Memory Management I

Paging

COMS W4118

References: Operating Systems Concepts (9e), Linux Kernel Development, previous W4118s

Copyright notice: care has been taken to use only those web images deemed by the instructor to be in the public domain. If you see a copyrighted image on any slide and are the copyright owner, please contact the instructor. It will be removed.
Outline

• Overview

• Paging

• TLB

• Page sharing
Multiple address spaces co-exist

Logical view

Physical view

max

AS1

0

max

AS2

0

max

AS3

0

PHYSTOP

0
Memory management wish-list

• Sharing
  – multiple processes coexist in main memory

• Transparency
  – Processes are not aware that memory is shared
  – Run regardless of number/locations of other processes

• Protection
  – Cannot access data of OS or other processes

• Efficiency: should have reasonable performance
  – Purpose of sharing is to increase efficiency
  – Do not waste CPU or memory resources (fragmentation)
Memory Management Unit (MMU)

- Map program-generated address (virtual address) to hardware address (physical address) dynamically at every reference
- Check range and permissions
- Programmed by OS
x86 address translation

• CPU generates virtual address (seg, offset)
  – Given to segmentation unit
    • Which produces linear addresses
  – Linear address given to paging unit
    • Which generates physical address in main memory
Segmentation

• Divide virtual address space into separate logical segments; each is part of physical mem
x86 segmentation hardware

Logical address

selector

offset

Global descriptor table

base limit perm
base limit perm
base limit perm

Compute: base + offset
Check: offset <= limit
Check: permissions

Linear address
Paging overview

• Goal
  – Eliminate fragmentation due to large segments
  – Don’t allocate memory that will not be used
  – Enable fine-grained sharing

• Paging: divide memory into fixed-sized pages
  – For both virtual and physical memory

• Another terminology
  – A virtual page: page
  – A physical page: frame
Page translation

- Address bits = page number + page offset
- Translate virtual page number (vpn) to physical page (frame) number (ppn/pfn) using page table

\[
\text{pa} = \text{page\_table}[\text{va/pg\_sz}] + \text{va}\%\text{pg\_sz}
\]
Page translation example

Virtual Memory

Page 0
Page 1
Page 2
Page 3

Page table

0 1
1 4
2 3
3 7

Physical Memory

Page 0
Page 2
Page 1
Page 3
• 8-bit virtual address, 10-bit physical address, each page is 64 bytes

1. How many virtual pages?
   – $2^8 / 64 = 4$ virtual pages

2. How many physical pages?
   – $2^{10}/64 = 16$ physical pages

3. How many entries in page table?
   – Page table contains 4 entries

4. Given page table = [2, 5, 1, 8], what’s the physical address for virtual address 241?
   – $241 = 11110001b$
   – $241/64 = 3 = 11b$
   – $241\%64 = 49 = 110001b$
   – page_table[3] = 8 = 1000b
   – Physical address = $8 \times 64 + 49 = 561 = 1000110001b$
Page translation exercise

m-bit virtual address, n-bit physical address, k-bit page size

- # of virtual pages: $2^{(m-k)}$
- # of physical pages: $2^{(n-k)}$
- # of entries in page table: $2^{(m-k)}$
- $\text{vpn} = \text{va} / 2^k$
- $\text{offset} = \text{va} \% 2^k$
- $\text{ppn} = \text{page_table}[\text{vpn}]$
- $\text{pa} = \text{ppn} * 2^k + \text{offset}$
Page protection

• Implemented by associating protection bits with each virtual page in page table

• Why do we need protection bits?

• Protection bits
  – present bit: map to a valid physical page?
  – read/write/execute bits: can read/write/execute?
  – user bit: can access in user mode?
  – x86: PTE_P, PTE_W, PTE_U

• Checked by MMU on each memory access
• What kind of pages?

Virtual Memory

Page table

Physical Memory

Page 0
Page 1
Page 3

0 1 101
1 4 110
2 3 000
3 7 111

pwu
• Page table is stored in memory
  – Page table base register (PTBR) points to the base of page table
    • x86: cr3
  – OS stores base in process control block (PCB)
  – OS switches PTBR on each context switch

• Problem: each data/instruction access requires two memory accesses
  – Extra memory access for page table
Page table size issues

• Given:
  – A 32 bit address space (4 GB)
  – 4 KB pages
  – A page table entry of 4 bytes

• Implication: page table is 4 MB per process!

• Observation: address space are often sparse
  – Few programs use all of $2^{32}$ bytes

• Change page table structures to save memory
  – Trade translation time for page table space
Hierarchical page table

• Break up virtual address space into multiple page tables at different levels

![Diagram of hierarchical page table]

outer page table → page table → page of page table → memory

- Page table
- Page of page table
- Outer page table

- Memory
Hierarchical page tables

Logical address

\[ p_1 \quad p_2 \quad d \]

Outer page table

Page of page table

\( p_1 \)

\( p_2 \)

\( d \)
x86 page translation with 4KB pages

• 32-bit address space, 4 KB page
  – 4KB page \(\Rightarrow\) 12 bits for page offset

• How many bits for \(2^{nd}\)-level page table?
  – Desirable to fit a \(2^{nd}\)-level page table in one page
  – 4KB/4B = 1024 \(\Rightarrow\) 10 bits for \(2^{nd}\)-level page table

• Address bits for top-level page table: \(32 – 10 – 12 = 10\)

<table>
<thead>
<tr>
<th>page number</th>
<th>page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>(p_1)</td>
<td>(p_2)</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>
x86 paging architecture

- A better picture here (page 26):
Avoiding extra memory accesses

- **Observation:** locality
  - **Temporal:** access locations accessed just now
  - **Spatial:** access locations adjacent to locations accessed just now
  - Process often needs only a small number of vpn→ppn mappings at any moment!

- **Fast-lookup hardware cache called associative memory or translation look-aside buffers (TLBs)**
  - Fast parallel search (CPU speed)
  - Small
Paging hardware with TLB
Effective access time with TLB

- Assume memory cycle time is **1 unit time**
- TLB Lookup time = \( \varepsilon \)
- TLB Hit ratio = \( \alpha \)
  - Percentage of times that a vpn\( \rightarrow \)ppn mapping is found in TLB

- **Effective Access Time (EAT)**

\[
EAT = (1 + \varepsilon) \alpha + (2 + \varepsilon)(1 - \alpha) \\
= \alpha + \varepsilon \alpha + 2 + \varepsilon - \varepsilon \alpha - 2\alpha \\
= 2 + \varepsilon - \alpha
\]
TLB and context switches

• What happens to TLB on context switches?

• Option 1: flush entire TLB
  – x86
    • “load cr3” (load page table base) flushes TLB

• Option 2: attach process ID to TLB entries
  – ASID: Address Space Identifier
  – MIPS, SPARC

• x86 “INVLPG addr” invalidates one TLB entry
Motivation for page sharing

• **Efficient communication.** Processes communicate by write to shared pages

• **Memory efficiency.** One copy of read-only code/data shared among processes
  – Example 1: multiple instances of the shell program
  – Example 2: copy-on-write fork. Parent and child processes share pages right after fork; copy only when either writes to a page
Page sharing example

- Process $P_1$:
  - Page_table:
    - Page 0: 3
    - Page 1: 4
    - Page 2: 6
    - Page 3: 1

- Process $P_2$:
  - Page_table:
    - Page 4: 3
    - Page 5: 4
    - Page 6: 6
    - Page 7: 7

- Process $P_3$:
  - Page_table:
    - Page 8: 3
    - Page 9: 4
    - Page 10: 6
    - Page 11: 2

- Pages:
  - 0: data 1
  - 1: data 3
  - 2: ed 1
  - 3: ed 2
  - 4: ed 3
  - 5: data 2
A cool trick: copy-on-write

• In `fork()`, parent and child often share significant amount of memory
  – Expensive to copy all pages

• COW Idea: exploit VA to PA indirection
  – Instead of copying all pages, share them
  – If either process writes to shared pages, only then is the page copied

• Real: used in virtually all modern OS
Before Process 1 Modifies Page C
After Process 1 Modifies Page C