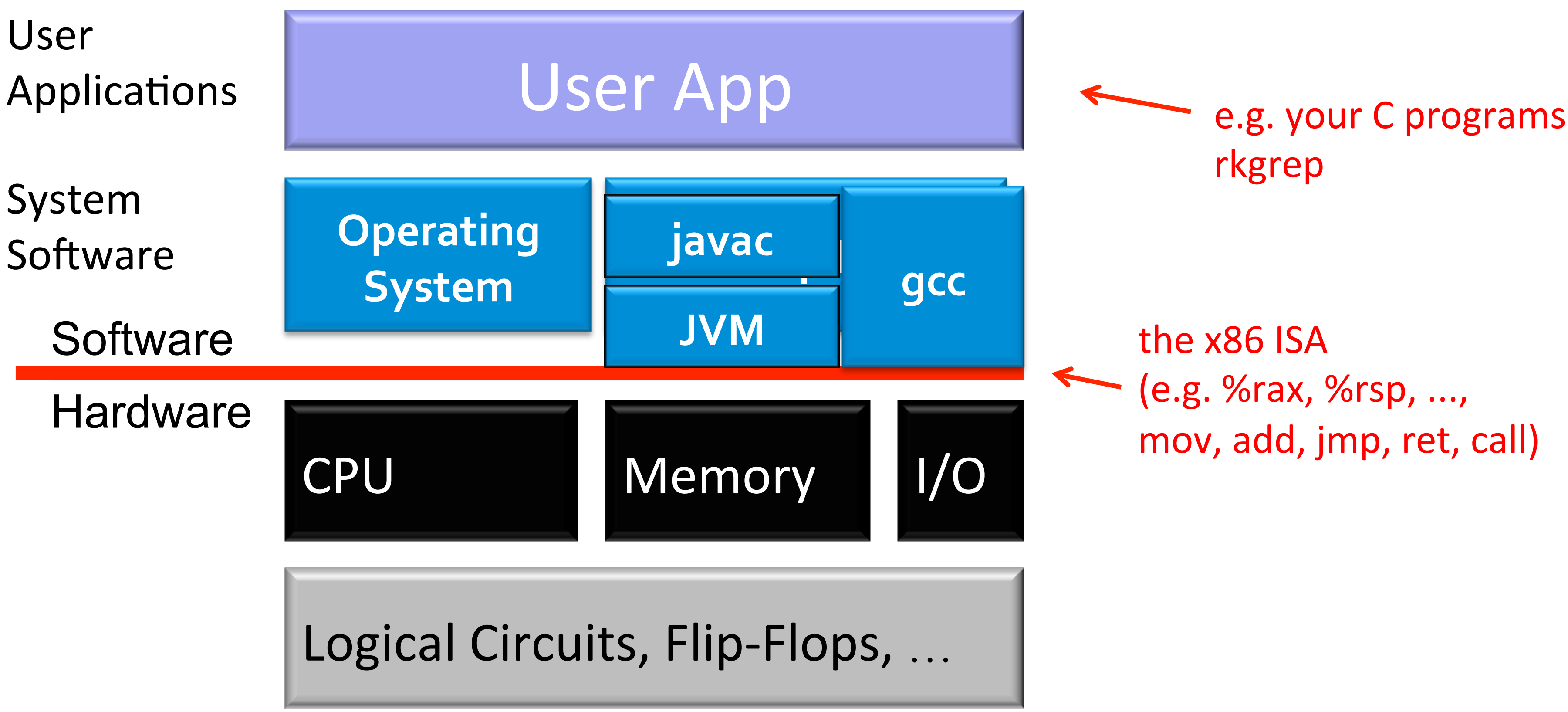
Code optimization & linking

Jinyang Li

Slides adapted from Bryant and O'Hallaron

What we've learnt so far



What we've learnt so far

- C program → x86 instructions
  - data storage
  - control flows: sequential, jumps, call/ret
- Buffer overflow
  - overwrite a code pointer (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, -Og

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

Common optimization: code motion

- Move computation outside of loop if result remains the same

```
void set_row(long *matrix,
 long i, long n)
{
  for (long j = 0; j < n; j++)
    matrix[n*i+j] = 0;
}
```

done inside the loop

done outside the loop

```
set_row:
  testq  %rcx, %rcx              # Test n
  jle  .L1                       # If 0, goto done
  imulq %rcx, %rdx               # ni = n*i
  leaq   (%rdi,%rdx,8), %rdx     # rowp = A + ni*8
  movq  $0, %rax                 # j = 0
.L3:
  movq  $0, (%rdx,%rax,8)        # M[rowp+8*j] = 0
  addq  $1, %rax                 # j++
  cmpq  %rcx, %rax               # j:n
  jne  .L3                       # if !=, goto loop .L3
.L1:
  ret
```

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

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

assembly not shown
this is equivalent C code

Common Optimization: reuse common subexpressions

```
// Sum neighbors of i,j
up =   val[(i-1)*n + j];
down =  val[(i+1)*n + j];
left =  val[i*n + j-1];
right = val[i*n + j+1];
sum = up + down + left + right;
```

```
long inj = i*n + j;
up =      val[inj - n];
down =  val[inj + n];
left =     val[inj - 1];
right =   val[inj + 1];
sum = up + down + left + right;
```

3 multiplications:
(i-1)*n, (i+1)*n, i*n

1 multiplication:
i*n

assembly not shown
this is equivalent C code

What prevents optimization?

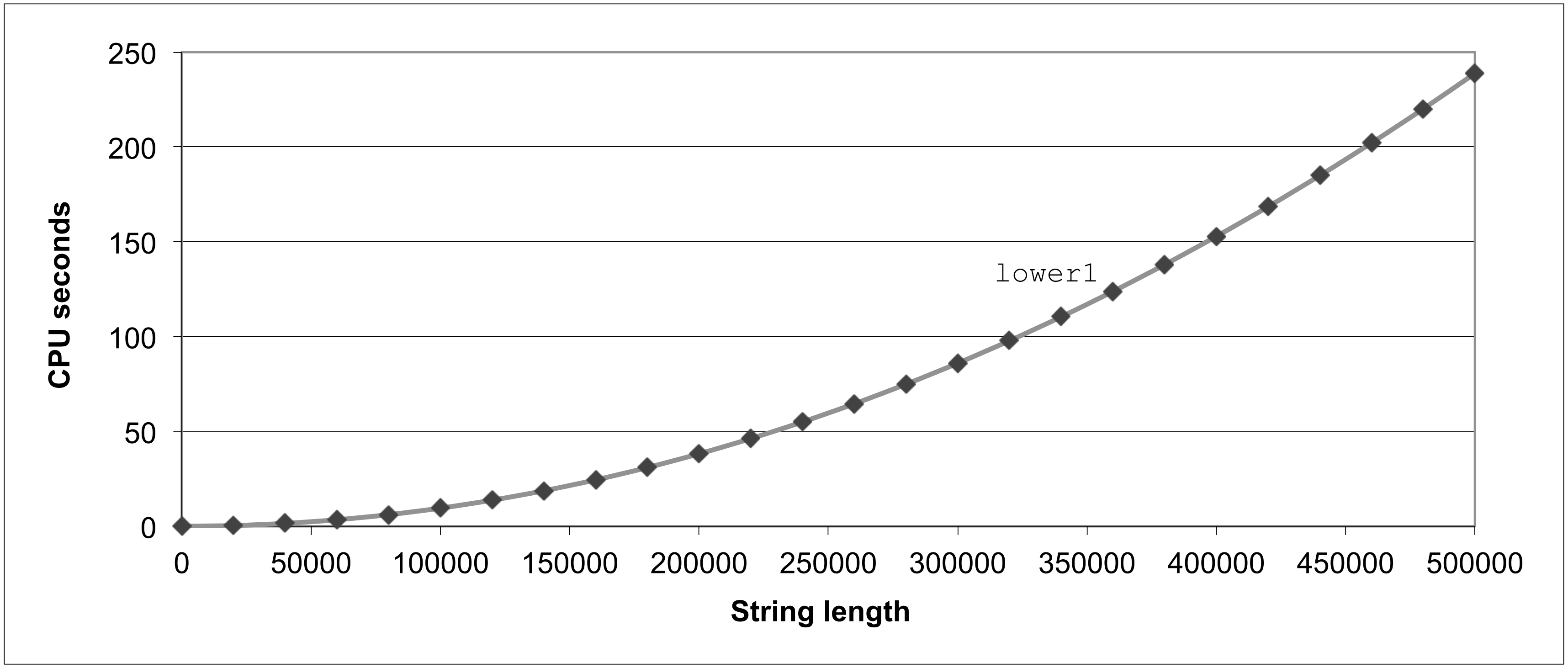
Optimization obstacle #1: Procedure Calls

```
// 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');
      }
   }
}
```

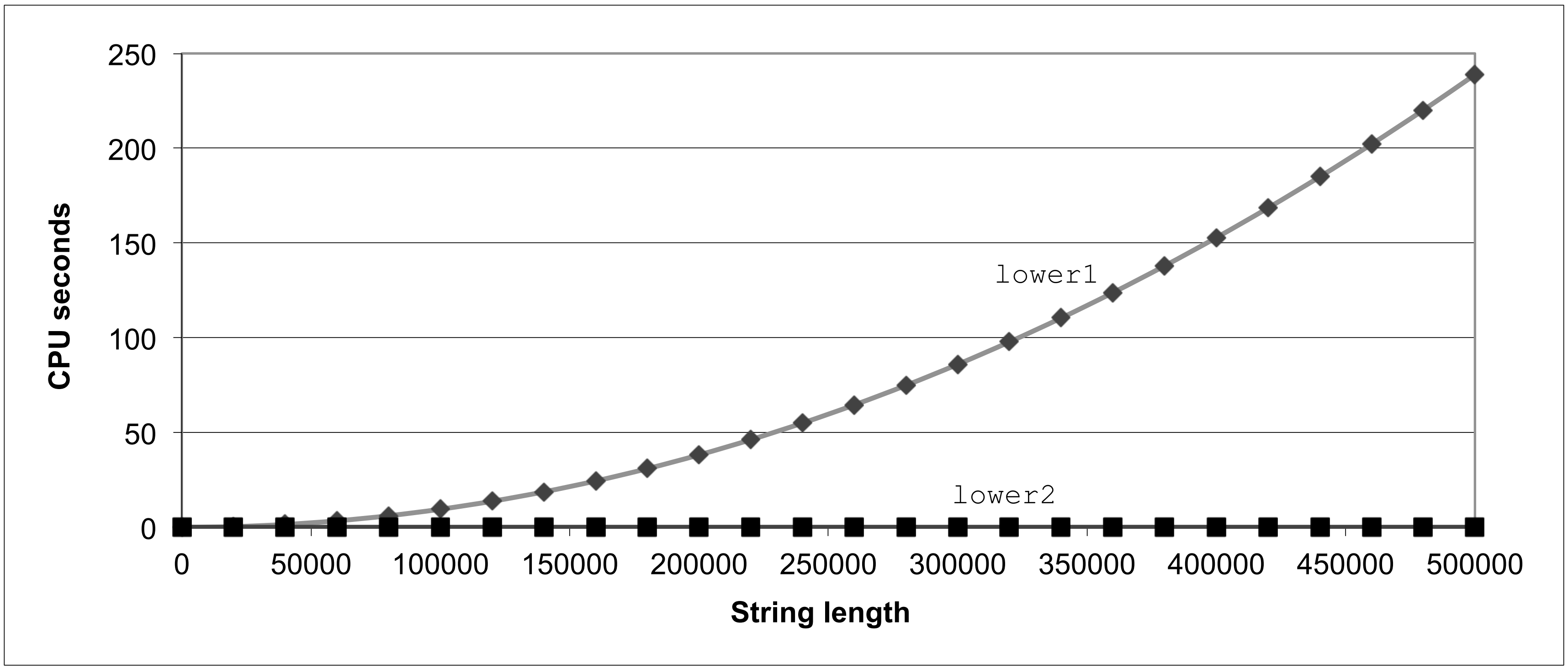
- Strlen takes O(n) to finish
- Strlen is called n times

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);
```

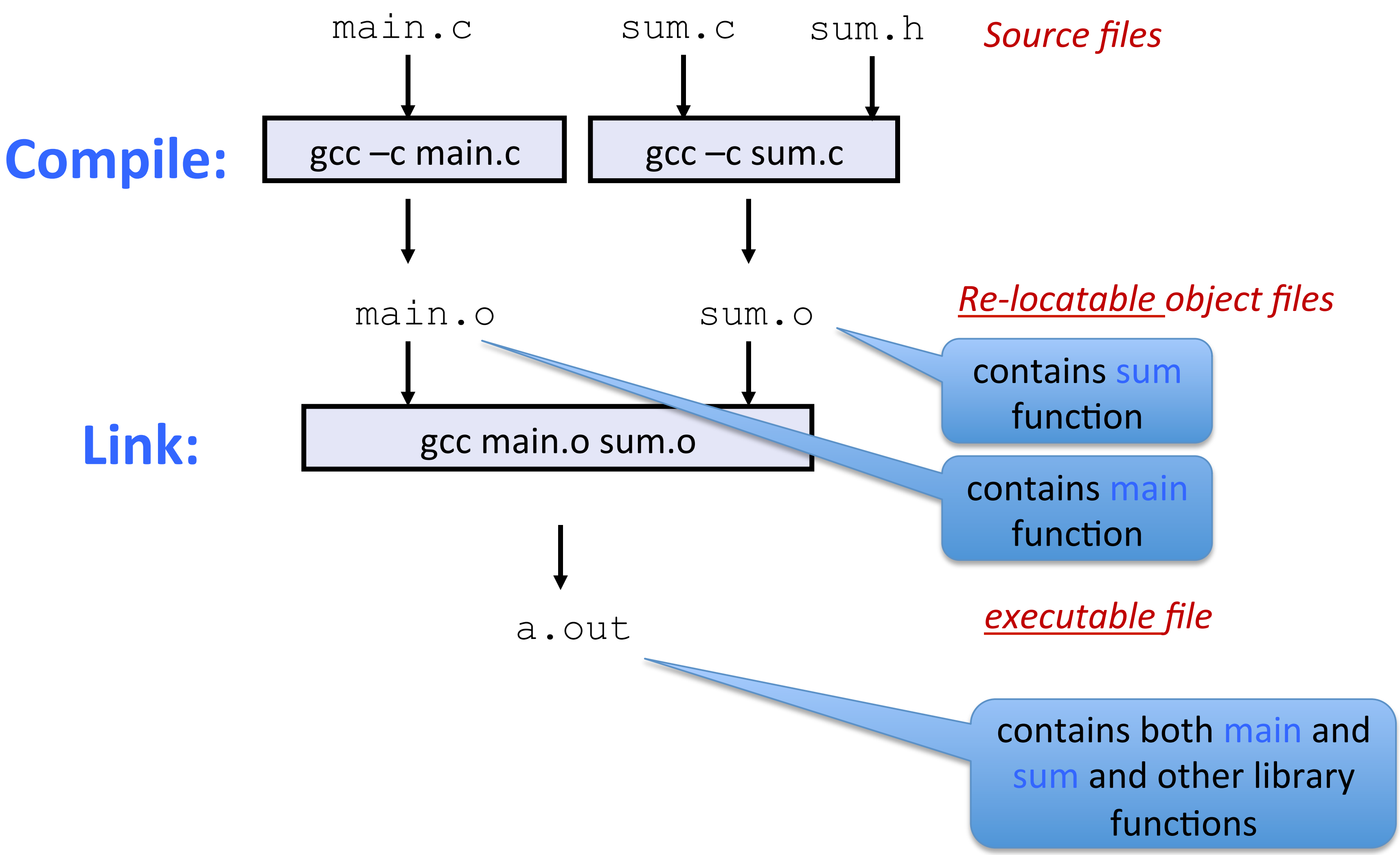
sum.h

```
#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.c
sum.h
Source files
Compile:
gcc –c main.c
gcc –c sum.c
main.o
sum.o
Re-locatable object files
contains sum function
contains main function
Link:
gcc main.o sum.o
a.out
executable 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 anc 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 ...”, compiler gives a defined symbol a temporary address. When referencing an unknown symbol, compiler uses a temporary placeholder
  - Re-locates symbols in the `.o` files to their final memory locations in the executable. Replace placeholders with actual addresses.

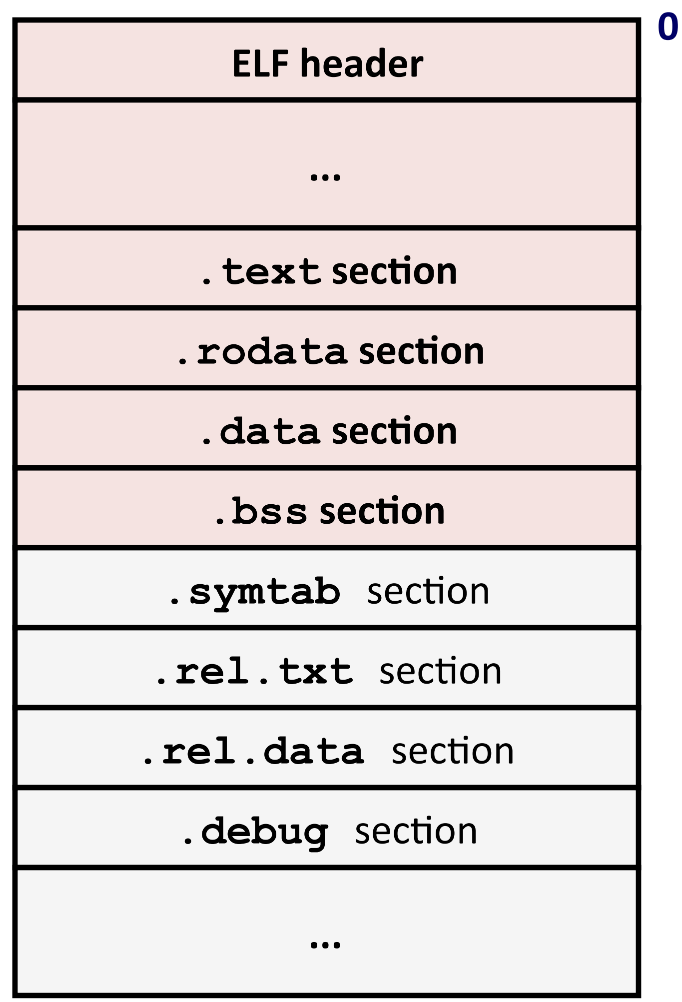
Let’s look at these two steps in more detail….

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`)

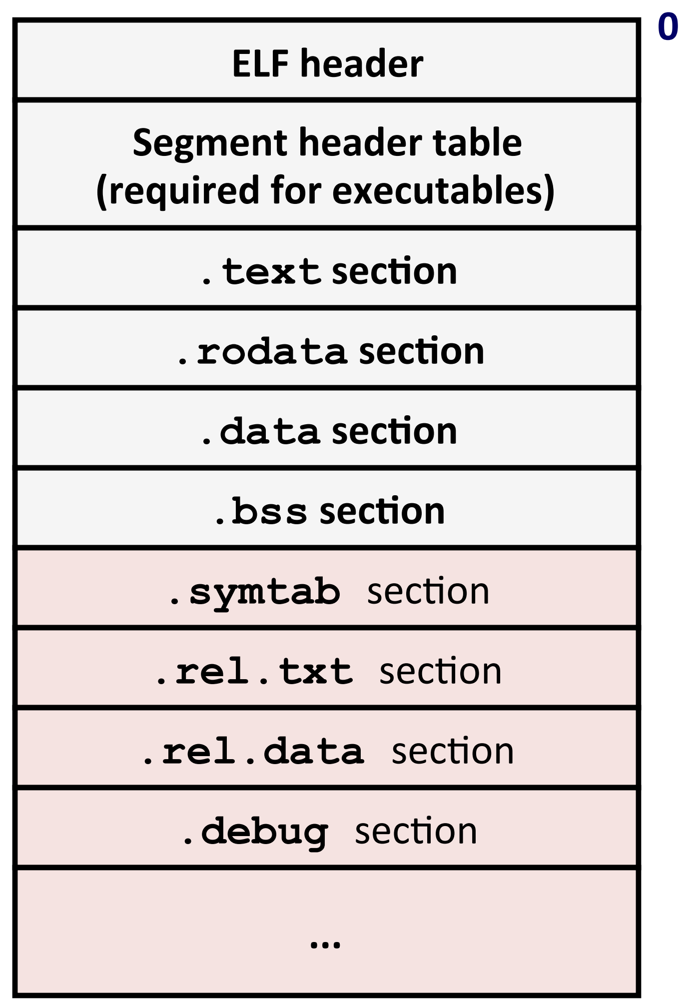
ELF Object File Format

- 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

Referencing a global…

…that's defined here

```c
#include "sum.h"

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

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

main.c

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

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

sum.c

Defining a global

Linker knows nothing of `val`

Referencing a global…

…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	p2.c	
strong	`int foo=5;`	`int foo;`	weak
strong	`p1() {` `}`	`p2() {` `}`	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() {}
```

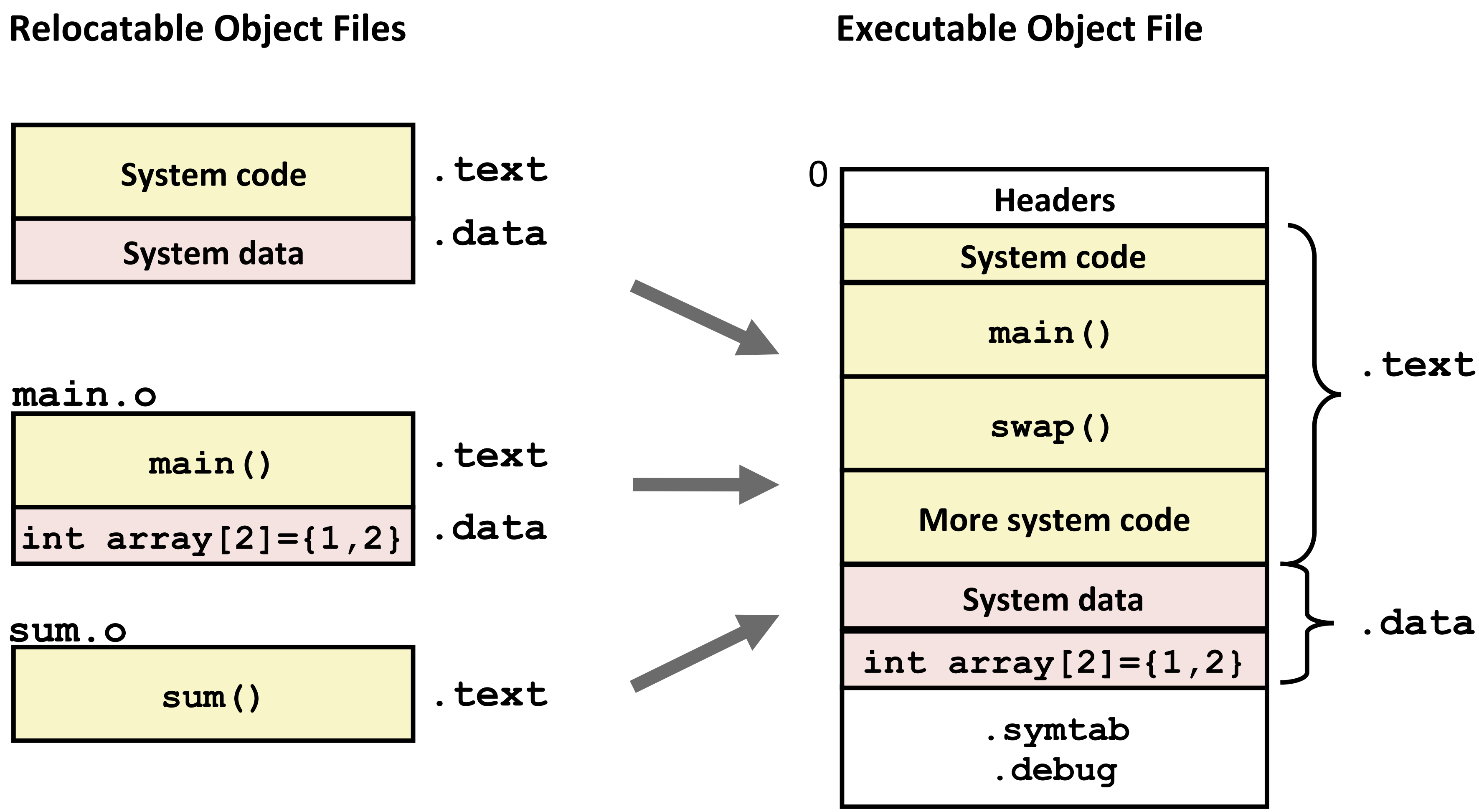
```
double x;
p2() {}
```

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
  - Use `extern` if you reference an external global variable

Step 2: Relocation



Relocation Entries

```c
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,%edi      # %edi = &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`

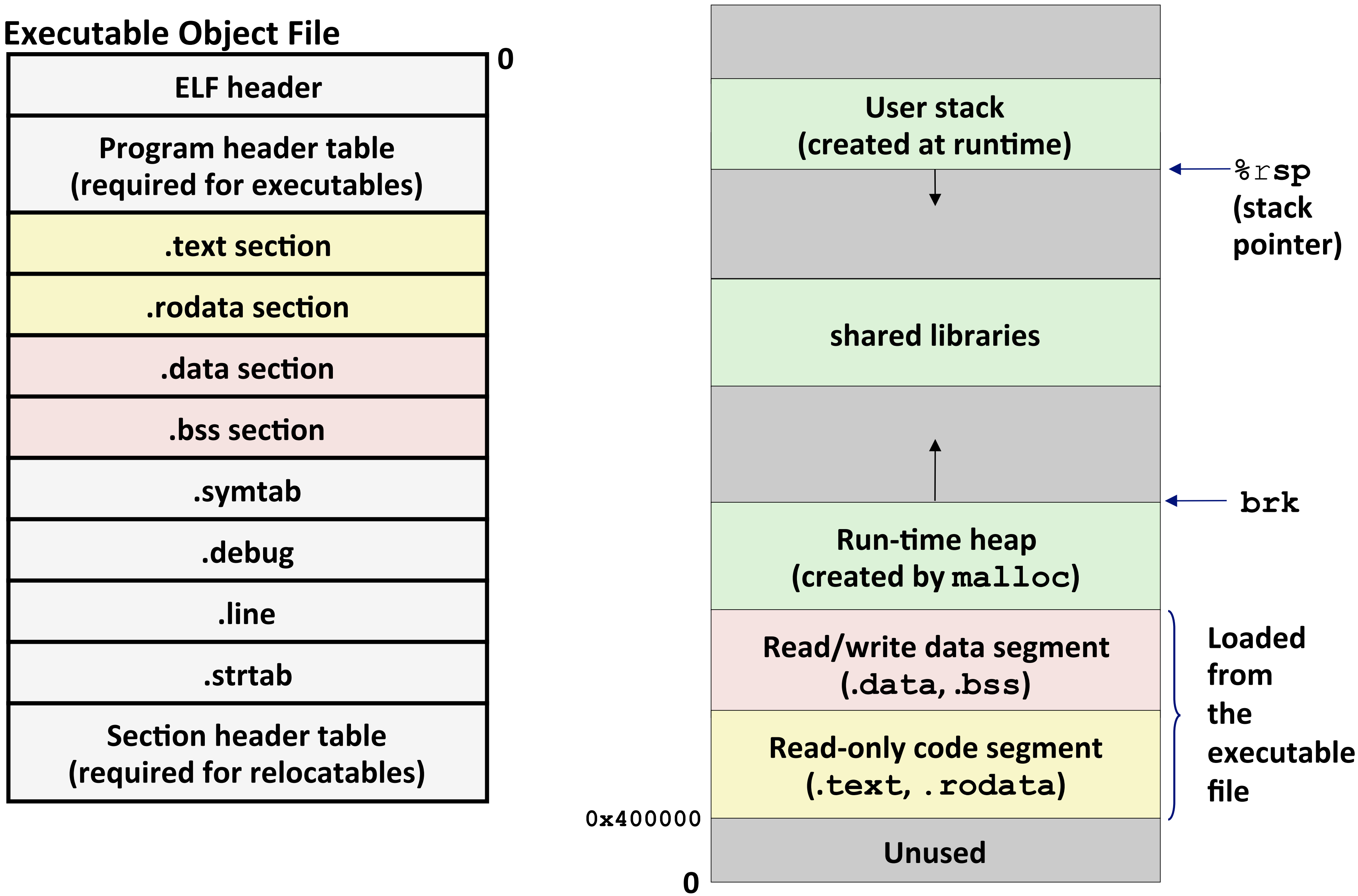
Relocated .text section

```
0000000004004d0 <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,%edi  # %edi = &array
  4004de:       e8 05 00 00 00      callq  4004e8 <sum>    # sum()
  4004e3:       48 83 c4 08         add    $0x8,%rsp
  4004e7:       c3                  retq

0000000004004e8 <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



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`**.
- Shared library routines can be shared by multiple running programs.
  - More on this when we learn about virtual memory

Dynamic Linking at Load-time