

Storage: FS Caching

ARC

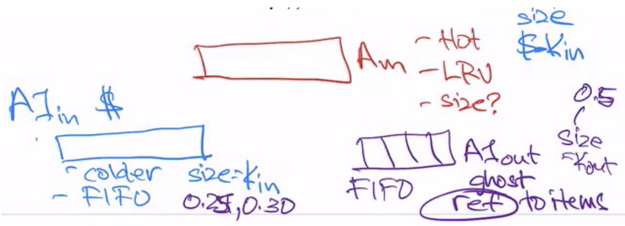
Link: https://www.usenix.org/legacy/events/fast03/tech/full_papers/megiddo/megiddo.pdf

Motivation

- Metrics / goals:
  - High hit rate
  - Low implementation overhead - low algorithm complexity
  - Space overhead
  - *Scan resistance*
  - No priori parameters to tune - self-adaptive to the workload
  - Balance between recency & frequency
- Problems with LRU:
  - Not scan-resistant
  - Poor concurrency
- Problems with LFU:
  - Implementation complexity
  - Remembers ancient knowledge - not adapting to changes
- Problems with LRU-k:
  - Replaces the one with oldest recent k-th reference
  - Very expensive to maintain this information

Background

- Optimal offline replacement: given the future reference trace, replace the entry that will be seen again farthest in the future
- 2Q algorithm:

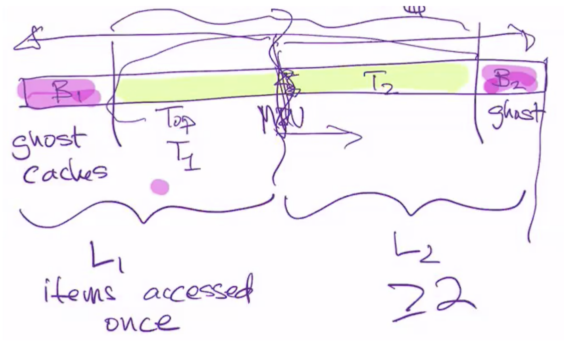


  - Re-ref of entry in A_m: move to MRU position of A_m
  - Re-ref of entry in ghost FIFO A_{1out}: is a cache miss; promote to MRU position of A_{1out}
  - New ref of an entry: is a cache miss; promote to A_{1in} FIFO tail
  - Re-ref of entry in A_{1in}: move to A_{1in} FIFO tail, NOT to A_m (intuition: it is possible that an entry appears to be hot at first but the workload is actually a scan, so we wait until it falls into A_{1out}, pay a tax of cache miss, and only then be confident that this is a hot entry and promote it to A_m)
  - A sequential scan with 2Q will go through A_{1in} and A_{1out}, but will not pollute A_m
- MQ algorithm:

- Have frequency count f_c, and the entry will be in queue $\log_2(f_c)$
- A sequential scan with MQ will go through Q_0 and Q_{out}

Contribution

- ARC algorithm:



  - Re-ref in T_1 or T_2: move to MRU position of T_2
  - Re-ref of entry in ghost B_1: is a cache miss; move to MRU position of T_2; Increase p (enlarge T_1) by $\text{size}(T_2)/\text{size}(T_1)$
  - Re-ref of entry in ghost B_2: is a cache miss; move to MRU position of T_2; Decrease p (enlarge T_2) by $\text{size}(T_1)/\text{size}(T_2)$
  - New ref of an entry: is a cache miss; move to MRU position of T_1
  - A sequential scan with ARC will go through T_1 and B_1

Drawbacks

- Only investigating the replacement policy
- No prefetching, purely on-demand paging
- Only reading, not discussing write-allocation policy
- Assuming a strict storage hierarchy

Storage: Crash Consistency

ALICE

Link: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/pillai

Background

- Benefits of doing *journaling*:
  - Providing crash consistency for *update-in-place* file systems
  - Achieves atomic FS updates despite crashes - turns multiple disk writes into a single atomic action; Example: a file append must update three places atomically
    - The bitmap to mark new data block as allocated
    - The inode to add pointers to the new block
    - The new data block
- *Ordered mode* journaling ensures FS metadata consistency: $D|J_M \rightarrow J_C \rightarrow M$
  - D is the new data block
  - J_M contains:
    - `transaction begin` block
    - data bitmap update
    - file inode update
  - J_C contains the `transaction end (commit)` block (when replaying the journal, only replay committed entries)
  - M is the actual metadata in-place updates
- *Write-back journaling is weaker: does not ensure ordering of D before M
- *Data journaling mode* is stricter: protects user data writes, but involves *write-twice penalty*

Motivation

- Different FS provide different persistency properties guarantees

- Applications care about crash consistency but they cannot assume running over a data journaling FS, so they have to implement their own sophisticated data consistency protocols

```
creat(/x/log1);
write(/x/log1, "2, 3, checksum, foo");
fsync(/x/log1);
fsync(/x);
pwrite(/x/f1, 2, "bar");
fsync(/x/f1);
unlink(/x/log1);
```

  - More checksums & more `fsync()`'s
  - Sometimes unnecessary when running over strict-mode FS

Contribution

- Use *BOB* to study FS persistency properties; Different FS provide very different set of consistency semantics
- Use *ALICE* to run application syscall traces over abstract FS models, reorder those syscall logical operations, and see under what circumstances will it have inconsistency vulnerabilities

OptFS

Link: https://research.cs.wisc.edu/adsl/Publications/optfs-sosp13.pdf

Motivation

- Default FS journaling is *pessimistic*: forcing users to call `fsync()`'s frequently
  - The `FLUSH` command (part of `fsync()`) itself does not enforce ordering
  - It just flushes the *disk cache* and make sure they are actually persistent
- Disk flushes introduce huge overhead (Fig 1)

Contribution

- Model *probabilistic* crash consistency by measuring the inconsistency *window* length (Sec 3)
- Get rid of the two flushes in $D|J_M \rightarrow J_C \rightarrow M$ *optimistically* while still maintaining *atomicity*:
  - *Checksumming* over $\overline{D|J_M|J_C}$ so that these three can be reordered whatever the disk wants; But this does not improve durability because we are not forcing anything to be persisted - we are just now able to check what goes wrong after a crash and ignore/abort these
  - *Asynchronous durability notifications* (ADN, a new disk interface) to get rid of the second flush; FS/user applications do not have to block on a flush - instead, the metadata in-place update M will simply wait for the ADNs of all three D, J_M, J_C and the FS can do other stuff at this time; It aborts this operation after crash if haven't received those ADNs before the crash
- Splits `fsync()` into two different interfaces:
  - `osync()` only ensures ordering but not durability (Slide 66)
  - `dsync()` as the original `fsync()`

Drawbacks

- Worsens *freshness* (*durability*): after a crash, state will be consistent but not very fresh - all operations without receiving metadata in-place update's ADN will be forgotten
- Introduces a new disk interface ADN and argues that disks should provide such interface - limits compatibility on current hardware

Storage: SSD Contract

Unwritten Contract

Link: http://pages.cs.wisc.edu/~jhe/nvmw18-he.pdf

Background

- SSD structure; how NAND flash chips work and how they form an SSD device
  - *Channels* of *blocks* of *pages*
    - Read in pages
    - Write (program) some 0's only in erased pages
    - Erase a whole block to 1's
  - SLC (expensive, robust) vs. MLC (more capacity, less robust)

- *Flash translation layer* (FTL) responsible for logical $\rightarrow$ physical page mapping and:
  - Reallocate on update writes or trim pages
  - *Hybrid* FTL has coarse-grained block-level mapping for most of physical address + page-level mapping for only recent data, to save the space for the mapping
    - Full merge
    - Partial merge
    - Switch merge
  - Garbage collection
  - Wear leveling

Contribution

- The written contract is the device interface specifications
- Reveals the *unwritten contract* of SSD devices: how they perform and react to different workload characteristics; Rules for good performance:
  - Large request scale: to utilize high internal parallelism of channels
  - Access with locality: to avoid translation cache misses
  - Aligned sequentiality: for hybrid-mapping FTL mapping cache, improves hit rate and encourage switch merges
  - Group by death time: reduce garbage collection work
  - Uniform lifetime: reduce wear leveling work

Storage: ML for Sys

Bourbon

Link: https://www.usenix.org/conference/osdi20/presentation/dai

Motivation

- *Log-structured merge* (LSM) trees were for HDDs to get better sequentiality, but has very large I/O amplification
  - Large write amplification - lots of merges
  - Large read amplification - lots of lookup steps
- LevelDB uses LSM trees
  - Lookup returns latest version of an item on top-most level
  - `sstable` files are sorted & immutable; In L0, `sstables` may not cover disjoint ranges; In L1 and below, range of keys is disjoint across files in that level
  - Lookup steps at a level:
    - FindFiles: find candidate file(s) that may contain the key (*indexing*)
    - LoadIB + FB: in a file, load index block and bloom-filter block
    - SearchIB: search index block for data block that may contain the key (*indexing*)
    - SearchFB: use bloom filter to see if key definitely not in the data block (*indexing*)
    - LoadDB: load data block if bloom-filter says positive
    - SearchDB: binary search the data block (*indexing*)
    - ReadValue: if found the key, read and return the value
- WiscKey reduces LevelDB write amplification by storing values in a separate value log and only store a pointer to the value in the `sstable`'s (Fig 1)
  - Breaks some sequentiality but reduces write amplification - actually values won't move during merges
  - Must perform an extra random read to fetch a value
- ML for systems is a hot topic; *Learned indexes* have the potential to be applied to LSM trees, but they are more tailored to read-only settings

Contribution

- Applies learned indexes to LSM trees by modeling a file / a whole level as a *piece-wise linear regression* (PLR) model
  - Essentially, breaks a file / level into monotonic segments with different slopes
  - For a key k, predicts its offset in file / level with error bound as $[p-\delta, p+\delta]$; True offset guaranteed to be in the error bound
  - To lookup the key in file / level:
    1. Binary search for the segment - $O(\log s)$

2. Predict final location using the segment - $O(1)$
3. Linear search within error bound - $O(1)$

- Bourbon benefits more when cost of indexing steps is high compared to cost of loading data
- File model vs. level model
  - Models last longer for
    - File models than level models (any file in level changes leads to invalidating the level model)
    - Workloads with fewer writes
    - Lower levels
  - A valid model is more beneficial for
    - Higher levels
    - Level models than file models
  - Learns a model by all files ever existing at that level
- Performance behavior with different write percentages (Fig 12)

Storage: Persistent Memory

Mnemosyne

Link: https://research.cs.wisc.edu/sonar/papers/mnemosyne-asplos2011.pdf

Motivation

- *Storage-class memory* (SCM) coming out: persistent NVRAM connecting to memory bus
  - Ultra-fast latency
  - Writes significantly worse than reads: needs write reduction
  - Endurance pretty bad: needs good wear leveling
- Existing storage systems cannot manage such devices effectively
  - Persistent naming: what a persistent virtual address maps to should always be on persistent memory
  - Crash consistency: hard to do because memory bus directly controlled by the CPU
  - Kernel FS system calls overhead: too large for fast SCM - build *direct-access* user-level libs
  - Performance degradation for reducing writes

Contribution

- Persistent naming region: virtual memory segments mapped to SCM
  - Swaps SCM pages to per-region backing file
  - Sandboxes user SCM memory leaks
- Supports consistent updating by:
  - Assuming in-place small writes ($\leq$ 8 bytes) are atomic
  - For appends: use logging mechanisms, just like journaling FS; Use "cache line flush" and "memory fence" instructions to force ordering
  - For in-place updates: do *shadowing*, i.e., write in a replica and simply atomically swap the pointer
- Introduces the *raw word log* for detecting torn writes (Slide 50)
- Supports memory transactions to bundle a bunch of operations and send them to SCM atomically (Sec 5)

LevelHash

Link: https://www.usenix.org/system/files/osdi18-zuo.pdf

Motivation

- Same as Mnemosyne above
- Indexing structures are common, so hash tables do matter

Contribution

- Dual-level hash table (Fig 1)
- Insertion operation steps (Sec 3.1 - D_4)
- Resizing operation steps (Sec 3.2; Fig 3)
- Consistency guarantees brought by the assumption that updating the bitmap header of a bucket is small & atomic
  - Delete: clear a bit in the bitmap in one atomic write (*log-free*)

    - Insert: write the element first, then set a bit in the bitmap in one atomic write (*log-free*)
    - Update:
        - If there is an empty slot in the bucket, then write the new value to an empty slot first, then swap the two bits in the bitmap in one atomic write (*log-free*)
        - Otherwise, do logging

Synchronization: Monitors

Monitors

Link: https://john.cs.olemiss.edu/~dwilkins/Seminar/S05/Monitors.pdf

Motivation

- Semaphores for OS resource synchronization and its lack of abstraction

Contribution

- The concept of *monitors* for OS resource synchronization
    - Associates (encapsulates) data with monitor
    - Handles mutex locking automatically - less error-prone
    - Monitors + CV = separates mutual exclusion & scheduling, whereas semaphores mixes these two jobs
    - Helps formalism - can specify mathematical *invariants*
    - Monitors implement 3 types of procedures:
        - *Entry*: grabs lock
        - *Internal*: assumes lock held
        - *External*: does not grab lock
- Introduces *condition variables*:
    - Has a queue; Supports `cv.wait()` and `cv.signal()` semantics
    - Fundamental difference with semaphores:
        - Condition variables rely on external conditions to decide whether to wait / signal - more flexibility & generality for the programmer
        - Semaphores track that condition with itself, as the semaphore value
- Locks + condition variables has the equivalent semantics power as semaphores

Drawbacks

- Implicitly assumes the semantics that Signaling process must stop running and immediately relinquish monitor lock; Waiting process, after woken up, must acquire the lock and run immediately
    - Brings extra context switches for some situations
    - Complex interactions between the monitor & the scheduler
    - NO need of the `while` loop on condition variables

Mesa

Link: https://people.eecs.berkeley.edu/~brewer/cs262/Mesa.pdf

Motivation

- The experience of using monitors in the Mesa system

Contribution

- Discusses why non-preemptive scheduler is problematic to implement mutual exclusion (Page 3):
    - Malicious process can get complete control
    - Disables the ability to handle device interrupts
    - Reduces *modularity* of critical sections - cannot call a procedure that yields the processor, e.g., page faults
    - Cannot handle multiprocessors
- Implements the condition variable semantics in a practical way:
    - Signal wakes up any waiting process
    - Woken up process not guaranteed to be scheduled next
    - Woken up process not guaranteed to acquire monitor lock next

    - In this way, waking up a process only pulls it out of CV queue and puts in into monitor lock queue
    - MUST have the `while` loop on condition variables, because the condition may no longer hold when the woken up process actually scheduled to get the monitor lock
- Other optimizations:
    - Broadcast semantic: `cv.notifyAll()` - more context switches, worse performance
    - Timeout & Aborting from a wait
    - Introduces the "naked notify" problem (Sec 4.2):
        - Hardware device may call a `notify` without grabbing a lock
        - Kernel cannot do:

        ```
        while (!cond)
                // <- WRONG! device notify can happen here.
            cv.wait();
        ```

        Instead, change it into a binary semaphore so that the condition checking and waiting decision happens in one atomic instruction
    - Fixes the problem of *priority inversion* by dynamic *priority donation*
    - Nested monitor calls: split outer monitor into two parts

Drawbacks

- Adapting monitor semantics from Hoare to Mesa isn't always changing `if` to `while`! Consider this RW lock implementation:

```
startread() {
    if (busy || OKtowrite.queue)    // Prevents writer starving.
        OKtoread.wait();
    readercount++;
    OKtoread.signal();  // Allows more readers to come in.
}

endread() {
    readercount--;
    if (readercount == 0)
        OKtowrite.signal();
}

startwrite() {
    if (readercount > 0 || busy)
        OKtowrite.wait();
    busy = true;
}

endwrite() {
    busy = false;
    if (OKtoread.queue)
        OKtoread.signal();
    else
        OKtowrite.signal();
}
```

Adapting it to Mesa semantics:

```
startread() {
    while (busy || OKtowrite.queue)     // WRONG! there might be a writer in queue
        OKtoread.wait();                //        when woken up by first writer's end
    ...
}
```

Better way:

```
startread() {
    if (OKtowrite.queue)
        OKtoread.wait();
    while (busy)
        OKtoread.wait();
}
```

Synchronization: Multicore Scalability

Linux Scalability

Link: https://www.usenix.org/conference/osdi10/analysis-linux-scalability-many-cores

Motivation

- Many-core architectures are becoming popular these days - most of them are NUMA, i.e., multiple chip sockets
  - Cores on the same socket share an L3 cache; Different sockets do not have shared L3 cache
  - Each socket has its fast local memory and accessing other sockets' "remote" memory is slow
- Traditional Linux kernel does not seem to scale well onto multicore hardware - this work does not go for a microkernel / message-passing kernel design, it tries to argue that Linux can be adapted to scale well on multicore computers as well
- Common scalability limitation causes:
  - Locking shared data structure / counter
  - Writing shared memory location - waiting for cache coherence protocol
  - Competing for shared hardware resources, e.g., interconnect bus
  - Too few tasks to keep all cores busy
- Linux implements spin-lock as a *ticket lock*:

```
void spin_lock(lock) {
    t = atomic_inc(&lock->next_ticket);
    while (t != lock->current_ticket);
    ...
}

void spin_unlock(lock) {
    lock->current_ticket++;
}
```

Contribution

- Picks a collection of kernel-intensive applications to form *MosBench*
- Runs *MosBench* and identifies common scalability problems in the Linux kernel (Fig 1) - not application-specific bottlenecks
  - Fix the bottlenecks and re-run the benchmark, until good performance
  - Or until they find out that this is an application-specific bottleneck
- Example: the Exim application's primary collapse cause is contention on a spin-lock on shared `vfsmount` table
  - Ticket locks cause intense cache coherence protocol traffic on the lock structure itself
  - Even if the critical section is small - which is good - the lock structure itself can cause contention
- Optimizations proposed by the paper:
  - Per-core data-structure cache to ease the lock contention problem above
  - *Sloppy* distributed counters to delay global counter access only when background garbage collection work starts

Commutativity Rule

Link: http://www.read.seas.harvard.edu/~kohler/pubs/clements13scalable.pdf

Motivation

- The above paper focuses on developer effort to identify bottlenecks
  - New workloads may expose new bottlenecks
  - More cores may expose new bottlenecks
  - The real bottlenecks may be in the interface design itself

- Goal: guideline to design scalable interface

Contribution

- Interface scalability issue example:
  - POSIX `open()` must return the current lowest unused non-negative integer as FD
  - If we do not force returning the lowest FD, then the interface can be scalable
- The *commutativity* rule: whenever interface operations *commute*, they can be implemented in a way that scales because they are *conflict-free*
- Commutativity is sensitive to operations, arguments, and internal state (e.g., whether creating files in the same directory or not)
- Develops a *commuter* framework to detect scalability bottlenecks on a concrete model:
  - Input interface specification model, e.g., POSIX, produced by symbolic execution and model checking
  - Auto-generate commutativity conditions testcases
  - Run the tests through a concrete implementation, e.g., Linux, and identify whether each case is conflict-free
- Guidelines for designing a scalable FS (Sec 6.3)

Drawbacks

- This study is oblivious to applications; It only checks whether a kernel implementation meets commutativity requirements when it should; It does not consider whether those violations are severe - whether a particular application is actually calling those interfaces (or using bad arguments) at all

Synchronization: Scalable Locking

RCL

Link: https://www.usenix.org/conference/atc12/technical-sessions/presentation/lozi

Motivation

- Mutex locks tend to contend a lot on many-core machines, due to:
  - Lock data structure itself bouncing off among caches - just as presented in the Linux Scalability work
  - A critical section may access some global data structure, which may perform badly when cache locality on this data structure is bad - many cores grabbing this structure into their own cache line
- Goals:
  - Implement entirely in software
  - Support legacy applications
  - Support blocking in critical sections, nested critical sections, and condition variables

Background

- MCS lock to improve locality (Slide 16):
  - Every thread spinning on self-cached variable instead of a global lock variable (measurement: Fig 7)
  - MCS-TP: a further optimization which *parks* a thread of it has spinned for too long - avoids busy waiting

Contribution

- Identify that cache locality is important for speeding up highly-contended critical sections (Fig 3)
- *Delegate* a specific thread for a critical section, so that everything needed by the critical section will always be cached in that core's cache
  - Use cache line-sized mailboxes to notify the server
  - Server loops through mailboxes and serve any requests to execute critical section (Fig 2)
- Tricky thing is to identify what locks are beneficial to be converted to RCL; Non-contended critical sections or very short critical sections will not be suitable for RCL and may not benefit

Drawbacks

- Does not show application performance without high contention - the applications themselves may not scale well, perhaps not the problem of the lock-contended critical sections
- Blocking in critical sections / nested critical sections
  - Adds multiple threads as server cores
  - Further leads to having extra CAS locks for server threads to grab mailbox requests atomically

- Multiple independent locks & critical sections:
  - Adds multiple threads as server cores (Fig 13): false serialization kind of solved by having two server cores
- Final performance results seem awkward (Fig 9): pay attention to the vertical numbers!

Shuffling

Link: https://taesoo.kim/pubs/2019/kashyap:shfllock.pdf

Motivation

- Same as the RCL work above
- Goals:
  - Adapting to different contention levels / over-subscription levels / workloads
  - Minimizing memory *footprint* (i.e., memory usage)

Background

- Hierarchical locks (Slide 9): high memory usage; poor normal single thread performance

Contribution

- *Shuffling* mechanism to sort thread waiters on the fly
  - Shuffler will be a waiter, so queue re-ordering work is off the critical path; Stop shuffling when:
    - It gets the lock
    - It completes one pass through the queue
  - Groups waiters on the same NUMA socket, i.e., same socket id, together in the queue
  - The last thread moved in the group will be the next shuffler
  - For RW locks, only group writers together since readers will access concurrently anyway

Drawbacks

- Queue re-ordering affects lock *fairness* and may cause *starvation*: keeps a simple quota count trying to solve this
- Bad data representation & graph drawing (Slide 44): non-linear x-axis!

Scheduling: User-Level Threads

Scheduler Activations

Link: https://web.eecs.umich.edu/~mosharaf/Readings/Scheduler-Activations.pdf

Motivation

- People want user-level threads support
  - Ultra-fast thread management & context switching
  - Flexible application-optimized thread scheduling policy

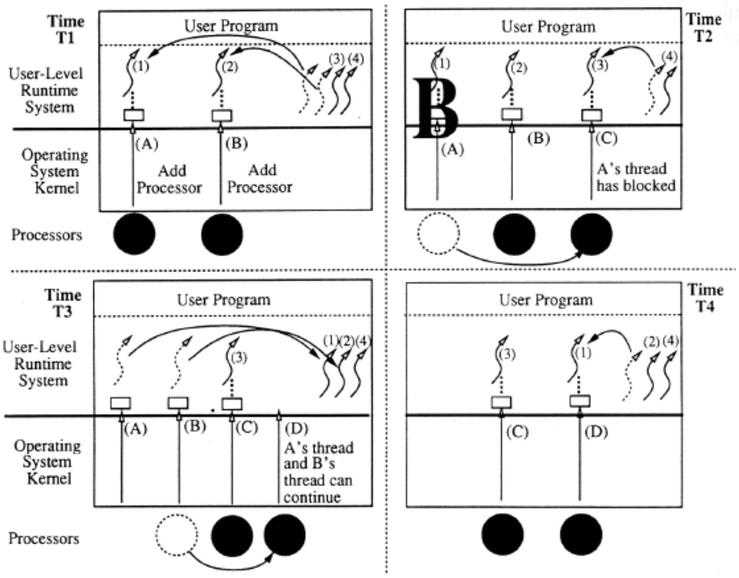
Background

- Problems with naïve user-level threads:
  - Kernel thread vessels may block and get preempted without notifying the user
    - Common solution is to create more kernel vessels than physical processors, and when one vessel blocks, more are available
    - Problem: there will be too many kernel threads at some time and extra OS scheduling will happen
  - Kernel thread vessels scheduled obliviously to user-level thread state, for example the user thread may be holding a lock

Contribution

- Each application given virtual multiprocessor
  - Application knows how many and which processor cores it has
  - Application has complete control over threads running on those cores
  - OS kernel knows how many processors would be useful to each application
  - OS kernel has complete control over which cores are given to which application
- Every processor the kernel gives to an application resides in a *scheduler activation* (SA, vessel)

- Kernel → User thread system upcalls:
  - Add this processor
  - Processor has been preempted
  - SA x has blocked
  - SA x has unblocked
- User thread system → Kernel downcalls:
  - Give me more processors
  - This processor is now idle
- Demonstration (Fig 1):



  - Note: every upcall comes with a new SA vessel, so at time T_3, to convey the message that `A`'s thread has unblocked, the kernel actually needs to revoke a current SA (`B` in this case) and give a new one (`D`) to make the notification
  - The kernel picks `B` to preempt at T_3, but the user may prefer preempting `C` - so there needs to be an interface to tell the OS which SA is preferred to be preempted
  - For the same reason, for a "Processor preempted" upcall, the kernel actually needs to take away two SAs and give a new one to do that notification
- If a thread is in a critical section and kernel wants to preempt it, checks if its PC range is in a critical section; If so, silently run the copy that returns to thread library; In common case, won't slow down normal critical sections

Drawbacks

- Only focuses on CPU - not memory or I/O
- How to do fair allocation of how many processors to give to each application?

Arachne

Link: https://www.usenix.org/conference/osdi18/presentation/qin

Motivation

- Concrete implementation of a user-level thread management system that reduces latency in datacenters
- Difference in assumptions from scheduler activations:
  - Tasks tend to be very short-lived
  - Assumes lots of memory so that threads do memory mapping and don't block on page faults
  - Assumes asynchronous interfaces for performing I/O so that threads don't block on I/O

Contribution

- The basic design is very similar to scheduler activations (Fig 1)
- *Cooperative revoking* of cores: the *arbiter* asks runtime to hand back a core and assumes that the runtime will periodically check that flag - works well for short-lived threads
- Extremely cache-optimized design:

    - Prepares thread contexts (vessels) in advance; Arbiter loops through them and check for "runnable" ones to give to runtime
    - Application runtime $\rightarrow$ Arbiter: sockets, slow but rare
    - Arbiter $\rightarrow$ Application runtime: shared memory page
- Comparison across thread management solutions (Tab 2; Fig 2)
    - Arachne places new threads on different core
    - Go creates thread on parent's core (load balance later if needed)
- Improvement to Memcached (Fig 3 (a)):
    - $\downarrow$ Arachne takes up 2 cores so they cannot be used for work: Less max throughput; At higher throughput, latency goes up high
    - $\uparrow$ Great latency improvement ratio in less-loaded scenarios
    - $\uparrow$ For a fixed latency, Arachne delivers higher throughput
- Isolates latency behavior from background, high-throughput jobs (Fig 4)

Drawbacks

- Does not handle kernel thread blocking

Scheduling: System Services

SEDA

Link: http://www.sosp.org/2001/papers/welsh.pdf

Background

- *Work dispatch* model: dispatch a new thread for every new network request
    - Easy to program √
    - Overhead in thread creation, scheduling, and lock contention
    - Can use a bounded thread pool, but still fairness issues
- *Event-driven* model: one thread per CPU, loops continuously handling events (of different types)
    - Robust throughput √
    - Threads cannot block, tough on I/O
    - Ugly to code all logics in one event-handling thread

Contribution

- An architecture of a stream of stages (Fig 5)
- Every stage has a thread pool and two *controllers* (Fig 6 & 7):
    - *Thread pool controller* adjusts the number of threads based on incoming queue length
        - Add when queue length $>$ threshold
        - Reduce when some thread goes idle
    - *Batching controller* adjusts the number of events processed by each iteration of the event handler (batch size) based on outgoing rate (throughput)
        - Maintain the lowest batch size that sustains high throughput
- Async I/O for sockets; Outstanding I/O threads for files; Blocking I/O may be solved by using mechanisms like scheduler activations

TAM

https://www.usenix.org/conference/osdi18/presentation/yang

Motivation

- Scheduling in current system-intensive DB/FS applications are complex and broken
- Needs a way to understand scheduling points within these applications

Contribution

- The *thread architecture model* (TAM) (Fig 2) helps expose 5 scheduling problems:
    1. Lack of scheduling points
    2. Unknown resource usage (Slide 16)
    3. Hidden contention between threads

4. Uncontrolled blocking (Slide 20)
5. Ordering constraints upon requests
- Helps improve current systems scheduling by optimizing its TAM architecture, verify that through simulation, and then implement those changes into production code

Monotasks

Link: http://kayousterhout.org/publications/sosp17-final183.pdf

Motivation

- Help users with *performance clarity* and *prediction*: answer what-if questions that how much faster would my application run if given x% more of this type of resource?
- Traditional *multitasks* pipeline multiple types of resources at fine-time granularity - bottleneck shifts over time

Contribution

- Proposes the *monotasks* model: each task uses only one resource
  - Every monotask starts only if all its dependencies have been finished
  - Each resource has its own dedicated scheduler
  - Answering what-if questions: see Slide 34 for an example
- Two layers of schedulers on each worker machine:
  - Local DAG scheduler to decompose and track dependencies for monotasks
  - Per-resource scheduler with queues
- Global scheduler dispatches multitasks to worker machines; E.g., if a worker machine can do 4 CPU monotasks + 4 network monotasks + 2 disk monotasks at the same time, then dispatch 10 multitasks to it for full utilization

Drawbacks

- On disk monotasks, they no longer buffer any writes, but instead writes to disk immediately to ensure complete control over the disk resource
- Monotasks hurt performance (significantly) if:
  - There isn't enough concurrency (Fig 8)
  - Disables buffer cache - Comparison to Spark with buffer cache off, not fair comparison! (Sec 5.3)

Scheduling: Scheduler Algorithms

Lottery Scheduling

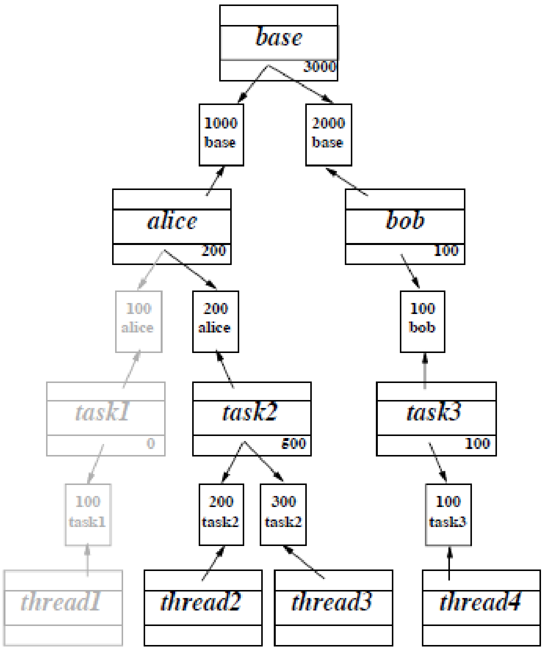
Link: https://www.usenix.org/legacy/publications/library/proceedings/osdi/full_papers/waldspurger.pdf

Motivation

- *Priority*-based schedulers do not naturally consider providing a fair proportional share - they just let the highest-priority thread to run; They are difficult to control and poorly understood
- Want to provide *proportional share* and provide hierarchical *modularity*

Contribution

- Using *lottery tickets* to represent resource rights; At every scheduling decision, the thread winning the lottery will get scheduled
  - Relative
  - Abstract
  - Uniform
- At every level of the *currency graph*, the ratio is local (Tab 1; Fig 3): when thread 1 becomes active, Alice would now have a total of 300



- Ticket *transferring*:
  - Process doing work on another process's behalf
  - Process is waiting for another process (e.g., locks)
- *Compensation tickets*: if a thread is only active for a short period of time (f fraction of the time slot) and then goes to sleep, on the next scheduling point, inflate its tickets by $\frac{1}{f}$ to provide *instantaneous fairness*
- Results:
  - Can decrease proportion of CPU given to tasks that need less work (Fig 6)
  - Can transfer tickets to process working on your behalf (Fig 7): clients $\rightarrow$ the server thread
  - Can insulate changes in tickets across different currencies (Fig 9)

Drawbacks

- Does not guard against starvation: probabilistically, starvation won't happen
- Hard to transfer the notion of tickets to I/O

Linux Scheduler

Link: https://www.ece.ubc.ca/~sasha/papers/eurosys16-final29.pdf

Background

- Performance bugs are hard to spot, identify, and solve, because they don't trigger panics/crashes
- Linux *completely fair scheduler* (CFS):
  - Each CPU core has a runqueue, CFS periodically load balances tasks across cores
  - Some tasks have lower *niceness* (i.e., higher *priority*), hence longer *timeslice* allowed to run
  - When a thread is done running for its timeslice, enqueue again
  - Hierarchical load balancing: first balance every pair of two cores, then balance across pairs using average load

Contribution

- Identified 4 Linux scheduler bugs:
  1. *Group imbalance* (Slide 14): using average of cores in pair for that pair's load - possible that one is loaded by a heavy task and the other is idle

     ⨀ Solution: use minimum instead of average here
  2. *Scheduling group construction* (Fig 4): each *scheduling domain* contains *groups* that are lower-level domains
     - Each pair of core contains cores
     - Each CPU contains pairs of cores
     - Each group of directly-connected sockets (nodes, CPUs) contain CPUs
     - The whole machine contains groups of directly-connected sockets

     At the last level, groups might overlap - if an application runs on overlapped nodes, will not trigger balancing

   ⊙ Solution: make groups disjoint at the last level
3. *Missing scheduling domains*: just a code bug
4. *Overload on wakeup* (Fig 3): wakeup algorithm only considers local CPU core when a thread unblocks
   ⊙ Solution: wakeup on core idle for the longest time, not restricted to local

Scheduling: Resource Tracking

Resource Containers

Link: https://www.usenix.org/legacy/publications/library/proceedings/osdi99/full_papers/banga/banga.pdf

Motivation

- Resource management is important
  - Want to exert explicit QoS policies
  - Handle DoS attacks
- Scheduling must know which entities are doing work for which clients

Background

- Currently, *protection domain* + *resource principal* are combined in the *process* abstraction
  - One process may handle different work over time
  - One work might be fault-isolated into multiple processes (Fig 6)
  - No kernel tasks accounting (Fig 5): e.g., networking
- Some web servers use dispatcher model, others use event-driven model

Contribution

- Processes for protection domain
- *Resource containers* for resource principal: every resource container covers exactly a specific client/work, including the kernel accounting or across multiple threads
  - Multi-threaded server (Fig 9)
  - Event-driven server (Fig 10)
- Properties:
  - A thread can register itself to and switch to different resource containers over time
  - Multiple threads can be bound to a single resource container
  - Network activity in kernel needs to specify the socket and the resource container ASAP
- Able to defend against SYN-flooding DoS attack (Fig 14): mark client bad when SYN dropped, so the application can bring down that resource container's priority

Drawbacks

- The actual scheduling is not 100% accurate: it would be too costly if we de-schedule a thread every time it switches to a low-priority resource container
- Does not handle I/O share (Fig 11): reads the same 1KB file to ensure it is in buffer cache; More problems if we have batched work

Scheduling: GPU Scheduling

Themis

Link: https://www.usenix.org/conference/nsdi20/presentation/mahajan

Motivation

- Deep learning makes shared GPU clusters popular
- Fig 2 shows that:
  - Most applications are composed of multiple jobs
  - Most applications explore various hyper-parameters for a given model
  - Only about 5% of applications contain about 1 job
- *Sharing incentive* (SI): an application should not run slower on a shared cluster with N apps (T_{sh}) than on a dedicated cluster with $\frac{1}{N}$ of the GPU resources (T_{id})

Background

- Current scheduler - DRF: allocate on resource distribution to minimize application time to completion
  - Does try to optimize SI, PE, & EF
  - Uses only instantaneous resource fairness
  - Fails to consider the long durations of non-preemptive ML tasks
  - Does not take into account placement preference of ML apps
- Current scheduler - LAS (Tiresias): allocate on lease duration to minimize application attained service (#GPUs $\times$ time)
  - Does better than DRF for ML
  - But still does not take into account placement preference (Slide 29)
- All shared cluster schedulers (including Themis) require tasks to provide their estimation of resource demand

Contribution

- *Finish-time fairness* metric: $\rho = \frac{T_{sh}}{T_{id}}$; SI means $\rho \leq 1$ for all apps
  - Ask application agent and its hyper-parameter optimizer to calculate this for us
  - $T_{id} = T_{cluster} \cdot N_{avg}$, where $T_{cluster}$ = observed finish-time of app in whole cluster
  - $T_{sh} = T_{current} - T_{start} + \text{iters\_left} \cdot \text{iter\_time}$
  - iter_time involves a slowdown S to account for placement penalty (e.g., 1.3 for cross-rack)
  - May not work well if the number of competing apps changes dramatically over its lifetime
- Optimization SI objective: $\min{(\max{\rho})}$
  - Strawman approach: whenever there is free GPU, allocate to app with currently highest ρ for lease duration
    - Does not consider placement efficiency
    - Apps can lie with high ρ values
  - Themis: filter $(1 - f)$ fraction of apps with max ρ values and do *auctions* to give the GPUs (apps bid on how much beneficial it would be if I'm given which of these free GPUs); uniformly randomly distribute auction-dropped GPUs to the remaining f fraction of apps
- Fairness results (Fig 9):
  - Other scheduling algorithms have long tails due to placement inefficiency
  - Themis is fair enough ($\rho \leq 1.2$)
  - Themis minimizes maximum ρ compared to others
- Tradeoffs for f and lease time (Fig 19): increasing f limits scheduling choices, hence worse performance

OS Structure: OS Models

THE

Link: https://www.cs.utexas.edu/users/dahlin/Classes/GradOS/papers/p341-dijkstra.pdf

Motivation

- Providing a *multi-programming* environment

Contribution

- *Layered monolithic* OS design:
  - Processes with synchronization: defines a *sequential process*, and use semaphores for synchronization
  - Automatic backing storage allocation: virtual memory
  - Strict system hierarchy (Slide 14): every layer trusts the layer beneath, so can develop from the lower level
- *Semaphores*:
  - `P(sem)` to down a semaphore and continue on `sem.value >= 0`
  - `V(sem)` to up a semaphore and wake waiting process if now `sem.value <= 0`
  - Initialize semaphore to value `1` to implement a mutex
  - *Private semaphore* means only one process will call `P` on it

```
/** Consumer. */
P(mutex);
if (elem on q)
    V(private);     // Enables atomic checking of condition
V(mutex);
P(private);         // If previous condition is false, will block here
get(q, elem);

/** Producer. */
P(mutex);
put(q, elem);
V(private);
V(mutex);
```

- Interesting summary in system research principles

Nucleus

Link: http://brinch-hansen.net/papers/1970a.pdf

Motivation

- Provide basic primitives that allow extensible OS

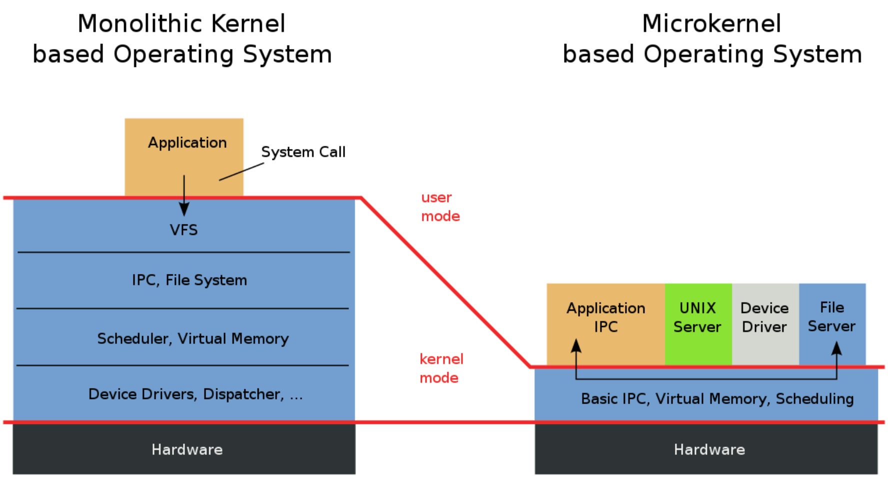
Contribution

- The early work on *extensible* systems and the prototype of *microkernels*:
    - Process interposition: can replace a running process with another (Slide 19)
        - *Internal*: executes interruptible program logic
        - *External*: interprets messages from internal processes and initiates I/O with storage devices - i.e., device drivers
    - Message passing for IPC
        - `send_msg(receiver_pid, msg, &buffer)`: Nucleus will allocate the buffer, and the sender can continue
        - `wait_answer(&result, &answer, buffer)`: sender actually wants to wait for the answer
        - Allows for easy process interposition
        - The buffer helps accelerate the response since it has already been allocated
        - The buffer removes potential deadlocks since buffer is guaranteed to exist; If the device driver dies with a pending `send_answer`, Nucleus will send an "error" answer to the sender

        ```
        /** Client. */
        send_msg(disk, "READ file A @ offset B", &buffer);
        wait_answer(&result, &answer, buffer);

        /** Device driver process. */
        while (1) {
            wait_msg(&sender, &req, &buffer);
            ...
            send_answer(result, answer, buffer);
        }
        ```

- Claims no "built-in" strategy: all OS strategy is implemented in processes' program logic



Drawbacks

- Many processes, so much context switching overheads
- IPC performance becomes critical, and is fundamental to extensible systems

Exokernel

Link: https://cs.nyu.edu/~mwalfish/classes/14fa/ref/engler95exokernel.pdf

Motivation

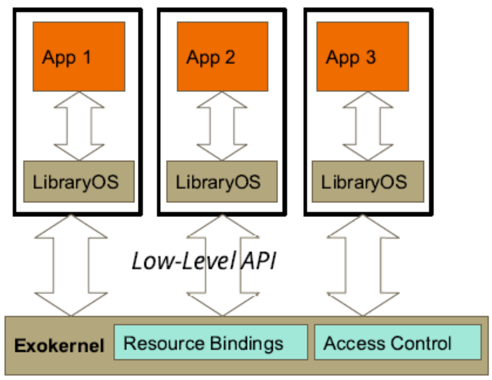
- OS provides high-level abstractions but hides the semantic & performance characteristics of underlying hardware, so may not suit all kinds of applications well

Background

- Roles of OS:
  - Hardware abstraction & virtualization
  - Resource sharing & concurrency
  - Process isolation & protection

Contribution

- Proposes the *exokernel* model (Fig 1; Slide 5): kernel only exposes low-level device APIs and provides access control; Library OS responsible for high-level abstractions and highly-optimized for the application



- Scheduling in this context:
  - Exokernel round-robins all library OSs
  - Library OS handles context switching within it
  - Exokernel uses an *abort protocol* to relinquish resource from a non-responsive library OS
- Networking in this context:
  - Application can install code into the exokernel network surface (*downloading code*, application-specific safe handlers, ASH), e.g., packet filters
  - When packets arrive, filtering happens early so it avoids switching between kernel & library OS every time

Drawbacks

- For performance issues, putting everything in library OSs is infeasible; Sometimes we need to comprise and lean back towards a monolithic model, e.g., the network ASHs

Arrakis

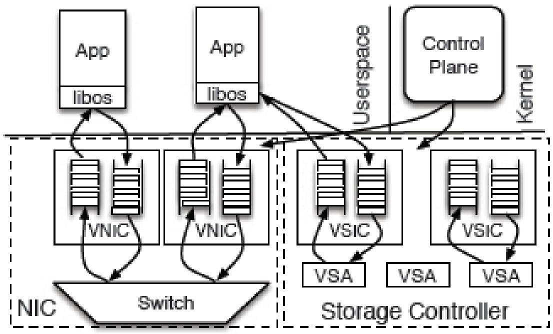
Link: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/peter

Motivation

- Modern I/O devices are ultra-fast, and they provide native support for virtualized multiplexing (on-chip VNICs, VSICs)
- Kernel overhead is becoming too much

Contribution

- A successful *kernel bypassing* design:



  - Removes unnecessary logic out of the critical *data path*
    - I/O processing logic in the application library
    - Device handles multiplexing & I/O scheduling; Requires sophisticated hardware controller support
  - The kernel is responsible for *control plane* operations: naming, access control
    - Device directly notifies the application about new events in queue if it is now running; Kernel is involved only if the application is now not active
- Low latency, high throughput, & good scalability for I/O over fast devices

OS Structure: Disaggregation

LegoOS

Link: https://www.usenix.org/conference/osdi18/presentation/shan

Motivation

- Target environment: datacenters
  - Ultra-fast network & RDMA
  - Processes do not write-share memory, no cache coherence problems
- Provide these properties which current kernel models do not provide:
  - Resource utilization
  - Heterogeneity
  - Elasticity: easy to plug and remove resources
  - Fault tolerance: failing a component should not fail the whole server node

Contribution

- The abstraction of *virtual nodes* (vNodes):
  - A *vNode* is made up of different shares of different physical hardware components
  - *pComponent* - processor node; *mComponent* - memory node; *sComponent* - storage node
  - Each physical component runs its own monitor
- Separates processor with memory (Slide 27):
  - Move the whole virtual memory system to the DRAM side
  - CPU cache on pComps uses virtual address (Sec 4.2; Fig 5):
    - *Homonyms* - multiple different address spaces use the same virtual address for different data

      ⊙ Solution: label cacheline with address space ID (ASID)

        - *Synonyms* - multiple virtual addresses map to the same physical address

          ⊙ Solution: LegoOS does not allow write-share memory, hence does not have synonyms

    - But a kernel runs on pComp and uses physical addresses directly, hence still leaves some physical memory on pComp
    - Adds a small *ExCache* on the pComp side to improve performance
- Separates storage from others (Fig 6):
    - Storage monitors do not maintain any state, e.g., file descriptors; pComp sends full path names
    - sComp uses hash map for fast lookup from full path name to file
    - Each file maps to a particular mComp as its buffer cache
        - Reads first go to the mComp; On a miss, go to sComp and refill the cache
        - Writes / `fsync`'s push stuff from mComp all the way to sComp (4 hops round-trip, slow)

Drawbacks

- Storage throughput results (Fig 9):
    - Random reads do well because network can keep up
    - Sequential reads / writes are bad due to the network overheads / extra hops

OS Structure: OS in HLL

Biscuit

Link: https://www.usenix.org/conference/osdi18/presentation/cutler

Motivation

- Pros of using *High-level languages* (HLLs):
    - HLLs provide much better memory safety and fewer bugs
        - Pitfall: HLLs can still panic on out-of-bound errors
        - Pitfall: HLLs have bugs in their compiler/runtime as well
    - HLLs are much simpler to code
- Cons:
    - Safety tax: bounds, casts, *garbage collection* (GC)
    - GC CPU and memory overhead
    - GC pause time

Contribution

- New HLL kernel from scratch, written in Go
    - Good compiler
    - Easy concurrency
    - Easy static analysis
    - Garbage collector: *stop-the-world* pauses for tens of microseconds $\rightarrow$ harms tail latency
- Handling kernel *heap exhaustion*:
    - C kernels rewind back from an allocation failure (+ "too small to fail"); Cannot do that in Go because Go does not expose failed allocations and implicitly allocates for you
    - Instead, static analysis over all syscalls code to get max memory footprint - reserve that much at syscall entry
- Breakdown of HLL tax (Fig 7:
    1. GC cycles: not much if memory is abundant
    2. Prologue cycles: check stack expansion, ...
    3. Write barrier cycles
    4. Safety cycles
- Observations:
    - Good OS performance is more about tons of optimization techniques, less about HLL
    - Same code path comparison: 15% worse performance
- Rust seems a promising future for HLL OS kernels: compile-time memory safety and no GC

Drawbacks

- Does not include some functionalities/features that Linux has:
  - File permissions
  - Scheduling priority (relies on Go runtime scheduler)
  - No swapping to disk
  - Not NUMA-aware
  - Security features
- Should better discuss how we could improve Go to support OS development
  - Control GC behavior
  - Memory fence support
  - Exposing allocations

OS Structure: Virtual Machines

Disco

Link: http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=1473D91F21DBDF43FEF78259A24F0F2D?doi=10.1.1.103.714&rep=rep1&type=pdf

Motivation

- Enable existing commodity OSs to handle NUMA architectures
- But ended up being a classic VM system design; Challenges of using VMs:
  - Time overhead: syscalls, extra layer of TLB translations
  - Space overhead: OS code, file buffer cache
  - Lost information about resource usage: memory in use, CPU idle thread
  - Handle sharing: most commodity OS require exclusive access to disk

Background

- *cc-NUMA*: cache-coherent non-uniform memory architecture
  - Each CPU socket (node) has its own local memory
  - Accessing other node's memory is uniform, but slower

Contribution

- *Type-1* virtual machine manager (VMM) layer called Disco
  - Syscall procedure:

    User app $\rightarrow$ Disco $\rightarrow$ Guest OS (IRIX) $\rightarrow$ Disco $\rightarrow$ User
  - Memory management: lets TLB store v.a. $\rightarrow$ m.a. translation directly!

    User app $-v.a.\rightarrow$ Disco $\rightarrow$ IRIX $- < vpn, ppn > \rightarrow$ Disco looks up per-guest *pmap* to find out $< ppn, mpn >$ and replaces in TLB $< vpn, mpn >$ $\rightarrow$ User
    - User app - virtual addresses (v.a.)
    - Guest OS - "physical" addresses (p.a.)
    - Disco - machine addresses (m.a.) - actual memory address on hardware
  - Use software 2nd-level TLB to cache translations $< ASID, vpn, mpn >$; Migrating a guest OS to a different NUMA node only requires moving memory content and updating the pmap - IRIX does not know it happened
- Page sharing, i.e., memory deduplication (Fig 4; Slide 21): multiple guests' physical pages of the same content can point to the same machine page
  - Read sharing is easy
  - Write from one guest invalidates its old TLB entry, copies and writes to a new machine page
- Time overhead of virtualization on uniprocessor (Fig 5):
  - Pmake & Database are system-intensive workloads, hence large overhead, big black bar
  - IRIX Kernel time decreases because Disco is now doing zeroing pages, 2nd-level TLB, ...
- Page sharing effectiveness (Fig 6): does a great job on IRIX_text and buffer cache, but not that great for IRIX_data
- Fig 7 shows comparison between IRIX on bare metal vs. Disco opening 8 VMs; Fig 8 shows the power of Disco placing optimal memory to reduce remote memory access time

Drawbacks

- Requires slight modifications to the guest OS code, i.e., it is *paravirtualization*:

- IRIX itself lives in unmapped `kseg0`, hence Disco needs slight modification to IRIX to make it not live in `kseg0` to enable TLB interposition
    - Replace simple syscalls with memory reads/writes to boost performance
    - Disco already zeros pages at allocation for privacy; IRIX does not need to do that again
    - More on Slide 23 ...

VMware ESX

Link: https://www.vmware.com/pdf/usenix_resource_mgmt.pdf

Motivation

- Must run unmodified guest commodity OSs
- Server consolidation & Oversubscription: memory may be overcommitted

Background

- Increasing concern and popularity in virtual machines
    - Modularity
    - Equivalence: exactly the same virtual environment as the underlying hardware
    - Safety: isolation across VMs
    - Performance: cannot show major decrease in speed
    - Server consolidation
    - VMMs got the name *Hypervisor*
- *Type-1* v.s *Type-2* hypervisors:
    - Type-1: Hypervisor running directly on hardware, no host kernel

      Examples: Disco, ESX, Xen, ...

        - option a) - hypervisor implements all the device drivers
        - option b) - a "domain-0" OS provides drivers implementation, hypervisor sends "domain-U" OS device requests to the "domain-0" OS
    - Type-2: Host OS supports hardware-assisted virtualization, VMM is just a supported mode / kernel module

      Examples: KVM

- *Popek/Goldberg Theorem* (Slide 12): "sensitive" instructions vs. "privileged" instructions
    - Privileged instructions are those that will trigger a trap from user mode
    - Sensitive instructions are fundamental instructions updating shared host state
    - A system is virtualizable ONLY IF the user never calls sensitive instructions directly w/o trapping, i.e., sensitive instructions must form a strict subset of privileged instructions!

Contribution

- Assumes hardware TLB, so needs software shadow page table managed by ESX hypervisor
- The *double paging* problem: hypervisor pages out dirty page P from VM to hypervisor swap area, but then VM runs and guest OS decides to page out P to its virtual disk - results in P gets read back and written to a different area again

  ⊙ Solution: *ballooning* - Insert a balloon device driver in guest OS; The driver asks for more memory and sends to hypervisor the allocated ppn's to act as taking memory back from that VM

    - If guest memory is scarce, it is the guest OS that decides which pages to swap out
    - Avoids double paging
    - Negligible overhead compared to guest OS just configured to have that much memory (Fig 2)
- Similar memory deduplication technique but uses content-based hash fingerprints like data deduplication (Fig 3); In contrast, Disco modified IRIX code of `bcopy()` to do copy-on-write
    - Periodic scan of memory and maintain a hash table
    - By default, a frame is set to NOT doing COW - a hint frame hit on non-COW page hence must do full comparison of page content for confirmation
    - Effective even with just 1 VM (Fig 4)
- Memory QoS: allocate memory to guests proportional to shares, but can exceed than when hardware underutilized
    - If contention, do *min-funding* revocation from guest with $\min\left(r = s/P\right)$
    - Sometimes, completely fair isn't wanted:
        - Lower system-wide performance

        - Idle clients may hoard resources
        - Busy clients get more benefit from resource

    ⊙ Solution: *Idle memory tax* $r = S/(P \cdot (f + k(1 - f)))$, where f is the active fraction; f accounted through periodic *sampling* that invalidates a small few pages to let the hypervisor know when they get re-accessed

ReVirt

Link: https://web.eecs.umich.edu/virtual/papers/dunlap02.pdf

Motivation

- Improving security by analyzing attacks after they occur is important
- Previous logging mechanisms not sufficient:
    - Assumes OS is trustworthy
    - Log coverage is not complete

Contribution

- Encapsulate target system into a VM, then place logger below the VM to log everything happened
    - Using a Type-2 hypervisor + UMLinux guests
    - Syscall procedure: host OS provides virtual modes for guest kernel vs. guest user

      User $\rightarrow$ Host OS checks virtual mode bit $\rightarrow$ Guest OS $\rightarrow$ User
- The *trusted computing base* (TCB) becomes just the host kernel + logger and nothing else
- Not all external inputs need to be logged:
    - Reading from a disk is assumed deterministic (disk available at replaying)
    - Network data is huge and becomes problematic

      ⊙ Possible solution: *cooperative logging* to log the sender as well, then those become the sender's output
    - At replaying: signal feeding point is identified by the PC reg + branch PMC counter

      Sensitive instructions like `rdtsc`, `rdpmc` are problematic...

Drawbacks

- Since using a host kernel, it is arguable saying the TCB is becoming smaller - one reasonable argument is that guest OS only uses 7% of host syscalls