Lecture 7: Virtual Memory
Review: Replacement Policies

- least-recently-used (LRU)
  - evict the line that has been least recently referenced
    - need to keep track of order that lines in a set have been referenced
    - overhead to do this gets worse as associativity increases

- random
  - just pick one at random
    - easy to implement
    - slightly lower hit rates than LRU on average

- not-most-recently-used
  - track which line in a set was referenced most recently, pick randomly from the others
    - compromise in both hit rate and implementation difficulty

- virtual memories
  - use similar policies but spend more effort to improve hit rate
Review: Pseudo LRU Algorithm

AB/CD bit (L0)

A/B bit (L1) C/D bit (L2)

Way A Way B Way C Way D

L2L1L0 = 000, there is a hit in Way B, what is the new updated L2L1L0?

LRU update algorithm

<table>
<thead>
<tr>
<th>Way hit</th>
<th>L2</th>
<th>L1</th>
<th>L0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Way A</td>
<td>---</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Way B</td>
<td>---</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Way C</td>
<td>0</td>
<td>---</td>
<td>1</td>
</tr>
<tr>
<td>Way D</td>
<td>1</td>
<td>---</td>
<td>1</td>
</tr>
</tbody>
</table>

Replacement Decision

<table>
<thead>
<tr>
<th>L2</th>
<th>L1</th>
<th>L0</th>
<th>Way to replace</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>1</td>
<td>1</td>
<td>Way A</td>
</tr>
<tr>
<td>X</td>
<td>0</td>
<td>1</td>
<td>Way B</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>0</td>
<td>Way C</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>0</td>
<td>Way D</td>
</tr>
</tbody>
</table>

less hardware than LRU & faster than LRU
Virtual Memory

- virtual memory – separation of logical memory from physical memory.
  - only a part of the program needs to be in memory for execution
    - logical address space can be much larger than physical address space
  - allows address spaces to be shared by several processes (or threads)
  - allows for more efficient process creation

- virtual memory can be implemented via:
  - demand paging
  - demand segmentation

main memory is like a cache to the hard disc!
Virtual Address

- concept of a virtual (or logical) address space that is bound to a separate physical address space is central to memory management
  - virtual address – generated by CPU
  - physical address – seen by memory

- virtual and physical addresses
  - are the same in compile-time
  - differ in execution-time address-binding schemes
Advantages of Virtual Memory

- **translation**
  - program can be given consistent view of memory, even though physical memory is scrambled
  - only the most important part of program ("working set") must be in physical memory
  - contiguous structures (like stacks) use only as much physical memory as necessary yet grow later

- **protection**
  - different threads (or processes) protected from each other
  - different pages can be given special behavior
    - (read only, invisible to user programs, etc.).
  - kernel data protected from user programs
  - very important for protection from malicious programs
    - far more "viruses" under Microsoft Windows

- **sharing**
  - map the same physical page to multiple users ("shared memory")
Use of Virtual Memory

- Stack
- Shared Libraries
- Heap
- Static Data
- Code

Process A

Process B

Shared page
Virtual vs. Physical Address Space

Virtual Memory

| Virtual Address | 0   | 4k  | 8k  | 12k | 4G
|-----------------|-----|-----|-----|-----|-----
|                 | A   | B   | C   | D   | .   |

Physical Address

<table>
<thead>
<tr>
<th>Physical Address</th>
<th>0</th>
<th>4k</th>
<th>8k</th>
<th>12k</th>
<th>16k</th>
<th>20k</th>
<th>24k</th>
<th>28k</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>C</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>4k</td>
<td>.</td>
<td>C</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>8k</td>
<td>.</td>
<td>.</td>
<td>C</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>12k</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>D</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>16k</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>A</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>20k</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>B</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>24k</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>A</td>
<td>.</td>
</tr>
<tr>
<td>28k</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>B</td>
</tr>
</tbody>
</table>

Main Memory

<table>
<thead>
<tr>
<th>Main Memory</th>
<th>0</th>
<th>4k</th>
<th>8k</th>
<th>12k</th>
<th>16k</th>
<th>20k</th>
<th>24k</th>
<th>28k</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>C</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>C</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>.</td>
<td>C</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>D</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>A</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>B</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>A</td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>B</td>
</tr>
</tbody>
</table>

Disk

<table>
<thead>
<tr>
<th>Disk</th>
<th>D</th>
</tr>
</thead>
</table>

ECE 411 COMPUTER ORGANIZATION AND DESIGN
Paging

- frame
  - divide physical memory into fixed-size blocks (e.g., 4KB)

- pages
  - divide logical memory into blocks of same size (4KB)
  - to run a program of size \( n \) pages, need to find \( n \) free frames and load program
  - set up a page table to map page addresses to frame addresses
    - operating system sets up the page table
Page Table and Address Translation

virtual page number (VPN)  page offset

page table

physical page # (PPN)

II

main memory

physical address
Page Table Structure Examples

- one-to-one mapping, space?
  - large pages
    - Internal fragmentation (similar to having large line sizes in caches)
  - small pages
    - page table size issues

- multi-level paging

- inverted page table

Example:
64 bit address space, 4 KB pages (12 bits), 512 MB (29 bits) RAM

Number of pages = \( \frac{2^{64}}{2^{12}} = 2^{52} \)
(The page table has as many entries)

Each entry is \(~4\) bytes, the size of the Page table is \(2^{54}\) Bytes = 16 Petabytes!

Can’t fit the page table in the 512 MB RAM!
### Handling a Page Fault (2)

1. if there is ever a reference to a page not in memory, first reference will cause page fault.

2. page fault is handled by the appropriate OS service routines.

3. locate needed page on disk (in file or in backing store).

4. swap page into free frame (assume available).

5. reset page tables – valid-invalid bit = v.

6. restart the instruction that caused the page fault.
Fast Address Translation

- how often address translation occurs?
- where the page table is kept?
- keep translation in the hardware
- use Translation Lookaside Buffer (TLB)
  - instruction-TLB & data-TLB
  - essentially a cache (tag array = VPN, data array=PPN)
  - small (32 to 256 entries are typical)
  - typically fully associative (implemented as a content addressable memory, CAM) or highly associative to minimize conflicts
What Happens on a Context Switch?

- each process has its own address space
- so, each process has its own page table
- so, page-table entries are only relevant for a particular process
- thus, the TLB must be flushed on a context switch
  - this is why context switches are so expensive
Alternative to Flushing

address space IDs

- we can avoid flushing the TLB if entries are associated w/ an address space

<table>
<thead>
<tr>
<th>ASID</th>
<th>V</th>
<th>R</th>
<th>M</th>
<th>prot</th>
<th>page frame number</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>20</td>
</tr>
</tbody>
</table>

- when would this work well?
- when would this not work well?
**Example: Alpha 21264 data TLB**

<table>
<thead>
<tr>
<th>VPN &lt;35&gt;</th>
<th>offset &lt;13&gt;</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>&lt;8&gt;</th>
<th>&lt;4&gt;&lt;1&gt;</th>
<th>&lt;35&gt;</th>
<th>&lt;31&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASN</td>
<td>Prot</td>
<td>Tag</td>
<td>PPN</td>
</tr>
</tbody>
</table>

Address Space Number

128:1 mux

44-bit physical address
Hard versus Soft Page Faults

- **hard page faults**
  - those page faults that require issuing a read from secondary storage.

- **soft page faults**
  - those page faults where the page is already in main memory but the TLB and/or the PTE has marked the page as invalid.
  - soft faults are used when hardware support is not available to handle TLB misses
Caches and Virtual Memory

- do we send virtual or physical addresses to the cache?
  - virtual \(\rightarrow\) faster, because don’t have to translate
    - issue: different programs can reference the same virtual address, either creates security/correctness hole or requires flushing the cache every time you context switch
  - physical \(\rightarrow\) slower, but no security issue

- actually, there are four possibilities
  - VIVT: Virtually-indexed Virtually-tagged Cache
  - PIPT: Physically-indexed Physically-tagged Cache
  - VIPT: Virtually-indexed Physically-tagged Cache
  - PIVT: Physically-indexed Virtually-tagged Cache
Virtually-Indexed Virtually-Tagged

- fast cache access
- only require address translation when going to memory (miss)
- issues?
**VIVT Cache Issues - Aliasing**

- **homonym**
  - ✓ same VA maps to different PAs
  - ✓ occurs when there is a context switch
  - ✓ solutions
    - o include process ID (PID) in cache or
    - o flush cache upon context switches

- **synonym (also a problem in VIPT)**
  - ✓ different VAs map to the same PA
  - ✓ occurs when data is shared by multiple processes
  - ✓ duplicated cache line in VIPT cache and VIVT$ w/ PID
  - ✓ data is inconsistent due to duplicated locations
  - ✓ solution
    - o can write-through solve the problem?
    - o flush cache upon context switch
    - o if (index+offset) < page offset, can the problem be solved? (discussed later in VIPT)
Physically-Indexed Physically-Tagged

- slower, always translate address before accessing memory
- simpler for data coherence

Diagram:
- Processor core
  - VA (Virtual Address)
  - TLB
  - PA (Physical Address)
  - PIPT cache
  - Main memory
- cache line return
- hit
- miss
Virtually-Indexed Physically-Tagged

- gain benefit of a VIVT and PIPT
- parallel access to TLB and VIPT cache
- no homonym
  ✓ how about synonym?

Diagram:
- Processor core → Virtual Address (VA) → TLB → Physical Address (PA) → Main Memory
  - Hit: VIPT cache line return
  - Miss: Main memory → Cache line return
Deal w/ Synonym in VIPT Cache

VPN A

process a

VPN B

process b

index

point to the same location within a page

• VPN A != VPN B
• how to eliminate duplication?
• make cache index a == index b?

tag array
data array
Synonym in VIPT Cache

- If two VPNs do not differ in a then there is no synonym problem, since they will be indexed to the same set of a VIPT cache.
- Imply # of sets cannot be too big.
- Max number of sets = page size / cache line size.
  - Ex: 4KB page, 32B line, max set = 128.
- A complicated solution in MIPS R10000.

<table>
<thead>
<tr>
<th>VPN</th>
<th>Page Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache tag</td>
<td>Set index</td>
</tr>
</tbody>
</table>

Ex: 4KB page, 32B line, max set = 128
R10000’s Solution to Synonym

- 32KB 2-Way virtually-indexed L1
  
  ![VPN 12 bit Table]
  
<table>
<thead>
<tr>
<th>VPN</th>
<th>12 bit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10 bit</td>
</tr>
</tbody>
</table>

- direct-mapped physical L2
  - L2 is *inclusive* of L1
  - VPN[1:0] is *appended* to the “tag” of L2

- Given two virtual addresses *VA1* and *VA2* that differs in *VPN[1:0]* and both map to the same physical address *PA*
  - Suppose *VA1* is accessed first so blocks are allocated in L1&L2
  - What happens when *VA2* is referenced?
    1. *VA2* indexes to a different block in L1 and misses
    2. *VA2* translates to *PA* and goes to the same block as *VA1* in L2
    3. Tag comparison fails (since *VA1*[1:0]≠*VA2*[1:0])
    4. Treated just like as a L2 conflict miss ⇒ *VA1’s* entry in L1 is ejected (or dirty-written back if needed) due to inclusion policy
Deal with Synonym in MIPS R10000

L1 VIPT cache

TLB

L2 PIPT cache

ECE 411 COMPUTER ORGANIZATION AND DESIGN
Deal w/ Synonym in MIPS R10000

only one copy is present in L1

L1 VIPT cache

L2 PIPT cache

TLB

data return

ECE 411 COMPUTER ORGANIZATION AND DESIGN
Announcement

- today’s lecture: virtual memory
  - Ch. 5.6 – 5.7 (HP1)

- next lecture: more virtual memory
  - Ch. 5.6 – 5.7 (HP1)

- MP assignment
  - MP2 check-point 1 due on 9/24 5pm