Memory Hierarchy
(Main Memory & Virtual Memory & paging)

Memory (Programmer’s View)
Abstraction: Virtual vs. Physical Memory

- **Programmer** sees virtual memory
  - Can assume the memory is “infinite”
- **Reality:** Physical memory size is much smaller than what the programmer assumes
- **The system** (system software + hardware, cooperatively) maps virtual memory addresses are to physical memory
  - The system automatically manages the physical memory space transparently to the programmer

A System with Physical Memory Only

- **Examples:**
  - most Cray machines
  - early PCs
  - nearly all embedded systems

CPU’s load or store addresses used directly to access memory
The Problem

• Physical memory is of limited size (cost)
  – What if you need more?
  – Should the programmer be concerned about the size of code/data blocks fitting physical memory?
  – Should the programmer manage data movement from disk to physical memory?
  – Should the programmer ensure two processes do not use the same physical memory?

• Also, ISA can have an address space greater than the physical memory size
  – E.g., a 64-bit address space with byte addressability
  – What if you do not have enough physical memory?

Difficulties of Direct Physical Addressing

• Programmer needs to manage physical memory space
  – Inconvenient & hard
  – Harder when you have multiple processes

• Difficult to support code and data relocation

• Difficult to support multiple processes
  – Protection and isolation between multiple processes
  – Sharing of physical memory space

• Difficult to support data/code sharing across processes
Virtual Memory

• Idea: Give the programmer the illusion of a large address space while having a small physical memory
  – So that the programmer does not worry about managing physical memory

• Programmer can assume he/she has “infinite” amount of physical memory

• Hardware and software cooperatively and automatically manage the physical memory space to provide the illusion
  – Illusion is maintained for each independent process

Basic Mechanism

• Indirection (in addressing)

• Address generated by each instruction in a program is a “virtual address”
  – i.e., it is not the physical address used to address main memory
  – called “linear address” in x86

• An “address translation” mechanism maps this address to a “physical address”
  – called “real address” in x86
  – Address translation mechanism can be implemented in hardware and software together
A System with Virtual Memory (Page based)

- **Address Translation**: The hardware converts virtual addresses into physical addresses via an OS-managed lookup table (page table)

---

Virtual Pages, Physical Frames

- **Virtual** address space divided into **pages**
- **Physical** address space divided into **frames**
- A virtual page is mapped to
  - A physical frame, if the page is in physical memory
  - A location in disk, otherwise

- If an accessed virtual page is not in memory, but on disk
  - Virtual memory system brings the page into a physical frame and adjusts the mapping → this is called demand paging
- **Page table** is the table that stores the mapping of virtual pages to physical frames
Physical Memory as a Cache

- Physical memory is a cache for pages stored on disk
  - In fact, it is a fully associative cache in modern systems (a virtual page can be mapped to any physical frame)

- Similar caching issues exist as we have covered earlier:
  - Placement: where and how to place/find a page in cache?
  - Replacement: what page to remove to make room in cache?
  - Granularity of management: large, small, uniform pages?
  - Write policy: what do we do about writes? Write back?

Supporting Virtual Memory

- Virtual memory requires both HW+SW support
  - Page Table is in memory
  - Can be cached in special hardware structures called Translation Lookaside Buffers (TLBs)

- The hardware component is called the MMU (memory management unit)
  - Includes Page Table Base Register(s), TLBs, page walkers

- It is the job of the software to leverage the MMU to
  - Populate page tables, decide what to replace in physical memory
  - Change the Page Table Register on context switch (to use the running thread’s page table)
  - Handle page faults and ensure correct mapping
Some System Software Jobs for VM

- Keeping track of which physical frames are free
- Allocating free physical frames to virtual pages
- Page replacement policy
  - When no physical frame is free, what should be swapped out?
- Sharing pages between processes
- Copy-on-write optimization

Page Fault (“A Miss in Physical Memory”)

- If a page is not in physical memory but disk
  - Page table entry indicates virtual page not in memory
  - Access to such a page triggers a page fault exception
  - OS trap handler invoked to move data from disk into memory
    » Other processes can continue executing
    » OS has full control over placement
Servicing a Page Fault

• (1) Processor signals controller
  – Read block of length $P$ starting at disk address $X$ and store starting at memory address $Y$

• (2) Read occurs
  – Direct Memory Access (DMA)
  – Under control of I/O controller

• (3) Controller signals completion
  – Interrupt processor
  – OS resumes suspended process

Page Table is Per Process

• Each process has its own virtual address space
  – Full address space for each program
  – Simplifies memory allocation, sharing, linking and loading.
Address Translation

- How to obtain the physical address from a virtual address?
- Page size specified by the ISA
  - VAX: 512 bytes
  - Today: 4KB, 8KB, 2GB, ... (small and large pages mixed together)
  - Trade-offs? (caches?)

- Page Table contains an entry for each virtual page
  - Called Page Table Entry (PTE)

Address Translation (III)

- Parameters
  - \( P = 2^p \) = page size (bytes).
  - \( N = 2^n \) = Virtual-address limit
  - \( M = 2^m \) = Physical-address limit

\[
\begin{array}{c}
\text{virtual page number} \\
\text{page offset} \\
\text{address translation} \\
\text{physical frame number} \\
\text{page offset} \\
\text{physical address}
\end{array}
\]

Page offset bits don’t change as a result of translation
**Address Translation (IV)**

- Separate (set of) page table(s) per process
- VPN forms index into page table (points to a page table entry)
- Page Table Entry (PTE) provides information about page

---

**Address Translation: Page Hit**

1) Processor sends virtual address to MMU
2-3) MMU fetches PTE from page table in memory
4) MMU sends physical address to L1 cache
5) L1 cache sends data word to processor
Address Translation: Page Fault

1) Processor sends virtual address to MMU
2-3) MMU fetches PTE from page table in memory
4) Valid bit is zero, so MMU triggers page fault exception
5) Handler identifies victim, and if dirty pages it out to disk
6) Handler pages in new page and updates PTE in memory
7) Handler returns to original process, restarting faulting instruction.

What Is in a Page Table Entry (PTE)?

- Page table is the "tag store" for the physical memory data store
  - A mapping table between virtual memory and physical memory
- PTE is the "tag store entry" for a virtual page in memory
  - Need a valid bit \( \rightarrow \) to indicate validity/presence in physical memory
  - Need tag bits (PFN) \( \rightarrow \) to support translation
  - Need bits to support replacement
  - Need a dirty bit to support "write back caching"
  - Need protection bits to enable access control and protection
Remember: Cache versus Page Replacement

- Physical memory (DRAM) is a cache for disk
  - Usually managed by system software via the virtual memory subsystem

- Page replacement is similar to cache replacement
- Page table is the "tag store" for physical memory data store

- What is the difference?
  - Required speed of access to cache vs. physical memory
  - Number of blocks in a cache vs. physical memory
  - "Tolerable" amount of time to find a replacement candidate (disk versus memory access latency)
  - Role of hardware versus software

Page Replacement Algorithms

- If physical memory is full (i.e., list of free physical pages is empty), which physical frame to replace on a page fault?

- Is True LRU feasible?
  - 4GB memory, 4KB pages, how many possibilities of ordering?

- Modern systems use approximations of LRU
  - E.g., the CLOCK algorithm

- And, more sophisticated algorithms to take into account "frequency" of use
  - E.g., the ARC algorithm
CLOCK Page Replacement Algorithm

- Keep a circular list of physical frames in memory
- Keep a pointer (hand) to the last-examined frame in the list
- When a page is accessed, set the R bit in the PTE
- When a frame needs to be replaced, replace the first frame that has the reference (R) bit not set, traversing the circular list starting from the pointer (hand) clockwise
  - During traversal, clear the R bits of examined frames
  - Set the hand pointer to the next frame in the list

Page Size Trade Offs

- What is the granularity of management of physical memory?
- Large vs. small pages
- Tradeoffs have analogies to large vs. small cache blocks

- Many different tradeoffs with advantages and disadvantages
  - Size of the Page Table (tag store)
  - Reach of the Translation Lookaside Buffer (we will see this later)
  - Transfer size from disk to memory (waste of bandwidth?)
  - Waste of space within a page (internal fragmentation)
  - Waste of space within the entire physical memory (external fragmentation)
  - Granularity of access protection
Page-Level Access Control (Protection)

- Not every process is allowed to access every page
  - E.g., may need supervisor level privilege to access system pages

- Idea: Store access control information on a page basis in the process’s page table

- Enforce access control at the same time as translation

→ Virtual memory system serves two functions today
  - Address translation (for illusion of large physical memory)
  - Access control (protection)

---

VM as a Tool for Memory Access Protection

- Extend Page Table Entries (PTEs) with permission bits
- Check bits on each access and during a page fault
  - If violated, generate exception (Access Protection exception)
Privilege Levels in x86

Some Issues in Virtual Memory
Three Major Issues

- How large is the page table and how do we store and access it?
- How can we speed up translation & access control check?
- When do we do the translation in relation to cache access?
- There are many other issues we will not cover in detail
  - What happens on a context switch?
  - How can you handle multiple page sizes?
  - ...

Virtual Memory Issue I

- How large is the page table?
- Where do we store it?
  - In hardware?
  - In physical memory? (Where is the PTBR?)
  - In virtual memory? (Where is the PTBR?)
- How can we store it efficiently without requiring physical memory that can store all page tables?
  - Idea: multi-level page tables
  - Only the first-level page table has to be in physical memory
  - Remaining levels are in virtual memory (but get cached in physical memory when accessed)
**Issue: Page Table Size**

- Suppose 64-bit VA and 40-bit PA, how large is the page table?
  \[ 2^{52} \text{ entries} \times 4 \text{ bytes} \approx 16 \times 10^{15} \text{ Bytes} \]

  and that is for just one process!
  and the process may not be using the entire VM space!

**Solution: Multi-Level Page Tables**

Example from x86 architecture

- This page mapping example is for 4-KByte pages and the normal 32-bit physical address size.

*Physical Address*
Page Table Access

- How do we access the Page Table?

- Page Table Base Register (CR3 in x86)
- Page Table Limit Register

- If VPN is out of the bounds (exceeds PTLR) then the process did not allocate the virtual page → access control exception

- Page Table Base Register is part of a process’s context
  - Just like PC, status registers, general purpose registers
  - Needs to be loaded when the process is context-switched in

More on x86 Page Tables (I): Small Pages

Figure 4-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging
More on x86 Page Tables (II): Large Pages

Figure 4-3. Linear-Address Translation to a 4-Mbyte Page using 32-Bit Paging

x86 PTE (4KB page)

Table 4-6. Format of a 32-Bit Page-Table Entry that Maps a 4-KByte Page

<table>
<thead>
<tr>
<th>Bit Position(s)</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (P)</td>
<td>Present; must be 1 to map a 4-KByte page</td>
</tr>
<tr>
<td>1 (R/W)</td>
<td>Read/Write; if 0, writes may not be allowed to the 4-KByte page referenced by this entry (depends on CPL and CR0.WF; see Section 4.6)</td>
</tr>
<tr>
<td>2 (U/S)</td>
<td>User/supervisor; if 0, accesses with CPL = 3 are not allowed to the 4-KByte page referenced by this entry (see Section 4.6)</td>
</tr>
<tr>
<td>3 (PWT)</td>
<td>Page-level write-through; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9)</td>
</tr>
<tr>
<td>4 (PCD)</td>
<td>Page-privilege cache disable; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9)</td>
</tr>
<tr>
<td>5 (A)</td>
<td>Accessed; indicates whether software has accessed the 4-KByte page referenced by this entry (see Section 4.8)</td>
</tr>
<tr>
<td>6 (D)</td>
<td>Dirty; indicates whether software has written to the 4-KByte page referenced by this entry (see Section 4.8)</td>
</tr>
</tbody>
</table>
| 7 (PAT)         | If the PAT is supported, indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9); otherwise, reserved (must be 0)
| 8 (G)           | Global; if CR4.PGE = 1, determines whether the translation is global (see Section 4.10); ignored otherwise |
| 11:9            | Ignored |
| 31:12           | Physical address of the 4-KByte page referenced by this entry |
Virtual Memory Issue II

- How fast is the address translation?
  - How can we make it fast?

- Idea: Use a hardware structure that caches PTEs → Translation lookaside buffer

- What should be done on a TLB miss?
  - What TLB entry to replace?
  - Who handles the TLB miss? HW vs. SW?

- What should be done on a page fault?
  - What virtual page to replace from physical memory?
  - Who handles the page fault? HW vs. SW?
Speeding up Translation with a TLB

- Essentially a cache of recent address translations
  - Avoids going to the page table on every reference

- Index = lower bits of VPN (virtual page #)
- Tag = unused bits of VPN + process ID
- Data = a page-table entry
- Status = valid, dirty

The usual cache design choices (placement, replacement policy, multi-level, etc.) apply here too.

Handling TLB Misses

- The TLB is small; it cannot hold all PTEs
  - Some translations will inevitably miss in the TLB
  - Must access memory to find the appropriate PTE
    - Called walking the page directory/table
    - Large performance penalty

- Who handles TLB misses? Hardware or software?
Handling TLB Misses (II)

• Approach #1. **Hardware-Managed** (e.g., x86)
  – The hardware does the **page walk**
  – The hardware fetches the PTE and inserts it into the TLB
    » If the TLB is full, the entry **replaces** another entry
  – Done transparently to system software

• Approach #2. **Software-Managed** (e.g., MIPS)
  – The hardware raises an exception
  – The operating system does the **page walk**
  – The operating system fetches the PTE
  – The operating system inserts/evicts entries in the TLB

Handling TLB Misses (III)

• Hardware-Managed TLB
  – **Pro:** No exception on TLB miss. Instruction just stalls
  – **Pro:** Independent instructions may continue
  – **Pro:** No extra instructions/data brought into caches.
  – **Con:** Page directory/table organization is etched into the system: OS has little flexibility in deciding these

• Software-Managed TLB
  – **Pro:** The OS can define page table organization
  – **Pro:** More sophisticated TLB replacement policies are possible
  – **Con:** Need to generate an exception → performance overhead due to pipeline flush, exception handler execution, extra instructions brought to caches
Virtual Memory and Cache Interaction

Address Translation and Caching

- When do we do the address translation?
  - Before or after accessing the L1 cache?
- In other words, is the cache virtually addressed or physically addressed?
  - Virtual versus physical cache
- What are the issues with a virtually addressed cache?

- Synonym problem:
  - Two different virtual addresses can map to the same physical address → same physical address can be present in multiple locations in the cache → can lead to inconsistency in data
Homonyms and Synonyms

- **Homonym**: Same VA can map to two different PAs
  - Why?
    - VA is in different processes

- **Synonym**: Different VAs can map to the same PA
  - Why?
    - Different pages can share the same physical frame within or across processes
    - Reasons: shared libraries, shared data, copy-on-write pages within the same process, ...

---

**Cache-VM Interaction**

```
  CPU
  ▼
  TLB
  ▼
  cache
  ▼
  lower hier.

  virtual (L1) cache
```

```
  CPU
  ▼
  cache
  ▼
  lower hier.

  virtual-physical cache
```