Virtual Memory
October 14, 2008

Topics
- Address spaces
- Motivations for virtual memory
- Address translation
- Accelerating translation with TLBs

Announcements

**Autolab outage**
- The autolab machine was hacked on Saturday
  - not the autolab programs, but the underlying OS
  - rebuilt and brought back online Monday
- Should not block your progress on shelllab
  - all files needed were made available on class website (under docs link)
  - fish machines are working fine
- Re-submit tshlab, if you finished before the outage
  - as always, there was a time gap between last backup and the breakin

**Programs Refer to Virtual Memory Addresses**
- Conceptually very large array of bytes
- Actually implemented with hierarchy of different memory types
- System provides address space private to particular “process”
  - Program being executed
  - Program can clobber its own data, but not that of others

**Compiler + Run-Time System Control Allocation**
- Where different program objects should be stored
- All allocation within single virtual address space

**Simple Addressing Modes**
- Normal (R) Mem[Reg[R]]
  - Register R specifies memory address
    - `movl (%ecx),%eax`
- Displacement D(R) Mem[Reg[R]+D]
  - Register R specifies start of memory region
  - Constant displacement D specifies offset
    - `movl 8(%ebp),%edx`
How does everything fit?
- 32-bit addresses: ~$4,000,000,000$ (4 billion) bytes
- 64-bit addresses: ~$16,000,000,000,000,000,000$ (16 quintillion) bytes

How to decide which memory to use in your program?
- How about after a fork()?

What if another process stores data into your memory?
- How could you debug your program?

So, we add a level of indirection
One simple trick solves all three problems
- Each process gets its own private image of memory
  - appears to be a full-sized private memory range
- This fixes “how to choose” and “others shouldn’t mess w/ yours”
  - surprisingly, it also fixes “making everything fit”
- Implementation: translate addresses transparently
  - add a mapping function
  - to map private addresses to physical addresses
  - do the mapping on every load or store

This mapping trick is the heart of virtual memory

Address Spaces

A linear address space is an ordered set of contiguous nonnegative integer addresses:
\[
\{0, 1, 2, 3, \ldots \}
\]

A virtual address space is a set of \( N = 2^n \) virtual addresses:
\[
\{0, 1, 2, \ldots, N-1\}
\]

A physical address space is a set of \( M = 2^n \) (for convenience) physical addresses:
\[
\{0, 1, 2, \ldots, M-1\}
\]

In a system based on virtual addressing, each byte of main memory has a physical address and a virtual address (or more)

A System Using Physical Addressing

Used by many embedded microcontrollers in devices like cars, elevators, and digital picture frames
A System Using Virtual Addressing

One of the great ideas in computer science
- used by all modern desktop and laptop microprocessors

Why Virtual Memory?
(1) VM allows efficient use of limited main memory (RAM)
- Use RAM as a cache for the parts of a virtual address space
- some non-cached parts stored on disk
- some (unallocated) non-cached parts stored nowhere
- Keep only active areas of virtual address space in memory
- transfer data back and forth as needed

(2) VM simplifies memory management for programmers
- Each process gets a full, private linear address space

(3) VM isolates address spaces
- One process can't interfere with another's memory
- because they operate in different address spaces
- User process cannot access privileged information
- different sections of address spaces have different permissions

(1) VM as a Tool for Caching

Virtual memory is an array of N contiguous bytes
- think of the array as being stored on disk
The contents of the array on disk are cached in physical memory (DRAM cache)

DRAM Cache Organization

DRAM cache organization driven by the enormous miss penalty
- DRAM is about 10x slower than SRAM
- Disk is about 100,000x slower than a DRAM
- to get first byte, though fast for next byte

DRAM cache properties
- Large page (block) size (typically 4-8 KB)
- Fully associative
- Any virtual page can be placed in any physical page
- Requires a “large” mapping function – different from CPU caches
- Highly sophisticated replacement algorithms
- Too complicated and open-ended to be implemented in hardware
- Write-back rather than write-through
Reminder: MMU checks the cache

One of the great ideas in computer science
- used by all modern desktop and laptop microprocessors

How? Page Tables

A page table is an array of page table entries (PTEs) that maps virtual pages to physical pages

- Per-process kernel data structure in DRAM

Address Translation with a Page Table

Page table base register (PTBR)

The VPN acts as index into the page table

If valid=0 then page not in memory (page fault)

Page Hits

A page hit is a reference to a VM word that is in physical (main) memory
Page Faults

A page fault is caused by a reference to a VM word that is not in physical (main) memory.

- Example: An instruction references a word contained in VP 3, a miss that triggers a page fault exception.

Handling a Page Fault

The kernel’s page fault handler selects VP 4 as the victim and replaces it with a copy of VP 3 from disk (demand paging).

- When the offending instruction restarts, it executes normally, without generating an exception.

Why does it work? Locality

Virtual memory works because of locality.

At any point in time, programs tend to access a set of active virtual pages called the working set.

- Programs with better temporal locality will have smaller working sets.

If (working set size < main memory size)

- Good performance for one process after compulsory misses.

If (SUM(working set sizes) > main memory size)

- Thrashing: Performance meltdown where pages are swapped (copied) in and out continuously.

(2) VM as a Tool for Memory Mgmt

Key idea: each process has its own virtual address space.

- It can view memory as a simple linear array.
- Mapping function scatters addresses through physical memory.
- Well chosen mappings simplify memory allocation and management.
Simplifying Sharing and Allocation

Memory allocation
- Each virtual page can be mapped to any physical page
  - A virtual page can be stored in different physical pages at different times – the program never knows

Sharing code and data among processes
- Map virtual pages to the same physical page (PP 7)

Virtual Address Space for Process 1:
- VP 1
- VP 2
- PP 2
- (e.g., read-only library code)

Physical Address Space (DRAM)
- 0
- 0
- N-1
- M-1

IA32 Linux Memory Layout

- Stack
  - Runtime stack (8MB limit)

- Heap
  - Dynamically allocated storage
  - When call malloc(), calloc(), new()

- Data
  - Statically allocated data
  - E.g., arrays & strings declared in code

- Text
  - Executable machine instructions
  - Read-only

Simplifying Linking and Loading

Linking
- Each program has similar virtual address space
- Code, stack, and shared libraries always start at the same address
- The .text and .data sections are copied, page by page, on demand by the virtual memory system.

Loading
- execve() maps PTEs to the appropriate location in the executable binary file
- The .text and .data sections are copied, page by page, on demand by the virtual memory system.

(3) VM as a Tool for Memory Protection

Extend PTEs with permission bits
Page fault handler checks these before remapping
- If violated, send process SIGSEGV (segmentation fault)
Reminder: MMU checks the cache

One of the great ideas in computer science
- used by all modern desktop and laptop microprocessors

Address Translation: Page Hit

1) Processor sends virtual address to MMU
2-3) MMU fetches PTE from page table in memory
4) MMU sends physical address to cache/memory
5) Cache/memory sends data word to processor

Address Translation: Page Fault

1) Processor sends virtual address to MMU
2-3) MMU fetches PTE from page table in memory
4) Valid bit is zero, so MMU triggers page fault exception
5) Handler identifies victim (and, if dirty, pages it out to disk)
6) Handler pages in new page and updates PTE in memory
7) Handler returns to original process, restarting faulting instruction

Speeding up Translation with a TLB

Page table entries (PTEs) are cached in L1 like any other memory word
- PTEs may be evicted by other data references
- PTE hit still requires a 1-cycle delay

Solution: Translation Lookaside Buffer (TLB)
- Small hardware cache in MMU
- Maps virtual page numbers to physical page numbers
- Contains complete page table entries for small number of pages
TLB Hit

A TLB hit eliminates a memory access

TLB Miss

A TLB miss incurs an add'l memory access (the PTE)

Fortunately, TLB misses are rare

Simple Memory System Example

Addressing

- 14-bit virtual addresses
- 12-bit physical address
- Page size = 64 bytes

Simple Memory System Page Table

Only show first 16 entries (out of 256)
Simple Memory System TLB

TLB
- 16 entries
- 4-way associative

\[
\begin{array}{ccccccccccc}
13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\hline
\text{TLB} & \text{TLB} & \text{TLB} & \text{TLB} & \text{VPN} & \text{VPO} & & & & & & & & \\
\end{array}
\]

Address Translation Example #1

Virtual Address 0x03D4

Physical Address

Address Translation Example #2

Virtual Address 0x0B8F

Physical Address
### Address Translation Example #3

**Virtual Address** 0x0020

<table>
<thead>
<tr>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- **VPN**: 0x00
- **TLBI**: 0
- **TLBT**: 0
- **TLB Hit?**: NO
- **Page Fault?**: NO
- **PPN**: 0x28

**Physical Address**

<table>
<thead>
<tr>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- **CT**: Offset
- **CI**: 0x8
- **Hit?**: NO
- **Byte**: MEM

### Summary

**Programmer’s View of Virtual Memory**
- Each process has its own private linear address space
- Cannot be corrupted by other processes

**System View of Virtual Memory**
- Uses memory efficiently by caching virtual memory pages
- Efficient only because of locality
- Simplifies memory management and programming
- Simplifies protection by providing a convenient interpositioning point to check permissions

### Allocating Virtual Pages

**Example: Allocating new virtual page VP5**

- Kernel allocates VP 5 on disk and points PTE 5 to it

### Multi-Level Page Tables

**Given:**
- 4KB (2^12) page size
- 48-bit address space
- 4-byte PTE

**Problem:**
- Would need a 256 GB page table!
- $2^{48} \times 2^{-12} \times 2^2 = 2^{38}$ bytes

**Common solution**
- Multi-level page tables
- Example: 2-level page table
  - Level 1 table: each PTE points to a page table (memory resident)
  - Level 2 table: Each PTE points to a page (paged in and out like other data)
- Level 1 table stays in memory
- Level 2 tables paged in and out
A Two-Level Page Table Hierarchy

Level 1 page table
- PTE 0
- PTE 1
- PTE 2 (null)
- PTE 3 (null)
- PTE 4 (null)
- PTE 5 (null)
- PTE 6 (null)
- PTE 7 (null)
- PTE 8
- (1K - 9) null PTEs

Level 2 page tables
- VP 0
- VP 1023
- VP 1024
- VP 2047
- Gap

Virtual memory
- VP 0
- VP 1022
- VP 1024
- VP 2047

2K allocated VM pages for code and data
6K unallocated VM pages
1023 unallocated pages
1 allocated VM page for the stack

Translating with a k-level Page Table

Virtual address
- VPN 1
- VPN 2
- ... 
- VPN k
- VPO

Physical address
- PPN
- PPO

Servicing a Page Fault

1. Processor signals disk controller
   - Read block of length P starting at disk address X and store starting at memory address Y
2. Read occurs
   - Direct Memory Access (DMA)
   - Under control of I/O controller
3. Controller signals completion
   - Interrupts processor
   - OS resumes suspended process