Address Translation
Main Points

• Address Translation Concept
  – How do we convert a virtual address to a physical address?

• Flexible Address Translation
  – Segmentation
  – Paging
  – Multilevel translation

• Efficient Address Translation
  – Translation Lookaside Buffers
  – Virtually and physically addressed caches
Address Translation Concept

[Diagram showing the process flow from Processor to Physical Memory, including Translation and MMU.]
Address Translation Goals

• Memory protection
  – Isolate process to its only memory
  – Prevent virus from re-writing machine instructions
• Memory sharing
  – Shared libraries, interprocess communication
• Sparse addresses
  – Dynamically allocated regions: heaps, stacks, mmap
• Efficiency
  – Reduce fragmentation and copying
  – Runtime lookup cost and TLB hit rate
  – Translation table size
• Portability
Bonus Feature

• What if the kernel can regain control whenever a program reads or writes a particular virtual memory location?

• Examples:
  – Copy on write
  – Zero on reference
  – Fill on demand
  – Demand paging
  – Memory mapped files
  – ...

Virtually Addressed Base and Bounds

Processor's View

Virtual Address

Virtual Memory

Implementation

Base

Virtual Address + Physical Address

Bound

Raise Exception

Physical Memory

Base

Base + Bound
Virtually Addressed Base and Bounds

• Pros?
  – Simple
  – Fast (2 registers, adder, comparator)
  – Safe
  – Can relocate in physical memory without changing process

• Cons?
  – Can’t keep program from accidentally overwriting its own code
  – Can’t share code/data with other processes
  – Can’t grow stack/heap as needed
Process Regions or Segments

• Every process has logical regions or segments
  – Contiguous region of process memory
• Code, data, heap, stack, dynamic library (code, data), memory mapped files, ...
• Each with its own
  – protection: read-only, read-write, execute-only
  – sharing: code vs. data
  – access pattern: code vs. mmap file
Segmentation

- Segment is a contiguous region of *virtual* memory
- Each process has a segment table (in hardware)
  - Entry in table = segment
- Segment can be located anywhere in physical memory
  - Each segment has: start, length, access permission
- Processes can share segments
  - Same start, length, same/different access permissions
Segmentation

Processor's View

Virtual Memory

Virtual Address

Code

Data

Heap

Stack

Implementation

Processor

Virtual Address

Segment Table

Segment

Offset

Base

Bound

Access

Read

R/W

R/W

R/W

Physical Address

Raise Exception

Physical Memory

Base 3

Stack

Base+ Bound 3

Base 0

Code

Base+ Bound 0

Base 1

Base+ Bound 1

Base 2

Base+ Bound 2

external fragmentation
<table>
<thead>
<tr>
<th>Segment start</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>code</td>
<td>0x700</td>
</tr>
<tr>
<td>data</td>
<td>0x500</td>
</tr>
<tr>
<td>heap</td>
<td></td>
</tr>
<tr>
<td>stack</td>
<td>0x1000</td>
</tr>
</tbody>
</table>

### Virtual Memory

- **main**: 0:240  
  - store #108, r2
- 0:244  
  - store pc+8, r31
- 0:248  
  - jump 360
- 0:24c
- ...  
- **strlen**: 0:360  
  - loadbyte (r2), r3
- ...  

### Physical Memory

- **x**: 108  
  - a b c \0
- ...  
- **main**: 4240  
  - store #1108, r2
- 4244  
  - store pc+8, r31
- 4248  
  - jump 360
- 424c
- ...  
- **strlen**: 4360  
  - loadbyte (r2), r3
- ...  
- **x**: 1:108  
  - a b c \0
- ...
Question

- With segmentation, what is saved/restored on a process context switch?

x86: global descriptor table, register, segment table
Segmentation

• Pros?
  – Can share code/data segments between processes
  – Can protect code segment from being overwritten

• Cons? Complex memory management
  – Need to find chunk of a particular size
  – May need to rearrange memory to make room for new segment or growing segment (e.g., sbrk)
  – External fragmentation: wasted space between chunks
Paged Translation

- Manage memory in fixed size units, or pages
- Finding a free page is easy
  - Bitmap allocation: 0011111100000001100
  - Each bit represents one physical page frame
- Each process has its own page table
  - Stored in physical memory
  - Hardware registers
    - pointer to page table start
    - page table length
Processor's View

Physical Memory

VPage 0
Code
Data
Heap
Stack

VPage 1

VPage N

Frame 0
Code0
Data0
Heap1
Code1
Heap0
Data1

Heap2
Stack1
Stack0

Frame M
Paging Questions

• With paging, what is saved/restored on a process context switch?
  – Pointer to page table, size of page table
  – Page table itself is in main memory

• What if page size is very small?

• What if page size is very large?
  – Internal fragmentation: if we don’t need all of the space inside a fixed size chunk
Paging and Sharing

• Can we use page tables to share memory between processes?

• Set page tables to point to same page frame

• Need *core map*
  - Array of information about each physical page frame
  - Set of processes pointing to that page frame
  - When reference count goes to zero, can reclaim!
Question

• How big a user stack should I allocate?

• What if some programs need a large stack and others need a small one?
Expand Stack on Reference

- When program references memory beyond end of stack
  - Page fault into OS kernel
  - Kernel allocates some additional memory
    - How much?
  - Remember to zero the memory to avoid accidentally leaking information!
  - Modify page table
  - Resume process
UNIX fork seems inefficient

- Makes a complete copy of process
- Throw copy away on exec
- Do we need to make the copy?
  - One solution: change the syscall interface!
Copy on Write

- Paging allows an efficient fork
  - Copy page table of parent into child
  - Mark all pages (in new/old page tables) as read-only
  - Start child process; restart parent
  - Trap into kernel on write (in child or parent)
  - Copy page
  - Mark both as writeable
  - Resume execution