Goals of memory management

- Allocate scarce memory resources among competing processes, maximizing memory utilization and system throughput
- Provide a convenient abstraction for programming (and for compilers, etc.)
- Provide isolation between processes
  - we have come to view “addressability” and “protection” as inextricably linked, even though they’re really orthogonal

Tools of memory management

- Base and limit registers
- Swapping
- Paging (and page tables and TLBs)
- Segmentation (and segment tables)
- 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
  – how 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
  – all partitions are equally sized, 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: memory in a partition not used by its owning process isn’t available to other processes
  – external fragmentation: two small partitions left, but one big job – what sizes should the partitions be??

Mechanics of fixed partitions

physical memory

<table>
<thead>
<tr>
<th>partition 0</th>
<th>partition 1</th>
<th>partition 2</th>
<th>partition 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2K</td>
<td>4K</td>
<td>6K</td>
</tr>
<tr>
<td>2K</td>
<td>P2’s base: 6K</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• offset
• limit register
• base register
• protection fault

virtual address
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
    - Slightly different than the external fragmentation for fixed partition systems

Mechanics of variable partitions

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"
Mechanics of address translation

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
    • 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
  – the referenced bit says whether the page has been accessed
    • It is set when a page has been read or written to
  – the modified 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

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”
  – Leads naturally to virtual memory
    • entire program need not be memory resident
    • take page faults using “valid” bit
    • 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 typically have separate page tables per process
    • 25 processes = 100MB of page tables
  – solution: page the page tables (TTM)
    • (low, 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
  – 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)
  – “local descriptor table” segment (not really used)
  – all of these segments are paged
• Note: this is a very limited/boring use of segments!