Some slides adapted (and slightly modified) from:

• Clark Barrett
• Jinyang Li
• Randy Bryant
• Dave O’Hallaron
**Toy Memory System Example**

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

![Diagram showing virtual and physical page number and offset](image-url)
**Toy Memory System Page Table**

1-level page table: How many PTEs?

<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>
Address Translation Example

Virtual Address: 0x0354

What’s the corresponding PPN? Physical address?
Case study: Core i7/Linux memory system (Nehalem microarchitecture)
Intel Core i7 Memory System

Processor chip package

One core (4 total)

- Registers
- Instruction fetch
- MMU (addr translation)
- L1 d-cache 32 KB
- L1 i-cache 32 KB
- L2 unified cache 256 KB
- L1 d-TLB 64 entries
- L1 i-TLB 128 entries
- L2 unified TLB 512 entries
- 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
i7 Memory Hierarchy

- 48-bit virtual address
- 52-bit physical address
- TLBs are virtually addressed
- Caches are physically addressed
- Page size can be configured at start-up time as either 4KB or 4MB
  - Linux uses 4KB
- i7 uses 4-level page table hierarchy
- Each process has its own private page table hierarchy
Core i7 Page Table Translation

CR3

Physical address of L1 PT

VPN 1 9

L1 PT
Page global directory

VPN 2 9

L2 PT
Page upper directory

VPN 3 9

L3 PT
Page middle directory

VPN 4 9

L4 PT
Page table

VPO 12

VPN 4

Offset into physical and virtual page

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

Physical address of page

PPN 40

Physical address

PPO 12

Virtual address

Region

12 512 GB region per entry

9 1 GB region per entry

9 2 MB region per entry

9 4 KB region per entry

40
Core i7 Page Table Entry (level-4)

- **Unused**
- **PPN**
- **Unused**
- **D** (Dirty bit)
- **A**
- **U/S**
- **R/W**
- **P** (High-order bit)

- **Dirty bit** (set by MMU on writes, cleared by OS)
- **Reference bit** (set by MMU on reads and writes, cleared by OS)
- **User or supervisor mode access**
- **Read-only or read-write permission**
- **Page in memory or not**
End-to-end Core i7 Address Translation

Virtual address (VA)

CPU

VPN
VPO

VPN1 VPN2 VPN3 VPN4

VPN1
PTE

VPN2
PTE

VPN3
PTE

VPN4
PTE

Page tables

L1 TLB

TLB hit

L1 cache

L1 miss

L2, L3, and main memory

32/64

Result

Physical address (PA)

Page tables

PPN

PPO

L1 TLB

CR3

PTE

PTE

PTE

PTE

VPN

VPN

VPN

VPN
Memory mapping in Linux
Virtual Memory of a Linux Process

- **Kernel code and data**
- **Memory mapped region for shared libraries**
- **Runtime heap** (malloc)
- **Uninitialized data (.bss)**
- **Initialized data (.data)**
- **Program text (.text)**

**Process specific data**
- Identical for each process:
  - Process-specific data structs (ptables, task and mm structs, kernel stack)

**Kernel virtual memory**
- Different for each process:
  - Process virtual memory
Linux Organizes VM as Collection of “Areas”

- **pgd**: Page global directory address
  - Points to 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

- **vm_area_struct**
  - vm_end
  - vm_start
  - vm_prot
  - vm_flags

- **Process virtual memory**
  - shared libraries
  - data
  - text

1. Read
2. Write
3. Read

**Segmentation fault:**
- accessing a non-existing page

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

**Normal page fault**
Memory Mapping

• VM areas initialized by associating them with disk objects.

• 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 kicked out from memory, OS copies it to a special **swap area** on disk
Demand paging

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

• Crucial for time and space efficiency
Sharing under demand-paging

- Process 1 maps the shared object.
Sharing under demand-paging

- Process 2 maps the shared object.
- Notice 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!
fork

• To create virtual address for new child process
  – Create 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

• Subsequent writes create new pages using COW mechanism.
To load and run a new program `a.out` in the current process using `execve`:

- 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 fault in code and data pages as needed.
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
- `start` (or address chosen by kernel)
- `offset` (bytes)

Disk file specified by file descriptor `fd`

Process virtual memory
Conclusions

• In this lecture we have seen VM in action.

• It is important to know how the following pieces interact:
  – Processor
  – MMU
  – DRAM
  – Cache
  – Kernel