Using Synchronisation

Concurrency strategy: specify code to parallelise, ensure safety, reduce contention, reduce overheads.

Kernel threads: OS managed, expensive, balanced across CPUs, handle blocking.

User threads: language implemented, multiplexed, fast creation, don't handle blocking or parallelism.

Commands: things to be done.

Executors: things that arrange execution of commands:

- Direct - synchronous
- Locked - synchronous, one at a time
- Queued - synchronous, one at a time
- Pooled - asynchronous, bounded # of threads
- Threaded - asynchronous, unbounded # of threads

Thread pools have a number of threads and a shared work pool. Many options to deal with queue overflow (blocking, exception, synchronous execution, discard existing item). Work items must use asynchronous I/O.

Can use per-worker thread queue to allow task thread-affinity for I/O resumption. Queue underflow motivates work stealing pools.

Two lock concurrent queue: head always points to a dummy node, separate locks protect the *push_tail* and *pop_head* operations.

Reducing contention:

- Confinement: guarantee some objects remain thread-local (no locking)
- Accept stale/changing data
- Copy-on-write
- Increase locking granularity: but beware of deadlock and overhead

Implementing Synchronisation

Coherency requires that there is at most one dirty version of a cache line.

Uniprocessor: invalid/clean/dirty.

MSI: modified/shared/invalid with new exclusive read signal.

MESI: MSI+exclusive for cache lines clean but private to reduce bus noise upon write.

MOESI: MESI+owned where lines may be modified despite other CPUs having the line shared (must transmit changes).

ccNUMA:

- CPUs and local memory grouped into nodes
- Per-node directories provide a serialization point, track which processors have dirty/clean copies of lines
- Read/write requests are issued to the memory block home node which can forward requests to the actual owner (worst case 3 hop miss).

Modern processors execute instructions out of order. Memory barriers are needed to enforce ordering.

Sequential consistency: every operation happens in program order from the point of view of remote processors.

Spin locks can be improved by having an inner loop to wait for unlockness to become visible (reducing interconnect traffic), but this will still lead to a stampede upon unlock.

Queue based spin lock: threads spin on separate locations with a linked list of waiters:

Lock:

```
QNode mine = new QNode(true, null);
QNode seenTail = tail;
while (CAS(&tail, seenTail, mine) != seenTail)
{ seenTail = tail; }
if (seenTail != null)
{ seenTail.next = mine; }
else
{ mine.blocked = false; }
```

Unlock:

```
if (mine.next != null)
{ mine.next.blocked = false; }
else {
if (CAS(&tail, mine, null) != mine) {
QNode seenNext;
while ((seenNext = CAS(&mine, next, null)) == null)
{}
seenNext.blocked = false;
}}
```

Big reader locks: per-CPU locks, readers acquire the lock for their CPU but writers must acquire them all.

Mutexes can cause deadlock, priority inversion and preemption or termination while holding locks.

Non-blocking data structures ensure that the system as a whole can still make progress if any finite number of threads in it are suspended.

- Wait freedom: per-thread progress bound
- Lock freedom: system wide progress bound (i.e. within a bounded number of cycles some thread in the system will get work done)
- Obstruction freedom: system wide progress is bound if threads run in isolation (i.e. are waiting on different locations)

Operations are linearizable if for events A and B such that A $<$ B in the global clock ordering and are not overlapped in time then that's always the order observed between the threads. Serializability does not make reference to a global clock.

Routing

Forwarding is the process of determining the next hop, routing is that of establishing end-to-end paths.

Link state:

- Topology information flooded within routing domain
- Best end-to-end paths are computed locally
- Best end-to-end paths determine next hops
- Minimizes some notion of distance
- Works only for shared and uniform policy
- OSPF: can have hierarchical areas with unique LS databases, routes exchanged between areas by distance vector

Distance vector:

- Ever router knows a little about network topology
- Best next-hops are chosen by each router for each destination
- Best end-to-end paths result from composition of next-hop distances
- Does not require any notion of distance
- Does not require uniform policy
- RIP: distance is hop count
- Can have a problem with counting to infinity, solved by letting $\infty = 16$

Link state converges faster but requires more memory, CPU and messages. Both protocols can induce transient forwarding loops during convergence. Distance vector has fewer policy requirements.

BGP

IP forwarding table is logically a list of destination, next hop, interface tuples. They can be constructed by administrator (statically) for control or by routing protocols (dynamically) which adapts to network changes and scales better. However, fast dynamic adjustment can result in routing table volatility due to bursty traffic characteristics.

Classful IPs: 24, 16, 8 bits of host addresses based on top bits of address. CIDR: explicitly entered network prefix mask.

Autonomous system numbers: centrally allocated to all EGP participants. Graphs of these do not show topology, but each AS is typically connected internally.

BGP records the AS path of a route so you can reject routes which contain your own AS. Traffic may not necessarily follow the AS path.

Deaggregation due to multihoming may contribute to table growth (same prefix is announced by multiple upstream providers).

Transit ASes allow traffic with neither source nor destination within the AS to flow across the network. Non-transit ASes allow only traffic originating from the AS or traffic with a destination within the AS to be transported. Peers provide transit between customers but not between other peers.

Peering reduces upstream transit costs, can increase performance and may be the only way to connect to the Tier 1 internet, but you don't want to peer yourself because peers are your competition and you would rather have them as customers.

Parents typically filter the announcements of their customers to prevent black-holing (announcement of unreachable routes by ASes). This is implemented by route import policy.

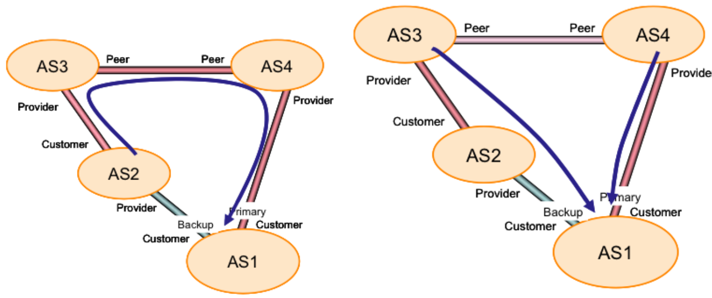
When exporting routes: customers get everything, peers/providers get customers/ISP routes. This is implemented via route export policy.

Next hop attribute is updated every time a route announcement crosses an AS boundary to the address of the border router that announced the route.

Routes are tagged with a list of community values. Customers will typically fiddle the communities to modify the local (routing) preference of upstream networks which tiebreaks identical length AS paths. Local preference can be used to implement customer backup links for outbound traffic, but for inbound traffic must ensure you export routes with padded AS paths on the backup link!

Hot potato routing: export routes via the egress point that your IGP claims is the closest to the ingress point to reduce costs. Customers tend to want their providers to carry the bits however, so they export multi export discriminator attributes in their routes which reflect their IGP costs so the upstream provider can consider these costs before their own IGP distance.

BGP wedgie: interaction of local policies allows routings which are consistent but unintended. No single group of operators necessarily has all the information available to debug the problem. "Full" wedgies may not be resolved by simple bouncing of an aberrant link.



Inter-domain routing is often asymmetric due to hot-potato routing. Latency is not a metric space.

The goals of fast convergence, minimal updates and path redundancy are at odds. No one knows where all the updates come from: it may be path exploration, IGP instabilities being exported by routes directly, BGP sessions being reset due to congestion or MED oscillation exporting internal instability. Used to be aggravated by Cisco's stateless ("just in case") withdrawals.

MEDs are only comparable within ASes, so the way they are used means that BGPs distance comparison is not even a partial order on routes. MED oscillation can lead to a constant churning of routes.

Route flap damping: routes are given a penalty for changing. If their total penalty exceeds the suppress limit, the route is dampened. While it is not changing, the penalty decays exponentially until it reaches the reuse limit. However, you may see spurious changes to a route due to a single announcement arriving via multiple paths! This tends to punish small updates for well-connected destinations.

Rate limiting: only publish routes every interval: currently set to 30 seconds. This may delay convergence if it is set too low or too high.

Distributed Shared Virtual Memory

Shared memory: easy to use but suffers from race conditions and synchronisation cost

Message passing: gives control, protection and performance

Idea: each page has a home processor and can be mapped in across the network. Centralized page manager runs on a single processor and maintains *owner(page)* and *copyset*(*page*). On read fault:

1. Reader contacts manager with request, forwarded to owner
2. Owner sends the page to reader and reader acknowledges to the manager, who adds them to the copy-set

On write fault:

1. Writer contacts manager with request, manager invalidates copy-set with a broadcast
2. Manager contacts the old owner who relinquishes the page to the writer who acknowledges to the manager

Can spread managers out across multiple processors and even make the owner the manager. In this case, may maintain per-processor hint of who any given page is owned by, updated on forwarding or invalidation.

Improved with weaker memory consistency than sequential consistency e.g. release consistency (must *acquire* and *release* for synchronisation). May label pages as e.g. private (ignore), write-once (just service reads), read-mostly (owner broadcasts updates), write-many (release consistency and buffering) or synchronisation (strong consistency). Lazy release consistency is best: updates not on release but on next acquire.

Can always do better with explicit messages, and making it work well increases complexity beyond that of message passing. Used on non-cache-coherent NUMA systems though.

Other Stupid Virtual Memory

Persistent virtual memory came up due to a desire for orthogonal persistence. Most virtual memory is backed by non-volatile storage anyway.

Multics: users saw a large number of orthogonal linear regions ("segments") of virtual address space which was their store. Segments were created and named by users.

Technical limitations of 32 bit pointers resulted in pointer swizzling. All persistent data uses persistent pointers, but upon access to the page these are rewritten by the system to point to newly allocated but invalid real pages that would again be handled specially upon access.

Recoverable virtual memory applies transactional semantics to virtual memory. Typically just implement atomicity and some durability (other characteristics expensive):

1. *begin_transaction(rmode)*, where *rmode* specifies how much you care about crashes
2. *set_range(t, base, size)* which prompts a copy of the region which is added to the undo log (allows aborts)
3. *end_transaction(t, cmode)* causes synchronous write of all ranges to the redo log: this does not care about consistency
4. When the redo log gets full log contents is reflected to external segments and the log is truncated!!

LRVM involves three copies of data and requires expensive writes. Rio Vista uses a battery backed NVRAM file cache to store an undo log. Due to durability no redo log is necessary, reducing copies required. When a machine reboots the NVRAM contents are flushed to disk.

Capabilities

Capabilities are protected names for objects: possession is necessary and sufficient for object access. They must be unforgeable and supplied by the system, and can only be manipulated in a restricted set of ways. They allow more flexible (non-hierarchical) delegation.

In CAP, capabilities consisted of a base, limit and access code. Had C-type segments which were the only ones allowed to transfer to/from capability registers. Loading was done implicitly whenever a capability is referred to. TCB had D and C type capabilities on same segment.

Domains of protection are the sets of capabilities to which a process has access. They are entered and left via special instructions. This gives rise to a hierarchy of control but not of protection. A hierarchy of protection can be realised by specifying capabilities relative to some other parent capability.

Microkernels

Microkernels increase modularity and hence allow you to take advantage of SMP, reduce rebooting and isolate crashes. They rely on IPC systems, typically based on *ports* (resource references) and *messages* (the unit of communication). In Mach these are secured via capabilities: every port had exactly one read capability and many write capabilities, and capabilities can be sent via ports.

Microkernels suffer from many kernel crossings (worse locality, more large block copies). L3/L4 tried to rectify this with a set of very fast primitives (recursive address space construction, threads, IPC, unique identifier support). Threads execute with an address space (no separate process abstraction) and IPC is message passing between address spaces.

EROS is a persistent software capability microkernel. Persistence implemented via circular log with checkpointing. It got performance by translating a lot of capability infrastructure to things that could be handled by the TLB (which acted as a sort of capability cache).

Virtual Machines

Disco attempted to run commodity OSes on ccNUMA. Other benefits: fault tolerance, sharing between OSes, run OSes specific to workload. However x86 is not virtualizable so the commercial spin-off uses binary rewriting to manually insert traps. Physical to machine address mapping is realized via shadow page tables.

Denali/Xen use the virtual machine monitor as an isolation kernel (fairness and security through lack of sharing). The OS is rewritten to be virtualizable (para-virtualization) by using events instead of interrupts and hypercalls instead of unvirtualizable instructions. Xen charges for resource usage to discourage denial of service attacks. Asynchronous ring buffers used for network and disk transfer.

VMMs are popular because of increased security, small OS static size compared to memory and desire for flexibility.

Extensibility

Desirable to fix mistakes, support unanticipated features or hardware, efficiency and individualism. Can also be used to implement monitoring and allow for better system/application integration (e.g. for soft real time applications).

Can provide everyone with a virtual machine but does not allow layering violations.

Kernel-level schemes can be based on proof carrying code or sandboxing. With PCC check that the proof says that the code cannot violate the safety policy. However, requires formal specification/semantics/language/algorithms/method for generating proofs. With sandboxing transform untrusted code into safe code. Software fault isolation inserts instructions to perform bounds checking around memory accesses and detects unsafe instructions. It is difficult to do this right (e.g. detect stack modification) and it causes code bloat. VINO combined this with resource quotas, timeouts and transactions on kernel data structures. Isolation can also be enforced via language level safety, as in SPIN: however, this cannot deal with locking policies and lack of termination.

User-level schemes can be based on microkernels or exokernels. Exokernels separate the concepts of protection and abstraction: abstractions deny optimizations and discourage innovation. This is implemented via a per-application "library" operating system that works with real resources being multiplexed and initialized by the kernel.

Databases

Minimize I/O, maximise concurrency. Use write-ahead log for reliability.

Fixed format records depend on schema for interpretation, variable format records are self describing. Can store records directly in disk blocks sequentially by primary key.

Spare index: smaller, so more of it fits within memory and it's better for item inserts and deletes of items not in the index.

Dense index: can do an existence check without accessing blocks and needed for indexes on secondary keys but large. Indexes could just store (small) block pointers for a range of items rather than per-item record pointers.

B+ trees: all nodes have n keys and $n + 1$ pointers. Non leaf node pointers point to nodes with key values $<$ right key but

$\geq$ left key. Leaves point directly to a record. All leaves are at the same depth (balance) i.e. we store at least $\lfloor \frac{n+1}{2} \rfloor$ data pointers in leaves and $\lceil \frac{n+1}{2} \rceil$ in non-leaves. Lookup time for N records is $\Theta(log(n)log_n(N))$, so for a slow CPU choose a low n (less binary search) and for a fast one choose high n (less disk seeks). Upon insertion may need to recursively split from leaf to the root. Upon deletion may need to either redistribute keys or coalesce siblings (but most implementations don't coalesce due to hard concurrency problem).

Postgres: set-oriented POSTQUEL query language, small number of concepts and standard control flow but large memory footprint. Uses no-overwrite storage manager: transaction state just becomes a flag. Abort and recovery are very cheap and you can do historical queries. However, records must be flushed on commit, multiple (time varying) indexes may be required and you need to flush the log to write-once media periodically. Time travel queries turned out to be hard to express.

OSes are not suited to DBMS. OSes do extra data copies to/from disks. Typical LRU replacement policies conflict with linear/cyclic access to blocks and random access with zero probability of re-reference: want MRU really! OSes have no support for synchronous reorder barriers for disk writes. Can get double paging effect due to interaction between userspace buffer and VM.

Distributed Storage

Scalability, location fault tolerance, mobility of access, centralised data management.

NAS distributed storage at the file-system level. SAN distributes storage at the block level (over fibre channel).

NFS: client/server RPC protocol. Initially stateless with idempotent requests: good for recovery but synchronous disk write sucks and cannot help client caching. Version 3 allowed asynchronous writes with explicit commit, version 4 total rewrite to be stateful (e.g. explicit open and close) has compound operations etc.

AFS is a purely remote protocol with versioning to support caching being separate from consistency. Supports live replication and relocation of volumes! Coda is an evolution that supports disconnected operation by caching a "working set" locally and integrating upon reconnect.

LBFS designed for low bandwidth operation. Files are divided into chunks based on the occurrence of a particular Rabin (incremental) fingerprint value. Each chunk is hashed and if clients want to write file portions they just transmit the hashes of the relevant chunks, can avoid transfer if they match with remote chunk database. Hash collisions are ignored in practice!

Serverless file systems distribute data across all nodes. xFS has clients, cleaners, managers and storage servers. To read a file, you lookup its manager in a globally replicated map, contact the manager with request, get redirected to cache or disk. To write file, obtain write token from manager and append all changes to log. When you hit a threshold, flush to stripe group. Fast due to parallelism (striping) for large writes and co-operative caching. Managers are replicated for fault tolerance. JetFile uses scalable reliable multicast to multicast requests for (versioned) data and hopefully receive a response. Write-on-close semantics bumps version number, with the client becoming the server for the new version.

SAN file systems can be asymmetric (use a synchronized metadata manager, clients access data disks directly) or symmetric (clients access data and metadata directly and use distributed locks to synchronise). Symmetric systems are more scalable and fault tolerant.