Address Translation

Chapter 8 OSPP
Part I: Advanced
Sparse Address Spaces

- What if virtual address space is large?
  - 32-bits, 4KB pages => 500K page table entries
  - 64-bits => 4 quadrillion page table entries

- Famous quote:
  - “Any programming problem can be solved by adding a level of indirection”
Multi-level Translation

• Tree of translation tables
  – Paged segmentation
  – Multi-level page tables
  – Multi-level paged segmentation

• Stress: hardware is doing the translation!
Paged Segmentation

• Process memory is segmented
• Segment table entry:
  – Pointer to page table
  – Page table length (# of pages in segment)
  – Access permissions
• Page table entry:
  – Page frame
  – Access permissions
• Share/protection at either page or segment-level
Paged Segmentation (Implementation)
Multics

Logical address

Segment table

Segment length

Page–table base

≥

Yes

No

STBR

Page table for segment s

Physical address

Memory
Multilevel Paging

- Page the page table!
Two-Level Paging Example

• A logical address (on 32-bit machine with 4K page size) is divided:
  – a page number consisting of 20 bits
  – a page offset consisting of 12 bits

• Since the page table is paged, the page number is further divided:
  – a 10-bit page number
  – a 10-bit page offset

• Thus, a logical address is as follows:

<table>
<thead>
<tr>
<th>page number</th>
<th>page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_i$</td>
<td>$p_2$</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

• where $p_i$ is an index into the outer page table, and $p_2$ is the displacement within the page of the outer page table.
Address-Translation Scheme

• Address-translation scheme for a two-level 32-bit paging architecture

![Diagram showing Address-Translation Scheme]

How big should the outer-page table be? Assume PTE=4
Multilevel Translation

• Pros:
  – Allocate/fill only page table entries that are in use
  – Simple memory allocation
  – Share at segment or page level

• Cons:
Portability

• Many operating systems keep their own memory translation data structures
  – List of memory objects (segments)
  – Virtual page -> physical page frame
  – Physical page frame -> set of virtual pages

• One approach: Inverted page table
  – Hash from virtual page -> physical page
  – Space proportional to # of physical pages!!
Inverted Page Table

pid, vpn, frame, permissions
Back to TLBs
Efficient Address Translation

• Translation lookaside buffer (TLB)
  – Cache of recent virtual page -> physical page translations
  – If cache hit, use translation
  – If cache miss, walk multi-level page table

• Cost of translation =
  \[ \text{Pr(TLB hit)} \times \text{Cost of TLB lookup} + \]
  \[ \text{Pr(TLB miss)} \times \text{cost of page table lookup} \]

• Even more critical when multi-level tables: why?
TLB and Page Table Translation

Diagram:
- Processor
  - Virtual Address
  - TLB
    - Hit
      - Frame
      - Offset
      - Physical Address
      - Physical Memory
      - Data
  - TLB Miss
    - Page Table
      - Valid
        - Frame
      - Physical Address
      - Physical Memory
      - Data
  - Invalid
    - Raise Exception
TLB Miss

• Done all in hardware

• Or in software (software-loaded TLB)
  – Since TLB miss is rare ...
  – Trap to the OS on TLB
  – Let OS do the lookup and insert into the TLB
  – A little slower ...
TLB Lookup

TLB usually a set-associative cache

Access: valid or not, permissions (early handling of exceptions)
TLB is critical

• What happens on a context switch?
  – Reuse TLB?
  – Discard TLB?

• Solution: Tagged TLB
  – Each TLB entry has process ID
  – TLB hit only if process ID matches current process
TLB consistency

• What happens when the OS changes the permissions on a page?
  – For demand paging, copy on write, zero on reference, ... or is marked invalid!

• TLB may contain old translation
  – OS must ask hardware to purge TLB entry

• On a multicore: TLB shootdown
  – OS must ask each CPU to purge TLB entry
  – Similar to invalid above
## TLB Shootdown

<table>
<thead>
<tr>
<th>Process ID</th>
<th>Virtual Page</th>
<th>Page Frame</th>
<th>Access</th>
</tr>
</thead>
</table>
| Processor 1 TLB
| 0           | 0x0053       | 0x0003     | R/W     |
| 1           | 0x40FF       | 0x0012     | R/W     |
| Processor 2 TLB
| 0           | 0x0053       | 0x0003     | R/W     |
| 0           | 0x0001       | 0x0005     | Read    |
| Processor 3 TLB
| 1           | 0x40FF       | 0x0012     | R/W     |
| 0           | 0x0001       | 0x0005     | Read    |
TLB Optimizations
Virtually Addressed vs. Physically Addressed Caches

• Too slow to first access TLB to find physical address, then look up address in the cache
  – VA -> data
• Instead, first level cache is virtually addressed
• In parallel, access TLB to generate physical address in case of a cache miss
Virtually Addressed Caches

Same issues w/r to context-switches and consistency
Problem

• Aliasing: two aliases for same physical memory and both in cache
  – P1 (VPN 0) -> D1
  – P2 (VPN 3) -> D1

• If P1 changes D1 the 2\textsuperscript{nd} mapping won’t see it
Aliasing Solution 1

• Alias: two (or more) virtual cache entries that refer to the same physical memory
  – A consequence of a tagged virtually addressed cache!
  – A write to one copy needs to update all copies

• Typical solution
  – Keep both virtual and physical address for each entry in virtually addressed cache
  – Lookup virtually addressed cache and TLB in parallel
  – Check if physical address from TLB matches multiple entries, and update/invalidate other copies
Aliasing Solution 2: Physically Addressed Cache

Cache physical translations: at any level! (e.g. frame->data)
Superpages

- On many systems, TLB entry can be
  - A page
  - A superpage: a set of contiguous pages
- x86: superpage is set of pages in one page table
  - x86 TLB entries
    - 4KB
    - 2MB
    - 1GB
Walk an Entire Chunk of Memory

- Video Frame Buffer:
  - 32 bits x 1K x 1K = 4MB
- Lots of TLB accesses
- Superpage can reduce this
Address Translation Uses

- **Process isolation**
  - Keep a process from touching anyone else’s memory, or the kernel’s
- **Efficient interprocess communication**
  - Shared regions of memory between processes
- **Shared code segments**
  - E.g., common libraries used by many different programs
- **Program initialization**
  - Start running a program before it is entirely in memory
- **Dynamic memory allocation**
  - Allocate and initialize stack/heap pages on demand
Address Translation (more)

• Cache management
  – Page coloring
• Program debugging
  – Data breakpoints when address is accessed
• Zero-copy I/O
  – Directly from I/O device into/out of user memory
• Memory mapped files
  – Access file data using load/store instructions
• Demand-paged virtual memory
  – Illusion of near-infinite memory, backed by disk or memory on other machines
Address Translation (even more)

- Checkpointing/restart
  - Transparently save a copy of a process, without stopping the program while the save happens
- Persistent data structures
  - Implement data structures that can survive system reboots
- Process migration
  - Transparently move processes between machines
- Information flow control
  - Track what data is being shared externally
- Distributed shared memory
  - Illusion of memory that is shared between machines
Overview

• Huge data sets => memory hogs
  – Insufficient RAM
  – “out-of-core” applications > physical memory
  – E.g. scientific visualization

• Virtual memory + paging
  – Resource competition: processes impact each other
  – LRU penalizes interactive processes ... why?
The Problem

Figure 1. Impact of sharing the machine with an out-of-core matrix-vector multiplication (MATVEC) on the response time of an interactive task across a range of sleep times between touching 1 MB of data.
Page Replacement Options

• Local
  – this would help but very inefficient
  – allocation not according to need

• Global
  – no regard for ownership
Be Smarter

- I/O cost is high for out-of-core applications (I/O waits)
  - Pre-fetch pages before needed
  - Release pages

- Application may know about its memory use
  - Help the OS
  - automate in compiler
Working Together

• OS role
  – Provide available memory

• Compiler role
  – Insert pre-fetch, release hints

• Not perfect
  – Augment with run-time library

Figure 2. Information flow between components of our system.
Compiler Analysis Example

(a) Source code for averaging nearest-neighbors

\[
\begin{align*}
\text{for} \ (i = 0; \ i < N; \ i++) \\
\text{for} \ (j = 0; \ j < N; \ j++) \\
\quad a[i][j] &= (a[i+1][j-1] + a[i+1][j] + a[i+1][j+1] + a[i][j-1] + a[i][j] + a[i][j+1] + a[i-1][j-1] + a[i-1][j] + a[i-1][j+1]) / 9.0;
\end{align*}
\]

(b) View of data references to the matrix a

Figure 3. Example source code showing multiple references with different types of reuse, and graphical view of the data accesses during a single iteration of the innermost loop.
OS Support

• OS maintains a shared page per application to communication to runtime library
  – Current pages in use and which are in memory
  – Upper limit on pages that can be used per application

• Handling prefetches
  – Discard prefetch if no free pages
  – Prefetches not fully validated and not mapped into TLB
OS Support

• Releaser – new system daemon
  – Identify candidate pages for release – how?
  – Prioritized
  – Leave time for rescue
  – Victims: Write back dirty pages
OS Support

Upper limit = 
min(max_rss, 
current_size + tot_freemem – 
min_freemem)

process limit – take locally

take globally
Compiler support

• Infer memory access patterns
• Most useful with arrays with static sizes, nested loops
• Schedule prefetches
• Schedule releases
  – Assign priority
Runtime System

• Buffers releases
  – Prioritized queues
  – Higher priority = needed again sooner
  – Decides how many to release and which ones
  – Compensate for compiler mistakes
Overall Results

• Analyzed five applications
  – data significantly larger than physical RAM

• With pre-fetching
  – avoided I/O waits in 85% of test cases
  – reduced additional OS overhead by avoiding page faults
    • fewer context switches
Results (cont.)

• Pre-fetching + releases
  – Reduced total run time 30% - 60%

• Interactive apps could remain in memory
Out-of-core app performance

Figure 7. Impact of prefetching and releasing on the execution times of the out-of-core applications. (O = original, P = with prefetching, R = with prefetching and releasing, B = with prefetching and release buffering)
Impact on interactive apps

(a) Impact of MATVEC on response time (last 3 lines in key overlap).
Conclusion

- Too much data, not enough memory
- Application can help out the OS
- Compiler inserts data pre-fetch and release
- Adaptive run-time system
- Reduces thrashing, improves performance, plays nicely with other apps on the system