Sharing Main Memory, Segmentation

ECE469, Feb 21

Yiying Zhang

Reading assignment

- Dinosaur Chapter 8
- Comet Chapter 13, 15, 16

Connecting the dots

main.c
math.c

compiler

main.o
math.o

linker

a.out

loader

Load a.out to mem
Manage mem for proc

memory
management

Instruction
execution

arch

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

main.c
math.c

compiler

main.o
math.o

linker

a.out

loader

memory
management

Load a.out to mem
Manage mem for proc

Logical memory

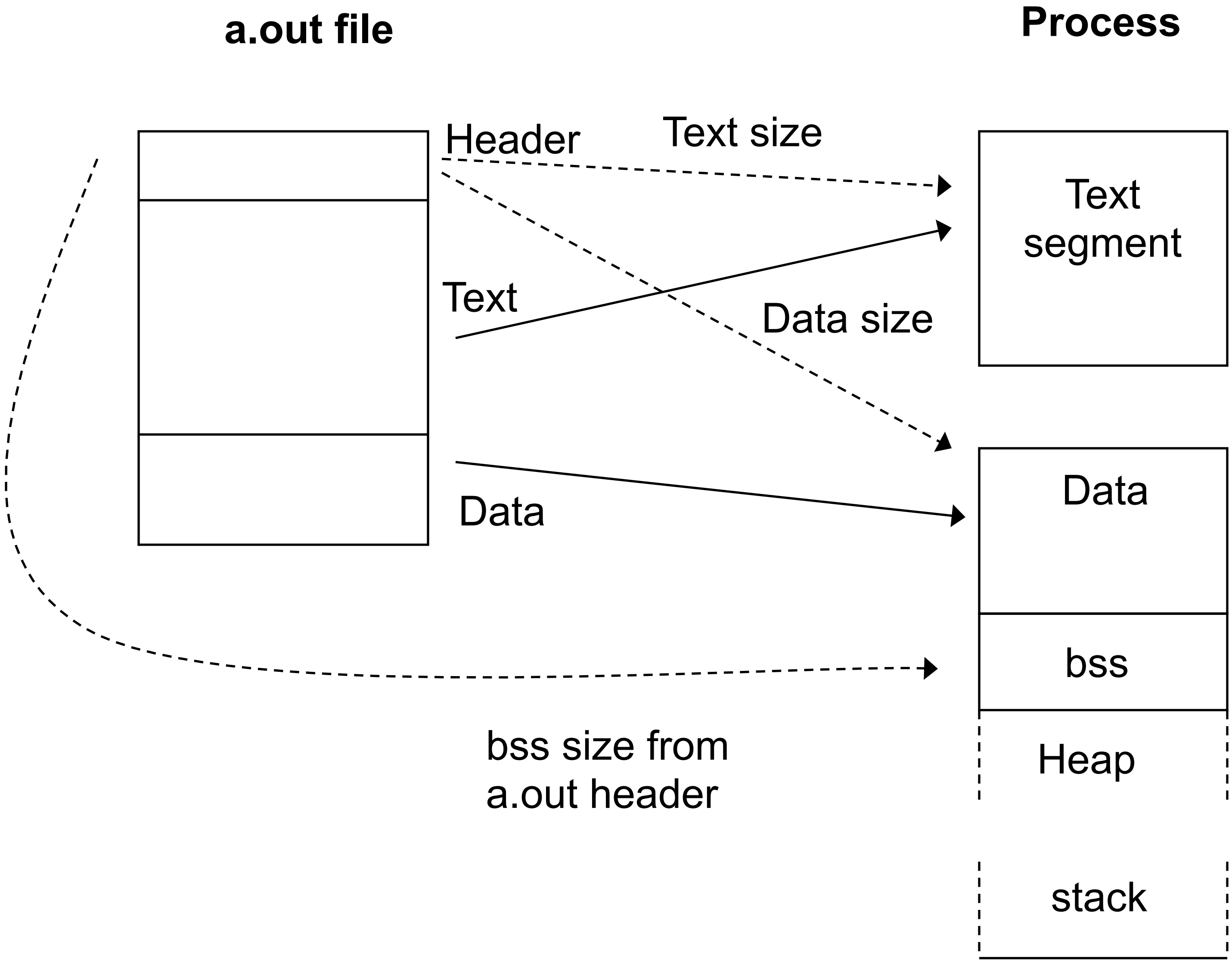
OS

Logical memory &
Physical memory

arch

Instruction
execution

Loading



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

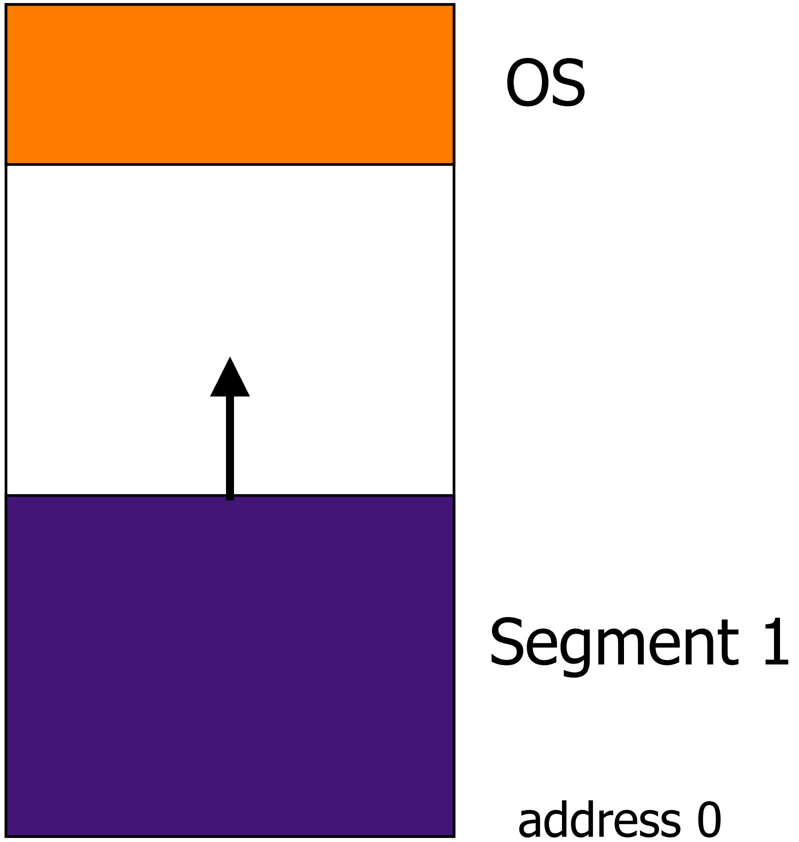
Dynamic memory allocation during program execution

- 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

Physical memory



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

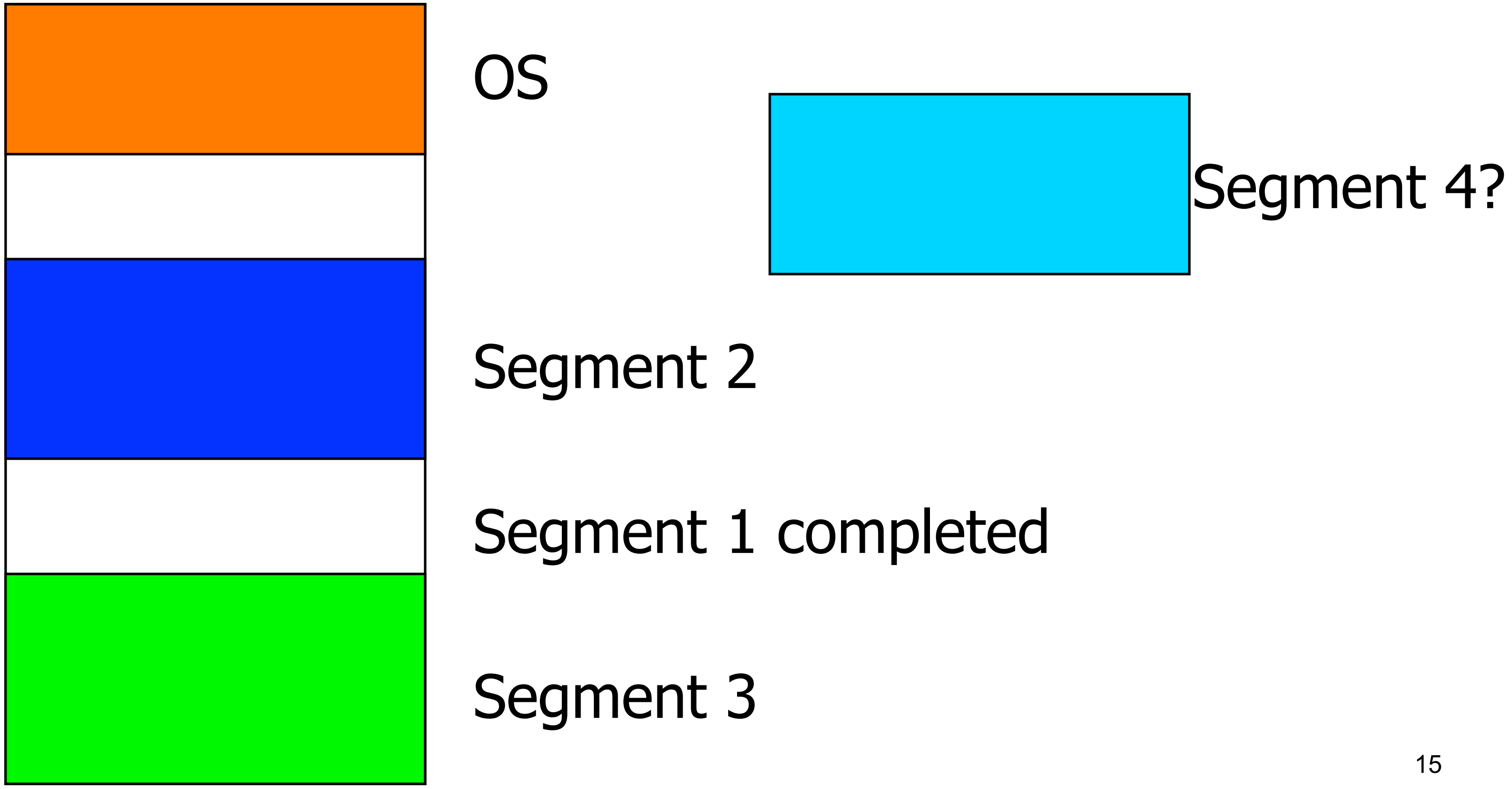
2. Simple multiprogramming

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

- four 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

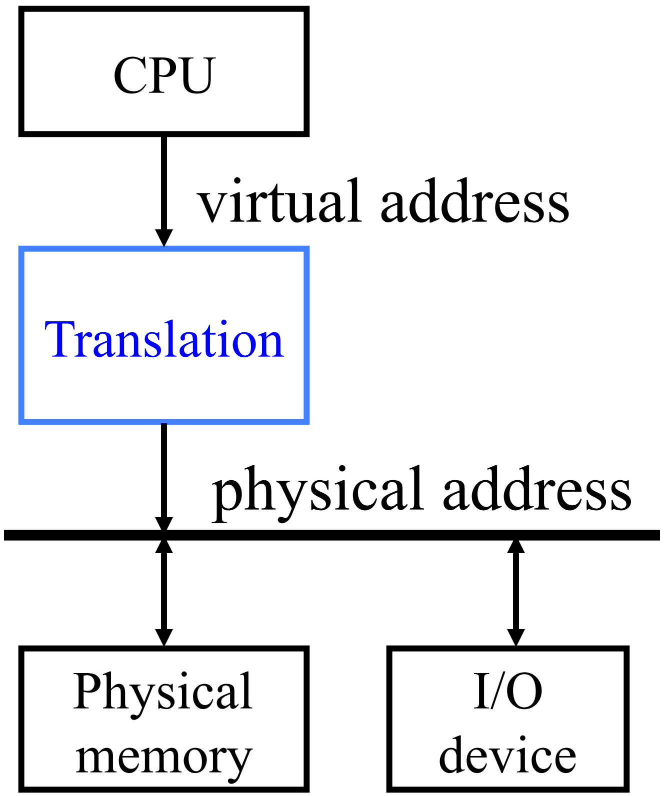
- Let us try execution time!

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*

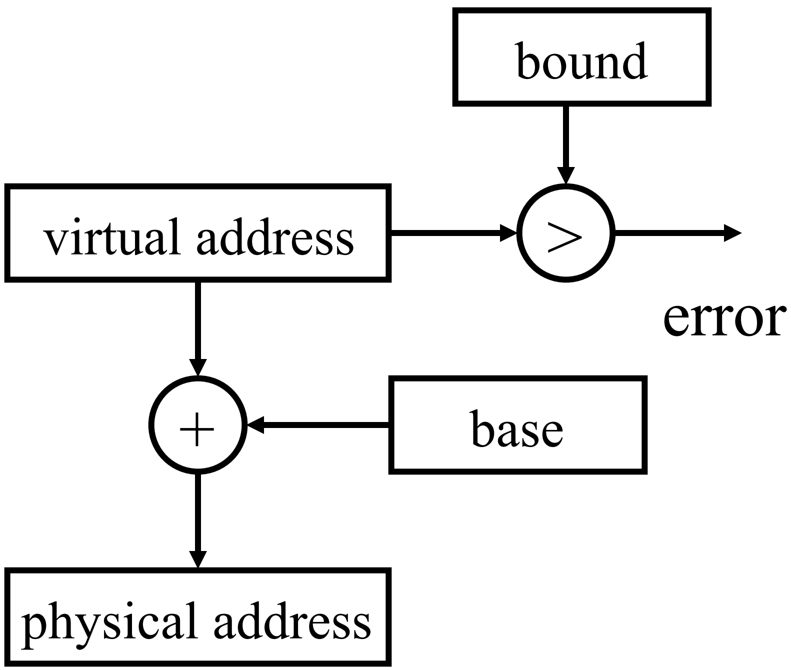
Can this be done in software?

Translation overview



- Actual translation process is usually performed by hardware
- Translation table is set up by software
- CPU view
  - what program sees, virtual addresses
- Memory view
  - physical memory addresses

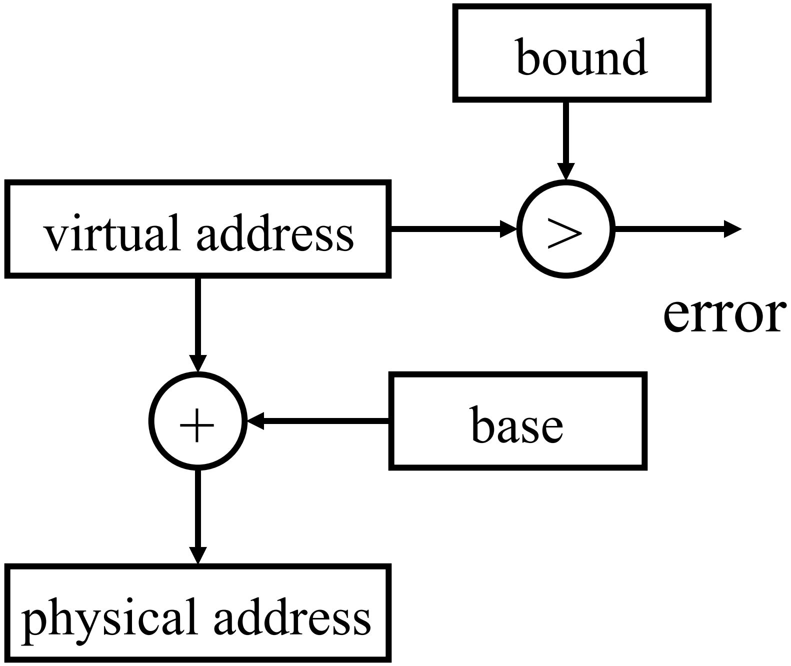
3.1 Base and bound



- 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

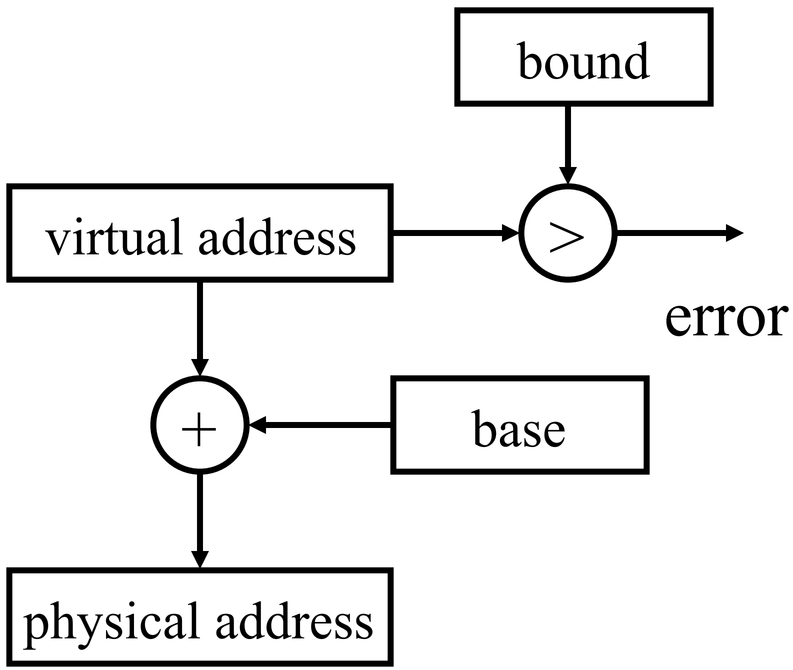
- Pros:
  - simple, fast translation, cheap
  - Can relocate segment

3.1 Base and bound



- The essence:
  - A level of (static) indirection
  - Phy. Addr = Vir. Addr + base

3.1 Base and bound



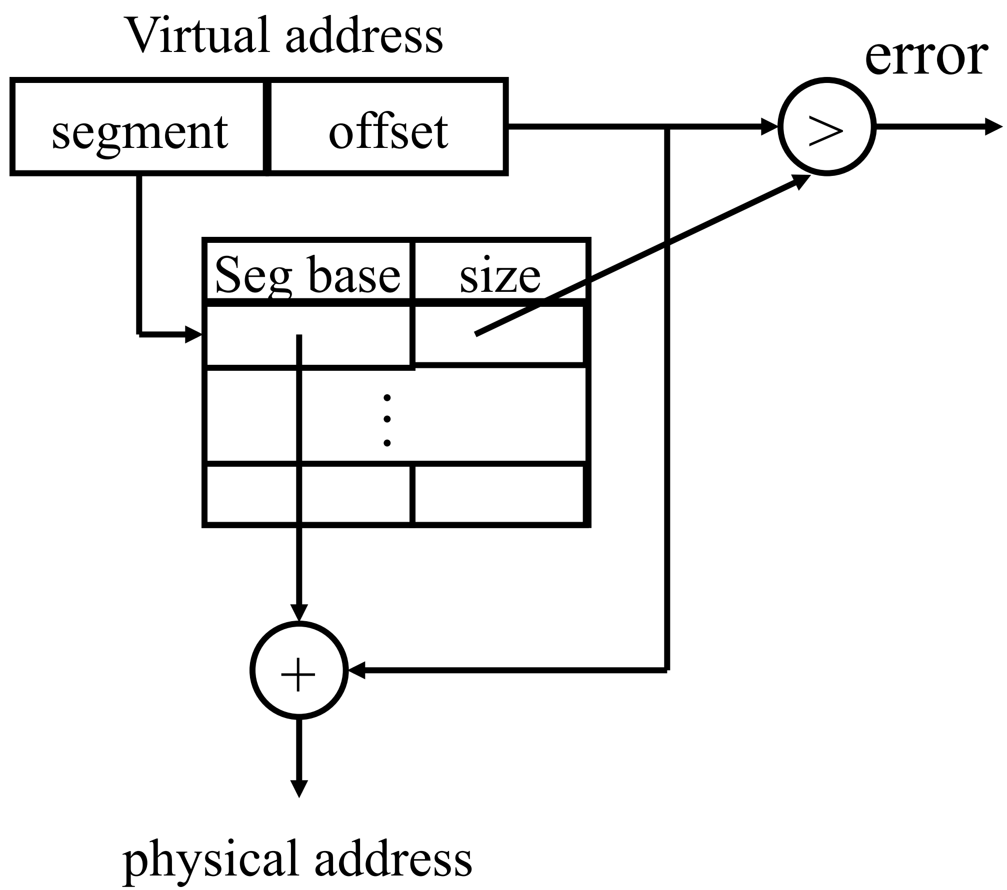
- Cons:
  - Only one segment
  - How can two processes share code while keeping private data areas (shared editors)?
    - Can it be done safely with a single-segment scheme?

What have we solved?

- four 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!

3.2 Multiple Segments



- Have a table of (seg, size)
- Further protection: each entry has
  (nil, read, write, exec)
- On a context switch: save/restore the table (or a pointer to the table) in kernel memory

How does this allow two processes to share code segment?

Segmentation example

text segment [0x0000, 0x04B0]

foo:

019A: LD R1, 15DC

01C2: jmp 01F4

01E0: call 0320

01F4: X:

bar procedure

0320: bar:

Data segment [0x1000, 0x16A0]

15DC: _Y:

2-bit segment number, 12-bit offset

Segment	Base	Bounds	RW
0	4000	4B0	10
1	0	6A0	11
2	3000	FFF	11
3	--	--	00

Segmentation example

text segment [0x0000, 0x04B0]

foo:

019A: LD R1, 15DC

01C2: jmp 01F4

01E0: call 0320

01F4: X:

bar procedure

0320: bar:

Data segment [0x1000, 0x16A0]

15DC: _Y:

2-bit segment number, 12-bit offset

Segment	Base	Bounds	RW
0	4000	4B0	10
1	0	6A0	11
2	3000	FFF	11
3	--	--	00

➔ Where is 01F4 in physical memory?

Segmentation example

text segment [0x0000, 0x04B0]

foo:

019A: LD R1, 15DC

01C2: jmp 01F4

01E0: call 0320

01F4: X:

bar procedure

0320: bar:

Data segment [0x1000, 0x16A0]

15DC: _Y:

2-bit segment number, 12-bit offset

Segment	Base	Bounds	RW
0	4000	4B0	10
1	0	6A0	11
2	3000	FFF	11
3	--	--	00

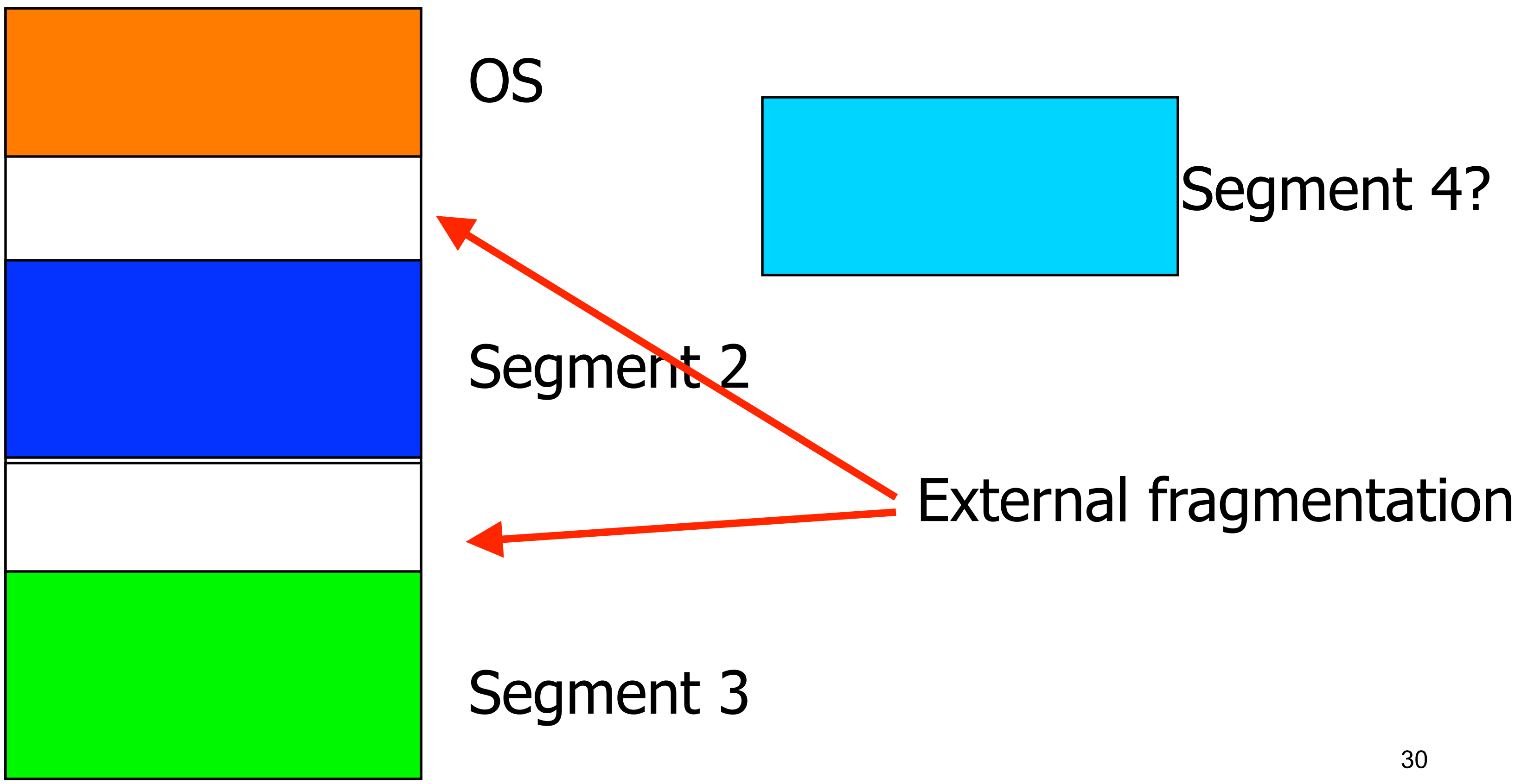
➔ Where is 15DC in physical memory?

Pros/cons of segmentation

- Pros:
  - Process can be split among several segments
    - Allows sharing
  - Segments can be assigned, moved, or swapped independently

- Cons:
  - External fragmentation: many holes in physical memory
    - Also happens in base and bound scheme

Simple multiprogramming:
Single segment per process, static relocation



What fundamentally causes external fragmentation?

1. Segments of many different sizes

2. Each has to be allocated contiguously

Dynamic memory allocation problem

- Problem: External fragmentation caused by holes too small

- How much can a smart allocator help?
  - The allocator maintains a free list of holes
  - Allocation algorithms differ in how to allocate from the free list

Dynamic allocation algorithms

- Best fit: allocate the smallest chunk big enough
- First fit: allocate the first chunk big enough
  - Rotating first fit

- Is best fit necessarily better than first fit?
  - Example: 2 free blocks of size 20 and 15
  - If allocation ops are 10 then 20, which one wins?
  - If ops are 8, 12, then 12, which one wins?

Dynamic allocation algorithms

- Analysis shows
  - First fit tends to leave average-size holes
  - Best fit tends to leave some very large holes, very small holes

- Knuth claims that if storage is close to running out, it will run out regardless of which scheme is used

  → Pick the easiest or most efficient (e.g. first fit)

Segmentation: OS implementation

- Keep segment table in PCB
- When creating process, allocate space for segments, fill in PCB bases/bounds
- When process dies, return physical space used by segments to free pool
- Context switch?
  - Saves old segment table / Loads new segment table
  - What about context switch of threads?
  - True-or-false: CS between threads of same process cheaper than CS between processes

[lec2] Kernel data structure: Process Control Block (Process Table)

- Process management info
  - State (ready, running, blocked)
  - PC & Registers, parents, etc
  - CPU scheduling info (priorities, etc.)
- Memory management info
  - Segments, page table, stats, etc
- I/O and file management
  - Communication ports, directories, file descriptors, etc.

Managing segments (cont)

To enlarge a segment:

- See if space above segment is free. If so, just update the bound and use that space
- Or, move this segment to disk and bring it back into a larger hole (or maybe just copy it to a large hole)

Managing segments (cont)

- When there is no space to allocate a new segment:
  - Compact memory – how?

Summary: Evolution of Memory Management (so far)

Scheme	How	Pros	Cons
Simple uniprogramming	1 segment loaded to starting address 0	Simple	1 process 1 segment No protection
Simple multiprogramming	1 segment relocated at loading time	Simple, Multiple processes	1 segment/process No protection External frag.
Base & Bound	Dynamic mem relocation at runtime	Simple hardware, Multiple processes Protection	1 segment/process, External frag.
Multiple segments	Dynamic mem relocation at runtime	Sharing, Protection, multi segs/process	More hardware, External frag.

Dynamic storage allocation

- Why isn’t static allocation sufficient?

- Need dynamic memory allocation for
  - both main memory
  - file space on disk

Dynamic storage allocation: two basic operations

- Allocate

- Free

Dynamic storage allocation: two general ways

- Stack
  - Restricted
  - Simple and efficient

- Heap
  - More general
  - Less efficient
  - More difficult to implement

Stack organization

- Memory allocation & freeing are predictable
  - *“We do better when we can predict future”*
  - Example:
    - Procedure call
- Allocation is LIFO
- Stack keeps all the free space together

Heap organization

- Allocation & freeing are unpredictable
- For arbitrary, complex data structures
  - Example: payroll system
    - Don't know when employee will join or leave the company
    - Must keep track of all of them
- Memory consists of allocated areas and free areas (holes) → lots of holes inevitable
- Goal: keep # of holes small, size large

Heap organization

- *Fragmentation*: inefficient use of memory due to holes too small
  - What happens in stack? Do we have fragmentation there?
- Typically, heap allocation uses a *free list* of holes
- Allocation algorithms differ in how to manage the free list

Heap allocation

- Best fit
- First fit
  - Rotating first fit

- Is best fit necessarily better than first fit?
  - Example: 2 free blocks of size 20 and 15
  - If allocation ops are 10 then 20, which one wins?
  - If ops are 8, 12, then 12, which one wins?

Heap allocation

- First fit tends to leave average-size holes
- Best fit tends to leave some very large holes, very small holes
- Knuth claims that if storage is close to running out, it will run out regardless of which scheme is used

  → Pick the easiest or most efficient (first fit)

Implementation

- Bit map
  - For fixed-size chunks (e.g., disk blocks)

- Pools
  - A separate allocation pool for each popular size
  - Fast, no fragmentation
  - But some pools may run out faster than others

Reclamation

- When can dynamically-allocated memory be freed?
  - Easy if a chunk is used in one place
  - Hard when a chunk is shared
  - Sharing is indicated by presence of pointers to the data

Reclamation techniques

- Reference counts:
  - Keep track of the number of outstanding pointers to each chunk of memory
  - When this goes to 0, free the memory
  - Example:
    - File descriptors in UNIX
  - Works fine with hierarchical structures
  - What about circular structures?

Reclamation techniques

- Garbage collection
  - Storage isn't freed explicitly (using free), rather implicitly, i.e., by deleting pointers
  - When the system needs storage, it scans through all pointers (all of them!!!) and collects things not used
  - For circular structures, this is the only way
  - Makes life easier on programmers, but GCs are hard to implement

Reclamation techniques

- Garbage collection – implementation
  - Must be able to find all objects
  - Must be able to find all pointers to objects
  - Pass1: mark
    - Go through all statically-allocated and procedure local variables, looking for pointers
    - Mark each obj pointed to, and recurs
    - Compiler has to help by saving info about pointers with structures
  - Pass 2: sweep
    - Go through all objs, free up those that aren’t marked