Virtual Memory: Systems

15-213 / 18-213: Introduction to Computer Systems
17th Lecture, Mar. 22, 2012

Instructors:
Todd C. Mowry & Anthony Rowe

Today

- Virtual memory questions and answers
- Simple memory system example
- Bonus: Case study: Core i7/Linux memory system
- Bonus: Memory mapping

Virtual memory reminder/review

- Programmer’s view of virtual memory
  - Each process has its own private linear address space
  - Cannot be corrupted by other processes

- System view of virtual memory
  - Uses memory efficiently by caching virtual memory pages
    - Efficient only because of locality
  - Simplifies memory management and programming
  - Simplifies protection by providing a convenient interpositioning point to check permissions

Recall: Address Translation With a Page Table
Recall: Address Translation: Page Hit

1) Processor sends virtual address to MMU
2-3) MMU fetches PTE from page table in memory
4) MMU sends physical address to cache/memory
5) Cache/memory sends data word to processor

Question #1

- Are the PTEs cached like other memory accesses?
  - Yes (and no: see next question)

Question #2

- Isn’t it slow to have to go to memory twice every time?
  - Yes, it would be… so, real MMUs don’t
Speeding up Translation with a TLB

- Page table entries (PTEs) are cached in L1 like any other memory word
  - PTEs may be evicted by other data references
  - PTE hit still requires a small L1 delay
- Solution: Translation Lookaside Buffer (TLB)
  - Small, dedicated, super-fast hardware cache of PTEs in MMU
  - Contains complete page table entries for small number of pages

TLB Hit

A TLB hit eliminates a memory access

Question #3

- Isn’t the page table huge? How can it be stored in RAM?
- Yes, it would be... so, real page tables aren’t simple arrays

A TLB miss incurs an additional memory access (the PTE)

Fortunately, TLB misses are rare. Why?
Multi-Level Page Tables

- **Suppose:**
  - 4KB \(2^{12}\) page size, 64-bit address space, 8-byte PTE

- **Problem:**
  - Would need a 32,000 TB page table!
    - \(2^{64} \times 2^{12} \times 2^3 = 2^{55}\) bytes

- **Common solution:**
  - Multi-level page tables
  - Example: 2-level page table
    - Level 1 table: each PTE points to a page table (always memory resident)
    - Level 2 table: each PTE points to a page (paged in and out like any other data)

Translating with a k-level Page Table

A Two-Level Page Table Hierarchy

Question #4

- Shouldn’t fork() be really slow, since the child needs a copy of the parent’s address space?

- Yes, it would be... so, fork() doesn’t really work that way
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

- `fork` provides private address space for each process.
- To create virtual address for new process:
  - Create exact copies of parent page tables.
  - Flag each page in both processes (parent and child) as read-only.
  - Flag writeable areas in both processes as private COW.
- On return, each process has exact copy of virtual memory.
- Subsequent writes create new physical pages using COW mechanism.
- Perfect approach for common case of `fork()` followed by `exec()`.
  - Why?

Today

- Virtual memory questions and answers.
- Simple memory system example.
- Bonus: Case study: Core i7/Linux memory system.
- Bonus: Memory mapping.

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):
  - VPO: Virtual page offset.
  - VPN: Virtual page number.
  - TLBI: TLB index.
  - TLBT: TLB tag.
- 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.
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>

Simple Memory System TLB

- 16 entries
- 4-way associative

<table>
<thead>
<tr>
<th>Set</th>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>03</td>
<td>0</td>
<td>09</td>
</tr>
<tr>
<td>01</td>
<td>2D</td>
<td>1</td>
<td>02</td>
</tr>
<tr>
<td>02</td>
<td>0E</td>
<td>0</td>
<td>06</td>
</tr>
<tr>
<td>03</td>
<td>0D</td>
<td>1</td>
<td>0A</td>
</tr>
</tbody>
</table>

Simple Memory System Cache

- 16 lines, 4-byte block size
- Physically addressed
- Direct mapped

Address Translation Example #1

Virtual Address: 0x03D4

Physical Address
Today

- Virtual memory questions and answers
- Simple memory system example
- Bonus: Case study: Core i7/Linux memory system
- Bonus: Memory mapping
### Review of Symbols

- **Basic Parameters**
  - \( N = 2^x \): Number of addresses in virtual address space
  - \( M = 2^y \): Number of addresses in physical address space
  - \( P = 2^z \): 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

### Core i7 Level 1-3 Page Table Entries

<table>
<thead>
<tr>
<th>XD</th>
<th>Unused</th>
<th>Page table physical base address</th>
<th>Unused</th>
<th>G</th>
<th>PS</th>
<th>A</th>
<th>CD</th>
<th>WT</th>
<th>U/S</th>
<th>R/W</th>
<th>P=x</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>62</td>
<td>52</td>
<td>51</td>
<td>12</td>
<td>11</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

Each entry references a 4K child page table

- 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 mode access
- 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

<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=x</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>62</td>
<td>52</td>
<td>51</td>
<td>12</td>
<td>11</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

Each entry references a 4K 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
- WT: Write-through or write-back cache policy for this page
- 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)

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

- Virtual memory questions and answers
- Simple memory system example
- Bonus: Case study: Core i7/Linux memory system
- Bonus: 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
  - 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.

Cute 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
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

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

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

- len bytes starting at (or address chosen by kernel)
- 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`)

Using `mmap` to Copy Files

- Copying without transferring data to user space.

```c
#include "csapp.h"

/* mmapcopy - uses mmap to copy */
void mmapcopy(int fd, int size)
{
  char *bufp;
  bufp = Mmap(NULL, size, PROT_READ, MAP_PRIVATE, fd, 0);
  Write(1, bufp, size);
  return;
}
```
Virtual Memory of a Linux Process

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

Physical memory

- %esp
- brk

Different for each process

Identical for each process

Kernel virtual memory

Process virtual memory

Virtual Memory of a Linux Process

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

Process virtual memory

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)

The execve Function Revisited

- To load and run a new program `a.out` in the current process using `execve`

  - Free vm_area_struct’s and page tables for old areas
  - Create vm_area_struct’s and 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.