Virtual Memory

Memory Management Unit

The position and function of the MMU

Virtual Address Space
- Virtual Memory
  - Divided into equal-sized pages
  - A mapping is a translation between
    • A page and a frame
    • A page and null
  - Mappings defined at runtime
  - They can change
  - Address space can have holes
  - Process does not have to be contiguous in physical memory
- Physical Memory
  - Divided into equal-sized frames

Typical Address Space Layout
- Stack region is at top, and can grow down
- Heap has free space to grow up
- Text is typically read-only
- Kernel is in a reserved, protected, shared region
- 0-th page typically not used, why?

Programmer’s perspective: logically present
System’s perspective: Not mapped, data on disk

What happens if we access non-resident memory?
Page Faults

- Referencing an invalid page triggers a page fault
  - An exception handled by the OS
- Broadly, two standard page fault types
  - Illegal Address (protection error)
    - Signal or kill the process
  - Page not resident
    - Get an empty frame
    - Load page from disk
    - Update page (translation) table (enter frame #, set valid bit, etc.)
    - Restart the faulting instruction

Note: Some implementations store disk block numbers of non-resident pages in the page table (with valid bit Unset)

Shared Pages

- Private code and data
  - Each process has own copy of code and data
  - Code and data can appear anywhere in the address space
- Shared code
  - Single copy of code shared between all processes executing it
  - Code must not be self modifying
  - Code must appear at same address in all processes

Page Table Structure

- Page table is (logically) an array of frame numbers
  - Index by page number
- Each page-table entry (PTE) also has other bits
  - Caching disabled
  - Modified
  - Present/absent
PTE bits

- Present/Absent bit
  - Also called valid bit, it indicates a valid mapping for the page
- Modified bit
  - Also called dirty bit, it indicates the page may have been modified in memory
- Reference bit
  - Indicates the page has been accessed
- Protection bits
  - Read permission, Write permission, Execute permission
  - Or combinations of the above
- Caching bit
  - Use to indicate processor should bypass the cache when accessing memory
  - Example: to access device registers or memory

Address Translation

- Every (virtual) memory address issued by the CPU must be translated to physical memory
  - Every load and every store instruction
  - Every instruction fetch
- Need Translation Hardware
  - In paging system, translation involves replace page number with a frame number

Page tables (recap)

Page Tables

- Assume we have
  - 32-bit virtual address (4 Gbyte address space)
  - 4 KByte page size
  - How many page table entries do we need for one process?

virtual memory

- Programs use virtual addresses
- Virtual to physical mapping by MMU
  - First check if page present (present/absent bit)
  - If yes: address in page table form MSBs in physical address
  - If no: bring in the page from disk ➔ page fault
Page Tables

- Assume we have
  - 64-bit virtual address (humongous address space)
  - 4 KByte page size
  - How many page table entries do we need for one process?

- Problem:
  - Page table is very large
  - Access has to be fast, lookup for every memory reference
  - Where do we store the page table?
    - Registers?
    - Main memory?

Page Tables

- Page tables are implemented as data structures in main memory
- Most processes do not use the full 4GB address space
  - e.g., 0.1 – 1 MB text, 0.1 – 10 MB data, 0.1 MB stack
- We need a compact representation that does not waste space
  - But is still very fast to search
- Three basic schemes
  - Use data structures that adapt to sparsity
  - Use data structures which only represent resident pages
  - Use VM techniques for page tables (details left to extended OS)

Two-level Page Table

- 2nd-level page tables representing unmapped pages are not allocated
  - Null in the top-level page table

Two-level Translation
Alternative: Inverted Page Table

PID | VPN | offset
--- | --- | ---

Hash

Index
0 1 2 3 4 5 6 ...

IPT: entry for each physical frame

Alternative: Inverted Page Table

PID | VPN | offset
--- | --- | ---
0 | 0x5 | 0x123

Hash

Index
0 1 2

Hash Anchor Table (HAT)

PID | VPN | ctrl | next
--- | --- | --- | ---
0 | 0x1A | 0x40C | 0x5 0x0
1 | 0x123 | 0x40C | 0x40C 0x40D
2 | 0x40C | 0x123 | ppn offset

Inverted Page Table (IPT)

- "Inverted page table" is an array of page numbers sorted (indexed) by frame number (it's a frame table).
- Algorithm
  - Compute hash of page number
  - Extract index from hash table
  - Use this to index into inverted page table
  - Match the PID and page number in the IPT entry
  - If match, use the index value as frame # for translation
  - If no match, get next candidate IPT entry from chain field
  - If NULL chain entry ⇒ page fault

Properties of IPTs

- IPT grows with size of RAM, NOT virtual address space
- Frame table is needed anyway (for page replacement, more later)
- Need a separate data structure for non-resident pages
- Saves a vast amount of space (especially on 64-bit systems)
- Used in some IBM and HP workstations

Given \( n \) processes

- how many page tables will the system have for
  - 'normal' page tables
  - inverted page tables?

Another look at sharing...
VM Implementation Issue

- **Problem:**
  - Each virtual memory reference can cause two physical memory accesses
    - One to fetch the page table entry
    - One to fetch/store the data
  - Intolerable performance impact!!
- **Solution:**
  - High-speed cache for page table entries (PTEs)
    - Called a translation look-aside buffer (TLB)
    - Contains recently used page table entries
    - Associative, high-speed memory, similar to cache memory
    - May be under OS control (unlike memory cache)

Translation Lookaside Buffer

- Given a virtual address, processor examines the TLB
- If matching PTE found (TLB hit), the address is translated
- Otherwise (TLB miss), the page number is used to index the process’s page table
  - If PT contains a valid entry, reload TLB and restart
  - Otherwise, (page fault) check if page is on disk
    - If on disk, swap it in
    - Otherwise, allocate a new page or raise an exception

TLB properties

- Page table is (logically) an array of frame numbers
- TLB holds a (recently used) subset of PT entries
  - Each TLB entry must be identified (tagged) with the page # it translates
  - Access is by associative lookup:
    - All TLB entries’ tags are concurrently compared to the page #
    - TLB is associative (or content-addressable) memory

TLB operation

- On-CPU hardware device!!
- Data structure in main memory
- TLB properties
  - Page table is (logically) an array of frame numbers
  - TLB holds a (recently used) subset of PT entries
  
Translation Lookaside Buffer

- Given a virtual address, processor examines the TLB
- If matching PTE found (TLB hit), the address is translated
- Otherwise (TLB miss), the page number is used to index the process’s page table
  - If PT contains a valid entry, reload TLB and restart
  - Otherwise, (page fault) check if page is on disk
    - If on disk, swap it in
    - Otherwise, allocate a new page or raise an exception

TLB properties

- Page table is (logically) an array of frame numbers
- TLB holds a (recently used) subset of PT entries
  - Each TLB entry must be identified (tagged) with the page # it translates
  - Access is by associative lookup:
    - All TLB entries’ tags are concurrently compared to the page #
    - TLB is associative (or content-addressable) memory

Translation Lookaside Buffer

- Given a virtual address, processor examines the TLB
- If matching PTE found (TLB hit), the address is translated
- Otherwise (TLB miss), the page number is used to index the process’s page table
  - If PT contains a valid entry, reload TLB and restart
  - Otherwise, (page fault) check if page is on disk
    - If on disk, swap it in
    - Otherwise, allocate a new page or raise an exception

Translation Lookaside Buffer

- Given a virtual address, processor examines the TLB
- If matching PTE found (TLB hit), the address is translated
- Otherwise (TLB miss), the page number is used to index the process’s page table
  - If PT contains a valid entry, reload TLB and restart
  - Otherwise, (page fault) check if page is on disk
    - If on disk, swap it in
    - Otherwise, allocate a new page or raise an exception
TLB properties

• TLB may or may not be under direct OS control
  – Hardware-loaded TLB
    • On miss, hardware performs PT lookup and reloads TLB
    • Example: x86, ARM
  – Software-loaded TLB
    • On miss, hardware generates a TLB miss exception, and
      exception handler reloads TLB
    • Example: MIPS, Itanium (optionally)

• TLB size: typically 64-128 entries
• Can have separate TLBs for instruction fetch
  and data access
• TLBs can also be used with inverted page tables
  (and others)

TLB and context switching

• TLB is a shared piece of hardware
• Normal page tables are per-process (address space)
• TLB entries are process-specific
  – On context switch need to flush the TLB (invalidate all
    entries)
  – or tag entries with address-space ID (ASID)
  • called a tagged TLB
  • used (in some form) on all modern architectures
  • TLB entry: ASID, page #, frame #, valid and write-protect bits

TLB effect

• Without TLB
  – Average number of physical memory references per virtual reference
    = 2
• With TLB (assume 99% hit ratio)
  – Average number of physical memory references per virtual reference
    = .99 * 1 + 0.01 * 2
    = 1.01

Recap - Simplified Components of VM System

R3000 TLB

<table>
<thead>
<tr>
<th>31</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEN</td>
<td>ASID</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

EntryHi Register (TLB key fields)

<table>
<thead>
<tr>
<th>31</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEN</td>
<td>N</td>
<td>D</td>
<td>V</td>
<td>G</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

EntryLo Register (TLB data fields)

- **N** = Not cacheable
- **D** = Dirty = Write protect
- **G** = Global (ignore ASID in lookup)
- **V** = valid bit
- 64 TLB entries
- Accessed via software through Co-processor 0 registers
- EntryHi and EntryLo

MIPS R3000 TLB

<table>
<thead>
<tr>
<th>31</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEN</td>
<td>ASID</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

EntryHi Register (TLB key fields)

<table>
<thead>
<tr>
<th>31</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>VEN</td>
<td>N</td>
<td>D</td>
<td>V</td>
<td>G</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

EntryLo Register (TLB data fields)

- **N** = Not cacheable
- **D** = Dirty = Write protect
- **G** = Global (ignore ASID in lookup)

R3000 Address Space Layout

- **kuseg**: 2 gigabytes
  - TLB translated (mapped)
  - Cacheable (depending on ‘N’ bit)
  - user-mode and kernel mode accessible
  - Page size is 4K
- **kseg0**: 4 gigabytes
- **kseg1**: 4 gigabytes
- **kseg2**: 4 gigabytes

0xFFFFFFFF

0xC0000000

0xA0000000

0x80000000

0x00000000
**R3000 Address Space Layout**

- **kseg0:**
  - 512 megabytes
  - Fixed translation window to physical memory
  - Cacheable
  - Only kernel-mode accessible
  - Usually where the kernel code is placed
  - TLB not used
  - 0x00000000 S 0x1fffffff virtual = 0x00000000 S 0x1fffffff physical

- **kseg1:**
  - 512 megabytes
  - Fixed translation window to physical memory
  - NOT cacheable
  - Only kernel-mode accessible
  - Where devices are accessed (and boot ROM)
  - 0xa0000000 S 0xbfffffff virtual = 0x00000000 S 0x1fffffff physical

- **kseg2:**
  - 1024 megabytes
  - TLB translated (mapped)
  - Cacheable
  - Depending on the 'N'-bit
  - Only kernel-mode accessible
  - Can be used to store the virtual linear array page table

---

**Diagrams**

- Diagram illustrating the address space layout with kseg0, kseg1, and kseg2 segments.
- Physical memory is mapped accordingly for each kseg segment.