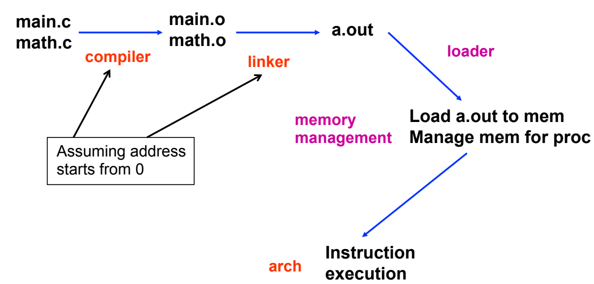
Demand Paging

ECE595
Oct 18

Y. Charlie Hu

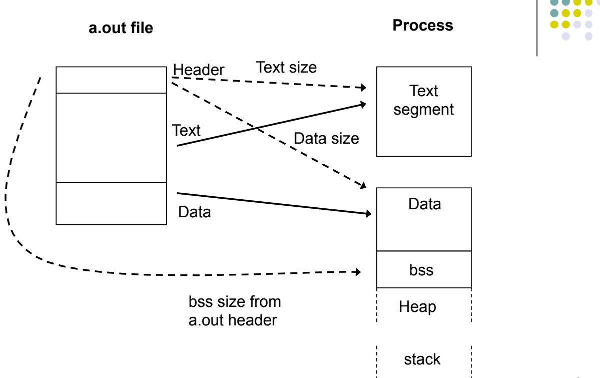
1

[lec13] The big picture – connecting the dots



2

[lec13] Loading

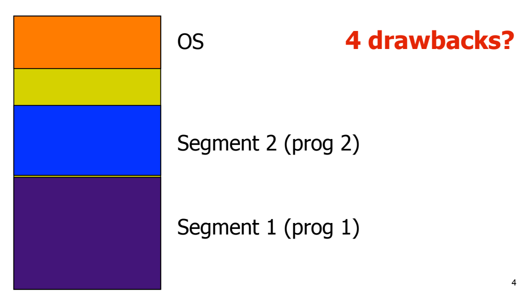


bss = block started by symbols (un-initialized data segment)

3

Review: sharing main memory

- Simple multiprogramming



4

[lec14] Sharing main memory

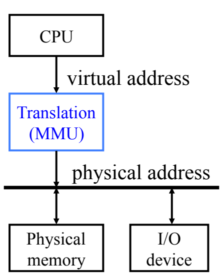
- Simple multiprogramming – 4 drawbacks
  - Lack of protection
  - Cannot relocate dynamically
  - Single segment per process
  - Entire address space needs to fit in mem

Dynamic Memory Relocation

[lec15] 3. Dynamic memory relocation

- Instead of changing the address of a program before it's loaded, change the address dynamically *during every reference*
  - Under dynamic relocation, each program-generated address (called a *logical address* or *virtual address*) is translated in hardware to a *physical* or *real address* at runtime

[lec15] Translation overview



- Actual translation is in hardware (MMU)
- Controlled in software
- CPU view
  - what program sees, virtual memory
- Memory view
  - physical memory

What is the essence of going through MMU?

Sharing main memory

- Simple multiprogramming – 4 drawbacks
  - Lack of protection
  - Cannot relocate dynamically
    → Base-and-bound
  - Single segment per process
    → Paging, segmentation, etc.
  - Entire address space needs to fit in mem

Dynamic Memory Relocation

Sharing main memory

- Simple multiprogramming – 4 drawbacks
  - Lack of protection
  - Cannot relocate dynamically
    → dynamic memory relocation: base& bound
  - Single segment per process
    → dynamic memory relocation: segmentation, paging
  - Entire address space needs to fit in mem
    - More need for swapping
    - Need to swap whole, very expensive!

The last drawback

- So far we've separated the process's view of memory from the OS's view using a mapping mechanism
  - Each sees a different organization
  - Allows OS to shuffle processes around
  - Simplifies memory sharing
  - *What is the essence of the mechanism that enables this?*
- But, a user process had to be completely loaded into memory before it could run

→ Wasteful since a process only needs a small amount of its total memory at any time (*reference locality!*)

[lec1] What is an OS?

Extended (abstract) machine (answer 2)

- Much more ideal environment than the hardware
  - Ease to use
  - Fair (well-behaved)
  - Portable (back-compatible)
  - Reliable
  - Safe
- Illusion of infinite, private resources
  - Single processor → many separate processors
  - Single memory → many separate, larger memories

?

Virtual Memory

- Definition: *Virtual memory* permits a process to run with only some of its virtual address space loaded into physical memory
- Key idea: Virtual address space translated to either
  - Physical memory (small, fast) or
  - Disk (backing store), large but slow
- [Deep thinking] What made above possible?
- Objective:
  - To produce the illusion of memory as big as necessary

Virtual Memory

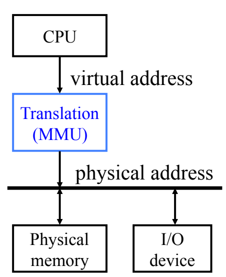
- “To produce the illusion of memory as big as necessary”
  - Without suffering a huge slowdown of execution
  - Why makes this possible?
  - *Principle of locality*
    - Knuth’s estimate of 90% of the time in 10% of the code
    - There is also significant locality in data references

Virtual Memory Implementation

- Virtual memory is typically implemented via *demand paging*
- *demand paging:* paging with swapping, e.g., physical pages are swapped in and out of memory

Demand Paging (paging with swapping)

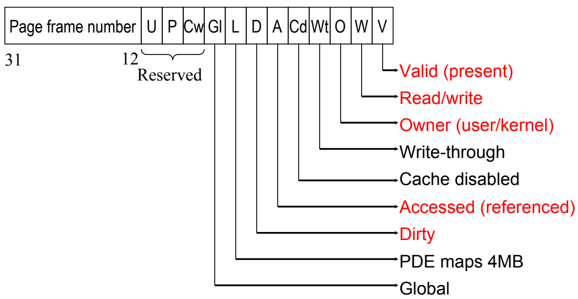
- If not all of a program is loaded when running, what happens when referencing a byte not loaded yet?
- How to detect this?
  - In software?



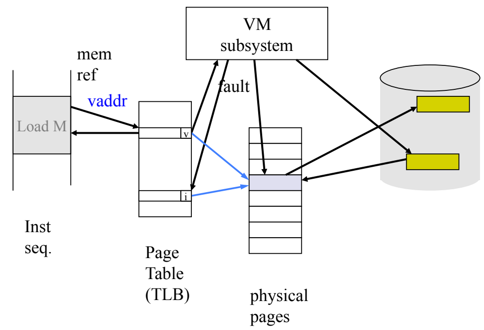
Demand Paging (paging with swapping)

- If not all of a program is loaded when running, what happens when referencing a byte not loaded yet?
- Hardware/software cooperate to make things work
  - Extend PTEs with an extra bit “present”
  - Any page not in main memory right now has the “present” bit cleared in its PTE
  - If “present” isn’t set, a reference to the page results in a trap by the paging hardware, called *page fault*
  - What needs to happen when page fault occurs?

x86 Page Table Entry



Page Fault Handling in Demand Paging



Page fault handling (cont)

- On a page fault
  - Find an unused phy. page or a used phy. page (how?)
  - If the phy. page is used
    - If it has been modified, write it to disk
    - Invalidate its current PTE and TLB entry (how?)
  - Load the new page from disk
  - Update the faulting PTE and its TLB entry
  - Restart the faulting instruction
- Supporting data structure
  - For speed: A list of unused physical pages (more later)
  - Data structure to map a phy. page to its pid and virtual address
    - Sounds faimiliar?

Page Fault Handling: Subtlety

- Page fault may have occurred in the middle of an instruction!
  - Suppose the instruction is restarted from the beginning
    - How is beginning located?
    - Side effects? MOVE (SP)+, R2
    - Require hardware support to keep track of side effects of instructions and undo them before restarting
  - You can make it easy when designing ISA
    - RISC (load-store) architecture make this relatively easy – only load/store can generate page faults

Deep thinking: VM implementation

- Virtual memory is typically implemented via *demand paging*
- It can also be implemented via *demand segmentation*
  - Double drawbacks?

Deep thinking: Virtual Address vs. Virtual Memory

- Virtual address
  - Supported with dynamic memory relocation
    - Segmentation
    - Paging
- Virtual memory
  - Dynamic memory relocation + swapping
    - Demand paging
    - Demand segmentation