Improving Miss Penalty

- When caches first became popular, Miss Penalty ~ 10 processor clock cycles
- Today 2400 MHz Processor (0.4 ns per clock cycle) and 80 ns to go to DRAM ⇒ 200 processor clock cycles!

Solution: another cache between memory and the processor cache: Second Level (L2) Cache

Analyzing Multi-level cache hierarchy

```
Avg Mem Access Time = L1 Hit Time + L1 Miss Rate * L1 Miss Penalty

L1 Miss Penalty = AMAT_L2 = L2 Hit Time + L2 Miss Rate * L2 Miss Penalty

Avg Mem Access Time = L1 Hit Time + L1 Miss Rate * (L2 Hit Time + L2 Miss Rate * L2 Miss Penalty)
```

Example: with L2 cache

- Assume
  - L1 Hit Time = 1 cycle
  - L1 Miss rate = 5%
  - L2 Hit Time = 5 cycles
  - L2 Miss rate = 15% (% L1 misses that miss)
  - L2 Miss Penalty = 200 cycles

- L1 miss penalty = 5 + 0.15 * 200 = 35
- Avg mem access time = 1 + 0.05 x 35 = 2.75 cycles

Typical Scale

- L1
  - size: tens of KB
  - hit time: complete in one clock cycle
  - miss rates: 1-5%

- L2:
  - size: hundreds of KB
  - hit time: few clock cycles
  - miss rates: 10-20%

- L2 miss rate is fraction of L1 misses that also miss in L2
  - why so high?
Example: without L2 cache

- Assume
  - L1 Hit Time = 1 cycle
  - L1 Miss rate = 5%
  - L1 Miss Penalty = 200 cycles
  - Avg mem access time = 1 + 0.05 \times 200 = 11 cycles

- 4x faster with L2 cache! (2.75 vs. 11)

Cache Summary

- Cache design choices:
  - size of cache: speed v. capacity
  - direct-mapped v. associative
  - for N-way set assoc: choice of N
  - block replacement policy
  - 2nd level cache?
  - Write through v. write back?

- Use performance model to pick between choices, depending on programs, technology, budget, ...

Generalized Caching

- We’ve discussed memory caching in detail. Caching in general shows up over and over in computer systems
  - Filesystem cache
  - Web page cache
  - Game Theory databases / tablebases
  - Software memoization
  - Others?

- Big idea: if something is expensive but we want to do it repeatedly, do it once and cache the result.

Another View of the Memory Hierarchy

Thus far

Next: Virtual Memory

Upper Level

Faster

Reg

Instr. Operands

L1 Cache

Blocks

L2 Cache

Blocks

Memory

Pages

Disk

Files

Lower Level

Larger

Tape

Memory Hierarchy Requirements

- What else might we want from our memory subsystem? ...

  - Share memory between multiple processes but still provide protection – don’t let one program read/write memory from another
    - Emacs on star

  - Address space – give each process the illusion that it has its own private memory
    - Implicit in our model of a linker

- Called Virtual Memory
Virtual Memory Big Ideas

• Each address that a program uses (pc, $sp, $gp, .data, etc) is fake (even after linking)!

• Processor inserts new step:
  • Every time we reference an address (in IF or MEM) …
  • Translate fake address to real one.

  virtual physical

VM Ramifications

• Immediate consequences:
  • Each program can operate in isolation!
  • OS can decide where and when each goes in memory!
  • HW/OS can grant different rights to different processes on same chunk of physical mem!

• Big question:
  How do we manage the VA → PA mappings?

(Weak) Analogy

• Book title like virtual address
• Library of Congress call number like physical address
• Card catalogue like page table, mapping from book title to call number
• On card for book, in local library vs. in another branch like valid bit indicating in main memory vs. on disk
• On card, available for 2-hour in library use (vs. 2-week checkout) like access rights

VM

• Ok, now how do we implement it?

• Simple solution:
  • Linker assumes start addr at 0x0.
  • Each process has a $base and $bound:
    - $base: start of physical address space
    - $bound: size of physical address space

• Algorithms:
  - VA → PA Mapping: \( PA = VA + \$base \)
  - Bounds check: \( VA < \$bound \)

Simple Example: Base and Bound Reg

- $base+
- $bound
- \( \infty \)
- User C
- User B
- User A
- OS
-有足够的空间为User D，但不连续的（“碎片化问题”）
- 同样缺点为freelist malloc!
- Also: what if process size > mem

VM Observations

• Working set of process is small, but distributed all over address space ➔
  • Arbitary mapping function,
    - keep working set in memory
    - rest on disk or unallocated.

• Fragmentation comes from variable-sized physical address spaces
  • Allocate physical memory in fixed-sized chunks (1 mapping per chunk)
  • FA placement of chunks
    - i.e. any V chunk of any process can map to any P chunk of memory.
Mapping Virtual Memory to Physical Memory

• Divide into equal sized chunks (about 4 KB - 8 KB)
• Any chunk of Virtual Memory assigned to any chunk of Physical Memory ("page")

Virtual Memory
Code
Static
Heap
Physical Memory

Virtual Memory Mapping Function

Page Number | Offset
--- | ---

• Use table lookup ("Page Table") for mappings: V Page number is index
• Mapping Function
  • Physical Offset = Virtual Offset
  • Physical Page Number = PageTable[Virtual Page Number]

FYI: P.P.N. also called "Page Frame" or "Frame #".

Page Table

• A page table: mapping function
  • There are several different ways, all up to the operating system, to keep this data around.
  • Each process running in the operating system has its own page table
    - Historically, OS changes page tables by changing contents of Page Table Base Register
    - Not anymore! We’ll explain soon.

Requirements revisited

• Remember the motivation for VM:
  • Sharing memory with protection
    • Different physical pages can be allocated to different processes (sharing)
    • A process can only touch pages in its own page table (protection)
  • Separate address spaces
    • Since programs work only with virtual addresses, different programs can have different data/code at the same address!
**Page Table Entry (PTE) Format**
- Contains either Physical Page Number or indication not in Main Memory
- OS maps to disk if Not Valid (V = 0)
- If valid, also check if have permission to use page: Access Rights (A.R.) may be Read Only, Read/Write, Executable

**Comparing the 2 levels of hierarchy**

- **Cache Version**
  - Block or Line: Page
  - Miss: Page Fault
  - Block Size: 32-64B
  - Placement: Direct Mapped, N-way Set Associative
  - Replacement: Least Recently Used (LRU)
  - Write Thru or Back

- **Virtual Memory vers.**
  - Page
  - Page Size: 4K-8KB
  - Fully Associative
  - Least Recently Used (LRU)
  - Write Back

**Notes on Page Table**
- OS must reserve "Swap Space" on disk for each process
- To grow a process, ask Operating System
  - If unused pages, OS uses them first
  - If not, OS swaps some old pages to disk
    - (Least Recently Used to pick pages to swap)
- Will add details, but Page Table is essence of Virtual Memory

**Peer Instruction**
- A. Locality is important yet different for cache and virtual memory (VM): temporal locality for caches but spatial locality for VM
- B. Cache management is done by hardware (HW) and page table management is done by software
- C. VM helps both with security and cost

**And in conclusion...**
- Manage memory to disk? Treat as cache
  - Included protection as bonus, now critical
  - Use Page Table of mappings for each user vs. tag/data in cache
- Virtual Memory allows protected sharing of memory between processes
- Spatial Locality means Working Set of Pages is all that must be in memory for process to run fairly well