Computer Architecture

Lecture 07 – Address Translation & Virtual Mem

Pengju Ren
Institute of Artificial Intelligence and Robotics
Xi’an Jiaotong University

http://gr.xjtu.edu.cn/web/pengjuren
In a bare machine, the only kind of address is a physical address, corresponding to address lines of actual hardware memory.
Managing Memory in Bare Machines

- Early machines only ran one program at a time, with this program having unrestricted access to all memory and all I/O devices
  - This simple memory management model was also used in turn by the first minicomputer and first microcomputer systems
- Subroutine libraries became popular, were written in location-independent form
  - Different programs use different combination of routines
- To run program on bare machines, use linker or loader program to relocate library modules to actual locations in physical memory
Dynamic Address Translation

- **Motivation**
  - In early machines, I/O was slow and each I/O transfer involved the CPU (programmed I/O)
  - Higher throughput possible if CPU and I/O of 2 or more programs were overlapped, how?
    → multiprogramming with DMA I/O devices, interrupts

- **Location-independent programs**
  - Programming and storage management ease
    → need for a *base* register

- **Protection**
  - Independent programs should not affect each other inadvertently
    → need for a *bound* register

- **Multiprogramming** drives requirement for resident supervisor software to manage context switches between multiple programs
Base and bounds registers are visible/accessible only when processor is running in the *supervisor mode*.
Separate Areas for Program and Data
(Scheme used on all Cray vector supercomputers prior to X1, 2002)

What is an advantage of this separation?
What about more base/bound pairs?
Can fold addition of base register into \((\text{register}+\text{immediate})\) address calculation using a carry-save adder (sums three numbers with only a few gate delays more than adding two numbers)
As users come and go, the storage is “fragmented”. Therefore, at some stage programs have to be moved around to compact the storage.
What do we need Virtual Memory for?

Reason 1: Adding Disks to Hierarchy

Need to devise a mechanism to “Connect” memory and disk in the memory hierarchy.
What do we need Virtual Memory for?

Reason 2: Simplifying Memory for Apps

- Applications should see the straightforward memory layout we saw earlier ->
- User-space applications should think they own all of memory
- So we give them a virtual view of memory
What do we need Virtual Memory for?

Reason 3: Protection Between Processes

- With a bare system, addresses issued with loads/stores are real **physical** addresses

- This means any program can issue any address, therefore can access any part of memory, even areas which it doesn’t own
  - Ex: The OS data structures

- We should send all addresses through a mechanism that the OS controls, before they make it out to DRAM - a **translation and protection mechanism**
Paged Memory Systems (How)

- Program-generated *(virtual or logical)* address split into:

<table>
<thead>
<tr>
<th>Page Number</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

- Page Table contains physical address of start of each fixed-sized page in virtual address space

- Paging makes it possible to store a large contiguous virtual memory space using non-contiguous physical memory pages
Each user has a page Table. Page Table contains an entry for each user page.
Paging Simplifies Allocation

- Fixed-size pages can be kept on OS free list and allocated as needed to any process
- Process memory usage can easily grow and shrink dynamically
- Paging suffers from internal fragmentation where not all bytes on a page are used
  - Much less of an issue than external fragmentation or compaction for common page sizes (4-8KB)
  - But one reason that many oppose move to larger page sizes
Where should Page Tables Reside?

- Space required by the page table (PT) is proportional to the address space, number of users, ...
  - Space requirement is large
  - Too expensive to keep in registers

- Idea: Keep PT of the current user in special registers
  - May not be feasible for large page tables
  - Increases the cost of context swap

- Idea: Keep PTs in the main memory
  - Needs one reference to retrieve the page base address and another to access the data word (PTR, Page Table Register)
  - Double the number of memory references!
Page Tables Live in Memory

Simple linear page tables are too large, so **hierarchical page tables** are commonly used (see later)

Common for modern OS to place page tables in kernel’s virtual memory (page tables can be swapped to secondary storage)
Suppose an instruction references a memory page that is not in DRAM?

- We get a exception of type “page fault”
- Page fault handler does the following:
  - If virtual page doesn’t yet exist, assign an unused page in DRAM
  - Initiate transfer of the page we’re requesting from disk to DRAM, assigning to an unused page
  - If no unused page is left, a page currently in DRAM is selected to be replaced (based on usage)
  - The replaced page is written (back) to disk if it is ‘dirty’, page table entry that maps that VPN->PPN is marked as invalid/DPN
  - Page table entry of the page we’re requesting is updated with a (now) valid PPN
Coping with Limited Primary Storage

- Paging reduces fragmentation, but still many problems would not fit into primary memory, have to copy data to and from secondary storage (drum, disk).

- Two early approaches:
  - **Manual overlays**, programmer explicitly copies code and data in and out of primary memory
    - Tedious coding, error-prone (jumping to non-resident code?)
  - **Software interpretive coding** (Brooker 1960). Dynamic interpreter detects variables that are swapped out to drum and brings them back in
    - Simple for programmer, but inefficient

*Not just ancient black art, e.g., IBM Cell microprocessor using in Playstation-3 had explicitly managed local store! Many new “deep learning” accelerators have similar structure.*
Modern Virtual Memory Systems

*Illusion of a large, private, uniform store*

**Protection & Privacy**

Several users, each with their private address space and one or more shared address spaces

**Demand Paging**

Provides the ability to run programs larger than the primary memory

Hides differences in machine configurations

*The price is address translation on each memory reference*
Linear Page Table

- Page Table Entry (PTE) contains:
  - A bit to indicate if a page exists
  - **PPN** (physical page number) for a memory-resident page
  - **DPN** (disk page number) for a page on the disk
  - Status bits for protection and usage

- OS sets the Page Table Base Register whenever active user process changes
Hierarchical Page Table

Virtual Address from CPU

31 22 21 12 11 0

p1 p2 offset

10-bit L1 index
10-bit L2 index

Root of Current Page Table

(Processor Register, `satp` in RISC-V)

Level 1 Page Table

Level 2 Page Tables

Data Pages

Physical Memory

page in primary memory
page in secondary memory
PTE of a nonexistent page

RISC-V Sv32 Virtual Memory Scheme
Two-Level Page Tables in Physical Memory

Virtual Address Spaces

User 1

VA1

User 2

VA1

Physical Memory

Level 1 PT
User 1

Level 1 PT
User 2

User2/VA1

User1/VA1

Level 2 PT
User 2
Every instruction and data access needs address translation and protection checks. A good VM design needs to be fast (~ one cycle) and space efficient.

Idea: Cache the address translation of frequently used pages -- TLBs.
Translation-Lookaside Buffers (TLB)

Address translation is very expensive!
In a two-level page table, each reference becomes several memory accesses

Solution: Cache translations in TLB

- TLB hit $\Rightarrow$ Single-Cycle Translation
- TLB miss $\Rightarrow$ Page-Table Walk to refill

<table>
<thead>
<tr>
<th>V</th>
<th>R</th>
<th>W</th>
<th>D</th>
<th>tag</th>
<th>PPN</th>
</tr>
</thead>
</table>

- (VPN = virtual page number)
- (PPN = physical page number)

Virtual address

physical address

VPN offset

PPN offset

hit?
TLB Designs

- Typically 32-128 entries, usually fully associative
  - Each entry maps a large page, hence less spatial locality across pages ➔ more likely that two entries conflict
  - Sometimes larger TLBs (256-512 entries) are 4-8 way set-associative
  - Larger systems sometimes have multi-level (L1 and L2) TLBs

- Random or FIFO replacement policy

- TLB Reach: Size of largest virtual address space that can be simultaneously mapped by TLB
  - Example: 64 TLB entries, 4KB pages, one page per entry
    - TLB Reach = \( 64 \text{ entries} \times 4 \text{ KB} = 256 \text{ KB} \) (if contiguous)?
Variable-Size Page TLB

Some Systems support multiple page sizes

<table>
<thead>
<tr>
<th>Pagemask</th>
<th>Page Size</th>
<th>Pagemask</th>
<th>Page Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000_0000_0000</td>
<td>4KB</td>
<td>0000_1111_1111</td>
<td>1MB</td>
</tr>
<tr>
<td>0000_0000_0011</td>
<td>16KB</td>
<td>0011_1111_1111</td>
<td>4MB</td>
</tr>
<tr>
<td>0000_0000_1111</td>
<td>64KB</td>
<td>1111_1111_1111</td>
<td>16MB</td>
</tr>
<tr>
<td>0000_0011_1111</td>
<td>256KB</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MIPS using Pagemask mark different Page size (OS manage)
Handling a TLB Miss

**Software (MIPS, Alpha)**
- TLB miss causes an exception and the operating system walks the page tables and reloads TLB. A privileged “untranslated” addressing mode used for walk.
- Software TLB miss can be very expensive on out-of-order superscalar processor as requires a flush of pipeline to jump to trap handler.

**Hardware (SPARC v8, x86, PowerPC, RISC-V)**
- A memory management unit (MMU) walks the page tables and reloads the TLB.
- If a missing (data or PT) page is encountered during the TLB reloading, MMU gives up and signals a Page Fault exception for the original instruction.

**NOTE:** A given ISA can use either TLB miss strategy
Hierarchical Page Table Walk: SPARC v8

Virtual Address

Context Table Register

Context Register

Context Table

root ptr

Index 1

Index 2

Index 3

Offset

L1 Table

PTP

L2 Table

PTP

L3 Table

PTP

PTE

Physical Address

PPN

Offset

MMU does this table walk in hardware on a TLB miss
Page-Based Virtual-Memory Machine
(Hardware Page-Table Walk)

- Assumes page tables held in untranslated physical memory
Address Translation – Putting it all together

Virtual Address

- TLB Lookup
  - hit
  - miss
    - Page Table Walk
      - the page is
        - $\notin$ memory
        - $\in$ memory
          - Page Fault (OS loads page)
          - Update TLB
    - Protection Check
      - denied
      - permitted
        - Physical Address (to cache)

Where?
Names for Memory Locations

- Machine Language address
  - As specified in machine code

- Virtual Address
  - ISA specifies translation of machine code address into virtual address of program variable (sometime called effective address)

- Physical Address
  - Operating System specifies mapping of virtual address into name for a physical memory location
Page Fault Handler

- When the referenced page is not in DRAM:
  - The missing page is located (or created)
  - It is brought in from disk, and page table is updated
    - Another job may be run on the CPU while the first job waits for the requested page to be read from disk
  - If no free pages are left, a page is swapped out
    - Pseudo-LRU replacement policy, implemented in software

- Since it takes a long time to transfer a page (msecs), page faults are handled completely in software by OS
  - Untranslated addressing mode is essential to allow kernel to access page tables

- Keeping TLBs coherent with page table changes might require expensive “TLB shootdown”
  - Interrupt other processors to invalidate stale TLB entries
  - Some mainframes had hardware TLB coherence
Handling VM-related exceptions

- Handling a TLB miss needs a hardware or software mechanism to refill TLB
- Handling page fault (e.g., page is on disk) needs restartable exception so software handler can resume after retrieving page
  - Precise exceptions are easy to restart
  - Can be imprecise but restartable, but this complicates OS software
- A protection violation may abort process
  - But often handled the same as a page fault
Address Translation in CPU Pipeline

- Need to cope with additional latency of TLB:
  - slow down the clock?
  - pipeline the TLB and cache access?
  - virtual address caches
  - parallel TLB-cache access
Virtual-Address Caches

- one-step process in case of a hit (+)
- cache needs to be flushed on a context switch unless address space identifiers (ASIDs) included in tags (-)
- aliasing problems due to the sharing of pages (-)
- maintaining cache coherence (-)

Alternative: place the cache before the TLB

(StrongARM)
Sequential Access to TLB & Cache (Physical Index/Physical Tag)

Adding one more stage for TLB access will increase:
- For I-Cache the miss prediction penalty
- D-Cache load latency
Virtually Addressed Cache (Virtual Index/Virtual Tag)

Virtual Address

PC

Inst. Cache

Miss?

Inst. TLB

Physical Address

Instruction data

Decide

E

Page-Table Base Register

Virtual Address

Hardware Page Table Walker

Miss?

Data Cache

Physical Address

Memory Controller

Physical Address

Main Memory (DRAM)

Translate on *miss*
Aliasing in Virtual-Address Caches

Two virtual pages share one physical page. Virtual cache can have two copies of same physical data. Writes to one copy not visible to reads of other!

General Solution: *Prevent aliases coexisting in cache*

Software (i.e., OS) solution for direct-mapped cache

VAs of shared pages must agree in cache index bits; this ensures all VAs accessing same PA will conflict in direct-mapped cache (early SPARC)
Concurrent Access to TLB & Cache (Virtual Index/Physical Tag)

Index L is available without consulting the TLB → cache and TLB accesses can begin simultaneously!
Tag comparison is made after both accesses are completed

Cases: $L + b = k$, $L + b < k$, $L + b > k$

Actually, it is physical indexed Cache
Virtual-Index Physical-Tag Caches: 
Associative Organization

After the PPN is known, $2^a$ physical tags are compared.

*How does this scheme scale to larger caches?*
Concurrent Access to TLB & Large L1
The problem with L1 > Page size

Can VA₁ and VA₂ both map to same PA?
A solution via Second-Level Cache

Usually a common L2 cache backs up both Instruction and Data L1 caches

L2 is “inclusive” of both Instruction and Data caches
  • Inclusive means L2 has copy of any line in either L1
Anti-Aliasing Using L2 \cite{MIPS R10000, 1996}

- Suppose VA1 and VA2 both map to PA and VA1 is already in L1, L2 (VA1 ≠ VA2)
- After VA2 is resolved to PA, a collision will be detected in L2. \textbf{(Collision -> Field a is different)}
- VA1 will be purged from L1 and L2, and VA2 will be loaded \(⇒ no\ aliasing\)!
Anti-Aliasing using L2 for a Virtually Addressed L1

Physically-addressed L2 can also be used to avoid aliases in virtually-addressed L1
Why a Privileged Architecture?

- Profiles (Simple Embedded w/wo Protection, Unix-like OS, Cloud OS)
- Privileges and Modes
- Privileged Features
  - CSRs
  - Instructions
- Memory Addressing
  - Translation
  - Protection
- Trap Handling
  - Exceptions
  - Interrupts
- Counters
  - Time
  - Performance
RISC-V Privilege Modes

- **Machine mode (M-mode, highest privileges)**
  - A.K.A monitor mode, microcode mode, ...
- **Hypervisor-Extended Supervisor Mode (HS-Mode)**
- **Supervisor Mode (S-mode)**
- **User Mode (U-mode, lowest privileges)**

**Supported combinations of modes:**
- M (simple embedded systems)
- M, U (embedded systems with security)
- M, S, U (systems running Unix-like operating systems)
- M, S, HS, U (systems running hypervisors, Cloud OS Capable)
RISC-V System State

- Processor registers
  - Compute registers
    - General-purpose (x0-x31)
    - Optional floating-point (f0-f31)
    - Optional vector (v0-v31)
    - Optional custom
  - Control and status registers (CSRs)
    - Accessibility controlled by privilege mode or higher

- System main memory

- System I/O devices

- All system memory and device control registers mapped into flat machine physical address space
An optional physical memory protection (PMP) unit provides per-hart machine-mode control registers to allow physical memory access privileges (read, write, execute) to be specified for each physical memory region.
M-Mode controls PMPs

- M-mode has access to entire machine after reset
- Configures PMPs and ioPMPs to contain each active context inside a physical partition
- Can even restrict M-mode access to regions until next reset
- M-mode can dynamically swap PMP settings to run different security contexts on a hart
- RISC-V hardware thread (hart)
PMP checks are applied to all accesses when the hart is running in S or U modes
RISC-V Secure Embedded Systems (M, U modes)

- M-mode runs secure boot and runtime monitor
- Embedded code runs in U-mode
- Physical memory protection (PMP) on U-mode accesses
- Interrupt handling can be delegated to U-mode code
  - User-level interrupt support (N-extension)
- Provides arbitrary number of isolated security contexts
Interrupt and Privilege Change

- Combine privilege level change with interrupt/exception transfer
  - switch to next higher privilege level on interrupt
  - privilege level restored on return from interrupt

- Interrupt control transfer is only gateway to privileged mode
  - lower-level code can never escape into privileged mode
  - lower-level code don’t even need to know there is a privileged mode
RISC-V Virtual Memory Architectures (M, S, U modes)

- Designed to support current Unix-style operating systems
- Sv32 (RV32)
  - Demand-paged 32-bit virtual-address spaces
  - 2-level page table
  - 4 KiB pages, 4 MiB megapages
- Sv39 (RV64)
  - Demand-paged 39-bit virtual-address spaces
  - 3-level page table
  - 4 KiB pages, 2 MiB megapages, 1 GiB gigapages
- Sv48, Sv57, Sv64 (RV64)
  - Sv39 + 1/2/3 more page-table levels
S-Mode runs on top of M-mode

- M-mode runs secure boot and monitor
- S-mode runs OS
- U-mode runs application on top of OS or M-mode

- PMP checks are also applied to page-table accesses for virtual-address translation, for which the effective privilege mode is S. Optionally, PMP checks may additionally apply to M-mode accesses.
- PMP can grant permissions to S and U modes, which by default have none, and can revoke permissions from M-mode, which by default has full permissions.
Virtual Memory Use Today - 1

- Servers/desktops/laptops/smartphones have full demand-paged virtual memory
  - Portability between machines with different memory sizes
  - Protection between multiple users or multiple tasks
  - Share small physical memory among active tasks
  - Simplifies implementation of some OS features

- Vector supercomputers have translation and protection but rarely complete demand-paging
  (Older Crays: base&bound, Japanese & Cray X1/X2: pages)
  - Don’t waste expensive CPU time thrashing to disk (make jobs fit in memory)
  - Mostly run in batch mode (run set of jobs that fits in memory)
  - Difficult to implement restartable vector instructions
Most embedded processors and DSPs provide physical addressing only

- Can’t afford area/speed/power budget for virtual memory support
- Often there is no secondary storage to swap to!
- Programs custom written for particular memory configuration in product
- Difficult to implement restartable instructions for exposed architectures
Cache-TLB Interactions

- Physically indexed / Physically Tagged
- Virtually indexed / Virtually Tagged
- Virtually Indexed / Physically Tagged
  - Concurrent cache access with TLB Translation
- Both Indexed / Physically Tagged
  - Small enough cache or highly associative cache will have fewer indexes than page size
  - Concurrent cache access with TLB Translation
Caching v.s Demand Paging

**Caching**
- cache entry
- cache block (~32 bytes)
- cache miss rate (1% to 20%)
- cache hit (~1 cycle)
- cache miss (~100 cycles)
- a miss is handled in *hardware*

**Demand paging**
- page frame
- page (~4K bytes)
- page miss rate (<0.001%)
- page hit (~100 cycles)
- page miss (~5M cycles)
- a miss is handled mostly in *software*
## Summary

<table>
<thead>
<tr>
<th>Situation</th>
<th>TLB</th>
<th>Page Table</th>
<th>D-Cache</th>
<th>Physical Page</th>
<th>notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>D-Cache hit</td>
<td>Hit</td>
<td>TLB hit, it hit</td>
<td>Hit</td>
<td>D-Cache hit, it hit</td>
<td>No need to check Phys. Mem</td>
</tr>
<tr>
<td>D-Cache miss</td>
<td>Hit</td>
<td>TLB hit, it hit</td>
<td>Miss</td>
<td>Hit</td>
<td>Update D-Cache</td>
</tr>
<tr>
<td>TLB miss</td>
<td>Miss</td>
<td>Hit</td>
<td>hit</td>
<td>D-Cache hit, it hit</td>
<td>Update TLB, then access TLB again</td>
</tr>
<tr>
<td>TLB+D and Cache miss</td>
<td>Miss</td>
<td>Hit</td>
<td>Miss</td>
<td>Hit</td>
<td>Update TLB and D-Cache, then access TLB again</td>
</tr>
<tr>
<td>Page Fault</td>
<td>Miss</td>
<td>Miss</td>
<td>Miss</td>
<td>miss</td>
<td>Page Fault process</td>
</tr>
</tbody>
</table>

TLB、Page Table、D-Cache and Physical Page (Total $2^4=16$ cases)
- IF TLB hit, then Page Table will hit
- IF D-Cache hit, then Physical Page will hit
- IF Page Table hit, then Physical Page hit
Next Lecture: Branch Prediction
Acknowledgements

- Some slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - David Patterson (UCB)
  - David Wentzlaff (Princeton University)

- MIT material derived from course 6.823
- UCB material derived from course CS252 and CS 61C
Hashed Page Table: Approximating Associative Addressing

- Hashed Page Table is typically 2 to 3 times larger than the number of PPN’s to reduce collision probability.
- It can also contain DPN’s for some non-resident pages (not common).
- If a translation cannot be resolved in this table then the software consults a data structure that has an entry for every existing page (e.g., full page table).
Power PC: Hashed Page Table

- Each hash table slot has 8 PTE's \(<\text{VPN},\text{PPN}>\) that are searched sequentially.
- If the first hash slot fails, an alternate hash function is used to look in another slot.
  
  *All these steps are done in hardware!*

- Hashed Table is typically 2 to 3 times larger than the number of physical pages.
- The full backup Page Table is managed in software.
VM features track historical uses:

- **Bare machine, only physical addresses**
  - One program owned entire machine

- **Batch-style multiprogramming**
  - Several programs sharing CPU while waiting for I/O
  - Base & bound: translation and protection between programs (supports *swapping* entire programs but not demand-paged virtual memory)
  - Problem with external fragmentation (holes in memory), needed occasional memory defragmentation as new jobs arrived

- **Time sharing**
  - More interactive programs, waiting for user. Also, more jobs/second.
  - Motivated move to fixed-size page translation and protection, no external fragmentation (but now internal fragmentation, wasted bytes in page)
  - Motivated adoption of virtual memory to allow more jobs to share limited physical memory resources while holding working set in memory

- **Virtual Machine Monitors**
  - Run multiple operating systems on one machine
  - Idea from 1970s IBM mainframes, now common on laptops
    - e.g., run Windows on top of Mac OS X
  - Hardware support for two levels of translation/protection
    - Guest OS virtual -> Guest OS physical -> Host machine physical