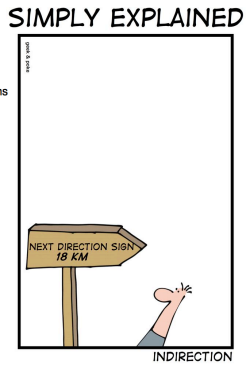
Today

- Virtual Memory
  - Optimizations

Indirection

A famous aphorism of David Wheeler goes: "All problems in computer science can be solved by another level of indirection";[1] this is often deliberately mis-quoted with "abstraction layer" substituted for "level of indirection". Kevlin Henney's corollary to this is, "...except for the problem of too many layers of indirection."



Review: Memory Management

- In general, what is the memory abstraction that the OS provides users?
- How does the OS allow multiprogramming?
- We talked about two main techniques that allow non-contiguous memory allocation
  - What is non-contiguous memory allocation?
  - Why would we want non-contiguous memory allocation?
  - What are those two techniques?

The Big Picture: Virtual Memory

How can the OS build the abstraction of a private, potentially large address space for multiple running processes (all sharing memory) on top of a single, physical memory?

Virtualizing Memory

- Logical to Physical Address mappings
  - Running program thinks it's running at 0
  - When you print out addresses in programs, those are the logical addresses
- Protection
  - A process can only access certain parts of memory
- Swapping
  - Swap out running processes' memory
  - Cost becomes prohibitive as size of process's memory increases

Review: Noncontiguous Memory

- Idea: if there is available memory, let's use it!
- Segmentation
  - Break process's memory into logical chunks
  - Base and limit for each offset
- Paging
  - Partition memory into small equal-size chunks
  - TLB – keep track of mapping from virtual addresses to physical addresses

Background

- Code needs to be in memory to execute
- Entire program code not needed at same time
- Consider ability to execute *partially-loaded* program
  - How is this possible?
  - What is the impact?

Background

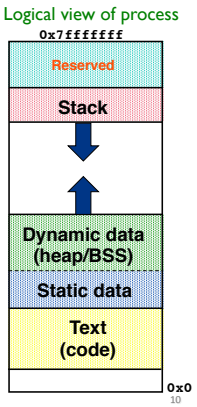
- Code needs to be in memory to execute, BUT entire program rarely used
  - Error code, unusual routines, larger-than-necessary data structures
- Entire program code not needed at same time
- Consider ability to execute partially-loaded program
  - Program no longer constrained by limits of physical memory
  - Each program takes less memory while running → more programs run at the same time
    - Increased CPU utilization and throughput with no increase in response time or turnaround time
  - Less I/O needed to load or swap programs into memory → each user program runs faster

Virtual Memory

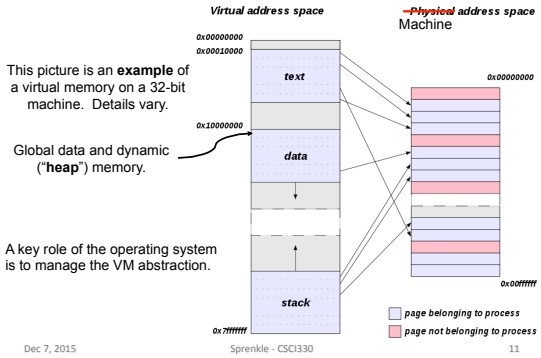
- **Idea**: use physical memory to hold only the portions of each executing process that are currently being used
  - Only part of the program needs to be in memory for execution
  - Parts of executing process that are not currently being used are held on secondary storage until needed.
- **Impact**:
  - Logical address space can be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Less I/O needed to load or swap processes

Virtual Memory

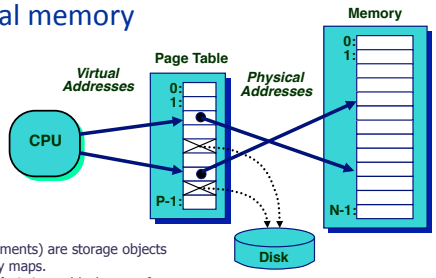
- Virtual address space
  - Logical view of how process is stored in memory
    - Usually start at address 0, contiguous addresses until end of space
  - Physical memory organized in page frames
  - MMU must map logical to physical
- Can be implemented via:
  - **Demand** paging
  - **Demand** segmentation



VM Page Maps



Virtual memory

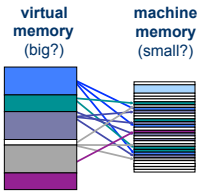


VMs (or segments) are storage objects described by maps.
A **page table** is just a block map of one or more VM segments in memory.
The hardware hides the indirection from user programs.

Virtual addressing

Code running on a core addresses memory through **virtual addresses**.

The machine translates virtual addresses via an in-memory **page table**.



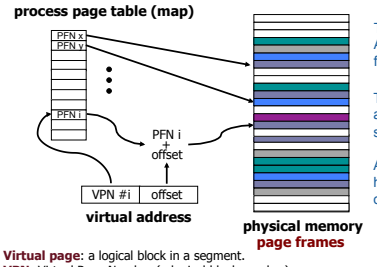
The machine allows a user process to access memory only by a valid **translation** in the page table.

The OS controls the contents of the page table.

The page table represents a functional mapping of virtual pages (VPNs) to page frames (PFNs) for **resident** pages.
If a page is not resident in memory, then its page table entry is marked as invalid.

The specific mechanisms for virtual address translation are **machine-dependent**.

Cartoon view of a page table



This is an **example**. Any PFN may be used for any VPN.

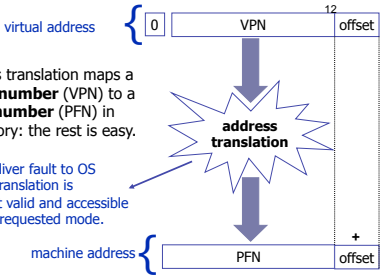
The map itself is just another data structure stored in memory.

A protected CPU register holds the machine address of the current map.

Virtual page: a logical block in a segment.
VPN: Virtual Page Number (a logical block number).
Page frame: a physical block in machine memory.
PFN: Page Frame Number (a block pointer).
PTE: Page Table Entry (an entry in the block map).

Virtual Address Translation

Example only: a typical 32-bit architecture with 4KB pages.

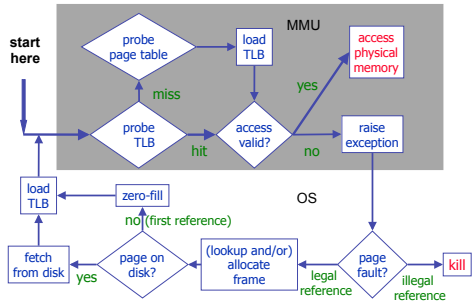


Virtual address translation maps a **virtual page number** (VPN) to a **page frame number** (PFN) in machine memory: the rest is easy.

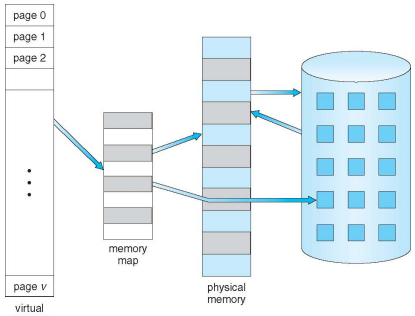
Deliver fault to OS if translation is not valid and accessible In requested mode.

Addresses Phil's Question

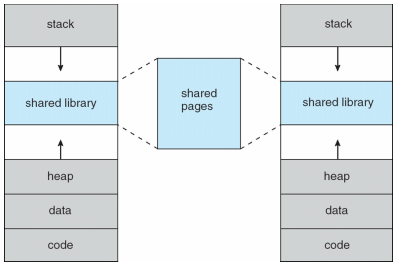
Virtual Addressing: Under the Hood



Virtual Memory Larger Than Physical Memory



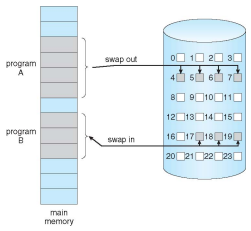
Shared Library Using Virtual Memory



Two logical pages pointing to same physical page.

Demand Paging

- Could bring entire process into memory at load time
- **Or**, bring page into memory *only* when it is needed
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users
- Similar to paging system with swapping
- Page needed ⇒ reference to it
  - not-in-memory ⇒ bring to memory
- Lazy swapper: never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a pager



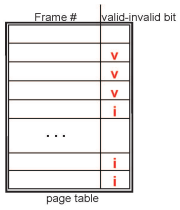
Basic Concepts

- With swapping, pager guesses which pages will be used before swapping out again
- Instead, pager brings in only those pages into memory
- How to determine that set of pages?
  - Need new MMU functionality to implement demand paging
- If needed pages are already memory resident,
  - No difference from non-demand-paging
- If page needed and not memory resident
  - Need to detect and load the page into memory from storage
    - Without changing program behavior
    - Without programmer needing to change code

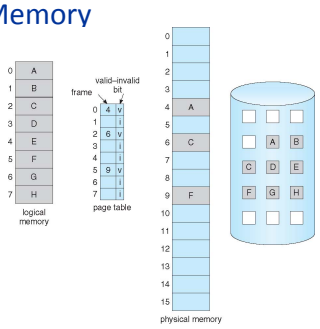
Valid-Invalid Bit

- a valid–invalid bit is associated with each page table entry
  - *v* ⇒ in-memory – memory resident
  - *i* ⇒ not-in-memory
- Initially valid-invalid bit is set to *i* on all entries

Page table snapshot:



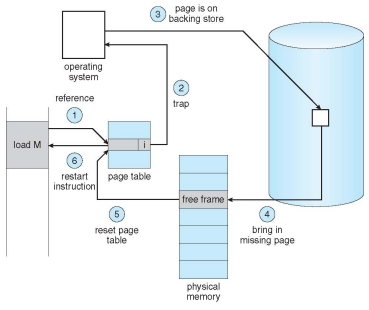
Page Table When Some Pages Are Not in Main Memory



Page Fault

- If there is a reference to a page, first reference to that page will trap to operating system: ***page fault***
- Operating system looks at another table to decide:
  - Invalid reference ⇒ abort
  - Just not in memory
- Find free frame
- Swap page into frame via scheduled disk operation
- Reset tables to indicate page now in memory
  - Set validation bit = v
- Restart the instruction that caused the page fault

Steps in Handling a Page Fault



Aspects of Demand Paging

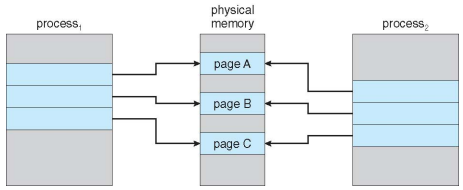
- Extreme case – start process with no pages in memory
  - OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
  - And for every other process pages on first access
  - Pure *demand* paging
- A given instruction could access multiple pages → multiple page faults
  - Consider fetch and decode of instruction
    - adds 2 numbers from memory and stores result back to memory
  - Cost decreased because of locality of reference
- Hardware support needed for demand paging
  - Page table with valid / invalid bit
  - Secondary memory (swap device with swap space)
  - Instruction restart

DEMAND PAGING OPTIMIZATIONS

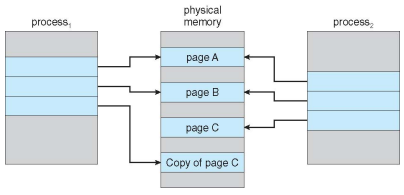
Copy-on-Write

- Allows both parent and child processes to initially share the same pages in memory
  - If either process modifies a shared page, only then is the page copied
- Allows more efficient process creation as only modified pages are copied

Before Process 1 Modifies Page C



After Process 1 Modifies Page C



What Happens if There is No Free Frame?

- Used up by process pages
- Also in demand from the kernel, I/O buffers, etc
- How much to allocate to each?
- Page replacement – find some page in memory, but not really in use, page it out
  - Algorithm – terminate? swap out? replace the page?
  - Performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times

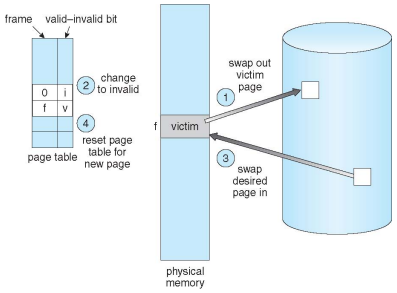
Page Replacement

- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement
- Use modify (dirty) bit to reduce overhead of page transfers – only modified pages are written to disk
- Page replacement completes separation between logical memory and physical memory
  - large virtual memory can be provided on a smaller physical memory

Basic Page Replacement

1. Find the location of the desired page on disk
2. Find a free frame
   1. If there is a free frame, use it
   2. If there is no free frame, use a page replacement algorithm to select a **victim frame**
   3. Write victim frame to disk if dirty
3. Bring the desired page into the (newly) free frame; update the page and frame tables
4. Continue the process by restarting the instruction that caused the trap

Page Replacement



Page-Buffering Algorithms

- Keep a pool of free frames, always
  - Frame available when needed, not found at fault time
  - Read page into free frame and select victim to evict and add to free pool
  - When convenient, evict victim
- Possibly, keep list of modified pages
  - When backing store otherwise idle, write pages there and set to non-dirty
- Possibly, keep free frame contents intact and note what is in them
  - If referenced again before reused, no need to load contents again from disk
  - Generally useful to reduce penalty if wrong victim frame selected

Applications and Page Replacement

- All of these algorithms have OS guessing about future page access
- Some applications have better knowledge – i.e. databases
- Memory intensive applications can cause double buffering
  - OS keeps copy of page in memory as I/O buffer
  - Application keeps page in memory for its own work
- Operating system can given direct access to the disk, getting out of the way of the applications
  - Raw disk mode
- Bypasses buffering, locking, etc

Thrashing

- If a process does not have "enough" pages, page-fault rate is very high
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back
  - This leads to:
    - Low CPU utilization
    - Operating system thinking that it needs to increase the degree of multiprogramming
    - Another process added to the system
- Thrashing ≡ a process is busy swapping pages in and out

Demand Paging and Thrashing

- Why does demand paging work?
  Locality model
  - Process migrates from one locality to another
  - Localities may overlap
- Why does thrashing occur?
  Σ size of locality > total memory size
  - Limit effects by using local or priority page replacement

Allocation Summary

- Variable partitioning is a pain
  - But, we need it for heaps and for other cases (e.g., address space layout).
- But for files we can break the objects down into "pieces".
  - When access to files is through an API, we can add some code behind that API to represent the file contents with a dynamic linked data structure (a map).
  - If the pieces are fixed-size (called pages or logical blocks), we can use fixed partitioning to allocate the underlying storage, which is efficient and trivial.
  - With that solution, internal fragmentation is an issue, but only for small objects. (Why?)
- That approach can work for VM segments too
  - have VM hardware to support it since the 1970s

Looking Ahead

- Project 5 due Friday
- Exam envelopes – due Friday
- Exam prep document – out later today