Virtual Memory: Concepts

15-213: Introduction to Computer Systems
17th Lecture, March 22, 2018

Instructors:
Franz Franchetti, Seth Copen Goldstein, and Brian Raling
Hmmm, How Does This Work?!

Solution: Virtual Memory (today and next lecture)
Today

- Address spaces
- VM as a tool for caching
- VM as a tool for memory management
- VM as a tool for memory protection
- Address translation
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
Address Spaces

- **Linear address space:** Ordered set of contiguous non-negative integer addresses:

  \{0, 1, 2, 3 \ldots \}\n
- **Virtual address space:** Set of \(N = 2^n\) virtual addresses

  \{0, 1, 2, 3, \ldots, N-1\}\n
- **Physical address space:** Set of \(M = 2^m\) physical addresses

  \{0, 1, 2, 3, \ldots, M-1\}
Why Virtual Memory (VM)?

- **Simplifies memory management**
  - Each process gets the same uniform linear address space

- **Isolates address spaces**
  - One process can’t interfere with another’s memory
  - User program cannot access privileged kernel information and code

- **Uses main memory efficiently**
  - Use DRAM as a cache for parts of a virtual address space
Today

- Address spaces
- VM as a tool for caching
- VM as a tool for memory management
- VM as a tool for memory protection
- Address translation
Remember: Set Associative Cache

E = 2: Two lines per set
Assume: cache block size 8 bytes

2 lines per set

Index to find set

S sets

Address:

Block offset

t bits 0...01 100
Fully Associative Cache

S=1: Assume: cache block size 8 bytes

All lines in 1 set
VM as a Tool for Caching

- 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)

![Diagram of virtual and physical memory]

- Virtual memory
  - VP 0
    - Unallocated
    - Cached
    - Uncached
    - Unallocated
  - VP 1
    - Cached
    - Uncached
  - VP $2^{n-p}-1$
    - Cached
    - Uncached

- Physical memory
  - PP 0
    - Empty
  - PP 1
    - Empty
  - PP $2^{m-p}-1$
    - Empty

Virtual pages (VPs) stored on disk

Physical pages (PPs) cached in DRAM
DRAM Cache Organization

- **DRAM cache organization driven by the 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
Enabling Data Structure: Page Table

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

<table>
<thead>
<tr>
<th>PTE 0</th>
<th>Physical page number or disk address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid</td>
<td>null</td>
</tr>
<tr>
<td>0</td>
<td>null</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PTE 7</th>
<th>Physical page number or disk address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid</td>
<td>null</td>
</tr>
<tr>
<td>0</td>
<td>null</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Physical memory (DRAM)

- PP 0
  - VP 1
  - VP 2
  - VP 7
  - VP 4

Virtual memory (disk)

- VP 1
- VP 2
- VP 3
- VP 4
- VP 6
- VP 7
Page Hit

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

![Diagram of virtual memory and page tables](image_url)
Page Fault

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

![Diagram of memory layout and page fault handling](image)
Handling Page Fault

Page miss causes page fault (an exception)
Handling Page Fault

- Page miss causes page fault (an exception)
- Page fault handler selects a victim to be evicted (here VP 4)
Handling Page Fault

- Page miss causes page fault (an exception)
- Page fault handler selects a victim to be evicted (here VP 4)
Handling Page Fault

- Page miss causes page fault (an exception)
- Page fault handler selects a victim to be evicted (here VP 4)
- 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 of memory allocation process](image)
Locality to the Rescue Again!

- 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
Today

- Address spaces
- VM as a tool for caching
- **VM as a tool for memory management**
- VM as a tool for memory protection
- Address translation
VM as a Tool 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

Virtual Address Space for Process 1:

Virtual Address Space for Process 2:

Address translation

Physical Address Space (DRAM)

(e.g., read-only library code)
VM as a Tool for Memory 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)
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
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
Today

- Address spaces
- VM as a tool for caching
- VM as a tool for memory management
- VM as a tool for memory protection
- Address translation
VM as a Tool for Memory Protection

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

<table>
<thead>
<tr>
<th>Process i:</th>
<th>SUP</th>
<th>READ</th>
<th>WRITE</th>
<th>EXEC</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 0:</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>PP 6</td>
</tr>
<tr>
<td>VP 1:</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 4</td>
</tr>
<tr>
<td>VP 2:</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>PP 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process j:</th>
<th>SUP</th>
<th>READ</th>
<th>WRITE</th>
<th>EXEC</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 0:</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>PP 9</td>
</tr>
<tr>
<td>VP 1:</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 6</td>
</tr>
<tr>
<td>VP 2:</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 11</td>
</tr>
</tbody>
</table>

Physical Address Space:
- PP 2
- PP 4
- PP 6
- PP 8
- PP 9
- PP 11
Break Time!

**skedaddle**: “To hurry somewhere"

Check out:

**Quiz: day 17: VM**

https://canvas.cmu.edu/courses/3822
volatile sig_atomic_t children = 0;
volatile sig_atomic_t handles = 0;

void handler(int sig) {
    handles++;
    while (wait(NULL) > 0) children++;
    return;
}

int main(int argc, char *argv[]) {
    int i;
    pid_t parent = getpid();
    signal(SIGUSR1, handler);
    for (i = 0 ; i < 5; i++) {
        if (fork() == 0) {
            kill(parent, SIGUSR1);
            exit(0);
        }
    }
    while (children < 5) /* Do nothing */;
    printf("handles = %d\n", handles);
    return 0;
}
int main(int argc, char *argv[]) {
    int fd1, fd2;
    char x, y, z;
    char *fname = argv[1];
    fd1 = open(fname, O_RDONLY, 0);
    fd2 = open(fname, O_RDONLY, 0);

    read(fd1, &x, 1);
    dup2(fd2, fd1);
    read(fd1, &y, 1);
    read(fd2, &z, 1);
    printf("x = %c, y = %c, z = %c\n", x, y, z);

    close(fd1);
    close(fd2);
    return 0;
}
Today

- Address spaces
- VM as a tool for caching
- VM as a tool for memory management
- VM as a tool for memory protection
- Address translation
VM Address Translation

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

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

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

Page table base register (PTBR) (CR3 in x86)

Physical page table address for the current process

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

Valid bit = 1

Physical page number (PPN)  Physical page offset (PPO)

Physical address
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 cache/memory
5) Cache/memory 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
Integrating VM and Cache

CPU Chip

CPU

VA

MMU

PTE

PTEA

PA

Memory

Data

PA

L1 cache

PTEA hit

PTEA miss

PA miss

PA hit

VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address
Speeding up Translation with a TLB

- 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
Summary of 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
Accessing the TLB

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

  TLBT matches tag of line within set
  TLBI selects the set

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

  T = 2^t sets

  n-1  p+t  p+t-1  p  p-1  0

  Set 0

  v  tag  PTE

  v  tag  PTE

  Set 1

  v  tag  PTE

  v  tag  PTE

  ...
TLB Hit

A TLB hit eliminates a memory access
A TLB miss incurs an additional memory access (the PTE)
Fortunately, TLB misses are rare. Why?
Multi-Level Page Tables

- **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
- VP 0
- ... (null)
- VP 1023
- VP 1024
- ... (null)
- VP 2047
- Gap

- 0

- 1023 null PTEs
- PTE 1023

- 1023 unallocated pages
- VP 9215
- ... (null)

- 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

Page table base register (PTBR)

VIRTUAL ADDRESS

VPN 1  VPN 2  ...  VPN k

the Level 1 page table  a Level 2 page table  a Level k page table

PHYSICAL ADDRESS

PPN

PPN

VPN k

VPO

0  p-1

m-1

n-1

p-1  0
Summary

- **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
  - Simplifies protection by providing a convenient interpositioning point to check permissions