Purpose of This Module

• Most/all of this material was covered in CSE 378 or CSE 351
• These slides just provide review, plus perhaps some specific context you may not have seen before
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)
  – Efficient use of hardware (real memory)
    • VM enables programs to execute without requiring their entire address space to be resident in physical memory
    • many programs don’t need all of their code or data at once (or ever)
      – no need to allocate memory for it, OS should adjust amount allocated based on **run-time** behavior
  – Program flexibility
    • programs can execute on machines with less RAM than they “need”
      – On the other hand, paging is really, really slow...
  – Protection
    • virtual memory **isolates** address spaces from each other
VM Requires Hardware and OS Support

- 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

- Note: hardware is 64-bit, but software is still (mainly) 32-bit
  - Limits the size of the virtual address space of any individual process to 4GB
A Brief History of Memory Management

• Why?
  – Because it’s instructive
  – Because embedded processors (98% or more of all processors) typically don’t have virtual memory
  – Because some aspects are pertinent to allocating pieces of the virtual address space
    • i.e., e.g., malloc()

• 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!
Uniprogramming

• 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!

• 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
  – At least it worked...
Multiprogramming

• Then came multiprogramming
  – multiple processes/jobs in memory at once
    • to overlap I/O and computation

• Multiprogramming 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
  – 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 translated 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: the fixed size partition is larger than what was requested
  - external fragmentation: two small partitions left, but one big job – what sizes should the partitions be??
Mechanics of fixed partitions

offset
virtual address

<?

yes

no

raise protection fault

limit register
2K

base register
P2’s base: 6K

physical memory

partition 0

partition 1

partition 2

partition 3

0

2K

4K

6K

8K
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

Physical memory

- partition 0
- partition 1
- partition 2
- partition 3
- partition 4

Virtual address

- offset
  - limit register: P3’s size
  - base register: P3’s base

- virtual address
  - offset
  - yes
  - <?
  - no
  - raise protection fault

- offset
  - yes
  - +

- offset
  - no
  - raise protection fault
Dealing with fragmentation

• Compact memory by copying
Modern technique: Paging

- Solve the external fragmentation problem by using fixed sized units in both physical and virtual memory
- Solve the internal fragmentation problem by making the units small
Life Is Easy...

- For developers:
  - Processes view memory as a contiguous address space from bytes 0 through N
  - N is independent of the actual hardware

- For the memory manager (OS):
  - Efficient use of memory, because very little internal fragmentation
  - Efficient use of the system because no external fragmentation at all
    - No need to copy big chunks of memory around to coalesce free space

- For the protection system (OS):
  - One process cannot name another process's memory, so there is complete isolation
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
  – 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

page frame #
offset

physical memory

page frame 0

page frame 1

page frame 2

page frame 3

...

page frame Y
PTE's: An Opportunity

• So long as there's a PTE lookup per memory reference, we might as well add some functionality
  
  – We can add protection
    • A virtual page can be read-only, and result in a fault if a store to it is attempted
    • Some pages may not map to anything
      – E.g., page 0
  
  – We can add some “accounting information”
    • Can't do anything fancy, as address translation has to be fast
    • Can keep track of whether or not a virtual page is being used, though
      – (This is intended primarily to help the paging algorithm, once we get to paging)
Page Table Entries (PTEs)

- 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 Pros/Cons

• Pros:
  – Easy to allocate physical memory
  – Leads naturally to virtual memory

• Cons:
  – Address translation time
    • 2 references per load/store
      – Solution: caching
  – Page tables can be large:
    • 32-bit AS w/ 4KB pages = $2^{20}$ PTEs = 1,048,576 PTEs
    • 64-bit address space: !!!
Segmentation
(We will be back to paging soon!)

• Paging
  – view an address space as a linear array of bytes

• Segmentation
  – partition an address space into *logical* units
    • E.g., 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

虚地址

段 #
偏移量

段表

限制
基地址

物理内存

段 0
段 1
段 2
段 3
段 4

是否

保护异常
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

<table>
<thead>
<tr>
<th>Segment #</th>
<th>Page #</th>
<th>Offset within page</th>
</tr>
</thead>
</table>

  no contiguous allocation, no external fragmentation

  Offset within segment
• Linux:
  – 1 kernel code segment, 1 kernel data segment
  – 1 user code segment, 1 user data segment
  – all of these segments are paged

• Note: this is a very limited/boring use of segments!