Chapter 5: Large and Fast: Exploiting Memory Hierarchy

5.7: Virtual Memory

CSCE 212 Introduction to Computer Architecture, Spring 2019
https://passlab.github.io/CSCE212/

By Dr. Song Wang
Department of Computer Science and Engineering
Yonghong Yan
yanyh@cse.sc.edu
http://cse.sc.edu/~yanyh
Chapter 5: Large and Fast: Exploiting Memory Hierarchy

- Lecture 22
  - 5.1 Introduction
  - 5.2 Memory Technologies

- Lecture 23 - 24
  - 5.3 The Basics of Caches
  - 5.4 Measuring and Improving Cache Performance
  - 5.5 Dependable Memory Hierarchy
  - 5.6 Virtual Machines

- Lecture 25
  - 5.6 Virtual Memory (By Dr. Song Wang)
  - 5.8 A Common Framework for Memory Hierarchy

- Lecture 26
  - 5.9 Using a Finite-State Machine to Control a Simple Cache
  - 5.10 Parallelism and Memory Hierarchies: Cache Coherence
  - 5.11 Parallelism and Memory Hierarchy: Redundant Arrays of Inexpensive Disks
  - 5.12 Advanced Material: Implementing Cache Controllers
  - 5.13 Real Stuff: The ARM Cortex-A8 and Intel Core i7 Memory Hierarchies
  - 5.14 Going Faster: Cache Blocking and Matrix Multiply
  - 5.15 Fallacies and Pitfalls
  - 5.16 Concluding Remarks
Green Boxes: Cache, on-chip, SRAM, fast, small, expensive
Blue Boxes: Main memory, off-chip, DRAM, slower, large not expensive

CPU is The chip.
In a bare machine, the only kind of address is a physical address

- The address of a memory byte
Virtual Memory

• Programs share main memory
  – Each gets a private virtual address space holding its frequently used code and data
  – Protected from other programs

• CPU and OS translate virtual addresses to physical addresses
  – VM “block” is called a page
  – VM translation “miss” is called a page fault

• Use main memory as a “cache” for secondary (disk) storage
  – Managed jointly by CPU hardware and the operating system (OS)
Virtual Memory to Physical Memory Mapping Example
Virtual Memory: Motivations and Benefits

- **Protection**: to allow efficient and safe sharing of memory among multiple programs
  - Conventional multi-programming, time-sharing OS
  - Today for the memory needed by multiple virtual machines for cloud computing

- **Virtualization**: to remove the programming burdens of a small, limited amount of main memory.
  - 4G memory space of 32-bit OS/machine even with << 4GB physical memory, e.g. 256MB
    - Each sees 0x00000000 – 0xFFFFFFFF memory

- **Relocation**: simplifies loading the program for execution.
  - allows the same program to run in any location in physical memory.

- It is called **Virtual Memory**, thus NOT REAL memory
Address Translation

- Fixed-size pages (e.g., 4K)
  - 4K \(2^{12}\) bytes per page
  - 12 bits to address a byte within a page

- Address translation: to map the upper \([31:12]\) bits of the virtual address, i.e. virtual page number, to a physical page number.

![Diagram of address translation]

Virtual address:

\[
\begin{array}{cccccccccccc}
31 & 30 & 29 & 28 & 27 & \cdots & 15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & \cdots & 3 & 2 & 1 & 0 \\
\end{array}
\]

Virtual page number: \(31\ 30\ 29\ 28\ 27\ \cdots\ 15\ 14\ 13\ 12\ 11\ 10\ 9\ 8\ \cdots\ 3\ 2\ 1\ 0\)

Physical page number: \(29\ 28\ 27\ \cdots\ 15\ 14\ 13\ 12\ 11\ 10\ 9\ 8\ \cdots\ 3\ 2\ 1\ 0\)

Page offset: \(3\ 2\ 1\ 0\)

Disk addresses

Address translation

Physical addresses

Diagram of address translation
The physical address of 0x0FFE1230

- A 32-bit computer with 1Gbyte physical memory
  - Each process (the execution of a program) can access $2^{32}$ (4G) bytes of address space, thus a virtual address has 32 bits
    - For 4K-byte ($2^{12}$) of pages, a process can have $2^{20}$ virtual pages
    - For 1Gbyte of physical memory, which is $2^{30}$ bytes
      - We have $2^{18}$ physical pages, → a physical page number has 18 bits
- Address translation
  - No need to translate the lower 12 bit (230) since it is for addressing a byte within a page, i.e. 0x0FFE1230
  - Only need to translate 0x0FFE1 (virtual page number) to physical page number in 18 bits, e.g. 0x16AB4 (first 1 only has 2 bits)
  - Thus the physical address is: 0x16AB4230
Translation Using a Page Table

Virtual address

Virtual page number

Page offset

Page table register

0xOFFE1230

Valid

Physical page number

2^20 entries

Page table

If 0 then page is not present in memory

0x16AB4230

Physical page number

Page offset

Physical address
Page Tables for Address Translation

- Stores mapping information between virtual page number to physical page number
  - Array of page table entries, indexed by virtual page number
  - Page table register in CPU points to page table in physical memory

- If page is present in memory
  - PTE stores the physical page number
  - Plus other status bits (referenced, dirty, ...)  

- If page is not present
  - PTE can refer to location in swap space on disk

![Page Table Entry Diagram]
Each user (process) has a page table
Page table contains an entry for each user page
  Each entry stores the physical page address and other info
Where Should Page Tables Reside?

• Space required by the page tables (PT) is proportional to the address space, number of users, ...
  ⇒ Too large to keep in registers

• Idea: Keep PTs in the main memory
  – needs one reference to retrieve the page base address and another to access the data word
  • ⇒ doubles the number of memory references!

• A page table register is used to store the address of the page table
• Each user has her own page table
Mapping Pages to Storage

- **Demand paging**
  - Valid bit to indicate whether a page is in physical mem or not
Linear Page Table Example (4K pages)

- 32-bit address
- 4K-size page
- 4-byte PTE
Linear Page Table

• With 32-bit addresses, 4-KB pages & 4-byte PTEs:
  – $2^{20}$ PTEs, i.e, 4 MB page table per user
  – 4 GB of swap needed to back up full virtual address space

• Larger pages?
  – Internal fragmentation (Not all memory in page is used)
  – Larger page fault penalty (more time to read from disk)

• What about 64-bit virtual address space???
  – How many page table entries (PTEs)?
Page Fault and Page Fault Penalty

• Page Fault: when fetching a byte whose page is not in memory

  ld x10, 0x3540(x5)

    Virtual address

• On page fault, the page must be fetched from disk
  – Takes millions of clock cycles
  – Handled by OS code

• Try to minimize page fault rate
  – Fully associative placement
  – Smart replacement algorithms
Page Fault Handler

- Use faulting virtual address to find PTE
- Locate page on disk
- Choose page to replace
  - If dirty, write to disk first
- Read page into memory and update page table
- Make process runnable again
  - Restart from faulting instruction
Replacement and Writes

• To reduce page fault rate, prefer least-recently used (LRU) replacement
  – Reference bit (aka use bit) in PTE set to 1 on access to page
  – Periodically cleared to 0 by OS
  – A page with reference bit = 0 has not been used recently

• Disk writes take millions of cycles
  – Block at once, not individual locations
  – Write through is impractical
  – Use write-back
  – Dirty bit in PTE set when page is written
Fast Translation Using a TLB

TLB are cache (in SRAM) for page tables

• Address translation would appear to require extra memory references
  – One to access the PTE
  – Then the actual memory access

• But access to page tables has good locality
  – So use a fast cache of PTEs within the CPU
  – Called a Translation Look-aside Buffer (TLB)
  – Typical: 16–512 PTEs, 0.5–1 cycle for hit, 10–100 cycles for miss, 0.01%–1% miss rate
  – Misses could be handled by hardware or software
Fast Translation Using a TLB

SRAM (Cache)

DRAM (Mem)

FIGURE 5.29 The TLB acts as a cache of the page table for the entries that map to physical pages only. The TLB contains a subset of the virtual-to-physical page mappings that are in the
TLB Misses

• If page is in memory
  – Load the PTE from memory and retry
  – Could be handled in hardware
    • Can get complex for more complicated page table structures
  – Or in software
    • Raise a special exception, with optimized handler

• If page is not in memory (page fault)
  – OS handles fetching the page and updating the page table
  – Then restart the faulting instruction
TLB Miss Handler

• TLB miss indicates
  – Page present, but PTE not in TLB
  – Page not preset

• Must recognize TLB miss before destination register overwritten
  – Raise exception

• Handler copies PTE from memory to TLB
  – Then restarts instruction
  – If page not present, page fault will occur
TLB and Cache Interaction:
From VA to Byte via TLB and L-1 Cache

- If cache tag uses physical address
  - Need to translate before cache lookup
- Alternative: use virtual address tag
  - Complications due to aliasing
    - Different virtual addresses for shared physical address
From VA to Data via TLB and Cache

FIGURE 5.1 The basic structure of a memory hierarchy. By implementing the memory system as a hierarchy, the user has the illusion of a memory that is as large as the largest level of the hierarchy, but can be accessed as if it were all built from the fastest memory. Flash memory has replaced disks in many personal mobile devices, and may lead to a new level in the storage hierarchy for desktop and server computers; see Section 5.2.

FIGURE 5.31 Processing a read or a write-through in the Intrinsity FastMATH TLB and cache. If the TLB generates a hit, the cache can be accessed with the resulting physical address. For a read, the cache generates a hit or miss and supplies the data or causes a stall.
VM: On-Demand Paging and Swap: Protection, Virtualization and Relocation

- **Fixed-size pages** (e.g., 4K)

  \[
  \text{ld } x10, \text{ 0x3540}(x5)
  \]

- **Protection**: with multiple virtual address spaces, errors are confined to one address space
  - *Between programs (processes)*

- **Virtualization** via on-demand paging: move only frequently used pages to VM
  - *Principle of locality*

- **Relocation**: pages on disk can be loaded to any free physical pages
VM: Address Translation & Protection

- Every instruction and data access needs address translation and protection checks
  - Within a program: writes to EXE or Read-only segment are violations

- A good VM design needs to be fast (~ one cycle) and space efficient
Summary

• Virtual Memory:
  – Protection, Virtualization and Relocation

• Paging:
  – Page table, address translation
  – In main memory

• TLB:
  – Cache for page tables