Memory Management Policies

• Recall from CSC369, 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 2 reasons why choosing physical pages carefully can be better than random placement
  - Cache conflicts
  - NUMA multiprocessors
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:
  1. Each block can be stored in exactly 1 location in the cache
     - direct-mapped
     - Mapping from block address to cache index is simply
       \[(\text{block address}) \mod \text{(\# blocks in cache)}\]
  2. Any block can be stored in any cache line
     - fully associative
  3. Each block can be stored in a restricted set of locations in the cache
     - set associative
     - Map block address to set first using
       \[(\text{block address}) \% \text{(\# of sets)}\]
       and then place block in any location within selected set
     - If N locations in a set, it is 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)

```c
for (i=0; i < 32; i++)
    A[i] = B[i];
```

**Case 2:** access all bytes on pages 2 and 4 (A still on page 2, B now on page 4)

```c
for (i=0; i < 32; i++)
    A[i] = B[i];
```

A & B conflict

```c
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
  • Num colours = (cache size) / (pg size * associativity)
  • Previous example: how many colours?
  • A page’s colour is
  • (page number) % (num colours)
Avoiding cache conflicts

• 2 main page allocation strategies for OS:
  1. Page coloring
  2. Bin Hopping

Both of these strategies assign colours to pages and use the colours to allocate pages with the goal of avoiding cache conflicts!
• We will look at how each one operates.
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

Virtual address space

Physical address space
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

• Goal: Allocate “local” memory as much as possible
  • But, 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
  • Default is to allocate from local memory
    • Scheduler tries to avoid migrating thread to other CPU
Part 2: Distributed Shared Memory
Distributed Shared Memory

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

- **Multiple** computers
  - May be heterogeneous, 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, SMP, or NUMA 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 distributed 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 acknowledgement

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

![Diagram showing physical memory pages and remote pages with respect to Node 1.]

Colored bars represent physical memory pages. Red pages are remote with respect to Node 1. White pages are free.
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

Write A succeeds!
```

```
Request A
mmap(A, PROT_WRITE)

SIGSEGV

mprotect(A, PROT_READ)
```
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.

- Page *migrates* to the node where most recent access happened.

1. Node 2 pg faults
2. Consult central server to locate data
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
### 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>N1</th>
<th>N2</th>
<th>N4</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="#" alt="Table 1" /></td>
<td><img src="#" alt="Table 2" /></td>
<td><img src="#" alt="Table 3" /></td>
</tr>
</tbody>
</table>

- Table 1:
  - Page: 0000, 0004, 0008, 000C, ...
  - Locn: N3, N1, N2, N2, ...

- Table 2:
  - Page: 0001, 0005, 0009, 000D, ...
  - Locn: N1, N3, N4, N1, ...

- Table 3:
  - Page: 0003, 0007, 000B, 000F, ...
  - Locn: N4, N2, N3, N3, ...
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

• There are several “flavours” of weak consistency
• Depends on which sequential consistency 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
Why is relaxed consistency ok?

- Consider operation in critical section:
  - Synchronization should be used for all shared data operations
  - One thread actively reading/writing
  - No other thread will access shared data until current thread leaves critical section

  ➔ No need to propagate writes sequentially, or at all, until thread 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.
• Essentially, an 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

- This is called *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

- This is called *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

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