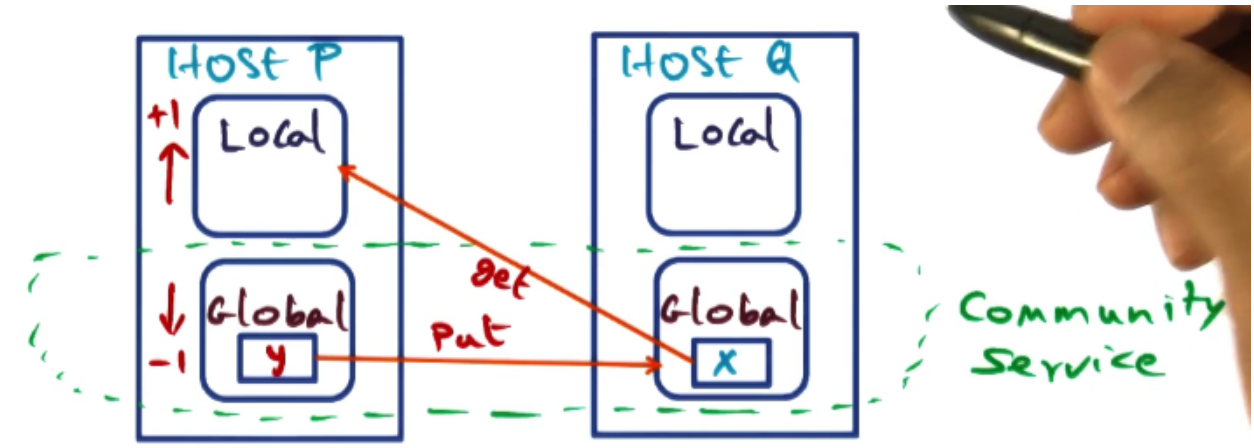
Distributed Subsystems

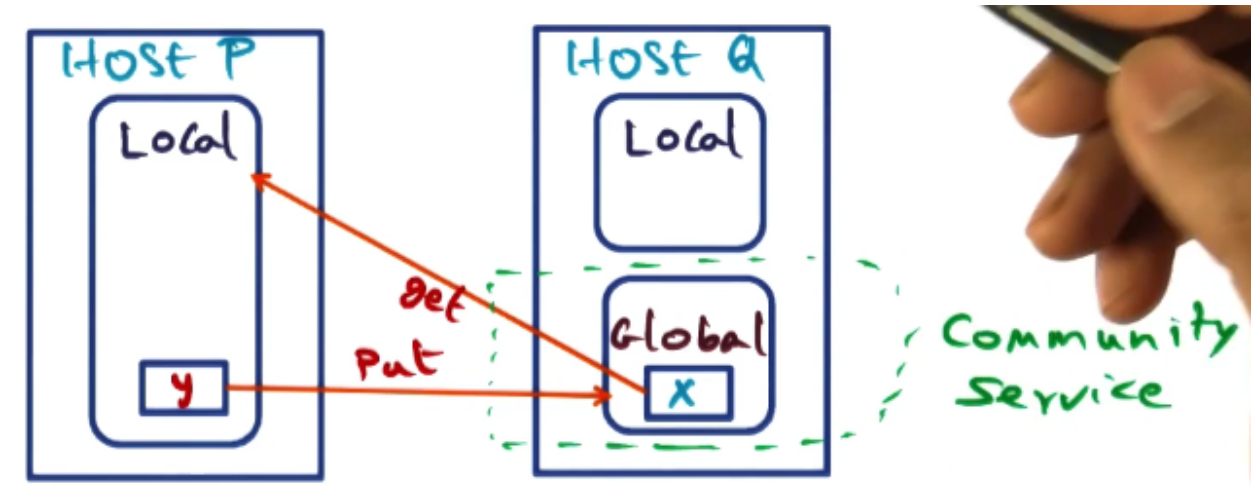
Global Memory Systems

* Global Memory Systems Introduction
    - Services discussed are interesting, but not as important as the technological insights they offer on constructing distributed services
    - GMS: How can we use peer memory for paging across LAN?
    - DSM: Can we make the cluster appear like a shared memory machine?
    - DFS: How to use cluster memory for cooperative caching of files?
* Context for Global Memory System
    - Virtual address space of a process is much larger than physical memory
    - Memory pressure
        + Different for each node
        + How to use idle cluster memory?
        + Remote memory access faster than disk
    - Memory manager
        + VA -> PA OR disk
    - GMS
        + VA -> PA OR cluster memory OR disk
        + Integrates cluster memory into local memory
    - Paging through the LAN might be faster than going to disk due to increases in network speeds
        + Only for reads, doesn't affect writes
        + Dirty copies of pages must be written to disk, so even if a node crashes, no data is lost
        + When a page gets evicted from local memory, GMS goes to peer memory instead of disk
* GMS Basics
    - "Cache" refers to physical memory (i.e., DRAM) not processor cache
    - Sense of "community" to handle page faults at a node
    - Physical memory of a node
        + Local: Working set (needed for myself)
        + Global: Spare memory (portion that can be used for other nodes)
        + Split is dynamic, responds to memory pressure
    - Two states for pages
        + Private to local node
        + Shared across nodes (if a computation is distributed)
    - Coherence for shared pages is an application issue, not GMS's problem
    - GMS uses an LRU replacement algorithm, but across the entire cluster
        + "Globally oldest page"
* Handling Page Faults - Case 1
    - Common case
        + Page fault for X on node P
        + Hit in global cache of some node Q
    - Process running on P faults on page X; X is in cache of host Q
        + Q sends page to P
        + P adds X to its local set (+1) and reduces its global set (-1)
        + P sends oldest page Y to Q so it can host it in its global cache
    - Memory pressure on P is increasing, Q remains unchanged



Page Fault Case 1

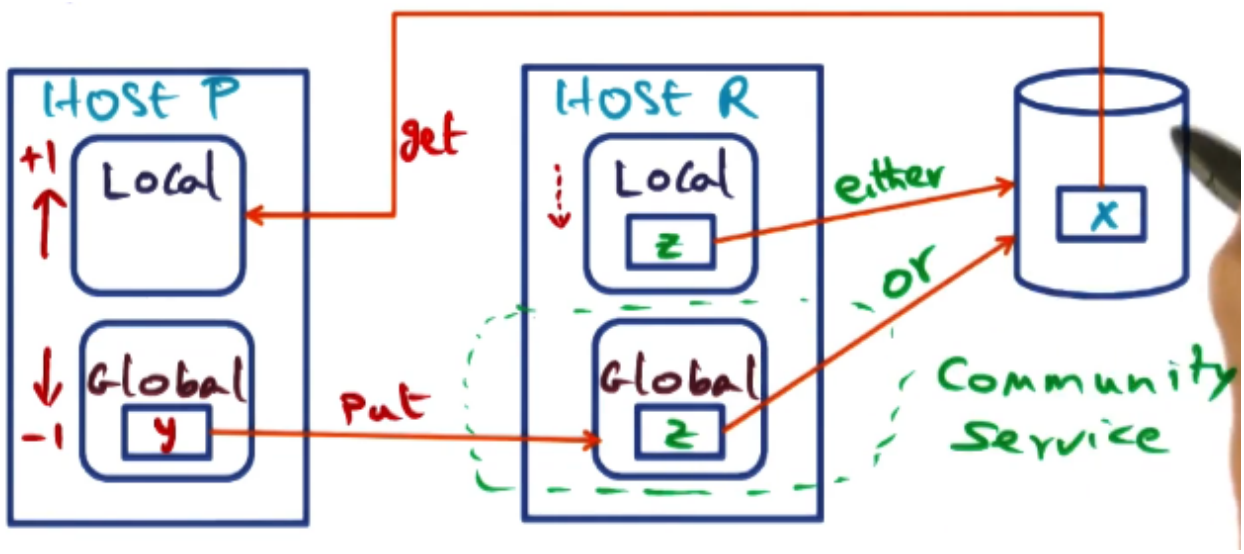
```
* Handling Page Faults - Case 2
    - Similar to case 1, but all of P's memory is being used locally
        + Page fault for X on node P
        + Must evict a page from working set to make room for missing page
        + Swap LRU page Y for X
    - Host Q doesn't change the sizes of its local/global caches
```



Page Fault Case 2

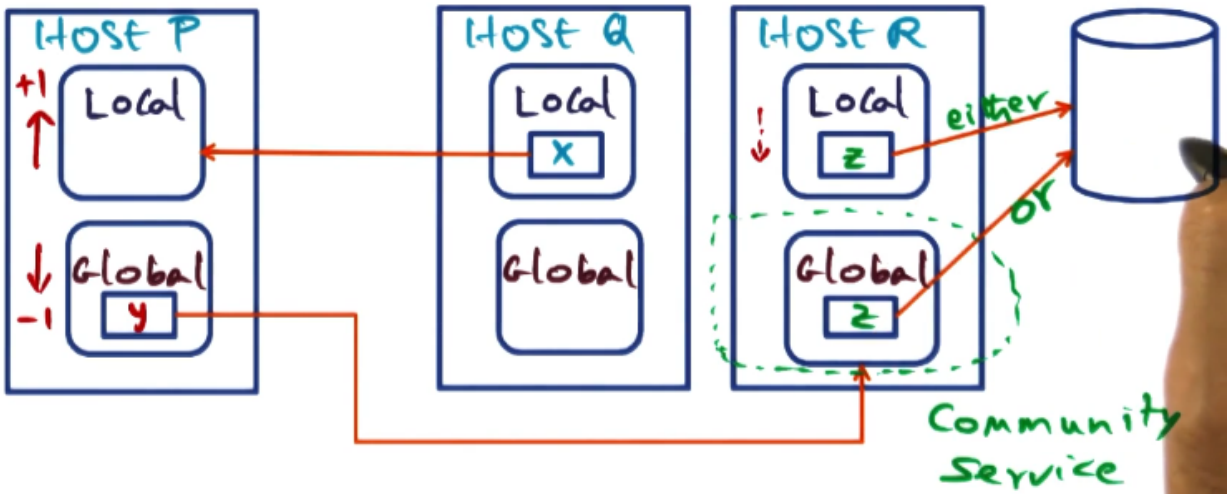
```
* Handling Page Faults - Case 2
    - Page fault for X on node P, but page not in cluster
    - P goes to disk to get X
        + Increases its local, decreases its global
        + Node P sends page y to R
    - When node R receives Y, it must write a page to disk
        + Page z on node R is the globally oldest page
        + R must write Z to disk to make room if Z is dirty
        + If Z is clean, simply overwrite
        + If Z is in the global cache, there is no change to R's global/
```

```
        local memory allocation
        + If Z is in the local cache, local -1 and global +1
```



Page Fault Case 3

```
* Handling Page Faults - Case 4
    - In cases 1-3, the faulting page was private to the node
    - Faulting page actively shared
        + Page fault for X on node P
        + Page in some peer node Q's local cache
    - Want to make a copy of page X on host P so the faulting process has
    access to the data
        + Local +1, global -1
        + Node P evicts Y to be put into peer memory (doesn't have to be LRU)
        + Send Y to R which has the globally oldest page
    - Node Q simply sends page X to node P
    - Node R receives page Y from node P
        + Writes LRU page Z to disk
        + If Z is in local cache, local -1 and global +1
        + If Z is in global cache, no change
    - If coherence needs to be maintained for copies of the same page across
    nodes, the application must maintain this
```



Page Fault Case 4

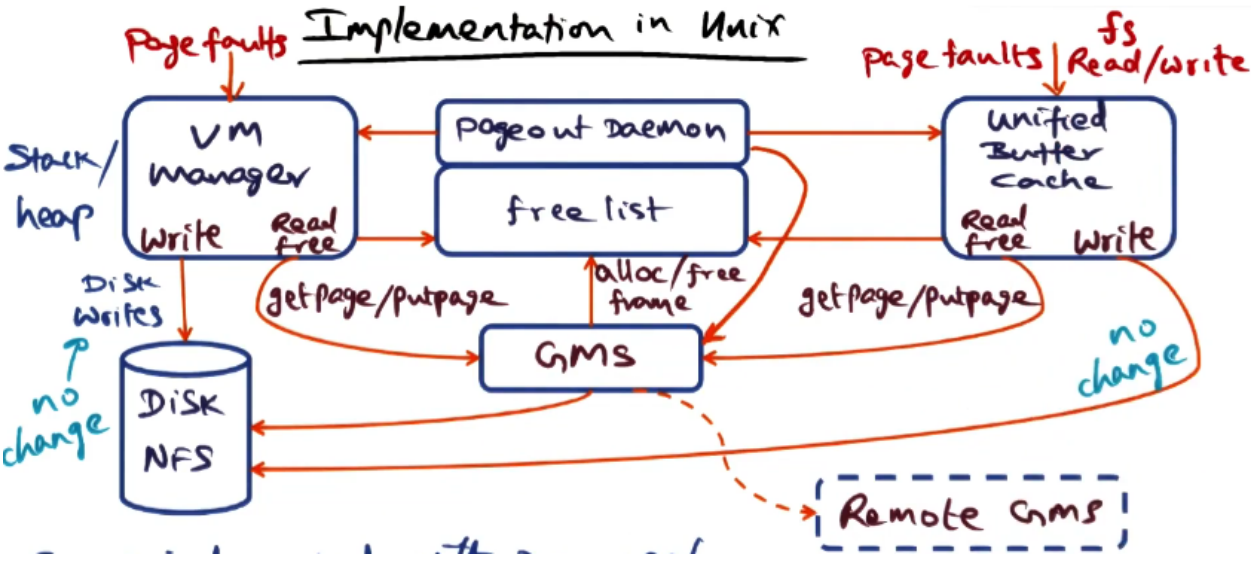
* Local and Global Boundary
    - Activity on Node R depends on if the LRU page comes from the local or global portion of memory
        + If in local, L-- and G++
        + If in global, no change

Faulting Page X	Faulting Node P	Node Q with Page X	Node R with LRU Page
In Q's global	L++ G–	No change	No change
In Q's global, G=0	No change	No change	No change
On disk	L++ G–	Not applicable	L– G++
Shared with Q	L++ G–	No change	L– G++

* Behavior of Algorithm
    - If a node is idle, its working set will continue to decrease as it accomodates more requests from peers
        - Eventually, its local set can become 0 and it becomes a memory server for peers on the cluster
* Geriatrics
    - Age management is the most important aspect of implementing GMS
    - Want management work to be distributed
    - Epoch parameters
        + T max duration (order of seconds)
        + M max replacements (orders of thousands of replacements)
    - Manager rotates every epoch (after T or M occurs)
    - Each epoch
        + Send age info to initiator
        + Receive {minAge, wi for all i}
        + minAge is smallest age for which pages won't be replaced; if a page is older than minAge, it is part of the M replacements
        + w is the weight parameter (percentage of pages to be replaced)
        + Each node receives weight for all other nodes
    - Initiator is trying to predict the future based on current heuristics
    - Management function shouldn't be statically allocated to a single node
    - If a node has weight 0.8, this means 80% of the page replacements will

come from this node
    + This means it isn't very active (lots of old pages)
    + Maximum weight determines next manager
- Action at a node on page fault
    + Page y eviction candidate
    + Age(page y) > minAge -> discard
    + Age(page y) < minAge -> send to peer Ni
    + Ni is selected based on the weight distribution; should be a node with a higher weight, but not necessarily the highest
- Approximates a global LRU; computing the true global LRU would be prohibitively expensive
- "Think global act local" means use local information to inform global decision making

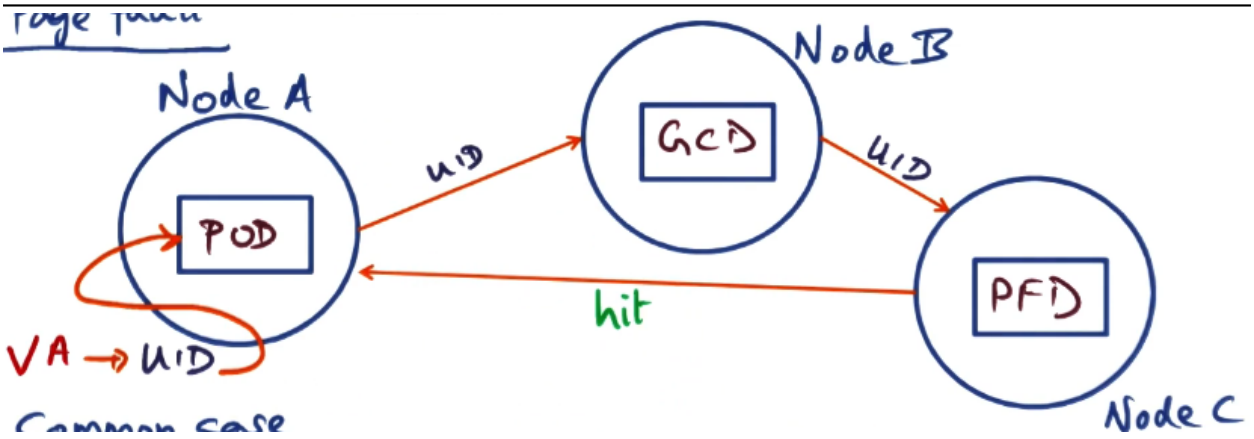
* Implementation in Unix
- VM Manager: Manages virtual to physical address mappings, manages page faults for processes
    + Can read and write to disk
- Unified buffer cache: Cache used by filesystem
    + Serves as abstraction for disk
    + Handles page faults for memory mapped files
    + Can read and write to disk
- Pageout daemon: What are pages that can be swapped to disk?
    + Runs occasionally
- GMS was integrated into the operating system (DEC OSF/1)
    + VM manager and unified buffer cache go to GMS instead of disk for reads
    + GMS knows whether a particular page is present in remote GMS
    + Does not affect how disk writes function
    + Access to anonymous pages and filesystem mapped pages go through GMS on reads
- Need to collect age information to approximate global LRU
    + For unified buffer cache, can intercept calls from OS to collect
    + VMM is more complicated; memory access is happening in hardware on CPU; OS does not see individual memory accesses
    + Reads and writes a process is making are anonymous as far as the operating system is concerned
    + GMS has a daemon that dumps information from the TLB (TLB contains information about recently accessed pages)
- Pageout daemon sends to GMS when discarding pages so GMS can move it to a different node



GMS Implementation

```
* Data Structures
    - VA -> UID (universal ID)
        + IP Address: Which node the virtual address emanated from
        + Disk partition: Where is the page on disk
        + Inode: What is the inode corresponding to this page
        + Offset: Offset within page for VA
        + Derived from VMM and UBC
    - Page Frame Directory (PFD)
        + Like a virtual address (UID -> PFN)
        + Physical frame number (PFN)
        + Similar to page table on caching node
        + States: local private/shared, global private, on disk
        + Global is only private because the pages are always clean
    - Global Cache Directory (GCD)
        + Cluster-wide partitioned hash table (distributed data structure)
        + Given a UID, says which node has PFD for this UID (UID -> Ni)
    - Page Ownership Directory (POD)
        + Given a UID, which node has GCD corresponding to this UID
        + Replicated on all nodes
        + POD doesn't change too often, but when it does, need to modify on
        all systems (only happens if a node goes up or down)
* Putting the Data Structures to Work
    - On page fault
        + Convert VA to UID
        + Go to POD (replicated, up to date information)
        + POD says which GCD to go to
        + GCD says which PFD to go to
        + PFD gives the PFN or says the page is on disk
    - Common case
        + Page not shared -> A and B are the same
        + Page fault service is fast
        + This eliminates the extra message between POD and GCD
    - Uncommon case: Miss on PFD
```

        + POD changing due to node addition/deletion
        + Recomputing the distribution of the UIDs
        + If POD evicts the page, it tells the GCD that it's been sent to a different node, but this may not have occurred yet, so the PFD might not have the data
    - On page eviction (Paging daemon)
        + When the freeList falls below a threshold...
        + putPage of the oldest pages on this node to a different node
        + Update GCD, PFD for the UIDs



Data Structures

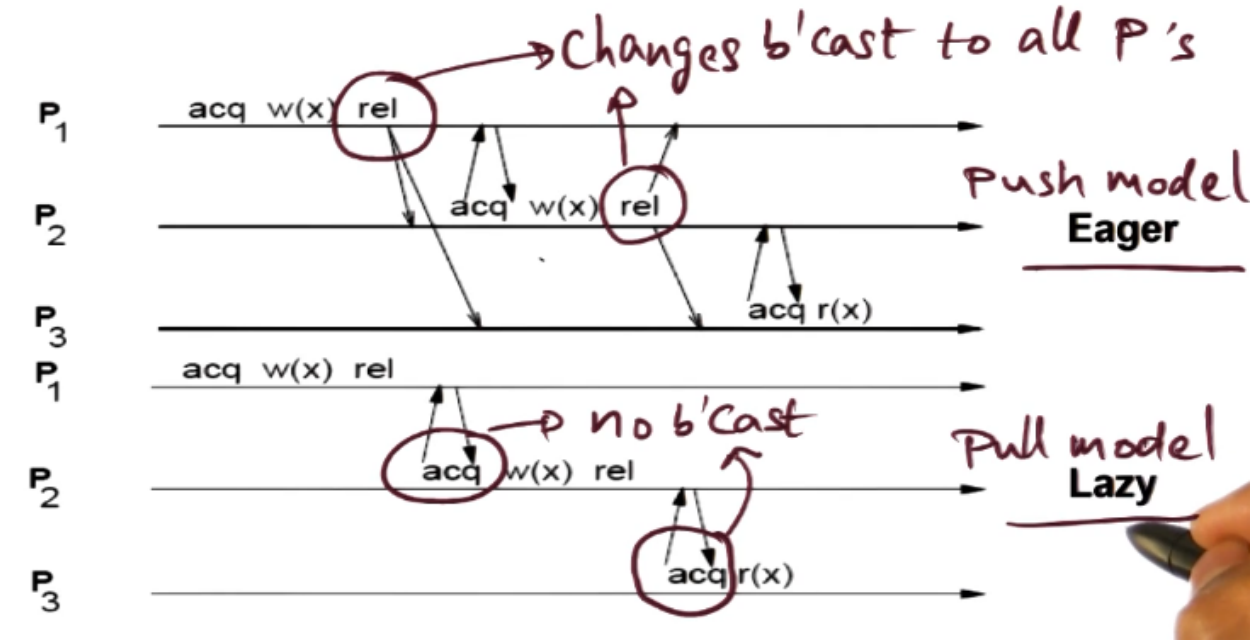
* Global Memory Systems Conclusion
    - Working out the details to go from an idea to an implementation is a nontrivial exercise when creating distributed systems
    - GMS is an interested idea, but may not be applicable to the intended use case (workstations on a LAN)
        + Might be useful in a modern data center

Distributed Shared Memory

* Distributed Shared Memory Introduction
    - Create an operating system abstraction that provides an illusion of shared memory to the applications
* Cluster as a Parallel Machine (Sequential Program)
    - Automatic parallelization: Instead of writing an explicitly parallel program, write a sequential program and allow somebody else to do the heavy lifting of identifying opportunities for parallelization
        + Implicitly parallel program
        + Onus of the tool (compiler) to parallelize
        + Limited potential for exploiting available parallelism
        + Data accesses need to be fairly static (determinable at compile time)
    - High Performance Fortran (HPF) is an example of this
        + Directives for data/computation
        + Data parallel programs
        + Maps computation to different nodes of a cluster
* Cluster as a Parallel Machine (Message Passing)
    - Application programmer will implement the algorithm in an explicitly parallel way

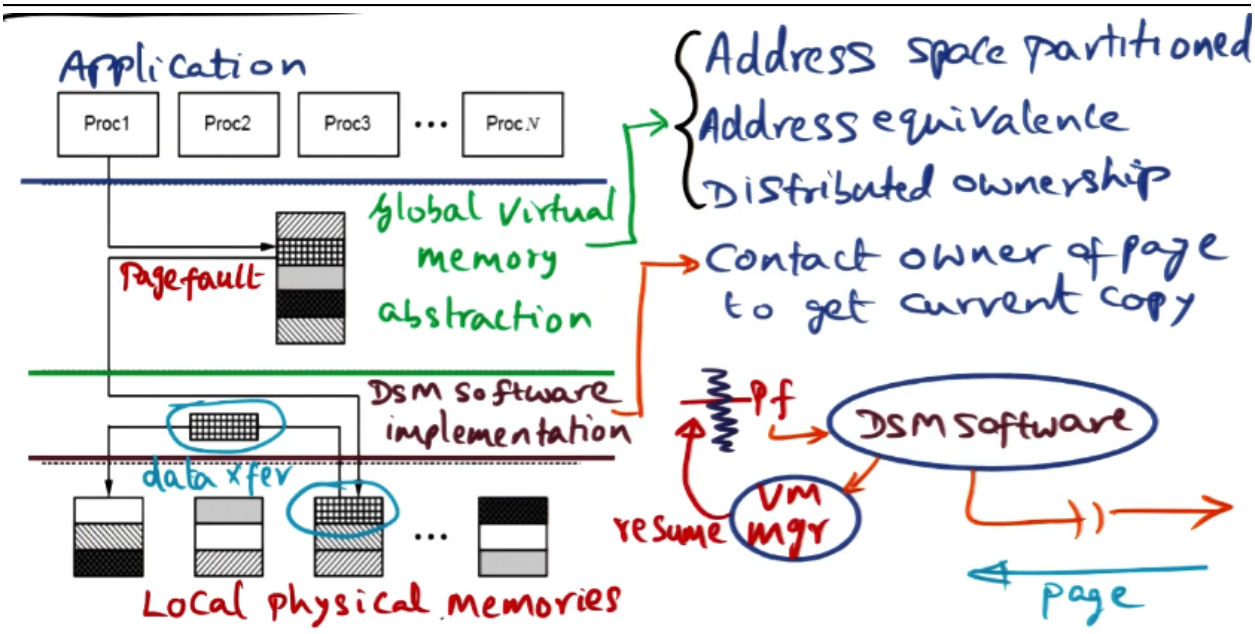
    - Runtime system provides message passing library for sending/receiving data across nodes (MPI, PVM, CLF)
    - Every processor has local memory that isn't shared; only way to communicate with other nodes is through message passing
    - Popular in scientific computing
    - Downside: Difficulty writing an explicitly parallel program
        + Notion of shared memory makes this easier
* Cluster as a Parallel Machine (DSM)
    - Provide the illusion of shared memory to an application memory
    - Handle the message passing on the backend so the application programmer doesn't have to worry about the implementation
        + Marshaling, unmarshaling, etc.
    - Feels like programming on a shared memory machine, but on a cluster
* History of Shared Memory Systems
    - Research in hardware DSM, software DSM, and structured DSM has been ongoing since the mid 80s
    - Structured DSM: Structured objects over a network
    - Clouds and Stampede were built at Georgia Tech
    - Modern research focuses on applying DSM to clusters of SMPs
* Shared Memory Programming
    - Two types of memory access
        1. Normal read/write to shared data
        2. Read/write to synchronization variables (locks, barriers)
* Memory Consistency and Cache Coherence
    - Memory consistency: What is the model presented to the programmer?
        + Contract between application programmer and system
    - Cache coherence: How is the system implementing the model in the presence of private caches?
* Sequential Consistency
    - Program order within a process + arbitrary interleaving across multiple processes
    - Individual read/write operations are atomic in a process
* Sequentially Consistent Memory Model
    - SC does not distinguish between data and synchronization reads/writes
    - Performs coherence action on every read/write access
* Typical Parallel Program
    - P1 locks L, reads from A and writes to B
    - Later, P2 locks L, writes to A and reads from B
    - P2 does not access data until P1 releases the lock
    - No need for coherence action for a and b until L is released
        + SC has more overhead to maintain coherence, limits scalability
* Release Consistency
    - a1: acquire(L) in process P1
        + data accesses
    - r1: release(L)
    - a1: acquire(L) in process P2
        + data accesses
    - r2: release(L)
    - If P1:r1 happens before P2:a2
        + All coherence actions prior to P1:r1 should be complete before P2:a2, but they can be delayed until the acquire
    - Don't need to block the processor on acquire, only on release
    - Arriving at a barrier is equivalent to an acquire, leaving a barrier is equivalent to a release

* Release Consistency Memory Model
    - Release consistency distinguishes between data and synchronization reads and writes
    - Advantage: Coherence action only when lock released
    - Don't block processor for synchronization actions to complete until a release operation occurs
* Distributed Shared Memory Example
    - Programmer's intent: P1 modifies struct(A), P2 waits for the modification to finish and uses struct(A)
    - Code for process P1:
    - modify(A);
    - lock(L);
        + flag = 1;
        + signal(C);
    -unlock(L);
    - Code for process P2:
    - lock(L);
        + while(flag == 0) { wait(C, L); }
    - unlock(L);
    - use(A);
    - In an RC memory model, the processor only needs to be blocked is when the code reaches the unlock
        + Everything else can occur in parallel across processes
* Advantage of RC over SC
    - No waiting for coherence actions on every memory access
        + Overlap computation with communication
    - Better performance for RC model compared to SC model
* Lazy Release Consistency
    - Eager release consistency: Entire system is cache coherent at the moment of release
    - Lazy release consistency: System is only cache coherent when the next process attempts to acquire the lock
        + Allows the coherence overhead to (potentially) be scheduled at a time when the processor is underutilized
* Eager vs Lazy Release Consistency
    - Eager: Changes broadcast to all processes
        + "Push" model
    - Lazy: No broadcast, only communication occurs at acquire
        + "Pull" model



Eager vs Lazy Release Consistency

* Pros and Cons of Lazy and Eager
    - Pros: Fewer messages
    - Cons: More latency at acquire
* Software Distributed Shared Memory
    - In a tightly coupled multiprocessor, coherence is maintained at the individual memory access level by the hardware
        + This will lead to too much overhead on a cluster (every memory access would require intervention from the system)
    - In a cluster, DSM model must implement coherence protocols
        + Maintain coherence at the page level instead of memory location
    - Global virtual memory abstraction
        + Address space partitioned into chunks that are managed individually on different nodes of the cluster
        + Address equivalence: If memory location X is accessed, this means exactly the same thing regardless of the node
        + Distributed ownership for different virtual pages across processes
    - Owner of a particular page is responsible for keeping coherence information for that page
        + Metadata saying which processes are sharing a page
    - When a page fault occurs, the DSM software knows which node has a current copy of the page and requests it from that node
        + The software DSM contacts the VMM to to update the page table so the process is aware that the page has been brought into memory
    - Yale's Ivy, Penn's Mirage, Georgia Tech's Clouds DSM systems used a multiple reader, single writer model
        + Potential for false sharing: Data appears to be shared even though programmatically, they are not
        + Two processes might have separate data structures on the same page which causes the page to be invalidated unnecessarily
        + Simpler to implement



Software DSM

```
* LRC with Multi Writer Coherence Protocol
    - OS operates at the page level, so DSM can be integrated with OS if
    it also operates on pages
    - X, Y, and Z are pages modified in the critical section by P1
    - Xd, Yd, and Zd are the diffs of those modifications
    - When P2 acquires the lock, X, Y, and Z are invalidated by the DSM if
    copies are resident in local memory of P2
        + If P2 tries to access any of the pages, it will see it is invalid
        + The DSM knows the previous holder has the diffs to update the page
    - If another process modified the page in between, P2 would need to get
    the diffs from both P1 and P3 and apply them
        + Even though the invalidation happened at acquire, the overhead
        isn't actually incurred until an access occurs
    - Likely that there are multiple data structures within page X
        + Programmer may have different locks for each data structure
        + DSM software only applies the diffs from the same lock; if lock L2
        was used to modify X and a process locks X with lock L, the DSM
        software won't invalidate X
* Implementation
    - DSM software holds a "twin" of the page when it's locked
    - When the lock is released, DSM computes the diff between the twin and
    the modified version
        + Computed as a runlength encoded diff (position 1->position 2 has
        been changed)
    - After the lock is released, the page is write protected to prevent
    further updates
    - After the diff is computed, the twin is discarded to free up memory
    - Writes to same portion of a page under different locks -> data race
        + This is a problem of the application
    - Cooperation between DSM and OS
    - There is space overhead (twins, diff data structure)
```

        + Diffs can accumulate over time, making overhead significant
        + DSM performs its own sort of garbage collection; when overhead reaches a threshold, the DSM software will begin applying the diffs to the pages and freeing the memory
    - TreadMarks uses lazy release consistency and multiple writer coherence
* Non Page Based DSM
    - Library-based is an alternative to page-based DSM systems
        + Annotate shared variables
        + Coherence actions inserted at point of access
        + Accessing a shared variable causes a trap so the DSM software can be accessed
        + By doing this at the variable level, there is no false sharing
    - Structured DSM
        + API for structs
        + Coherence actions on API calls
        + When an application makes an API call, the runtime determines what coherence actions need to occur
* Scalability
    - Pro: Exploit parallelism
    - Con: Increased overhead
    - Can't achieve expected performance due to the overhead
        + This occurs in SMPs, but is even more noticeable when the coherence is implemented in software, as in DSM, than hardware
* DSM and Speedup
    - If sharing is too fine-grained, the overhead due to the software makes performance less than ideal
        + Visible even when implemented in hardware
    - Competition to communication overhead needs to be minimized
        + Complex data structures can cause implicit data accesses and significant network overhead (pointers)
* Distributed Shared Memory Conclusion
    - Page-based DSM, as imagined, is dead
    - Structured DSM (providing higher level data abstractions for sharing among threads executing on different nodes of the cluster) still exists

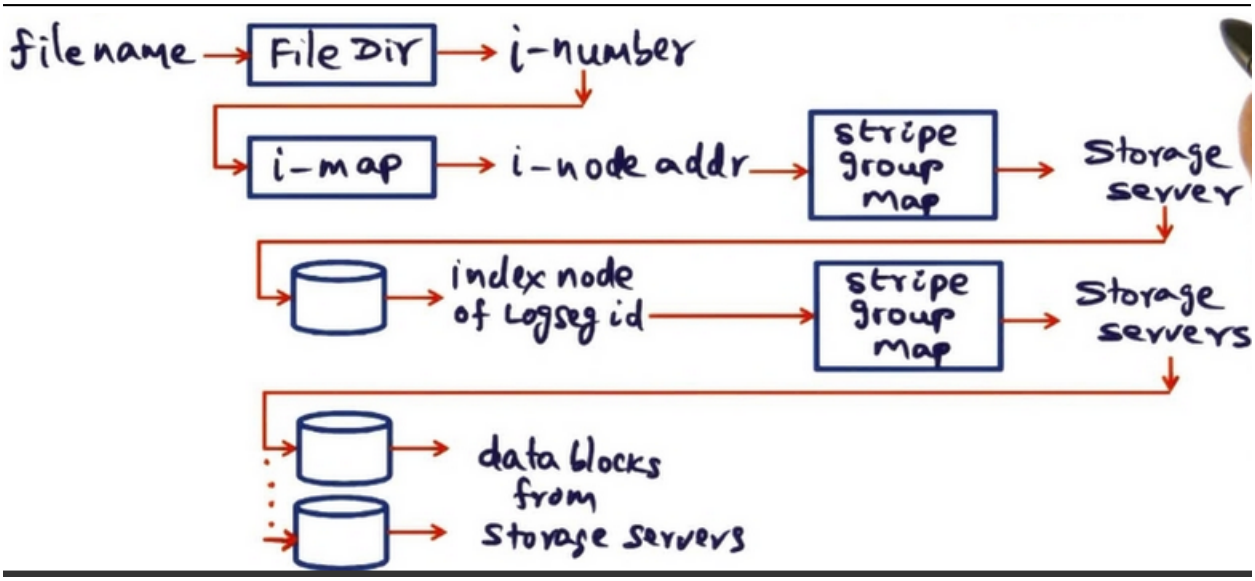
Distributed File Systems

* The First Network File System (NFS)
    - Generic name for any file system accessed remotely
    - First built by Sun in 1985
* Network File Systems
    - Clients distributed across the LAN and fileservers that are also attached to the network
        + Servers appear centralized, but may be distributed (one for faculty and one for students)
    - Servers cache files instead of going to the disk every time
    - Single server is a bottleneck for scalability
        + Cache is limited to memory capacity of central server
* Distributed File Systems
    - No central server, each file distributed across several nodes
    - DFS implemented across all disks in the network
    - Cumulative bandwidth of all servers can be utilized to service requests
    - Management of metadata (CPU cost) can be spread across servers
    - Larger memory footprint for file cache

        + Cooperative caching among nodes
    - DFS can be a serverless filesystem if the responsibility of managing, serving, and caching files to be equally distributed across nodes
    - DFS: How do we use cluster memory for cooperative caching of files?
* Preliminaries: Striping a File to Multiple Disks
    - RAID: Redundant array of inexpensive disks
    - Increase I/O bandwidth by striping files to parallel disks
    - Failure protection by ECC (error correction codes)
        + Detect errors when reading from disks using extra information that's been written to disk (hashes)
    - Drawbacks
        + Cost: multiple drives for a single file
        + Small write problem: Inefficient to read/write a small file to many different disks
* Preliminaries: Log Structured File Systems (LFS)
    - Buffer changes to multiple files in one contiguous log segment to disk once it fills up or periodically
        + Solves small write problem
    - Log segment data structure stores modifications to files
        + Not writing to files, but to this buffer
    - Log segments are sequential, which is ideal for disk access speed
    - If a read occurs before the log has been written, the server must reconstruct the file from the diffs
        + Could be latency associated with reading the first time
        + After that, the data is cached in memory
    - Need to clean logs periodically
        + If the same portion of a file has been overwritten, the old diff is no longer needed
    - Journaling file system: Has both log and data files
        + Applies log files to data files and discards logs
        + Goal is similar, but logs only exist for a short duration
* Preliminaries: Software RAID
    - Zebra File System (UC-Berkeley)
        + Combines LFS and RAID
        + Stripes log segment on multiple nodes' disks in software
        + LFS solves small write problem and RAID provides parallelism
    - Use commodity hardware on nodes connected over LAN
        + Similar to hardware RAID, but software handles the striping
* Putting Them All Together Plus More
    - XFS, a distributed file system (UC-Berkeley)
        + Log based striping (from Zebra)
        + Cooperative caching (from prior UCB work)
        + Dynamic management of data and metadata (serverless)
        + Subsetting storage servers
        + Distributed log cleaning
* Dynamic Management
    - Traditional NFS with centralized server(s)
        + Memory contents include metadata, file cache, and client caching directory
        + Client caching directory: Who is currently accessing files
        + Unix implementation is not concerned with multiple clients caching the same file
    - If one server has many hot files, its ability to service requests might be limited by bandwidth or other hardware constraints

        + If another server is housing cold files and not acting, the system is inefficient
    - XFS
        + Metadata management is dynamically distributed: Offloading work for specific files to other servers
        + Cooperative client file caching: If one file already has the file cached, it might be faster to retrieve it from the client instead of the server's disk
* Log Based Striping and Stripe Groups
    - When a client writes to a file, the changes are written to an append-only log in memory of the client
        + When the log fills up (predetermined threshold) they're written to disk
        + Compute the parity (ECC, checksum) and stripe across storage servers
        + Do this regularly to prevent data loss
    - Subset storage servers for each log segment (don't use all servers for every write)
* Stripe Group
    - Subset servers into "stripe group"
    - Parallel client activities (higher throughput)
    - Increased availability (can survive multiple server failures)
    - Efficient log cleaning (different cleaning service per stripe group)
* Cooperative Caching
    - Cache coherence
        + Single writer, multiple readers
        + File block unit of coherence, not entire file
    - Manager tracks metadata
        + "This file block F1 is in the cache of client C1 and C2"
        + If client C3 issues a write request, the manager must tell C1 and C2 that they can no longer access the file block (invalidate F1)
    - Write-request
        + Client receives token
        + Manager can revoke the token at any time (on read)
    - XFS uses the manager to implement cooperative caching
        + If the manager receives a new read request, it can go to the cache of one of the clients to retrieve the data
* Log Cleaning
    - Segment 1
        + Writes to file blocks 1, 2, and 5 (may belong to different files)
    - Segment 2
        + Writes to file blocks 1, 3, and 4 (must kill old block 1 diff)
        + Segment 1 has a hole
    - Segment 3
        + Writes to file block 3 (must kill old block 2 diff)
        + Segment 2 has a hole
    - Log cleaning refers to removing holes in log segments
        + Aggregate all "live blocks" into a new segment
        + Garbage collect all of the old segments (similar to cleaning up diff files in TreadMarks DSM)
        + Desire for this to be distributed as opposed to centralized
    - Client is responsible for managing its own segments
        + Each stripe group is responsible for cleaning activity
        + A stripe group has a leader that is responsible for reconciling clients wanting to change segments and garbage collector trying to

free segments
* Unix File System
    - Filename, offset -> inode -> data blocks on disk
* XFS Data Structures
    - Client node action
        + Filename -> mmap -> metadata manager
    - Manager node actions
        + File directory: Contains filename to inode number mapping
        + i-map: Returns inode address for the log segment for this filename
        + Stripe group map: How the file is striped
        + Use stripe group map to get the log segment (since it's striped)
        + Use the log segment and stripe group map to access the blocks from storage servers
    - File directory: Client node where created
    - Mmap: Statically assigned at creation, replicated globally
    - Imap: Partitioned among managers
    - i-node: One per file
    - Stripe group: Assigned at file/log creation time, replicated globally
    - Log segment: Set of logs for a file
    - Data blocks: Actual data pertaining to a file



XFS Data Structures

* Client Reading a File From Its Own Cache
    - Filename, offset -> directory -> inode #, offset -> Unix cache -> data
        + All of this is locally cached in client memory
        + Fastest path for file access
    - If file isn't in local cache, go to manager node, which accesses the file from a peer's cache (much faster than disk)
        + Second fastest path for file access
    - If file isn't in a peer's cache, manager goes to stripe group, gets the log segment from disk, uses the segment and stripe group to get the data block address, and goes to each disk to get the data
        + Might be able to eliminate the log segment lookup if the manager has the segment cached

        + Longest path for file access
* Client Writing a File
    - Client aggregates writes into its log segment in memory
    - Writes to stripe group
    - Notifies manager about the log segments that have been flushed to disk
* Distributed File Systems Conclusion
    - Technical innovations
        + Log based striping and subsetting storage servers over which a block is striped is a technical innovation
        + Cooperative caching with dynamic management of data and metadata
        + Distributed log cleaning
        + These could be applied to the design of other distributed subsytems
    - Building scalable NFS products is still part of industry