CS 134: Operating Systems
More Memory Management
Overview

Segmentation Recap

Paging
Logical address consists of the pair

<segment-number, offset>

**Example**

Use 32-bit logical address

- High-order 8 bits are segment number
- Low-order 24 bits are offset within segment

256 segments, of max size 16,777,216 bytes (16MB)
Processor needs to map 2D user-defined addresses into 1D physical addresses.

In *segment table*, each entry has:

- **Base**—Starting address of the segment in physical memory
- **Limit**—Length of the segment
Segment Translation

![Diagram of segment translation process]

- Logical address
- Segment table
- Physical address
- Physical memory
- Trap
Segmentation Recap

Segmentation Architecture

- Relocation
  - Dynamic
  - By segment table
- Sharing
  - Shared segments
  - Same segment number
- Allocation
  - First fit/best fit
  - External fragmentation

Class Exercise
Do shared segments need to have the same segment number.

- If so, why?
- If not, why? (Why might we give them the same segment number anyway?)
Class Exercise

Does our segmentation scheme capture the *difference* between code and data segments?
- If not, what would we need to fix it?

Class Exercise

What if a program wants more contiguous data space than a segment can hold? Is this a problem?
Properties

- All pages are the same size (e.g., 4K)
- No need for limit registers
- No longer reflect program structure
- Physical locations for pages are called page frames
Now have a lot of pages.

- 4K pages & 32-bit logical address
  - 20-bit page number, 12-bit offset
- 20-bit page number ⇒ 1,048,576 possible pages!
- Too many to remember inside processor
Sparsely Filled Address Spaces

For example,

- Nothing at address zero (why?)
- Code low down in memory
- Static and heap data after code (room to grow up)
- Stack high up (room to grow down)
Sparsely Filled Address Spaces

For example,
- Nothing at address zero (why?)
- Code low down in memory
- Static and heap data after code (room to grow up)
- Stack high up (room to grow down)
- Kernel really high up

Solution
- Two-level (or three-level) page tables
  - 10-bit upper page number (0-1023)
  - 10-bit lower page number (0-1023)
  - 12-bit offset (0-4095)
Sparsely Filled Address Spaces

For example,

- Nothing at address zero (why?)
- Code low down in memory
- Static and heap data after code (room to grow up)
- Stack high up (room to grow down)
- Kernel really high up

Solution (?)

Two-level (or three-level) page tables

- 10-bit upper page number (0-1023)
- 10-bit lower page number (0-1023)
- 12-bit offset (0-4095)
Huh?
Class Exercise

What are the pros and cons?

How big a TLB do you want?
Here’s what we want:

- Needs to be in memory
- Size is $O(\text{frames})$
- Want $O(1)$ performance
- Needs to act like a TLB, i.e.,
  - Can be seen as “just a big cache”
  - Maps pages $\rightarrow$ frames
  - Don’t want to have to flush it all the time
Inverted Page Tables

- One row per physical frame, with reverse mapping
- Given virtual address, how to find physical one?
  - Basically a search problem

Question:
Is the hash table bigger than the number of frames?
Inverted Page Tables

- One row per physical frame, with reverse mapping
- Given virtual address, how to find physical one?
  - Basically a search problem
  - Hash tables to the rescue!

**Question:** Is the hash table bigger than the number of frames?
Hashed (Inverted) Page Tables

- Logical address: pid \( \rightarrow \) hash \( \rightarrow \) page, offset
- Physical address: frame \( \rightarrow \) offset

<table>
<thead>
<tr>
<th>pid</th>
<th>page</th>
<th>frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>17</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>15</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>14</td>
<td>4</td>
</tr>
</tbody>
</table>

 hash table

 physical memory (frames)
A question

Does this claim make sense?

method or interprocess communication. Some operating systems implement shared memory using shared pages.

Systems that use inverted page tables have difficulty implementing shared memory. Shared memory is usually implemented as multiple virtual addresses (one for each process sharing the memory) that are mapped to one physical address. This standard method cannot be used, however, as there is only one virtual page entry for every physical page, so one physical page cannot have two (or more) shared virtual addresses.

Organizing memory according to pages provides numerous other benefits.

*Operating Systems Concepts*, Silberschatz & Galvin
## Processors Compared

<table>
<thead>
<tr>
<th>Processor</th>
<th>Physical addr</th>
<th>Virtual addr</th>
<th>TLB Size</th>
<th>Segments</th>
<th>Pages</th>
<th>Hashed addr</th>
<th>Hashed page tables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pentium 4</td>
<td>36-bit</td>
<td>32-bit</td>
<td>64</td>
<td>varied</td>
<td>4k, 4M</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Opteron</td>
<td>40-bit</td>
<td>48-bit</td>
<td>1088</td>
<td>varied</td>
<td>4k, 4M</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Itanium 2</td>
<td>50-bit</td>
<td>64-bit</td>
<td>4 × 32</td>
<td>—</td>
<td>4k…4G</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>PowerPC 604</td>
<td>32-bit</td>
<td>52-bit</td>
<td>256</td>
<td>&lt; 256MB</td>
<td>4k</td>
<td>Yes</td>
<td>—</td>
</tr>
<tr>
<td>PowerPC 970</td>
<td>42-bit</td>
<td>64-bit</td>
<td>1024</td>
<td>&lt; 256MB</td>
<td>4k</td>
<td>Yes</td>
<td>—</td>
</tr>
<tr>
<td>UltraSparc</td>
<td>36-bit</td>
<td>64-bit</td>
<td>64</td>
<td>—</td>
<td>8k…4M</td>
<td>Yes</td>
<td>—</td>
</tr>
<tr>
<td>Alpha</td>
<td>41-bit</td>
<td>64-bit</td>
<td>256</td>
<td>—</td>
<td>8k…4M</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>MIPS R3000</td>
<td>32-bit</td>
<td>32-bit</td>
<td>64</td>
<td>—</td>
<td>4k…</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>
Programs do not need all their code all the time...
On modern Unix systems

- handle = dlopen(filename, mode)
- addr = dlsym(handle, sym)
- err = dlclose(handle)

Issues...?
We now have a memory scheme where

- Programs use *logical addresses*
- Memory sharing is easy
- Processes are either in memory or swapped out
- Hardware can detect *invalid* memory accesses to trap to the OS

We can already “swap out” whole programs, but can we do better...?
Demand Paging

Need to
- Bring a page into memory only when it is needed.
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users & processes
- Mark pages not in memory as invalid in page table

When program accesses an invalid page, two possibilities...
Demand Paging—Hardware Support

Thus,

- Invalid accesses generate a trap
- Need to restart program after the trap
- Must seem like “nothing happened”

**Example:** The C-code for:

```c
--mystack = new_item;
```

might be implemented as a single instruction:

```assembly
mov -(r6),r1
```

**Class Exercise**

Why is this instruction potentially problematic?
What needs to happen when a page fault occurs?
Page Faults

What happens...

- User process accesses invalid memory—traps to OS
- OS saves process state
- OS checks access was actually legal
- Find a free frame
- Read from swap to free frame—I/O wait, process blocked
- Interrupt from disk (I/O complete)—process ready
- Scheduler restarts process—process running
- Adjust page table
- Restore process state
- Return to user code
How long?

- Disk is slow
- 5–15 ms is a conservative guess
- Main memory takes 5–15 ns
- Page fault is about \textbf{1 million times slower} than a regular memory access
- Page faults must be rare! (Need locality!)
A “Back of an Envelope Calculation”

How often are there page faults?

An example from a desktop machine:

- In 14 days
  - 378,110 page-ins
  - Average load < 4% → 12 hours actual compute time
  - 8.75 page faults per second average
- 1,000,000,000 memory accesses per second (a guess)
- 43,200,000,000,000 memory accesses in 12 hours
- 1 page-in every 114,252,466 memory accesses
- Using 5 ns for memory, 5 ms for disk:
  \[ t_{avg} = \frac{5,000,000 \times 1 + 5 \times 114,252,465}{114,252,466} \]
  \[ t_{avg} = 5.04\text{ns} \]
Other kinds of page faults:

- Demand-page executables from their files, not swap device
- Copy-on-write memory—great for fork
- Lazy memory allocation
- Other tricks...