CSCI-UA.0201

Computer Systems Organization

Virtual Memory

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com

Some slides adapted (and slightly modified) from:
• Clark Barrett
• Jinyang Li
• Randy Bryant
• Dave O’Hallaron
Have you ever thought ...

- Why the text segment (Do you remember heap, stack, text, and data?) of any process starts at the same address?
- Why don’t you run out of memory even if the sum of memory requirement of each program you are running at the same time exceeds the amount of memory you have in your machine?
- The address is 64-bit, in 64-bit machines. It accesses memory from 0 to $2^{64} - 1$ which is way more than the amount of memory you have on your machine?

How come?
Use part of the disk as extension to the memory!
Virtual Addressing

- Used in all modern machines.

**MMU**: Memory Management Unit

### Diagram

- **CPU Chip**
- **CPU**: Virtual address (VA) 4100
- **MMU**: Physical address (PA) 4
- **Main memory**:
  - 0:
  - 1:
  - 2:
  - 3:
  - 4: (Selected)
  - 5:
  - 6:
  - 7:
  - 8:
  - ...:
  - M-1:
Why Virtual Memory (VM)?

• Simplified memory management
  – Each process gets an exclusive linear address space (that is, each process thinks that it has the whole memory for itself).

• Process Isolation
  – Different processes have different virtual address spaces
  – One process can't interfere with another's memory (that is, each process does not even know that other processes exist and are sharing the memory with it!)

• Uses main memory efficiently
  – Parts of a process not needed are sent to the disk

Any program currently running on your machine is called a process.
Address translation

- Key idea of VM: each process has its own virtual address space

Granularity of mapping?

- **Byte-level**: Map each byte in VA to a byte in PA → too expensive
- **Page-level**: Map each consecutive $2^p$ byte address range in VA to a $2^p$ byte address range in PA

Valid address:
- [0, $2^{64}$) for 64-bit machine

Valid address:
- [0, $2^m$) depending on how much memory your machine has

<table>
<thead>
<tr>
<th>VA</th>
<th>PA</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0x08</td>
<td>0x98</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>invalid</td>
</tr>
</tbody>
</table>
Address Translation w/ a Page Table

Virtual address (i.e. address generated by CPU)

Virtual page number (VPN)  Virtual page offset (VPO)

Page table base register (PTBR)

Page table

Valid  Physical page number (PPN)

PTE (page table entry)

Physical page number (PPN)  Physical page offset (PPO)

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

How OS tells hardware where to find the page table

Physical address (i.e. the real address used to access the memory)
Page Table Base Register (PTBR)

- The operating system maintains information about each process in a process control block.
- The page table base address for the process is stored there.
- The operating system loads this address into the PTBR whenever a process is scheduled for execution.
- Only the kernel (i.e. the OS) can access PTBR
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
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 in kernel**

If VA is invalid, then kill process (SIGSEGV)
If VA has been paged out to disk, then swaps in faulted page, update page table, resume faulted process
There are two challenges

- **Speed**: VA to PA translation means we need to access the memory twice for each CPU memory request!

- **Size**: page table can be huge. And we have a page table per process.
Tackling the Speed Problem
Speeding up Translation with a TLB

• VA→PA translation can be expensive
  – One additional memory reference for every normal memory reference!
  – How about this: Page table entries (PTEs) are cached in L1 like others? …. Not very good idea.
    • PTEs may be evicted by other data references
    • PTE hit still requires a small L1 delay

• Solution: Translation Lookaside Buffer (TLB)
  – Small hardware cache inside the MMU (i.e. on chip)
  – Maps virtual page numbers to physical page numbers
  – Contains complete page table entries for small number of pages
A TLB hit eliminates a memory access
A TLB miss incurs an additional memory access (the PTE).
 Fortunately, TLB misses are rare.
Tackling the size (i.e. memory requirement of page tables) problem
Reduce Page Table Size

• 4KB-page, 48-bit address space (x86 machines zero the 16 MSBits from the 64 bit address), 8-byte PTE
• Size of page table needed?
  – \(2^{48-12} \times 2^3 = 2^{39} = 512\) GB
• Wasteful: most PTEs are invalid
  – i.e. not used
• Solution: multi-level page table
  – Example: 2-level page table
    • Level 1 table: each PTE points to a page table
    • Level 2 table: each PTE points to a page
Why Two-level Page Table Reduces Memory Requirement?

• If a PTE in the level 1 table is null, then the corresponding level 2 page table does not even have to exist.

• Only the level 1 table needs to be in main memory at all times.

• The level 2 page tables can be created and paged in and out by the VM system as they are needed.
We can now say that VM is useful for:

- Memory management
- Simplified linking and loading
- Memory protection
Memory management and protection

- Each process has an **exclusive VA space**
  - One process cannot overwrite another one’s memory!
- **Sharing** among processes
  - Map different virtual pages to the same physical page
Simplified Linking and Loading

• Linking
  – Each program has similar virtual address space
  – Code, stack, and shared libraries always start at the same address

• Loading
  – Starting a program execution causes kernel to allocate virtual pages
  – Kernel copies .text and .data sections, page by page, from disk to memory
Memory Protection

- How to protect shared pages from corruption?
  - E.g. bad process overwrites shared kernel code/data, shared libc code etc.
- Extend PTEs with permission bits

<table>
<thead>
<tr>
<th></th>
<th>Process i:</th>
<th></th>
<th>Process j:</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 0:</td>
<td>SUP</td>
<td>READ</td>
<td>WRITE</td>
</tr>
<tr>
<td>VP 1:</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>VP 2:</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>VP 2:</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>VP 0:</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>VP 1:</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>VP 2:</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

SUP: whether processes must be running in kernel (supervisor) mode to access the page.
Toy Memory System Example

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

![Addressing Diagram]

- **VPN** (Virtual Page Number)
- **VPO** (Virtual Page Offset)
- **PPN** (Physical Page Number)
- **PPO** (Physical Page Offset)
Toy Memory System Page Table

<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>28</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>02</td>
<td>33</td>
<td>1</td>
</tr>
<tr>
<td>03</td>
<td>02</td>
<td>1</td>
</tr>
<tr>
<td>04</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>05</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>06</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>07</td>
<td>–</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>08</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>09</td>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>0A</td>
<td>09</td>
<td>1</td>
</tr>
<tr>
<td>0B</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>0C</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>0D</td>
<td>2D</td>
<td>1</td>
</tr>
<tr>
<td>0E</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>0F</td>
<td>0D</td>
<td>1</td>
</tr>
</tbody>
</table>

1-level page table: How many PTEs?
Address Translation Example

**Virtual Address:** 0x0354

What’s the corresponding PPN? Physical address?
Integrating Caches and VM

• Shall we access caches with virtual address? or physical address?
• With virtual address:
  – good: cache accessed as soon as possible
  – bad: aliasing (i.e. two different virtual addresses can map to the same cache block and vice versa) \( \rightarrow \) more complicated cache needed to handle this.

• With physical address:
  – good: no aliasing
  – bad: must wait for address translation (VA \( \rightarrow \) PA) before cache can be accessed.
Conclusions

• In this lecture we have seen VM in action.
• It is important to know how the following pieces interact:
  – CPU, MMU, DRAM, Cache, Kernel
• 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 moving memory pages between disk and memory