Virtual Memory: Concepts

15-213/18-213/14-513/15-513/18-613:
Introduction to Computer Systems
17th Lecture, October 22, 2019
Hmmm, How Does This Work?!
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)

![Diagram showing virtual memory and physical memory]
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
  - Time to load block from disk > 1ms (> 1 million clock cycles)
    - CPU can do a lot of computation during that time

- **Consequences**
  - Large page (block) size: typically 4 KB
    - Linux “huge pages” are 2 MB (default) to 1 GB
  - 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 Table Diagram]

- Physical page number or disk address
- Memory resident page table (DRAM)
- Virtual memory (disk)
- Physical memory (DRAM)
**Page Hit**

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

![Diagram showing memory resident page table and virtual memory](image-url)
Page Fault

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

---

**Diagram:**
- Virtual address
- Physical page number or disk address
- Valid
- Memory resident page table (DRAM)
- Physical memory (DRAM)
- Virtual memory (disk)
Triggering a Page Fault

- User writes to memory location

- That portion (page) of user’s memory is currently on disk

- MMU triggers page fault exception
  - (More details in later lecture)
  - Raise privilege level to supervisor mode
  - Causes procedure call to software page fault handler

User code

Kernel code

```
int a[1000];
main ()
{
    a[500] = 13;
}
```

```
80483b7:  c7 05 10 9d 04 08 0d  movl  $0xd,0x8049d10
```

movl  Exception: page fault

Execute page fault handler
Handling Page Fault

- Page miss causes page fault (an exception)
Handling Page Fault

- Page miss causes page fault (an exception)
- Page fault handler selects a victim to be evicted (here VP 4)
Handling Page Fault

- Page miss causes page fault (an exception)
- Page fault handler selects a victim to be evicted (here VP 4)
Handling Page Fault

- Page miss causes page fault (an 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*
Completing page fault

- Page fault handler executes return from interrupt (iret) instruction
  - Like ret instruction, but also restores privilege level
  - Return to instruction that caused fault
  - But, this time there is no page fault

```c
int a[1000];
main ()
{
   a[500] = 13;
}
```

User code

Kernel code

80483b7: c7 05 10 9d 04 08 0d movl $0xd,0x8049d10

Exception: page fault

Copy page from disk to memory

Return and reexecute movl
Allocating Pages

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

- Subsequent miss will bring it into memory.
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 cold misses)

- If (working set size > main memory size)
  - **Thrashing**: Performance meltdown where pages are swapped (copied) in and out continuously
  - If multiple processes run at the same time, thrashing occurs if their total working set size > main memory size
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
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](image)

![Virtual Address Space for Process 2](image)

![Address translation](image)

![Physical Address Space (DRAM)](image)
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
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 Protection

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

### Process i:

<table>
<thead>
<tr>
<th></th>
<th>SUP</th>
<th>READ</th>
<th>WRITE</th>
<th>EXEC</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 0:</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>PP 6</td>
</tr>
<tr>
<td>VP 1:</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 4</td>
</tr>
<tr>
<td>VP 2:</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>PP 2</td>
</tr>
</tbody>
</table>

### Process j:

<table>
<thead>
<tr>
<th></th>
<th>SUP</th>
<th>READ</th>
<th>WRITE</th>
<th>EXEC</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 0:</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>PP 9</td>
</tr>
<tr>
<td>VP 1:</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 6</td>
</tr>
<tr>
<td>VP 2:</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 11</td>
</tr>
</tbody>
</table>

**Physical Address Space**

- PP 2
- PP 4
- PP 6
- PP 8
- PP 9
- PP 11

**SUP: requires kernel mode**
Quiz Time!

Check out:

https://canvas.cmu.edu/courses/10968
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)**
  - **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

Virtual address

Page table base register (PTBR) (CR3 in x86)

Physical page table address for the current process

Valid bit = 0:
Page not in memory (page fault)

Valid bit = 1

Physical address

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
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
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
Accessing the TLB

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

```
TLB tag (TLBT) | TLB index (TLBI) | VPO
```

\[
\begin{align*}
\text{VPN} &= (n-1)(p+t) + p + p-1 + 0 \\
T &= 2^t \text{ sets}
\end{align*}
\]

TLBT matches tag of line within set

TLBI selects the set
TLB Hit

A TLB hit eliminates a cache/memory access
A TLB miss incurs an additional cache/memory access (the PTE).
Fortunately, TLB misses are rare. Why?
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)
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**

- PTE 0
  - ... (null)
  - PTE 1023

**Virtual memory**

- VP 0
  - VP 1023
  - VP 1024
  - VP 2047

- Gap

- 1023 unallocated pages
- VP 9215

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

64 bit addresses, 8KB pages, 8-byte PTEs

Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition
Translating with a k-level Page Table

Page table base register (PTBR)

VIRTUAL ADDRESS

 VPN 1 | VPN 2 | ... | VPN k

the Level 1 page table | a Level 2 page table | ... | a Level k page table

PPN

PHYSICAL ADDRESS

PPN | PPO
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

- **Implemented via combination of hardware & software**
  - MMU, TLB, exception handling mechanisms part of hardware
  - Page fault handlers, TLB management performed in software