CS5460/6460: Operating Systems

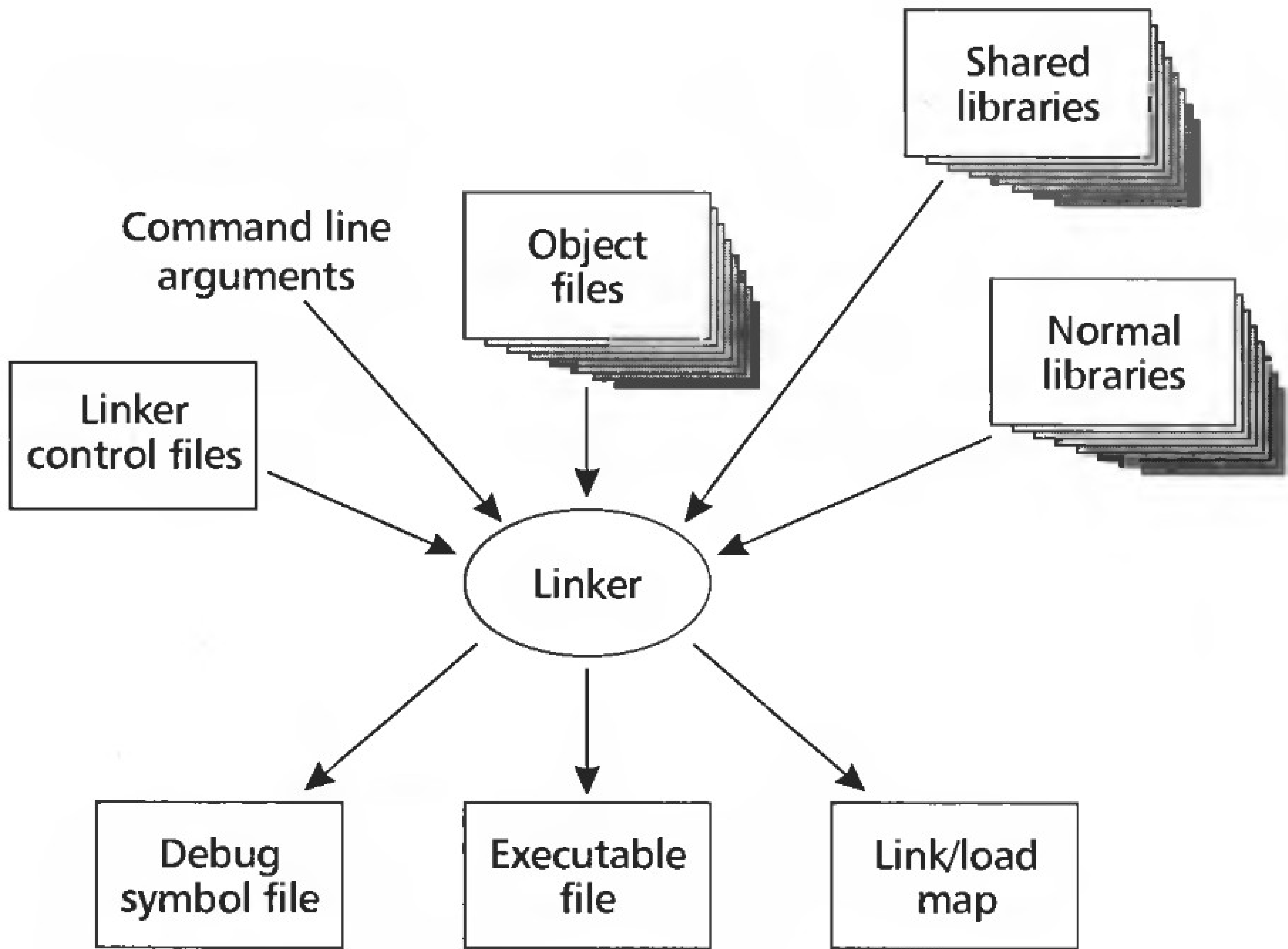
Lecture 20: Program linking and loading

Anton Burtsev

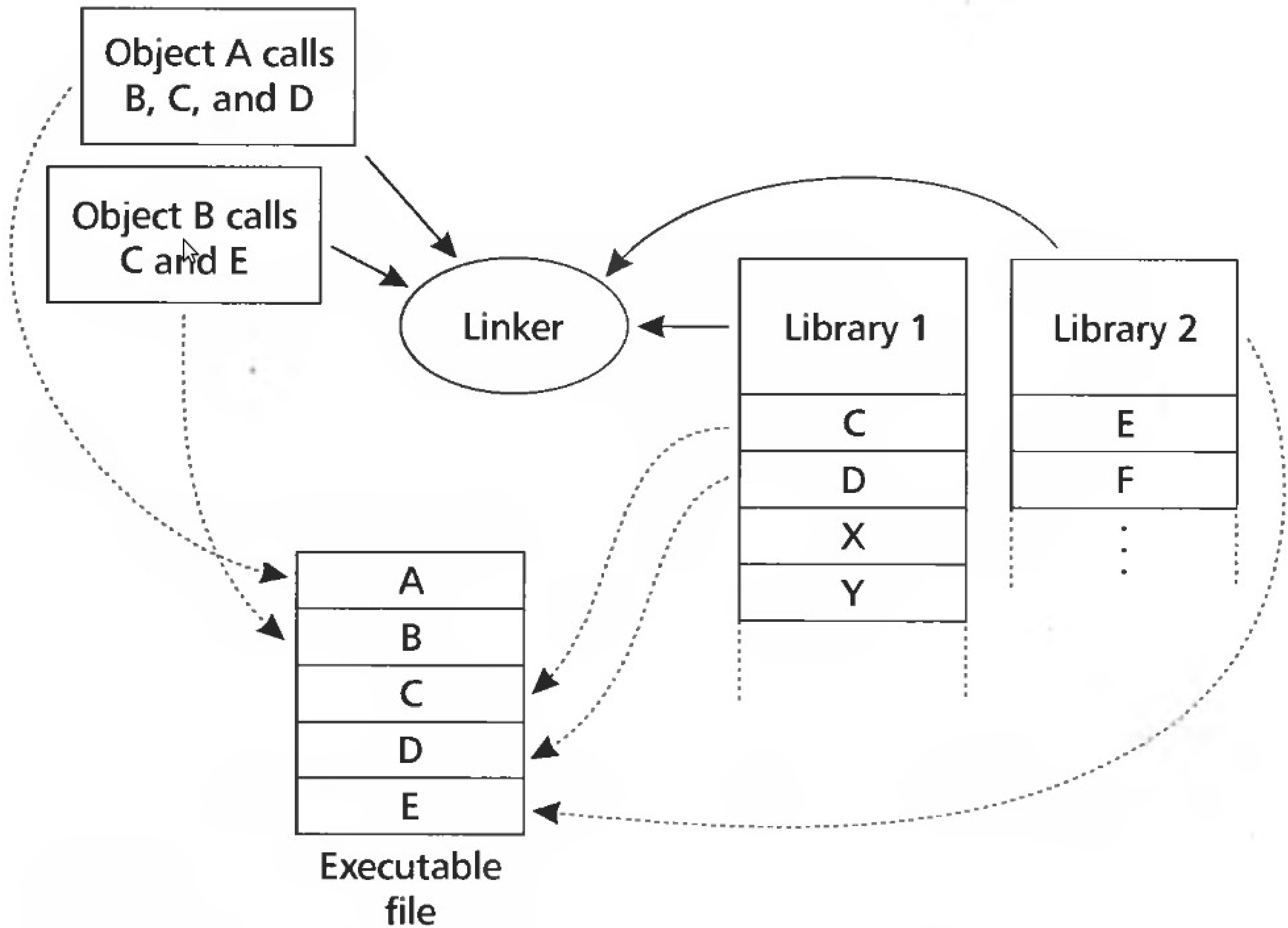
March, 2014

Linking and loading

- Linking
  - Combining multiple code modules into a single executable
  - E.g., use standard libraries in your own code
- Loading
  - Process of getting an executable running on the machine


Shared libraries
Command line arguments
Object files
Normal libraries
Linker control files
Linker
Debug symbol file
Executable file
Link/load map

- Input: object files (code modules)
- Each object file contains
  - A set of segments
    - Code
    - Data
  - A symbol table
    - Imported & exported symbols
- Output: executable file, library, etc.


Object A calls
B, C, and D
Object B calls
C and E
Linker
Library 1
C
D
X
Y
Library 2
E
F
A
B
C
D
E
Executable
file

Why linking?

- Modularity
  - Program can be written as a collection of modules
  - Can build libraries of common functions
- Efficiency
  - Code compilation
    - Change one source file, recompile it, and re-link the executable
  - Space efficiency
    - Share common code across executables
    - On disk and in memory
    -

Two path process

- Path 1: scan input files
  - Identify boundaries of each segment
  - Collect all defined and undefined symbol information
  - Determine sizes and locations of each segment
- Path 2
  - Adjust memory addresses in code and data to reflect relocated segment addresses

Example

- Save a into b, e.g., b = a

```
mov a,%eax
mov %eax,b
```

- Generated code
  - a is defined in the same file at 0x1234, b is imported
  - Each instruction is 1 byte opcode + 4 bytes address

```
A1 34 12 00 00 mov a,%eax
A3 00 00 00 00 mov %eax,b
```

- Assume that a is relocated by 0x10000 bytes, and b is found at 0x9a12

```
A1 34 12 01 00 mov a,%eax
A3 12 9A 00 00 mov %eax,b
```

More realistic example

- Source file m.c

```
extern void a(char *);
int main(int ac, char **av)
{
  static char string[] = "Hello, world!\n";
  a(string);
}
```

- Source file a.c

```
#include <unistd.h>
#include <string.h>
void a(char *s)
{
  write(1, s, strlen(s));
}
```

More realistic example

```
Sections:
 Idx Name Size     VMA      LMA      File off Algn
  0 .text 00000010 00000000 00000000 00000020 2**3
  1 .data 00000010 00000010 00000010 00000030 2**3
Disassembly of section .text:
00000000 <_main>:
  0: 55             pushl %ebp
  1: 89 e5          movl %esp,%ebp
  3: 68 10 00 00 00 pushl $0x10
    4: 32 .data
  8: e8 f3 ff ff ff call 0
    9: DISP32 _a
  d: c9             leave
  e: c3             ret
  ...
```

More realistic example

- Two sections:
  - Text (0x10 – 16 bytes)
  - Data

```
Sections:
 Idx Name  Size     VMA      LMA      File off Algn
   0 .text 00000010 00000000 00000000 00000020 2**3
   1 .data 00000010 00000010 00000010 00000030 2**3
Disassembly of section .text:
00000000 <_main>:
   0: 55             pushl %ebp
   1: 89 e5          movl %esp,%ebp
   3: 68 10 00 00 00 pushl $0x10
     4: 32 .data
   8: e8 f3 ff ff ff call 0
     9: DISP32 _a
   d: c9             leave
   e: c3             ret
   ...
```

More realistic example

```
Sections:
 Idx Name Size      VMA      LMA      File off Algn
   0 .t             000000 00000000 00000020 2**3
   1 .d             000010 00000010 00000030 2**3
Disassem  y of section .text:
00000000 <_main>:
   0: 55             pushl %ebp
   1: 89 e5          movl %esp,%ebp
   3: 68 10 00 00 00 pushl $0x10
     4: 32 .data
   8: e8 f3 ff ff ff call 0
     9: DISP32 _a
   d: c9             leave
   e: c3             ret
   ...
```

• Code starts at 0x0

More realistic example

```
Sections:
 Idx Name Size     VMA      LMA      File off Algn
  0 .text 00000010 00000000 00000000 00000020 2**3
  1 .data 00000010 00000010 00000010 00000030 2**3
Disassembly of section .text:
00000000 <_main>:
  0: 55             pushl %ebp
  1: 89 e5          movl %esp,%ebp
  3: 68 10 00 00 00 pushl $0x10
    4: 32 .data
  8: e8 f3 ff ff ff call 0
    9: DISP32 _a
  d: c9             leave
  e: c3             ret
  ...
```

- First relocation entry
  - Marks pushl 0x10
  - 0x10 is beginning of the data section

More realistic example

```
Sections:
 Idx Name Size     VMA      LMA      File off Algn
  0 .text 00000010 00000000 00000000 00000020 2**3
  1 .data 00000010 00000010 00000010 00000030 2**3
Disassembly of section .text:
00000000 <_main>:
  0: 55             pushl %ebp
  1: 89 e5          movl %esp,%ebp
  3: 68 10 00 00 00 pushl $0x10
    4: 32 .data
  8: e8 f3 ff ff ff call 0
    9: DISP32 _a
  d: c9             leave
  e: c3             ret
  ...
```

- Second relocation entry
  - Marks call
  - 0x10 is beginning of the data section

More realistic example

```
Sections:
 Idx Name Size     VMA      LMA      File off Algn
  0 .text 0000001c 00000000 00000000 00000020 2**2
    CONTENTS, ALLOC, LOAD, RELOC, CODE
  1 .data 00000000 0000001c 0000001c 0000003c 2**2
    CONTENTS, ALLOC, LOAD, DATA
Disassembly of section .text:
  00000000 <_a>:
  0: 55              pushl %ebp
  1: 89 e5           movl %esp,%ebp
  3: 53              pushl %ebx
  4: 8b 5d 08        movl 0x8(%ebp),%ebx
  7: 53              pushl %ebx
  8: e8 f3 ff ff ff  call 0
    9: DISP32 _strlen
  d: 50              pushl %eax
  e: 53              pushl %ebx
  f: 6a 01           pushl $0x1
  11: e8 ea ff ff ff call 0
    12: DISP32 _write
  16: 8d 65 fc       leal -4(%ebp),%esp
  19: 5b             popl %ebx
  1a: c9             leave
  1b: c3             ret
```

Producing an executable

- Combine corresponding segments from each object file
  - Combined text segment
  - Combined data segment
- Pad each segment to 4KB to match the page size

```
Sections:
 Idx Name Size      VMA      LMA      File off Algn
  0 .text 00000fe0 00001020 00001020 00000020 2**3
  1 .data 00001000 00002000 00002000 00001000 2**3
  2 .bss  00000000 00003000 00003000 00000000 2**3
Disassembly of section .text:
00001020 <start-c>:
  ...
  1092: e8 0d 00 00 00 call 10a4 <_main>
  ...
000010a4 <_main>:

  10a7: 68 24 20 00 00 pushl $0x2024
  10ac: e8 03 00 00 00 call 10b4 <_a>
  ...
000010b4 <_a>:

  10bc: e8 37 00 00 00 call 10f8 <_strlen>
  ...
  10c3: 6a 01 pushl $0x1
  10c5: e8 a2 00 00 00 call 116c <_write>
  ...
000010f8 <_strlen>:
  ...
0000116c <_write>:
  ...
```

Linked executable

Tasks involved

- Program loading
  - Copy a program from disk to memory so it is ready to run
    - Allocation of memory
    - Setting protection bits (e.g. read only)
- Relocation
  - Assign load address to each object file
  - Adjust the code
- Symbol resolution
  - Resolve symbols imported from other object files

Object files

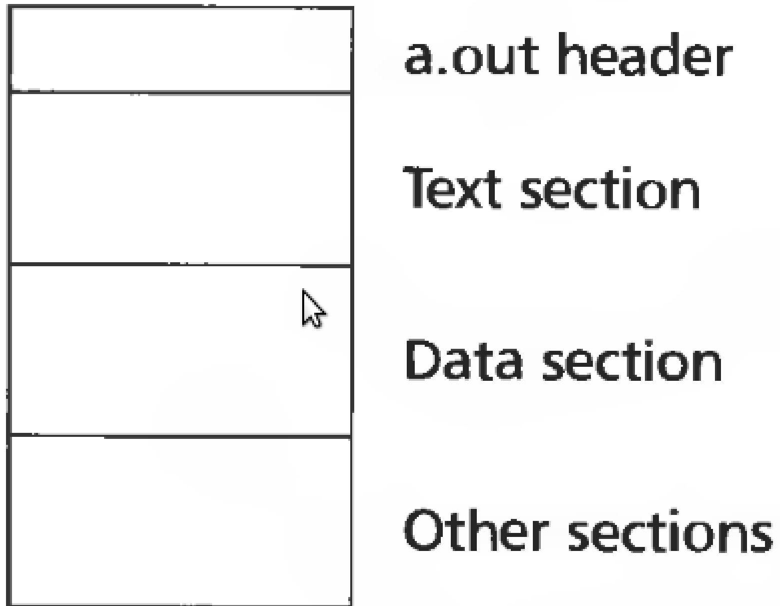
Object files

- Conceptually: five kinds of information
  - Header: code size, name of the source file, creation date
  - Object code: binary instruction and data generated by the compiler
  - Relocation information: list of places in the object code that need to be patched
  - Symbols: global symbols defined by this module
    - Symbols to be imported from other modules
  - Debugging information: source file and file number information, local symbols, data structure description

Simplest object file: DOS .com

- Only binary code
  - Loaded at 0x100 offset
  - 0x00 – 0xFF is reserved for program prefix
    - Command line arguments
- Set EIP to 0x100
- Set ESP to the top of the segment
- Run!

UNIX A.OUT



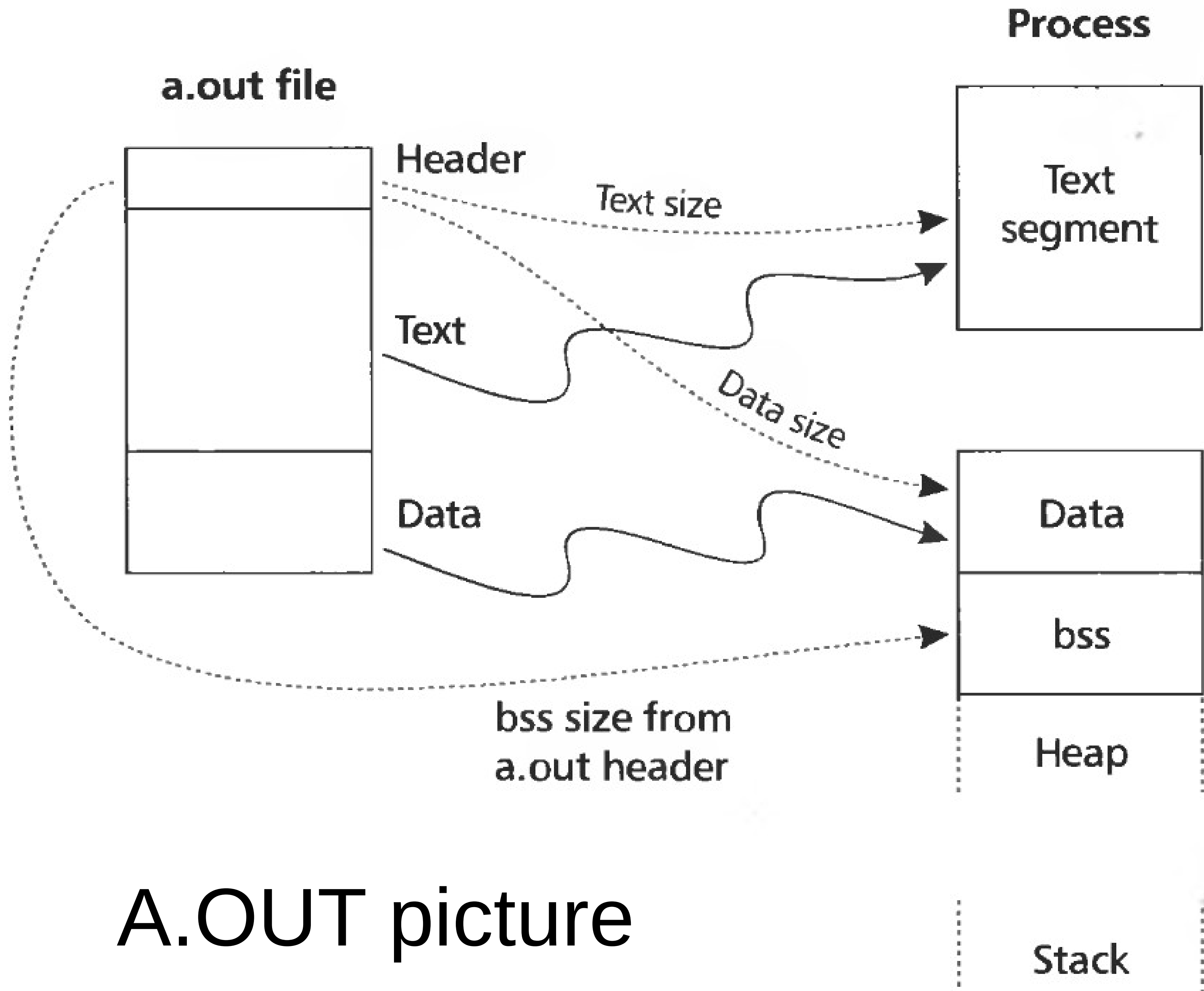
- Small header
- Text section
  - Executable code
- Data section
  - Initial values for static data

- A.OUT header

```
int a_magic;   // magic number
int a_text;    // text segment size
int a_data;    // initialized data size
int a_bss;     // uninitialized data size
int a_syms;    // symbol table size
int a_entry;   // entry point
int a_trsize;  // text relocation size
int a_drsize;  // data relocation size
```

A.OUT loading

- Read the header to get segment sizes
- Check if there is a shareable code segment for this file
  - If not, create one,
  - Map into the address space,
  - Read segment from a file into the address space
- Create a private data segment
  - Large enough for data and BSS
  - Read data segment, zero out the BSS segment
- Create and map stack segment
  - Place arguments from the command line on the stack
- Jump to the entry point

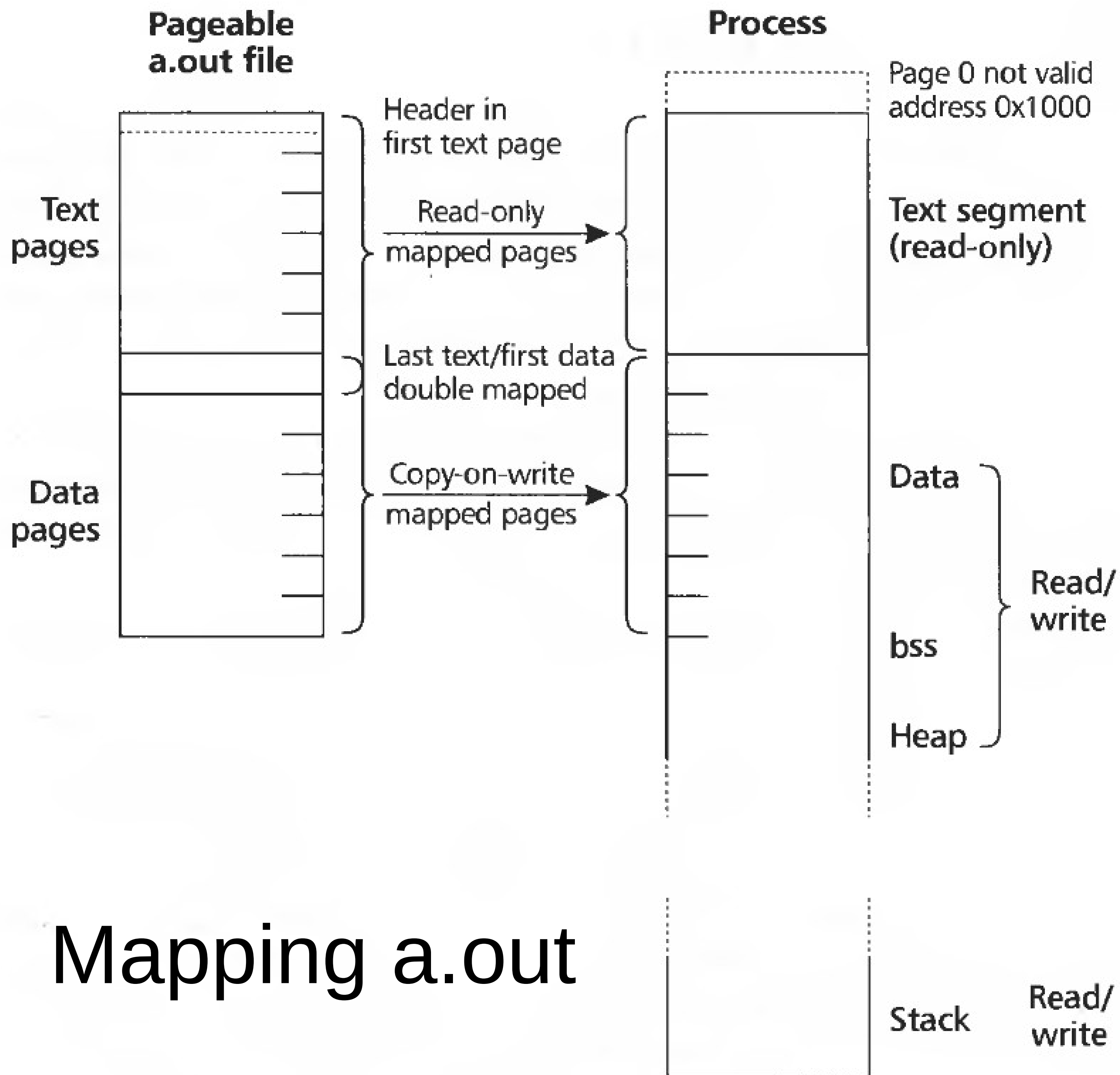


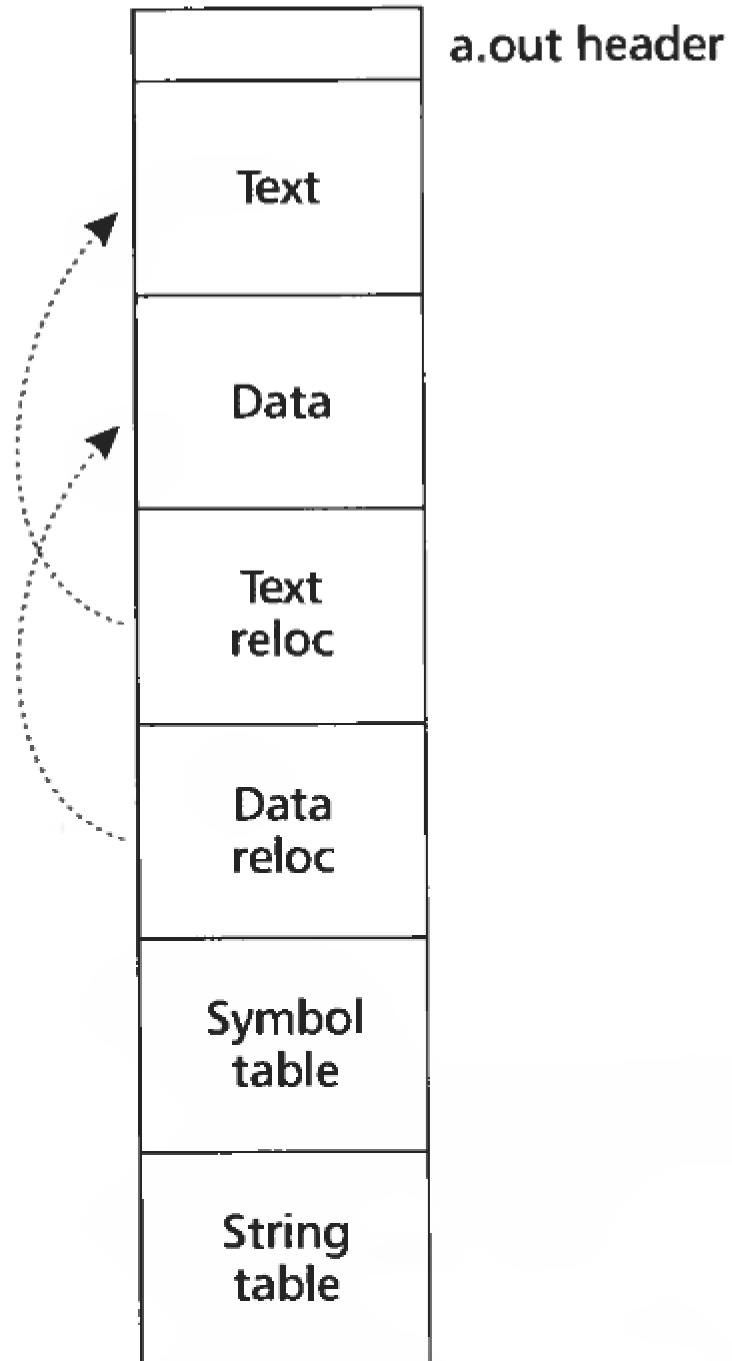
A.OUT picture

Interaction with virtual memory

- Virtual memory unifies paging and file I/O
  - Memory mapped files
    - Memory pages are backed up by files
  - Loading a segment is just mapping it into memory
- Linker must provide some support
  - Sections are page aligned

Mapping a.out

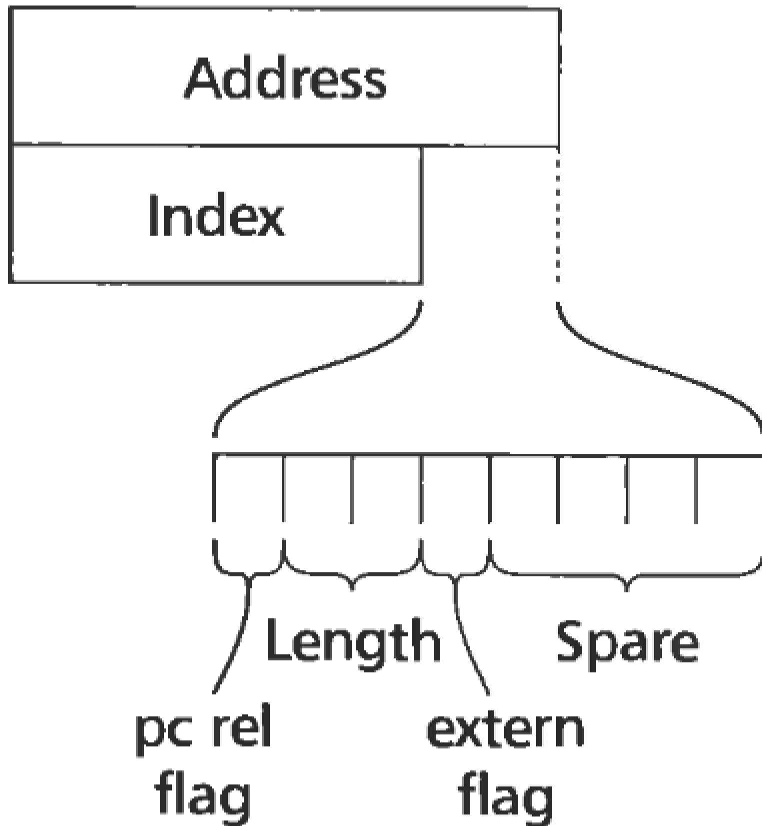




Relocatable A.OUT

- Add relocation information for each section

Relocation entries



- Address relative to the segment
- Length
  - 1, 2, 4, 8 bytes
- Extern
  - Local or extern symbol
- Index
  - Segment number if local
  - Index in the symbol table

Symbol table

Name offset		
Type	Spare	Debug info
Value		

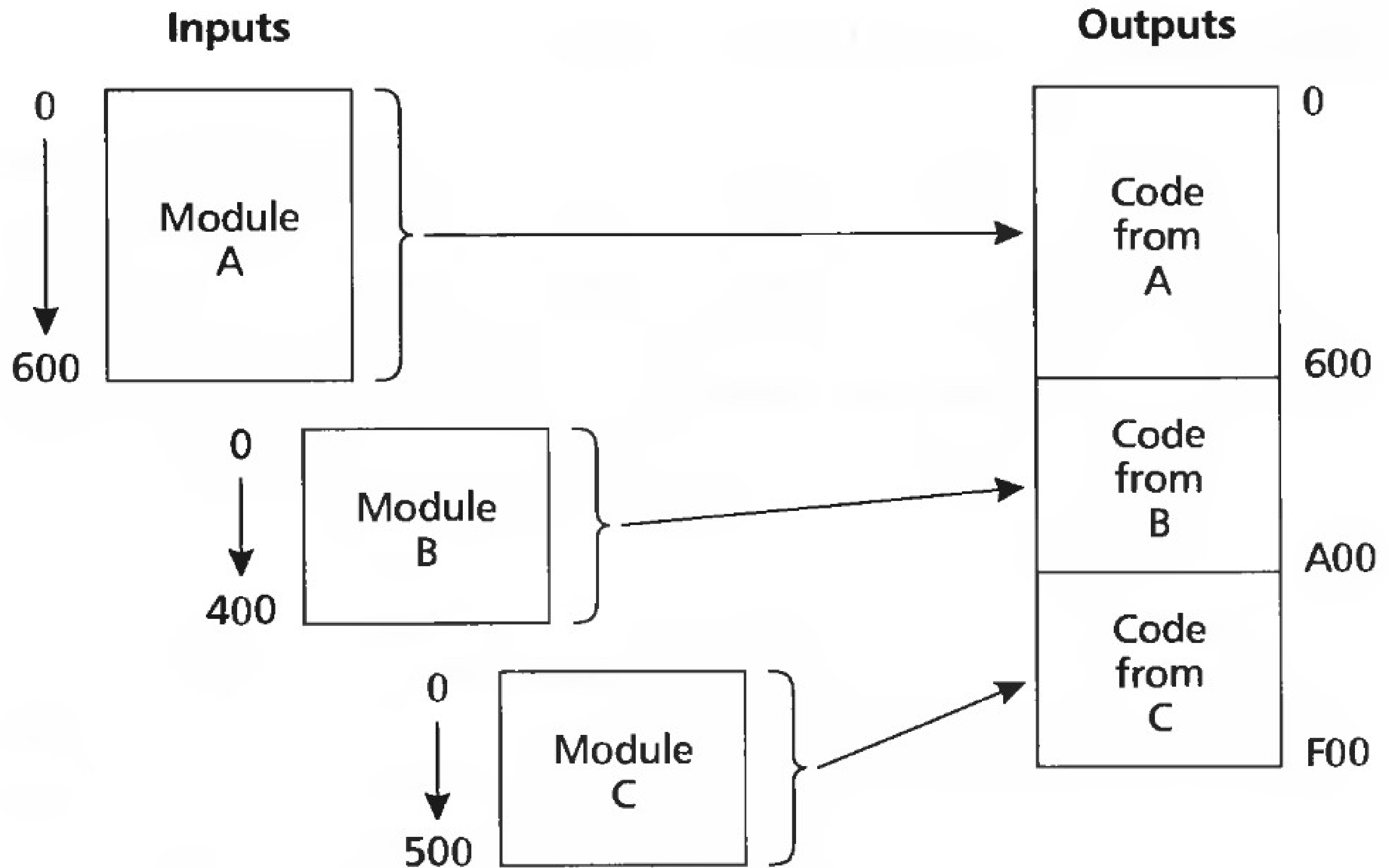
- Name offset
  - Offset into the string table
  - UNIX supports symbols of any length
    - Null terminated strings
- Type
  - Whether it is visible to other modules

Weak vs strong symbols

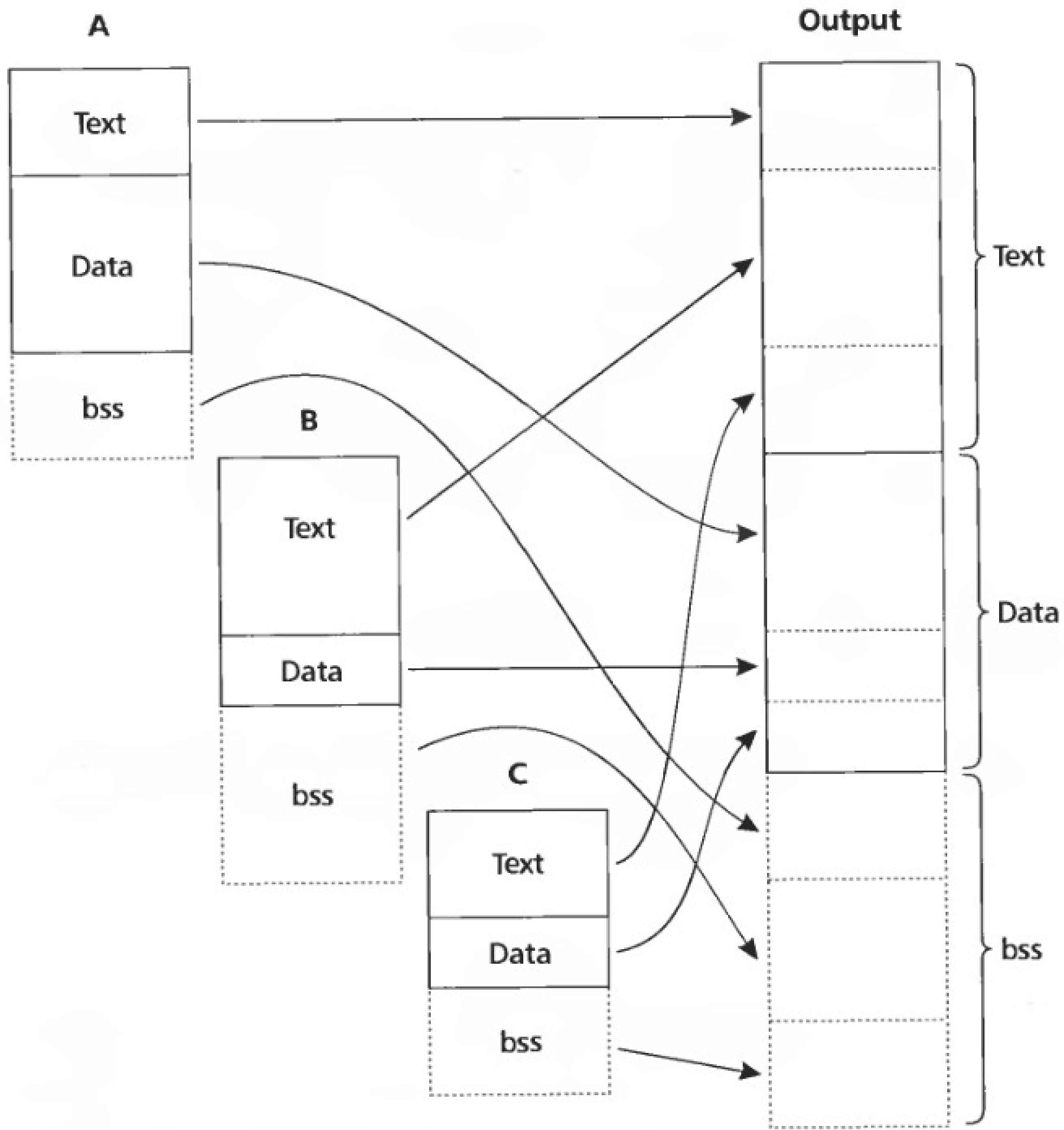
- Virtually every program uses printf
  - Printf can convert floating-point numbers to strings
    - Printf uses fcvt()
  - Does this mean that every program needs to link against floating-point libraries?
- Weak symbols allow symbols to be undefined
  - If program uses floating numbers, it links against the floating-point libraries
    - fcvt() is defined an everything is fine
  - If program doesn't use floating-point libraries
    - fcvt() remains NULL but is never called

Storage allocation

Multiple object files



Merging segments



Initializers and finalizers

- C++ needs a segment for invoking constructors for static variables
  - List of pointers to startup routines
    - Startup code in every module is put into an anonymous startup routine
    - Put into a segment called .init
- Problem
  - Order matters
  - Ideally you should track dependencies
    - This is not done
  - Simple hack
    - System libraries go first (.init), then user (.ctor)
    -

Relocation

Why relocate?

- Each program gets its own private space, why relocate?
  - Linkers combine multiple libraries into a single executable
  - Each library assumes private address space
    - E.g., starts at 0x0
- Is it possible to go away with segments?
  - Each library gets a private segment (starts at 0x0)
  - All cross-library references are patched to use segment numbers
- Possible!
  - But slow.
  - Segment lookups are slow

Relocation

- Each relocatable object file contains a relocation table
  - List of places in each segment that need to be relocated
  - Example:
    - Pointer in the text segment points to offset 200 in the data segment
    - Input file: text starts at 0, data starts at 2000, stored pointer has value 2200
    - Output file: Data segment starts at 15000
      - Linker adds relocated base of the data segment 13000 (DR)
    - Output file: will have pointer value of 15200
  - All jumps are relative on x86
    - No need to relocate
    - Unless its a cross-segment jump, e.g. text segment to data segment

Conclusion

- Program loading
  - Storage allocation
- Relocation
  - Assign load address to each object file
  - Patch the code
- Symbol resolution
  - Resolve symbols imported from other object files

Next time

- Static and shared libraries
- Dynamic linking and loading
- Position independent code
- OS management of user space

Thank you!