Virtual Memory: Systems

CS351: Systems Programming
Day 20: Nov. 01, 2022

Instructor:
Nik Sultana

Slides adapted from Bryant and O’Hallaron
Next time: recorded lecture

| Oct 31 Lab | Nov 01 | LEC 20: Virtual Memory: Systems  
Preparation: Read CS:APP 9.7-9.8 | Nov 02 | Nov 03 | LEC 21: Storage Allocation: Basic  
Preparation: Read CS:APP 9.9 |
|------------|--------|---------------------------------|--------|--------|---------------------------------|
| Nov 07 Lab | Nov 06 | LEC 22: Storage Allocation: Advanced  
Preparation: Read CS:APP 9.9-9.11  
Preparation: Read CS:APP 11.1-11.4 |

- **LEC 20 and LEC 21 will be pre-recorded and circulated on Blackboard.**
  - Do not come to SB104 those days – there will not be an in-person lecture.
  - My away-at-a-conference days are marked on the course calendar.
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

![Diagram of memory system with virtual and physical page numbers and offsets.](image_url)
1. 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>

TLBT | TLBI | VPN | VPO
# 2. 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>
3. Simple Memory System Cache

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

```
<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>1B</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>
<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

- TLB Hit? __
- Page Fault? __
- PPN: ____

- TLBT __
- TLBI __

VPN ___

- VPN 0x0F
- TLBI 0x3
- TLBT 0x03
- TLB Hit? Y
- Page Fault? N
- PPN: 0x0D

Physical Address

- CT __
- CI __

CO ___

- CO 0
- CI 0x5
- CT 0x0D
- Hit? Y

Byte: 0x36
Address Translation Example #2

Virtual Address: 0x0020

Virtual Address:

\[\begin{array}{cccccccccccccc}
13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
\end{array}\]

VPN ___ TLBI ___ TLBT ____ TLB Hit? __ Page Fault? __ PPN: ____

VPN 0x00 TLBI 0 TLBT 0x00 TLB Hit? N Page Fault? N PPN: 0x28

Physical Address

Physical Address:

\[\begin{array}{cccccccccccccc}
11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
\end{array}\]

CT ____ Cl ____ CO ____

CO 0 CI 0x8 CT 0x28 Hit? N Byte: Mem
Address Translation Example #3

Virtual Address: 0x0020

Physical Address

VPN 0x00  TLBI 0  TLBT 0x00  TLB Hit? N  Page Fault? N  PPN: 0x28

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

- Simple memory system example
- Case study: Core i7/Linux memory system
- Memory mapping
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
- L1 d-TLB 64 entries, 4-way
- L1 i-TLB 128 entries, 4-way
- L2 unified TLB 512 entries, 4-way
- L3 unified cache 8 MB, 16-way (shared by all cores)
- DDR3 Memory controller 3 x 64 bit @ 10.66 GB/s
  32 GB/s total (shared by all cores)
- MMU (addr translation)
- Instruction fetch
- QuickPath interconnect 4 links @ 25.6 GB/s each

To other cores
To I/O bridge

Main memory
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

CPU

Virtual address (VA)

VPN \(\rightarrow\) VPO

TLBT \(\rightarrow\) TLBI

TLB miss

L1 TLB (16 sets, 4 entries/set)

VPN1, VPN2, VPN3, VPN4

TLB hit

L1 hit

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

Result

L1 miss

Page tables

Physical address (PA)

PTE

PTE

PTE

PTE

CR3

VPN

VPO

36

12

32

4

9

9

9

9

40

12

40

6

6

CT

CI

CO

Illinois Tech CS351 Fall 2022
Core i7 Level 1-3 Page Table Entries

| 63 | 62 | 52 | 51 | 12 | 11 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|---|
| XD | Unused | Page table physical base address | Unused | G | PS | A | CD | WT | U/S | R/W | P=1 |

Available for OS (page table location on disk)  

P=0

Each entry references a 4K child page table. Significant fields:

**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 (kernel) mode access permission for all reachable pages.

**WT**: Write-through or write-back cache policy 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).

**Page table physical base address**: 40 most significant bits of physical page table address (forces page tables to be 4KB aligned)

**XD**: Disable or enable instruction fetches from all pages reachable from this PTE.
Core i7 Level 4 Page Table Entries

<table>
<thead>
<tr>
<th>63</th>
<th>62</th>
<th>52</th>
<th>51</th>
<th>12</th>
<th>11</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>XD</td>
<td>Unused</td>
<td>Page physical base address</td>
<td>Unused</td>
<td>G</td>
<td>D</td>
<td>A</td>
<td>CD</td>
<td>WT</td>
<td>U/S</td>
<td>R/W</td>
<td>P=1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Each entry references a 4K child page. Significant fields:

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

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)

XD: Disable or enable instruction fetches from this page.
Core i7 Page Table Translation

[Diagram showing the page table hierarchy with virtual addresses on the left and physical addresses on the right, including CR3, PPN, and PPO.]
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 Address Space of a Linux Process

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

**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

**Process virtual memory**

**Kernel virtual memory**
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**

- **vm_area_struct**
  - `vm_end`
  - `vm_start`
  - `vm_prot`
  - `vm_flags`

- **Process virtual memory**
  - **vm_next**
  - `vm_end`
  - `vm_start`
  - `vm_prot`
  - `vm_flags`

- **VM Area**
  - **shared libraries**
  - **data**
  - **text**

**1. Segment fault:**
accessing a non-existing page

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

**3. Normal page fault**
write
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.
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

- 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 `mm_struct`, `vm_area_struct`, and page tables.
  - Flag each page in both processes as read-only
  - Flag each `vm_area_struct` in both processes as private COW

- On return, each process has exact copy of virtual memory

- Subsequent writes create new pages using COW mechanism.
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.
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)

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

len bytes

Disk file specified by file descriptor fd

Process virtual memory

len bytes

start

(or address chosen by kernel)

offset

(bytes)
Example: Using `mmap` to Copy Files

- Copying a file to `stdout` without transferring data to user space.

```c
#include "csapp.h"

void mmapcopy(int fd, int size) {
    char *bufp;
    bufp = Mmap(NULL, size,
                 PROT_READ,
                 MAP_PRIVATE,
                 fd, 0);
    Write(1, bufp, size);
    return;
}
```

```c
int main(int argc, char **argv) {
    struct stat stat;
    int fd;

    if (argc != 2) {
        printf("usage: %s <filename>\n", argv[0]);
        exit(0);
    }

    fd = Open(argv[1], O_RDONLY, 0);
    Fstat(fd, &stat);
    mmapcopy(fd, stat.st_size);
    exit(0);
}
```
Per-lecture feedback

- Better sooner rather than later!
- I can help with issues sooner.
- There is a per-lecture feedback form.
- The form is anonymous. (It checks that you’re at Illinois Tech to filter abuse, but I don’t see who submitted any of the forms.)
- [https://forms.gle/qoeEbBuTYXo5FiU1A](https://forms.gle/qoeEbBuTYXo5FiU1A)
- I’ll remind about this at each lecture.