Code optimization & linking

Jinyang Li

Slides adapted from Bryant and O'Hallaron

What we've learnt so far

- C program → x86 instructions
  - Memory layout
  - control flows: sequential, jumps, call/ret
- Buffer overflow
  - Hijack control flow by overwriting a return address
  - Execute code intended by the attacker

Today's lesson plan

- Code optimization (done by the compiler)
  - common optimization techniques
  - what prevents optimization
- C linker

Optimizing Compilers

- Goal: generate efficient, correct machine code
  - allocate registers, choose instructions, ...

gcc's optimization levels: -O1, -O2, -O3

Generated code must have the same behavior as the original C program under all scenarios

Common optimization: code motion

- Move computation outside loop if possible.

```
void set_arr(long *arr, long n)
{
  for (long i = 0; i < n; i++)
    arr[i] = n*n;
}
```

done inside loop

```
  testq  %rsi, %rsi                # Test n
  jle  .L1                         # If 0, goto done
  movq %rsi, %rdx
  leaq (%rdi, %rsi, 8), %rax      # rax = &arr[n]
  imulq %rsi, %rdx                # rdx = n*n
.L3:
  movq  %rdx, (%rdi)           # (*p) =rdx
  addq $8, %rdi                # p++;
  cmpq  %rax, %rdi             # cmp &arr[n]) vs. p
  jne  .L3                     # if !=, goto Loop .L3
.L1:
  ret
```

```
void set_arr(long *arr, long n)
{
  long t = n*n;
  for (long i = 0; i < n; i++)
    arr[i] = t;
}
```

Equivalent C code

Common Optimization: use simpler instructions

- Replace costly operation with simpler one
  - Shift, add instead of multiply or divide

```
16*x   -->    x << 4
```

  - Recognize sequence of products

```
for (long i=0; i<n; i++ {
   arr[i] = n*i;
}
```

```
long ni = 0;
for (long i = 0; i < n; i++)
{
  arr[i] = ni;
  ni += n;
}
```

assembly not shown
this is the equivalent C code

Common Optimization: reuse common sub-expressions

```
x = a*b + c;
y = a*b*d;
```

```
tmp = a*b;
x = tmp + c;
y = tmp*d;
```

assembly not shown
this is the equivalent C code

3 multiplications:
a*b, a*b*c

2 multiplications:
a*b, tmp*d;

What prevents optimization?

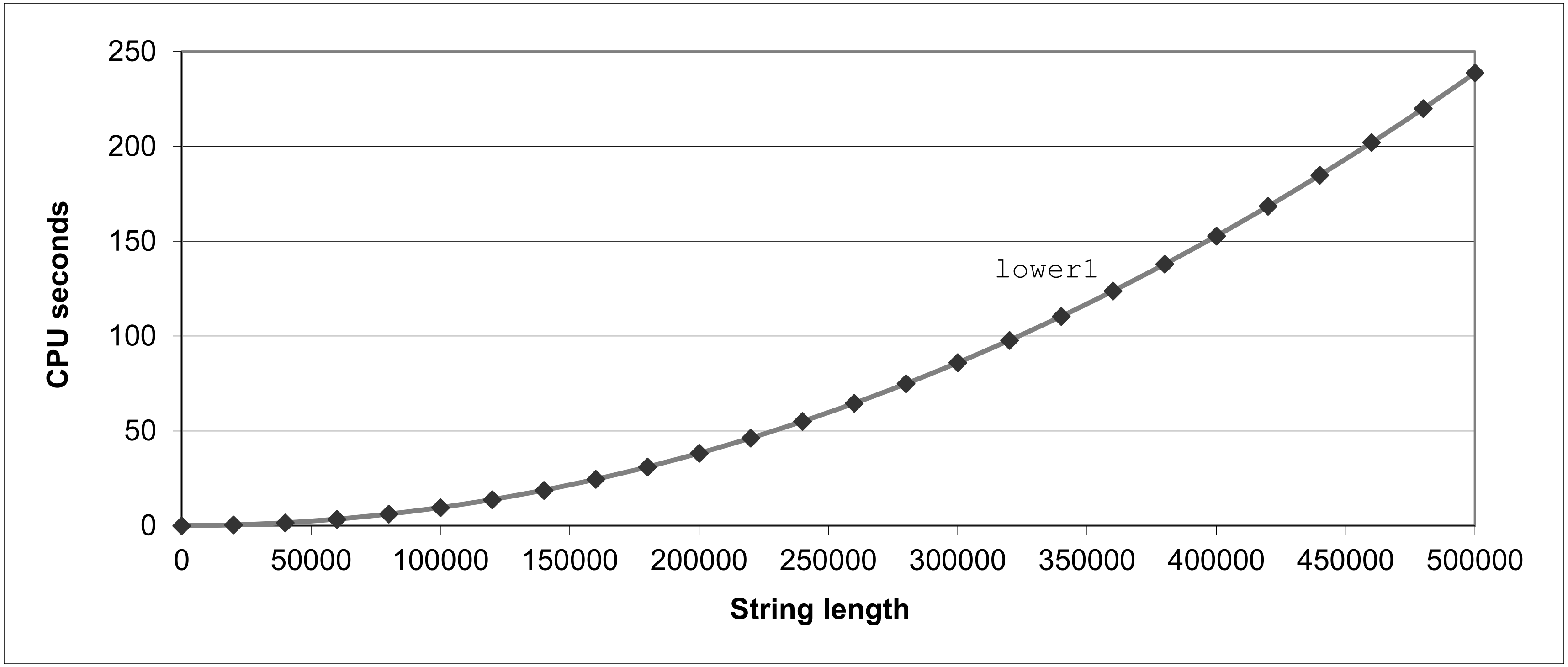
Optimization obstacle #1: Procedure Calls

```
// convert uppercase letters in string to lowercase
void lower(char *s) {
   for (size_t i=0; i<strlen(s); i++) {
      if (s[i] >= 'A' && s[i] <= 'Z') {
         s[i] -= ('A' - 'a');
      }
   }
}
```

Question: What’s the big-O runtime of lower, O(n)?

Lower Case Conversion Performance

– Quadratic performance!



Calling strlen in loop

```
// convert uppercase letters in string to lowercase
void lower(char *s) {

   for (size_t i=0; i<strlen(s); i++) {
      if (s[i] >= 'A' && s[i] <= 'Z') {
         s[i] -= ('A' - 'a');
      }
   }
}
```

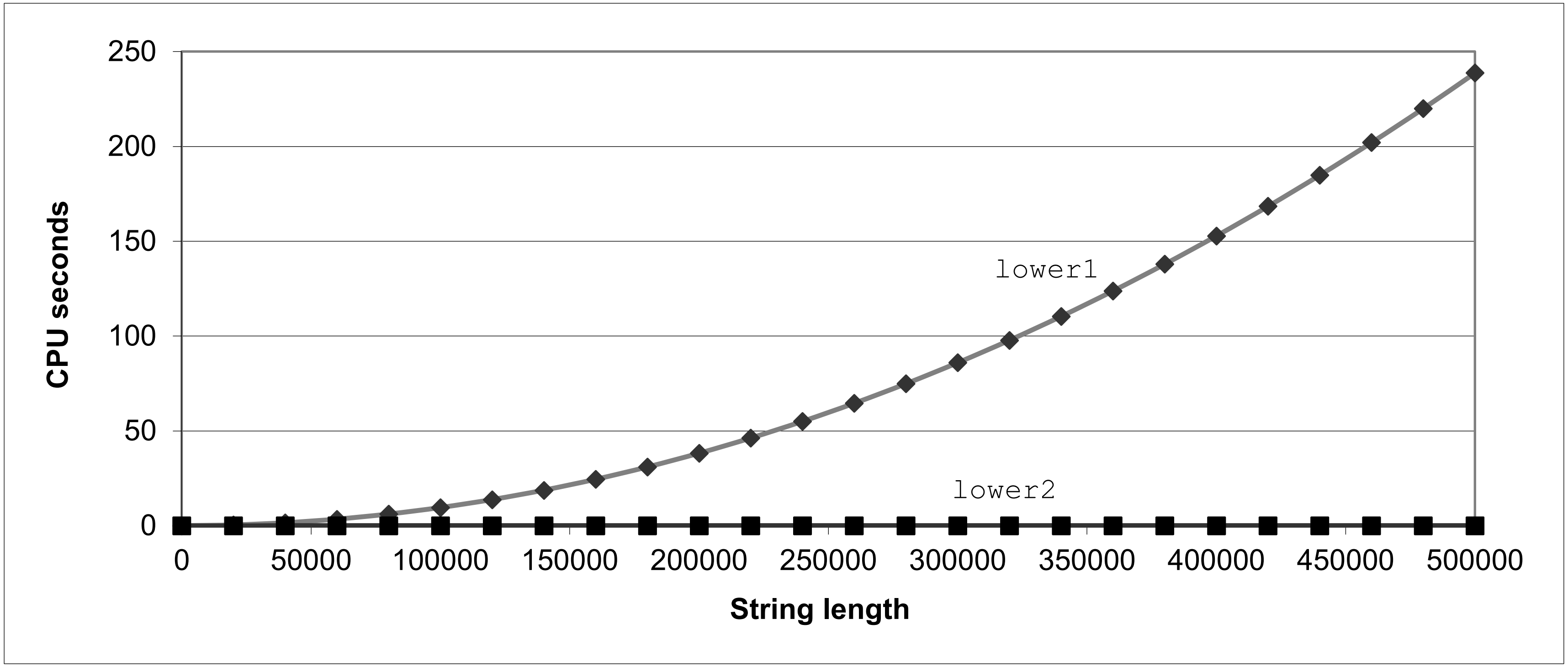
- Strlen takes O(n) to finish
- Strlen is called n times

Calling strlen in loop

```
// convert uppercase letters in string to lowercase
void lower(char *s) {
   size_t len = strlen(s);
   for (size_t i=0; i<len; i++) {
      if (s[i] >= 'A' && s[i] <= 'Z') {
         s[i] -= ('A' - 'a');
      }
   }
}
```

Lower Case Conversion Performance

- Now performance is linear w/ length, as expected



Optimization obstacle: Procedure Calls

- Why can't compiler move `strlen` out of inner loop?
  - Procedure may have side effects
    - May alter global state
  - Procedure may not return same value given same arguments
    - May depend on global state
- Compiler optimization is conservative:
  - Typically treat procedure call as a black box
  - Weak optimizations near them
- Remedy:
  - Do your own code motion

Optimization obstacle 2: Memory aliasing

```
//sum all elements of the array “a”
void sum(long *a, long n, long *result) {
    *result = 0;
    for (long i = 0; i < n; i++) {
        (*result) += a[i];
    }
}
```

```
        movq    $0, (%rdx)
        movl    $0, %eax
        jmp     .L2
.L3:
        movq    (%rdi,%rax,8), %rcx
        addq    %rcx, (%rdx)
        addq    $1, %rax
.L2:
        cmpq    %rsi, %rax
        jl      .L3
        ret
```

- Code updates *result on every iteration
- Why not keep sum in a register and write once at the end?

Memory aliasing: different pointers may point to the same location

```
void sum(long *a, long n, long *result) {
    *result = 0;
    for (long i = 0; i < n; i++) {
        (*result) += a[i];
    }
}
```

*result may alias to some location in array a
→ updates to *result may change a

```
int main() {
   long a[3] = {1, 1, 1};
   long *result;
   long r;

   result = &r;
   sum(a, 3, result);

   result = &a[2];
   sum(a, 3, result);
}
```

Value of `a`:

`before loop:` {1, 1, 0}

`after i = 0:` {1, 1, 1}

`after i = 1:` {1, 1, 2}

`after i = 2:` {1, 1, 4}

Optimization obstacle: memory aliasing

- Compiler cannot optimize due to potential aliasing
- Manual “optimization”

```
void sum(long *a, long n, long *result) {
   long sum = 0;
   for (long i = 0; i < n; i++) {
      sum += a[i];
   }
   *result = sum;
}
```

Getting High Performance

- Use compiler optimization flags
- Watch out for:
  - hidden algorithmic inefficiencies
  - Optimization obstacles:
    procedure calls & memory aliasing
- Profile the program’s performance

Today’s lesson plan

- Common code optimization (done by the compiler)
  - common optimization
  - what prevents optimization
- C linker

Example C Program

```
#include “sum.h”
int array[2] = {1, 2};

int main()
{
    int val = sum(array, 2);
    return val;
}
```

`main.c`

```
int sum(int *a, int n);
```

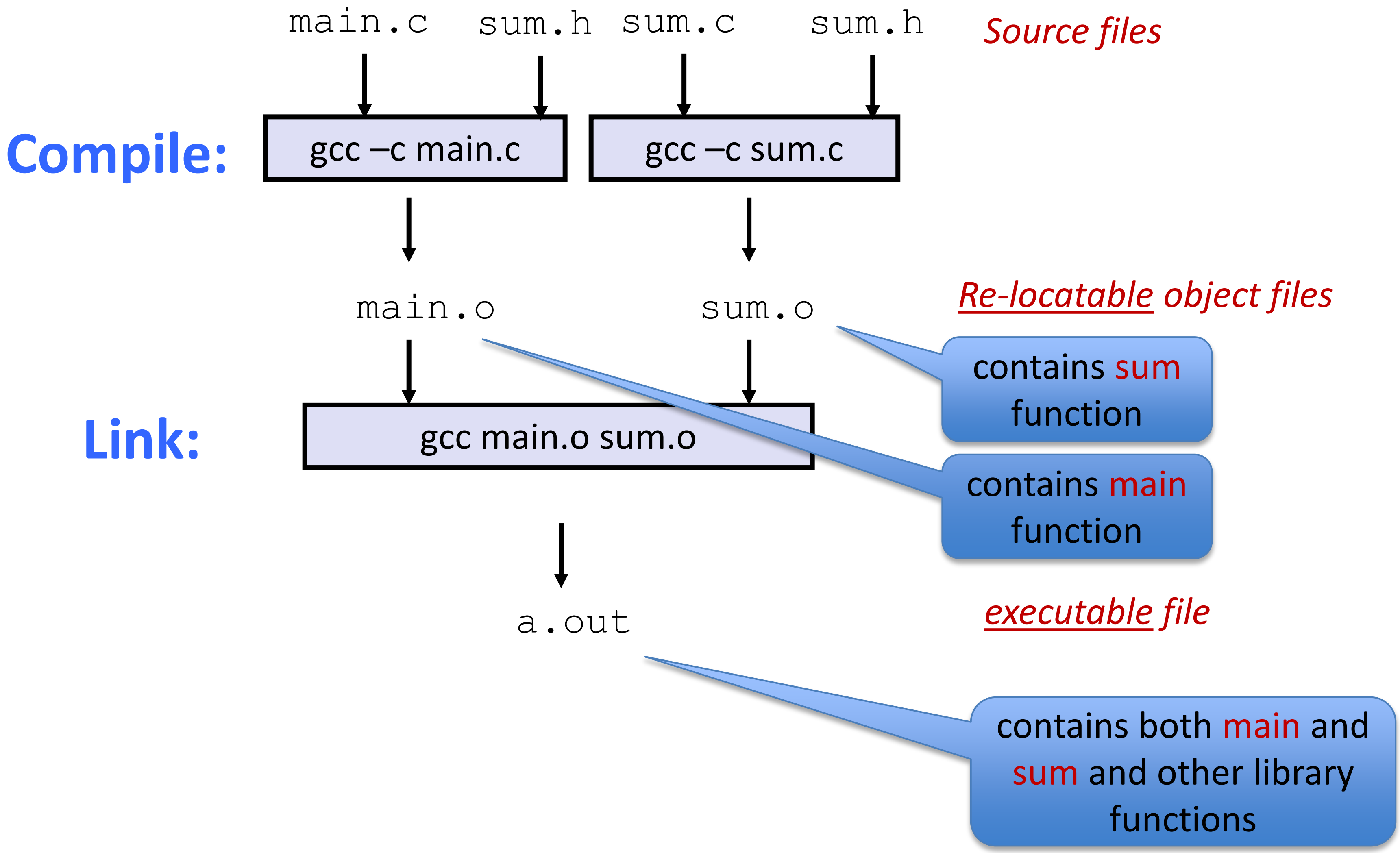
`sum.h`

```
#include “sum.h”

int sum(int *a, int n)
{
  int s = 0;
  for (int i = 0; i < n; i++) {
        s += a[i];
    }
    return s;
}
```

`sum.c`

Linking



`main.c` `sum.h` `sum.c` `sum.h`

Source files

Compile:

gcc –c main.c

gcc –c sum.c

`main.o` `sum.o`

<u>Re-locatable</u> object files

contains sum function

contains main function

Link:

gcc main.o sum.o

`a.out`

<u>executable</u> file

contains both main and sum and other library functions

Why a separate link phase?

- Modular code & efficient compilation
  - Better to structure a program as smaller source files
  - Change of a source file requires only re-compile that file, and then relink.
- Support libraries (no source needed)
  - Build libraries of common functions, other files link against libraries
    - e.g., Math library, standard C library

How does linker merge object files?

- Step 1: Symbol resolution
  - Programs define and reference *symbols* (global variables and functions):
    - `void swap() {…}    // define symbol swap`
    - `swap();            // reference symbol swap`
    - `int count;     // define global variable (symbol) count`
  - Symbol definitions are stored in object file in *symbol table*.
    - Each symbol table entry contains size, and location of symbol.
  - **Linker associates each symbol reference with its symbol definition (i.e. the address of that symbol)**

How does linker merge object files?

- Step 2: Relocation
  - With “gcc –c ...”, whenever compiler sees references to an unknown symbol, it uses a temporary placeholder
  - Linker re-locates symbols in the `.o` files to their final memory locations in the executable. Replace placeholders with actual addresses.

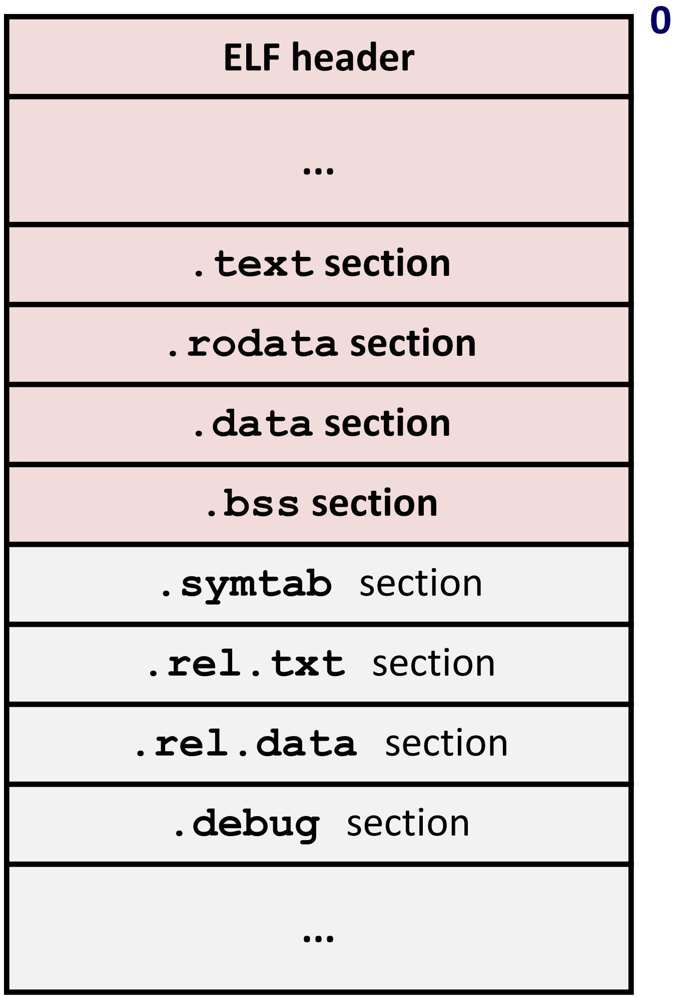
Let’s look at these two steps in more detail….

Format of the object files

- ELF is Linux's binary format for object files, including
  - Object files (`.o`),
  - Executable object files `(a.out)`
  - Shared object files, i.e. libraries (`.so`)

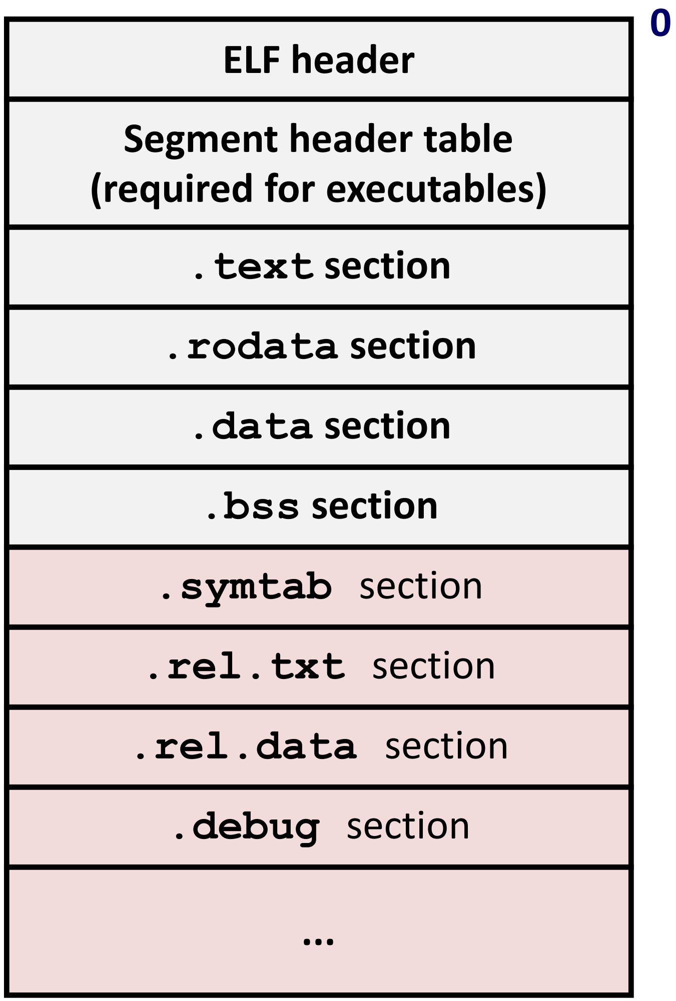
ELF Object File Format

- Elf header
  - file type (.o, exec, .so) ...
- `.text` section
  - Code
- `.rodata` section
  - Read only data
- `.data` section
  - Initialized global variables
- `.bss` section
  - Uninitialized global variables
  - “Better Save Space”
  - Has section header but occupies no space



ELF Object File Format (cont.)

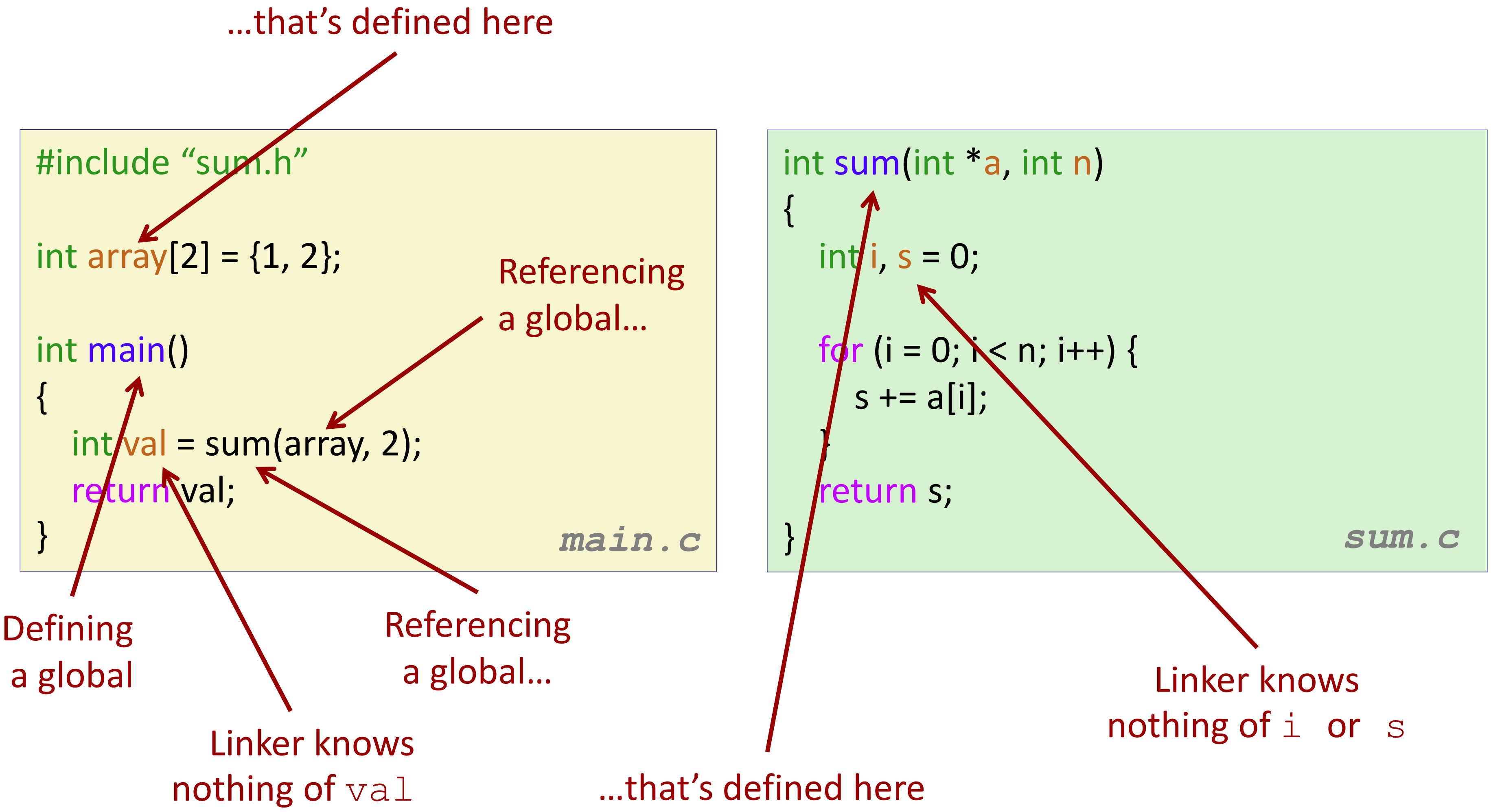
- `.symtab` section
  - Symbol table (symbol name, type, address)
- `.rel.text` section
  - Relocation info for **`.text`** section
  - Addresses of instructions that will need to be modified in the executable
- `.rel.data` section
  - Relocation info for **`.data`** section
  - Addresses of pointer data that will need to be modified in the merged executable
- `.debug` section
  - Info for symbolic debugging (**`gcc -g`**)



Linker Symbols

- Global symbols
  - Symbols that can be referenced by other object files
  - E.g. non-`static` functions & global variables.
- Local symbols
  - Symbols that can only be referenced by this object file.
  - E.g. static functions & global variables
- External symbols ← needs to be resolved
  - Symbols referenced by this object file but defined in other object files.

Step 1: Symbol Resolution



…that’s defined here

```
#include “sum.h”

int array[2] = {1, 2};

int main()
{
    int val = sum(array, 2);
    return val;
}
```

main.c

Referencing a global…

Defining a global

Linker knows nothing of `val`

Referencing a global…

```
int sum(int *a, int n)
{
    int i, s = 0;

    for (i = 0; i < n; i++) {
        s += a[i];
    }
    return s;
}
```

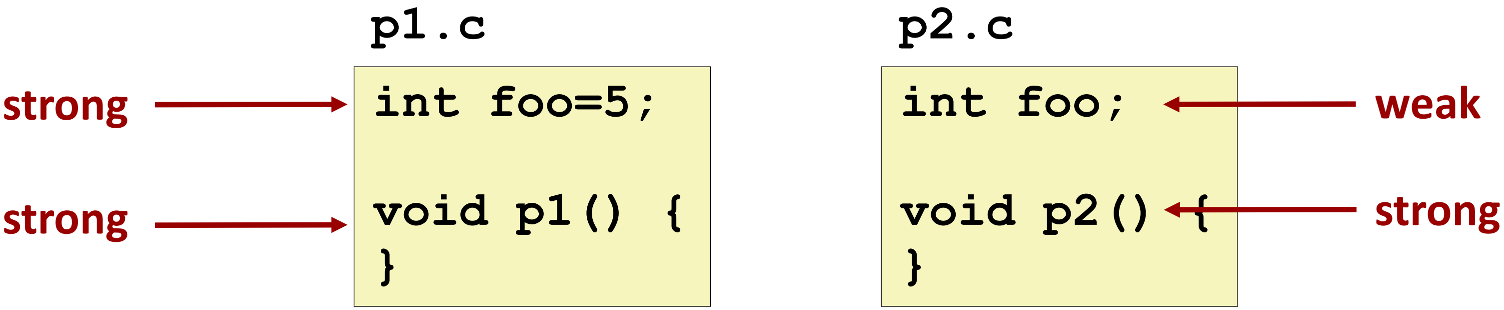
sum.c

…that’s defined here

Linker knows nothing of `i` or `s`

C linker quirks: it allows symbol name collision!

- Program symbols are either *strong* or *weak*
  - ***Strong***: procedures and initialized globals
  - ***Weak***: uninitialized globals



`p1.c`

```
int foo=5;

void p1() {
}
```

strong

strong

`p2.c`

```
int foo;

void p2() {
}
```

weak

strong

Symbol resolution in the face of name collision

- Rule 1: Multiple strong symbols are not allowed
  - Otherwise: Linker error

- Rule 2: If there's a strong symbol and multiple weak symbols, they all resolve to the strong symbol.

- Rule 3: If there are multiple weak symbols, pick an arbitrary one
  - Can override this with `gcc -fno-common`

Linker Puzzles

```
int x;
p1() {}
```

```

p1() {}
```

Link time error: two strong symbols (`p1`)

```
int x;
p1() {}
```

```
int x;
p2() {}
```

References to `x` will refer to the same uninitialized int. Is this what you really want?

```
int x=7;
int y=5;
p1() {}
```

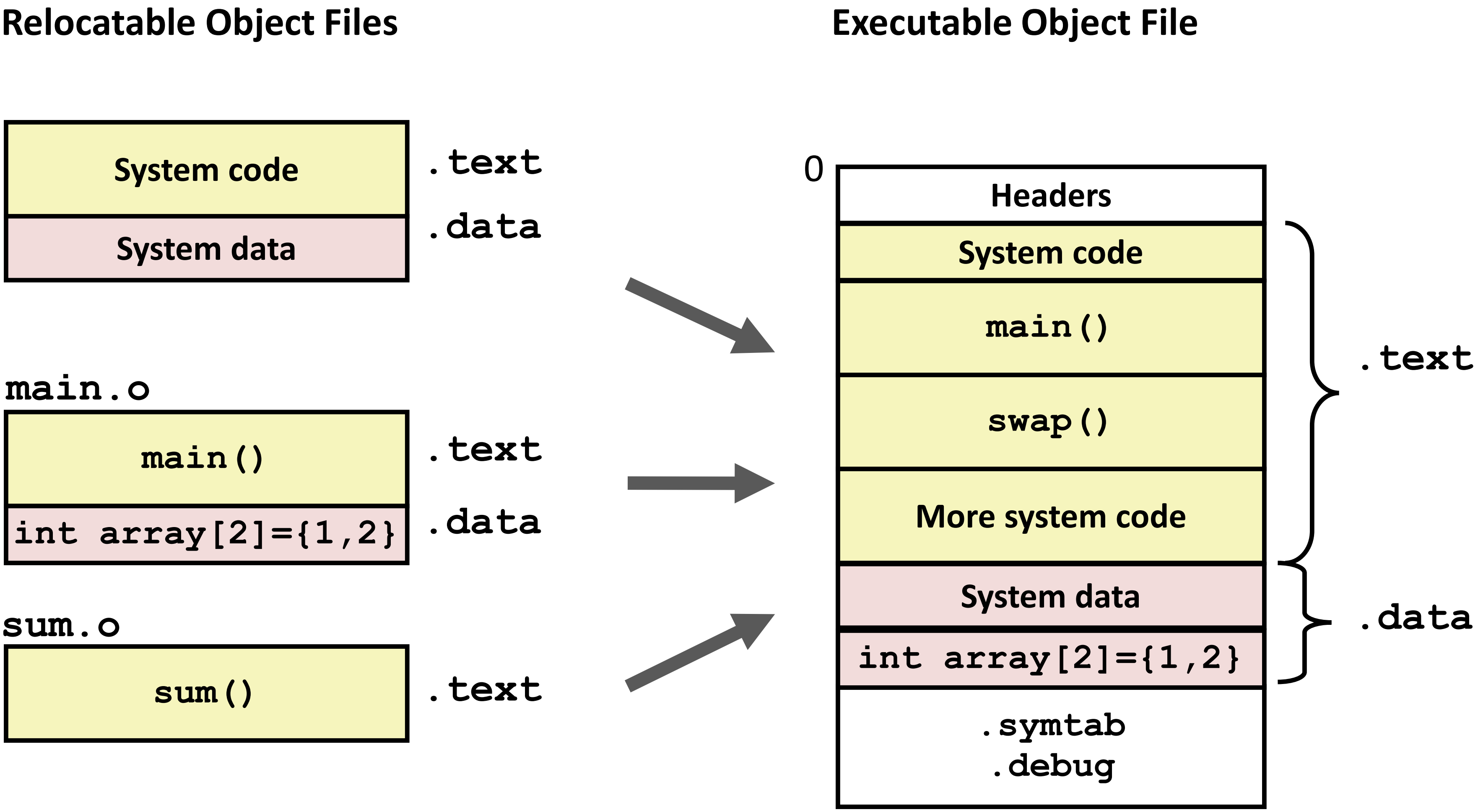
```
double x;
p2() {}
```

Writes to `x` in `p2` will overwrite `y`!
Nasty!

How to avoid symbol resolution confusion

- Avoid global variables if you can
- Otherwise
  - Use `static` if you can
  - Initialize if you define a global variable

Step 2: Relocation



Relocation Entries

```
int array[2] = {1, 2};

int main()
{
    int val = sum(array, 2);
    return val;
}
```

`main.c`

```
0000000000000000 <main>:
  0:   48 83 ec 08            sub    $0x8,%rsp
  4:   be 02 00 00 00         mov    $0x2,%esi
  9:   bf 00 00 00 00         mov    $0x0,%rdi      # %rdi = &array
                  a: R_X86_64_32 array          # Relocation entry

  e:   e8 00 00 00 00         callq  13 <main+0x13> # sum()
                  f: R_X86_64_PC32 sum-0x4      # Relocation entry
 13:   48 83 c4 08            add    $0x8,%rsp
 17:   c3                     retq
```

`main.o`

Source: `objdump -r -d main.o`

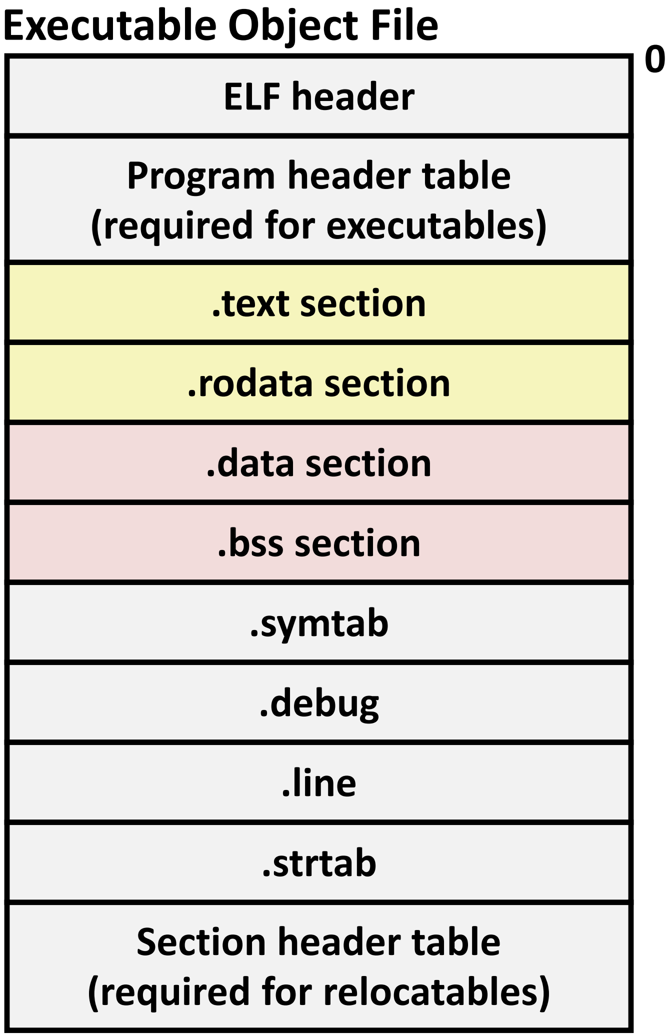
Relocated .text section

```
00000000004004d0 <main>:
  4004d0:   48 83 ec 08      sub    $0x8,%rsp
  4004d4:   be 02 00 00 00   mov    $0x2,%esi
  4004d9:   bf 18 10 60 00   mov    $0x601018,%rdi  # %rdi = &array
  4004de:   e8 05 00 00 00   callq  4004e8 <sum>   # sum()
  4004e3:   48 83 c4 08      add    $0x8,%rsp
  4004e7:   c3               retq

00000000004004e8 <sum>:
  4004e8:   b8 00 00 00 00        mov    $0x0,%eax
  4004ed:   ba 00 00 00 00        mov    $0x0,%edx
  4004f2:   eb 09                 jmp    4004fd <sum+0x15>
  4004f4:   48 63 ca              movslq %edx,%rcx
  4004f7:   03 04 8f              add    (%rdi,%rcx,4),%eax
  4004fa:   83 c2 01              add    $0x1,%edx
  4004fd:   39 f2                 cmp    %esi,%edx
  4004ff:   7c f3                 jl     4004f4 <sum+0xc>
  400501:   c3                    retq
```

`objdump -d a.out`

Loading Executable Object Files



Executable Object File

0

ELF header

Program header table
(required for executables)

.text section

.rodata section

.data section

.bss section

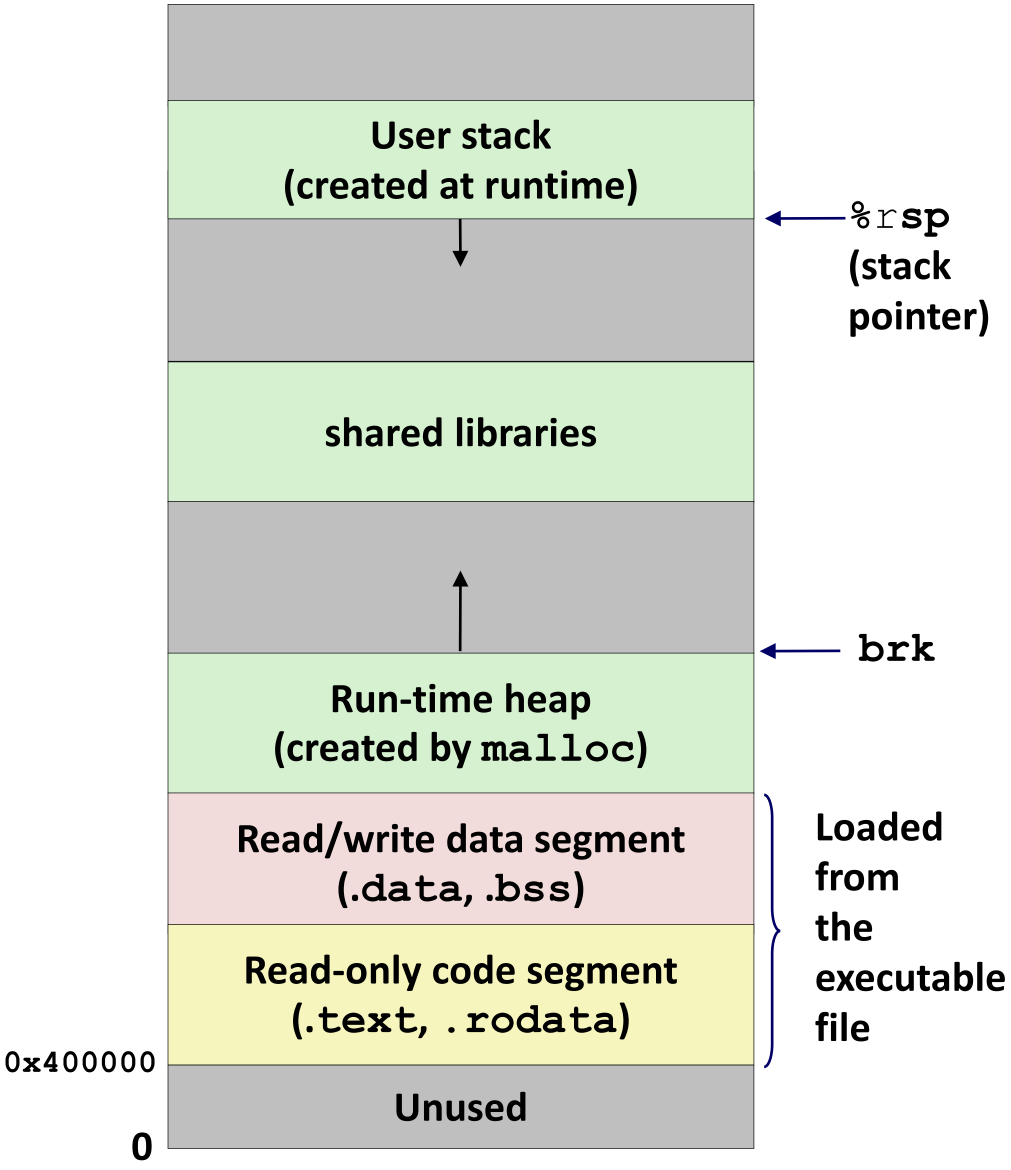
.symtab

.debug

.line

.strtab

Section header table
(required for relocatables)



User stack
(created at runtime)

`%rsp`
(stack
pointer)

shared libraries

`brk`

Run-time heap
(created by `malloc`)

Read/write data segment
(`.data`, `.bss`)

Read-only code segment
(`.text`, `.rodata`)

Loaded
from
the
executable
file

`0x400000`

Unused

0

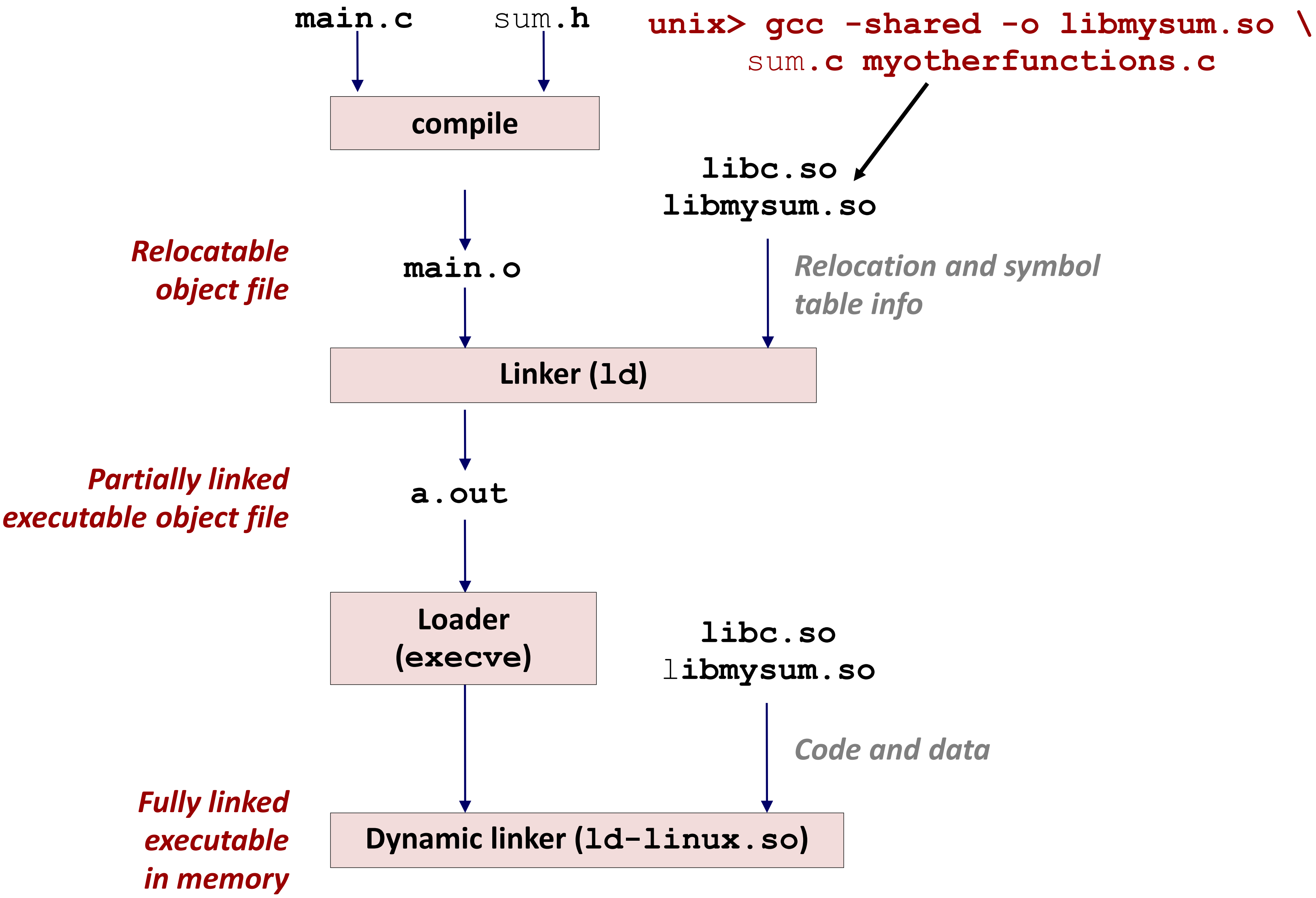
Summary

- Common compiler optimization
  - What it can do:
    - Code motion
    - Common sub-expression elimination
  - What it cannot do due to:
    - Function calls
    - Memory aliasing
- Linking
  - Symbol relocation
  - Be aware of silent symbol collision

Dynamic linking: Shared Libraries

- Dynamic linking can occur at program load-time
  - Handled automatically by the dynamic linker (**`ld-linux.so`**).
  - Standard C library (**`libc.so`**) usually dynamically linked.
- Dynamic linking can also occur at run-time.
  - In Linux, this is done by **`dlopen`**.

Dynamic Linking at Load-time



```
unix> gcc -shared -o libmysum.so \
        sum.c myotherfunctions.c
```

`main.c` `sum.h`

compile

`libc.so`
`libmysum.so`

Relocatable object file

`main.o`

Relocation and symbol table info

Linker (`ld`)

Partially linked executable object file

`a.out`

Loader (`execve`)

`libc.so`
`libmysum.so`

Code and data

Fully linked executable in memory

Dynamic linker (`ld-linux.so`)