# Linkers

Mendel Rosenblum

# Linkers

Optional readings: Operating Systems: Principles and Practice: None

- First ⅓ of class: CPU issues (threads, processes, synch, sched) **done**

Looking ahead:

- Next ⅓ of class: main memory issues
  - Process memory layout
  - Virtual memory
  - Paging
- Last ⅓ of class:  Storage (Disk/Flash) & File Systems

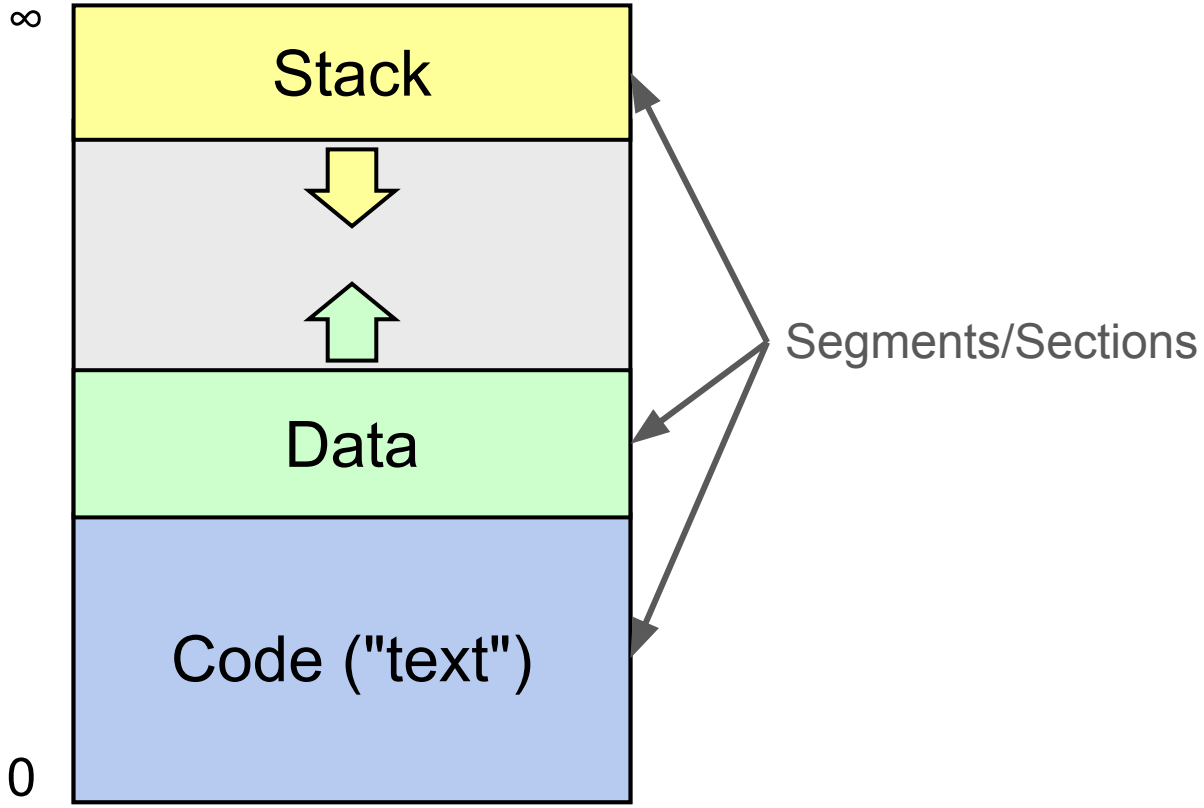
# Main Memory

- Technology:  DRAM (dynamic random access memory)
- Persistence: Volatile
- Accessibility: Byte addressable, accessed in 64 byte (cache line size) units
- Access time: ~60–100 ns, 200-300 CPU cycles
- Capacity:
  - Laptop:  16–64 GB
  - Desktop:  32 - 256 GB
  - Server:  512 - 4096 GB NUMA (non-uniform memory architecture)

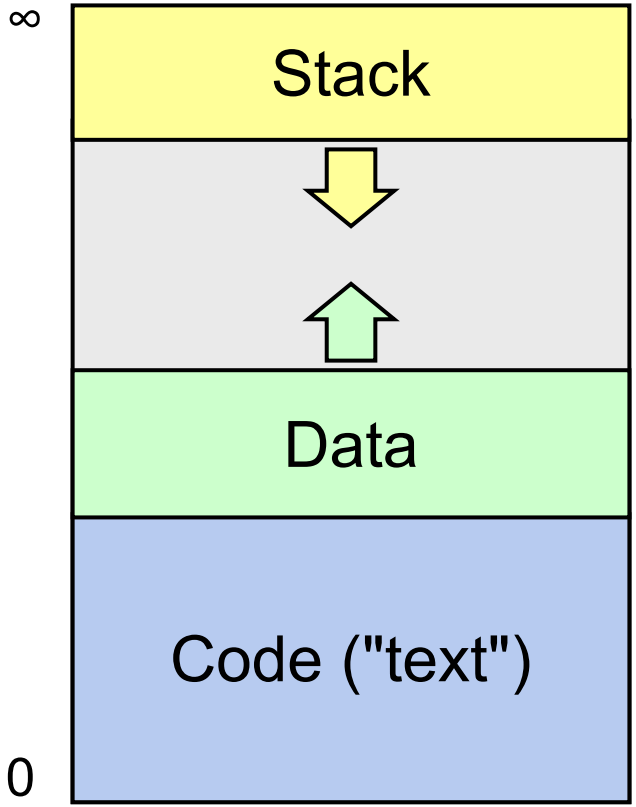
# Application memory of a process

```
int global = 7;           // global data
int* gptr = &global;      // global pointer (contains an address)

void func(int x) {
    int local = x;        // stack variable
    int* lptr = &local;   // stack pointer (contains an address)

    int* heap = new int(42);  // heap object
    lptr = heap;              // pointer now holds heap address

    delete heap;
}
```

# Memory Layout for Process



# Map of program to segments



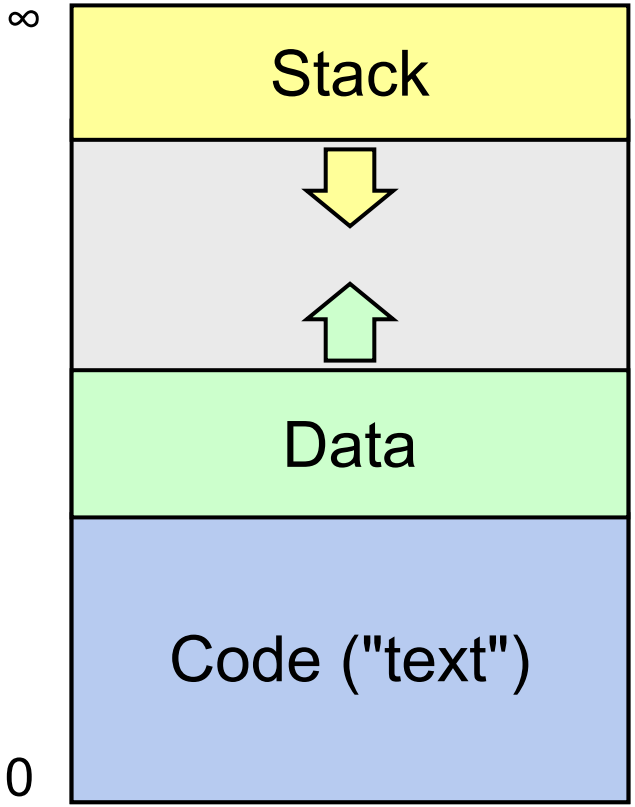

```
int global = 7;
int* gptr = &global;
```

```
void func(int x) {
    int local = x;
    int* lptr = &local;
    int* heap = new int(42);
    lptr = heap;
    delete heap;
}
```

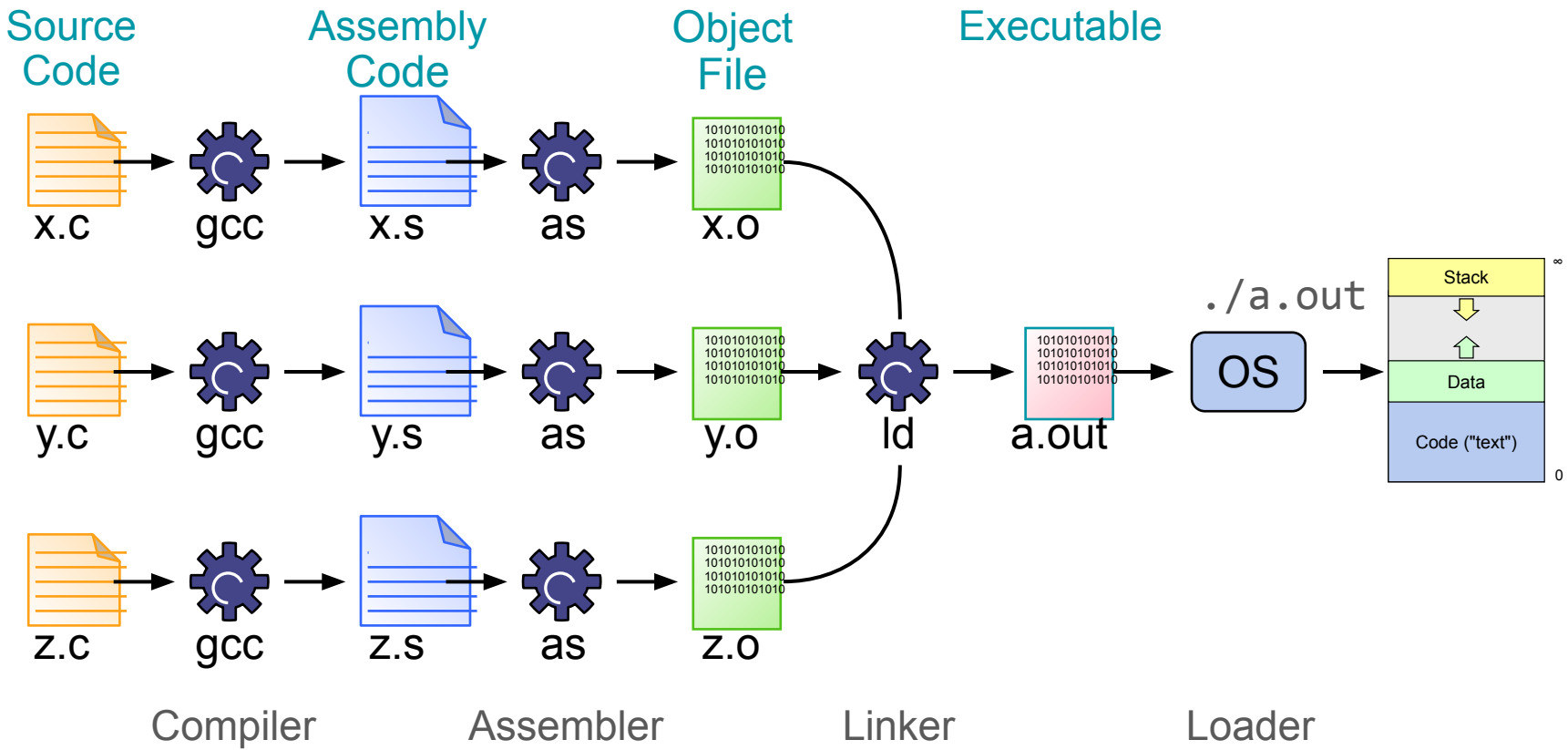
# Memory layout questions

- What if there are 2 processes loaded into memory at once?
  - Who gets address 0?
- Where does the operating system's code and data go?
- What if there are multiple threads within a process?
  - Need private stack on thread
- What if I want to add more code to a running application?

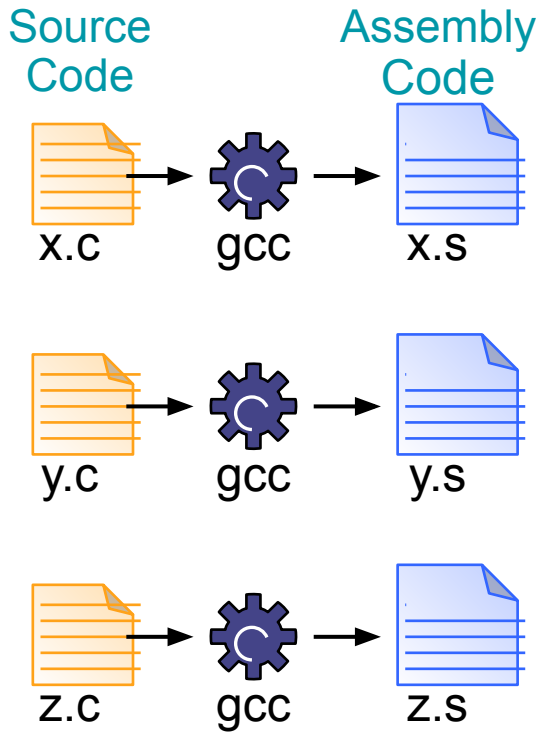
# Will comeback to this figure



# Creating a Process Memory Layout

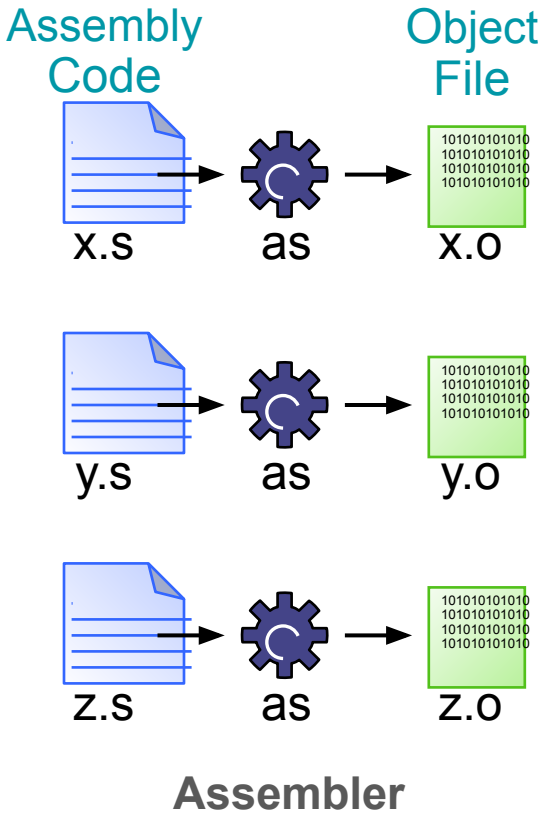


# Compiler transforms source code into assembly code




**Compiler**

# Assembler transforms assembly code into binary



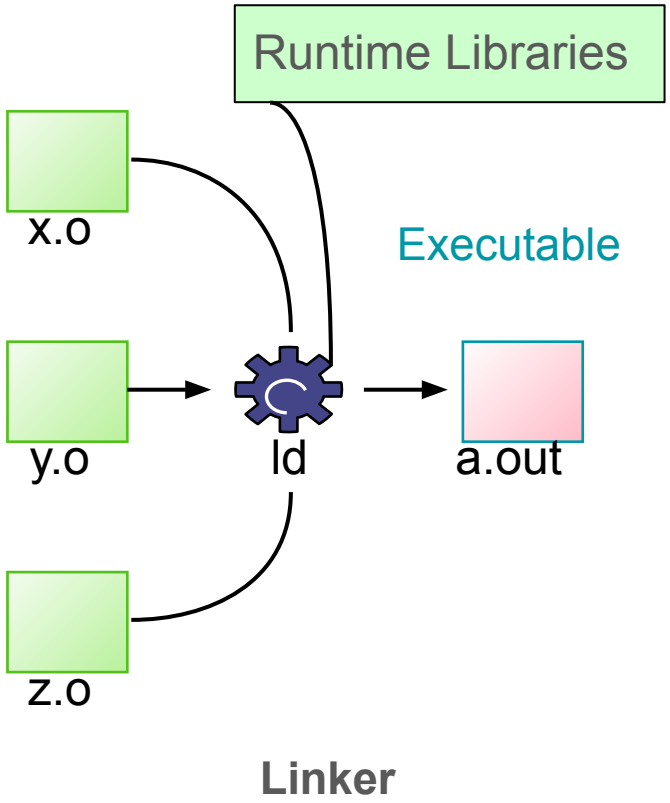

**Object files are incomplete**

Missing information:

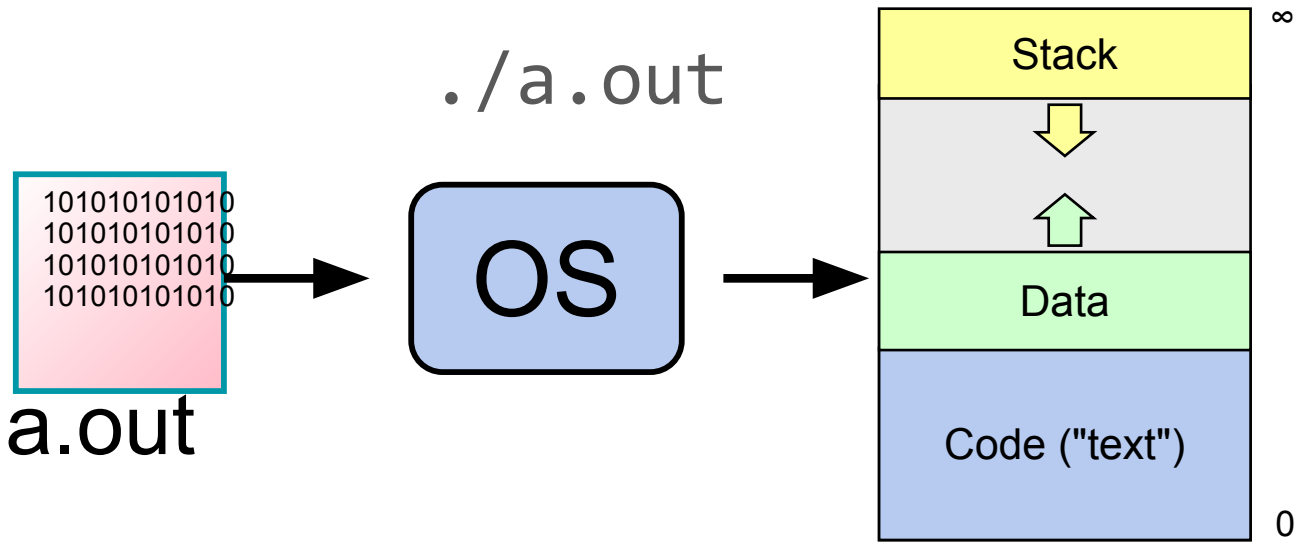
Where things land in memory?

- Function calls
- External data references

# Linkage editor (linker) combines object files



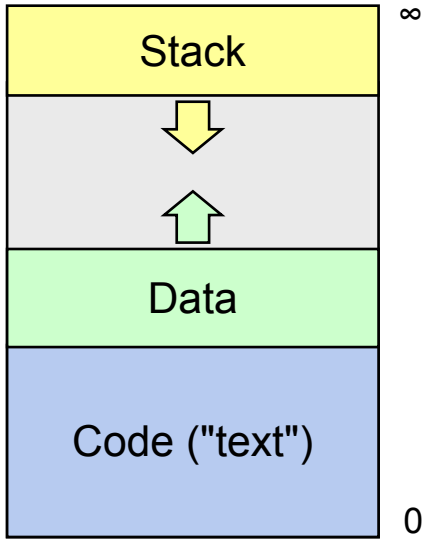
# Loader in OS loads executable into memory




Jumps to the start location in code segment

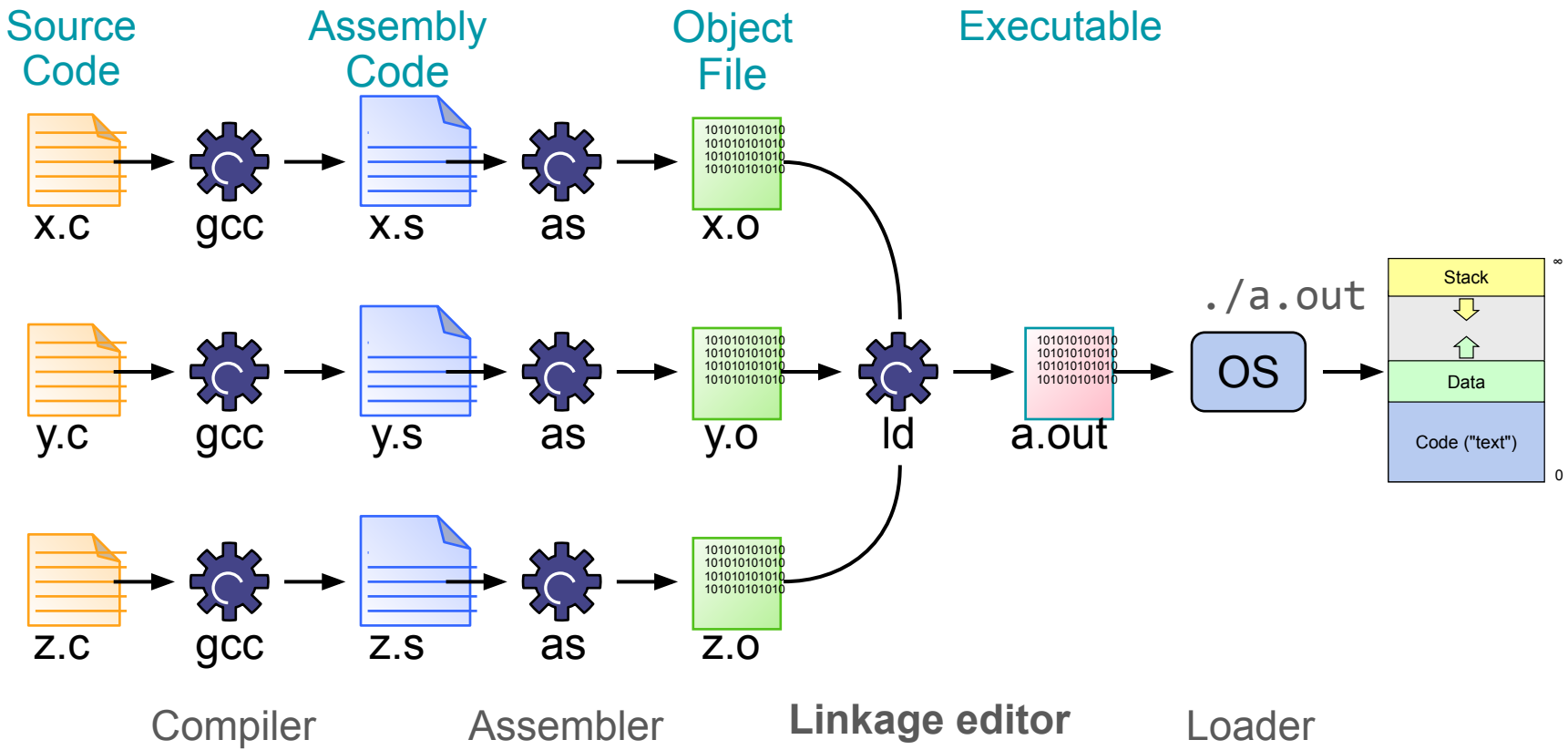
# Runtime libraries

- Memory allocation routines
  - `malloc` in C or `new` in C++
  - Calls into OS to grow data segment
- Stack segments typically grown on demand
- System call stub routines
  - Makes calls into operating system kernel



# Linkage Editors

# Source code to running process



# Linkage Editors

`ld` in Linux, `LINK` on Windows

Challange:  assembler doesn't know ultimate memory locations of things

Assembler use a placeholder value like 0 for unknowns

Linkage editor steps:

1. Combine like things (all code segment)
2. Figures out memory layout
    - How big is code segment?
3. Modifies address to match memory layout
    - Function calls
    - External data references

# Object file contents

- *Sections*: code ("text") and data (stack starts empty):
  - size
  - starting address
  - initial contents (if any)
- *Symbol table*: For routines and (non-stack) variables of external interests
  - name
  - current location
- *Unresolved references*:
  - Location
  - Address
- Debugging information
  - Examples: source line numbers for breakpoints, layout of structures, variable locations

# A Simple Example

main.c

```
extern float sin();
extern printf(), scanf();

main() {
  double x, result;
  printf("Type number: ");
  scanf("%f", &x);
  result = sin(x);
  printf("Sine is %f\n",
         result);
}
```

math.c

```
double sin(double x) {
  ...
}
```

stdio.c

```
FILE* stdin, stdout;

int printf(const char* format,...) {
  ...
  fputc(c, stdout);
  ...
}

int scanf(const char* format,...) {
  ...
  c = fgetc(stdin);
  ...
}
```

# main.o Object File

main.c

```
extern float sin();
extern printf(), scanf();

main() {
  double x, result;
  printf("Type number: ");
  scanf("%f", &x);
  result = sin(x);
  printf("Sine is %f\n",
         result);
}
```

main.o

```
 0  main:                          text section
    ...
30  call printf
    ...
52  call scanf
    ...
60  call sin
    ...
86  call printf

 0  _s1: "Type number: "           data section
14  _s2: "%f"
17  _s3: "Sine is %f\n"

    main    T[0]                   symbols
    _s1     D[0]
    _s2     D[14]
    _s3     D[17]

    printf  T[30]                  unresolved
    printf  T[86]                  references
    scanf   T[52]
    sin     T[60]
    _s1     T[24]
    _s2     T[54]
    _s3     T[80]
```

*"Store the final location of sin at offset 60 in the text section"*

# stdio.o Object File

stdio.c

```
FILE* stdin, stdout;

int printf(const char* format,
    ...) {
  ...
  fputc(c, stdout);
  ...
}

int scanf(const char* format,
    ...) {
  ...
  c = fgetc(stdin);
  ...
}
```

stdio.o

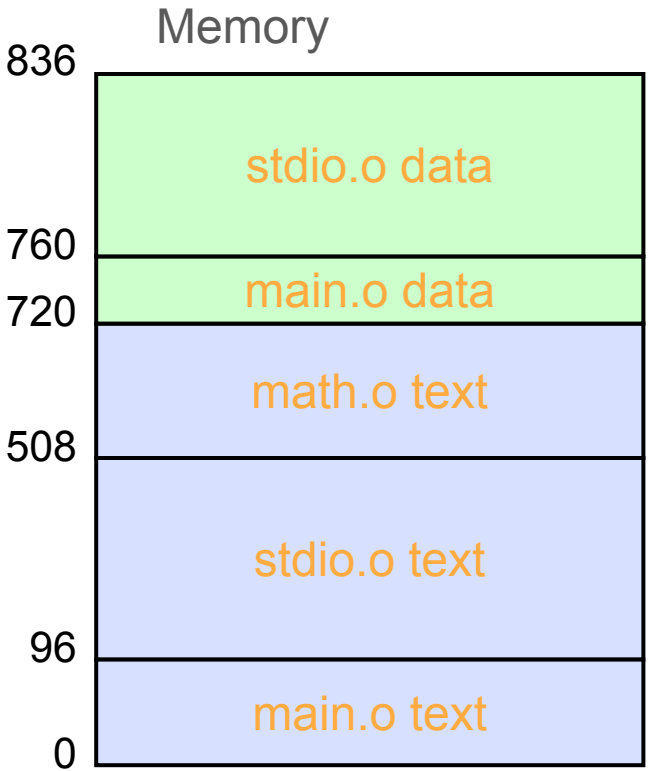
| | | |
|---|---|---|
| | ... | text section |
| 44 | printf: | |
| | ... | |
| 118 | load stdout | |
| | ... | |
| 232 | scanf: | |
| | ... | |
| 306 | load stdin | |
| | ... | |
| 0 | stdin: | data section |
| 8 | stdout: | |
| | printf T[44] | symbols |
| | scanf T[232] | |
| | stdin D[0] | |
| | stdout D[8] | |
| | stdout T[118] | unresolved references |
| | fputc T[122] | |
| | stdin T[306] | |
| | fgetc T[310] | |

# Linker execution

Three scans over the object files:

- Pass 1:
  - Read in section sizes
  - Compute memory layout
- Pass 2:
  - Read in all symbols
  - Create complete symbol table in linker's memory.
- Pass 3:
  - Read in sections and unresolved references
  - Update addresses
  - Write out executable file.

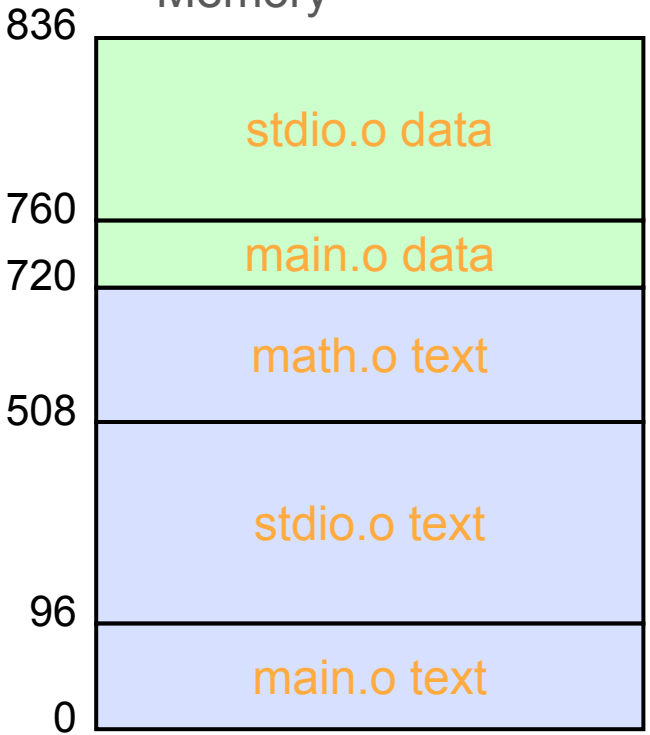
# After Pass 1




- Pass 1:
  - Read in section sizes
  - Compute memory layout

# After Pass 2

Memory

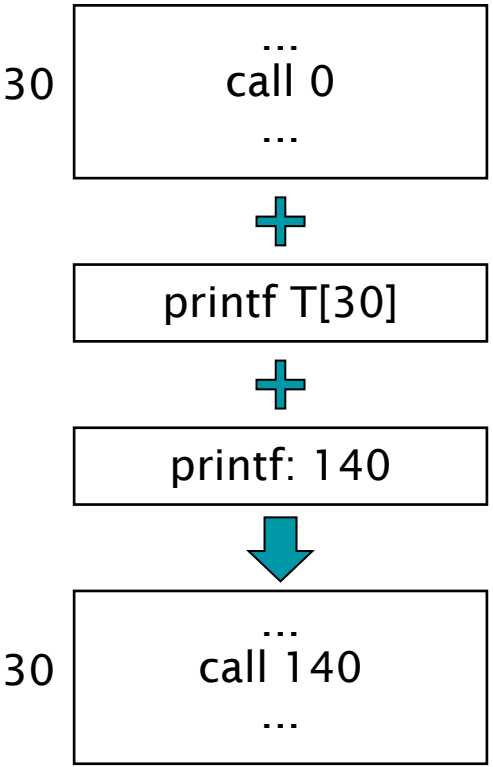



Symbol table:

| Name | File | Sec | Offset | Addr |
|---|---|---|---|---|
| main | main.o | T | 0 | 0 |
| _s1 | main.o | D | 0 | 720 |
| _s2 | main.o | D | 14 | 734 |
| _s3 | main.o | D | 17 | 737 |
| printf | stdio.o | T | 44 | 140 |
| scanf | stdio.o | T | 232 | 328 |
| stdin | stdio.o | D | 0 | 760 |
| stdout | stdio.o | D | 8 | 768 |
| sin | math.o | T | 0 | 508 |

- Pass 2:
  - Read in all symbols
  - Create complete symbol table in linker's memory.

# Pass 3: Resolving References




text section in main.o

unresolved reference in main.o

symbol table

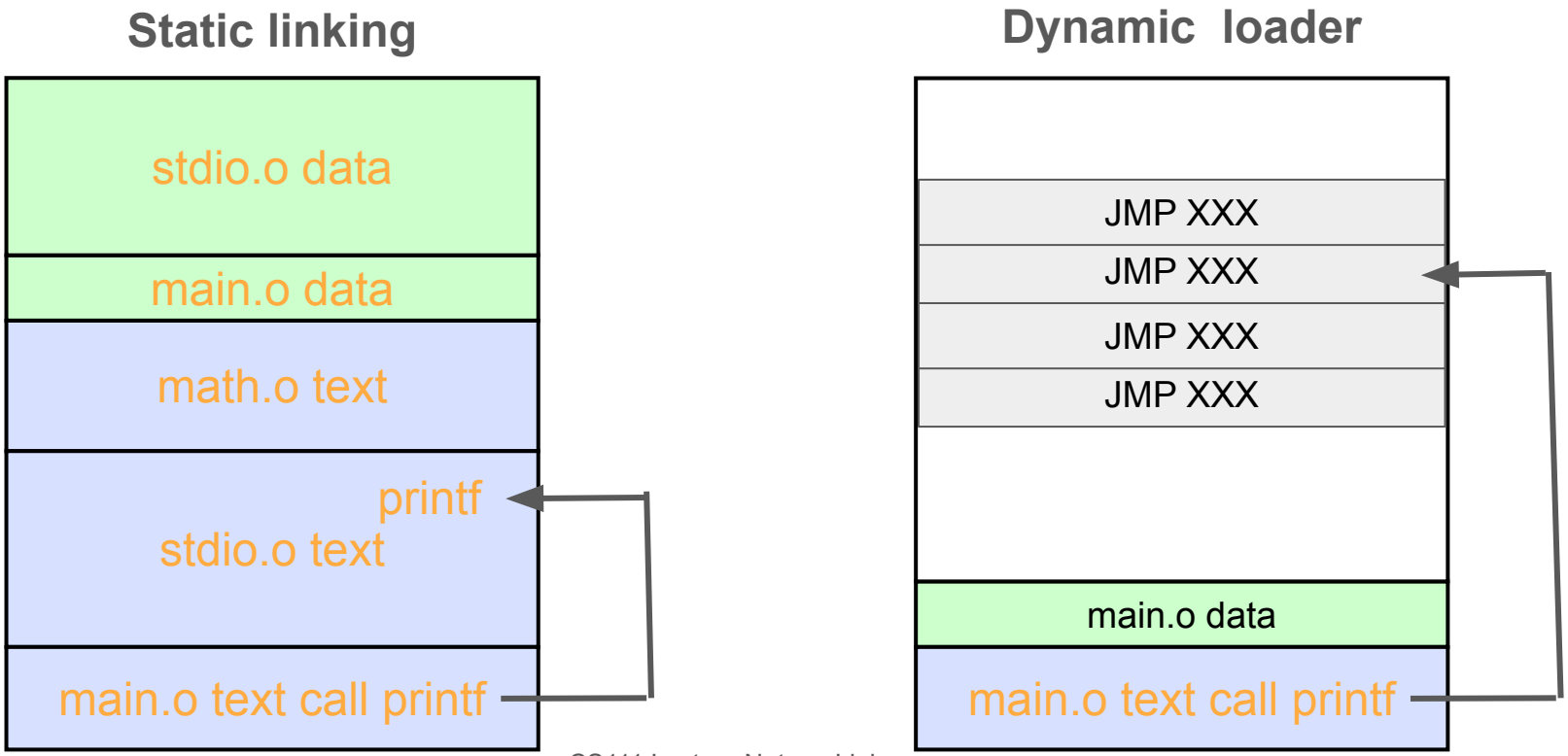
text section in a.out

- Pass 3:
  - Read in sections and unresolved references
  - Update addresses
  - Write out executable file.

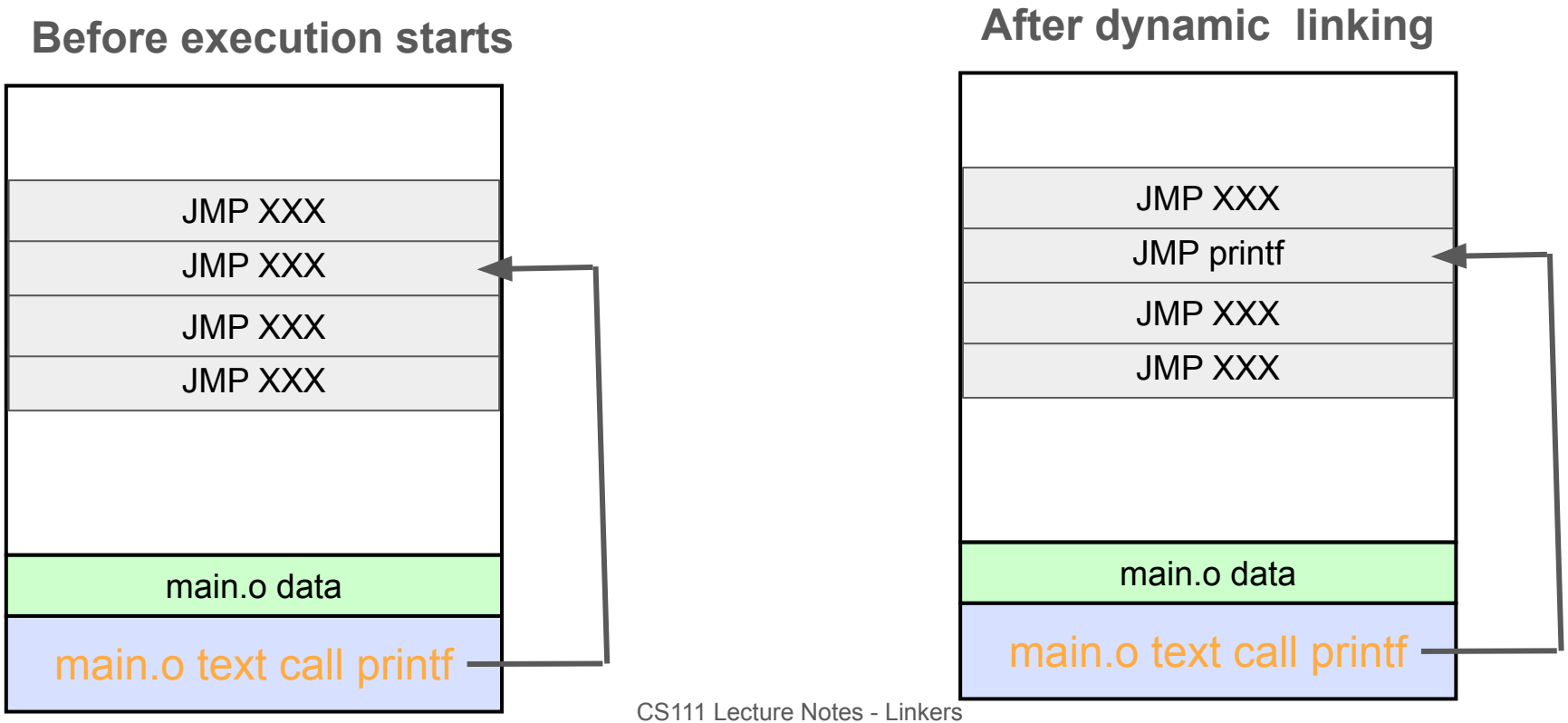
# Dynamic linking

- Lecture so far has described what is called **static linking**
  - Each program complete
  - All references ***resolved***
- Much wasted memory as language libraries grew
- Systems today support ***shared libraries*** and ***dynamic linking***
  - Single copy of libraries in memory, shared by all processes.
  - Library location isn't known until program is loaded into memory
    - Resolve references dynamically, during program execution

# Dynamic linking: One approach: Jump Tables



# Dynamic linking: Jump Table program start up



# Jump Table and Dynamic loader

- Jump Table Entry:
  - Name of a function (e.g. printf)
  - Name of shared library file containing function
  - Jump instruction to the beginning of that function (not known during linking)
- Dynamic load code invoked at program startup: scan jump table:
  - Map in the shared libraries mentioned in the jump table
  - Fill in the table jump instructions.
- Jump table stored in data section
- Call dynamic loader before program `main`