Virtual Memory: Systems

15-213 / 18-213: Introduction to Computer Systems
17th Lecture, Oct. 23, 2014

Instructors:
Greg Ganger, Greg Kesden, Dave O’Hallaron
Today

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

Virtual address

Virtual page number (VPN)  Virtual page offset (VPO)

Page table base register (PTBR)

Page table address for process

Valid bit = 0: page not in memory (page fault)

Physical address

Physical page number (PPN)  Physical page offset (PPO)
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)
Page tables in memory, like other data

VA: virtual address, PA: physical address, PTE: page table entry, PTEA = PTE address
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
A TLB miss incurs an additional memory access (the PTE)
Fortunately, TLB misses are rare. Why?
Question #3

- Aren’t the TLB contents wrong after a context switch?

- Yes, they would be, so something must be done..
  - Option 1: flush TLB on context switch
  - Option 2: associate a process ID with each TLB entry
Question #4

- 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
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)
A Two-Level Page Table Hierarchy

Level 1 page table
- PTE 0
- PTE 1
- PTE 2 (null)
- PTE 3 (null)
- PTE 4 (null)
- PTE 5 (null)
- PTE 6 (null)
- PTE 7 (null)
- PTE 8
- (1K - 9) null PTEs

Level 2 page tables
- PTE 0
- ... (null)
- PTE 1023

Virtual memory
- VP 0
- ... (null)
- VP 1023
- VP 1024
- ... (null)
- VP 2047
- Gap
- 1023 null PTEs
- PTE 1023
- 1023 unallocated pages
- VP 9215

32 bit addresses, 4KB pages, 4-byte PTEs

2K allocated VM pages for code and data

6K unallocated VM pages

1023 unallocated pages

1 allocated VM page for the stack
Translating with a k-level Page Table

![Diagram of page table translation with virtual and physical addresses.]
Question #5

- 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
Physical memory can be shared

- Process 1 maps the shared pages
Physical memory can be shared

- Process 2 maps the shared pages
- Notice how the virtual addresses can be different
Private Copy-on-write (COW) sharing

- Two processes mapping *private copy-on-write (COW)* pages
- Area flagged as private copy-on-write
- PTEs in private areas are flagged as read-only
Private Copy-on-write (COW) sharing

- 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:** Memory mapping
- **Bonus:** Case study: Core i7/Linux memory system
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>

<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>
Simple Memory System TLB

- 16 entries
- 4-way associative

<table>
<thead>
<tr>
<th>Set</th>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
<th>Tag</th>
<th>PPN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>03</td>
<td>–</td>
<td>0</td>
<td>09</td>
<td>0D</td>
<td>1</td>
<td>00</td>
<td>–</td>
<td>0</td>
<td>07</td>
<td>02</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>03</td>
<td>2D</td>
<td>1</td>
<td>02</td>
<td>–</td>
<td>0</td>
<td>04</td>
<td>–</td>
<td>0</td>
<td>0A</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>–</td>
<td>0</td>
<td>08</td>
<td>–</td>
<td>0</td>
<td>06</td>
<td>–</td>
<td>0</td>
<td>03</td>
<td>–</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>07</td>
<td>–</td>
<td>0</td>
<td>03</td>
<td>0D</td>
<td>1</td>
<td>0A</td>
<td>34</td>
<td>1</td>
<td>02</td>
<td>–</td>
<td>0</td>
</tr>
</tbody>
</table>
Simple Memory System Cache

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

### Table

<table>
<thead>
<tr>
<th>Idx</th>
<th>Tag</th>
<th>Valid</th>
<th>B0</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>19</td>
<td>1</td>
<td>99</td>
<td>11</td>
<td>23</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>15</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>2</td>
<td>18</td>
<td>1</td>
<td>00</td>
<td>02</td>
<td>04</td>
<td>08</td>
</tr>
<tr>
<td>3</td>
<td>36</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>1</td>
<td>43</td>
<td>6D</td>
<td>8F</td>
<td>09</td>
</tr>
<tr>
<td>5</td>
<td>0D</td>
<td>1</td>
<td>36</td>
<td>72</td>
<td>F0</td>
<td>1D</td>
</tr>
<tr>
<td>6</td>
<td>31</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>7</td>
<td>16</td>
<td>1</td>
<td>11</td>
<td>C2</td>
<td>DF</td>
<td>03</td>
</tr>
</tbody>
</table>

### Table (continued)

<table>
<thead>
<tr>
<th>Idx</th>
<th>Tag</th>
<th>Valid</th>
<th>B0</th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>24</td>
<td>1</td>
<td>3A</td>
<td>00</td>
<td>51</td>
<td>89</td>
</tr>
<tr>
<td>9</td>
<td>2D</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>A</td>
<td>2D</td>
<td>1</td>
<td>93</td>
<td>15</td>
<td>DA</td>
<td>3B</td>
</tr>
<tr>
<td>B</td>
<td>0B</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>C</td>
<td>12</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>D</td>
<td>16</td>
<td>1</td>
<td>04</td>
<td>96</td>
<td>34</td>
<td>15</td>
</tr>
<tr>
<td>E</td>
<td>13</td>
<td>1</td>
<td>83</td>
<td>77</td>
<td>1B</td>
<td>D3</td>
</tr>
<tr>
<td>F</td>
<td>14</td>
<td>0</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>
Address Translation Example #1

Virtual Address: 0x03D4

VPN 0xF TLBI 0x3 TLBT 0x03 TLB Hit? Y Page Fault? N PPN: 0xD

Physical Address

CO 0 CI 0x5 CT 0x0D Hit? Y Byte: 0x36
Address Translation Example #2

Virtual Address: 0x0B8F

Physical Address

VPN 0x2E  TLBI 2  TLBT 0x0B  TLB Hit? N  Page Fault? Y  PPN: TBD

CO ___  CI ___  CT ___  Hit? ___  Byte: ___
Address Translation Example #3

Virtual Address: 0x0020

Physical Address

CO 0 CI 0x8 CT 0x28 Hit? N Byte: Mem
Today

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

```c
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 Copy Files

- Copying without transferring data to user space.

```c
#include "csapp.h"

/*
 * 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;
}

/* 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>
", 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);
}
```
Today

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

Processor package

Core x4

- Registers
- L1 d-cache 32 KB, 8-way
- L1 i-cache 32 KB, 8-way
- L2 unified cache 256 KB, 8-way
- L3 unified cache 8 MB, 16-way (shared by all cores)
- Instruction fetch
- L1 d-TLB 64 entries, 4-way
- L1 i-TLB 128 entries, 4-way
- MMU (addr translation)
- L2 unified 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 32 GB/s total (shared by all cores)
- Main memory

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

Virtual address (VA) -> CPU

VPN  VPO

TLBT  TLBI

TLB miss

L1 TLB (16 sets, 4 entries/set)

VPN1  VPN2  VPN3  VPN4

CR3

PTE

Page tables

TLB hit

L1 hit

L1 d-cache (64 sets, 8 lines/set)

PPN  PPO

Physical address (PA)

L1 miss

L2, L3, and main memory

32/64

Result

CT  CI  CO
# 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=1</th>
</tr>
</thead>
</table>

Available for OS (page table location on disk)  
P=0

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 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

<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=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>P=0</td>
<td></td>
<td>Available for OS (page location on disk)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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)

**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)
Core i7 Page Table Translation

CR3
Physical address of L1 PT

VPN 1
9
L1 PT
Page global directory

L1 PTE
512 GB region per entry

VPN 2
9
L2 PT
Page upper directory

L2 PTE
1 GB region per entry

VPN 3
9
L3 PT
Page middle directory

L3 PTE
2 MB region per entry

VPN 4
9
L4 PT
Page table

L4 PTE
4 KB region per entry

VPO
12
Virtual address

VPN

Offset into physical and virtual page

Physical address

VPN

Physical address

L1 PT

L2 PT

L3 PT

L4 PT

VPN 1

VPN 2

VPN 3

VPN 4

VPO

PPN

40

12

PPO

40

12

40

40

40

40

40
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
Virtual Memory of a Linux Process

Different for each process
- Process-specific data structs (ptables, task and mm structs, kernel stack)

Identical for each process
- Physical memory
- Kernel code and data
- User stack
- Memory mapped region for shared libraries
- Runtime heap (malloc)
- Uninitialized data (.bss)
- Initialized data (.data)
- Program text (.text)

Kernel virtual memory
- Process virtual memory

Physical memory
- brk
- %esp

Process-­‐specific data structs (ptables, task and mm structs, kernel stack)

Kernel virtual memory

Process virtual memory

0x00400000 (64)
0x08048000 (32)
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

```
task_struct
    mm
    mm_struct
      pgd
      mmap
    vm_area_struct
      vm_end
      vm_start
      vm_prot
      vm_flags

Process virtual memory
  Shared libraries
  Data
  Text
```
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.