Virtual Memory Mapping

Computer Systems Principles

Note: these slides are based on those provided by Randal E. Bryant and David R. O’Hallaron and used in their course at CMU.
Today

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

```
13 12 11 10 9 8 7 6 5 4 3 2 1 0
```

![Diagram](image.png)

- **VPN** Virtual Page Number
- **VPO** Virtual Page Offset
- **PPN** Physical Page Number
- **PPO** Physical Page Offset
1. 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>
### 2. 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>
### PAGE TABLE AND TLB

![Page Table and TLB Diagram]

<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
<th>Valid</th>
<th>VPN</th>
<th>PPN</th>
<th>Valid</th>
<th>VPN</th>
<th>PPN</th>
<th>Valid</th>
<th>VPN</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>28</td>
<td>1</td>
<td>04</td>
<td>-</td>
<td>0</td>
<td>08</td>
<td>13</td>
<td>1</td>
<td>0C</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>-</td>
<td>0</td>
<td>05</td>
<td>16</td>
<td>1</td>
<td>09</td>
<td>17</td>
<td>1</td>
<td>0D</td>
<td>2D</td>
<td>1</td>
</tr>
<tr>
<td>02</td>
<td>33</td>
<td>1</td>
<td>06</td>
<td>-</td>
<td>0</td>
<td>0A</td>
<td>09</td>
<td>1</td>
<td>0E</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>03</td>
<td>02</td>
<td>1</td>
<td>07</td>
<td>-</td>
<td>0</td>
<td>0B</td>
<td>-</td>
<td>0</td>
<td>0F</td>
<td>0D</td>
<td>1</td>
</tr>
</tbody>
</table>

### TLB

<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>28</td>
<td>1</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>
Address Translation Exercise #1

Virtual Address: 0x03D4

VPN 0x0F  TLBI 0x3  TLBT 0x03  TLB Hit? Y  Page Fault? N  PPN: 0x0D
Address Translation Exercise #2

Virtual Address: 0x0020

<table>
<thead>
<tr>
<th>VPN</th>
<th>TLBI</th>
<th>TLBT</th>
<th>VPO</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

VPN 0x00 TLBI 0 TLBT 0x00 TLB Hit? N Page Fault? N PPN: 0x28
Today

- Simple memory system example
- Memory mapping
- Case study: Core i7/Linux memory system
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
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 (and private COW)

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

- Subsequent writes create new pages using COW mechanism.
The `execve` Function Revisited

- To load and run a new program `a.out` in the current process using `execve`:
  - Free page tables for old areas
  - Create 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.

Diagram:

- **User stack**: Private, demand-zero
- **Memory mapped region for shared libraries**: Shared, file-backed
- **Runtime heap (via malloc)**: Private, demand-zero
- **Uninitialized data (.bss)**: Private, demand-zero
- **Initialized data (.data)**: Private, file-backed
- **Program text (.text)**: Private, file-backed
- **libc.so**
  - `.data`
  - `.text`
- **a.out**
  - `.data`
  - `.text`
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

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

- **len** bytes
- **offset** (bytes)
- **Disk file specified by file descriptor fd**
- **Process virtual memory**
- **start** (or address chosen by kernel)
Example: Using `mmap` to Copy Files

- Copying a file to `stdout` without transferring data to user space.

```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;
}

/* 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>\n", 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);
}
```

`mmapcopy.c`
User-Level Memory Mapping

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

Process 1
virtual memory

Disk file specified by file descriptor fd

Process 2
virtual memory

len bytes

(start or address chosen by kernel)

offset (bytes)

len bytes
Today

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

Processor package

Core x4

Registers
  L1 d-cache
    32 KB, 8-way
  L1 i-cache
    32 KB, 8-way

Instruction fetch
  L1 d-cache
    32 KB, 8-way
  L1 i-cache
    32 KB, 8-way

MMU (addr translation)
  L1 d-TLB
    64 entries, 4-way
  L1 i-TLB
    128 entries, 4-way

L2 unified cache
  256 KB, 8-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)**
  - 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 hit

TLB miss

L1 TLB (16 sets, 4 entries/set)

VPN1 VPN2 VPN3 VPN4

PTE PTE PTE PTE

Page tables

CR3

VPN1 VPN2 VPN3 VPN4

PTE PTE PTE PTE

Page tables

32/64

Result

L2, L3, and main memory

L1 hit

L1 miss

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

Physical address (PA)

PPN PPO

CT CI CO

Physical address (PA)
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>

Each entry references a 4K child page table. Significant fields:

- **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.
- **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).

**Page table physical base address**: 40 most significant bits of physical page table address (forces page tables to be 4KB aligned)

**XD**: Disable or enable instruction fetches from all pages reachable from this PTE.
Core i7 Level 4 Page Table Entries

<table>
<thead>
<tr>
<th>XD</th>
<th>Unused</th>
<th>Page physical base address</th>
<th>Unused</th>
<th>G</th>
<th>D</th>
<th>A</th>
<th>CD</th>
<th>WT</th>
<th>U/S</th>
<th>R/W</th>
<th>P=1</th>
</tr>
</thead>
</table>

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

Each entry references a 4K child page. Significant fields:

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

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

**XD**: Disable or enable instruction fetches from this page.
Core i7 Page Table Translation

- **Virtual address**
- **Offset into physical and virtual page**
- **Physical address of page**

**CR3 Physical address of L1 PT**

**VPN 1**
- **L1 PT Page global directory**
- **L1 PTE**
  - 512 GB region per entry

**VPN 2**
- **L2 PT Page upper directory**
- **L2 PTE**
  - 1 GB region per entry

**VPN 3**
- **L3 PT Page middle directory**
- **L3 PTE**
  - 2 MB region per entry

**VPN 4**
- **L4 PT Page table**
- **L4 PTE**
  - 4 KB region per entry

**VPN 5**
- **VPO**

**Physical address**

**PPN**

**PPO**