A system with physical memory only

Challenges

* Physical memory has limited size
* Problem 1: how does everything fit?
* Problem 2: memory management: what goes where?
* Problem 3: how to not overwrite memory accidentally
* Problem 4: how to divide it up

Solution: level of indirection

* Each process is mapped to its own private memory space

Why virtual memory?

* Uses main memory efficiently
  + Uses DRAM as a cache for parts of a virtual address space
* Simplifies memory management
  + Each process gets the same uniform linear address space
* Isolates address spaces
  + One process can’t interfere with another’s memory
  + User program cannot access privileged kernel information and code

Virtual memory

* Programs refer to virtual memory addresses
  + Movl, %ecx, %eax
  + Conceptually very large array of bytes
  + Each byte has its own address
  + Implemented with a hierarchy of different memory types
  + System provides address space private to a particular process

A system with virtual memory

* Used in all modern servers laptops, and smart phones

Virtual vs physical memory

* Programmer sees virtual memory
  + Can assume the memory is infinite
* Reality: physical memory size is much smaller than what the programmer assumes
* The system (software + hardware) maps virtual memory addresses are to physical memory
* The system automatically manages the physical memory space transparently to the programmer
  + Programmer does not need to know the physical size of memory nor manage it
  + A small physical memory can appear as a huge one to the programmer
  + Life is easier for the programmer
* Trade-off: more complex system software and architecture

Virtual memory as a tool for caching

* Conceptually: virtual memory is an array of N contiguous bytes stored on a disk
* The contents of the array on disk are cached in physical memory (DRAM cache)
  + These cache blocks are called pages (size is P = 2^p bytes)

DRAM cache organization

* Driven by the enormous miss penalty
  + DRAM is about 10x slower than SRAM
  + Disk is about 10,000x slower than DRAM
* Consequences
  + Large page (block) size: typically 4KB, sometimes 4MB
  + Fully associative
    - Any virtual page can be placed in any physical page
    - Requires a large mapping function – different from cache memories
  + Highly sophisticated, expensive replacement algorithms
    - Too complicated and open-ended to be implemented in hardware
  + Write-back rather than write-through
    - Write-through: immediately write cache block to the next lower level
    - Write-back: defers the update as long as possible

Enabling data structure: page table

* A page table is an array of page table entries (PTEs) that maps virtual pages to physical pages
  + Per-process kernel data structure in DRAM

Examples

* Determine the number of page table entries (PTEs) that are needed
  + N = virtual address size
  + P = page size
* 1. N = 12, P = 2^p = 1K
  + 1K = 2^10, so p = 10
  + (2^n)/(2^p) = (2^12)/(2^10) = 2^(12-10) = 2^2 = 4
* 2. N = 16, P = 16K
  + 16K = 16 \* 1K = 16 \* 2^10 = 2^4 \* 2^10 = 2^(4 + 10) = 2^14
  + (2^16)/(2^14) = 2^2 = 4

Page hit

* Reference to virtual memory word that is in physical memory (DRAM cache hit)

Page fault

* Reference to virtual memory word that is not in physical memory (DRAM cache miss)

Handling page fault

* Page miss causes page fault (an exception)
* Page fault handler selects a victim to be evicted
* Then the instruction is restarted and this time it’s a hit
* **Demand paging**: waiting until the miss to copy the page to DRAM

Allocating pages

* Allocating a new page of virtual memory

Locality to the rescue again

* Virtual memory seems terribly inefficient, but it works because of locality
* At any point in time, programs tend to access a set of active virtual pages called the working set
  + Programs with better temporal locality will have smaller working sets
* If working set size < main memory size:
  + Good performance for one process after compulsory misses
* If the sum of working set sizes > main memory size:
  + **Thrashing**: performance meltdown where pages are swapped (copied) in and out continuously

Virtual memory as a tool for memory management

* Key idea: each process has its own virtual address space
  + It can view memory as a simple linear array
  + Mapping function scatters addresses through physical memory
    - Well-chosen mappings can improve locality
* Simplifying memory allocation
  + Each virtual page can be mapped to any physical page
  + A virtual page can be stored in different physical pages at different times
* Sharing code ad data among processes
  + Map virtual pages to the same physical page
* Memory protection
  + Extend PTEs with permission bits
  + MMU checks these bits on each access
  + The **SUP bit** indicates whether processes must be running in kernel mode to access the page

Virtual memory address translation

* Virtual address space: V = {0, 1, …, N-1}
* Physical address space: P = {0, 1, …, M-1}
* Address translation
  + MAP: V -> P U {empty}
  + For virtual address a:
    - MAP(a) = a’ if data at virtual address a is at physical address a’ in P
    - MAP(a) = {empty} if data at virtual address a is not in physical memory (either invalid or stored on disk)

Address translation symbols

* Basic parameters
  + N = 2^n: number of addresses in virtual address space
  + M = 2^m: number of addresses in physical address space
  + P = 2^p: page size in bytes
* Components of the virtual address (VA)
  + VPO: virtual page offset
  + VPN: virtual page number
* Components of the physical address (PA)
  + PPO: physical page offset (same as VPO)
  + PPN: physical page number

Accessing the TLB

* MMU uses the VPN portion of the virtual address to access the TLB

TLB hit / miss

* A tlb hit eliminates a memory access
* A tlb miss incurs an additional memory access (the PTE)

Multi-level page tables

* Problem: we would need too much memory for the page table (2^n / 2^p size)
* Multi-level page table is the solution
* Ex 2-level page table:
  + Level 1: each PTE points to a page table
  + Level 2: each PTE points to a page (paged In and out like any other data)

End-to-end address translation

Addressing example

* 14 bit virtual addresses
* 12 bit physical address
* Page size = 64 bytes (so we need logbase2(64) = 6 bits for page offset
* This leaves 8 bits in the virtual addresses for cache index, so there are 2^8 = 256 entries

**Example on slide 28**

* Virtual address: Get the vpn -> go to the thing with 256 entries on the other slide -> check if valid -> go to the TLB on slide 25 -> go to the set index -> search each entry at the set for the correct tag to get PPN
* Physical address (build it from the virtual address):
  + Get the PPO from the VPO of the virtual address
  + Get the PPN from the virtual address
  + To find the data stored at the physical address: -> go to cache on slide 27 -> go to set index -> check tag -> go to offset

Linux memory system

* Linux organizes VM as a collection of areas
* Pgd
  + Page global directory address
  + Points to L1 page table
* Vm\_prot
  + Read / write permissions for this area
* Vm\_flags
  + Pages shared with other processes or private to this process

Memory-related perils and pitfalls

1. Dereferencing bad pointers
   1. The classic scanf bug
      1. Were supposed to save the value to the address of where the variable is located, so we need a &
2. Reading uninitialized memory
   1. Assuming that heap data is initialized to zero
   2. Should have used calloc instead of malloc. We assumed y is initialized to zero, but that might not be the case with malloc, so calloc will initialize all indexes of y to zero. (y could have junk in it so we want to clear it)
3. Overwriting memory
   1. Allocating the possibly wrong sized object
   2. Off-by-one error
      1. We allocated N sized data, but the loop is accessing N+1 times
   3. Not checking the max string size
      1. Basis for classic buffer overflow attacks
   4. Misunderstanding pointer arithmetic
      1. We are increasing p by size of int (4 bytes) but we should be increasing it by size of pointer. So you can just do p++
   5. Referencing a pointer instead of the object it points to
      1. We want to remove the first item in a binary heap of \*size items and the reheapify the remaining \*size -1 items