Main Memory: Address Translation

(Chapter 12-17)

CS 4410
Operating Systems
Can’t We All Just Get Along?

Physical Reality: different processes/threads share the same hardware → need to multiplex

• CPU (temporal)
• Memory (spatial)
• Disk and devices (later)

Why worry about memory sharing?

• Complete working state of process and/or kernel is defined by its data (memory, registers, disk)
• Don’t want different processes to have access to each other’s memory (protection)
Aspects of Memory Multiplexing

**Isolation**

*Don’t want* distinct process states collided in physical memory (unintended overlap → chaos)

**Sharing**

*Want* option to overlap when desired (for efficiency and communication)

**Virtualization**

*Want* to create the illusion of more resources than exist in underlying physical system

**Utilization**

*Want* to best use of this limited resource
A Day in the Life of a Program

Compiler
(+ Assembler + Linker)

source files

sum.c

executable

sum

process

pid xxx

"It's alive!"

[Diagram showing the flow of compilation and execution]

```
#include <stdio.h>

int max = 10;

int main () {
    int i;
    int sum = 0;
    add(m, &sum);
    printf("%d", i);
    ...
}
```

[Code snippet showing the main function with variables and operations]

[Diagram showing memory layout with registers and segments]

[Binary and hexadecimal code representation]
Virtual view of process memory

0xffffffff

stack

heap
data
text

0x00000000
Paged Translation

Virtual Page N

Virtual Page 0

Process View

TERMINOLOGY ALERT:
Page: virtual
Frame: physical

No more external fragmentation!
Paging Overview

Divide:
- Physical memory into fixed-sized blocks called **frames**
- Virtual memory into blocks of same size called **pages**

Management:
- Keep track of which pages are mapped to which frames
- Keep track of all free frames

Notice:
- Not all pages may be mapped to frames
Address Translation, Conceptually

- Processor
- Virtual Address
- Translation
- Invalid → Raise Exception
- Valid
- Physical Address
- Physical Memory
- Data

Who does this?
Memory Management Unit (MMU)

- Hardware device
- Maps virtual to physical address (used to access data)

User Process:
- deals with *virtual* addresses
- Never sees the physical address

Physical Memory:
- deals with *physical* addresses
- Never sees the virtual address
High-Level Address Translation

Where is the red cube in physical memory?

- red cube is 255th byte in page 2.
Virtual Address Components

Page number – Upper bits
• Must be translated into a physical frame number

Page offset – Lower bits
• Does not change in translation

<table>
<thead>
<tr>
<th>page number</th>
<th>page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m - n$</td>
<td>$n$</td>
</tr>
</tbody>
</table>

For given logical address space $2^m$ and page size $2^n$
High-Level Address Translation

Who keeps track of the mapping?

Virtual Memory

- stack
- heap
- data
- text

Physical Memory

- STACK 0
- HEAP 0
- TEXT 1
- HEAP 1
- DATA 0
- TEXT 0
- STACK 1

Page Table

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>5...</td>
<td>5</td>
</tr>
</tbody>
</table>

0x0000 0x1000 0x2000 0x3000 0x4000 0x5000 0x6000 0x? ? ? ? ?
Simple Page Table

Lives in Memory

Page-table base register (PTBR)
  • Points to the page table
  • Saved/restored on context switch
Leveraging Paging

- Protection
- Demand Loading
- Copy-On-Write
Full Page Table

Meta Data about each frame
Protection R/W/X, Modified, Valid, etc.
MMU Enforces R/W/X protection
(illegal access throws a page fault)
Leveraging Paging

- Protection
- Demand Loading
- Copy-On-Write
Demand Loading

- Page not mapped until it is used
- Requires free frame allocation
  - What if there is no free frame???
- May involve reading page contents from disk or over the network
Leveraging Paging

- Protection
- Demand Loading
- Copy-On-Write (or fork() for free)
Copy on Write (COW)

- P1 forks()
- P2 created with
  - own page table
  - same translations
- All pages marked COW (in Page Table)
Option 1: fork, then keep executing

Now one process tries to write to the stack (for example):
• Page fault
• Allocate new frame
• Copy page
• Both pages no longer COW
Option 2: fork, then call exec

Before P2 calls `exec()`
Option 2: fork, then call `exec`

After P2 calls `exec()`

- Allocate new frames
- Load in new pages
- Pages no longer COW
Problems with Paging

Memory Consumption:
- **Internal Fragmentation**
  - Make pages smaller? But then…
- **Page Table Space**: consider 48-bit address space, 2KB page size, each PTE 8 bytes
  - How big is this page table?
    - Note: you need one for each process
  - How many pages in memory does it need?

Performance: every data/instruction access requires two memory accesses:
- One for the page table
- One for the data/instruction
Address Translation

- Paged Translation
- Efficient Address Translation
  - Multi-Level Page Tables
  - Inverted Page Tables
- TLBs
Multi-Level Page Tables to reduce page table space

+ Allocate only PTEs in use
+ Simple memory allocation
- more lookups per memory reference
Two-Level Paging Example

32-bit machine, 1KB page size

• Logical address is divided into:
  – a page offset of 10 bits \((1024 = 2^{10})\)
  – a page number of 22 bits \((32-10)\)

• Since the page table is paged, the page number is further divided into:
  – a 12-bit first index
  – a 10-bit second index

• Thus, a logical address is as follows:

<table>
<thead>
<tr>
<th>page number</th>
<th>page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>index 1</td>
<td>index 2</td>
</tr>
<tr>
<td>12</td>
<td>10</td>
</tr>
</tbody>
</table>
This one goes to three levels!

+ First Level requires less contiguous memory
- even more lookups per memory reference
Complete Page Table Entry (PTE)

<table>
<thead>
<tr>
<th>Valid</th>
<th>Protection R/W/X</th>
<th>Ref</th>
<th>Dirty</th>
<th>Index</th>
</tr>
</thead>
</table>

**Index** is an index into:
- frames
  - physical process memory or next level page table
- backing store
  - if page was swapped out

Synonyms:
- Valid bit == Present bit
- Dirty bit == Modified bit
- Referenced bit == Accessed bit
Address Translation

• Paged Translation
• Efficient Address Translation
  • Multi-Level Page Tables
  • Inverted Page Tables
• TLBs
So many virtual pages...

... comparatively few physical frames

Traditional Page Tables:
• map pages to frames
• are *numerous and sparse*
Inverted Page Table: Implementation

Implementation:
• 1 Page Table for entire system
• 1 entry per frame in memory
• Why don’t we store the frame #?

Not to scale! Page table << Memory
Inverted Page Table: Discussion

Tradeoffs:
↓ memory to store page tables
↑ time to search page tables

Solution: hashing
• hash(page,pid) \(\rightarrow\) PT entry (or chain of entries)
• What about:
  • collisions…
  • sharing…
Address Translation

• Paged Translation
• Efficient Address Translation
  • Multi-Level Page Tables
  • Inverted Page Tables
• TLBs
Translation Lookaside Buffer (TLB)

Associative cache of virtual to physical page translations
Address Translation with TLB

Access TLB before you access memory.

Processor

Virtual Address

TLB

Virtual Address

Miss

Page Table

Invalid

Raise Exception

Hit

Frame

Offset

Physical Memory

Data

Virtual Address

Valid

Frame

Physical Address

Data

Trick: access TLB \textbf{while} you access the cache.
Address Translation Uses!

Process isolation
  • Keep a process from touching anyone else’s memory, or the kernel’s

Efficient inter-process communication
  • Shared regions of memory between processes

Shared code segments
  • common libraries used by many different programs

Program initialization
  • Start running a program before it is entirely in memory

Dynamic memory allocation
  • Allocate and initialize stack/heap pages on demand
MORE Address Translation Uses!

Program debugging
• Data breakpoints when address is accessed

Memory mapped files
• Access file data using load/store instructions

Demand-paged virtual memory
• Illusion of near-infinite memory, backed by disk or memory on other machines

Checkpointing/restart
• Transparently save a copy of a process, without stopping the program while the save happens

Distributed shared memory
• Illusion of memory that is shared between machines