Virtual Memory

Learning Outcomes

• An understanding of page-based virtual memory in depth.
  – Including the R3000’s support for virtual memory.

Memory Management Unit (or TLB)

• 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

Page-based VM

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

A process may be only partially resident
  – Allows OS to store individual pages on disk
  – Saves memory for infrequently used data & code

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

**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
PTE Attributes (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

Virtual Memory Summary

- 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
  - 32-bit virtual address (4 Gbyte 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

Example Translations

Summarising Two-level Page Tables

- Translating a 32-bit virtual address into a 32-bit physical
- Recall:
  - the level 1 page table node has $2^{10}$ entries
  - $2^{10} \times 4 = 4$ KB node
  - the level 2 page table node have $2^{12}$ entries
  - $2^{12} \times 4 = 4$ KB node
Index bits determine node sizes

- Translating a 32-bit virtual address into a 32-bit physical
- Changing the indexing:
  - the level 1 page table node has $2^3$ entries
  - the level 2 page table node has $2^6$ entries
  - $2^9 + 4 = 1$ KB node
  - $2^{12} + 4 = 16$ KB node

Supporting 64-bit Virtual to Physical Translation

- Translating a 64-bit virtual address into a 64-bit physical
- Support 64-bits:
  - the level 1 page table node has $2^{12}$ entries
  - the level 2 page table node has $2^{16}$ entries
  - $2^{20} + 8 = 512$ MB node

Multi-level Page Tables

- Translating a 64-bit virtual address into a 64-bit physical (Intel/AMD pre-Ice Lake)
  - Only support 48-bit addresses
  - the level 1 page table node has $2^3$ entries
  - the level 2 page table node has $2^6$ entries
  - $2^9 + 4 = 4$ KB node
  - $2^{12} + 4 = 4$ KB node
  - $2^{15} + 4 = 4$ KB node

Intel 4-Level Page Tables

Alternative: Inverted Page Table

<table>
<thead>
<tr>
<th>PID</th>
<th>VPN</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PID</td>
<td>VPN</td>
<td>ctrl next</td>
</tr>
<tr>
<td>0</td>
<td>0x5</td>
<td>0x123</td>
</tr>
</tbody>
</table>

Hash Anchor Table (HAT)

Alternative: Inverted Page Table

<table>
<thead>
<tr>
<th>PID</th>
<th>VPN</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PID</td>
<td>VPN</td>
<td>ctrl next</td>
</tr>
<tr>
<td>0</td>
<td>0x5</td>
<td>0x123</td>
</tr>
</tbody>
</table>

Hash Anchor Table (HAT)
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...

Two (or more) processes running the same program and sharing the text section

Improving the IPT: Hashed Page Table

- Retain fast lookup of IPT
  - A single memory reference in best case
- Retain page table sized based on physical memory size (not virtual)
  - Enable efficient frame sharing
  - Support more than one mapping for same frame
**Hashed Page Table**

<table>
<thead>
<tr>
<th>PID</th>
<th>VPN offset</th>
<th>HPT: Frame number stored in table</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Sharing Example**

<table>
<thead>
<tr>
<th>PID</th>
<th>VPN offset</th>
<th>ppp offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0x42 0x123</td>
</tr>
</tbody>
</table>

**Sizing the Hashed Page Table**

- HPT sized based on physical memory size
- With sharing
  - Each frame can have more than one PTE
  - More sharing increases number of slots used
    - Increases collision likelihood
- However, we can tune HPT size based on:
  - Physical memory size
  - Expected sharing
  - Hash collision avoidance.
- HPT a power of 2 multiple of number of physical memory frame

**VM Implementation Issue**

- Performance?
  - 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)

**TLB operation**

- On-CPU hardware device!!
- Data structure in main memory
- Page table
- Page fault
- Physical address
- Virtual address
- Translation look-aside buffer (TLB)
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 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)
    - high context-switching overhead (Intel x86)
    - 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
Recap - Simplified Components of VM System

- CPU
- Frame Pool
- Physical Memory

Virtual Address Spaces
- Inverted Page Table
- Physical Memory
- Frame Table

MIPS R3000 TLB

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

<table>
<thead>
<tr>
<th>CPU</th>
<th>ASID</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>EntryLo Register (TLB data fields)</th>
</tr>
</thead>
<tbody>
<tr>
<td>N = valid bit</td>
</tr>
<tr>
<td>64 TLB entries</td>
</tr>
<tr>
<td>Accessed via software through Cooperator 0 registers</td>
</tr>
<tr>
<td>– EntryHi and EntryLo</td>
</tr>
</tbody>
</table>

R3000 Address Space Layout

- kseg0:
  - 512 megabytes
  - Fixed translation window to physical memory
  - 0x80000000 - 0xffffffff virtual = 0x00000000 - 0xffffffff physical
  - TLB not used
  - Cacheable
  - Only kernel-mode accessible
  - Usually where the kernel code and data is placed

- kseg:
  - 2 gigabytes
  - TLB translated (mapped)
  - Cacheable (depending on 'N' bit)
  - user-mode and kernel mode accessible
  - Page size is 4K

R3000 Address Space Layout

- Switching processes switches the translation (page table) for kseg

- Proc 1
  - kseg
  - kuseg

- Proc 2
  - kseg
  - kuseg

- Proc 3
  - kseg
  - kuseg
R3000 Address Space Layout

- kseg1:
  - 512 megabytes
  - Fixed translation window to physical memory
  - 0xa0000000 - 0xbfffffff virtual = 0x00000000 - 0x1fffffff physical
  - TLB not used
    - NOT cacheable
    - Only kernel-modes accessible
    - Binary ROMs are accessible (and
      boot ROM)

- kseg0
- kuseg
- kseg1
- kseg2

Physical Memory