Virtual Memory: Systems Level

Andrew Case

Slides adapted from jinyang Li, Randy Bryant and Dave O’Hallaron
Topics

- Memory system example
- Case study: Core i7 memory system
- Memory mapping in Linux
Memory System Example

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

![Diagram](image)
## Memory System: Page Table

First 16 entries (out of 256) shown

<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

Works the same way as our Caching systems (based on similar hardware caching)

<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

CT – Cache Tag
CI – Cache Index
CO – Cache Offset

<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 (based on cache state)

results based on TLB/cache states

VPN 0x0F  TLBI 0x3  TLBT 0x03  TLB Hit? Y  Page Fault? N  PPN: 0x0D

Byte: 0x36
Address Translation Example #2

Virtual Address: 0x0B8F

- TLBT
- TLBI

<table>
<thead>
<tr>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</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>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

- VPN
- VPO

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

Physical Address

- CT
- CI
- CO

<table>
<thead>
<tr>
<th>11</th>
<th>10</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>
</table>

- PPN
- PPO

CO ___ CI ___ CT ___ Hit? __ Byte: ___

Results based on TLB/cache states
Address Translation Example #3

Virtual Address: 0x0020

Physical Address

Results based on TLB/cache states
Topics

- Simple memory system example
- Case study: Core i7/Linux memory system
- Memory mapping
Intel Core i7 Memory System

Processor package

Core x4

- **Registers**
- **Instruction fetch**
- **MMU** (addr translation)

- **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

- **L3 unified cache**
  - 8 MB, 16-way
  - (shared by all cores)

- **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)**
  - PDBR: Page directory base registers
  - TLB: Translation lookaside buffer
  - 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
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>Unused</td>
<td>Page table PA</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</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Each entry references a 4K child page table

**P:** child page table present in physical memory or not

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

**U/S:** User or supervisor (kernel) mode access

**WT:** Write-through or write-back cache policy

**CD:** Caching disabled or enabled

**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

| 63 | 62 | 52 | 51 | 12 | 11 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|---|
| Unused | PPN | Unused | G | D | A | CD | WT | U/S | R/W | P |

Each entry references a 4K page

**P:** page is present in memory or not

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

**U/S:** User or supervisor (kernel) mode access

**WT:** Write-through or write-back cache policy

**CD:** Caching disabled or enabled

**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

- Level 1 Page Table (L1 PT)
- Level 2 Page Table (L2 PT)
- Level 3 Page Table (L3 PT)
- Level 4 Page Table (L4 PT)

Physical address of L1 PT Stored in CR3 register

512 GB region per entry
1 GB region per entry
2 MB region per entry
4 KB region per entry

Physical address
Offset into physical and virtual page
Virtual address

PDBR

Physical address

VPN 1
VPN 2
VPN 3
VPN 4
VPO

L1 PTE
L2 PTE
L3 PTE
L4 PTE

VPN 1
VPN 2
VPN 3
VPN 4

40
40
40
40

Physical address

PPN

12

12

40

15

40

40

12

12

512 GB region per entry
1 GB region per entry
2 MB region per entry
4 KB region per entry
End-to-end Core i7 Address Translation

Memory Request

Virtual address (VA)

VPN
VPO

36
12

L1 TLB

32/64

Result

L1 cache

L2, L3, and main memory

Physical address (PA)

PDBR (stored in CR3 register)

PTE
Page tables

VPN1 VPN2 VPN3 VPN4

9  9  9  9

PPN PPO

40
12

TLB hit

TLB miss

L1 hit

L1 miss
End-to-end Core i7 Address Translation

Memory Request

Virtual address (VA)

VPN  VPO

36  12

TLBT  TLBI

L1 TLB
(16 sets, 4 entries/set)

L1 hit

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

TLB miss

VPN1  VPN2  VPN3  VPN4

9  9  9  9

PDBR (stored in CR3 register)

Page tables

PTE

L2, L3, and main memory

CT  CI  CO

Physical address (PA)

32/64

Result

TLB hit

Result

L1 hit

L1 miss

PPN  PPO

40  12

9  9  9

PTE

PTE

PTE

PTE

PTE

32

12

Physical address (PA)
Virtual Memory of a Linux Process

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

Different for each process

Identical for each process

Kernel virtual memory

Process virtual memory
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

Process virtual memory

1. read

Segmentation fault: accessing a non-existing page

2. write

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

3. read

Normal page fault

vm_area_struct

vm_end
vm_start
vm_prot
vm_flags
vm_next

shared libraries

data

text

vm_end
vm_start
vm_prot
vm_flags
vm_next

vm_end
vm_start
vm_prot
vm_flags
vm_next

vm_end
vm_start
vm_prot
vm_flags
vm_next
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
  - *Nothing*
    - First fault will allocate a physical page full of 0's (*demand-zero page*)

- If a dirty page is evicted, OS copies it to a special *swap* area on disk.
Demand paging

- **Key point:** OS delays copying virtual pages into physical memory until they are referenced!

- Crucial for time and space efficiency
Shared objects under demand-paging

- Process 1 maps the shared object.
Shared objects under demand-paging

- Process 2 maps the shared object.
- Note: the same object can be mapped to different virtual addresses.
Sharing: 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
Sharing: 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

- **Clones the current process**
- **Creates virtual address for new process**
  - An exact copy of parent’s memory mapping for the child
  - Flag each memory area in both processes at COW and set each page in both processes as read-only
    - Each process starts creating it’s own data independently
- **Subsequent writes create new pages using COW mechanism.**
The exec Function

To load and run a new program from the current process using `exec`:

- Free old mapped areas and page tables
- Create new mapped areas and corresponding page table entries
- Set PC to entry point in `.text`
- Subsequently, OS will page fault in code and load data pages as needed
User-Level Memory Mapping

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

- **Good for:**
  - Process to process memory sharing (shared RO data, or IPC)
  - Reading/writing to large files non-sequentially (if you need to move around in the file)

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

- **Returns a pointer to start of mapped area (may not map to addr depending on if the kernel allowed that)**
User-Level Memory Mapping

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

- **len** bytes
- **start** (or address chosen by kernel)
- **offset** (bytes)

Diagram:
- Disk file specified by file descriptor `fd`
- Process virtual memory
Using `mmap` to Copy Files

- No extra copy has to be created in user space (kernel caching)
- Doesn’t have to have a system call to process data

```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);
}
```