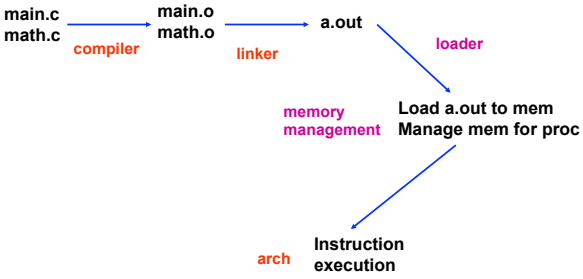
Sharing Main Memory, Segmentation

ECE595
Feb 8

Y. Charlie Hu

Connecting the dots



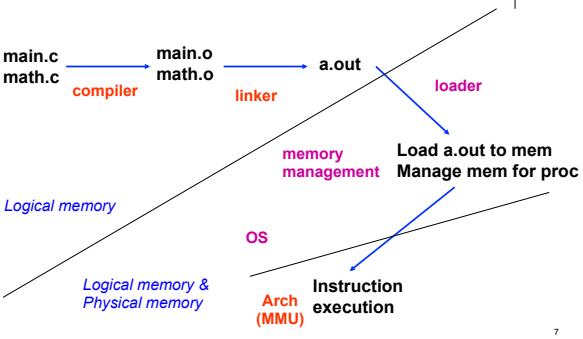
The big picture

- a.out needs address space for
  - text seg, data seg, and (hypothetical) heap, stack
- A running process needs phy. memory for
  - text seg, data seg, heap, stack
- But no way of knowing where in phy mem at
  - Programming time, compile time, linking time
- Best way out?
  - Make agreement to divide responsibility
    - Assume address starts at 0 at prog/compile/link time
    - OS needs to work hard at loading/runing time

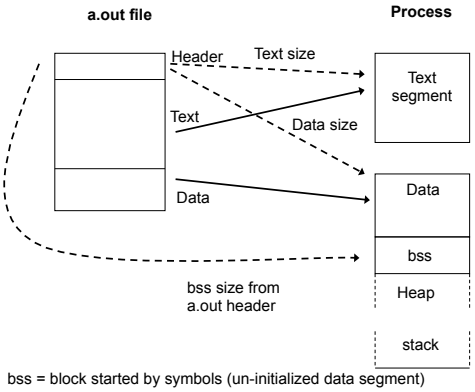
Big picture (cont)

- OS deals with physical memory
  - Loading
  - Sharing physical memory between processes
  - Dynamic memory allocation

Connecting the dots



Loading



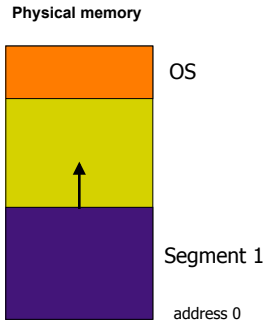
bss = block started by symbols (un-initialized data segment)

Dynamic memory allocation during program execution

- Stack: for procedure calls
- Heap: for malloc()
- Both dynamically growing/shrinking

- Assumption for now:
  - Heap and stack are fixed size
  - OS has to worry about loading 4 segments per process:
    - Text
    - Data
    - Heap
    - stack

1. Simple uniprogramming: Single segment (code, data, stack heap) per process



Simple uniprogramming:
Single segment per process

- Highest memory holds OS
- Process is allocated memory starting at 0, up to the OS area
- When loading a process, just bring it in at 0
  - virtual address == physical address!
- Examples:
  - early batch monitor which ran only one job at a time
    - if the job wrecks the OS, reboot OS
  - 1st generation PCs operated in a similar fashion
- Pros / Cons?

Multiprogramming

- Want to let several processes coexist in main memory

Issues in sharing main memory

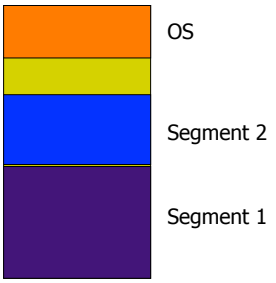
- Transparency:
  - Processes should not know memory is shared
  - Run regardless of the number/locations of processes
- Safety:
  - Processes mustn't be able to corrupt each other
- Efficiency:
  - Both CPU and memory utilization shouldn't be degraded badly by sharing

2. Simple multiprogramming

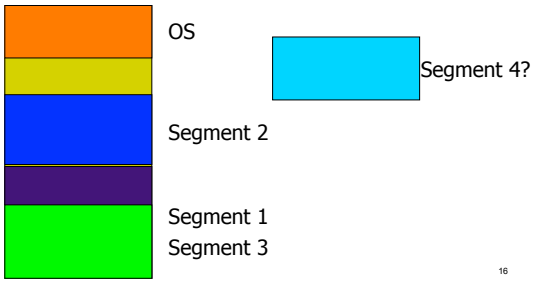
With static software memory relocation, no protection, 1 segment per process:

- Highest memory holds OS
- Processes allocated memory starting at 0, up to the OS area
- When a process is loaded, relocate it so that it can run in its allocated memory area
  - How? (use symble table and relocation info)
- Analogy to linking?

Simple multiprogramming:
Single segment per process, static relocation



Simple multiprogramming:
Single segment per process, static relocation



Simple multiprogramming:
Single segment per process, static relocation

- 4 drawbacks
  1. No protection
  2. Low utilization -- Cannot relocate *dynamically*
     - Binary is fixed (after loading)
     - Cannot do anything about holes
  3. No sharing -- Single segment per process
     - Cannot share part of process address space (e.g. text)
  4. Entire address space needs to fit in mem
     - Need to swap whole, very expensive!

What else can we do?

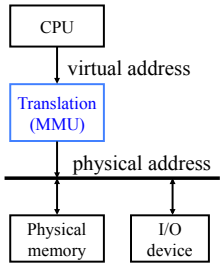
- Already tried
  - Compile time / linking time
  - Loading time
- Let us try execution time!

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*

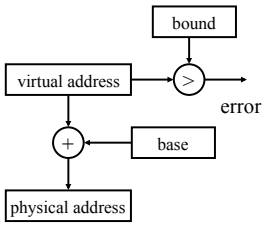
Can this be done in software?

Translation overview



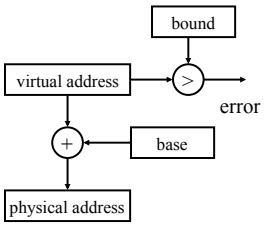
- Actual translation is in hardware (MMU)
- Controlled in software
- CPU view
  - what program sees, virtual addresses
- Memory view
  - physical memory addresses

3.1 Base and bound



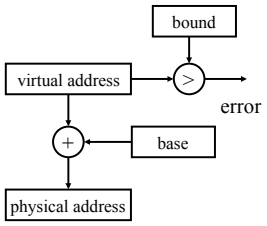
- Built in Cray-1 (1976)
- A program can only access physical memory in [base, base+bound]
- On a context switch: save/restore base, bound registers
- Pros:
  - simple, fast, cheap
  - Can relocate segment

3.1 Base and bound



- The essence:
  - A level of indirection
  - Phy. Addr = Vir. Addr + base
- How to relocate segment in physical memory?
  - From Base 1 to Base 2?

Base and bound



- Cons:
  - Only one segment
  - How can two processes share code while keeping private data areas (shared editors)?

What have we achieved?

- 4 drawbacks

1. No protection
2. Low utilization -- Cannot relocate dynamically
   - Cannot do anything about holes
3. No sharing -- Single segment per process
   - Cannot share part of process address space (e.g. text)
4. Entire address space needs to fit in mem
   - Need to swap whole, very expensive!

Readings

- Chapter 8