Main Memory: Address Translation
(Chapter 8)

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

Physical Reality: different processes/threads share the same hardware $\rightarrow$ 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 in memory (+ registers)
- Don’t want different threads 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 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
Address Translation

- Paged Translation
- Efficient Address Translation

All in the context of the OS
A Day in the Life of a Program

Compiler
(+ Assembler + Linker)

source files

sum.c

executable

sum

process

pid xxx

“It’s alive!”

#include <stdio.h>

int max = 10;

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

```
0040 0000
  .text
  main

0400 023C
  21035000
  1b80050c
  8C048004
  21047002
  0C400020

1000 0000
  .data
  max

4000 0000
  10201000
  21040330
  22500102
  ...

PC	GP	SP

0xffffffff

0x00000000

stack

heap

data

text

max

jal

addi

0x00000000

0x00000000

0x10000000

0xffffffff
What’s wrong with this... in the context of:
multiple processes?
multiple threads?

Logical view of process memory
Paged Translation

TERMINOLOGY ALERT:
Page: the data itself
Frame: physical location

No more external fragmentation!
Paging Overview

Divide:

- Physical memory into fixed-sized blocks called **frames**
- Logical memory into blocks of same size called **pages**

Management:

- Keep track of all free frames.
- To run a program with \( n \) pages, need to find \( n \) free frames and load program

Notice:

- Logical address space can be noncontiguous!
- Process given frames when/where available
Address Translation, Conceptually

Processor → Translation

Virtual Address → Translation

Valid → Physical Memory

Physical Address → Physical Memory

Data → Physical Memory

Physical Memory → Data

Invalid → Raise Exception

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
red cube is 255th byte in page 2.

Where is the red cube in physical memory?
Logical 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?

→ 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>
Simplified Page Table

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

Lives in Memory
Leveraging Paging

- Protection
- Dynamic Loading
- Dynamic Linking
- Copy-On-Write
Full Page Table

Meta Data about each frame
Protection R/W/X, Modified, Valid, etc.
Leveraging Paging

- Protection
- Dynamic Loading
- Dynamic Linking
- Copy-On-Write
Dynamic Loading & Linking

Dynamic Loading
- Routine is not loaded until it is called
- Better memory-space utilization; unused routine is never loaded
- No special support from the OS needed

Dynamic Linking
- Routine is not linked until execution time
- Locate (or load) library routine when called
- AKA **shared libraries** (e.g., DLLs)
Leveraging Paging

- Protection
- Dynamic Loading
- Dynamic Linking
- Copy-On-Write
Copy on Write (COW)

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

Option #1: Child keeps executing

Upon page fault:
- Allocate new frame
- Copy frame
- Both pages no longer COW
COW, then call exec (before)

Option #2: Child calls exec()

• Load new frames
• Copy frame
• Both pages now COW
COW, then call exec (after)

Option #2: Child calls exec()

• Load new frames
• Copy frame
• Both pages no longer COW
Downsides to Paging

Memory Consumption:
- Internal Fragmentation
  - Make pages smaller? But then...
- Page Table Space: consider 32-bit address space, 4KB page size, each PTE 8 bytes
  - How big is this page table?
  - How many pages in memory does it need?

Performance: every data/instruction access requires \textit{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 the Rescue!

+ 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>
<tr>
<td></td>
<td>offset</td>
</tr>
<tr>
<td></td>
<td>10</td>
</tr>
</tbody>
</table>
This one goes to three!

+ 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:

- table of memory frames (if bottom level)
- table of page table frames (if multilevel 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
Inverted Page Table: Motivation

So many virtual pages…

…comparatively few physical frames

Traditional Page Tables:
• map pages to frames
• are numerous and sparse

Why not map frames to pages? (How?)
Inverted Page Table: Implementation

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

Virtual address

Page #

offset

Search For matching page & pid

Page-table

frame page pid

Physical Memory

offset

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) → 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)

Cache of virtual to physical page translations

Major efficiency improvement
Address Translation with TLB

Access TLB before you access memory.

Trick: access TLB while you access the cache.
Address Translation Uses!

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

Efficient interprocess 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