CSCI-UA.0201-003

Computer Systems Organization

Lecture 18: Virtual Memory: Systems

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com

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

![Addressing Diagram]

- **Virtual Page Number (VPN)**
- **Virtual Page Offset (VPO)**
- **Physical Page Number (PPN)**
- **Physical Page Offset (PPO)**
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

<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>

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
- 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
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

VPO
Virtual address

Offset into physical and virtual page

PPN
Physical address

PPO
Physical address

VPN 1
VPN 2
VPN 3
VPN 4

40

12

12

512 GB region per entry
1 GB region per entry
2 MB region per entry
4 KB region per entry
Core i7 Page Table Entry (level-4)

| 63 | 62 | 52 | 51 | 12 | 11 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|---|
|    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| Unused | PPN | Unused |    |    |    |    |    |    |    |    |    |    |    |    |    |    |

- **PPN**: Physical page number
- **Unused**: Unused bits
- **D**: Dirty bit (set by MMU on writes, cleared by OS)
- **A**: Access bit (set by MMU on reads and writes, cleared by OS)
- **U/S**: User or supervisor mode access
- **R/W**: Read-only or read-write permission
- **P**: Page in memory or not
End-to-end Core i7 Address Translation

Virtual address (VA) → CPU

- VPN
- VPO

VPN → VPN1, VPN2, VPN3, VPN4

- CR3
- PTE
- Page tables

TLB miss → L1 TLB

- TLB hit

L1 TLB → L1 cache

- L1 hit
- L1 miss

L1 cache → Result

- 32/64

Result → L2, L3, and main memory

- Physical address (PA)
- PPN
- PPO
Memory mapping in Linux
Virtual Memory of a Linux Process

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

**Kernel virtual memory**

**Process virtual memory**

Identical for each process

Different for each process

Process virtual memory

0x08048000 (32)
0x00400000 (64)
**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

---

**task_struct**

**mm_struct**

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

**vm_area_struct**

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

---

**Process virtual memory**

- **Shared libraries**
- **Data**
- **Text**

0
Linux Page Fault Handling

Process virtual memory

- shared libraries
- data
- text

vm_area_struct

- vm_end
- vm_start
- vm_prot
- vm_flags

vm_next

vm_end
vm_start
vm_prot
vm_flags

1. read
2. write
3. read

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)
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.

Diagram:
- Process 1 virtual memory
- Physical memory
- Process 2 virtual memory
- Private copy-on-write area
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!
Revisiting 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.
Revisiting `execve`

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.
User-Level Memory Mapping

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)
```

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

Disk file specified by file descriptor `fd`

Process virtual memory
Using *mmap* to Copy Files

- Copying without transferring data to user space.

```c
/*
 * 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;
}
```

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