Virtual Memory from 10,000 feet

- The basic abstraction that the OS provides for memory management is virtual memory (VM).
- VM enables programs to execute without requiring their entire address space to be resident in physical memory.
- Program can also execute on machines with less RAM than it "needs".
- Many programs don't need all of their code or data at once (or ever).
- e.g., branches they never take, or data they never read/write.
- No need to allocate memory for it, OS should adjust amount allocated based on its run-time behavior.
- Virtual memory isolates processes from each other.
- One process cannot name addresses visible to others; each process has its own isolated address space.
- VM requires hardware and OS support.
  - MMU's, TLB's, page tables, ...

Virtual Addresses

- To make it easier to manage memory of multiple processes, make processes use virtual addresses.
  - Virtual addresses are independent of location in physical memory (RAM) that referenced data lives.
  - OS determines location in physical memory.
  - Instructions issued by CPU reference virtual addresses.
  - E.g., pointers, arguments to load/store instruction, PC, ...
  - Virtual addresses are translated by hardware into physical addresses (with some help from OS).
- The set of virtual addresses a process can reference is its address space.
  - Many different possible mechanisms for translating virtual addresses to physical addresses.
  - Will take a historical walk through them, ending up with current techniques.

Memory Management

- We're beginning a new multiple-lecture topic.
  - Goals of memory management.
  - Convenient abstraction for programming.
  - Isolation between processes.
  - Allocate scarce memory resources between competing processes: Maximize performance (minimize overhead).
  - Mechanisms.
    - Physical vs. virtual address spaces.
    - Page table management, segmentation policies.
    - Page replacement policies.

In the beginning...

- First, there was batch programming.
  - Programs used physical addresses directly.
  - OS loads job, runs it, unloads it.
- Then came multiprogramming.
  - Need multiple processes in memory at once.
  - To overlap I/O and computation.
- Memory requirements:
  - Protection: Restrict which addresses processes can use, so they can't stomp on each other.
  - Fast translation: Memory lookups must be fast, in spite of protection scheme.
  - Fast context switching: When swap between jobs, updating memory hardware (protection and translation) must be quick.

Old technique #1: Fixed Partitions

- Physical memory is broken up into fixed partitions.
  - All partitions are equally sized, partitioning never changes.
  - Hardware requirement: Base register.
  - Physical address = virtual address + base register.
  - Base register loaded by OS when it switches to a process.
  - How can we ensure protection?
- Advantages:
  - Simple, ultra-fast context switch.
- Problems:
  - Internal fragmentation: Memory in a partition not used by its owning process isn't available to other processes.
  - Partition size problem: No one size is appropriate for all processes.
  - Fragmentation vs. fitting large programs in partition.
Fixed Partitions (K bytes)

Old technique #2: Variable Partitions
- Obvious next step: physical memory is broken up into variable-sized partitions
  - hardware requirements: base register, limit register
  - physical address = virtual address + base register
  - how do we provide protection?
    - if physical address = base + limit then...
- Advantages
  - no internal fragmentation
  - simply allocate partition size to be just big enough for process
    (assuming we know what that is!)
- Problems
  - external fragmentation
    - as we load and unload jobs, holes are left scattered throughout physical memory

Variable Partitions

Modern technique: Paging
- Solve the external fragmentation problem by using fixed sized units in both physical and virtual memory

User's Perspective
- Processes view memory as a contiguous address space from bytes 0 through N
  - virtual address space (VAS)
- In reality, virtual pages are scattered across physical memory frames
  - virtual-to-physical mapping
  - this mapping is invisible to the program
- Protection is provided because a program cannot reference memory outside of it’s VAS
  - the virtual address 0xDEADBEEF maps to different physical addresses for different processes

Paging
- Translating virtual addresses
  - a virtual address has two parts: virtual page number & offset
  - virtual page number (VPN) is index into a page table
  - page table entry contains page frame number (PFN)
  - physical address is PFN:offset
- Page tables
  - managed by the OS
  - map virtual page number (VPN) to page frame number (PFN)
  - VPN is simply an index into the page table
  - one page table entry (PTE) per page in virtual address space
  - i.e., one PTE per VPN
Paging

• assume 32 bit addresses
  - assume page size is 4KB (4096 bytes, or 2^{12} bytes)
  - VPN is 20 bits long (2^{20} VPNs), offset is 12 bits long

• let’s translate virtual address 0x13325328
  - VPN is 0x13325, and offset is 0x328
  - assume page table entry 0x13325 contains value 0x03004
  - VPN 0x13325 maps to PFN 0x03004
  - physical address = PFN::offset = 0x03004328

Page Table Entries (PTEs)

• PTE’s control mapping
  - the valid bit says whether or not the PTE can be used
  - says whether or not a virtual address is valid
  - it is checked each time a virtual address is used
  - the reference bit says whether the page has been accessed
  - if it is set when a page has been read or written to
  - the modify bit says whether or not the page is dirty
  - it is set when a write to the page has occurred
  - the protection bits control which operations are allowed
  - read, write, execute
  - the page frame number determines the physical page
  - physical page start address = PFN << (#bits/page)

Paging Advantages

• Easy to allocate physical memory
  - physical memory is allocated from free list of frames
  - to allocate a frame, just remove it from its free list
  - external fragmentation is not a problem!
  - complication for kernel contiguous physical memory allocation
  - many lists, each keeps track of free regions of particular size
  - regions’ sizes are multiples of page sizes
  - “buddy algorithm”

• Easy to “page out” chunks of programs
  - all chunks are the same size (page size)
  - use valid bit to detect references to “paged-out” pages
  - also, page sizes are usually chosen to be convenient multiples of disk block sizes

Paging Disadvantages

• Can still have internal fragmentation
  - process may not use memory in exact multiples of pages
• Memory reference overhead
  - 2 references per address lookup (page table, then memory)
  - solution: use a hardware cache to absorb page table lookups
  - translation lookaside buffer (TLB) – next class
• Memory required to hold page tables can be large
  - need one PTE per page in virtual address space
  - 32 bit AS with 4KB pages = 2^{20} PTEs = 1,048,576 PTEs
  - 4 bytes/PT = 4MB per page table
  - OS’s typically have separate page tables per process
  - 25 processes = 100MB of page tables
  - solution: page the page tables (!!!)
  - (ow, my brain hurts...more later)