Virtual Memory

Today

• Address spaces
• VM as a tool for caching
• VM as a tool for memory management
• VM as a tool for memory protection
• Address translation

A System Using Physical Addressing

• Used in "simple" systems like embedded microcontrollers in devices like cars, elevators, and digital picture frames

A System Using Virtual Addressing

• Used in all modern servers, laptops, and smart phones
• One of the great ideas in computer science
Address Spaces

• **Linear address space:** Ordered set of contiguous non-negative integer addresses: 
  \{0, 1, 2, 3 \ldots \}

• **Virtual address space:** Set of \( N = 2^n \) virtual addresses 
  \{0, 1, 2, 3, \ldots, N-1\}

• **Physical address space:** Set of \( M = 2^m \) physical addresses 
  \{0, 1, 2, 3, \ldots, M-1\}

Why Virtual Memory (VM)?

• Uses main memory efficiently
  – Use DRAM as a cache for parts of a virtual address space

• Simplifies memory management
  – Each process gets the same uniform linear address space

• Isolates address spaces
  – One process can’t interfere with another’s memory
  – User program cannot access privileged kernel information and code

Today

• Address spaces
• VM as a tool for caching
• VM as a tool for memory management
• VM as a tool for memory protection
• Address translation

VM as a Tool for Caching

• Conceptually, **virtual memory** is an array of \( N \) contiguous bytes stored on disk.
• The contents of the array on disk are cached in **physical memory (DRAM cache)**
  – These cache blocks are called **pages** (size is \( P = 2^p \) bytes)
DRAM Cache Organization

- DRAM cache organization driven by the enormous miss penalty
  - DRAM is about 10x slower than SRAM
  - Disk is about 10,000x slower than DRAM
- Consequences
  - Large page (block) size: typically 4 KB, sometimes 4 MB
  - Fully associative
    - Any VP can be placed in any PP
    - Requires a "large" mapping function – different from cache memories
  - Highly sophisticated, expensive replacement algorithms
    - Too complicated and open-ended to be implemented in hardware
  - Write-back rather than write-through

Enabling Data Structure: Page Table

- 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

**Page Hit**

- Page hit: reference to VM word that is in physical memory (DRAM cache hit)

**Page Fault**

- Page fault: reference to VM word that is not in physical memory (DRAM cache miss)
Handling Page Fault

- Page miss causes page fault (an exception)
- Page fault causes a memory exception
- Page fault handler selects a victim to be evicted (here VP 4)
- Offending instruction is restarted: page hit!

Key point: Waiting until the miss to copy the page to DRAM is known as demand paging
Allocating Pages

- Allocating a new page (VP 5) of virtual memory.

<table>
<thead>
<tr>
<th>PTE 0</th>
<th>PTE 1</th>
<th>PTE 2</th>
<th>PTE 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid</td>
<td>null</td>
<td>Valid</td>
<td>Valid</td>
</tr>
</tbody>
</table>

Physical page number or disk address

Virtual memory (disk)

<table>
<thead>
<tr>
<th>VP 0</th>
<th>VP 1</th>
<th>VP 2</th>
<th>VP 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>PP 0</td>
<td>PP 1</td>
<td>PP 2</td>
<td>PP 3</td>
</tr>
</tbody>
</table>

Physical memory (DRAM)

<table>
<thead>
<tr>
<th>VP 0</th>
<th>VP 1</th>
<th>VP 2</th>
<th>VP 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>PP 0</td>
<td>PP 1</td>
<td>PP 2</td>
<td>PP 3</td>
</tr>
</tbody>
</table>

Memory resident page table (DRAM)

Locality to the Rescue Again!

- Virtual memory seems terribly inefficient, but it 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.

Today

- Address spaces
- VM as a tool for caching
- VM as a tool for memory management
- VM as a tool for memory protection
- Address translation

VM as a Tool for Memory Management

- 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 can improve locality.

Virtual Address Space for Process 1:

Virtual Address Space for Process 2:

(e.g., read-only library code)
VM as a Tool for Memory Management

- Simplifying 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
- Sharing code and data among processes
  - Map virtual pages to the same physical page (here: PP 6)

Virtual Address Space for Process 1:

0: VP 1
VP 2: PP 2
N-1: PP 6
(e.g., read-only library code)

Virtual Address Space for Process 2:

0: VP 1
VP 2: PP 6
N-1: PP 8

Today

- Address spaces
- VM as a tool for caching
- VM as a tool for memory management
- VM as a tool for memory protection
- Address translation

Simplifying Linking and Loading

- Linking
  - Each program has similar virtual address space
  - Code, data, and heap always start at the same addresses.
- Loading
  - `execve` allocates virtual pages for .text and .data sections & creates PTEs marked as invalid
  - The .text and .data sections are copied, page by page, on demand by the virtual memory system

Kernel virtual memory
User stack (created at runtime)
%sp (stack pointer)
Memory-mapped region for shared libraries
Read/write segment (.data, .bss)
Read-only segment (.init, .text, .rodata)
Run-time heap (created by `malloc`)
Unused
Loaded from the executable file

VM as a Tool for Memory Protection

- Extend PTEs with permission bits
- MMU checks these bits on each access

Process i:

<table>
<thead>
<tr>
<th>Address</th>
<th>EXEC</th>
<th>READ</th>
<th>WRITE</th>
<th>SUP</th>
</tr>
</thead>
<tbody>
<tr>
<td>PP 9</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>PP 6</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>PP 4</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Process j:

<table>
<thead>
<tr>
<th>Address</th>
<th>EXEC</th>
<th>READ</th>
<th>WRITE</th>
<th>SUP</th>
</tr>
</thead>
<tbody>
<tr>
<td>PP 9</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>PP 6</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>PP 4</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Physical Address Space
Today

- Address spaces
- VM as a tool for caching
- VM as a tool for memory management
- VM as a tool for memory protection
- Address translation

VM Address Translation

- Virtual Address Space
  - \( V = \{0, 1, \ldots, N-1\} \)
- Physical Address Space
  - \( P = \{0, 1, \ldots, M-1\} \)
- Address Translation
  - \( MAP: V \rightarrow P \cup \{\emptyset\} \)
  - For virtual address \( a \):
    - \( MAP(a) = a' \) if data at virtual address \( a \) is at physical address \( a' \) in \( P \)
    - \( MAP(a) = \emptyset \) if data at virtual address \( a \) is not in physical memory
      - Either invalid or stored on disk

Summary of Address Translation Symbols

- Basic Parameters
  - \( N = 2^n \): Number of addresses in virtual address space
  - \( M = 2^m \): Number of addresses in physical address space
  - \( P = 2^p \): Page size (bytes)
- Components of the virtual address (VA)
  - TLBI: TLB index
  - TLBT: TLB tag
  - VPO: Virtual page offset
  - VPN: Virtual page number
- Components of the physical address (PA)
  - PPO: Physical page offset (same as VPO)
  - PPN: Physical page number

Address Translation With a Page Table

- Page table base register (PTBR)
- Physical page table address for the current process
- Valid bit = 1: Page in memory
- Valid bit = 0: Page not in memory (page fault)
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

MMU
CPU Chip
VA
PTE
PA
Data
PM
PTEA

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

PTEA
PTE
PA

Integrating VM and Cache

VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address

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 small L1 delay
• Solution: Translation Lookaside Buffer (TLB)
  – Small set-associative hardware cache in MMU
  – Maps virtual page numbers to physical page numbers
  – Contains complete page table entries for small number of pages
Accessing the TLB

- MMU uses the VPN portion of the virtual address to access the TLB:

\[ \text{TLB tag (TLBT)} \]
\[ \text{TLB index (TLBI)} \]
\[ \text{VPO} \]

\( T = 2^t \) sets

TLB matches tag of line within set

\( p^{-1} \)

\( p^t \)

\( p^t \)

\( p^{-1} \)

\( 0 \)

Set 0

\[ \begin{array}{c}
\text{tag} \\
\text{PTE}
\end{array} \]

\[ \begin{array}{c}
\text{tag} \\
\text{PTE}
\end{array} \]

Set 1

\[ \begin{array}{c}
\text{tag} \\
\text{PTE}
\end{array} \]

\[ \begin{array}{c}
\text{tag} \\
\text{PTE}
\end{array} \]

Set \( T-1 \)

\[ \begin{array}{c}
\text{tag} \\
\text{PTE}
\end{array} \]

\[ \begin{array}{c}
\text{tag} \\
\text{PTE}
\end{array} \]

TLBI selects the set

Breakout

- Your machine has a 32-bit virtual address space, with a virtual memory page size of 8 KBytes. The translation look-aside buffer (TLB) is organized to have 64 total entries and is 4-way set associative.

- How many tag bits are required in the TLB in order to find the virtual to physical page translation for a virtual address?
Multi-Level Page Tables

- Suppose:
  - 4KB ($2^{12}$) page size, 48-bit address space, 8-byte PTE

- Problem:
  - Would need a 512 GB page table!
    - $2^{48} \times 2^{12} \times 2^3 = 2^{39}$ bytes

- Common solution: Multi-level page table
- Example: 2-level page table
  - Level 1 table: each PTE points to a page table (always memory resident)
  - Level 2 table: each PTE points to a page (paged in and out like any other data)

Translating with a k-level Page Table

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