CIS 371
Computer Organization and Design

Unit 8: Virtual Memory

Slides developed by Milo Martin & Amir Roth at the University of Pennsylvania
with sources that included University of Wisconsin slides
by Mark Hill, Guri Sohi, Jim Smith, and David Wood.
This Unit: Virtual Memory

- The operating system (OS)
  - A super-application
  - Hardware support for an OS
- Virtual memory
  - Page tables and address translation
  - TLBs and memory hierarchy issues
Readings

- P&H
  - Virtual Memory: 5.4
In-Class Exercise

• What is a “trie” data structure
  • Also called a “prefix tree”

• What is it used for?

• What properties does it have?
  • How is it different from a binary tree?
  • How is it different than a hash table?
A Computer System: Hardware

- CPUs and memories
  - Connected by memory bus
- **I/O peripherals**: storage, input, display, network, ...
  - With separate or built-in DMA
  - Connected by system bus (which is connected to memory bus)
A Computer System: + App Software

- **Application software**: computer must do something
A Computer System: + OS

- **Operating System (OS):** virtualizes hardware for apps
  - **Abstraction:** provides services (e.g., threads, files, etc.)
    - Simplifies app programming model, raw hardware is nasty
  - **Isolation:** gives each app illusion of private CPU, memory, I/O
    - Simplifies app programming model
    - Increases hardware resource utilization

![Diagram of computer system architecture](image)
Operating System (OS) and User Apps

- Sane system development requires a split
  - Hardware itself facilitates/enforces this split

- **Operating System (OS):** a super-privileged process
  - Manages hardware resource allocation/revocation for all processes
  - Has direct access to resource allocation features
  - Aware of many nasty hardware details
  - Aware of other processes
  - Talks directly to input/output devices (device driver software)

- **User-level apps:** ignorance is bliss
  - Unaware of most nasty hardware details
  - Unaware of other apps (and OS)
  - Explicitly denied access to resource allocation features
System Calls

- Controlled transfers to/from OS

- **System Call**: a user-level app “function call” to OS
  - Leave description of what you want done in registers
  - SYSCALL instruction (also called TRAP or INT)
    - Can’t allow user-level apps to invoke arbitrary OS code
    - Restricted set of legal OS addresses to jump to (**trap vector**)
  - Processor jumps to OS using trap vector
    - Sets privileged mode
  - OS performs operation
  - OS does a “return from system call”
    - Unsets privileged mode

- Used for I/O and other operating system services
Input/Output (I/O)

- Applications use “system calls” to initiate I/O
- Only operating system (OS) talks directly to the I/O device
  - Send commands, query status, etc.
  - OS software uses special uncached load/store operations
  - Hardware sends these reads/writes across I/O bus to device
- Hardware also provides “Direct Memory Access (DMA)”
  - For big transfers, the I/O device accesses the memory directly
  - Example: DMA used to transfer an entire block to/from disk
- Interrupt-driven I/O
  - The I/O device tells the software its transfer is complete
  - Tells the hardware to raise an “interrupt” (door bell)
  - Processor jumps into the OS
  - Inefficient alternative: polling
Interrupts

- **Exceptions**: synchronous, generated by running app
  - E.g., illegal insn, divide by zero, etc.
- **Interrupts**: asynchronous events generated externally
  - E.g., timer, I/O request/reply, etc.
- **“Interrupt” handling**: same mechanism for both
  - “Interrupts” are on-chip signals/bits
    - Either internal (e.g., timer, exceptions) or from I/O devices
  - Processor continuously monitors interrupt status, when one is high...
  - Hardware jumps to some preset address in OS code (interrupt vector)
  - Like an asynchronous, non-programmatic SYSCALL
- **Timer**: programmable on-chip interrupt
  - Initialize with some number of micro-seconds
  - Timer counts down and interrupts when reaches zero
A Computer System: + OS

![Diagram of a computer system with components connected through buses and a bridge]

- Application
- OS
- Memory bus
  - CPU/$
  - CPU/$
  - Memory
- System (I/O) bus
  - DMA
  - DMA
  - I/O ctrl
  - kbd
  - Disk
  - display
  - NIC

Bridge connects Memory bus and System (I/O) bus.
A Computer System: + OS

Diagram:
- OS
- Applications
- Memory bus
- System (I/O) bus
- CPU/$
- Memory
- DMA
- Disk
- Display
- NIC
- kbd
- bridge
Virtualizing Processors

• How do multiple apps (and OS) share the processors?
  • **Goal:** applications think there are an infinite # of processors

• Solution: time-share the resource
  • Trigger a **context switch** at a regular interval (~1ms)
    • **Pre-emptive:** app doesn’t yield CPU, OS forcibly takes it
      + Stops greedy apps from starving others
  • **Architected state:** PC, registers
    • Save and restore them on context switches
    • Memory state?
  • **Non-architected state:** caches, predictor tables, etc.
    • Ignore or flush

• Operating system responsible to handle context switching
  • Hardware support is just a timer interrupt
Virtualizing Main Memory

• How do multiple apps (and the OS) share main memory?
  • **Goal:** each application thinks it has infinite memory

• One app may want more memory than is in the system
  • App’s insn/data footprint may be larger than main memory
  • **Requires main memory to act like a cache**
    • With disk as next level in memory hierarchy (slow)
    • Write-back, write-allocate, large blocks or “pages”
  • No notion of “program not fitting” in registers or caches (why?)

• Solution:
  • Part #1: treat memory as a “cache”
    • Store the overflowed blocks in “swap” space on disk
  • Part #2: add a level of indirection (address translation)
Virtual Memory (VM)

- **Virtual Memory (VM):**
  - Level of indirection
  - Application generated addresses are **virtual addresses (VAs)**
    - Each process *thinks* it has its own $2^N$ bytes of address space
  - Memory accessed using **physical addresses (PAs)**
  - VAs translated to PAs at some coarse granularity (page)
  - OS controls VA to PA mapping for itself and all other processes
  - Logically: translation performed before every insn fetch, load, store
  - Physically: hardware acceleration removes translation overhead

![Diagram of Virtual Memory](image-url)
Virtual Memory (VM)

- Programs use **virtual addresses (VA)**
  - VA size (N) aka machine size (e.g., Core 2 Duo: 48-bit)
- Memory uses **physical addresses (PA)**
  - PA size (M) typically M<N, especially if N=64
  - \(2^M\) is most physical memory machine supports
- VA→PA at **page** granularity (VP→PP)
  - Mapping need not preserve contiguity
  - VP need not be mapped to any PP
  - Unmapped VPs live on disk (swap) or nowhere (if not yet touched)
VM is an Old Idea: Older than Caches

- Original motivation: **single-program compatibility**
  - IBM System 370: a family of computers with one software suite
    + Same program could run on machines with different memory sizes
      - Prior, programmers explicitly accounted for memory size
  - But also: **full-associativity + software replacement**
    - Memory $t_{\text{miss}}$ is high: extremely important to reduce $%_{\text{miss}}$

<table>
<thead>
<tr>
<th>Parameter</th>
<th>I$/D$</th>
<th>L2</th>
<th>Main Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_{\text{hit}}$</td>
<td>2ns</td>
<td>10ns</td>
<td>30ns</td>
</tr>
<tr>
<td>$t_{\text{miss}}$</td>
<td>10ns</td>
<td>30ns</td>
<td>10ms (10M ns)</td>
</tr>
<tr>
<td>Capacity</td>
<td>8–64KB</td>
<td>128KB–2MB</td>
<td>64MB–64GB</td>
</tr>
<tr>
<td>Block size</td>
<td>16–32B</td>
<td>32–256B</td>
<td>4+KB</td>
</tr>
<tr>
<td>Assoc./Repl.</td>
<td>1–4, LRU</td>
<td>4–16, LRU</td>
<td>Full, “working set”</td>
</tr>
</tbody>
</table>
Uses of Virtual Memory

• More recently: **isolation** and **multi-programming**
  - Each app thinks it has $2^N$ B of memory, its stack starts 0xFFFFFFFF,...
  - Apps prevented from reading/writing each other’s memory
    - Can’t even address the other program’s memory!

• **Protection**
  - Each page with a read/write/execute permission set by OS
  - Enforced by hardware

• **Inter-process communication.**
  - Map same physical pages into multiple virtual address spaces
  - Or share files via the UNIX `mmap()` call
Address Translation

- VA → PA mapping called **address translation**
  - Split VA into **virtual page number (VPN)** & **page offset (POFS)**
  - Translate VPN into **physical page number (PPN)**
  - POFS is not translated
  - VA → PA = [VPN, POFS] → [PPN, POFS]

- Example above
  - 64KB pages → 16-bit POFS
  - 32-bit machine → 32-bit VA → 16-bit VPN
  - Maximum 256MB memory → 28-bit PA → 12-bit PPN
Address Translation Mechanics I

- How are addresses translated?
  - In software (for now) but with hardware acceleration (a little later)

- Each process allocated a page table (PT)
  - Software data structure constructed by OS
  - Maps VPs to PPs or to disk (swap) addresses
    - VP entries empty if page never referenced
  - Translation is table lookup
Page Table Example

Example: Memory access at address 0xFFA8AFBA

Address of Page Table Root: 0xFFFFF87F8

Virtual Page Number: 111111110101000
Page Offset: 101011111011100

Physical Address: 111110101111 101011111011100

Physical Page Number: 111110101111
Page Offset: 101011111011100
Page Table Size

- How big is a page table on the following machine?
  - 32-bit machine
  - 4B page table entries (PTEs)
  - 4KB pages

  32-bit machine → 32-bit VA → $2^{32} = 4$GB virtual memory
  4GB virtual memory / 4KB page size → 1M VPs
  1M VPs * 4 Bytes per PTE → 4MB

- How big would the page table be with 64KB pages?
- How big would it be for a 64-bit machine?
- Page tables can get big
  - There are ways of making them smaller
Multi-Level Page Table (PT)

• One way: multi-level page tables
  • Tree of page tables ("trie")
  • Lowest-level tables hold PTEs
  • Upper-level tables hold pointers to lower-level tables
  • Different parts of VPN used to index different levels

• 20-bit VPN
  • Upper 10 bits index 1st-level table
  • Lower 10 bits index 2nd-level table
  • In reality, often more than 2 levels
Multi-Level Address Translation

Example: Memory access at address 0xFFFFA8AFBA

Address of Page Table Root
0xFFFF87F8

Virtual Page Number
1111111110

Page Offset
1010001010

111111011100

Physical Address:
0xFFFFF87F8

Physical Page Number
1111111110

Page Offset
1010001010

0xFFFFFA88

0xFFFFCBE8

0xFFFFCBE8

0xFFFFCBE8

...
Multi-Level Page Table (PT)

- Have we saved any space?
  - Isn’t total size of 2nd level tables same as single-level table (i.e., 4MB)?
  - Yes, but...

- Large virtual address regions _unused_
  - Corresponding 2nd-level tables need not exist
  - Corresponding 1st-level pointers are _null_

- How large for contiguous layout of 256MB?
  - Each 2nd-level table maps 4MB of virtual addresses
  - One 1st-level + 64 2nd-level pages
  - 65 total pages = 260KB (much less than 4MB)
Page-Level Protection

- **Page-level protection**
  - Piggy-back page-table mechanism
  - Map VPN to PPN + Read/Write/Execute permission bits
  - Attempt to execute data, to write read-only data?
    - Exception → OS terminates program
  - Useful (for OS itself actually)
ARMv8-A: page table information

<table>
<thead>
<tr>
<th>VA Bits &lt;47:39&gt;</th>
<th>VA Bits &lt;38:30&gt;</th>
<th>VA Bits &lt;29:21&gt;</th>
<th>VA Bits &lt;20:12&gt;</th>
<th>VA Bits &lt;11:0&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 1 table index</td>
<td>Level 2 table index</td>
<td>Level 3 table index</td>
<td>Level 4 table (page) index</td>
<td>Page offset address</td>
</tr>
</tbody>
</table>

- 4-level lookup, 4KB translation granule, 48-bit address
- 9 address bits per level

<table>
<thead>
<tr>
<th>VA Bits &lt;41:29&gt;</th>
<th>VA Bits &lt;28:16&gt;</th>
<th>VA Bits &lt;15:0&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 1 table index</td>
<td>Level 2 table (page) index</td>
<td>Page offset address</td>
</tr>
</tbody>
</table>

- 2-level lookup, 64KB page/page table size, 42-bit address
- 13 address bits per level
- 3 levels for 48 bits of VA – top level table is a partial table

<table>
<thead>
<tr>
<th>6 3</th>
<th>5 2</th>
<th>4 8</th>
<th>1 2</th>
<th>2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Upper attributes</td>
<td>SBZ</td>
<td>Address out</td>
<td>SBZ</td>
<td>Lower attributes and validity</td>
</tr>
</tbody>
</table>

64-bit Translation table entry format

ARMv8 Technology Preview By Richard Grisenthwaite Lead Architect and Fellow. ARM
CIS 371: Comp. Org.  | Prof. Milo Martin  | Virtual Memory
Address Translation Mechanics II

- Conceptually
  - Translate VA to PA before every cache access
  - Walk the page table before every load/store/insn-fetch
    - Would be terribly inefficient (even in hardware)

- In reality
  - **Translation Lookaside Buffer (TLB):** cache translations
  - Only walk page table on TLB miss

- Hardware truisms
  - Functionality problem? Add indirection (e.g., VM)
  - Performance problem? Add cache (e.g., TLB)
Translation Lookaside Buffer

- **Translation lookaside buffer (TLB)**
  - Small cache: 16–64 entries
  - Associative (4+ way or fully associative)
  - Exploits temporal locality in page table
  - What if an entry isn’t found in the TLB?
    - Invoke TLB miss handler

![Diagram showing CPU, TLB, L2, Main Memory, and their interactions. The diagram includes virtual address (VA) and physical address (PA) mappings.](image-url)
Serial TLB & Cache Access

- **“Physical” caches**
  - Indexed and tagged by **physical addresses**
    - Natural, “lazy” sharing of caches between apps/OS
    - VM ensures isolation (via **physical addresses**)
    - No need to do anything on context switches
    - Multi-threading works too
    - Cached inter-process communication works
      - Single copy indexed by physical address
      - Slow: adds at least one cycle to $t_{hit}$

- **Note:** **TLBs are by definition “virtual”**
  - Indexed and tagged by **virtual addresses**
  - Flush across context switches
  - Or extend with process identifier tags (x86)
Parallel TLB & Cache Access

- Two ways to look at VA
  - Cache: tag+index+offset
  - TLB: VPN+page offset

- Parallel cache/TLB...
  - If address translation doesn’t change index
  - That is, VPN/index must not overlap
Parallel TLB & Cache Access

What about parallel access?

- Only if...
  \[
  \frac{\text{cache size}}{\text{associativity}} \leq \text{page size}
  \]
- Index bits same in virtual and physical addresses!
- Access TLB in parallel with cache
  - Cache access needs tag only at very end
    + Fast: no additional \( t_{\text{hit}} \) cycles
    + No context-switching/aliasing problems
- Dominant organization used today
- Example: Core 2, 4KB pages, 32KB, 8-way SA L1 data cache
  - Implication: associativity allows bigger caches
TLB Organization

- **Like caches**: TLBs also have ABCs
  - Capacity
  - Associativity (At least 4-way associative, fully-associative common)
  - What does it mean for a TLB to have a block size of two?
    - Two consecutive VPs share a single tag
  - **Like caches**: there can be second-level TLBs

- Example: AMD Opteron
  - 32-entry fully-assoc. TLBs, 512-entry 4-way L2 TLB (insn & data)
  - 4KB pages, 48-bit virtual addresses, four-level page table

- **Rule of thumb**: TLB should “cover” size of on-chip caches
  - In other words: (#PTEs in TLB) * page size ≥ cache size
  - Why? Consider relative miss latency in each...
**TLB Misses**

- **TLB miss:** translation not in TLB, but in page table
  - Two ways to “fill” it, both relatively fast

- **Hardware-managed TLB:** e.g., x86, recent SPARC, ARM
  - Page table root in hardware register, hardware “walks” table
    - Latency: saves cost of OS call (avoids pipeline flush)
    - Page table format is hard-coded

- **Software-managed TLB:** e.g., Alpha, MIPS
  - Short (~10 insn) OS routine walks page table, updates TLB
    - Keeps page table format flexible
    - Latency: one or two memory accesses + OS call (pipeline flush)

- Trend is towards hardware TLB miss handler
TLB Misses and Pipeline Stalls

- TLB misses stall pipeline just like data hazards...
  - ...if TLB is hardware-managed

- If TLB is software-managed...
  - ...must generate an interrupt
  - Hardware will not handle TLB miss
Page Faults

- **Page fault**: PTE not in TLB or page table
  - → page not in memory
  - Or no valid mapping → segmentation fault
  - Starts out as a TLB miss, detected by OS/hardware handler

- **OS software routine**: 
  - Choose a physical page to replace
    - "**Working set**": refined LRU, tracks active page usage
  - If dirty, write to disk
  - Read missing page from disk
    - Takes so long (~10ms), OS schedules another task
  - Requires yet another data structure: **frame map**
    - Maps physical pages to <process, virtual page> pairs
  - Treat like a normal TLB miss from here
Summary

- OS virtualizes memory and I/O devices

- Virtual memory
  - “infinite” memory, isolation, protection, inter-process communication
  - Page tables
  - Translation buffers
    - Parallel vs serial access, interaction with caching
  - Page faults