Virtual Memory

CS 485G-006: Systems Programming
Lectures 18 and 19: 7-9 Mar 2016
Today

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

Adapted from slides by R. Bryant and D. O’Hallaron (http://csapp.cs.cmu.edu/3e/instructors.html)
A System Using Physical Addressing

- Used in “simple” systems like embedded microcontrollers in devices like cars, elevators, and digital picture frames

Adapted from slides by R. Bryant and D. O’Hallaron (http://csapp.cs.cmu.edu/3e/instructors.html)
A System Using Virtual Addressing

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

Adapted from slides by R. Bryant and D. O’Hallaron (http://csapp.cs.cmu.edu/3e/instructors.html)
Address Spaces

- **Linear address space:** Ordered set of contiguous non-negative integer addresses:
  
  \{0, 1, 2, 3 \ldots \}  

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

- **Physical address space:** Set of $M = 2^m$ physical addresses
  
  \{0, 1, 2, 3, \ldots, M-1\}
Why Virtual Memory (VM)?

- Uses main memory efficiently
  - Use RAM as a cache for parts of a virtual address space
  - Supports \textit{paging}: transferring data between RAM and disk

- 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
Today

- Address spaces
- **VM as a tool for caching**
- **VM as a tool for memory management**
- **VM as a tool for memory protection**
- Memory mapping
- Address translation
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)

Virtual memory

- VP 0
  - Unallocated
  - Cached
  - Uncached
- VP 1
  - Unallocated
  - Cached
  - Uncached
- VP \( 2^{n-p} - 1 \)
  - 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 (main memory) is about \(10x\) slower than SRAM (cache)
  - 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
  - Highly sophisticated, expensive replacement algorithms
    - Too complicated and open-ended to be implemented in hardware
  - Write-back rather than write-through
    - Data written into RAM is not immediately written to disk
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

![Page Table Diagram]

Adapted from slides by R. Bryant and D. O’Hallaron (http://csapp.cs.cmu.edu/3e/instructors.html)
Page Hit

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

Adapted from slides by R. Bryant and D. O’Hallaron (http://csapp.cs.cmu.edu/3e/instructors.html)
Page Fault

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

```
Virtual address

Physical page number or disk address

Valid 0 1 1 0 1 0 1
PTE 0 null
PTE 7

Physical memory (DRAM)
VP 1
VP 2
VP 7
VP 4

Virtual memory (disk)
VP 1
VP 2
VP 3
VP 4
VP 6
VP 7

Memory resident page table (DRAM)
```
## Handling Page Fault

- Page miss causes page fault (an exception)

### Diagram

- **Virtual address**
- **Physical page number or disk address**
- **Valid**
  - PTE 0: 0 (null), 1, 1, 0, 1
  - PTE 7: 0 (null), 0, 1
- **Memory resident page table (DRAM)**
- **Physical memory (DRAM)**
  - PP 0: VP 1
  - PP 3: VP 2, VP 7, VP 4
- **Virtual memory (disk)**
  - VP 1, VP 4
  - VP 6, VP 7

---

Adapted from slides by R. Bryant and D. O’Hallaron ([http://csapp.cs.cmu.edu/3e/instructors.html](http://csapp.cs.cmu.edu/3e/instructors.html))
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.

- Memory resident page table (DRAM)
- Physical memory (DRAM)
- Virtual memory (disk)
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

Adapted from slides by R. Bryant and D. O’Hallaron (http://csapp.cs.cmu.edu/3e/instructors.html)
Today

- Address spaces
- VM as a tool for caching
- VM as a tool for memory management
- VM as a tool for memory protection
- Memory mapping
- 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)

Adapted from slides by R. Bryant and D. O’Hallaron (http://csapp.cs.cmu.edu/3e/instructors.html)
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)

![Address translation diagram]

Virtual Address Space for Process 1:

Virtual Address Space for Process 2:

Adapted from slides by R. Bryant and D. O’Hallaron (http://csapp.cs.cmu.edu/3e/instructors.html)
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
- Memory mapping
- Address translation
VM as a Tool for Memory Protection

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

### Process i:

<table>
<thead>
<tr>
<th>VP 0:</th>
<th>VP 1:</th>
<th>VP 2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUP</td>
<td>READ</td>
<td>WRITE</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>EXEC</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>PP 6</td>
</tr>
<tr>
<td>Yes</td>
<td>PP 4</td>
</tr>
<tr>
<td>No</td>
<td>PP 2</td>
</tr>
</tbody>
</table>

### Process j:

<table>
<thead>
<tr>
<th>VP 0:</th>
<th>VP 1:</th>
<th>VP 2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUP</td>
<td>READ</td>
<td>WRITE</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>EXEC</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>PP 9</td>
</tr>
<tr>
<td>Yes</td>
<td>PP 6</td>
</tr>
<tr>
<td>Yes</td>
<td>PP 11</td>
</tr>
</tbody>
</table>

Adapted from slides by R. Bryant and D. O’Hallaron (http://csapp.cs.cmu.edu/3e/instructors.html)
Today

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

Adapted from slides by R. Bryant and D. O’Hallaron (http://csapp.cs.cmu.edu/3e/instructors.html)
Memory Mapping

- VM areas initialized by associating them with disk objects.
  - Process is known as **memory mapping**.

- Area can be **backed by** (i.e., get its initial values from):
  - **Regular file** on disk (e.g., an executable object file)
    - Initial page bytes come from a section of a file
  - **Anonymous file** (e.g., nothing)
    - First fault will allocate a physical page full of 0's (**demand-zero page**)
    - Once the page is written to (**dirtied**), it is like any other page

- Dirty pages are copied back and forth between memory and a special **swap file**.
Sharing Revisited: Shared Objects

- Process 1 maps the shared object.
Sharing Revisited: Shared Objects

- Process 2 maps the shared object.
- Notice how the virtual addresses can be different.
Sharing Revisited: Private Copy-on-write (COW) Objects

- Two processes mapping a private copy-on-write (COW) object.
- Area flagged as private copy-on-write
- PTEs in private areas are flagged as read-only

Adapted from slides by R. Bryant and D. O’Hallaron (http://csapp.cs.cmu.edu/3e/instructors.html)
Sharing Revisited: Private Copy-on-write (COW) Objects

- Instruction writing to private page triggers protection fault.
- Handler creates new R/W page.
- Instruction restarts upon handler return.
- Copying deferred as long as possible!

Adapted from slides by R. Bryant and D. O’Hallaron (http://csapp.cs.cmu.edu/3e/instructors.html)
Memory mapping

- The mmap() system call can be used to map a file into the process’s address space.
  - Sets up a page table entry with the “valid” bit cleared.
  - Accessing memory triggers a page fault, loading data from the file.
  - Can also create *anonymous* pages that aren’t backed by a file.

- $\text{ptr} = \text{mmap}(\text{start}, \text{length}, \text{ PROT_READ, MAP_PRIVATE, fd, offset})$
  - $\text{addr}$ is just a hint (usually use NULL: let the OS pick)
  - $\text{fd}$ is a *file descriptor* returned by the open() system call.
    - If you used fopen(), fileno(file) gives you the descriptor.
  - PROT_READ | PROT_WRITE | PROT_EXEC
  - Many MAP_* flags supported.
    - MAP_ANONYMOUS: no file (and use $\text{fd} = -1$)
  - *man mmap* for more information

Adapted from slides by R. Bryant and D. O’Hallaron (http://csapp.cs.cmu.edu/3e/instructors.html)
User-Level Memory Mapping

```c
void *mmmap(void *start, int len,
            int prot, int flags, int fd, int offset)
```

- `len`: bytes
- `start`: (or address chosen by kernel)
- `len` bytes
- `offset`: (bytes)
- `Disk file specified by file descriptor fd`
- `Process virtual memory`

Adapted from slides by R. Bryant and D. O’Hallaron (http://csapp.cs.cmu.edu/3e/instructors.html)
Example: Using mmap to Copy Files

- Read-then-write copies data twice (input -> buffer -> output)
- mmap avoids the buffer and lets the VM system handle moving data between disk and RAM.

```c
#include "csapp.h"

void mmapcopy(int fd, int size)
{
    /* Ptr to memory mapped area */
    char *bufp;

    bufp = Mmap(NULL, size,
                 PROT_READ,
                 MAP_PRIVATE,
                 fd, 0);
    Write(1, bufp, size);
    return;
}
```

```c
/* mmapcopy driver */
int main(int argc, char **argv)
{
    struct stat stat;
    int fd;

    /* Check for required cmd line arg */
    if (argc != 2) {
        printf("usage: %s <filename>
", argv[0]);
        argv[0]);
        exit(0);
    }

    /* Copy input file to stdout */
    fd = Open(argv[1], O_RDONLY, 0);
    Fstat(fd, &stat);
    mmapcopy(fd, stat.st_size);
    exit(0);
}
```
Today

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

■ Virtual Address Space
  ▪ \( V = \{0, 1, ..., N-1\} \)

■ Physical Address Space
  ▪ \( P = \{0, 1, ..., M-1\} \)

■ 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

Adapted from slides by R. Bryant and D. O’Hallaron (http://csapp.cs.cmu.edu/3e/instructors.html)
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)

n-1  p  p-1  0

Page table

Valid Physical page number (PPN)

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

Valid bit = 1

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

m-1  p  p-1  0

Physical address

Physical page table address for the current process

Page table base register (PTBR)
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

Adapted from slides by R. Bryant and D. O’Hallaron (http://csapp.cs.cmu.edu/3e/instructors.html)
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

Adapted from slides by R. Bryant and D. O’Hallaron (http://csapp.cs.cmu.edu/3e/instructors.html)
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

Adapted from slides by R. Bryant and D. O’Hallaron (http://csapp.cs.cmu.edu/3e/instructors.html)
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

  VPN

  T = 2^t sets

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

  Set 0

  v tag PTE

  Set 1

  v tag PTE

  Set T-1

  v tag PTE

Adapted from slides by R. Bryant and D. O’Hallaron (http://csapp.cs.cmu.edu/3e/instructors.html)
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

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

Level 2

Page Tables

PTE 0
PTE 1
PTE 2 (null)
PTE 3 (null)
PTE 4 (null)
PTE 5 (null)
PTE 6 (null)
PTE 7 (null)
PTE 8

1023 null PTEs
PTE 1023

Virtual Memory

VP 0

... VP 1023

VP 2047

Gap

1023 unallocated pages

VP 9215

6K unallocated VM pages

2K allocated VM pages for code and data

1023 unallocated pages

1 allocated VM page for the stack

32 bit addresses, 4KB pages, 4-byte PTEs

Adapted from slides by R. Bryant and D. O'Hallaron (http://csapp.cs.cmu.edu/3e/instructors.html)
Translating with a k-level Page Table

Page table base register (PTBR)

VIRTUAL ADDRESS

VIRTUAL PAGE NUMBER (VPN)

PHYSICAL PAGE NUMBER (PPN)

Adapted from slides by R. Bryant and D. O’Hallaron (http://csapp.cs.cmu.edu/3e/instructors.html)
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