Virtual Memory: Systems

Kai Shen

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)

Virtual Address: 0x03D4

Intel Core i7 Memory System

Registers
Instruction fetch
MMU (addr translation)
L1 d-cache 32 KB, 8-way
L2 unified cache 256 KB, 8-way
L3 unified cache 8 MB, 16-way (shared by all cores)
L1 i-cache 32 KB, 8-way
L2 unified cache 512 entries, 4-way
L1 i-TLB 64 entries, 4-way
L1 d-TLB 128 entries, 4-way
L1 d-TLB 512 entries, 4-way
QuickPath interconnect 4 links @ 25.6 GB/s each
DDR3 Memory controller 3 x 64 bit @ 10.66 GB/s
Main memory

To other cores
To I/O bridge
End-to-end Core i7 Address Translation

Core i7 Page Table Translation

Virtual Memory of a Linux Process

- 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

Trick for Speeding Up L1 Access
Page Fault Handling

- Process virtual memory
  - shared libraries
    - read
  - data
    - read
  - text
    - write

  **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)

Demand paging

- **Key point:** instantiate the physical existence of virtual pages only when necessary!
  - Known as **demand paging**
  - Enabled by virtual memory

  - Crucial for space and time efficiency

Program Loading

- Initial execution of a program
  - Create page table mapping for the program text and data, but not necessarily load them right away
  - Page table entries for unloaded text/data areas are invalid; memory is loaded on-demand

  - It saves space since parts of the program image may never be needed
  - It also reduces to startup delay

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` maps file to buffer.
  - Lazy file read to buffer.
- Fast start;
- Real read is only performed on-demand.

Sharing Revisited: Shared Objects

- Process 1 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
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

- 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.

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.