Virtual Memory

Expanding Memory

Multiple Concurrent Processes
Virtual Memory

- In virtual memory addresses used by a program are not those where data is actually stored!
- A translation takes virtual address used by the processor to the actual physical address in memory
- Allows program to address more memory than physically exists
- Allows separate processes that share the processor to use the same virtual addresses
  - Only one process active at a time
  - Actual memory for different processes are stored in different physical locations
- Information may be stored in main memory or on disk
Virtual Memory Pages

• Unit of storage is a page
  – Typically 4-16 KB

• Low order bits determine location within page
  – e.g. 4 KB page needs 12 bits for offset within page

• Remainder of virtual address is the virtual page address
  – 20 bits for 32 bit addressing and 4 KB pages

• How do we find the actual information from a virtual address???
Page Table

- Each virtual address (high bits) maps to a physical address
- The mappings are kept in a page table indexed by the high bits of the virtual address
- 20 high bits (12 bits for offset) means $2^{20}$ table entries (for 4KB pages)
- Each entry is the physical location of the page
- Page table is stored in memory
- Page table register stores the start location of the page table in memory
In this example, physical memory is smaller than addressable virtual memory.
Page table register

Virtual address

31 30 29 28 27 ........................................ 15 14 13 12 11 10 9 8 .................................. 3 2 1 0

Virtual page number  Page offset

Valid

Physical page number

Page table

If 0 then page is not present in memory

29 28 27 ........................................ 15 14 13 12 11 10 9 8 .................................. 3 2 1 0

Physical page number  Page offset

Physical address
Page table

- Stored in memory (maybe virtual memory)
  - Page table for OS in main memory
- Indexed by high bits of virtual address
  - Added to page table register
- Contains physical address (high bits) and condition bits
  - Valid bit indicates in physical memory
  - Otherwise stored on hard disk
Find a physical address

1. Add high bits of virtual address to page table register value
2. Find that location in memory
3. Check valid bit
   • If set, then read high bits of physical memory
   • If not set, then page fault!
4. If found in physical memory, access
5. If page fault, throw exception for OS to handle
   • For hit, two memory accesses needed!!
   • Can we do better??
Translation Look-aside Buffer

• TLB is a cache for page table entries

• High order (index) virtual address bits access TLB
  – If fully associative, all bits in tag
  – If partially associative, bits split between index (low) and tag (high)
  – Holds physical address

• If virtual address is in TLB
  – Get physical address – single memory access for info
  – If not, go to page table
Virtual page number | Valid Dirty Ref | Tag | Physical page address
--- | --- | --- | ---
1 0 1
1 1 1
1 1 1
1 1 1
0 0 0
1 0 1
1 0 1

Physical memory

Page table

Valid Dirty Ref | Physical page or disk address
--- | ---
1 0 1
1 0 0
1 0 0
1 0 1
0 0 0
1 0 1
1 0 1
1 0 1
0 0 0
1 1 1
1 1 1
1 1 1
0 0 0
1 1 1

Disk storage
TLB Translation Look-aside buffer

- 16 – 512 entries
- Hit – 0.5 to 1.0 clock cycles
  - Hit gives physical address for memory access
  - Process stalled for memory access
- Miss penalty – 10 - 100 clock cycles
  - For page table look-up
  - Still needs memory access
  - Causes exception – OS handles miss
  - May mean page fault!
TLB organization

• Before Cache
  – Cache uses physical addresses
• After Cache
  – Cache uses virtual addresses
• Mixed
  – Virtual index but physical tag

• Example: Intrinsity TLB before Cache
Intrinsity FastMATH TLB
Sequence for TLB and Cache  
Assume Physical Addressed Cache

1. Memory address goes to TLB
2. If TLB hit, take physical address to Cache
3. If Cache hit, return information
   • If write, mark cache entry dirty
4. If Cache miss, start memory access
   • Use physical address from TLB
   • With TLB hit, must be in memory
5. If TLB miss
   • Exception – control to OS (using TLB ex-address)
# TLB miss exception registers for MIPS

<table>
<thead>
<tr>
<th>Register</th>
<th>CP0 register number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EPC</td>
<td>14</td>
<td>Where to restart after exception</td>
</tr>
<tr>
<td>Cause</td>
<td>13</td>
<td>Cause of exception</td>
</tr>
<tr>
<td>BadVAddr</td>
<td>8</td>
<td>Address that caused exception</td>
</tr>
<tr>
<td>Index</td>
<td>0</td>
<td>Location in TLB to be read or written</td>
</tr>
<tr>
<td>Random</td>
<td>1</td>
<td>Pseudorandom location in TLB</td>
</tr>
<tr>
<td>EntryLo</td>
<td>2</td>
<td>Physical page address and flags</td>
</tr>
<tr>
<td>EntryHi</td>
<td>10</td>
<td>Virtual page address</td>
</tr>
<tr>
<td>Context</td>
<td>4</td>
<td>Page table address and page number</td>
</tr>
</tbody>
</table>
TLB miss exception code for MIPS

mfc0 $k1, Context  #copy address of PTE into $k1
lw  $k1, 0($k1)   #put PTE into $k1
mtco $k1, EntryLo #put PTE into register EntryLo
        tlbwr     #put PTE (from EntryLo) into
              # TLB entry at a random
              # location
        eret      # return from TLB miss
              # exception (restore PC
              # from EPC)

Takes about 12 clock cycles
TLB Miss exception -- OS

1. Go to page table in memory, load line into TLB
2. Restart process to continue with TLB access
3. If valid, TLB hit, continue to cache access
4. If TLB entry is *not valid*, this is a *page fault* exception
Page Fault Exception

• Starts at regular exception address

1. Save complete state of process
   • See table 5.28 in book

2. Look up PTE and find disk location

3. Choose physical page to replace
   • If dirty, must be written to disk

4. Start read to bring referenced page into physical memory
   • Millions of processor clock cycles, so

5. Start an alternate process
Page Fault Exception Completion

- If writing dirty page, interrupt at completion
  - This is an exception, OS takes over, starts read of the page to be loaded, restarts alternate process
- When page is loaded, interrupt
  - This is an exception, OS takes over,
    - Restores state of original process
    - Original process restarts with instruction that caused exception.
Virtual memory and Protection

- Multiple processes and OS have same virtual addresses
- Must keep user processes from messing with others memory, especially OS
- Key is two-fold
  - Processor mode: user or supervisor (kernel, executive)
  - Only OS in supervisor mode can write critical elements
    - user/super mode bit, page table pointer, TLB
- Transfer from mode to mode
  - e.g. sys calls and exceptions invoke supervisor mode
  - OS return to user process invokes user mode