Virtual Memory (and Indirection)

An in-depth look at “Make the common case fast”
Today’s lecture

- Learning objectives
  - The penalties for bad memory management are even worse than we’ve told you so far.
  - If you find memory management confusing, be oh so thankful that virtual memory exists.

- Outline of topics
  - Motivations for virtual memory
  - Basic implementation details of virtual memory
  - How to make virtual memory fast: Translation Lookaside Buffers (TLBs) (i.e., caches!)
What happens if processes need more data than we have memory?

process1
I need 8GB of data!

process2
Me too!

4 GB of RAM
What happens if two processes want to use the same memory addresses?

4 GB of RAM

process1

store $t1, 0x00001111

process2

store $s2, 0x00001111
Use indirection to keep track of where things are stored and not the actual things

- “Any problem in CS can be solved by adding a level of indirection”

- Without Indirection

- With Indirection
The memory addresses in your programs are actually pointing to virtual addresses

lw $t1, 0($5)

Expectations vs. “Reality”

Physical Memory

Virtual address

Physical Memory

R[ 5] +0

R[ 5] +0
Virtual Memory translates “virtual addresses” into “physical addresses” in either physical memory or disk
All programs write to the same “virtual addresses” but to different “physical addresses”

- By allocating distinct regions of physical memory to A and B, they are prevented from reading/writing each other’s data.
We want the performance of memory but the size of disk

Memory

Modest capacity
(too small for all data)
Reasonably fast
(~100 cycles)

Disk

Enormous capacity
Painfully slow
(~10,000,000 cycles)
Virtual memory treats memory as a cache for disk

<table>
<thead>
<tr>
<th>Cache</th>
<th>Virtual memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache is fast memory</td>
<td>Memory is fast disk</td>
</tr>
<tr>
<td>Cache blocks (~32B-64B)</td>
<td>VM “pages” (~16 KB)</td>
</tr>
<tr>
<td>2- to 4-way set associativity</td>
<td>Fully associative</td>
</tr>
<tr>
<td>1 cycle for hits, 100 cycles for misses</td>
<td>100 cycles for hits, 10,000,000 cycles for misses</td>
</tr>
</tbody>
</table>
If virtual memory is fully associative, how do we find the right page without scanning all of memory?
**Use an index (the page table) to find pages**

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Physical Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000 to 0x00000FFF</td>
<td>0x23000 to 0x23FFF</td>
</tr>
<tr>
<td>0x00001000 to 0x00001FFF</td>
<td>0x4E000 to 0x4EFFF</td>
</tr>
<tr>
<td>0x00002000 to 0x00002FFF</td>
<td>0x78000 to 0x78FFF</td>
</tr>
<tr>
<td>0x00003000 to 0x00003FFF</td>
<td>0x93000 to 0x93FFF</td>
</tr>
<tr>
<td>0x00004000 to 0x00004FFF</td>
<td>0xDA000 to 0xDAFFF</td>
</tr>
<tr>
<td>0x00005000 to 0x00005FFF</td>
<td>0xBC000 to 0xBCFFF</td>
</tr>
<tr>
<td>0x00006000 to 0x00006FFF</td>
<td>0xAE000 to 0xAEFFF</td>
</tr>
</tbody>
</table>

**Virtual Page Number (VPN)**

<table>
<thead>
<tr>
<th>Page Offset</th>
</tr>
</thead>
</table>

**Physical Page Number (PPN)**
Each process has a separate page table
Page Table

Virtual address

31 30 29 28 27 ................. 15 14 13 12 11 10 9 8 ....... 3 2 1 0

Virtual page number

Page offset

Physical page number

Page table

If 0 then page is not present in memory

Physical address

VPN

Off	base

PTBR

PPN

base

Valid

20

18

31

Data

M[Addr]
Each page is comprised of several cache blocks.
Clicker Question

Consider the following page design

- 32-bit virtual addresses
- Page uses 14 bits for the page offset
- Cache uses 32B cache blocks

How many cache blocks will fit in a page

a) 128
b) 256
c) 512
d) 1024
e) $2^{18}$
How big is the page table?

- From the previous slide:
  - Virtual page number is 20 bits. \(2^{20}\) elements
  - Physical page number is 18 bits + valid bit -> round up to 32 bits. per element

\[
2^{20} \times 2^{2} \text{ B} = 4 \text{ MB}
\]

- How about for a 64b architecture?
  - 30 bit VPN \(\Rightarrow 2^{30}\) elements \(\approx 4 \text{ GB}\)

42 bit VA, 12 bits PU
Use more indirection to manage large page tables

Since most processes don’t use the whole address space, you don’t allocate the tables that aren’t needed

- Also, the 2nd and 3rd level page tables can be “paged” to disk.
Each process will likely use multiple pages at a time.
THE BAD NEWS: All memory accesses now take four accesses

\[ M[M[M[PTBR + VPN1<<2] + VPN2<<2] + VPN3<<2] + \text{offset} \]

How do we make our common case faster?
Translation Lookaside Buffer
Virtual to Physical translations are cached in a Translation Lookaside Buffer (TLB).
What happens on TLB hits and misses?
THE BADDER NEWS: If a page is not in memory, we have a page fault that must access disk!

“Any problem in CS can be solved by adding a level of indirection”

- except too many levels of indirection...
Page faults are analogous to cache misses

**Cache miss**
- Get cache block from memory
- Replace a block based on LRU
- Perform write-back to memory if block was “dirty”

**Page fault**
- Get page from disk
- Replace a page with approximation of LRU
- Perform write-back to disk if page was “dirty”
What happens on a page fault?

- The virtual address is passed to the TLB (Translation Lookaside Buffer).
- If the TLB has a tag miss, it looks up the page in the page table.
- The page table contains a PPN (Page Frame Number) which is used to access memory.
- If there is a cache hit, the data is accessed quickly.
- If there is a cache miss, the data is read from the disk.

Components:
- TLB (Translation Lookaside Buffer)
- Page Table
- Disk
- Register File
- Memory
- Cache

Key terms:
- Virtual page number (VPN)
- Page offset
- Tag
- Index
- Block offset
- Physical address
- PPN (Page Frame Number)
When prefetching, avoid causing page faults

- Prefetches typically dropped when they miss in the TLB
  - Don’t want to disrupt program’s execution for a prefetch.
  - May cause a hardware TLB fill on x86 platforms.

- HW prefetchers don’t cross page boundaries.
  - They use physical addresses
    - Don’t use the TLB.
  - After page boundary don’t know where next page lies
  - Sequential stream will have a few misses @ beginning of each page
Virtual memory is a fast common case solution for these problems but still a bad worst-case

process1

I need 8GB of data!

process1

store $t1, 0x00001111

process2

store $s2, 0x00001111
Virtual memory protects processes from each other by restricting virtual-to-physical translations to the kernel

User mode can modify

Kernel mode can modify

- Page table
- Page table base pointers
- TLB
Pages can be shared between programs

- For example, if you run two copies of a program, the O/S will share the code pages between the programs.
Summary

- Virtual memory is **pure manna from heaven:**
  - It means that we don’t have to manage our own memory.
  - It allows different programs to use the same (virtual) addresses.
  - It provides protection between different processes.
  - It allows controlled sharing between processes (albeit somewhat inflexibly).

- The key technique is **indirection:**
  - Yet another classic CS trick you’ve seen in this class.
  - Many problems can be solved with indirection.

- Caching made a few cameo appearances, too:
  - Virtual memory enables using physical memory as a cache for disk.
  - We used caching (in the form of the Translation Lookaside Buffer) to make Virtual Memory’s indirection fast.