Comp 204: Computer Systems and Their Implementation

Lecture 14: Segmentation

Today

- Dynamic Loading & Linking
  - Shared Libraries
- Memory organisation models
  - Segmentation
    - Address structure
    - Memory referencing

Dynamic Loading

- Not always necessary to load the entire image
  - Image can consist of:
    - Main program
    - Different routines, classes, etc
    - Error routines
  - Dynamic Loading allows only parts of the image to be loaded, as necessary
    - When a routine calls another routine, it checks to see if it has been loaded…
      - … if not, the relocatable linking loader is called
    - Advantage – unused routines are never loaded, thus the image is kept smaller

Linking

- Linking is the combination of user code with system or 3rd party libraries
  - Typically done as part of, or after the compilation process
- Static Linking
  - Copies of the libraries are included in the final binary program image
    - Can result in large images due to inclusion of many libraries (which in turn might link to other libraries…)
    - Wasteful both in terms of disc storage and main memory
    - Can be managed by dynamic loading, but shared libraries are still repeated in memory multiple times.

Linking (cont)

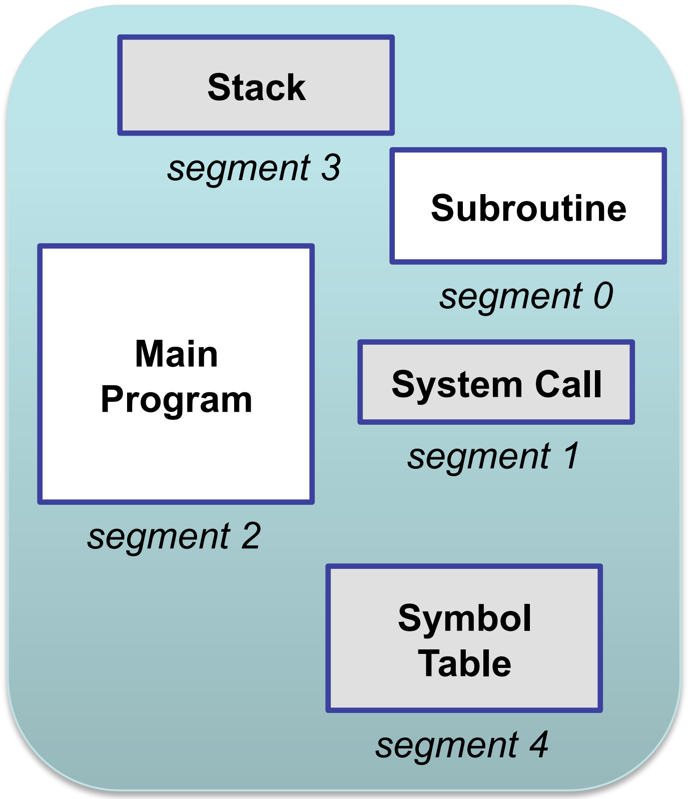
- Dynamic Linking
  - A stub is included in the image for each library routine
    - Indicates how to:
      - Locate memory resident library routine (if already loaded),
      - Load the library (if not loaded)
    - Allows re-entrant code to be shared between processes
      - Supports Library Updates (including versioning)
      - Keeps disc image small
  - Requires some assistance from the OS
    - Lower level memory organisation necessary…

Memory Organisation

To ameliorate some of the software problems arising from the *linear store,* more complex memory models are used which organise the store hierarchically:

- Segmentation
  - subdivision of the address space into logically separate regions
- Paging
  - physical subdivision of the address space for memory management

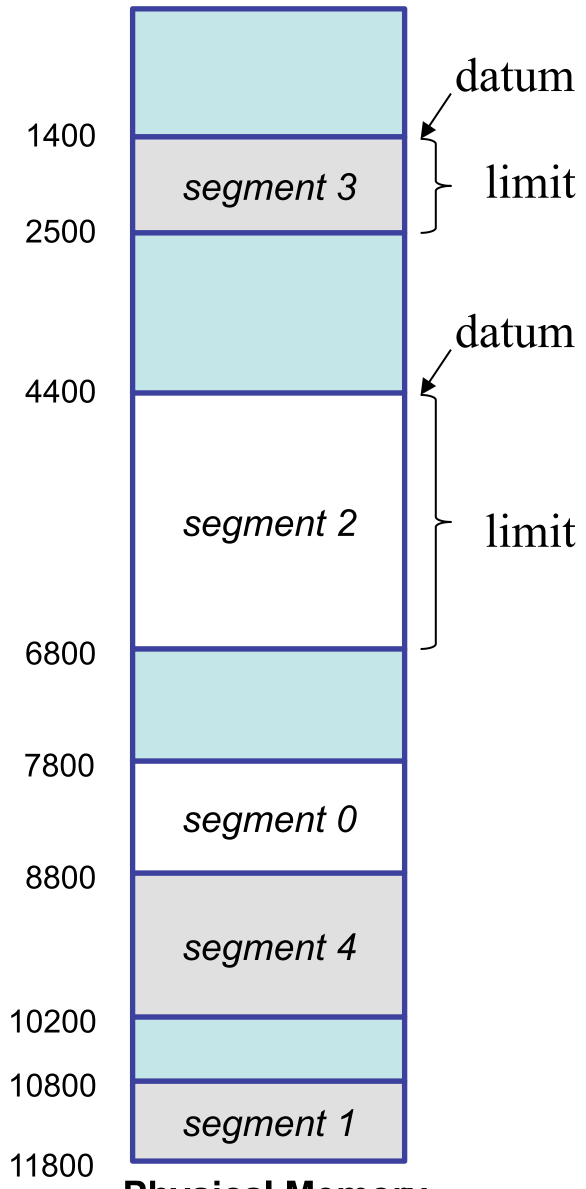
Segmentation



Logical Address Space

	Limit	Base
0	1000	7800
1	1000	10800
2	2400	4400
3	1100	1400
4	1400	8800

Segment Table

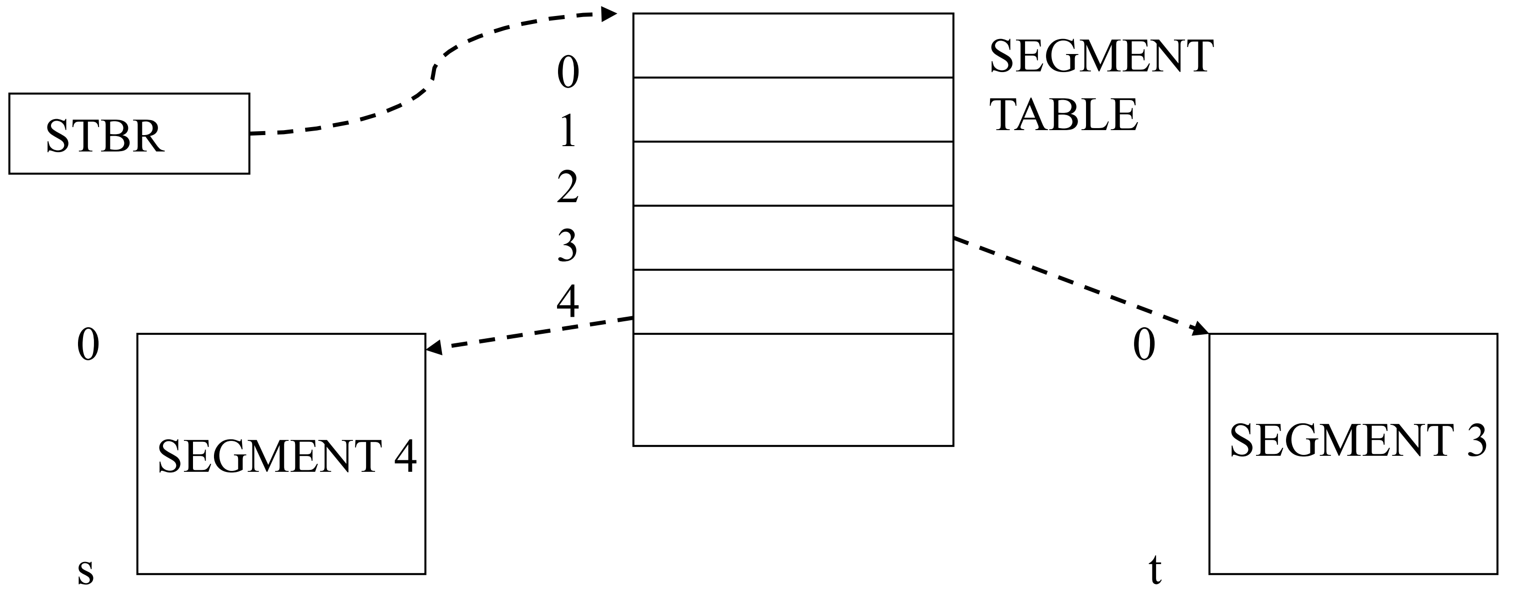


Physical Memory

NOTES: *Each segment has its own partition*
Segments need not be contiguous

Segmented Address Structure

- Each process has a segment table
  - contains datum and limit values of segments
- Address of table held in segment table base register (STBR) (saved during context switch)



Memory Referencing

- For linear store, machine code is LOAD ADDR
- For segmented store, machine code is LOAD SEG, ADDR
  - Hardware looks up segment base address in table, then adds in-segment address to produce absolute address

Question

- A program is split into 3 segments. The segment table contains the following information:

segment	datum	limit
0	1700	5500
1	5600	8100
2	8300	9985

- where 'limit' is the physical address following the end of the segment, and instructions take the form opcode segment, offset
- If the program executes
  - **LOAD 1, 135**
- what physical address is accessed?

a) 1835
b) 5735
c) 8435
d) 8235
e) 5635

Answer: b
5600 (from segment 1) + 135 (offset)

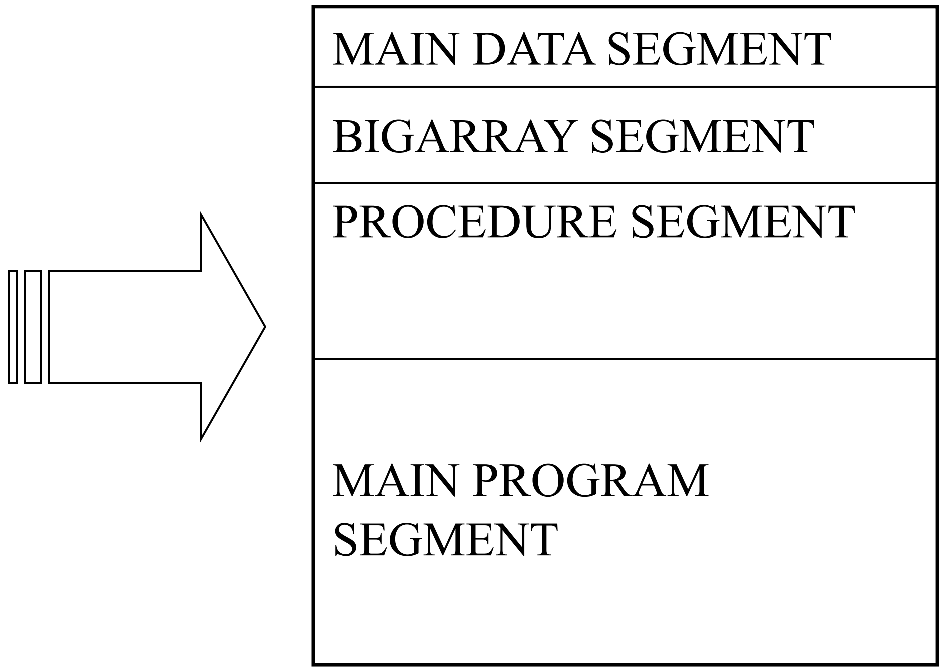
Advantages of Segmentation

- Memory allocation reflects program structure

```
begin
   int a, b, c; float …; etc.
   [1:n] float bigarray;
   procedure something …
   begin
       …
   end

   # MAIN PROGRAM #

end
```

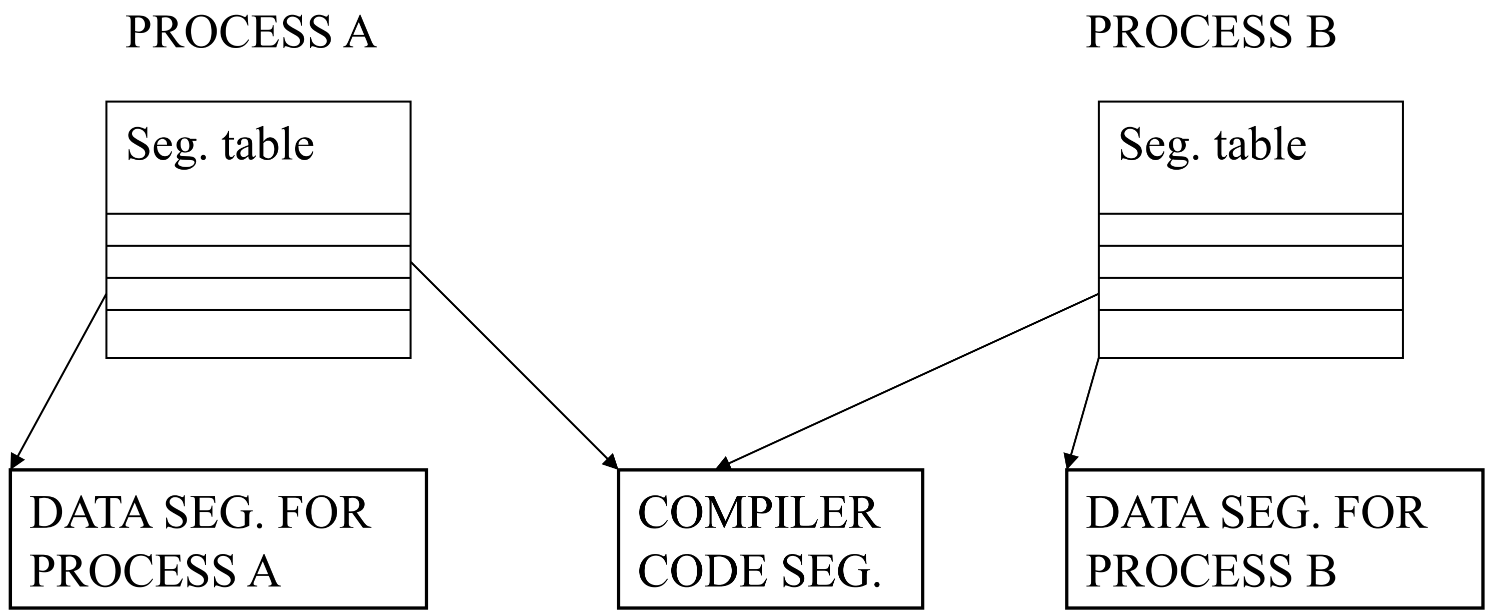


This means...

- It is easier to organise separate compilation of procedures
- Protection is facilitated
  - array bound checking can be done in hardware
  - code segments can be protected from being overwritten
  - data segments can be protected from execution
- Segments can be shared between processes

Segment Sharing

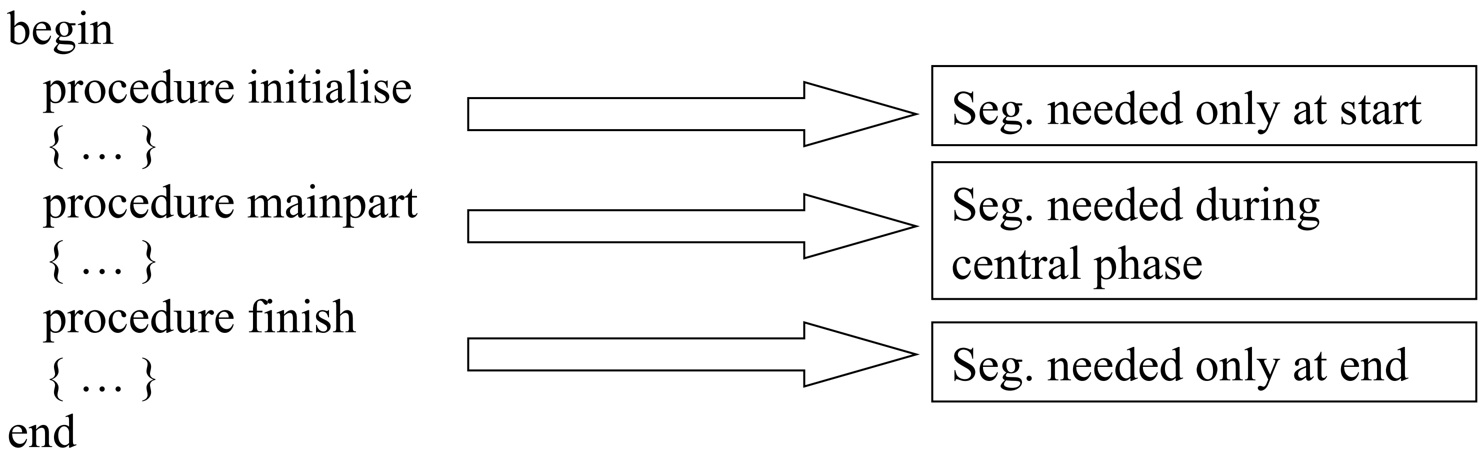
- Suppose 2 users compiling different programs



Processes have own data segments, but can share compiler code segment if it is pure (re-entrant), i.e. the code never modifies itself during execution.
-- Economises on memory, and more efficient.

Other Advantages

- Large programs broken into manageable units
  - Store allocation more flexible
  - Not all of a program need be in store at same time



```
begin
  procedure initialise
  { … }
  procedure mainpart
  { … }
  procedure finish
  { … }
end
```

Segments can be kept on backing store until needed, then swapped in
-- Allows more programs to be kept in memory
-- Swapping more efficient

Question

- Process A and process B both share the same code segment S. Which of the following statements is (are) true?

I. An entry for S appears in both segment tables
II. The segment code must be re-entrant
III. The segment code must be recursive

a) I only
b) II only
c) I and II
d) I and III
e) I, II and III

Answer: c
I and II – as the segment is shared, both processes need to index it (i.e. include an entry in their respective segment tables, and, the code must not be changed by its use (i.e. it should be re-entrant). Recursion is irrelevant to this issue.

But...

- Swapping can only work effectively if programs sensibly segmented

- Still have space allocation problems for awkward-sized segments
  - made worse by frequent need to find space whenever swapping-in occurs
  - fragmentation and blocking remain problems