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

Tools of memory management

- Address translation
- Base and limit registers
- Swapping
- Paging (and page tables and TLBs)
- Segmentation (and segment tables)
- Page faults => page fault handling => virtual memory
- The policies that govern the use of these mechanisms

Today’s desktop and server systems

- 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 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

A trip down Memory Lane …

- Why?
  - Because it’s instructive
  - Because embedded processors (98% or more of all processors) typically don’t have virtual memory

- First, there was job-at-a-time batch programming
  - programs used physical addresses directly
  - OS loads job (perhaps using a relocating loader to “offset” branch addresses), runs it, unloads it
  - what if the program wouldn’t fit into memory?
    - manual overlays!

- An embedded system may have only one program!
• Swapping
  – save a program’s entire state (including its memory image) to disk
  – allows another program to be run
  – first program can be swapped back in and re-started right where it was

• The first timesharing system, MIT’s “Compatible Time Sharing System” (CTSS), was a uni-programmed swapping system
  – only one memory-resident user
  – upon request completion or quantum expiration, a swap took place
  – bow wow wow … but it worked!

Then came multiprogramming
• multiple processes/jobs in memory at once
• to overlap I/O and computation
• memory management 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 the protection scheme
  – fast context switching: when switching between jobs, updating memory hardware (protection and translation) must be quick

Virtual addresses for multiprogramming
• To make it easier to manage memory of multiple processes, make processes use virtual addresses (which is not what we mean by "virtual memory" today!)
  – virtual addresses are independent of location in physical memory (RAM) where referenced data lives
    • OS determines location in physical memory
    • instructions issued by CPU reference virtual addresses
      • e.g., pointers, arguments to load/store instructions, PC ...
  – virtual addresses are translated by hardware into physical addresses (with some setup 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
  • we’ll take a historical walk through them, ending up with our current techniques
  • Note: We are not yet talking about paging, or virtual memory – only that the program issues addresses in a virtual address space, and these must be "adjusted" to reference memory (the physical address space)
    – for now, think of the program as having a contiguous virtual address space that starts at 0, and a contiguous physical address space that starts somewhere else

Old technique #1: Fixed partitions
• Physical memory is broken up into fixed partitions
  – partitions may have different sizes, but partitioning never changes
  – hardware requirement: base register, limit register
    • physical address = virtual address + base register
    • base register loaded by OS when it switches to a process
  – how do we provide protection?
    • if (physical address > base + limit) then…?

Advantages
• Simple

Problems
• internal fragmentation: the available partition is larger than what was requested
• external fragmentation: two small partitions left, but one big process – what sizes should the partitions be??
Old technique #2: Variable partitions

- Obvious next step: physical memory is broken up into partitions dynamically – partitions are tailored to programs
  - 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
    - slightly different than the external fragmentation for fixed partition systems

Mechanics of variable partitions

- Physical memory
- Base register
- Limit register
- Virtual address
- Offset
- Protection fault

Dealing with fragmentation

- Swap a program out
- Re-load it, adjacent to another
- Adjust its base register
- "Lather, rinse, repeat"
- Ugh

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 – not contiguous as earlier
  - virtual-to-physical mapping
  - this mapping is invisible to the program
- Protection is provided because a program cannot reference memory outside of its VAS
  - the virtual address 0xDEADBEEF maps to different physical addresses for different processes
- Note: Assume for now that all pages of the address space are resident in memory – no “page faults”

Address translation

- 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 (K byte pages)**

1. Page table
   - Page 0
   - Page 1
   - Page 2
   - Page 3

2. Virtual address space
   - Physical memory
   - Page frame 0
   - Page frame 1
   - Page frame 2
   - Page frame 3
   - Page frame 4
   - Page frame 5
   - Page frame 6
   - Page frame 7
   - Page frame 8
   - Page frame 9
   - Page frame 10

3. Page fault – next lecture!

**Mechanics of address translation**

- PTE's control mapping
  - Valid bit says whether or not the PTE can be used
  - Referenced bit says whether the page has been accessed
  - Modified bit says whether the page is dirty
  - Protection bits control which operations are allowed

**Page table entries (PTEs)**

- Page frame number determines the physical page
- Physical page start address = PFN

**Example of address translation**

- 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
  - Page frame number is 0x03004
  - Physical address = PFN:offset = 0x03004328

**Paging advantages**

- Easy to allocate physical memory
  - Physical memory is allocated from free list of frames
  - To allocate a frame, just remove it from the free list
  - External fragmentation is not a problem!
  - Managing variable-sized allocations is a huge pain in the neck
    - "Buddy system"
    - Problem still may exist in the kernel – e.g., page tables may need to be allocated contiguously

- Leads naturally to virtual memory
  - Entire program need not be memory resident
  - Take page faults using "valid" bit
  - All "chunks" are the same size (page size)
  - But paging was originally introduced to deal with external fragmentation, not to allow programs to be partially resident

**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/PTE = 4MB per page table
  - OS's have separate page tables per process
  - 25 processes * 100MB of page tables
  - Solution: page the page tables (!!!)
  - (Ow, my brain hurts...more later)
Segmentation
(We will be back to paging soon!)

- Paging
  - mitigates various memory allocation complexities (e.g., fragmentation)
  - view an address space as a linear array of bytes
  - divide it into pages of equal size (e.g., 4KB)
  - use a page table to map virtual pages to physical page frames
    - page (logical) => page frame (physical)
- Segmentation
  - partition an address space into logical units
    - stack, code, heap, subroutines, ...
  - a virtual address is <segment #, offset>

What’s the point?

- More “logical”
  - absent segmentation, a linker takes a bunch of independent modules that call each other and linearizes them
  - they are really independent; segmentation treats them as such
- Facilitates sharing and reuse
  - a segment is a natural unit of sharing – a subroutine or function
- A natural extension of variable-sized partitions
  - variable-sized partition = 1 segment/process
  - segmentation = many segments/process

Hardware support

- Segment table
  - multiple base/limit pairs, one per segment
  - segments named by segment #, used as index into table
    - a virtual address is <segment #, offset>
  - offset of virtual address added to base address of segment to yield physical address

Segment lookups

Pros and cons

- Yes, it’s “logical” and it facilitates sharing and reuse
- But it has all the horror of a variable partition system
  - except that linking is simpler, and the “chunks” that must be allocated are smaller than a “typical” linear address space
- What to do?

Combining segmentation and paging

- Can combine these techniques
  - x86 architecture supports both segments and paging
- Use segments to manage logical units
  - stack, heap, code module, file, ...
  - segments vary in size, but are typically large (multiple pages)
- Use pages to partition segments into fixed-size chunks
  - each segment has its own page table
    - there is a page table per segment, rather than per user address space
  - memory allocation becomes easy once again
    - no contiguous allocation, no external fragmentation
- Linux:
  - 1 kernel code segment, 1 kernel data segment
  - 1 user code segment, 1 user data segment
  - N task state segments (stores registers on context switch)
  - 1 "local descriptor table" segment (not really used)
  - all of these segments are paged
- Note: this is a very limited/boring use of segments!