A System Using Physical Addressing

Used in “simple” systems like embedded microcontrollers in devices like cars, elevators, and digital picture frames
A System Using Virtual Addressing

❖ Used in all modern servers, laptops, and smart phones
❖ One of the great ideas in computer science
Motivations

❖ **Use physical DRAM as a cache for the disk**
  - Address space of a process can exceed physical memory size
  - Multiple processes’ VM can exceed physical memory

❖ **Simplify memory management**
  - Multiple processes resident in main memory with their own address spaces
    - Provide virtually contiguous memory space
  - Only “active” code and data is actually in memory

❖ **Provide protection**
  - One process can’t interfere with another
  - User process cannot access privileged information (kernel area)
Conceptually, **virtual memory** is an array of \( N \) contiguous bytes stored on disk.

The contents of the array on disk are cached in **physical memory** (**DRAM cache**)

- These cache blocks are called **pages** (size is \( P = 2^p \) bytes)
DRAM Cache Organization

- **DRAM cache organization driven by enormous miss penalty**
  - DRAM is about $10x$ slower than SRAM
  - Disk is about $10,000x$ slower than DRAM

- **Consequences**
  - Large page (block) size: typically 4 KB, sometimes 4 MB
  - Fully associative
    - Any VP can be placed in any PP
    - Requires a “large” mapping function – different from cache memories
  - Highly sophisticated, expensive replacement algorithms
    - Too complicated and open-ended to be implemented in hardware
  - Write-back rather than write-through
Designing Cache for Disk

- **Design parameters**
  - Line (block) size?
    - Large, since disk better at transferring large blocks
  - Associativity? (Block placement)
    - High, to minimize miss rate
  - Block identification?
    - Using tables
  - Write through or write back? (Write strategy)
    - Write back, since can’t afford to perform small writes to disk
  - Block replacement?
    - Not to be used in the future
    - Based on LRU (Least Recently Used)
A page table is an array of page table entries (PTEs) that maps virtual pages to physical pages.

- Per-process kernel data structure in DRAM
Page Hit

- **Page hit**: reference to VM word that is in physical memory (DRAM cache hit)

![Diagram of page hit](image)

- Virtual address
- Physical page number or disk address
- Valid
- Memory resident page table (DRAM)
- Physical memory (DRAM)
- Virtual memory (disk)
Page Fault

- **Page fault**: reference to VM word that is not in physical memory (DRAM cache miss)

![Diagram showing virtual memory and page table]

- Virtual address
- Physical page number or disk address
- Memory resident page table (DRAM)
- Physical memory (DRAM)
- Virtual memory (disk)
Handling Page Fault

1. Page miss causes page fault (an exception)
2. Page fault handler selects a victim to be evicted (here VP 4)
1. Page miss causes page fault (an exception)
2. Page fault handler selects a victim to be evicted (here VP 4)
3. **Offending instruction is restarted: page hit!**

**Key point:** Waiting until the miss to copy the page to DRAM is known as *demand paging*
Allocating Pages

- Allocating a new page (VP 5) of virtual memory.

- Diagram showing the allocation process:
  - Memory resident page table (DRAM)
  - Physical memory (DRAM)
  - Virtual memory (disk)

- Table with columns:
  - Valid
  - Physical page number or disk address

- Example PTE entries:
  - PTE 0: Valid = 0, Physical page number = null
  - PTE 7: Valid = 1, Physical page number
Virtual memory seems terribly inefficient, but it works because of locality.

At any point in time, programs tend to access a set of active virtual pages called the **working set**
- Programs with better temporal locality will have smaller working sets

If (working set size < main memory size)
- Good performance for one process after compulsory misses

If ( SUM(working set sizes) > main memory size )
- *Thrashing:* Performance meltdown where pages are swapped (copied) in and out continuously
#2: VM for Memory Management

- **Key idea:** each process has its own virtual address space
  - It can view memory as a simple linear array
  - Mapping function scatters addresses through physical memory
    - Well-chosen mappings can improve locality
Ease of Management

- **Simplifying memory allocation**
  - Each virtual page can be mapped to any physical page
  - A virtual page can be stored in different physical pages at different times

- **Sharing code and data among processes**
  - Map virtual pages to the same physical page (here: PP 6)

---

Virtual Address Space for Process 1:

- VP 1
- VP 2
- ...
- N-1

Virtual Address Space for Process 2:

- VP 1
- VP 2
- ...
- N-1

Address translation

Physical Address Space (DRAM)

- PP 2
- PP 6
- PP 8
- ...
- M-1

(e.g., read-only library code)
**Simplifying Linking and Loading**

**Linking**
- Each program has similar virtual address space
- Code, data, and heap always start at the same addresses.

**Loading**
- `execve` allocates virtual pages for .text and .data sections & creates PTEs marked as invalid
- The .text and .data sections are copied, page by page, on demand by the virtual memory system
#3: VM for Memory Protection

- Extend PTEs with permission bits
- MMU checks these bits on each access
VM Address Translation

- **Virtual Address Space**
  - \( V = \{0, 1, \ldots, N-1\} \)

- **Physical Address Space**
  - \( P = \{0, 1, \ldots, M-1\} \)

- **Address Translation**
  - \( MAP: V \rightarrow P \cup \{\emptyset\} \)
  - For virtual address \( a \):
    - \( MAP(a) = a' \) if data at virtual address \( a \) is at physical address \( a' \) in \( P \)
    - \( MAP(a) = \emptyset \) if data at virtual address \( a \) is not in physical memory
      - Either invalid or stored on disk
Address Translation Symbols

❖ **Basic Parameters**
  - \( N = 2^n \): Number of addresses in virtual address space
  - \( M = 2^m \): Number of addresses in physical address space
  - \( P = 2^p \): Page size (bytes)

❖ **Components of the virtual address (VA)**
  - \( TLBI \): TLB index
  - \( TLBT \): TLB tag
  - \( VPO \): Virtual page offset
  - \( VPN \): Virtual page number

❖ **Components of the physical address (PA)**
  - \( PPO \): Physical page offset (same as VPO)
  - \( PPN \): Physical page number
Address Translation With a Page Table

Virtual address

Virtual page number (VPN) Virtual page offset (VPO)

Valid bit = 0:
Page not in memory (page fault)

Valid bit = 1

Page table

Valid Physical page number (PPN)

Physical page table address for the current process

Physical address

Physical page number (PPN) Physical page offset (PPO)
1) Processor sends virtual address to MMU
2-3) MMU fetches PTE from page table in memory
4) MMU sends physical address to cache/memory
5) Cache/memory sends data word to processor
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
Integrating VM and Cache

CPU Chip

CPU

VA

MMU

PTE

PTEA

PTEA hit

PTEA miss

PA

PA hit

PA miss

Memory

Data

L1 cache

PTE: page table entry, PTEA = PTE address

VA: virtual address, PA: physical address
Page table entries (PTEs) are cached in L1 like any other memory word
  - PTEs may be evicted by other data references
  - PTE hit still requires a small L1 delay

Solution: *Translation Lookaside Buffer (TLB)*
  - Small set-associative hardware cache in MMU
  - Maps virtual page numbers to physical page numbers
  - Contains complete page table entries for small number of pages
Accessing the TLB

- MMU uses the VPN portion of the virtual addresses to access the TLB:

![Diagram showing TLB access]

TLBT matches tag of line within set

TLBI selects the set

Set 0

Set 1

Set T-1

TLB tag (TLBT) | TLB index (TLBI) | VPO

VPN

T = 2^t sets

n-1 p+t p+t-1 p p-1 0
A TLB hit eliminates a memory access
A TLB miss incurs an additional memory access (the PTE)
Fortunately, TLB misses are rare. Why?
Suppose:
- 4KB ($2^{12}$) page size, 48-bit address space, 8-byte PTE

Problem:
- Would need a 512 GB page table!
  - $2^{48} \times 2^{-12} \times 2^3 = 2^{39}$ bytes

Common solution: Multi-level page table

Example: 2-level page table
- Level 1 table: each PTE points to a page table (always memory resident)
- Level 2 table: each PTE points to a page (paged in and out like any other data)
A Two-Level Page Table Hierarchy

Level 1
page table

Level 2
page tables

Virtual
memory

PTE 0
PTE 1
PTE 2 (null)
PTE 3 (null)
PTE 4 (null)
PTE 5 (null)
PTE 6 (null)
PTE 7 (null)
PTE 8
(1K - 9) null PTEs

PTE 0
... PTE 1023

PTE 0
... PTE 1023

PTE 0
... PTE 1023

PTE 0
... PTE 1023

VP 0
... VP 1023
VP 1024
... VP 2047

Gap

1023 unallocated pages
VP 9215

1023 unallocated pages

2K allocated VM pages for code and data

6K unallocated VM pages

1023 unallocated pages

1 allocated VM page for the stack

32 bit addresses, 4KB pages, 4-byte PTEs
Translating with a k-level Page Table
Programmer’s view of virtual memory
- Each process has its own private linear address space
- Cannot be corrupted by other processes

System view of virtual memory
- Uses memory efficiently by caching virtual memory pages
  - Efficient only because of locality
- Simplifies memory management and programming
  - Contiguously addressed virtual pages can be scattered on physical pages
- Simplifies protection by providing a convenient interposition point to check permissions