Week 8: Virtual Memory Management and Distributed Shared Memory

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

• Review virtual memory basics
  • Slides 3-6 not covered in lecture, but this is background that you should know!

• Today
  • Large (64-bit) virtual address spaces
  • Multiple Page Sizes

• Friday
  • 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
Address Translation

Virtual Address

<table>
<thead>
<tr>
<th>Page number</th>
<th>Offset</th>
</tr>
</thead>
</table>

Linear Page table

Physical Address

| Page frame | Offset |

Physical Memory
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.

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

Notice that some pages have all valid bits set to zeroes.

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
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.
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

The diagram illustrates the process of address translation in the Pentium processor. It begins with a virtual address (VA) being processed by the CPU. If the TLB (Translation Lookaside Buffer) has a hit, it directly provides the physical address (PA) to the data cache/main memory. If the TLB has a miss, it consults the page tables (PDE, PTE) to find the physical address. The diagram also includes components such as VPN (Virtual Page Number), VPO (Virtual Page Offset), TLBT (Translation Lookaside Buffer Table), TLBI (Translation Lookaside Buffer Index), VPN1, VPN2, TLB (16 sets, 4 entries/set), PPN (Page Number), and PPO (Page Offset).
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

<table>
<thead>
<tr>
<th>TAG (VPN)</th>
<th>VALUE (Page Table Entry)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00002</td>
<td>1</td>
</tr>
<tr>
<td>0xffffffff</td>
<td>1</td>
</tr>
<tr>
<td>0x40002</td>
<td>1</td>
</tr>
<tr>
<td>0x00010</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 $\Rightarrow$ 512KB
  - 16KB pages $\Rightarrow$ 2MB
- Typical TLB coverage $\approx$ 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%

≥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
• **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
- Physical memory
- Virtual address
- Physical address
- Base page entry (size=1)
- Superpage entry (size=4)
- TLB

Start of 4-page aligned virtual address range
Start of 4-page aligned physical address range
Superpage TLB Alignment

Base page entry (size=1)

Superpage entry (size=4)

Physical address

Virtual address

Physical memory

Virtual memory

Start of 4-page aligned virtual address range

Base virtual page can map to any physical page.

Start of 4-page aligned physical address range
Superpage TLB Alignment

- **Virtual Address**: Start of 4-page aligned virtual address range
- **Physical Address**: Start of 4-page aligned physical address range
- **Base Page Entry (size=1)**
- **Superpage Entry (size=4)**: 4-page superpage must map to 4-page aligned 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>
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”. 
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
Superpage allocation

Use *preemptible reservations*

---

How much do we reserve?
Goal: good TLB coverage, without internal fragmentation.
Superpage allocation

Use *preemptible reservations*

As nearby virtual pages are accessed, allocate physical frames from the reservation.
Use preemptible reservations

We can promote to superpage when all the base pages are valid. (Can we promote even earlier?)
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 *a priori* 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

- Supports 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 (i.e., reservations)
  - Fixed: code segment, memory-mapped file, vs. Dynamic-size: stacks, heap
  - *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

The largest unused (and aligned) chunk is the one that appears at the top:

- 4 frames are filled
- 2 frames are filled
- 1 frame is filled

Lists are sorted by time of most recent page frame allocation, making the best candidate for preemption the one at the 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.
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
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 (2003 matrix): 55%
  - 1000x1000 matrix transpose: 655%
- 30%+ in 8 out of 35 benchmarks
  - Modest slowdown (speedup ~0.99) in 2
More take-aways

- 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

- Graph shows comparison between no fragmentation control and fragmentation control.
- Key markers: full speedup, partial speedup, no speedup.
- Web server versus FFT tasks.
- Time scale: 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