Researchers at Stanford have developed “nanoscale single-mode LED”, which can transmit chip-to-chip data at 10 Gbs (10x what is currently used) at 1/1000th the energy. Pretty cool! (get it?) 😊

Review

- Next level in the memory hierarchy:
  - Provides program with **illusion** of a very large main memory:
  - Working set of “pages” reside in main memory - others reside on disk.

- Also allows OS to share memory, protect programs from each other

- Today, more important for **protection** vs. just another level of memory hierarchy

- Each process thinks it has all the memory to itself

- (Historically, it predates caches)
Review: View of the Memory Hierarchy

Thus far

This week: Virtual Memory

Upper Level

Faster

Larger

Lower Level

Regs

Instr. Operands

Cache

Blocks

L2 Cache

Blocks

Memory

Pages

Disk

Pages

Files

Tape

Thus far

This week: Virtual Memory
Mapping Virtual Memory to Physical Memory

- Divide into equal sized chunks (about 4 KB - 8 KB)
- Any chunk of Virtual Memory assigned to any chunk of Physical Memory ("page")

Virtual Memory

Physical Memory

64 MB

Current page 0

Mapping Virtual Memory to Physical Memory

- Divide into equal sized chunks (about 4 KB - 8 KB)
- Any chunk of Virtual Memory assigned to any chunk of Physical Memory ("page")
Another Model: Base and Bound Reg

- $\infty$
- $\text{User C}$
- $\text{User B}$
- $\text{User A}$
- OS

- $\text{base}$
- $\text{bound}$
- $\text{base + bound}$

- Enough space for User D, but discontinuous ("fragmentation problem")
  - Want:
    - discontinuous mapping
    - Process size >> mem
  - Addition not enough!
  => use Indirection!
Virtual Memory Mapping Function

- Cannot have simple function to predict arbitrary mapping
- Use table lookup of mappings
  
<table>
<thead>
<tr>
<th>Page Number</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical Page Number</td>
<td>PageTable[Virtual Page Number]</td>
</tr>
</tbody>
</table>

- Use table lookup ("Page Table") for mappings: Page number is index

- Virtual Memory Mapping Function
  - Physical Offset = Virtual Offset
  - Physical Page Number = PageTable[Virtual Page Number]
    (P.P.N. also called "Page Frame")
Address Mapping: Page Table

Virtual Address:

- page no.
- offset

Page Table

Base Reg

- index into page table

Page Table located in physical memory

Page Table:

- V
- A.R.
- P. P. A.
- Val
- id
- Access Rights
- Physical Page Address

- PPN
- offset

Physical Memory Address
Page Table

- A page table is an operating system structure which contains the mapping of virtual addresses to physical locations
  - There are several different ways, all up to the operating system, to keep this data around

- Each process running in the operating system has its own page table
  - “State” of process is PC, all registers, plus page table
  - OS changes page tables by changing contents of Page Table Base Register
Requirements revisited

- Remember the motivation for VM:
  - Sharing memory with protection
    - Different physical pages can be allocated to different processes (sharing)
    - A process can only touch pages in its own page table (protection)
  - Separate address spaces
    - Since programs work only with virtual addresses, different programs can have different data/code at the same address!

- What about the memory hierarchy?
Page Table Entry (PTE) Format

- Contains either Physical Page Number or indication not in Main Memory
- OS maps to disk if Not Valid \((V = 0)\)

If valid, also check if have permission to use page: **Access Rights** \((A.R.)\) may be Read Only, Read/Write, Executable
Paging/Virtual Memory Multiple Processes

User A:

Virtual Memory

∞

Code

Static

Code

0

Page Table

A

Physical Memory

64 MB

User B:

Virtual Memory

∞

Stack

Static

Code

0

Page Table

B

Physical Memory

64 MB

García, Fall 2011 © UCB
## Comparing the 2 levels of hierarchy

<table>
<thead>
<tr>
<th>Cache version</th>
<th>Virtual Memory vers.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block or Line</td>
<td>Page</td>
</tr>
<tr>
<td>Miss</td>
<td>Page Fault</td>
</tr>
<tr>
<td>Block Size: 32-64B</td>
<td>Page Size: 4K-8KB</td>
</tr>
<tr>
<td>Placement:</td>
<td>Fully Associative</td>
</tr>
<tr>
<td>Direct Mapped,</td>
<td></td>
</tr>
<tr>
<td>N-way Set Associative</td>
<td></td>
</tr>
<tr>
<td>Replacement:</td>
<td>Least Recently Used (LRU)</td>
</tr>
<tr>
<td>LRU or Random</td>
<td></td>
</tr>
<tr>
<td>Write Thru or Back</td>
<td>Write Back</td>
</tr>
</tbody>
</table>
Notes on Page Table

- Solves Fragmentation problem: all chunks same size, so all holes can be used
- OS must reserve “Swap Space” on disk for each process
- To grow a process, ask Operating System
  - If unused pages, OS uses them first
  - If not, OS swaps some old pages to disk
  - (Least Recently Used to pick pages to swap)
- Each process has own Page Table
- Will add details, but Page Table is essence of Virtual Memory
Why would a process need to “grow”?

- A program’s **address space** contains 4 regions:
  - **stack**: local variables, grows downward
  - **heap**: space requested for pointers via `malloc()`; resizes dynamically, grows upward
  - **static data**: variables declared outside main, does not grow or shrink
  - **code**: loaded when program starts, does not change

For now, OS somehow prevents accesses between stack and heap (gray hash lines).
Virtual Memory Problem #1

- Map every address $\Rightarrow$ 1 indirection via Page Table in memory per virtual address $\Rightarrow$ 1 virtual memory accesses = 2 physical memory accesses $\Rightarrow$ SLOW!
- Observation: since locality in pages of data, there must be locality in virtual address translations of those pages
- Since small is fast, why not use a small cache of virtual to physical address translations to make translation fast?
- For historical reasons, cache is called a Translation Lookaside Buffer, or TLB
Translation Look-Aside Buffers (TLBs)

- TLBs usually small, typically 128 - 256 entries
- Like any other cache, the TLB can be direct mapped, set associative, or fully associative

On TLB miss, get page table entry from main memory
Another Analogy

- Book title like virtual address
- Library of Congress call number like physical address
- Card catalogue like page table, mapping from book title to call #
- On card for book, in local library vs. in another branch like valid bit indicating in main memory vs. on disk
- On card, available for 2-hour in library use (vs. 2-week checkout) like access rights
Peer Instruction

1) Locality is important yet different for cache and virtual memory (VM): temporal locality for caches but spatial locality for VM

2) VM helps both with security and cost
Peer Instruction Answer

1) Locality is important, but different for cache and virtual memory (VM). Temporal locality for caches but spatial locality for VM.

   1. No. Both for VM and cache

2) VM helps both with security and cost.

   2. Yes. Protection and a bit smaller memory

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a)</td>
<td>FF</td>
</tr>
<tr>
<td>b)</td>
<td>FT</td>
</tr>
<tr>
<td>c)</td>
<td>TF</td>
</tr>
<tr>
<td>d)</td>
<td>TT</td>
</tr>
</tbody>
</table>
And in conclusion…

- Manage memory to disk? Treat as cache
  - Included protection as bonus, now critical
  - Use Page Table of mappings for each user vs. tag/data in cache
  - TLB is cache of Virtual ⇒ Physical addr trans

- Virtual Memory allows protected sharing of memory between processes
- Spatial Locality means Working Set of Pages is all that must be in memory for process to run fairly well