Duke Systems

CPS 310 / ECE 353

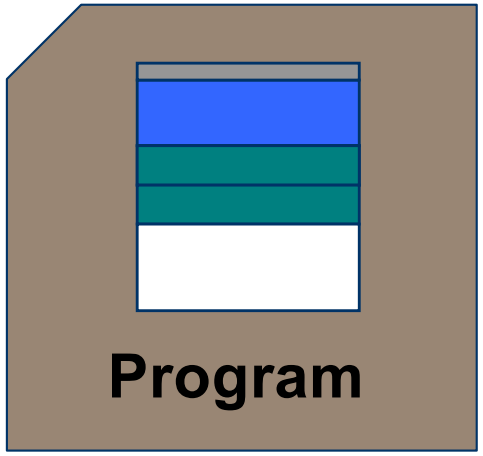
Programs and Processes

Jeff Chase

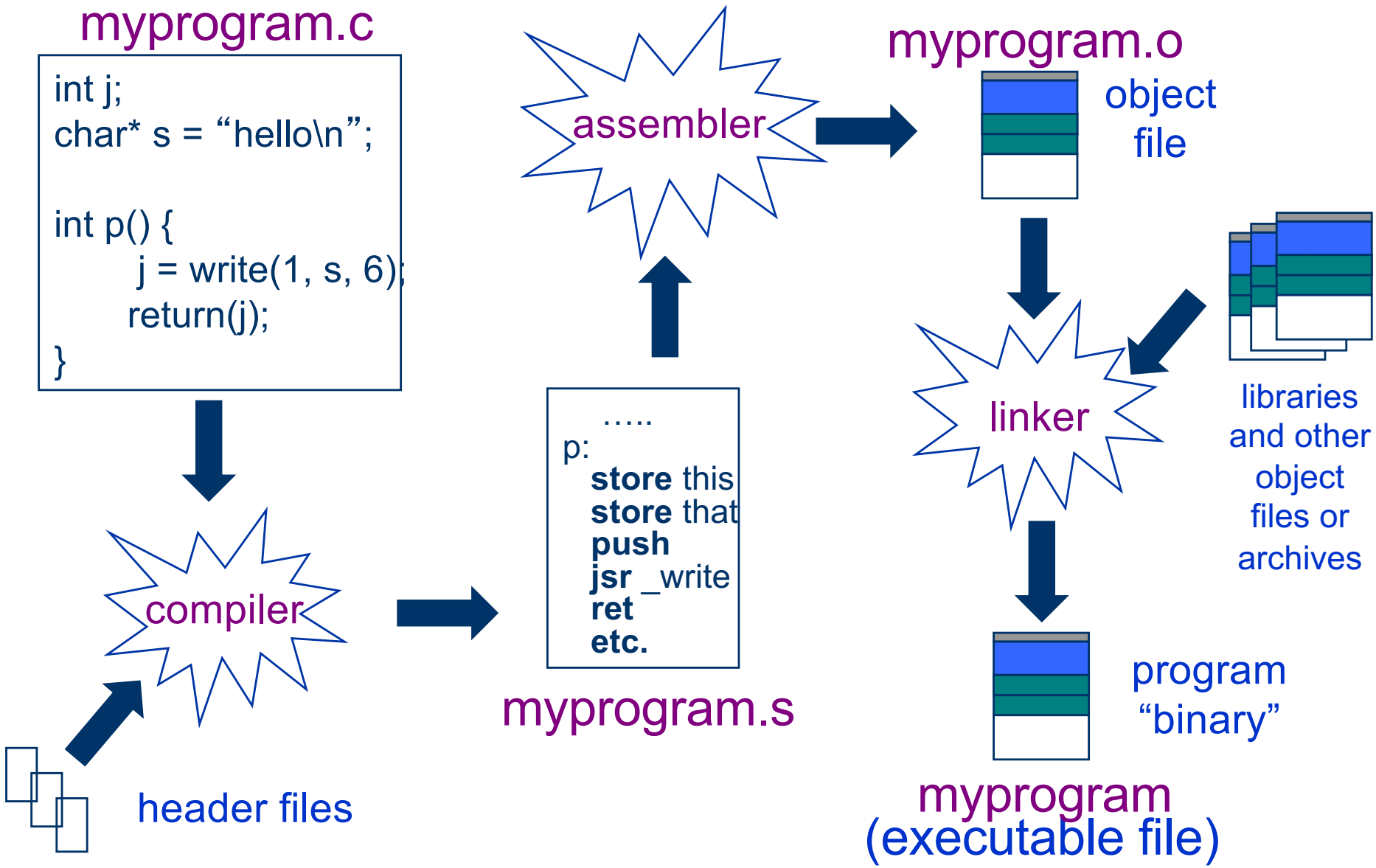
Duke University

What is a program?

- Is it a stupid question?
- A program is data that describes a process. ;-)
- For our purposes today, the data is stored in a file, in a format that the operating system can **execute**.
  - An **executable** or "**binary**" file
- Let's suppose that it is **statically linked** (self-contained).



Birth of a program (C/Ux)

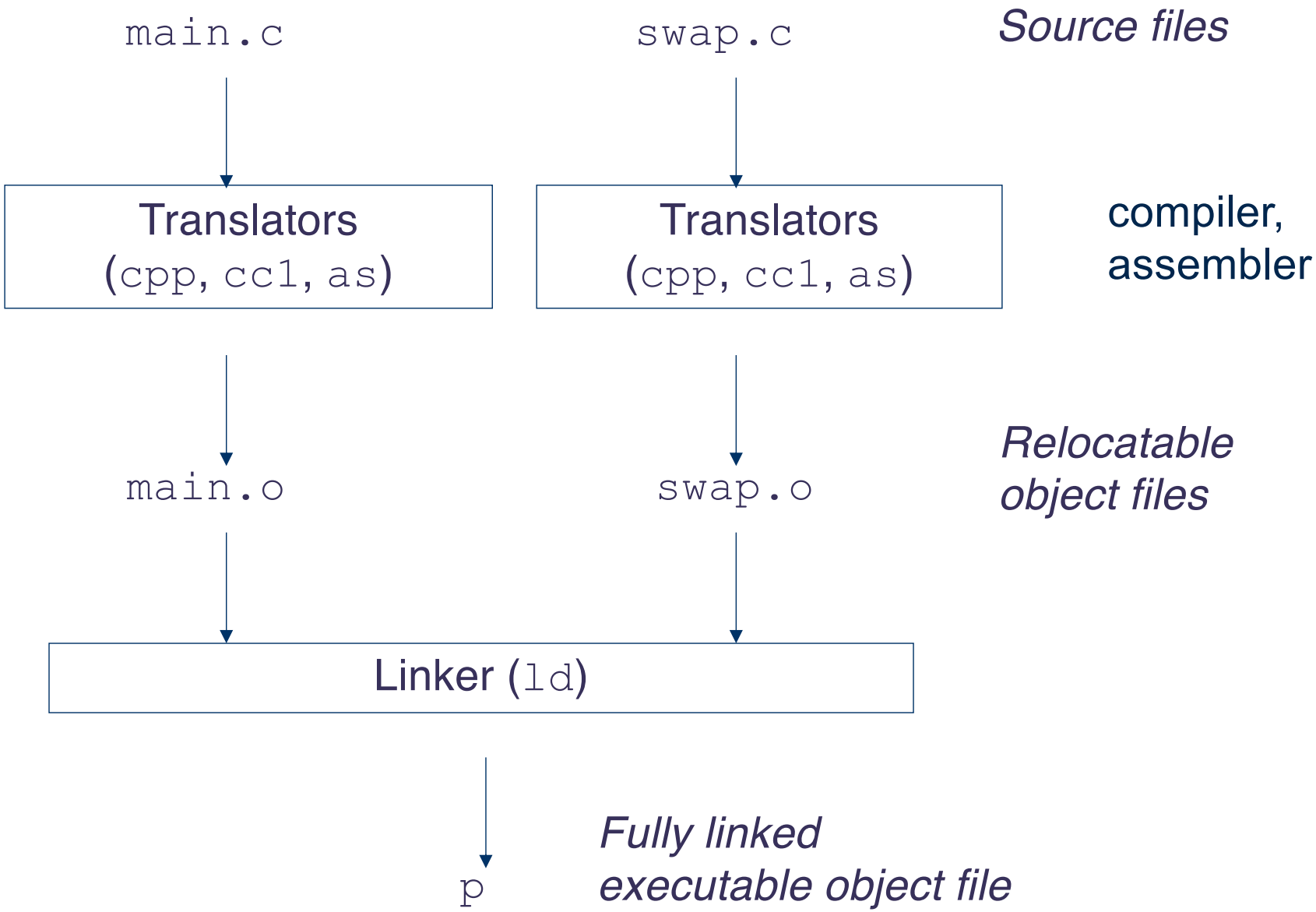


Role of the linker in the build toolchain

What's this all about?

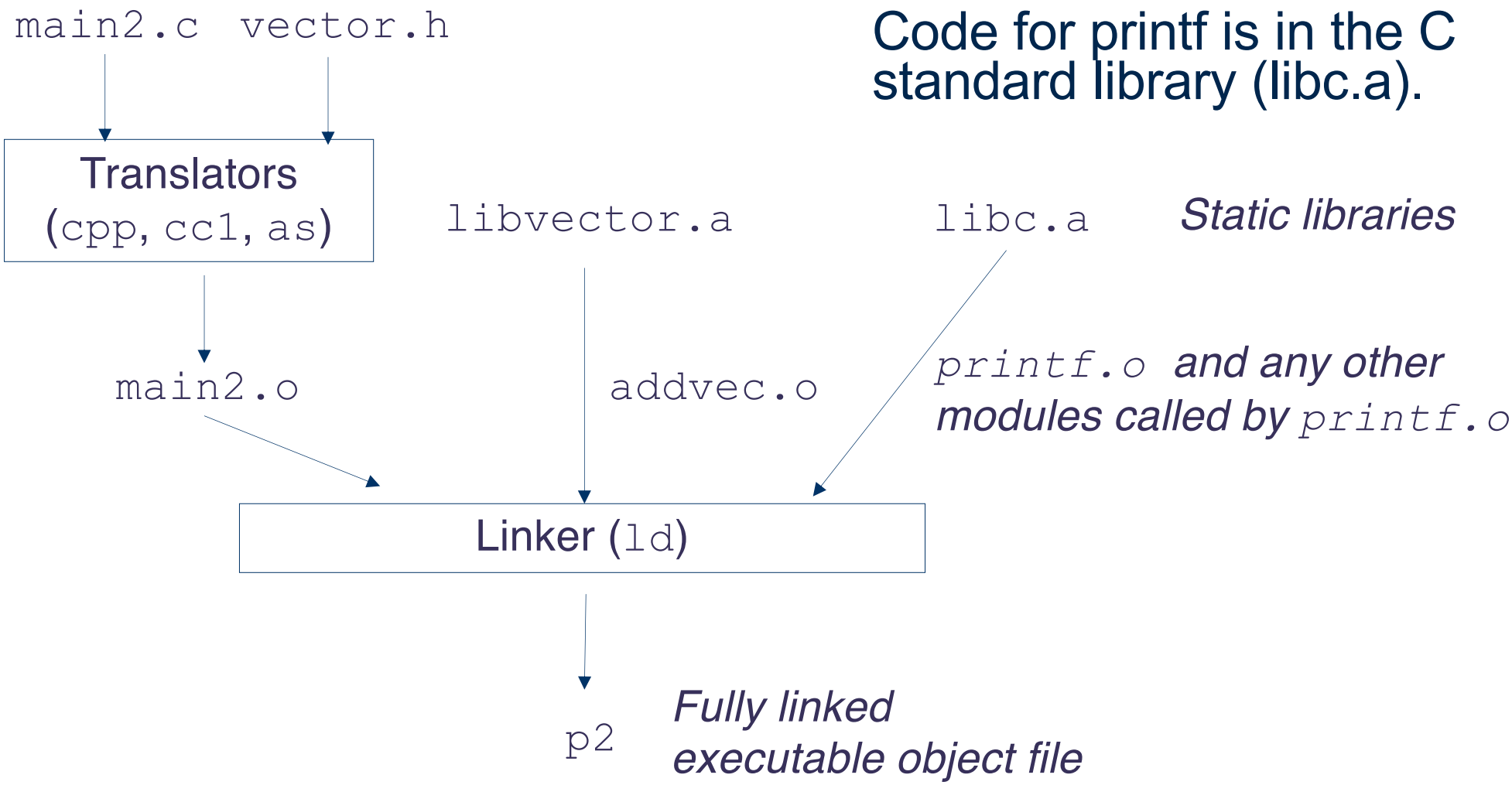
- Real programs are written in pieces by different people.
- Their code is spread across multiple files.
- They include "off the shelf" code used by other programs.
  - **Libraries**, e.g., the C **standard library** (libc, stdlib).
- We can compile the pieces separately, at different times.
- A piece might reference symbols defined in another.
- The linker combines the pieces into a program.
- C compiler also runs linker for you (unless –c), with stdlib.

Separate compilation and the linker



[cs:app]

Static linking with libraries (archives)



Code for printf is in the C standard library (libc.a).

[cs:app]

Where to start? main()

```
chase$ cc -c empty.c
chase$ file empty.o
empty.o: Mach-O 64-bit object x86_64
chase$ nm empty.o
0000000000000000 T _p
chase$ cc empty.c
Undefined symbols for architecture x86_64:
  "_main", referenced from:
     implicit entry/start for main executable
ld: symbol(s) not found…
clang: error: linker failed with exit code 1
chase$
```

empty.c

```
void
p() {}
```

This program compiles OK. It defines the procedure p: a symbol of type T for “text” (code). But it is not a complete program: the **linker doesn’t know where to start**! Programs start in main(), but it is missing.

Where to go? Calling out

```
chase$ cc –o nop nop.c
Undefined symbols…:
  "_p", referenced from:
      _main in nop-bbc335.o
ld: symbol(s) not found…
clang: error: linker command failed…
chase$ cc –o nop nop.c empty.o
chase$ ./nop
chase$
```

empty.c

```
void
p() {}
```

nop.c

```
void p();
int main()
   {p();}
```

This program doesn't do anything, but it is a complete program! It has a main(). But it also needs empty.o to define the symbol p, because main() calls p(). Given both files, the linker is happy. **And we want the linker to be happy.**

What's a library?

- A **library** is just a collection of object files bundled together into a single archive file…
- …with an index for fast linking.
- **ar** command utility creates/modifies archives.
- No mystery; no magic!

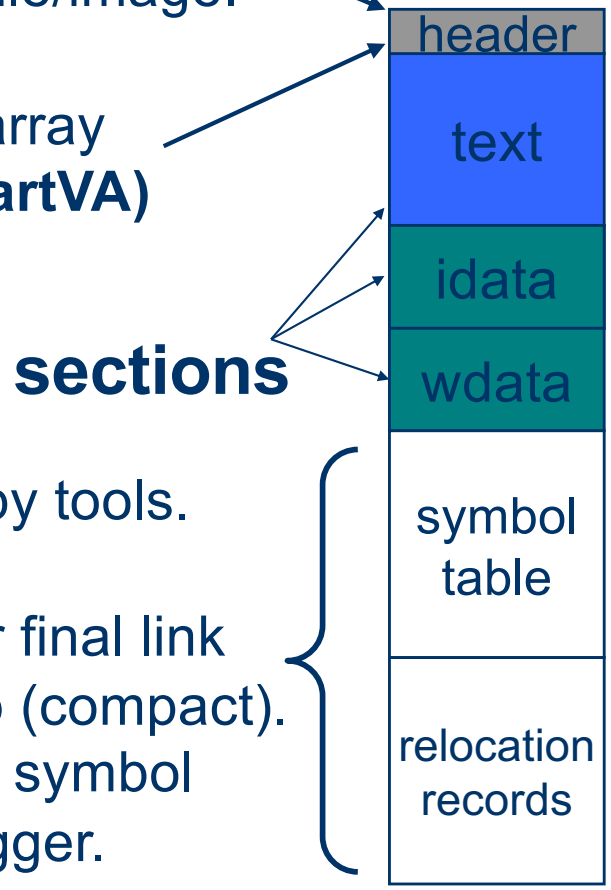
```
chase$ ar cr empty.a empty.o
chase$ file empty.a
empty.a: current ar archive random library
chase$ cc -o nop nop.c empty.a
chase$ ./nop
chase$
```

What's in an object file or executable?

e.g., an ELF or Mach-O file

Header "magic number" indicates type of file/image.

Section table an array of **(offset, len, startVA)**



sections

program instructions
p

immutable data (constants)
"hello\n"

writable global/static data
j, s, sbuf

j, s, p, sbuf

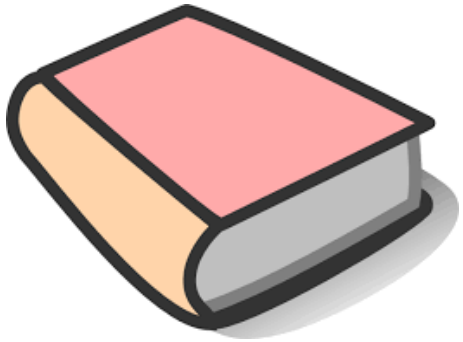
Metadata used by tools.
Options:
- Remove after final link step and strip (compact).
- Or: add more symbol info for debugger.

```
int j = 327;
char* s = "hello\n";
char sbuf[512];

int p() {
        int k = 0;
        j = write(1, s, 6);
       return(j);
}
```

Sections are byte offset ranges within the file.

ELF: read it like a book



An executable/linkable file (e.g., ELF) is “like a book”.

- **Sections.** They are “like chapters”, or…sections.
  - Each section has content of a given purpose/use, needed to link/run the program.
  - E.g., code, initial values of global data, constants.
- **Header**. Cover page and table of contents (ToC).
- **Symbol table**. Detailed ToC: all defined symbols.
- **Relocation records**. Index: lists references to symbols.

Inside ELF

EXECUTABLE AND LINKABLE FORMAT

ANGE ALBERTINI
http://www.corkami.com

```
me@nux:~$ ./mini
me@nux:~$ echo $?
42
```

```
     0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
00: 7F .E .L .F 01 01 01
10: 02 00 03 00 01 00 00 00 60 00 00 08 40 00 00 00
20:                         34 00 20 00 01 00

40: 01 00 00 00 00 00 00 00 00 00 00 08 00 00 00 08
50: 70 00 00 00 70 00 00 00 05 00 00 00

60: BB 2A 00 00 00 B8 01 00 00 00 CD 80
```

MINI

ELF HEADER

IDENTIFY AS AN ELF TYPE
SPECIFY THE ARCHITECTURE

FIELDS	VALUES
e_ident	
EI_MAG	0x7F, "ELF"
EI_CLASS, EI_DATA	1 ELFCLASS32, 1 ELFDATA2LSB
EI_VERSION	1 EV_CURRENT
e_type	2 ET_EXEC
e_machine	3 EM_386
e_version	1 EV_CURRENT
e_entry	0x8000060
e_phoff	0x0000040
e_ehsize	0x0034
e_phentsize	0x0020
e_phnum	0001

PROGRAM HEADER TABLE

EXECUTION INFORMATION

p_type	1 PT_LOAD
p_offset	0
p_vaddr	0x8000000
p_paddr	0x8000000
p_filesz	0x0000070
p_memsz	0x0000070
p_flags	5 PF_R|PF_X

CODE

X86 ASSEMBLY

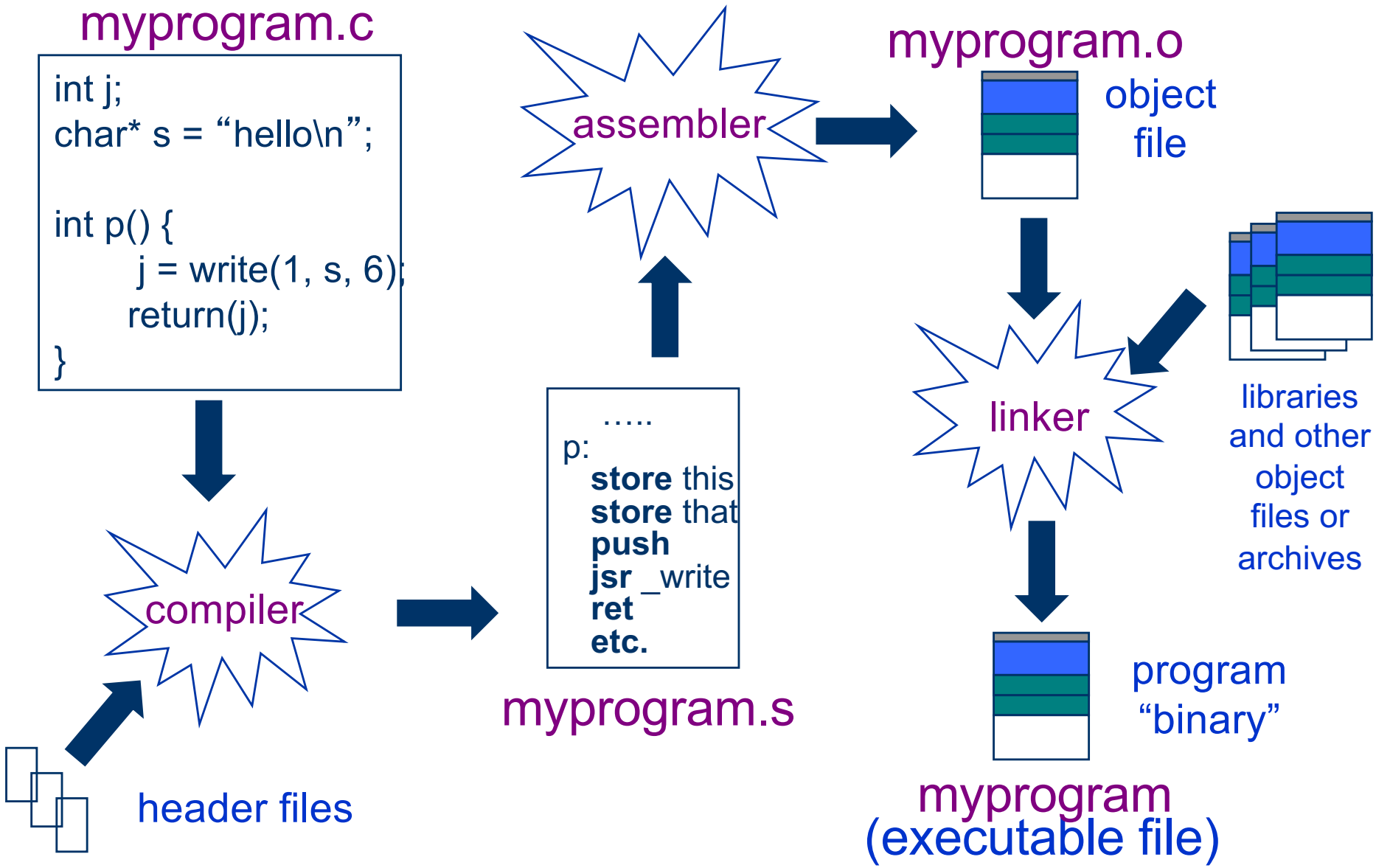
```
mov ebx, 42
mov eax, SC_EXIT[1]
int 80h
```

EQUIVALENT C CODE

```
return 42;
```

Linux uses ELF.
MacOS uses Mach-O.

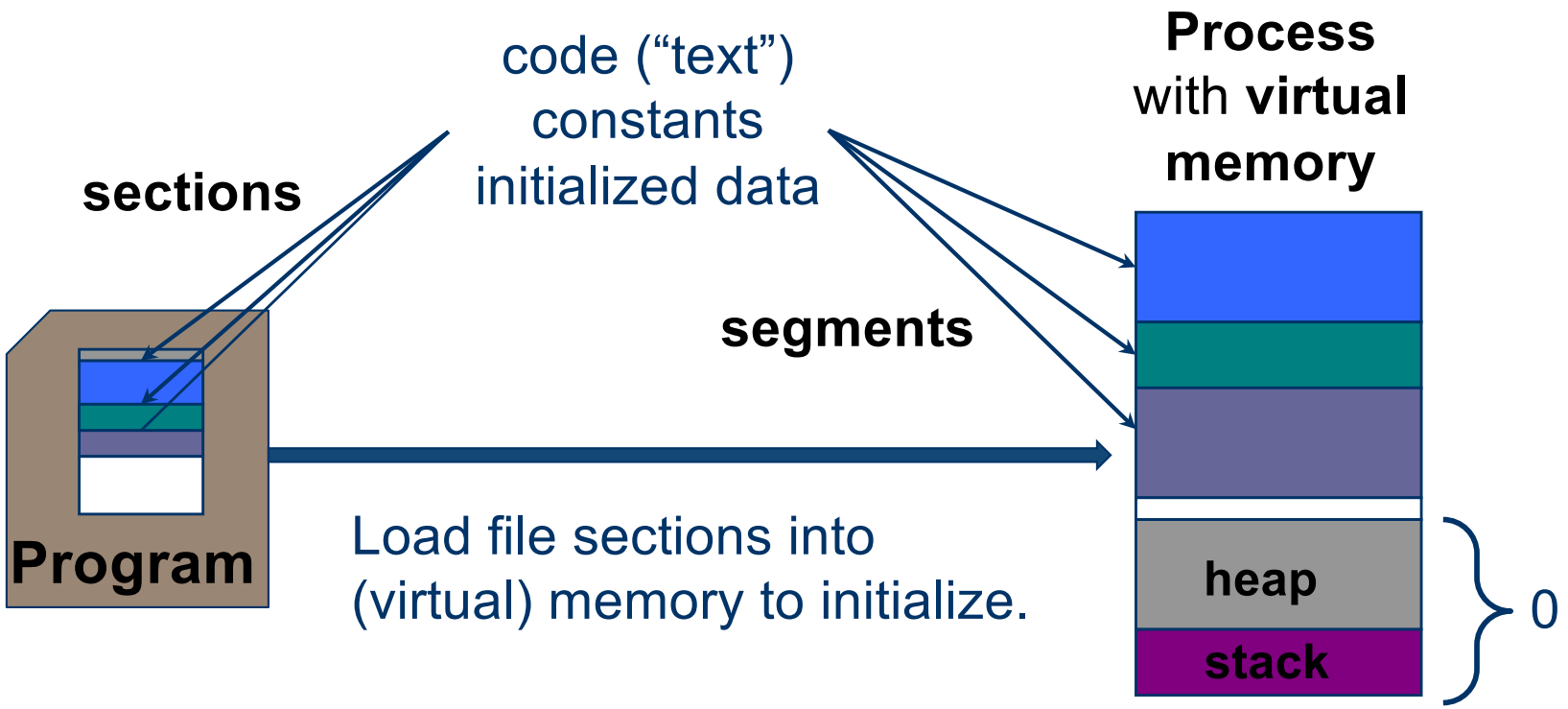
Birth of a program (C/Ux)



Building and running a program

```
chase:p0> make
gcc -I. -g3 -Wall -DNDEBUG -c dmm.c
...
1 warning generated.
gcc -I. -g -Wall -DNDEBUG -o basicdmmtest basicdmmtest.c dmm.o
gcc -I. -g -Wall -DNDEBUG -o test_basic test_basic.c dmm.o
gcc -I. -g -Wall -DNDEBUG -o test_coalesce test_coalesce.c dmm.o
gcc -I. -g -Wall -DNDEBUG -o test_stress1 test_stress1.c dmm.o
gcc -I. -g -Wall -DNDEBUG -o test_stress2 test_stress2.c dmm.o
chase:p0> /test_basic
calling malloc(10)
call to dmalloc() failed
chase:p0>
```

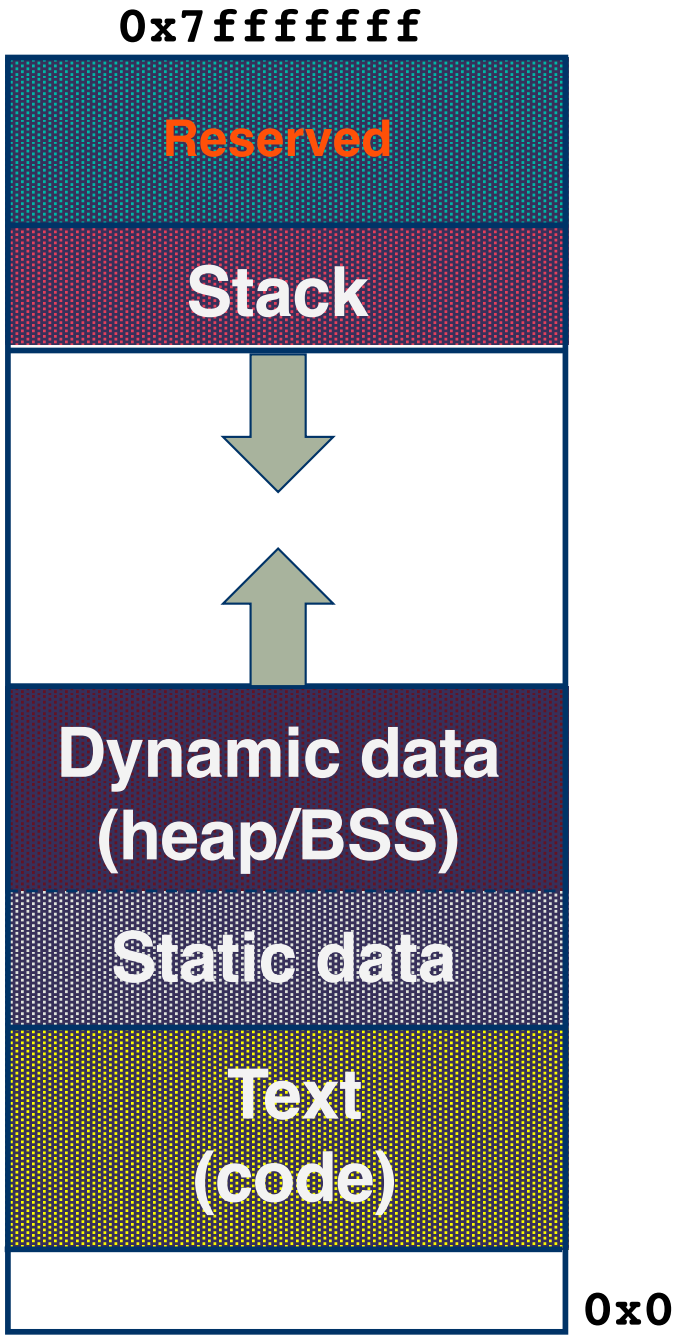
Running a program



When a program launches, the OS initializes a **process** with a **virtual memory** to store the running program's code and data. Sections of the executable file initialize **segments** (regions) of the VM.

Process VM/VAS
(32-bit example)

- The program uses virtual memory through its process' **Virtual Address Space**:
- An addressable array of bytes…
- Containing every instruction it can execute…
- And every piece of data those instructions can reference…
  - E.g., read/write == **load/store** on memory
- Partitioned into logical **segments** (**regions**) with distinct purpose and use.
- Every reference by a running program is interpreted in the context of its VAS.
  - Resolves to a location in machine memory



VM segments: a view from C

- Globals (static data):
  - Fixed-size segment
  - Writable by user program
  - May have initial values
- Text (instructions)
  - Fixed-size segment
  - Executable
  - Not writable
- Heap and stack
  - Variable-size segments
  - Writable
  - Zero-filled on demand