Virtual Memory: Systems

Kai Shen

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

Simple Memory System Example

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

<table>
<thead>
<tr>
<th>Virtual Page Number</th>
<th>Virtual Page Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Physical Page Number</th>
<th>Physical Page Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Simple Memory System Page Table

Only show first 16 entries (out of 256)

Virtual Address: 0x03D4

Physical Address

<table>
<thead>
<tr>
<th>Virtual Page Number</th>
<th>Virtual Page Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Physical Page Number</th>
<th>Physical Page Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Outline

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

Intel Core i7 Memory System

CPU

L2, L3, and main memory

L2/TLB

L1 d-cache

L1 i-cache

Registers

Instruction fetch

MMU

(add'nt translation)

L1 d-TLB

L1 i-TLB

L2 unified cache

256 KB, 8-way

L2 unified TLB

512 entries, 4-way

QuickPath interconnect

4 links @ 25.6 GB/sec

To other cores

To I/O bridge

L3 unified cache

8 MB, 16-way

(shared by all cores)

DDR3 Memory controller

3x64 bit @ 10.66 GB/s

32 GB/s total

(shared by all cores)

Main memory

Core i7 Level 1-3 Page Table Entries

Each entry references a 4KB child page table

P: Child page table present in physical memory (1) or not (0).
R/W: Read-only or read-write 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

Each entry references a 4KB 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
- 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)

Address Translation and Memory Access
- Address translation must proceed memory access. Why?
  - Can we parallelize?
    - Specifically, parallelize address translation with the first step of memory access.
    - What is the first step of memory access?
- A helpful address layout

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
Virtual Memory of a Linux Process

- Different for each process:
  - Process-specific data structs (ptables, task and mm structs, kernel stack)
  - Physical memory
  - Kernel code and data

- Identical for each process:
  - User stack
  - Memory mapped region for shared libraries
  - Runtime heap (malloc)
  - Uninitialized data (.bss)
  - Program text (.text)

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)

Outline

- Simple memory system example
- Case study: Core i7/Linux memory system
- 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
    - Can be **dirtied** later.
  - **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 to the disk lazily.

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

Sharing Revisited: Shared Objects

- Process 1 maps the shared object.
- Process 2 maps the shared object.
- Shared objects are typically read only.
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

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 memory area structures and page tables.
  - Flag each page in both processes as read-only
  - Flag each memory area in both processes as private COW
- On return, each process has exact copy of virtual memory.
- Subsequent writes create new pages using COW mechanism.

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
User-Level Memory Mapping

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

Using mmap to Read File

- mmap → lazy file read to buffer.
- Fast start;
- Real read is only performed on-demand.

Disclaimer

These slides were adapted from the CMU course slides provided along with the textbook of “Computer Systems: A programmer’s Perspective” by Bryant and O’Hallaron. The slides are intended for the sole purpose of teaching the computer organization course at the University of Rochester.