Week 8:
Virtual Memory Management
and
Distributed Shared Memory

(Superpage slides adapted from Juan Navarro’s OSDI presentation)
Topics

• Review virtual memory basics
• Large (64-bit) virtual address spaces
• Multiple Page Sizes
• Placement policy and cache effects
• NUMA multiprocessor memory management
• Distributed shared memory
Review: Virtual memory goals

• Efficiency

• Transparency
  • Relocation
  • Logical/Physical organization
    ➞ Address translation is required

• Protection and Sharing
  • A process’s memory should be protected from unwanted access by other processes, both intentional and accidental
    ➞ Requires hardware support
  • Need ways to specify and control what sharing is allowed
Review: Virtual address space

- process address space (A.S.) layout
  - logical or virtual address space
- CPU generates logical addresses in this space as program executes
  - Called virtual addresses
- Memory system must see physical (real) addresses
  - Translation is done by memory management unit (MMU)
  - Physical memory must be allocated for each virtual location used by the program
Review: Paging

- Partition memory into equal, fixed-size chunks
  - called *page frames* or simply *frames*
- Divide processes’ memory into chunks of the same size
  - These are called *pages*
- Any page can be assigned to any free page frame
  - No external fragmentation
  - Minimal internal fragmentation
- First seen in CTSS circa 1961
- Demand paging (automatic transfer to/from backing store) first used in the Atlas computer
  - Described in a 2-page CACM article, 1961
“Typical” Address Translation

- Virtual Address
  - Page number
  - Offset
  - Page table
  - Page frame

- Physical Address
  - Page frame
  - Offset

- Physical Memory
Page tables – space limitations

- Memory required for page table can be large
  - Need one PTE per page
  - 32 bit virtual address space w/ 4K pages \( \rightarrow 2^{20} \) PTEs
  - 4 bytes/PTE \( \rightarrow 4\)MB/page table
  - 100 processes \( \rightarrow 400\)MB just for page tables!
  - And modern processors have 64-bit address spaces
  - \( \rightarrow 16 \) petabytes for EACH page table!

- Solution 1: Use another level of indirection
  - Hierarchical page tables
  - Aka forward-mapped page tables
Page tables – space limitations

- Memory required for linear page table can be large
- Solution 1: Hierarchical page tables
  - Aka forward-mapped page tables
A valid bit indicates if a page is allocated. The page table can be divided into 4 pages.

Notice that some pages have all valid bits zeroes.

Valid bit in Page Directory == whole page of pages is (or is not) allocated.

Source: the OSTEP textbook (see CSC369)
Tradeoff: space vs. time

- Multi-level page table
  - Saves space
  - Adds more levels of indirection when translating addresses
  - How many memory accesses on each translation, compared to linear?
  - Also more complexity in the translation algorithm

- We’ll review how a TLB speeds up the “time” aspect
64-bit Address Space

- Really only using 48 bits
- Why? No need for more yet, wastes transistors.
- ISA supports 64-bit, but current CPUs only use lower 48-bits.
- Can be extended later to 64-bits without breaking compatibility.

![Bill Gates Quote](https://example.com/quote)

> 640K ought to be enough for anybody.
> -Bill Gates, Microsoft 1981
64-bit Address Spaces

- Suppose we just extended the hierarchical page tables with more levels
  - 4K pages => 12 bits for offset => 52 bits for page numbers
    - Assuming 8 bytes for PTE (common on 64-bit architectures)
    - Maximum 4KB/8B = 512 entries per level
    - 9 bits for each level => 52/9 means 6 levels
    - Too much overhead!
  - Try bigger pages: 16K pages => 14 bits for offset, 50 bits for page numbers
    - Maximum 16K/8B = 2k entries per level (pages are 4X larger)
    - 11 bits for each level => 50/11 means 5 levels
    - Better, but still a lot of memory reads
Paging Limitations - Time

- Memory reference overhead (time)
  - 2 memory reads (references) per address lookup
    - First read page table, then actual memory
  - Hierarchical page tables make it worse
    - One read per level of page table
  - Solution: use a hardware cache of lookups!

- Translation Lookaside Buffer (TLB)
  - Small, fully-associative **hardware** cache of recently used translations
  - Part of the Memory Management Unit (MMU)
Example: Pentium Address Translation
Translation Lookaside Buffer (TLB)

- Translates virtual page #s into PTEs (not physical addresses!)
  - Can be done in a single machine cycle
- TLBs are implemented in hardware
  - Mostly, fully associative cache (all entries looked up in parallel)
  - Tags: virtual page numbers
  - Values: PTEs (entries from page tables)
  - With PTE + offset, can directly calculate physical address

<table>
<thead>
<tr>
<th>VPN</th>
<th>Offset</th>
<th>TAG (VPN)</th>
<th>VALUE (Page Table Entry)</th>
<th>PFN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00002</td>
<td>0</td>
<td>0x00002</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0x7fffffff</td>
<td>0x7fffffff</td>
<td>0x7fffffff</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0x40002</td>
<td>0</td>
<td>0x40002</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0x00010</td>
<td>0</td>
<td>0x00010</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
TLBs Exploit Locality

• Processes only use a handful of pages at a time
  • Only need those pages to be “mapped”
  • 16-64 entry TLB, so 16-64 pages can be mapped (64-256K)

• Hit rates are very important
  • Typically >99% of translations should be *hits*
  • What happens when we miss?

• How well does it work?
TLB coverage

- Amount of memory that can be accessed without incurring TLB misses
  - E.g., 128 entries, 4KB pages => 512KB
  - 16KB pages => 2MB
- Typical TLB coverage ≈ 1 MB
  - Intel Xeon circa 2010 – 512 entries, 4k page = 2 MB
- Not a lot!
TLB coverage trend

TLB coverage as percentage of main memory

Factor of 100 decrease in 15 years

TLB miss overhead: ≤5%

5-10% ≥30%
How to increase TLB coverage

• Increase TLB size
  • Access time goes up!
• Increase page size!
• Use superpages!
  • Both large and small pages – power-of-2 size
  • 1 TLB entry per superpage
  • Contiguous, and virtually and physically aligned
  • Uniform attributes (protection, valid, ref, dirty)
• Benefit: Increase TLB coverage
  • no increase in TLB size
  • keeps internal fragmentation and disk traffic low

If only large pages: Higher internal fragmentation, more memory pressure, more I/O.
Superpage TLBs – hardware constraints

- **Superpage sizes** must be power-of-two multiples of the *base page size*
- Must be *aligned* in both virtual and physical memory (e.g. 4 MB superpage must begin on a 4 MB address boundary in both spaces)

- **TLB entry** for superpage:
  - Only a single reference bit, dirty bit and protection bits
  - Includes page size

- **Must be supported by MMU of that processor**
  - MIPS, UltraSPARC, Alpha, PowerPC …
  - Itanium II sizes: 4K, 8K, 16K, 64K, 256K, 1M, 4M, 16M, 64M, 256M, 1G, 4G
  - Supported also in later generations, e.g., Haswell, Broadwell, Skylake, etc..
Superpage TLB Alignment

- Virtual memory
  - Base page entry (size=1)
  - Superpage entry (size=4)

- Physical memory

virtual address

physical address
Why multiple superpage sizes

- Different apps have different “best” size
  - Different data structures in a single app have different “best” size

<table>
<thead>
<tr>
<th></th>
<th>64KB</th>
<th>512KB</th>
<th>4MB</th>
<th>All</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT</td>
<td>1%</td>
<td>0%</td>
<td>55%</td>
<td>55%</td>
</tr>
<tr>
<td>galgel</td>
<td>28%</td>
<td>28%</td>
<td>1%</td>
<td>29%</td>
</tr>
<tr>
<td>mcf</td>
<td>24%</td>
<td>31%</td>
<td>22%</td>
<td>68%</td>
</tr>
</tbody>
</table>

Reductions in execution time
The Superpage Problem

• Main Issues
  • Allocation
    • what frame to choose on page fault
  • Promotion
    • combine base pages into superpage
  • Demotion
    • break superpage into smaller superpages, or base pages
  • Fragmentation
    • need contiguous physical pages to create superpage
Allocation

• When we bring a page in main memory, we can
  • Put it anywhere in RAM
    • Will need to relocate it to a suitable place when we merge it into a superpage
    
    => relocation-based allocation
  • Put it in a location that would let us "grow" a superpage around it:
    • Must pick a maximum size for the superpage

=> reservation-based allocation
Previous research approaches

- Reservation-based
  - Talluri & Hill “Surpassing the TLB performance of superpages with less operating system support”
  - One superpage size only, designed to work with proposed partial sub-block TLBs (TLB entry: only one PPN, but may have multiple valid bits)

- Relocation-based
  - Move pages at promotion time (i.e., migrate the pages to contiguous region when superpages are “likely to be beneficial”)
  - Disadvantage: must recover copying costs
  - E.g. Romer et al. “Reducing TLB and memory overhead using online superpage promotion”.
  - Not known to be implemented in non-research OS!
Prior **commercial OS** approaches

- Eager superpage creation (IRIX, HP-UX) - reservation-based
  - Superpage is allocated at page fault time
  - Size specified by user: non-transparent
    - IRIX
      - Can select different page size for any suitably-aligned range of the virtual address space
    - OS maintains list of free pages of each size, *coalescing daemon* periodically tries to refresh
    - Large pages can be demoted under memory pressure
  - HP-UX
    - Can select different sizes for text and data segment only
    - Hint is associated with binary, not selectable at run-time
Design

• Now look in detail at Navarro et al.’s design decisions for
  • Allocation
  • Promotion
  • Demotion
  • Fragmentation control
Use *preemptible reservations*

How much do we reserve?

Goal: good TLB coverage, without internal fragmentation.
Key observation

Once an application touches the first page of a memory object then it is likely that it will quickly touch every page of that object

- Example: array initialization
- Optimistic policy
  - no apriori knowledge that other base pages on the same superpage will get accessed soon
  - superpages as large as possible (up to size of memory object) and as soon as possible (even on first page fault)
Allocation policy and reservation size

- Multiple superpage sizes
  - Physical memory classified into contiguous regions of different sizes, and managed by a *buddy allocator*
- On page fault, pick largest aligned superpage that contains faulting base page and does not overlap with other allocated pages or tentative superpages
  - *Fixed*: Go for **biggest size** that is no larger than the memory object
  - *Dynamic*: No such restriction, but limit to **current size of object** to avoid waste
- What if size is not available?
  - Try preemption before resigning to a smaller size (Preempted reservation had its chance)
  - Last resort: assign a smaller extent than desired
Reservation lists: keep track of frame extents that are not fully populated

- Lists sorted by time of most recent page frame allocation
  - => best candidate for preemption is at front

Idea: preempt reservation whose *most recently* populated frame happened the *least recently*
Incremental promotions

Promotion policy: opportunistic

• Superpage is created whenever any superpage-sized and aligned extent within a reservation is fully populated.

![Diagram showing incremental promotions]

- Superpage creation process with increasing sizes: 2, 4, 4+2, 8
Demotions: incremental, speculative

• Incremental demotion:
  • a. When a base page of a superpage is evicted from memory
    • Don’t just evict the whole superpage => incrementally demote first
  • b. When the access rights change for a subpart of a superpage
    • Why?

• Speculative demotion:
  • One reference bit per superpage
  • How do we detect portions of a superpage not referenced anymore?
  • On memory pressure, demote superpages speculatively => now a bit each
  • Unused base pages detected and evicted
  • Re-promote (incrementally) as pages are referenced
Demotions: dirty superpages

- Why is this an issue?
- One dirty bit per superpage
  - what’s dirty and what’s not?
  - page out entire superpage => unnecessary I/O is expensive!
- Demote on first write to a clean superpage

write

Re-promote (incrementally) as other pages are dirtied

increment
Fragmentation control

• Mostly done by buddy allocator
  • Coalescing available memory regions
  • Not enough though.. Why?

• Modified page replacement daemon
  • Contiguity-aware page replacement (read details in the paper!)

• Cluster wired pages
  • Memory pages used by the FreeBSD kernel for its internal data are wired (non-pageable!)
  • Tend to get scattered over time
  • Cluster them in a contiguous section of memory to avoid further fragmentation
Experimental setup

- FreeBSD 4.3
- Alpha 21264, 500 MHz, 512 MB RAM
- Page sizes: 8 KB, 64 KB, 512 KB, 4 MB pages
- 128-entry DTLB, 128-entry ITLB
- Unmodified applications
Best-case benefits

- TLB miss reduction usually above 95%
- SPEC CPU2000 integer
  - 11.2% improvement (0 to 38%)
- SPEC CPU2000 floating point
  - 11.0% improvement (-1.5% to 83%)
- Other benchmarks
  - FFT ($200^3$ matrix): 55%
  - 1000x1000 matrix transpose: 655%
- 30%+ in 8 out of 35 benchmarks
  - Modest slowdown (speedup ~0.99) in 2
• Different applications benefit most from different superpage sizes
  • Should let system choose among multiple page sizes
• Contiguity-aware page replacement daemon can maintain enough contiguous regions
• Huge penalty for not demoting dirty superpages
• Overheads are small
Fragmentation control

(normalized contiguity of free memory)

- 'no frag control'
- 'frag control'

- 'full speedup'
- 'partial speedup'
- 'no speedup'

Time: 10min
Conclusions

• Superpages: 30%+ improvement
  • transparently realized; low overhead
• Contiguity restoration is necessary
  • sustains benefits; low impact
• Multiple page sizes are important
  • scales to very large superpages
• Source code and more info at:
  • web.mit.edu/freebsd/head/sys/vm/vm_reserv.c
• Available in FreeBSD as of 7.2 (May 2009)
  • Linux efforts available, not part of mainline kernel
• See also MIX TLBs paper (ASPLOS'17):
  • https://guilhermecox.github.io/dw/gcox-asplos17.pdf
Recall from 369, 3 policies characterize a virtual memory management scheme:

- Fetch Policy – *when* to fetch a page
- Placement Policy – *where* to put the page
  - Are some physical pages preferable to others?
- Replacement Policy – *what* page to evict to make room?
Placement Policy

• Address translation allows us to allocate any physical page for any virtual page

• We’ll look at 3 reasons why choosing physical pages carefully can be better than random placement
  • Cache conflicts
  • NUMA multiprocessors
  • Energy savings
Cache Access

• Data is loaded into cache by blocks called *lines*
  • 32 – 128 byte line sizes are typical
• Restrictions on block placement create 3 categories of cache organization:
  • Each block can be stored in exactly 1 location in the cache ➞ *direct-mapped*
    • Mapping is (block address) modulo (# blocks in cache)
  • Any block can be stored in any cache line ➞ *fully associative*
  • Each block can be stored in a restricted set of locations in the cache ➞ *set associative*
    • Map block address to set first using (block addr) % (# of sets), then place block within set
    • If N locations in a set, called *N-way set associative*
Direct Mapped Example

- 8 byte line size, 8 lines in cache => 64 bytes total cache size
- 32 byte page size => data from one page will occupy 4 lines in cache

- Case 1: access all bytes on pages 2 and 3 (A on page 2, B on page 3)
  
  ```
  for (i=0; i < 32; i++)
  A[i] = B[i];
  ```

- Case 2: access all bytes on pages 2 and 4
  
  ```
  for (i=0; i < 32; i++)
  A[i] = B[i];
  ```
What address is used?

- **Virtual address**
  - ✓ Does not need to be translated before checking cache
  - ✓ Application programmer can reason about conflicts
  - ✗ Cache needs to be flushed on context switch

- **Physical address**
  - ✓ Data may stay in cache across context switches
  - ✗ Vaddr must be translated before checking cache
  - ✗ Conflicts depend on what physical page is allocated
Conflict-aware page placement

- OS can select physical pages on allocation to try to reduce cache conflicts
- IDEA: assign a colour to each page such that pages with different colours do not conflict in the cache
  - All pages with same colour map to same lines or sets in the cache
  - Number of colours = (cache size) / (pg size * associativity)
  - Previous example: how many colours?
  - A page’s colour is (page number) modulo (num colours)
- 2 main OS allocation strategies:
  - Page coloring
  - Bin Hopping
Page Coloring

- Assign colour to virtual and physical pages
- On page fault, allocate a physical page with the same colour as the virtual page
  - Exploits spatial locality
  - Programmer reasoning about virtual addresses still applies
  - Implemented in SGI Irix, Solaris, NT
  - OS keeps per-color free lists
Bin Hopping

- Assign colours to physical pages and keep per-colour free lists as before
- On page fault, allocate physical page of next colour from last one previously allocated
  - Exploits temporal locality
  - Implemented in Digital Unix
NUMA Multiprocessors

- NUMA == Non Uniform Memory Access
- Multiprocessor design where each processor (or small set of processors) have a bank of local memory, but can also access remote memory
  - Local memory is faster to access than remote
NUMA Page Placement

- Want to allocate “local” memory as much as possible
  - Local at allocation time may not be local at access time
  - May want to migrate pages
- Keep per-memory bank free lists
  - Possibly in addition to per-color lists
- SGI Irix made NUMA placement policy user-selectable
  - Round-robin
  - Random
  - First touch
  - Migratable / non-migratable
- Linux has numactl command line tool
  - Built using Linux get/set_memory_policy and mbind() APIs
Part 2:
Distributed Shared Memory
Distributed Shared Memory

• Overview on distributed system basics
• What is distributed shared memory?
• Design issues and tradeoffs
Distributed System Features

• **Multiple** computers
  • May be heterogenous, or homogeneous
  • May be controlled by a single organization or by distinct organizations or individuals
  • No physical shared memory, no shared clock

• Connected by a *communication network*
  • Typically a general-purpose network, not dedicated to supporting the distributed system
  • Messages are sent over network for communication

• **Co-operating** to share resources and services
  • Application processing occurs on more than one machine
Distributed IPC

• Option 1: Use message passing primitives
  • E.g. Unix sockets
    ✓ Good match for underlying structure
    ✗ Programmer has to deal with sending data

• Option 2: Use remote procedure call (RPC)
  ✓ Familiar programming model
  ✓ RPC system handles communication details
  ✗ Passing complex data types is hard
  ✗ Model is synchronous, not a good fit for parallel programming
(Local) Shared Memory

- Uniprocessor or SMP systems
- Processes can share part of their address space
  - Threads in a process share entire address space
- IPC provided through access to shared data
  - Easy to express concurrency, share complex data structures
  - Synchronization needed to prevent data races
- How is this implemented on single computer?
- Can we achieve same effect on dist. system?
Distributed Shared Memory (DSM)

- Goal: allow processes on networked computers to share physical memory through a single shared virtual address space
Central Server DSM

- Simplest implementation
  - All data maintained at server node
  - All read, write of shared data sent to server
  - Server handles request and sends ack

Disadvantages?
Sharing Granularity

Two main categories of DSM systems

- Object-based
  - Pure software approach (can be a library)
  - Individual objects are shared
  - Allows granularity to be determined by object size → less false sharing

- Page-based
  - Can leverage paging hardware (needs OS help)
  - Unit of sharing is (multiple of) page size
  - False sharing is more likely
Page-Based DSM Basics

- Physical memory on each node holds pages of shared virtual address space
  - *Local pages* are present in current node’s memory
  - *Remote pages* are in some other node’s memory
- Each node also has *private* (non-shared) memory

**Colored bars represent physical memory pages.**
**Red pages are remote with respect to Node 1.**
**White pages are free.**
Leveraging MMU Hardware / VM Support

- Page table entry for a page is valid if the page is local
- Access to non-local page causes a page fault
- DSM protocol handles page fault, retrieves remote data
- Operations are transparent to programmer
- Can be implemented at user-level using standard OS services
DSM Page Fault Handling

- DSM system maintains metadata about each shared page
  - Similar to OS page table entry (valid/invalid, read/write permission)
- Uses mmap/mprotect calls to control access
- Installs SIGSEGV signal handler to catch invalid access
Atomic Page Update Problem

- Multiple threads in process share OS page table
  - Need to control multiple threads accessing “missing” page
  - Page must be accessible to allow DSM protocol to update it

Node 1

T1  T2  DSM Library

<table>
<thead>
<tr>
<th>Write A Succeeds!</th>
<th>Read A</th>
<th>SIGSEGV</th>
<th>SIGSEGV Handler</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>mmap(A, PROT_WRITE) Request A</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>mprotect(A, PROT_READ)</td>
<td></td>
</tr>
</tbody>
</table>
Atomic Page Update Solution

- Map file to two virtual addresses
  - One for application, one for DSM system
  - Use different protections on each
  - SIGSEGV allows access to DSM system address to update page
  - Only grant access to application address when page fault is fully handled
Locating Remote Data

- Simplest Design: central server maintains a *directory* recording which machine currently holds each page

1. Node 2 pg faults
2. Consult central server to locate
3. Page requested from current owner, Node N
4. Owner invalidates, sends to new location, Node 2
5. Node 2 informs directory of new ownership

- page *migrates* to the node where most recent access happened
Problem 1

- Directory at central server becomes bottleneck
  - All page query requests go to this node
- Solution: Distributed directory
  - Each node is responsible for portion of address space
  - Responsible node = (page #) mod (num nodes)

<table>
<thead>
<tr>
<th>Page</th>
<th>Locn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>N3</td>
</tr>
<tr>
<td>0004</td>
<td>N1</td>
</tr>
<tr>
<td>0008</td>
<td>N2</td>
</tr>
<tr>
<td>000C</td>
<td>N2</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Page</th>
<th>Locn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>N1</td>
</tr>
<tr>
<td>0005</td>
<td>N3</td>
</tr>
<tr>
<td>0009</td>
<td>N4</td>
</tr>
<tr>
<td>000D</td>
<td>N1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Page</th>
<th>Locn</th>
</tr>
</thead>
<tbody>
<tr>
<td>0003</td>
<td>N4</td>
</tr>
<tr>
<td>0007</td>
<td>N2</td>
</tr>
<tr>
<td>000B</td>
<td>N3</td>
</tr>
<tr>
<td>000F</td>
<td>N3</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

...
Problem 2

• Each virtual page exists on only one machine at a time
  • No caching
• Actively shared pages may lead to *thrashing* .. why?
• Solution: allow replication (caching)
  • Read operations become cheaper
    • Simultaneous reads can be executed locally on multiple nodes
  • Write operations become more expensive
    • Cached copies need to be invalidated or updated
Simple Replication (Read Replication)

- Multiple Readers, Single Writer (MRSW)
  - One node can be granted a *read-write* copy
  - OR multiple nodes can be granted read-only copies

- On read operation:
  - Set access rights to read-only on any writeable copy on other nodes (should be at most one)
  - Acquire read-only copy of the page
Read Replication - Updates

- On write operation:
  - Revoke write permission from other writable copy (if any)
  - Get read-write copy of page
  - Invalidate all copies of page at other nodes
Full Replication

• Multiple readers, multiple writers
  • More than one node can have writable copy of page
  • Access to shared data must be controlled to maintain consistency
    • More on this in a minute....
Dealing with replication

• Must keep track of copies of the page
  • Extend directory with *copyset*
    • The set of all nodes that requested copies
• On request for page copy
  • Add requestor to copyset
  • Send page contents
• On request to invalidate page
  • Send invalidation requests to all nodes in copyset and wait for acknowledgements
Consistency Model

1. Defines when modifications to data may be seen at a given processor
2. Defines how memory will appear to a programmer
   - Restricts what values can be returned by a read of a memory location
3. Must be well-understood
   - Determines how programmer reasons about correctness of program
   - Determines what optimizations are allowed
Recall Sequential Consistency

• All memory operations must execute one at a time
• All operations of a single processor appear to execute in program order
• Interleaving among processors is ok
  • But all processors observe the same interleaving
Achieving Sequential Consistency

• Node must ensure that previous memory operation is complete before proceeding with the next one
  • Must get acknowledgement that write has completed
  • With caching, must send invalidate or update messages to all copies
    • **ALL** these messages must be acknowledged
• To improve performance we relax the rules
Relaxed (weak) consistency

- Depends on which sequential requirement we are relaxing
  - Either program order, or write atomicity
  - Data races and reordering constraints
- Allow reads/writes to different memory locations to be reordered
- Consider operation in critical section:
  - Synchronization should be used for all shared data operations
  - One process actively reading/writing
  - Nobody else will access until process leaves c.s.

=> No need to propagate writes sequentially, *or at all*, until process leaves critical section!
Synchronization Variables

• Weak Consistency Requirements:
  • Accesses to synchronization variables are sequentially consistent.
  • No access to a synchronization variable is allowed to be performed until all previous writes have completed everywhere.
  • No data access is allowed to be performed until all previous accesses to synchronization variables have been performed.

• Operation for synchronizing memory
  • Analog of *fences* in shared memory multiprocessors
  • All local writes get propagated
  • All remote writes are brought in to the local processor
  • Block until memory synchronized
Problems with Weak Consistency

• Inefficiency
  • Synchronization happens at begin and end of a critical section
  • Is process finished memory access? Or is it about to start?
• System must make sure that:
  • All locally-initiated writes have completed
  • All remote writes have been obtained
Can we do better?

- Separate synchronization into two stages:
  1. **acquire access**
     - Obtain valid copies of all pages
  2. **release access**
     - Send invalidations for shared pages that were modified locally to nodes that have copies

- *Eager Release Consistency*
Can do better still

- Release requires sending invalidations to all nodes with copy
  - And waiting for all to acknowledge
- Delay this process
  - On release, send invalidation to directory
  - On acquire, check with directory to see if new copy is needed
- Reduces message traffic on release

- Lazy Release Consistency
How do you propagate changes?

- Send entire page
  - Easy, but may be a lot of data
- Send only what changed
  - Local system must save original and compute differences

Node 1

Node 2

Page “twin”: copy of original data before write.
Create diff at Release

- Changes are encoded into diff
- Twin is discarded
- Page is marked invalid due to modifications at other node
- On next access, diffs are exchanged and applied
Page Allocation & Replacement

- Each node has limited physical memory to cache pages of the DSM
- Eviction can be to local disk, or to another node
- Each page is “owned” by some node, even if multiple copies exist
- Victim selection takes page characteristics into account
  - Read-only copy owned by other node can be discarded
  - Read-only copy owned by evicting node requires (at least) ownership transfer
  - Read-write copy requires actual page transfer