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 and temporal)
• 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 the best use of this limited resource
A Day in the Life of a Program

Compiler (+ Assembler + Linker)

Loader

"It's alive!"

source files

sum.c

executable

pid xxx

process

```
#include <stdio.h>

int max = 10;

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

0xffffffff

stack

heap
data
text

0x00000000 0xffffffff
Where do we store virtual memory?

Need to find a place where the physical memory of the process lives

→ Keep track of a “free list” of available memory blocks (so-called “holes”)
Dynamic Storage-Allocation Problem

- **First-fit**: Allocate *first* hole that is big enough
- **Next-fit**: Allocate *next* hole that is big enough
- **Best-fit**: Allocate *smallest* hole that is big enough; must search entire free list, unless ordered by size
  - Produces the smallest leftover hole
- **Worst-fit**: Allocate *largest* hole; must also search entire free list
  - Produces the largest leftover hole
Fragmentation

• **Internal Fragmentation** – allocated memory may be larger than requested memory; this size difference is memory internal to a partition, but not being used

• **External Fragmentation** – total memory space exists to satisfy a request, but it is not contiguous
How do we map virtual → physical

• Having found the physical memory, how do we map virtual addresses to physical addresses?
Early Days: Base and Limit Registers

**Base and Limit** registers for each process

- **Base** and **Limit** registers for each process

**Physical Memory**

- **Process 1**
  - Limit
  - Base
  - Physical address = virtual address + base
  - (segmentation fault if virtual address ≥ limit)

- **Process 2**
  - Limit
  - Base
Early Days: Base and Limit Registers

**Base** and **Limit** registers for each process

Physical Memory

<table>
<thead>
<tr>
<th>Base</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

process 1

physical address = virtual address + base

(segmentation fault if virtual address ≥ limit)

process 2

virtual hole between heap and stack leads to significant **internal** fragmentation
Next: segmentation

- **Base** and **Limit** register for each segment: code, data/heap, stack

Physical Memory

- **code**
  - Limit
  - Base

- **stack**
  - Limit
  - Base

- **data/heap**
  - Limit
  - Base

physical address = virtual address − virtual start of segment + base
Next: segmentation

- **Base** and **Limit** register for each segment: code, data/heap, stack

- Physical holes between segments leads to significant external fragmentation

- Physical address = virtual address - virtual start of segment + base
Paged Translation

TERMINOLOGY ALERT:
Page: virtual
Frame: physical

Solves both internal and external fragmentation! (to a large extent)
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 of a process may be mapped to frames
Address Translation, Conceptually

- Processor
- Virtual Address
- Translation
  - Valid
  - Invalid
    - Raise Exception
- 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 \textit{virtual} addresses
- Never sees the physical address

Physical Memory:
- deals with \textit{physical} addresses
- Never sees the virtual address
red cube is 255\textsuperscript{th} byte in page 2.

Where is the red cube in physical memory?
Virtual Address Components

**Page number** – Upper bits (most significant bits)
- Must be translated into a physical frame number

**Page offset** – Lower bits (least significant bits)
- Does not change in translation

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>
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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>5...</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
Simple Page Table

Lives in Memory

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

Performance: every data/instruction access requires *two* memory accesses:

• One for the page table
• One for the data/instruction
Optimal Page Size: P

- Overhead due to internal fragmentation: $P / 2$
- Overhead due to page table:
  - \#pages x PTE size
  - \#pages = average process memory size / P
- Total overhead:
  - \((\text{process size} / P) \times \text{PTE size} + (P / 2)\)
- Optimize for P
  - \(\frac{d(\text{overhead})}{dP} = 0\)
- Optimize for P
  - \(P = \sqrt{2 \times \text{process size} \times \text{PTE size}}\)
- Example: 1 MB process, 8 byte PTE
  - \(\sqrt{2 \times 2^{20} \times 2^3} = \sqrt{2^{24}} = 2^{12} = 4 \text{ KB}\)
Address Translation

- Paged Translation
- Efficient Address Translation
  - Multi-Level Page Tables
  - Inverted Page Tables
- TLBs
Physical Memory

Multi-Level Page Tables to reduce page table space

PTBR points to level 1 page table
+ 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 (say):
  – 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)

| Present | Protection R/W/X | Ref | Dirty | Index |

**Index** is an index into (depending on Present bit):
- frames
  - physical process memory or next level page table
- backing store
  - if page was swapped out

**Synonyms:**
- 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...

P1 virtual address space

P2 virtual address space

P3 virtual address space

P4 virtual address space

P5 virtual address space

... 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

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)

Sharing frames doesn’t work:
→ use ”segment identifier” instead of process identifiers in inverted page table
(multiple processes can share the same segment)
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.

Diagram:
- Processor
- TLB
- Page Table
- Physical Memory

Flow:
1. Processor sends Virtual Address to TLB.
2. TLB checks for a Hit.
3. If a Hit, TLB sends Frame to Page Table.
4. Page Table sends Valid Frame to TLB.
5. TLB adds Offset to Physical Address.
6. Physical Memory returns Data.
7. Data is sent to Processor.
8. If no Hit, TLB raises Exception.
Why not just have a large TLB?
Why not just have a large TLB?

• TLBs are fast because they are small
Software vs. Hardware-Loaded TLB

• Software-loaded: TLB-miss $\rightarrow$ software handler
• Hardware-loaded: TLB-miss $\rightarrow$ hardware ”walks” page table itself
• may lead to “page fault” if page is not in memory
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