Virtual Memory: Systems

CSci 2021: Machine Architecture and Organization
November 30th-December 3rd, 2018

Your instructor: Stephen McCamant

Based on slides originally by:
Randy Bryant, Dave O’Hallaron

Today

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

1. Simple Memory System TLB

- 16 entries
- 4-way associative

2. Simple Memory System Page Table

Only show first 16 entries (out of 256)
3. Simple Memory System Cache

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

Components of the physical address (PA) (shared by all cores)

- Page Fault?
- PPN

Memory mapping

- TLB
- L1 i-cache
- L2 unified cache
- DDR3 Memory controller
- Main memory

Intel Core i7 Memory System

- Processor package
- Core x4

Address Translation Example #1

Virtual Address: 0x03D4

Physical Address

Address Translation Example #2

Virtual Address: 0x0020

Physical Address

Today

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

Review of Symbols

- Basic Parameters
  - N = 2^m: Number of addresses in virtual address space
  - M = 2^p: Number of addresses in physical address space
  - P = 2^m: Page size (bytes)
- Components of the virtual address (VA)
  - TLBI: TLB index
  - TLTB: 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

Core i7 Level 4 Page Table Entries

Cute Trick for Speeding Up L1 Access

Virtual Address Space 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
```

Linux Page Fault Handling

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

Today

- 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
  - 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** (or partition).

Sharing Revisited: Shared Objects

- Process 1 maps the shared object.
  - Process 1 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

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!

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`)

Disk file specified by file descriptor `fd`
Process virtual memory

start

(len bytes)

(`or address chosen by kernel`)