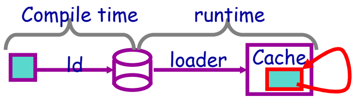
Memory Management

How to create a process?

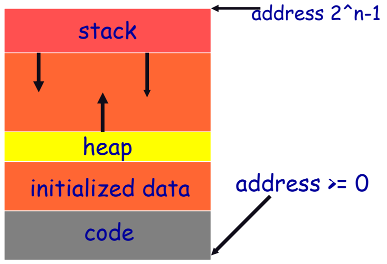
- On Unix systems, executable read by loader



- Compiler: generates one object file per source file
- Linker: combines all object files into one executable
- Loader: loads executable in memory

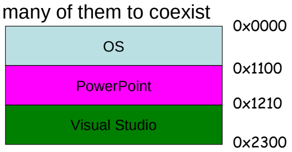
What does process look like?

Process divided into segments, each has:
- Code, data, heap (dynamic data) and stack (procedure calls)



Processes in Memory

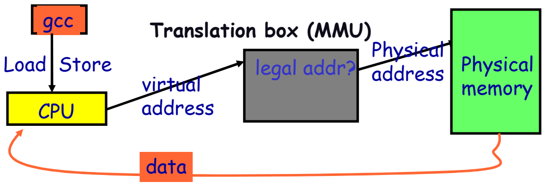
- We want many of them to coexist



- Issues:
  - PowerPoint needs more memory than there is on machine?
  - What is visual studio tries to access 0x0005?
  - If PowerPoint is not using enough memory?
  - If OS needs to expand?

Issues

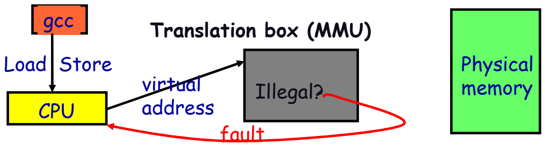
- Protection: Errors in process should not affect others
- Transparency: Should run despite memory size/location



How to do this mapping?

Issues

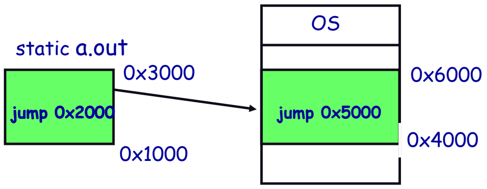
- Protection: Errors in process should not affect others
- Transparency: Should run despite memory size/location



How to do this mapping?

Scheme 1: Load-time Linking

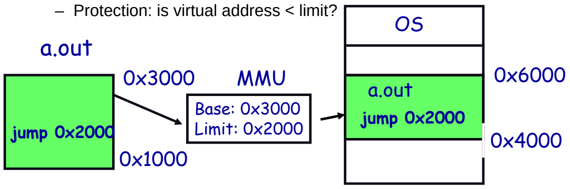
- Link as usual, but keep list of references
- At load time: determine the new base address
  - Accordingly adjust all references (addition)



- Issues: handling multiple segments, moving in memory

Scheme 2: Execution-time Linking

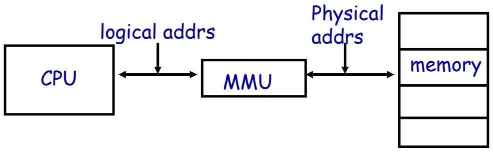
- Use hardware (base + limit reg) to solve the problem
  - Done for every memory access
  - Relocation: physical address = logical (virtual) address + base
  - Protection: is virtual address < limit?



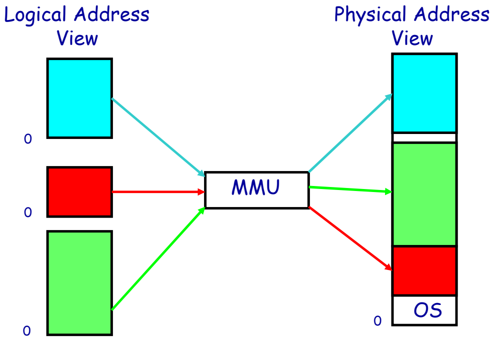
  - When process runs, base register = 0x3000, bounds register = 0x2000. Jump addr = 0x2000 + 0x3000 = 0x5000

Dynamic Translation

- Memory Management Unit in hardware
  - Every process has its own address space

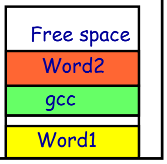


Logical and Physical Addresses



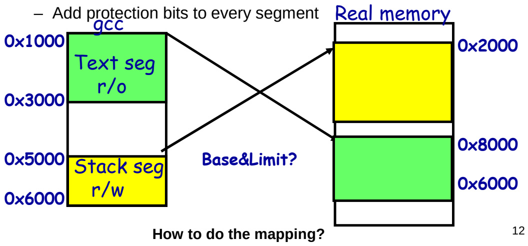
Scheme 2: Discussion

- Pro:
  - cheap in terms of hardware: only two registers
  - cheap in terms of cycles: do add and compare in parallel
- Con: only one segment
  - prob 1: growing processes. How to expand gcc?
  - prob 2: how to share code and data?? how can Word copies share code?
  - prob 3: how to separate code and data?
- A solution: multiple segments
  - **"segmentation"**



Segmentation

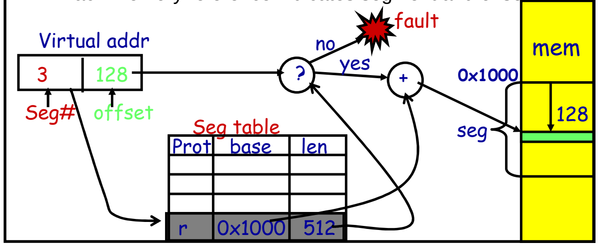
- Processes have multiple base + limit registers
- Processes address space has multiple segments
  - Each segment has its own base + limit registers
  - Add protection bits to every segment



How to do the mapping?

Mapping Segments

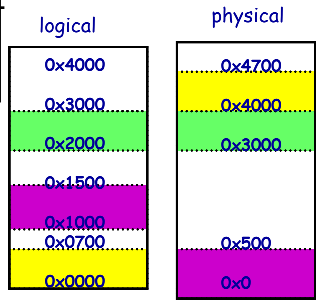
- Segment Table
  - An entry for each segment
  - Is a tuple <base, limit, protection>
- Each memory reference indicates segment and offset



Segmentation Example

- If first two bits are for segments, next 12 for offset

Seg	base	bounds	rw
0	0x4000	0x6ff	10
1	0x0000	0x4ff	11
2	0x3000	0xfff	11
3			00



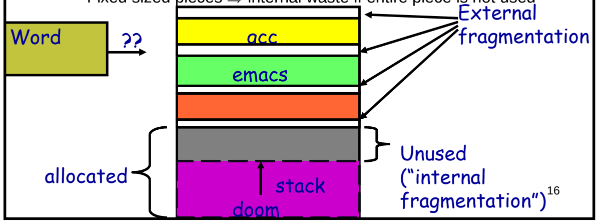
- where is 0x0240?
- 0x1108?
- 0x265c?
- 0x3002?
- 0x1700?

Segmentation: Discussion

- Advantages:
  - Allows multiple segments per process
  - Easy to allow sharing of code
  - Do not need to load entire process in memory
- Disadvantages:
  - Extra translation overhead:
    - Memory & speed
  - An entire segment needs to reside contiguously in memory!
    ⇒ Fragmentation

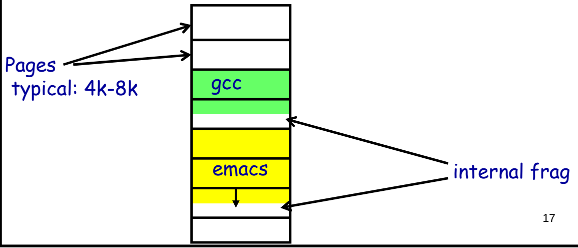
Fragmentation

- “The inability to use free memory”
- External Fragmentation:
  - Variable sized pieces ⇒ many small holes over time
- Internal Fragmentation:
  - Fixed sized pieces ⇒ internal waste if entire piece is not used



Paging

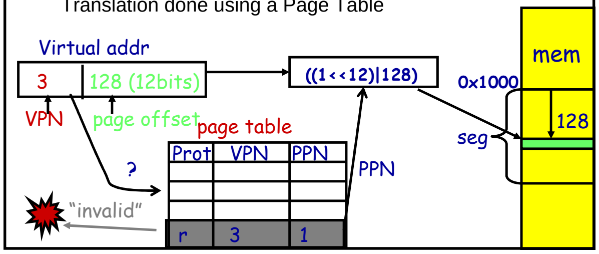
- Divide memory into fixed size pieces
  - Called “frames” or “pages”
- Pros: easy, no external fragmentation

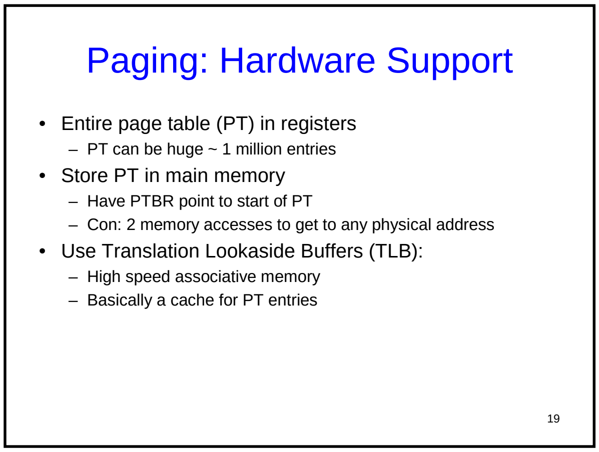


Mapping Pages

- If 2^m virtual address space, 2^n page size
  ⇒(m - n) bits to denote page number, n for offset within page

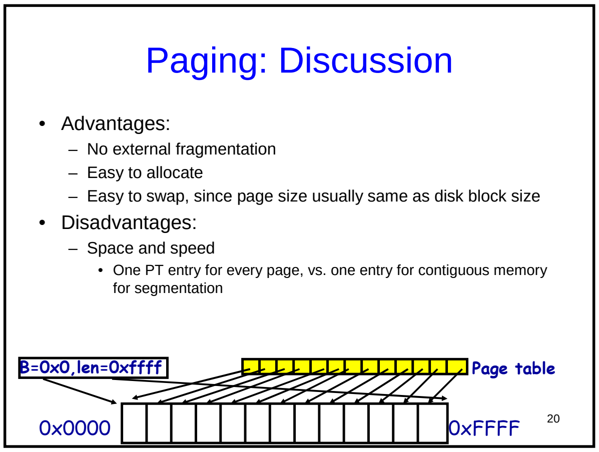
Translation done using a Page Table





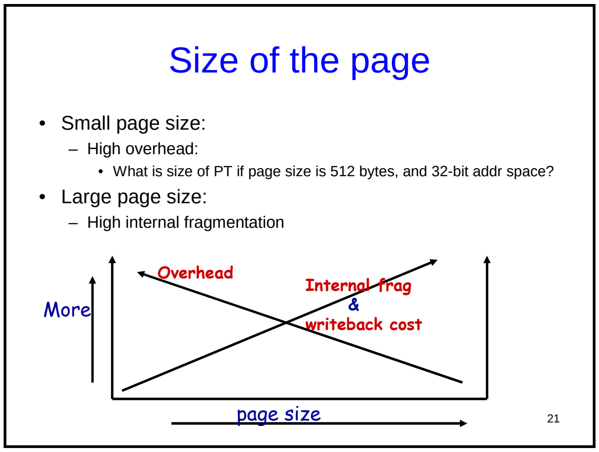
Paging: Hardware Support

- Entire page table (PT) in registers
  - PT can be huge ~ 1 million entries
- Store PT in main memory
  - Have PTBR point to start of PT
  - Con: 2 memory accesses to get to any physical address
- Use Translation Lookaside Buffers (TLB):
  - High speed associative memory
  - Basically a cache for PT entries



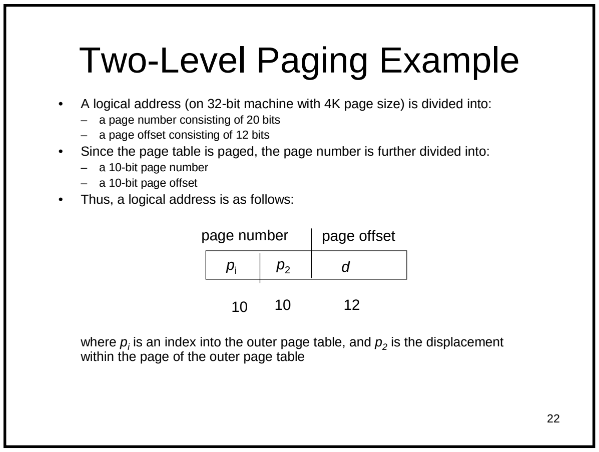
Paging: Discussion

- Advantages:
  - No external fragmentation
  - Easy to allocate
  - Easy to swap, since page size usually same as disk block size
- Disadvantages:
  - Space and speed
    - One PT entry for every page, vs. one entry for contiguous memory for segmentation



Size of the page

- Small page size:
  - High overhead:
    - What is size of PT if page size is 512 bytes, and 32-bit addr space?
- Large page size:
  - High internal fragmentation

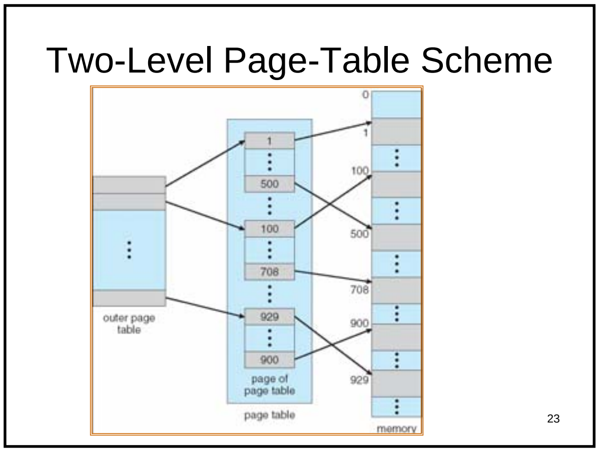


Two-Level Paging Example

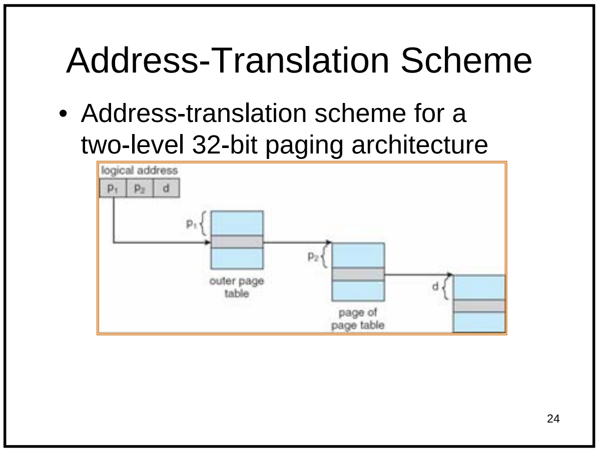
- A logical address (on 32-bit machine with 4K page size) is divided into:
  - a page number consisting of 20 bits
  - a page offset consisting of 12 bits
- Since the page table is paged, the page number is further divided into:
  - a 10-bit page number
  - a 10-bit page offset
- Thus, a logical address is as follows:

page number		page offset
p_i	p_2	d
10	10	12

where p_i is an index into the outer page table, and p_2 is the displacement within the page of the outer page table



Two-Level Page-Table Scheme



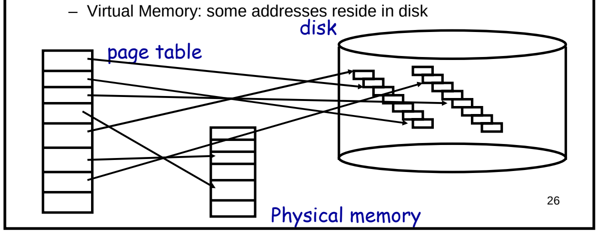
Address-Translation Scheme

- Address-translation scheme for a two-level 32-bit paging architecture

Virtual Memory

What is virtual memory?

- Each process has illusion of large address space
  - 2^{32} for 32-bit addressing
- However, physical memory is much smaller
- How do we give this illusion to multiple processes?
  - Virtual Memory: some addresses reside in disk

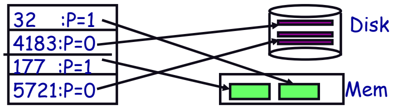


Virtual Memory

- Load entire process in memory (swapping), run it, exit
  - Is slow (for big processes)
  - Wasteful (might not require everything)
- Solutions: partial residency
  - Paging: only bring in pages, not all pages of process
  - Demand paging: bring only pages that are required
- Where to fetch page from?
  - Have a contiguous space in disk: swap file (pagefile.sys)

How does VM work?

- Modify Page Tables with another bit (“is present”)
  - If page in memory, *is_present = 1*, else *is_present = 0*
  - If page is in memory, translation works as before
  - If page is not in memory, translation causes a **page fault**



Page Faults

- On a page fault:
  - OS finds a free frame, or evicts one from memory (which one?)
    - Want knowledge of the future?
  - Issues disk request to fetch data for page (what to fetch?)
    - Just the requested page, or more?
  - Block current process, context switch to new process (how?)
    - Process might be executing an instruction
  - When disk completes, set present bit to 1, and current process in ready queue

Resuming after a page fault

- Should be able to restart the instruction
- For RISC processors this is simple:
  - Instructions are idempotent until references are done
- More complicated for CISC:
  - E.g. move 256 bytes from one location to another
  - Possible Solutions:
    - Ensure pages are in memory before the instruction executes

When to fetch?

- Just before the page is used!
  - Need to know the future
- Demand paging:
  - Fetch a page when it faults
- Prepaging:
  - Get the page on fault + some of its neighbors, or
  - Get all pages in use last time process was swapped

What to replace?

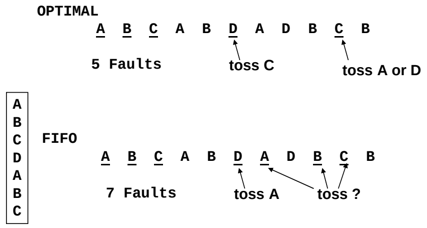
- Page Replacement
  - When process has used up all frames it is allowed to use
  - OS must select a page to eject from memory to allow new page
  - The page to eject is selected using the Page Replacement Algo
- Goal: Select page that minimizes future page faults

Page Replacement Algorithms

- Random: Pick any page to eject at random
  - Used mainly for comparison
- FIFO: The page brought in earliest is evicted
  - Ignores usage
  - Suffers from “Belady’s Anomaly”
    - Fault rate could increase on increasing number of pages
    - E.g. 0 1 2 3 0 1 4 0 1 2 3 4 with frame sizes 3 and 4
- OPT: Belady’s algorithm
  - Select page not used for longest time
- LRU: Evict page that hasn’t been used the longest
  - Past could be a good predictor of the future

Example: FIFO, OPT

Reference stream is A B C A B D A D B C

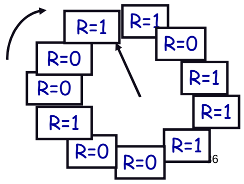


Implementing Perfect LRU

- On reference: Time stamp each page
- On eviction: Scan for oldest frame
- Problems:
  - Large page lists
  - Timestamps are costly
- Approximate LRU
  - LRU is already an approximation!

LRU: Clock Algorithm

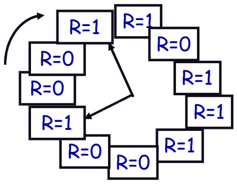
- Each page has a reference bit
  - Set on use, reset periodically by the OS
- Algorithm:
  - FIFO + reference bit (keep pages in circular list)
    - Scan: if ref bit is 1, set to 0, and proceed. If ref bit is 0, stop and evict.
- Problem:
  - Low accuracy for large memory



LRU with large memory

- Solution: Add another hand
  - Leading edge clears ref bits
  - Trailing edge evicts pages with ref bit 0

- What if angle small?
- What if angle big?



Clock Algorithm: Discussion

- Sensitive to sweeping interval
  - Fast: lose usage information
  - Slow: all pages look used
- Clock: add reference bits
  - Could use (ref bit, modified bit) as ordered pair
  - Might have to scan all pages
- LFU: Remove page with lowest count
  - No track of when the page was referenced
  - Use multiple bits. Shift right by 1 at regular intervals.
- MFU: remove the most frequently used page
- LFU and MFU do not approximate OPT well

Page Buffering

- Cute simple trick: (XP, 2K, Mach, VMS)
  - Keep a list of free pages
  - Track which page the free page corresponds to
  - Periodically write modified pages, and reset modified bit

Allocating Pages to Processes

- Global replacement
  - Single memory pool for entire system
  - On page fault, evict oldest page in the system
  - Problem: protection
- Local (per-process) replacement
  - Have a separate pool of pages for each process
  - Page fault in one process can only replace pages from its own process
  - Problem: might have idle resources