Lecture 17: Virtual Memory: Concepts

Some slides adapted (and slightly modified) from:
- Clark Barrett
- Jinyang Li
- Randy Bryant
- Dave O’Hallaron
All problems in CS can be solved by another level of indirection

Butler Lampson, co-inventor of PC
Virtual Addressing

- Used in all modern servers, desktops, and laptops
Why Virtual Memory (VM)?

• **Simplified memory management**
  – Each process gets an ``exclusive'' linear address space

• **Process Isolation**
  – Different processes have different virtual address spaces
  – One process can’t interfere with another’s memory

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

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

Strawman view of addr translation

<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>..</td>
</tr>
<tr>
<td>..</td>
<td>invalid</td>
</tr>
</tbody>
</table>

Valid address:
- [0, 2^32) for 32-bit machine
- [0, 2^64) for 64-bit machine

Granularity of mapping?
- Byte-level: Map each byte in VA to a byte in PA
- Page-level: Map each consecutive 2^p-byte address range in VA to a 2^p-byte address range in PA
Address Translation w/ a Page Table

How kernel tells h/w where to find the page table

Page table base register (PTBR)

Page table address for process

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

Virtual address

Physical address
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 can access PTBR
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
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
Speeding up Translation with a TLB

- VA→PA translation can be expensive
  - One additional memory reference for every normal memory reference!
  - Not too bad? Page table entries (PTEs) are cached in L1 like others
    - PTEs may be evicted by other data references
    - PTE hit still requires a small L1 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
A TLB hit eliminates a memory access
A TLB miss incurs an additional memory access (the PTE)
Fortunately, TLB misses are rare. Why?
Reduce Page Table Size

- 4KB-page, 48-bit address space, 8-byte PTE
- Size of page table needed?
  \[ 2^{48-12} \times 2^3 = 2^{39} = 512 \text{ GB} \]
- Wasteful: most PTEs are invalid...
- 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
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
- PTE 1
- PTE 2 (null)
- PTE 3 (null)
- PTE 4 (null)
- PTE 5 (null)
- PTE 6 (null)
- PTE 7 (null)
- PTE 8
- 1023 null PTEs
- 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

32 bit addresses, 4KB pages, 4-byte PTEs
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.
For memory management and protection

For caching
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

Virtual Address Space for Process 1:

Virtual Address Space for Process 2:

Physical Address Space (DRAM)

(e.g., libc code, kernel)
Simplified Linking and Loading

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

• **Loading**
  - `execve()` 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

### Process i:

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

### Process j:

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

**SUP:** whether processes must be running in kernel (supervisor) mode to access the page.
VM as a Tool for Caching

- Not all processes' valid VA pages fit in physical memory
- Key idea: treat DRAM-resident pages as a cache of on-disk pages

![Diagram showing virtual memory and physical memory states](image-url)
VM for Caching

- **General mechanism:**
  - On page fault, load corresponding on-disk page to memory, evict a previously memory-resident to disk, set appropriate PTE entry
- **Which entity should be performing this task?**
  - User-level process? OS? Hardware?
- **VM Caching policy (more sophisticated than CPU cache)**
  - Fully associative: any VP can be mapped to any PP
  - Write-back
Page Hit

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

![Diagram of virtual memory and physical memory structures](image-url)
Page Fault

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

Virtual address

Physical page number or disk address

Memory resident page table (DRAM)

Physical memory (DRAM)

Virtual memory (disk)

Kernel figures out where to find the corresponding on-disk page
Handling Page Fault

- Page miss causes page fault (an exception)

![Diagram showing the handling of page faults with virtual memory and page tables.]
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!

Memory resident page table (DRAM)

Virtual address

Physical page number or disk address

Valid

PTE 0

0

1

1

1

0

0

0

null

PTE 7

0

1

Memory resident page table (DRAM)

Physical memory (DRAM)

PP 0

VP 1

VP 2

VP 7

VP 3

PP 3

Virtual memory (disk)

VP 1

VP 2

VP 3

VP 4

VP 6

VP 7
Why should VM caching work?

• 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 (working set sizes > main memory size)
  – Thrashing: Performance meltdown where pages are swapped (copied) in and out continuously
Conclusions

• 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
  – Simplifies memory management & protection
  – Uses memory efficiently by caching virtual memory pages