Virtual Memory 1

Kevin Walsh
CS 3410, Spring 2010
Computer Science
Cornell University

P & H Chapter 5.4 (up to TLBs)
CPU address/data bus...
... routed through caches
... to main memory

- Simple, fast, but...

Q: What happens for LW/SW to an invalid location?

- 0x000000000 (NULL)
- uninitialized pointer
Running multiple processes...

*Time-multiplex* a single CPU core (*multi-tasking*)

- Web browser, skype, office, ... all must co-exist

Many cores per processor (*multi-core*)

- or many processors (*multi-processor*)

- Multiple programs run *simultaneously*
Q: What happens when another program is executed concurrently on another processor?

- Take turns using memory?
Can we relocate second program?

- What if they don’t fit?
- What if not contiguous?
- Need to recompile/relink?
- ...

Solution? Multiple processes/processors
All problems in computer science can be solved by another level of indirection.

– David Wheeler
– or, Butler Lampson
– or, Leslie Lamport
– or, Steve Bellovin
Virtual Memory: A Solution for All Problems

Each process has its own virtual address space

• Programmer can code as if they own all of memory

On-the-fly at runtime, for each memory access

• all access is *indirect* through a virtual address
• translate fake virtual address to a real physical address
• redirect load/store to the physical address
Programs load/store to virtual addresses
Actual memory uses physical addresses

**Memory Management Unit (MMU)**
- Responsible for translating on the fly
- Essentially, just a big array of integers:
  \[ \text{paddr} = \text{PageTable}[\text{vaddr}] \]
Advantages

Easy relocation

• Loader puts code anywhere in physical memory
• Creates virtual mappings to give illusion of correct layout

Higher memory utilization

• Provide illusion of contiguous memory
• Use all physical memory, even physical address 0x0

Easy sharing

• Different mappings for different programs / cores

And more to come…
Address Translation

Pages, Page Tables, and the Memory Management Unit (MMU)
Attempt #1: How does MMU translate addresses?

\[ \text{paddr} = \text{PageTable}[\text{vaddr}] \]

Granularity?

- Per word...
- Per block...
- Variable...

Typical:

- 4KB – 16KB \text{pages}
- 4MB – 256MB \text{jumbo pages}
Attempt #1: For any access to virtual address:

- Calculate **virtual page number** and **page offset**
- Lookup **physical page number** at PageTable[vpn]
- Calculate physical address as ppn:offset
Q: Where to store page tables?
A: In memory, of course...

Special *page table base register*
(CR3:PTBR on x86)
(Cop0:ContextRegister on MIPS)
* lies to children
Overhead for VM Attempt #1 (example)

Virtual address space (for each process):

• total memory: $2^{32}$ bytes = 4GB
• page size: $2^{12}$ bytes = 4KB
• entries in PageTable?
• size of PageTable?

Physical address space:

• total memory: $2^{29}$ bytes = 512MB
• overhead for 10 processes?

* lies to children
Cool Trick #1: Don’t map all pages

Need **valid bit** for each page table entry

Q: Why?
Assume most of PageTable is empty

How to translate addresses? Multi-level PageTable

* x86 does exactly this
## Cool Trick #2: Page permissions!

Keep **R, W, X permission bits** for each page table entry.

**Q: Why?**

<table>
<thead>
<tr>
<th>V</th>
<th>R</th>
<th>W</th>
<th>X</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>0x10045</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>0xC20A3</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>0x4123B</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>0x00000</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Cool Trick #3: **Aliasing**

Map the same physical page at several virtual addresses

Q: Why?
Paging
Can we run process larger than physical memory?

- The “virtual” in “virtual memory”

View memory as a “cache” for secondary storage

- **Swap** memory pages out to disk when not in use
- **Page** them back in when needed

Assumes Temporal/Spatial Locality

- Pages used recently most likely to be used again soon
Paging

Cool Trick #4: Paging/Swapping

Need more bits:

Dirty, RecentlyUsed, ...
Role of the Operating System

Context switches, working set, shared memory
Suppose Firefox needs a new page of memory

1. Invoke the Operating System
   ```c
   void *sbrk(int nbytes);
   ```

2. OS finds a free page of physical memory
   - clear the page (fill with zeros)
   - add a new entry to Firefox’s PageTable
Suppose Firefox is idle, but Skype wants to run

(1) Firefox invokes the Operating System
int sleep(int nseconds);

(2) OS saves Firefox’s registers, load skype’s
  • (more on this later)

(3) OS changes the CPU’s Page Table Base Register
  • Cop0:ContextRegister / CR3:PDBR

(4) OS returns to Skype
Suppose Firefox and Skype want to share data

(1) OS finds a free page of physical memory

- clear the page (fill with zeros)
- add a new entry to Firefox’s PageTable
- add a new entry to Skype’s PageTable
  - can be same or different vaddr
  - can be same or different page permissions
Suppose Skype needs a new page of memory, but Firefox is hogging it all

(1) Invoke the Operating System
   ```c
   void *sbrk(int nbytes);
   ```

(2) OS can’t find a free page of physical memory
   • Pick a page from Firefox instead (or other process)

(3) If page table entry has dirty bit set...
   • Copy the page contents to disk

(4) Mark Firefox’s page table entry as “on disk”
   • Firefox will fault if it tries to access the page

(5) Give the newly freed physical page to Skype
   • clear the page (fill with zeros)
   • add a new entry to Skyp’s PageTable
OS multiplexes physical memory among processes

• assumption # 1: processes use only a few pages at a time

• working set = set of process’s recently actively pages
Q: What if working set is too large?

Case 1: Single process using too many pages

Case 2: Too many processes
Thrashing b/c working set of process (or processes) greater than physical memory available

- Firefox steals page from Skype
- Skype steals page from Firefox

- I/O (disk activity) at 100% utilization
  - But no useful work is getting done

Ideal: Size of disk, speed of memory (or cache)
Non-ideal: Speed of disk
OS multiplexes physical memory among processes

• assumption # 2: recent accesses predict future accesses
• working set usually changes slowly over time
Q: What if working set changes rapidly or unpredictably?

A: Thrashing b/c recent accesses don’t predict future accesses
How to prevent thrashing?

• User: Don’t run too many apps
• Process: efficient and predictable mem usage
• OS: Don’t over-commit memory, memory-aware scheduling policies, etc.