Goals of memory management

• Allocate memory resources among competing processes, maximizing memory utilization and system throughput

• Provide isolation among processes
  – We have come to view “addressability” and “protection” as inextricably linked, even though they’re really orthogonal

• Provide a convenient abstraction for programming (and for compilers, etc.)
Tools of memory management

- Base and limit registers
- Swapping
- Paging (and page tables and TLB’s)
- Segmentation (and segment tables)
- Page faults => page fault handling => virtual memory
- Access rights (R/W/X)
- The policies that govern the use of these mechanisms
Today’s server, desktop, laptop, tablet, and phone systems

• The basic abstraction that the OS provides for memory management is virtual memory (VM)
  
  – Efficient use of hardware (real memory)
    • Many programs don’t need all of their code or data at once (or ever – branches they never taken, or data they never read/write)
    • No need to allocate memory for it, OS should adjust amount allocated based on run-time behavior
    • Appropriate allocation strategies “maximize” the number of instructions executed between page faults → “maximize” work done between I/O events
  
  – Program flexibility/portability
    • Programs can execute on machines with less RAM than they “need”
      – On the other hand, paging is really slow, so must be minimized!
  
  – Protection
    • Virtual memory isolates address spaces 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, page fault handling, …

• Typically accompanied by swapping, and at least limited segmentation
Brief History of Memory Protection

• 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 portions of a virtual address space – 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” pointer addresses), runs it, unloads it
  – what if the program wouldn’t fit into memory?
    • manual overlays!

• An embedded system may have only one program!
• **Real Memory / 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
  - fragmentation issues…

• **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 between processes/jobs, easing the task of the application programmer
  – 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
Translated addresses for multiprogramming

• To make it easier to manage memory of multiple processes, use **address translation**
  – at **compile time** it is assumed that code will execute at some particular address
    • e.g., assume it will be loaded at address 0
    • call the addresses used at compile time **virtual addresses**
  – at **load time**, load executable wherever there is free memory
    • probably not the assumed address!
  – at **run time**, translate the addresses issued by the code to correct physical addresses
    • e.g., add an offset (the starting address at which the code was loaded)
Address Space

• 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 some of 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 another address space (the physical address space)
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/limit registers set by OS when switching to a process
  - how do we provide protection?
    - if (physical address > base + limit) then… ?

- Advantages
  - Simple

- Problems
  - Must allocate contiguous physical memory
    - Why?
  - internal fragmentation: the available partition is larger than what was requested
  - external fragmentation: two small partitions free, but one big job
Mechanics of fixed partitions

offset
virtual address

<?

yes

no

raise protection fault

base register
P2’s base: 6K

limit register
2K

physical memory

partition 0

partition 1

partition 2

partition 3

0

2K

6K

8K

12K
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

offset\rightarrow\langle?\rangle\rightarrow\text{yes}\rightarrow+\rightarrow\text{physical memory}

\text{virtual address}\rightarrow\langle?\rangle\rightarrow\text{no}\rightarrow\text{raise protection fault}

\text{limit register}\rightarrow\text{P3’s size}\rightarrow\text{base register}\rightarrow\text{P3’s base}

\text{partition 0}\rightarrow\text{partition 1}\rightarrow\text{partition 2}\rightarrow\text{partition 3}\rightarrow\text{partition 4}
Dealing with fragmentation

- Compact memory by copying
  - Swap a program out
  - Re-load it, adjacent to another
  - Adjust its base register
  - Ugh
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 the programmer …
  – Processes view memory as a contiguous address space from bytes 0 through N – a **virtual address space**

• For the memory manager …
  – In reality, virtual pages are scattered across physical memory frames – not contiguous as earlier
    • Just need any free frame, not contiguous free frames
  – Efficient use of memory, because very little internal fragmentation
  – No external fragmentation at all
    • No need to copy big chunks of memory around to coalesce free space
• For the protection system
  – So long as the OS is careful about setting up the address translation registers, one process cannot “name” another process’s physical memory – there is complete isolation
    • The virtual address 0x01234567 maps to different physical addresses for different processes

Note: All the above is true even if we require the entire virtual address space to be loaded in physical memory – no paging
Address translation: paging

• Translating virtual addresses
  – a virtual address has two parts: virtual page number & offset within that page
  – virtual page number (VPN) is used to identify which physical frame holds the data
    • index into a page table that maps virtual page number to physical page frame number (PFN)
  – the offset in the physical space is the same as the offset in the virtual space
  – physical address is PFN::offset
Paging (K-byte pages)

Page table

<table>
<thead>
<tr>
<th>page</th>
<th>frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

Virtual address space

<table>
<thead>
<tr>
<th>page</th>
<th>frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

Physical memory

<table>
<thead>
<tr>
<th>page frame</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
</tbody>
</table>

Page fault – next lecture!
Mechanics of address translation

virtual address

virtual page # | offset

page table

page frame #

physical address

page frame # | offset

physical memory

frame 0

frame 1

frame 2

frame 3

frame Y
Example of paged 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
    • equivalently, physical address is PFN * (sizeof(frame)) + offset
Architecture requirements

• Address translation “must” be done in hardware
  – Why?

• How many memory accesses does it take for hardware to translate an address?
  – What do you do about that?

• How much memory is consumed by the page table?
  – What do you do about that?
Page Table Entries – an opportunity!

- As long as there’s hardware that does a PTE lookup per memory reference, we might as well add some functionality
  - 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 – a fault will occur if a reference is attempted
  - usage information
    - Can’t do anything fancy, since address translation must be fast
    - Can keep track of whether or not a virtual page is being referenced
      - This will help the paging algorithm, once we get to paging
Page Table Entries (PTE’s)

- **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
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

- (Close to) No Fragmentation
  - No external fragmentation
  - Internal fragmentation only on “last page”

- (Leads naturally to virtual memory
  - entire program need not be memory resident
  - but paging was originally introduced to deal with external fragmentation, not to allow programs to be partially resident)
Paging disadvantages

• Still some internal fragmentation
  – Process may not use memory in exact multiples of pages
  – But minor because of small page size relative to address space size

• Translation 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
  – What if addresses are 64 bits?
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 \(<\text{segment }\#, \text{ offset}>\)
  – offset of virtual address added to base address of segment to yield physical address
Segment lookups

- **virtual address**: segment #, offset

- **segment table**: limit, base

- **physical memory**:
  - segment 0
  - segment 1
  - segment 2
  - segment 3
  - segment 4

- **Responses**:
  - yes
  - no

- **Actions**:
  - raise protection fault

© 2017 Gribble, Lazowska, Levy, Zahorjan, Zbikowski
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
  - modern architectures support 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
  – all of these segments are paged

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