Virtual Memory: Systems

15-213 / 18-213: Introduction to Computer Systems
17th Lecture, Oct. 27, 2011

Instructors:
Dave O’Hallaron, Greg Ganger, and Greg Kesden
Today

- Virtual memory questions and answers
- Simple memory system example
- Bonus: Case study: Core i7/Linux memory system
- Bonus: Memory mapping
Virtual memory reminder/review

**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
Recall: Address Translation With a Page Table

Virtual address

Page table base register (PTBR)

Page table address for process

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

Physical address

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

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

n-1   p   p-1   0

m-1   p   p-1   0
Recall: 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
Question #1

- Are the PTEs cached like other memory accesses?

- Yes (and no: see next question)
Page tables in memory, like other data

VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address
Question #2

- Isn’t it slow to have to go to memory twice every time?

- Yes, it would be... so, real MMUs don’t
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, dedicated, super-fast hardware cache of PTEs in MMU
  - Contains complete page table entries for small number of pages
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?
Question #3

- Isn’t the page table huge? How can it be stored in RAM?

- Yes, it would be... so, real page tables aren’t simple arrays
Multi-Level Page Tables

- **Suppose:**
  - 4KB \(2^{12}\) page size, 64-bit address space, 8-byte PTE

- **Problem:**
  - Would need a 32,000 TB page table!
    - \(2^{64} \times 2^{-12} \times 2^3 = 2^{55}\) bytes

- **Common solution:**
  - Multi-level page tables
  - 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 1023
PTE 0
... 
PTE 1023
PTE 0
... 
PTE 1023
PTE 0
... 
PTE 1023

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

2K allocated VM pages for code and data

6K unallocated VM pages

1023 unallocated pages

1 allocated VM page for the stack

1023 null PTEs

(1K - 9) null PTEs

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

1023 unallocated pages

VP 9215

32 bit addresses, 4KB pages, 4-byte PTEs
Translating with a k-level Page Table

![Diagram showing the translation process with k-level page tables]

- **VPN** (Virtual Page Number) levels from 1 to k
- **VPO** (Virtual Page Offset)
- **PPN** (Physical Page Number)
- **PPO** (Physical Page Offset)

The diagram illustrates the translation process from virtual to physical addresses using a k-level page table mechanism.
Question #4

- Shouldn’t fork() be really slow, since the child needs a copy of the parent’s address space?

- Yes, it would be... so, fork() doesn’t really work that way
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 \textit{private copy-on-write (COW)} object.
- Area flagged as private copy-on-write
- PTEs in private areas are flagged as read-only
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.
The `fork` Function Revisited

- VM and memory mapping explain how `fork` provides private address space for each process.

- To create virtual address for new new process
  - Create exact copies of current page tables
  - Flag each page in both processes as read-only
  - Flag writeable areas in both processes as private COW

- On return, each process has exact copy of virtual memory

- Subsequent writes create new pages using COW mechanism

- Perfect approach for common case of `fork()` followed by `exec()`
  - Why?
Today

- Virtual memory questions and answers
- Simple memory system example
- Bonus: Case study: Core i7/Linux memory system
- Bonus: Memory mapping
Review of 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
  - CO: Byte offset within cache line
  - CI: Cache index
  - CT: Cache tag
Simple Memory System Example

- **Addressing**
  - 14-bit virtual addresses
  - 12-bit physical address
  - Page size = 64 bytes

![Diagram](image_url)
# Simple Memory System Page Table

Only show first 16 entries (out of 256)

<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>28</td>
<td>1</td>
</tr>
<tr>
<td>01</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>02</td>
<td>33</td>
<td>1</td>
</tr>
<tr>
<td>03</td>
<td>02</td>
<td>1</td>
</tr>
<tr>
<td>04</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>05</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>06</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>07</td>
<td>–</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>08</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>09</td>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>0A</td>
<td>09</td>
<td>1</td>
</tr>
<tr>
<td>0B</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>0C</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>0D</td>
<td>2D</td>
<td>1</td>
</tr>
<tr>
<td>0E</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>0F</td>
<td>0D</td>
<td>1</td>
</tr>
</tbody>
</table>
Simple Memory System TLB

- 16 entries
- 4-way associative

<table>
<thead>
<tr>
<th>Set</th>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>03</td>
<td>–</td>
<td>0</td>
<td>09</td>
<td>0D</td>
<td>1</td>
<td>00</td>
<td>–</td>
<td>0</td>
<td>07</td>
<td>02</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>03</td>
<td>2D</td>
<td>1</td>
<td>02</td>
<td>–</td>
<td>0</td>
<td>04</td>
<td>–</td>
<td>0</td>
<td>0A</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>–</td>
<td>0</td>
<td>08</td>
<td>–</td>
<td>0</td>
<td>06</td>
<td>–</td>
<td>0</td>
<td>03</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>07</td>
<td>–</td>
<td>0</td>
<td>03</td>
<td>0D</td>
<td>1</td>
<td>0A</td>
<td>34</td>
<td>1</td>
<td>02</td>
<td>–</td>
<td>0</td>
</tr>
</tbody>
</table>
Simple Memory System Cache

- 16 lines, 4-byte block size
- Physically addressed
- Direct mapped

```
<table>
<thead>
<tr>
<th>Idx</th>
<th>Tag</th>
<th>Valid</th>
<th>B0</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>19</td>
<td>1</td>
<td>99</td>
<td>11</td>
<td>23</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>15</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>1B</td>
<td>1</td>
<td>00</td>
<td>02</td>
<td>04</td>
<td>08</td>
</tr>
<tr>
<td>3</td>
<td>36</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>1</td>
<td>43</td>
<td>6D</td>
<td>8F</td>
<td>09</td>
</tr>
<tr>
<td>5</td>
<td>0D</td>
<td>1</td>
<td>36</td>
<td>72</td>
<td>F0</td>
<td>1D</td>
</tr>
<tr>
<td>6</td>
<td>31</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>16</td>
<td>1</td>
<td>11</td>
<td>C2</td>
<td>DF</td>
<td>03</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Idx</th>
<th>Tag</th>
<th>Valid</th>
<th>B0</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>24</td>
<td>1</td>
<td>3A</td>
<td>00</td>
<td>51</td>
<td>89</td>
</tr>
<tr>
<td>9</td>
<td>2D</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>A</td>
<td>2D</td>
<td>1</td>
<td>93</td>
<td>15</td>
<td>DA</td>
<td>3B</td>
</tr>
<tr>
<td>B</td>
<td>0B</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>C</td>
<td>12</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>D</td>
<td>16</td>
<td>1</td>
<td>04</td>
<td>96</td>
<td>34</td>
<td>15</td>
</tr>
<tr>
<td>E</td>
<td>13</td>
<td>1</td>
<td>83</td>
<td>77</td>
<td>1B</td>
<td>D3</td>
</tr>
<tr>
<td>F</td>
<td>14</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
```
Address Translation Example #1

Virtual Address: 0x03D4

Physical Address

Byte: 0x36
Address Translation Example #2

Virtual Address: 0x0B8F

Physical Address

VPN 0x2E TLBI 2 TLBT 0x0B TLB Hit? N Page Fault? Y PPN: TBD

PPN CO CI CT CO

Byte: _____
Address Translation Example #3

Virtual Address: 0x0020

Physical Address

Byte: Mem
Today

- Virtual memory questions and answers
- Simple memory system example
- **Bonus:** Case study: Core i7/Linux memory system
- **Bonus:** Memory mapping
Intel Core i7 Memory System

Processor package

Core x4

- Registers
- Instruction fetch
- L1 d-cache 32 KB, 8-way
- L1 i-cache 32 KB, 8-way
- L2 unified cache 256 KB, 8-way
- L1 d-TLB 64 entries, 4-way
- L1 i-TLB 128 entries, 4-way
- L2 unified TLB 512 entries, 4-way
- QuickPath interconnect 4 links @ 25.6 GB/s each
- DDR3 Memory controller 3 x 64 bit @ 10.66 GB/s
  32 GB/s total (shared by all cores)
- Main memory

To other cores
To I/O bridge
Review of 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
  - \( CO \): Byte offset within cache line
  - \( CI \): Cache index
  - \( CT \): Cache tag
End-to-end Core i7 Address Translation

CPU ➔ Virtual address (VA) ➔ VPN ➔ VPO ➔ TLBT ➔ TLBI ➔ TLB

TLB

hit

miss

L1 TLB (16 sets, 4 entries/set)

VPN1 ➔ VPN2 ➔ VPN3 ➔ VPN4 ➔ PTE ➔ CT ➔ CI ➔ CO

Page tables

CR3

VPN ➔ 36 ➔ 12

32/64

Result ➔ L2, L3, and main memory

VPN1 ➔ 9

VPN2 ➔ 9

VPN3 ➔ 9

VPN4 ➔ 9

L1 hit ➔ L1 d-cache (64 sets, 8 lines/set)

Physical address (PA)

VPN ➔ 12

VPN ➔ 40

VPN ➔ 12

VPN ➔ 40

VPN ➔ 6

VPN ➔ 6
Core i7 Level 1-3 Page Table Entries

<table>
<thead>
<tr>
<th>63</th>
<th>62</th>
<th>52</th>
<th>51</th>
<th>12</th>
<th>11</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>XD</td>
<td>Unused</td>
<td>Page table physical base address</td>
<td>Unused</td>
<td>G</td>
<td>PS</td>
<td>A</td>
<td>CD</td>
<td>WT</td>
<td>U/S</td>
<td>R/W</td>
<td>P=1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Available for OS (page table location on disk) | P=0

Each entry references a 4K child page table

**P:** Child page table present in physical memory (1) or not (0).

**R/W:** Read-only or read-write access access permission for all reachable pages.

**U/S:** user or supervisor (kernel) mode access permission for all reachable pages.

**WT:** Write-through or write-back cache policy for the child page table.

**CD:** Caching disabled or enabled for the child page table.

**A:** Reference bit (set by MMU on reads and writes, cleared by software).

**PS:** Page size either 4 KB or 4 MB (defined for Level 1 PTEs only).

**G:** Global page (don’t evict from TLB on task switch)

**Page table physical base address:** 40 most significant bits of physical page table address (forces page tables to be 4KB aligned)
## Core i7 Level 4 Page Table Entries

<table>
<thead>
<tr>
<th>12</th>
<th>11</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>XD</td>
<td>Unused</td>
<td>Page physical base address</td>
<td>Unused</td>
<td>G</td>
<td>D</td>
<td>A</td>
<td>CD</td>
<td>WT</td>
<td>U/S</td>
<td>R/W</td>
<td>P=1</td>
</tr>
</tbody>
</table>

Available for OS (page location on disk) | P=0

Each entry references a 4K child page

**P:** Child page is present in memory (1) or not (0)

**R/W:** Read-only or read-write access permission for child page

**U/S:** User or supervisor mode access

**WT:** Write-through or write-back cache policy for this page

**CD:** Cache disabled (1) or enabled (0)

**A:** Reference bit (set by MMU on reads and writes, cleared by software)

**D:** Dirty bit (set by MMU on writes, cleared by software)

**G:** Global page (don’t evict from TLB on task switch)

**Page physical base address:** 40 most significant bits of physical page address (forces pages to be 4KB aligned)
Core i7 Page Table Translation

- **CR3**: Physical address of L1 PT
- **VPN 1**: 9
- **VPN 2**: 9
- **VPN 3**: 9
- **VPN 4**: 9
- **VPO**: 12
- **L1 PT**: Page global directory
- **L2 PT**: Page upper directory
- **L3 PT**: Page middle directory
- **L4 PT**: Page table
- **L1 PTE**: 512 GB region per entry
- **L2 PTE**: 1 GB region per entry
- **L3 PTE**: 2 MB region per entry
- **L4 PTE**: 4 KB region per entry
- **PPN**: Physical address
- **PPO**: Offset into physical and virtual page

Virtual address

Offset into physical and virtual page

Physical address

Physical address per entry
Cute Trick for Speeding Up L1 Access

Observation

- Bits that determine CI identical in virtual and physical address
- Can index into cache while address translation taking place
- Generally we hit in TLB, so PPN bits (CT bits) available next
- “Virtually indexed, physically tagged”
- Cache carefully sized to make this possible
Today

- Virtual memory questions and answers
- Simple memory system example
- Bonus: Case study: Core i7/Linux memory system
- Bonus: Memory mapping
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*. 
Demand paging

- **Key point:** no virtual pages are copied into physical memory until they are referenced!
  - Known as *demand paging*

- Crucial for time and space efficiency
User-Level Memory Mapping

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

- Map `len` bytes starting at offset `offset` of the file specified by file description `fd`, preferably at address `start`
  - `start`: may be 0 for “pick an address”
  - `prot`: PROT_READ, PROT_WRITE, ...
  - `flags`: MAP_ANON, MAP_PRIVATE, MAP_SHARED, ...

- Return a pointer to start of mapped area (may not be `start`)
User-Level Memory Mapping

void *mmap(void *start, int len,
           int prot, int flags, int fd, int offset)

Disk file specified by file descriptor fd

Process virtual memory

len bytes (or address chosen by kernel)
Using mmap to Copy Files

- Copying without transferring data to user space.

```c
#include "csapp.h"

/* mmapcopy - uses mmap to copy file fd to stdout */
void mmapcopy(int fd, int size)
{
    /* Ptr to mem-mapped VM area */
    char *bufp;

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

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

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

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

Copying without transferring data to user space.
Virtual Memory of a Linux Process

- **Kernel virtual memory**
  - Process-specific data structs (ptables, task and mm structs, kernel stack)
  - Physical memory
  - Kernel code and data
  - User stack
  - Memory mapped region for shared libraries
  - Runtime heap (malloc)
  - Uninitialized data (.bss)
  - Initialized data (.data)
  - Program text (.text)

- **Process virtual memory**

- **Different for each process**
  - 0x08048000 (32)
  - 0x00400000 (64)

- **Identical for each process**
  - %esp
  - brk
Linux Organizes VM as Collection of “Areas”

- **pgd:**
  - Page global directory address
  - Points to L1 page table

- **vm_prot:**
  - Read/write permissions for this area

- **vm_flags**
  - Pages shared with other processes or **private** to this process
Linux Page Fault Handling

Segmentation fault: accessing a non-existing page

Normal page fault

Protection exception: e.g., violating permission by writing to a read-only page (Linux reports as Segmentation fault)
The execve Function Revisited

To load and run a new program `a.out` in the current process using `execve`:

- Free `vm_area_struct`'s and page tables for old areas
- Create `vm_area_struct`'s and page tables for new areas
  - Programs and initialized data backed by object files.
  - `.bss` and stack backed by anonymous files.
- Set PC to entry point in `.text`
  - Linux will fault in code and data pages as needed.