Memory Management: Main Memory

- It's all about bookkeeping
- The von Neumann model: fetch instructions from memory; decode instructions; possibly read or write data to or from memory; "rinse and repeat"
- In this model, we abstract memory completely as a linear, addressed array — the reality is more complex
- Common hardware elements: memory hierarchy (register/cache/main memory); hardware protection through *base* and *limit* registers

Address Binding

- When does a memory address in code become truly an address in physical memory?
- As in programming languages and compiler construction, "binding" refers to when a decision is made about a particular entity — in this case, memory
- *Compile-time binding* sets an absolute address in code
- *Load-time binding* sets addresses once, on load — code is therefore said to be *relocatable*
- *Execution-time binding* determines an address as instructions are executed — needs hardware support

Logical vs. Physical Addresses

- To facilitate execution-time binding, we must differentiate between addresses generated by the CPU (the *logical* or *virtual* address) from the actual address that gets accessed (the *physical* address)
- Current systems do this in hardware; that is, this *mapping* is not performed by the CPU but by a separate *memory management unit* (MMU)
- User code never sees the physical address — the process is completely transparent

Loading, Linking, and Sharing

Additional wrinkles before going deeper into main memory management:

- Programs seldom need all of their code in memory all the time, so *dynamic loading* is useful — only load code into memory as you need it
- Reusable code (e.g., system libraries) should not have to exist in every program (*static linking*); *dynamic linking* enables one in-memory copy for multiple processes
- *Shared libraries* extend dynamic linking to allow multiple versions of the same library to co-exist

Memory Allocation Basics

As with most operating system techniques, memory allocation started simple and grew in sophistication:

- Since multiprogramming really only runs a single process at a time, *swapping* has been used to keep processes in a *backing store* outside main memory
- *Contiguous memory allocation* devotes entire *partitions* of memory to processes; start with just the OS at the bottom at the stack at the top, with a large *hole* in the middle, then divide that hole among processes using some algorithm (e.g., *first fit*, *best fit*, *worst fit*)

Fragmentation

- A key issue with contiguous memory allocation that remains relevant today is *fragmentation* — essentially, wasted memory (allocated but unused)
- When there is enough total available memory but no single hole is large enough for a pending request, we have *external* fragmentation
- Since it is impractical to allocate partitions down to-the-byte, we usually have a larger minimum allocation unit (or block) — when not all of that block is used by a process, we have *internal* fragmentation

Paging

- *Paging* is one of two complementary memory management techniques that are widely used in current systems (the other is *segmentation*, which we will discuss next)
- The main trick in paging: add a level of indirection between logical and physical addresses so that a process's address space is (transparently) no longer physically contiguous
- Logical memory is divided into *pages*, and physical memory is divided into *frames* of the same size

- Logical addresses are now partitioned into a *page number* (or *p* for "page") and a *page offset* (or *d* for "delta" or "displacement") — since we're talking bits, it should be easy to see that page/frame sizes are powers of 2, and for an *m*-bit logical address space, a page size of 2^n results in 2^{m-n} possible pages
- Every process gets a *page table* that maps a logical page to a physical frame — whenever the CPU uses a logical address, the page value of that address is converted into the corresponding frame value, as indicated by the page table; this is the final, physical address
- Behind the scenes, the OS tracks which frame corresponds to which page of which process, using a *frame table* data structure

Hardware Support for Paging

- Page-to-frame translation is done in hardware — it is part of a new CPU or architecture's specifications
- For small page tables, hardware used to provide a dedicated set of registers
- With today's huge memory spaces, the page tables themselves stay in main memory, with a *page table base register* (PTBR) pointing to it
- But main memory is slow, so we add a *translation look-aside buffer* (TLB) — essentially a page table cache

Page Protection and Sharing

Paging can be used beyond just page-to-frame mapping to provide additional memory management features

- *Memory protection*: One or more bits can indicate read-only, read-write, or execute-only pages; another *valid-invalid* bit can indicate whether or not a page may be accessed by a process at all
- *Shared pages*: Shared libraries can be implemented via paging by letting multiple page tables point to the same frames — the ones with *reentrant* or *pure code*, such as what we would find in system libraries

Page Table Implementation: Hierarchical Paging

- Today's large address spaces — 32 going on 64 bits and more — make a single-level page table impractical: either too many entries or pages that are too large (resulting in lots of internal fragmentation)
- Hierarchical paging "pages the pages" — page table becomes a tree of page entries, resulting in multiple levels of indirection
- 2-level paging is sensible for 32-bit spaces (e.g., 10/10/12 results in 1024-entry tables with 4K pages); we need more for 64-bit

Hashed, Clustered, and Inverted Page Tables

- An alternative to a hierarchical page table is a *hashed page table* — a standard hash table based on the page number, leading to a linked list of page table entries (for all pages that hash to the same value)
- Another alternative is a *clustered page table*, where a single hash table entry actually consists of *multiple* page-to-frame mappings
- Finally, we have an *inverted page table* — instead of having one page table per process, we just keep one overall table, storing the process that "has" each frame

Segmentation

- Complementary to paging, *segmentation* divides a process's address space by *function* or *role*, such as the stack, certain libraries, static variables, etc.
- Instead of a single linear array, a logical address space is now a collection of named *segments*; an individual address is now a *tuple* instead of a single value
- Translation to a physical address (which, of course, is still a linear space) is accomplished through a *segment table*: for each segment, the table stores a *segment base* address and a *segment* [size] *limit*

- The segment component of the logical address indicates the entry in the segment table, with the second component being the displacement (or offset, or delta) of the address within that segment
- Thus, the final physical address is *segment base* + *displacement*, with *displacement* < *segment limit* being required or else we get an addressing error trap
- Note how segmentation and paging can be combined — for example, in the Intel Pentium family, logical addresses use segmentation, but the linear addresses that they generate can be subsequently viewed as page + displacement addresses, to be converted one more time into the final physical address

Main Memory Examples

- Intel Pentium D (IA-32)
  - Up to 16K segments per program of up to 4G in size, with 6 segment registers available (1 code, 1 stack, 4 data)
  - Paging may be enabled, with page sizes of 4K, 2M, or 4M, with separate TLBs for instructions and data
  - In 64-bit mode, segmentation is disabled, and the address space is completely flat/linear

- IBM PowerPC 970fx (a.k.a. "G5")
  - Natively 64-bit processor, with 32-bit compatibility
  - Code can access a full 64-bit address space, which the MMU maps to a 42-bit physical address space
  - Segmentation is optimized with a *segment lookaside buffer* (SLB)
  - Page sizes of 4K or 16M with a unified TLB (combined instructions and data)

Memory-Related Commands

- Unlike process management, *man -k memory* doesn't quite yield the same number of hits as for processes
- In fact, memory management is so closely bound to process management that commands such as *ps* and *top* frequently yield most of the memory-related information that you need
- Aside from that, there's *vmstat* (*vm_stat* in Mac OS X) for virtual memory statistics and *pmap* (*vmmap* in Mac OS X) for looking at individual processes (identified via the *pid*, of course)

- Mac OS X comes with some heap-related tools: *heap* lists a process's *malloc*ed blocks, while *leaks* lists the blocks that don't appear referenced at that moment
- Returning to *man -k memory* though, now is a good time to point out how *man* pages are divided into *sections* — these are the numbers that appear after *man* page names
  - The Linux *man* page for *man* (sounds funny, but that's how you say it) includes a table of what each section number typically contains
  - Sometimes, multiple sections contain pages of the same name; preceding the page name with the section number helps avoid confusion (e.g., *man 3 free*)