Memory Management I
Segmentation and Paging

COMS W4118
Prof. Kaustubh R. Joshi
krj@cs.columbia.edu

http://www.cs.columbia.edu/~krj/os

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

• Memory management goals

• Segmentation

• Paging

• TLB

• Page sharing
Uni- v.s. multi-programming

• Simple uniprogramming with a single segment per process

• Uniprogramming disadvantages
  – Only one process can run a time
  – Process can destroy OS

• Want multiprogramming!
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
A Simple MMU: Base/Limit Registers

- **Base** and **limit registers** define logical address space
- CPU checks every memory access generated in user mode to be sure it is between base and limit for that user
A better MMU: Relocatable Code

- Problem with base limit register solution?
  - Need to know address at which program will be before hand: linker/loader must rewrite instructions
  - Can’t change location once loaded, prone to fragmentation
- Solution: add a relocation register
  - Programmer uses addresses that are offsets from base
  - Hardware adds actual value of base at runtime to get final address
Problems with contiguous allocation

- Partition per program: how big should each partition be?
  - Entire size of address space? Impractical
  - How much program actually uses? May not know in advance
- Have to be conservative
  - Too small: must reallocate and move program (expensive)
  - Too big: wasted memory
- Fragmentation over time
  - **Hole** – block of available memory; scattered throughout memory
  - Need hole large enough to accommodate new processes

<table>
<thead>
<tr>
<th>OS</th>
<th>process 5</th>
<th>process 8</th>
<th>process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS</td>
<td>process 5</td>
<td>process 2</td>
<td></td>
</tr>
<tr>
<td>OS</td>
<td>process 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OS</td>
<td>process 5</td>
<td>process 9</td>
<td>process 2</td>
</tr>
<tr>
<td>OS</td>
<td>process 5</td>
<td>process 9</td>
<td>process 10</td>
</tr>
<tr>
<td>OS</td>
<td>process 5</td>
<td>process 9</td>
<td>process 2</td>
</tr>
</tbody>
</table>
Outline

• Memory management goals
  
  • Segmentation
  
  • Paging
  
  • TLB
  
  • Page sharing
Segmentation

- Divide virtual address space into separate logical segments; each is part of physical mem
Segmentation translation

- Virtual address: `<segment-number, offset>`

- Segment table maps segment number to segment information
  - **Base**: starting address of the segment in physical memory
  - **Limit**: length of the segment
  - Addition metadata includes *protection bits*

- Limit & protection checked on each access
80x86 segment selector

- Logical address: *segment selector* + offset
- **Segment selector** stored in segment registers (16-bit)
  - *cs*: code segment selector
  - *ss*: stack segment selector
  - *ds*: data segment selector
  - *es, fs, gs*

- Segment register can be implicitly or explicitly specified
  - Implicit by type of memory reference (*jmp*)
    - mov $8049780, %eax // implicitly use *ds*
  - Through special registers (*cs, ss, es, ds, fs, gs on x86*)
    - mov %ss:$8049780, %eax // explicitly use *ss*

- **Support for segmentation removed in x86-64**
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

3/27/13
COMS W4118. Spring 2013, Columbia University. Instructor: Dr. Kaustubh Joshi, AT&T Labs.
Linux Segments

• Not much to see
  – Rely mainly on paging (next topic)
  – Basic common segments that span entire memory

• Different permissions dependent on use
  – Kernel code: read + execute in kernel mode
  – Kernel data: writable in kernel mode
  – User code: readable + executable in user mode
  – User data: writable in user mode
  – These are all null mappings
    • Map to [0, 0xFFFFFFFF]
    • Linear address = Offset
Pros and cons of segmentation

• Advantages
  – Segment sharing
  – Easier to relocate segment than entire program
  – Avoids allocating unused memory
  – Flexible protection
  – Efficient translation
    • Segment table small → fit in MMU

• Disadvantages
  – Segments have variable lengths → how to fit?
  – Segments can be large → fragmentation
Outline

- Memory management goals
- Segmentation
- Paging
- TLB
- Page sharing
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

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

• 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 * 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: \(2^{(m-k)}\)
• \(\text{vpn} = \text{va} / 2^k\)
• \(\text{offset} = \text{va} \mod 2^k\)
• \(\text{ppn} = \text{page\_table}[\text{vpn}]\)
• \(\text{pa} = \text{ppn} \times 2^k + \text{offset}\)
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

<table>
<thead>
<tr>
<th>Page 0</th>
<th>Page 1</th>
<th>Page 3</th>
</tr>
</thead>
</table>

Page table

<table>
<thead>
<tr>
<th>pwu</th>
<th>0</th>
<th>101</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>110</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>000</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>111</td>
</tr>
</tbody>
</table>

Physical Memory

Page 0

Page 1

Page 3

3/27/13

COMS W4118. Spring 2013, Columbia University. Instructor: Dr. Kaustubh Joshi, AT&T Labs.
Page allocation

• Free page management
  – E.g., can put page on a free list

• Allocation policy
  – E.g., one page at a time, from head of free list

• We’ll see allocation policies later

2, 3, 6, 5, 0
Implementation of page table

• 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
Page table structures

- Hierarchical paging
- Hashed page tables
- Inverted page tables
• Break up virtual address space into multiple page tables at different levels
Hierarchical page tables

![Diagram showing hierarchical page tables]

A logical address is divided into three parts: $p_1$, $p_2$, and $d$. $p_1$ is used to access the outer page table, which in turn uses $p_2$ to access the page of page table.
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\textsuperscript{nd}-level page table?
  – Desirable to fit a 2\textsuperscript{nd}-level page table in one page
  – $4\text{KB}/4\text{B} = 1024$ $\rightarrow$ 10 bits for 2\textsuperscript{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
Intel x86-64 Paging

- Current generation Intel x86 architecture
- 64 bits is ginormous (> 16 exabytes)
- In practice only implement 48 bit addressing
  - Page sizes of 4 KB, 2 MB, 1 GB
  - Four levels of paging hierarchy
- Can also use PAE so virtual addresses are 48 bits and physical addresses are 52 bits
ARM Paging

- 32-bit CPU
- 4 KB and 16 KB pages
- 1 MB and 16 MB pages (termed sections)
- One-level paging for sections, two-level for smaller pages
- Two levels of TLBs
  - Outer level has two micro TLBs (one data, one instruction)
  - Inner is single main TLB
  - First inner is checked, on miss outers are checked, and on miss page table walk performed by CPU
Four-level Paging in Linux

• Abstracts paging across architecture
  – pgd: page global directory
  – pud: page upper directory
  – pmd: page middle directory
  – pte: page table entry

• Each architecture defines
  – Size of each directory, number of entries, bits
  – Bypass levels that arch doesn’t have
Other page table structures

• Hierarchical paging

• Hashed page tables

• Inverted page tables
Hashed page table

• Common in address spaces > 32 bits

• Page table contains a chain of elements hashing to the same location

• On page translation
  – Hash virtual page number into page table
  – Search chain for a match on virtual page number
Hashed page table example

Hash function

Physical address

Logical address

Hash table

Physical memory
Inverted page table

• One entry for each real page of memory
  – Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page

• Same page table shared by all processes
  – Need owner information

• Can use hash table to limit the search to one or at most a few page-table entries
Inverted page table example
Outline

• Memory management goals

• Segmentation

• Paging

• TLB

• Page sharing
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$\rightarrow$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

CPU

logical address

p d

page number

frame number

TLB

TLB hit

physical address

f d

physical memory

page table
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 Miss

• Depending on the architecture, TLB misses are handled in either hardware or software

• Hardware (CISC: x86)
  – Pros: hardware doesn’t have to trust OS!
  – Cons: complex hardware, inflexible

• Software (RISC: MIPS, SPARC)
  – In effect, TLB is hardware page table
  – Pros: simple hardware, flexible
  – Cons: code may have bug!
Reducing misses: TLB Reach

• Increase size of TLB
  – Content addressable memory (CAM) is expensive

• Increase amount of memory accessible from the TLB
  – TLB Reach = (TLB Size) X (Page Size)
  – Ideally, equal to working set
  – Otherwise lots of page faults

• Increase page size
  – More reach for same TLB size
  – Increase in fragmentation as well

• Provide multiple page sizes
  – Applications can choose which size fits their access pattern
  – Doesn’t increase fragmentation
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 "INVLPD addr" invalidates one TLB entry
### Address Space IDs (ASID)

#### Mechanism to reduce frequency of TLB invalidations

**Without ASID:**

<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
<th>valid</th>
<th>prot</th>
</tr>
</thead>
</table>
| [ 0  |  10 |  1    | rwx  ]
| [ ---- | ---- |  0    | ---- ]
| [ 0  |  17 |  1    | rwx  ]
| [ ---- | ---- |  0    | ---- ]

**With ASID:**

<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
<th>valid</th>
<th>prot</th>
<th>ASID</th>
</tr>
</thead>
</table>
| [ 0  |  10 |  1    | rwx  |  1   ]
| [ ---- | ---- |  0    | ---- | ---- ]
| [ 0  |  17 |  1    | rwx  |  2   ]
| [ ---- | ---- |  0    | ---- | ---- ]
Choosing a page size

• Many CPUs support multiple page sizes
• Page size selection affects (or is affected by):
  – Fragmentation?
    • Smaller is better.
  – Page table size?
    • Bigger is more efficient.
  – I/O overhead?
    • Larger is better (fewer seeks).
  – Resolution (locality)?
    • Smaller is better.
  – Number of page faults?
    • Larger or smaller could be better. 1 page per byte vs. 1 page for entire mem.
  – TLB size and effectiveness?
    • Larger is better.
• On average, growing over time
Outline

• Memory management goals

• Segmentation

• Paging

• TLB

• Page sharing
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 P₁**
  - Page table for P₁
  - **Eds:**
    - ed 1
    - ed 2
    - ed 3
  - **Data:**
    - data 1

- **Process P₂**
  - Page table for P₂
  - **Eds:**
    - ed 1
    - ed 2
    - ed 3
  - **Data:**
    - data 2

- **Process P₃**
  - Page table for P₃
  - **Eds:**
    - ed 1
    - ed 2
    - ed 3
  - **Data:**
    - data 3

- Page table for P₁:
  - Page 0: ed 1
  - Page 1: ed 2
  - Page 2: ed 3
  - Page 3: data 1

- Page table for P₂:
  - Page 0: ed 1
  - Page 1: ed 2
  - Page 2: ed 3
  - Page 3: data 2

- Page table for P₃:
  - Page 0: ed 1
  - Page 1: ed 2
  - Page 2: ed 3
  - Page 3: data 3
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
How to implement COW?

• (Ab)use page protection

• Mark pages as read-only in both parent and child address space

• On write, page fault occurs

• In OS page fault handler, distinguish COW fault from real fault
  — How?

• Copy page and update page table if COW fault
  — Always copy page?
Before Process 1 Modifies Page C
After Process 1 Modifies Page C