W4118: segmentation and paging

Instructor: Junfeng Yang

Outline

- Memory management goals
- Segmentation
- Paging
- TLB
Multiprogramming

- Simple uniprogramming with a single segment per process

- Uniprogramming disadvantages
  - Only one process can run at a time
  - Process can destroy OS

- Want multiprogramming!
Multiple address spaces co-exist

AS1

AS2

AS3

Logical view

Physical view
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
    - Internal fragmentation and external fragmentation
Outline

- Memory management goals
- Segmentation
- Paging
- TLB
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
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
x86 segmentation hardware

Logical address

selector

offset

Global descriptor table

base  limit  perm
base  limit  perm
base  limit  perm

Linear address

Logical address

selector

offset

Global descriptor table

base  limit  perm
base  limit  perm
base  limit  perm

Linear address
xv6 segments

- vm.c, ksegment()

- Kernel code: readable + executable 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

- Kernel CPU: shortcuts to per-CPU data
  - Base: &c->cpu
  - Limit: 8 bytes
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 $\rightarrow$ fit in MMU

- **Disadvantages**
  - Segments have variable lengths $\rightarrow$ dynamic allocation (best fit? first fit?)
  - External fragmentation: wasted memory
Outline

- Memory management goals
- Segmentation
- Paging
- TLB
Paging overview

- **Goal**
  - Eliminate external fragmentation
  - 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 number (ppn) using page table
  
  \[ pa = \text{page	able}[\text{va/pg}\_\text{sz}] + \text{va}\%\text{pg}\_\text{sz} \]
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 1
Page 2
Page 3
Page translation exercise

- 8-bit virtual address, 10-bit physical address, and each page is 64 bytes
  - How many virtual pages?
  - How many physical pages?
  - How many entries in page table?
  - Given page table = [2, 5, 1, 8], what's the physical address for virtual address 241?

- m-bit virtual address, n-bit physical address, k-bit page size
  - What are the answers to the above questions?
Page protection

- Implemented by associating **protection bits** with each virtual page in page table

- **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
Page protection example

Virtual Memory

Page table

Physical Memory

<table>
<thead>
<tr>
<th>Page</th>
<th>Address</th>
<th>Page Table Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>101</td>
</tr>
<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>
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

- xv6: kalloc.c

```
free_page_list

Page 0
Page 1
Page 3

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
Hierarchical page table

- Break up virtual address space into multiple page tables at different levels
Address translation with hierarchical page table

logical address

\[ p_1 \quad p_2 \quad d \]

\[ p_1 \{ \]

outer page table

\[ p_2 \{ \]

page of page table

\[ d \{ \]
x86 page translation

- 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: $32 - 12 - 12 = 10$

<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
xv6 address space implementation

- Split into kernel space and user space
  - User: 0-640KB
    - Map to end-16MB
  - Kernel: 640KB - 4GB
    - Direct (virtual = physical)
- Kernel: vm.c, setupkvm()
- User: vm.c, inituvm() and exec.c, exec()

```
0x00000000
0x000A0000
0x01000000
0x00100000
0xFE000000
0xFFFFFFFF

devices
unused
kernel heap, stack, and user pages
kernel text & data
I/O space
user text, data, stack, and heap
```
Outline

- Memory management goals
- Segmentation
- Paging
- TLB
Avoiding extra memory access

- **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 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)**
  - Pros: simple, flexible
  - Cons: code may have bug!
  - Question: what can’t a TLB miss handler do?
TLB and context switches

- What happens to TLB on context switches?

- Option 1: flush entire TLB
  - x86
    - load cr3 flushes TLB
    - INVLPGL addr: invalidates a single TLB entry

- Option 2: attach process ID to TLB entries
  - ASID: Address Space Identifier
  - MIPS, SPARC
Backup Slides
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