Virtual Memory
Learning to Play Well With Others

malloc(0x20000)

(Physical) Memory
0x10000 (64KB)

Stack

Heap
0x00000
Learning to Play Well With Others

(Physical) Memory
0x10000 (64KB)

Stack

Heap
0x00000
Learning to Play Well With Others
Learning to Play Well With Others

Virtual Memory

Stack

Heap

0x00000

0x400000 (4MB)

Physical Memory

0x00000

0x10000 (64KB)

Virtual Memory

Stack

Heap

0x00000

0xF000000 (240MB)

Disk (GBs)
Mapping

- Virtual-to-physical mapping
  - Virtual --> “virtual address space”
  - physical --> “physical address space”
- We will break both address spaces up into “pages”
  - Typically 4KB in size, although sometimes large
- Use a “page table” to map between virtual pages and physical pages.
- The processor generates “virtual” addresses
  - They are translated via “address translation” into physical addresses.
Implementing Virtual Memory

We need to keep track of this mapping…

Virtual Address Space

Physical Address Space
The Mapping Process

Virtual address (32 bits)

<table>
<thead>
<tr>
<th>Virtual Page Number</th>
<th>Page Offset (log(page size))</th>
</tr>
</thead>
</table>

Virtual-to-physical map

<table>
<thead>
<tr>
<th>Physical Page Number</th>
<th>Page Offset (log(page size))</th>
</tr>
</thead>
</table>

Physical address (32 bits)
Two Problems With VM

• How do we store the map compactly?
• How do we translation quickly?
How Big is the map?

- **32 bit address space:**
  - 4GB of virtual addresses
  - 1MPages
  - Each entry is 4 bytes (a 32 bit physical address)
  - 4MB of map

- **64 bit address space**
  - 16 exabytes of virtual address space
  - 4PetaPages
  - Entry is 8 bytes
  - 64PB of map
Shrinking the map

• Only store the entries that matter (i.e., enough for your physical address space)
• 64GB on a 64bit machine
  • 16M pages, 128MB of map
• This is still pretty big.
• Representing the map is now hard because we need a “sparse” representation.
  • The OS allocates stuff all over the place.
  • For security, convenience, or caching optimizations
  • For instance: The stack is at the “top” of memory. The heap is at the “bottom”
• How do you represent this “sparse” map?
Hierarchical Page Tables

• Break the virtual page number into several pieces
• If each piece has N bits, build an $2^N$-ary tree
• Only store the part of the tree that contain valid pages
• To do translation, walk down the tree using the pieces to select with child to visit.
Hierarchical Page Table

Virtual Address

31  22  21  12  11  0

p1   p2   offset

10-bit L1 index  10-bit L2 index

Root of the Current Page Table

(Processor Register)

Level 1 Page Table

p1

Level 2 Page Tables

p2

offset

Data Pages

Parts of the map that exist

Parts that don’t

Adapted from Arvind and Krste’s MIT Course 6.823 Fall 05
Making Translation Fast

• Address translation has to happen for every memory access
• This potentially puts it squarely on the critical path for memory operation (which are already slow)
“Solution 1” : Use the Page Table

• We could walk the page table on every memory access
• Result: every load or store requires an additional 3-4 loads to walk the page table.

• Unacceptable performance hit.
Solution 2: TLBs

- We have a large pile of data (i.e., the page table) and we want to access it very quickly (i.e., in one clock cycle)
- So, build a cache for the page mapping, but call it a “translation lookaside buffer” or “TLB”
TLBs

• TLBs are small (maybe 128 entries), highly-associative (often fully-associative) caches for page table entries.
• This raises the possibility of a TLB miss, which can be expensive
  • To make them cheaper, there are “hardware page table walkers” -- specialized state machines that can load page table entries into the TLB without OS intervention
  • This means that the page table format is now part of the big-A architecture.
• Typically, the OS can disable the walker and implement its own format.
Solution 3: Defer translating Accesses

- If we translate before we go to the cache, we have a “physical cache”, since cache works on physical addresses.
  - Critical path = TLB access time + Cache access time

- Alternately, we could translate after the cache
  - Translation is only required on a miss.
  - This is a “virtual cache”
The Danger Of Virtual Caches (1)

- Process A is running. It issues a memory request to address 0x10000
  - It is a miss, and 0x10000 is brought into the virtual cache
- A context switch occurs
- Process B starts running. It issues a request to 0x10000
- Will B get the right data?

No! We must flush virtual caches on a context switch.
The Danger Of Virtual Caches

• There is no rule that says that each virtual address maps to a different physical address.
• When this occurs, it is called “aliasing.”
• Example: An alias exists in the cache

- Store B to 0x1000

- Now, a load from 0x2000 will return the wrong value
The Danger Of Virtual Caches

• Why are aliases useful?
• Example: Copy on write
  • memcpy(A, B, 100000)

• Adjusting the page table is much faster for large copies
• The initial copy is free, and the OS will catch attempts to write to the copy, and do the actual copy lazily.
• There are also system calls that let you do this arbitrarily.
Solution (4): Virtually indexed physically tagged

key idea: page offset bits are not translated and thus can be presented to the cache immediately

Index L is available without consulting the TLB

© cache and TLB accesses can begin simultaneously

Critical path = max(cache time, TLB time)!!!

Tag comparison is made after both accesses are completed

Work if the size of one cache way ≤ Page Size
because then none of the cache inputs need to be translated
(i.e., the index bits in physical and virtual addresses are the same)
Virtualizing Memory

- We need to make it appear that there is more memory than there is in a system
  - Allow many programs to be “running” or at least “ready to run” at once (mostly)
  - Absorb memory leaks (sometimes... if you are programming in C or C++)
Page table with pages on disk

Virtual Address

31  22  21  12  11  0

p1  p2  offset

10-bit L1 index
10-bit L2 index

Root of the Current Page Table

(Processor Register)

Level 1 Page Table

p1

Level 2 Page Tables

offset

Data Pages

page in primary memory
page on disk
PTE of a nonexistent page

Adapted from Arvind and Krste’s MIT Course 6.823 Fall 05
The TLB With Disk

- TLB entries always point to memory, not disks
The Value of Paging

• Disk are really really slow.
• Paging is not very useful for expanding the active memory capacity of a system
  • It’s good for “coarse grain context switching” between apps
  • And for dealing with memory leaks ;-
• As a result, fast systems don’t page.
The End
Other uses for VM

- VM provides us a mechanism for adding "meta data" to different regions of memory.
- The primary piece of meta data is the location of the data in physical ram.
- But we can support other bits of information as well
Other uses for VM

• VM provides us a mechanism for adding “meta data” to different regions of memory.
  • The primary piece of meta data is the location of the data in physical ram.
  • But we can support other bits of information as well
• Backing memory to disk
  • next slide
• Protection
  • Pages can be readable, writable, or executable
  • Pages can be cachable or un-cachable
  • Pages can be write-through or write back.
• Other tricks
  • Arrays bounds checking
  • Copy on write, etc.