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
• Segmentation (and segment tables)
• Paging (and page tables and TLBs)
• Page fault handling
• Swapping
• 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 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
• Virtual memory requires hardware and OS support
  – MMU’s, TLB’s, page tables, page fault handling, …
• Typically accompanied by swapping, and at least limited segmentation
A trip down Memory Lane …

• Why?
  – Because it’s instructive
  – Because embedded processors (98% 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
  – 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 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 switch 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
  – 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, …
  – 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
    • 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
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

- Advantages
  - Simple

- 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
Mechanics of fixed partitions

- Virtual address
  - Offset
- Base register
  - 3K

Physical memory
- Partitions 0 to 5
- 0, 1K, 2K, 3K, 4K, 5K
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
Mechanics of variable partitions

- offset
- virtual address
- limit register
- P3’s size
- base register
- P3’s base
- yes
- no
- raise protection fault
- physical memory
- partition 0
- partition 1
- partition 2
- partition 3
- partition 4
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
  - 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
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
Mechanics of address translation

virtual address

virtual page #
offset

page table

page frame #

physical address

physical memory

page frame 0
page frame 1
page frame 2
page frame 3

...

page frame Y
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)

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>1</th>
<th>2</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>R</td>
<td>M</td>
<td>prot</td>
<td>page frame number</td>
</tr>
</tbody>
</table>

- **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 **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 is not 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 (!!!)
    • (ow, my brain hurts…more later)