Systems Programming

Code Optimization and Linking

Textbook coverage:
Ch 5: Optimizing Program Performance
Ch 7. Linking

Byoungyoung Lee
Seoul National University
byoungyoung@snu.ac.kr
https://lifeasageek.github.io

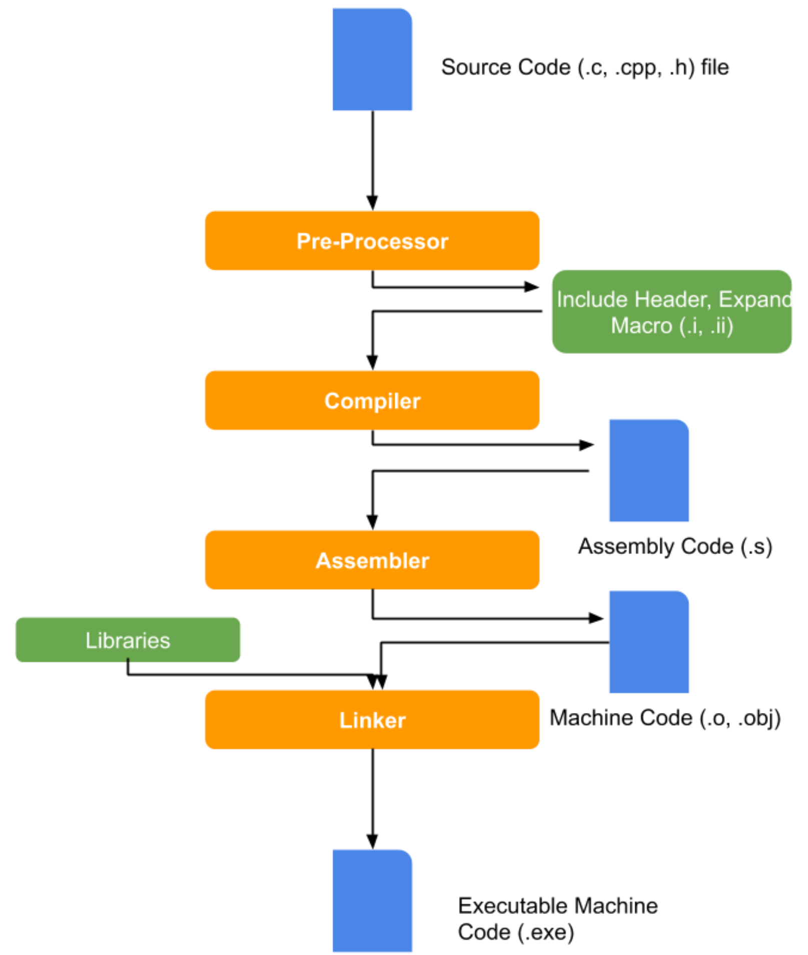
Lecture slides are prepared based on materials provided by CSAPP authors.

Today

- **Basics of compiler optimization**
  - Principles and goals
  - Some example optimizations
  - Obstacles to optimization
- **Linking: combining object files into programs**
  - Symbols and symbol resolution
  - Relocation
  - Static libraries
  - Dynamic libraries

What does it mean to compile code?

- **The CPU only understands *machine code* directly**
- **All other languages must be either**
  - ***interpreted:*** executed by software
  - ***compiled:*** translated to machine code by software



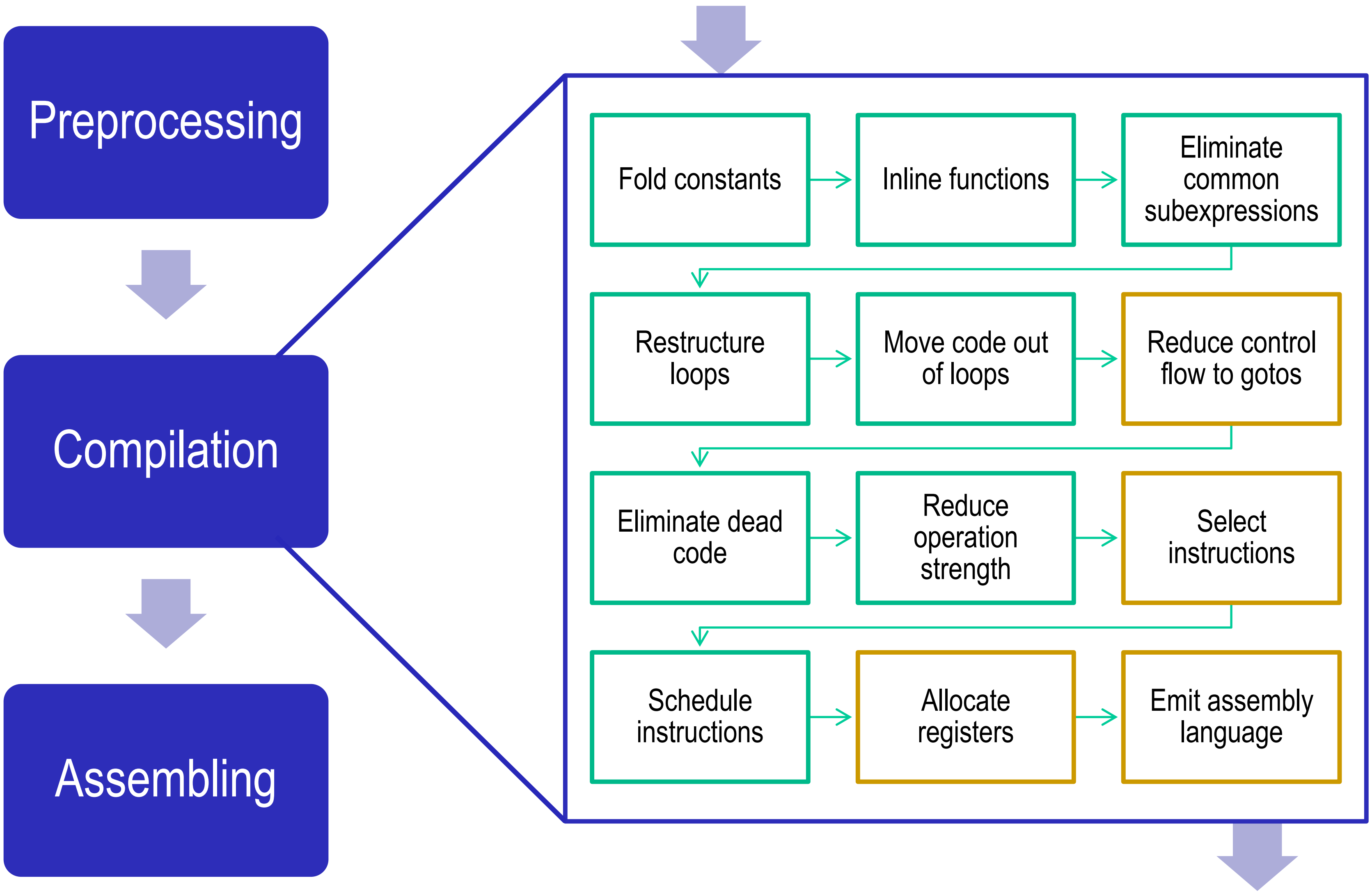
Goals of compiler optimization

- **Minimize number of instructions**
  - Don’t do calculations more than once
  - Don’t do unnecessary calculations at all
  - Avoid slow instructions (multiplication, division)
- **Avoid waiting for memory**
  - Keep everything in registers whenever possible
  - Access memory in cache-friendly patterns
  - Load data from memory early, and only once
- **Avoid branching**
  - Don’t make unnecessary decisions/branches at all
  - Make it easier for the CPU to predict branch destinations
  - “Unroll” loops to spread cost of branches over more instructions

Limits to compiler optimization

- **Generally cannot improve algorithmic complexity**
  - Only constant factors, but those can be worth 10x or more for some cases
- **Must not cause *any* change in program behavior**
  - Programmer expect the program runs as they developed/tested
  - Note: language may declare some changes acceptable
    - e.g., “Undefined behavior” (signed integer overflow)
- **Usually only analyze one function at a time**
  - Whole-program analysis (inter-procedure analysis) is usually too expensive or infeasible
- **Should (or should not) assume run-time inputs**
  - “Worst case” performance can be just as important as “normal case”
  - Especially for code exposed to *attacker-controlled* input (e.g. network servers, cryptocurrency networks)

Compilation is a pipeline



Two kinds of optimizations

- **Local optimizations**
  - Work inside a single *basic block*
  - Constant folding, strength reduction, (local) CSE, …

- **Global optimizations**
  - Process the entire *control flow graph* of a function
  - Loop nest optimization, code motion, (global) CSE, dead code elimination, …

Constant Folding

- **Do arithmetic in the compiler**

```
  long mask = 0xFF << 8;
➔ long mask = 0xFF00;
```

- **Any expression with constant inputs can be folded**

- **Might even be able to remove library calls…**

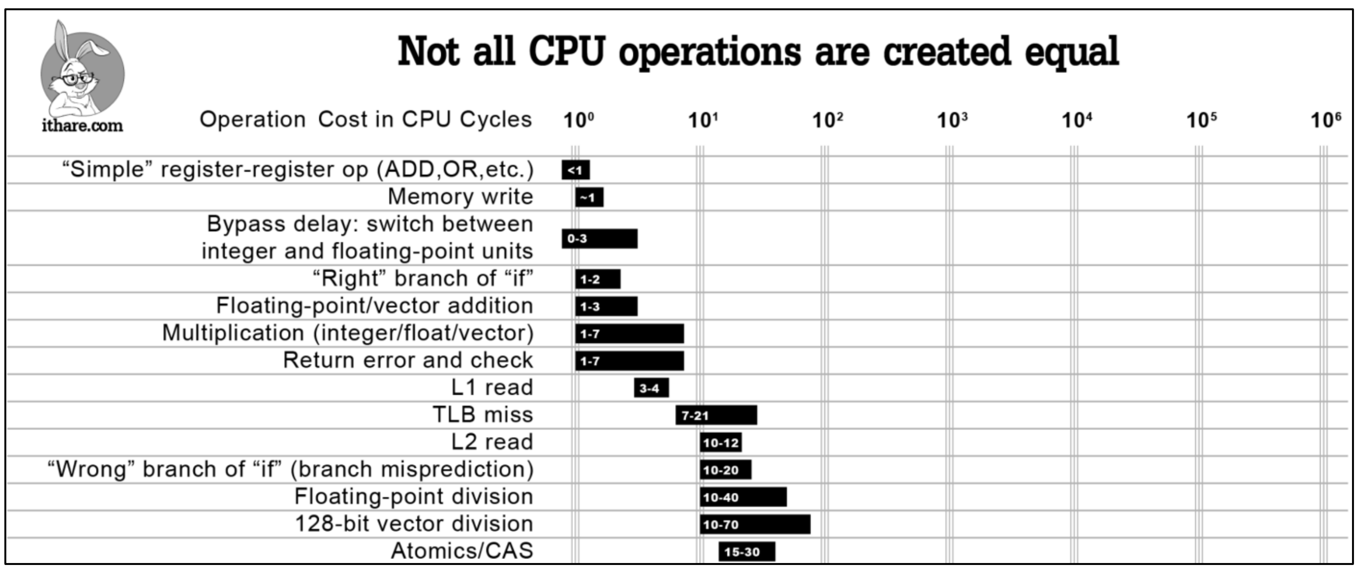
```
  size_t namelen = strlen("Harry Bovik");
➔ size_t namelen = 11;
```

Strength reduction

- **Replace expensive operations with cheaper ones**

```
  long a = b * 5;
➔ long a = (b << 2) + b;
```

- **Multiplication and division are the usual targets**



http://ithare.com/wp-content/uploads/part101_infographics_v08.png

Dead code elimination

- **Don't emit code that will never be executed**

```
if (0) { puts("Kilroy was here"); }
if (1) { puts("Only bozos on this bus"); }
```

- **Don't emit code whose result is overwritten**

```
x = 0;
x = 23;
```

- **These may look silly, but...**
  - Can be produced by other optimizations
  - Assignments to x might be far apart
  - Involve complex control-flows/data-flows

Common Subexpression Elimination (CSE)

- **Factor out common expression, only evaluate them once**

```
norm[i] = v[i].x * v[i].x + v[i].y * v[i].y;
```

```
elem = &v[i];
x = elem->x;
y = elem->y;
norm[i] = x*x + y*y;
```

Inlining

- **Copy body of a function into its caller(s)**
  - Can create opportunities for many other optimizations
  - Can make code much bigger and therefore slower

```
int pred(int x) {
    if (x == 0)
        return 0;
    else
        return x - 1;
}

int func(int y) {
    return pred(y)
         + pred(0)
         + pred(y+1);
}
```

```
int func(int y) {
  int tmp;
  if (y == 0) tmp = 0; else tmp = y - 1;
  if (0 == 0) tmp += 0; else tmp += 0 - 1;
  if (y+1 == 0) tmp += 0; else tmp += (y + 1) - 1;
  return tmp;
}
```

Inlining

- **Copy body of a function into its caller(s)**
  - Can create opportunities for many other optimizations
  - Can make code much bigger and therefore slower

```
int pred(int x) {
    if (x == 0)
        return 0;
    else
        return x - 1;
}

int func(int y) {
    return pred(y)
         + pred(0)
         + pred(y+1);
}
```

```
int func(int y) {
  int tmp;
  if (y == 0) tmp = 0; else tmp = y - 1;
  if (0 == 0) tmp += 0; else tmp += 0 - 1;
  if (y+1 == 0) tmp += 0; else tmp += (y + 1) - 1;
  return tmp;
}
```

Always true

Does nothing

Can constant fold

Inlining

- **Copy body of a function into its caller(s)**
  - Can create opportunities for many other optimizations
  - Can make code much bigger and therefore slower

```
int func(int y) {
  int tmp;
  if (y == 0) tmp = 0; else tmp = y - 1;
  if (0 == 0) tmp += 0; else tmp += 0 - 1;
  if (y+1 == 0) tmp += 0; else tmp += (y + 1) - 1;
  return tmp;
}
```

```
int func(int y) {
  int tmp = 0;
  if (y != 0) tmp = y - 1;

  if (y != -1) tmp += y;
  return tmp;
}
```

Code Motion

- **Move calculations out of a loop**
- **Only valid if every iteration would produce same result**
  - So called "loop invariant"

```
long j;
for (j = 0; j < n; j++)
    a[n*i+j] = b[j];
```

⇩

```
long j;
int n_i = n*i;
for (j = 0; j < n; j++)
    a[n_i+j] = b[j];
```

Loop Unrolling

- Amortize cost of loop condition by duplicating body
- Creates opportunities for CSE, code motion, etc.
- Can hurt performance by increasing code size

```
for (size_t i = 0; i < nelts; i++) {
    A[i] = B[i]*k + C[i];
}
```

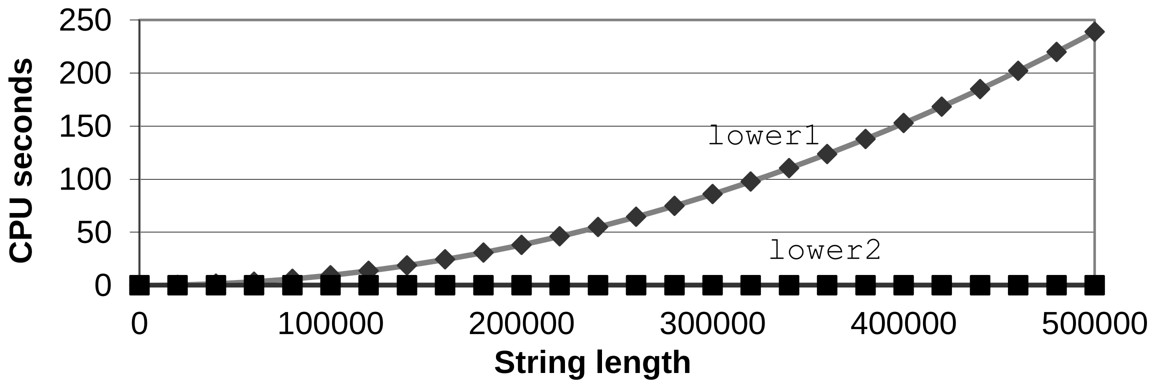
```
for (size_t i = 0; i < nelts - 4; i += 4) {
    A[i  ] = B[i  ]*k + C[i  ];
    A[i+1] = B[i+1]*k + C[i+1];
    A[i+2] = B[i+2]*k + C[i+2];
    A[i+3] = B[i+3]*k + C[i+3];
}
```

When the compiler can't move something

Q. What's the asymptotic complexity of lower1() and lower2()?
Q. Can the compiler optimize the code from lower1() to lower2()?

```
void lower1(char *s)
{
  size_t i;
  for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

```
void lower2(char *s)
{
  size_t i, n = strlen(s);
  for (i = 0; i < n; i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```



Question: Undefined Behavior

```
1  long long_cmp_opt(const int a, const int b)
2  {
3    if (a > 0) {                    // a is positive
4      do_something();
5      if (b < 0) {                  // b is negative
6        do_something_else();
7        if ((a - b) > 0)            // always true or can it be false?
8          do_another_thing();
9      }
10   }
11 }
```

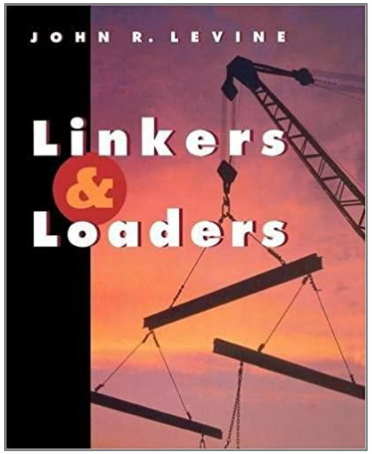
Q1. Is the condition check at line 7 always true?

Q2. Should compiler remove the condition check at line 7?

A. Signed overflow optimization hazards in the kernel (https://lwn.net/Articles/511259/)

Today

- **Basics of compiler optimization**
  - Principles and goals
  - Some example optimizations
  - Obstacles to optimization
- **Linking: combining object files into programs**
  - Symbols and symbol resolution
  - Relocation
  - Static libraries
  - Dynamic libraries



Linkers and Loaders, The Morgan Kaufmann Series in Software Engineering and Programming, 1st Edition, John R. Levine

Example C Program

- Two source files

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

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

int main(int argc, char** argv)
{
    int val = sum(array, 2);
    return val;
}
```

main.c

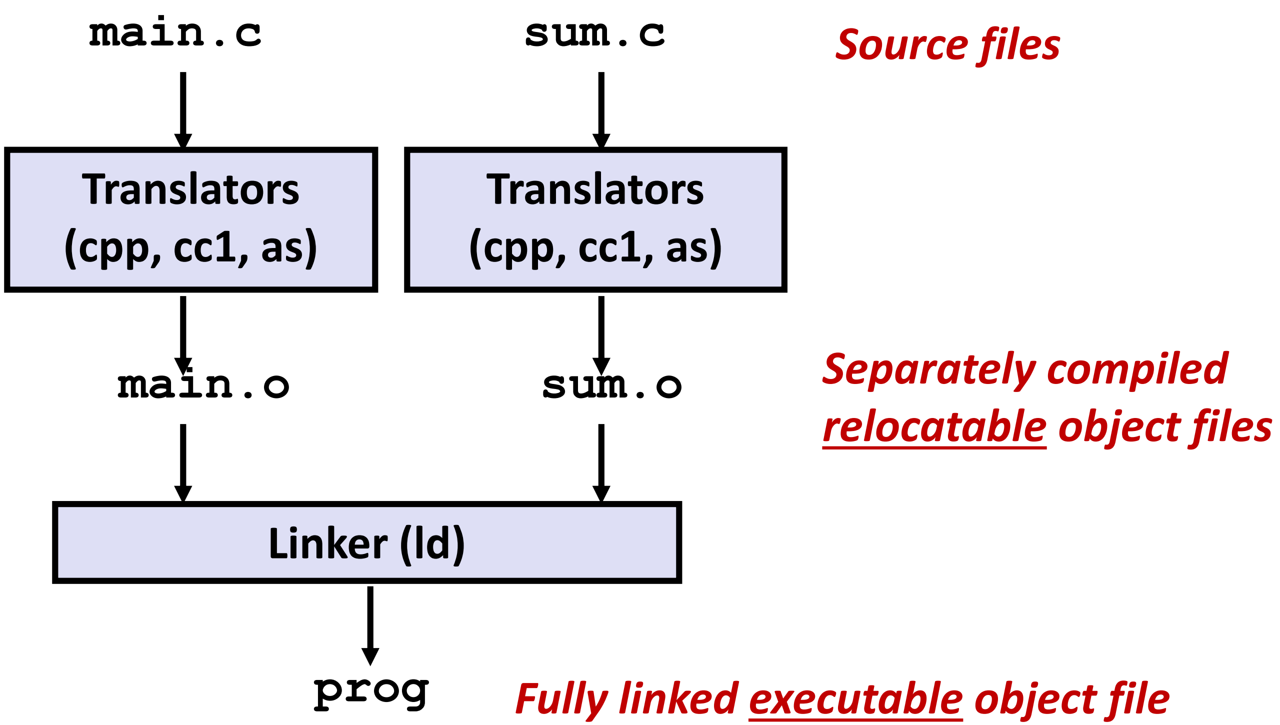
```c
int sum(int *a, int n)
{
    int i, s = 0;

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

sum.c

Linking

- Programs are translated and linked using a *compiler driver*:
  - `$ gcc -Og -o prog main.c sum.c`
  - `$ ./prog`



Why should you know “Linking”

- **To maintain a build system for a large-scale project**
  - Android? Python? TensorFlow?
  - In order to write a complex “Makefile/CMake”, you will need to understand the things behind the scene

- **To clearly understand the important systems concepts**
  - Shared libraries
  - Loading
  - Memory mapping
  - Virtual memory
  - The lifetime of “process”

What Do Linkers Do?

- **Step #1: Constructing symbol tables (assembler)**
- **Step #2: Symbol resolution (linker)**
- **Step #3: Relocation (linker)**

Step #1: Constructing symbol tables

- **Step #1: Constructing symbol tables**
  - **Symbol**
    - Definition and references of global variables and functions
      - `void swap() {…} /* define symbol swap */`
      - `swap();          /* reference symbol swap */`
      - `int *xp = &x;    /* define symbol xp, reference x */`
  - **Symbol table**
    - Symbol table stores symbol definitions, which are an array of entries
    - Each entry includes symbol name, size, and location of symbol.
  - Symbol resolution: the linker associates each symbol reference with the symbol definition

Symbols in Example C Program

Definitions

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

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

int main(int argc, char** argv)
{
    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

Reference

Linker Symbols

- **Every object file *m* has a table of symbols it defines or needs.**
- **Three types of symbols**
  - Global definitions
    - Symbols defined by *m* that can be referenced by other files.
    - In C, non-`static` functions and global variables.
  - Local definitions
    - Symbols that are defined by *m* but *cannot* be referenced by other files.
    - In C, functions and global/local variables defined with `static`.
    - **Note: this is different from local variables (in stack)**
  - External references
    - Symbols that *m* references but does not define.
    - These must be defined by some other module.

Step #2: Symbol Resolution

???

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

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

int main(int argc, char** argv)
{
    int val = sum(array, 2);
    return val;
}
```

main.c

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

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

sum.c

Relocation Entries in a object file

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

int main(int argc, char**
argv)
{
     int val = sum(array, 2);
     return val;
}                              main.c
```

Each relocation entry instructs how each reference should be relocated using the symbol table.

```
$objdump -r -d main.o
```

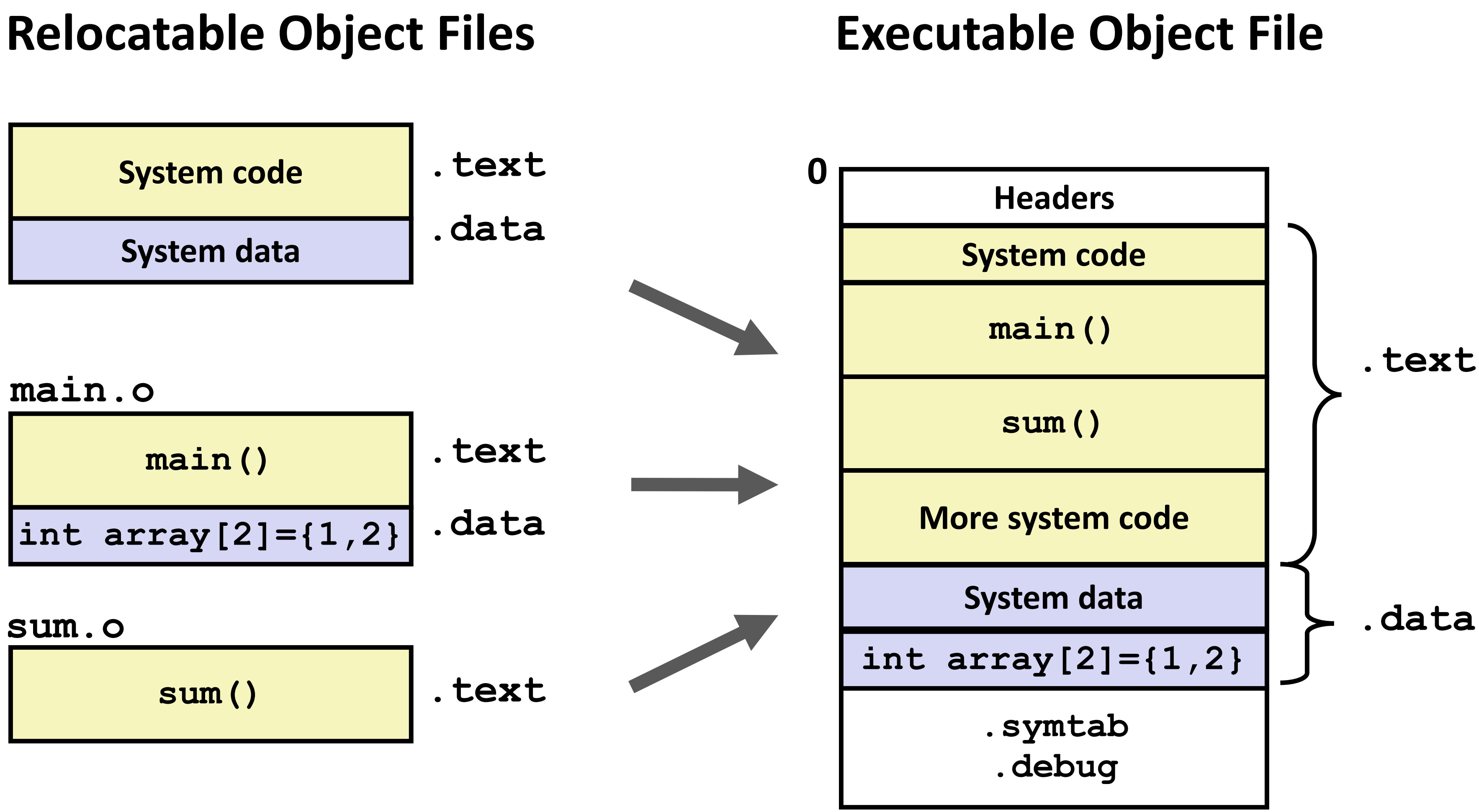
```
0000000000000000 <main>:
   0:    48 83 ec 08            sub    $0x8,%rsp
   4:    be 02 00 00 00         mov    $0x2,%esi
   9:    bf 00 00 00 00         mov    $0x0,%edi        # %edi = &array
                     a: R_X86_64_32 array               # Relocation entry

   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
```

Step #3. Relocation

- **Step #3. Relocation**
  - Takes place when linking
    - e.g., merging multiple object files (with multiple code/data sections) into a single executable file (with single code/data section)
  - Relocate symbols:
    - from their relative locations in the `.o` files
    - to their final absolute memory locations in the executable

Step #3: Relocation



Relocated .text section (executable)

```
$ objdump -d prog
```

```
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,%edi  # %edi = &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:       f3 c3                      repz retq
```

`callq` instruction uses PC-relative addressing for sum():

0x4004e8 = 0x4004e3 + 0x5

Libraries: Packaging a Set of Functions

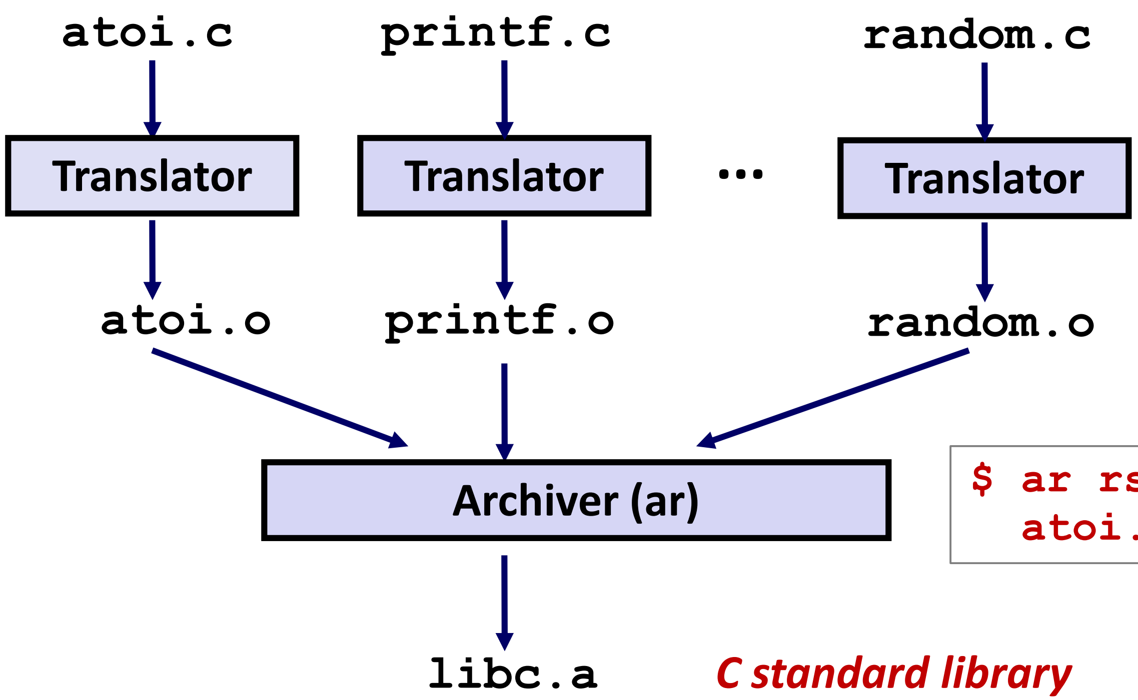
- **How to package functions commonly used by programmers?**
  - Math, I/O, memory management, string manipulation, etc.

- **Awkward, given the linker framework so far:**
  - **Option 1:** Put all functions into a single object file (e.g., a single lib file)
    - Programmers link big object file into their programs
    - Space and time inefficient
  - **Option 2:** Put each function in a separate object file (e.g., multiple lib files)
    - Programmers explicitly link appropriate binaries into their programs
    - More efficient, but burdensome on the programmer

Static Libraries

- **Static libraries (`.a` archive files)**
  - Concatenate related relocatable object files into a single file with an index (called an *archive*).
  - Linker resolves unresolved external references by looking for the symbols in one or more archives.
  - If an archive member file resolves reference, link it into the executable.

Creating Static Libraries




```
$ ar rs libc.a \
  atoi.o printf.o … random.o
```

- Archiver allows incremental updates
- Recompile function that changes and replace .o file in archive.

Commonly Used Libraries

`libc.a` (the C standard library)

- 4.6 MB archive of 1496 object files.
- I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math

`libm.a` (the C math library)

- 2 MB archive of 444 object files.
- floating point math (sin, cos, tan, log, exp, sqrt, …)

```
$ ar -t /usr/lib/libc.a | sort
…
fork.o
…
fprintf.o
fpu_control.o
fputc.o
freopen.o
fscanf.o
fseek.o
fstab.o
…
```

```
$ ar -t /usr/lib/libm.a | sort
…
e_acos.o
e_acosf.o
e_acosh.o
e_acoshf.o
e_acoshl.o
e_acosl.o
e_asin.o
e_asinf.o
e_asinl.o
…
```

Linking with Static Libraries

```c
#include <stdio.h>
#include "vector.h"

int x[2] = {1, 2};
int y[2] = {3, 4};
int z[2];

int main(int argc, char**
argv)
{
    addvec(x, y, z, 2);
    printf("z = [%d %d]\n”,
           z[0], z[1]);
    return 0;
}
```

main2.c

libvector.a

```c
void addvec(int *x, int *y,
            int *z, int n) {
    int i;

    for (i = 0; i < n; i++)
        z[i] = x[i] + y[i];
}
```

addvec.c

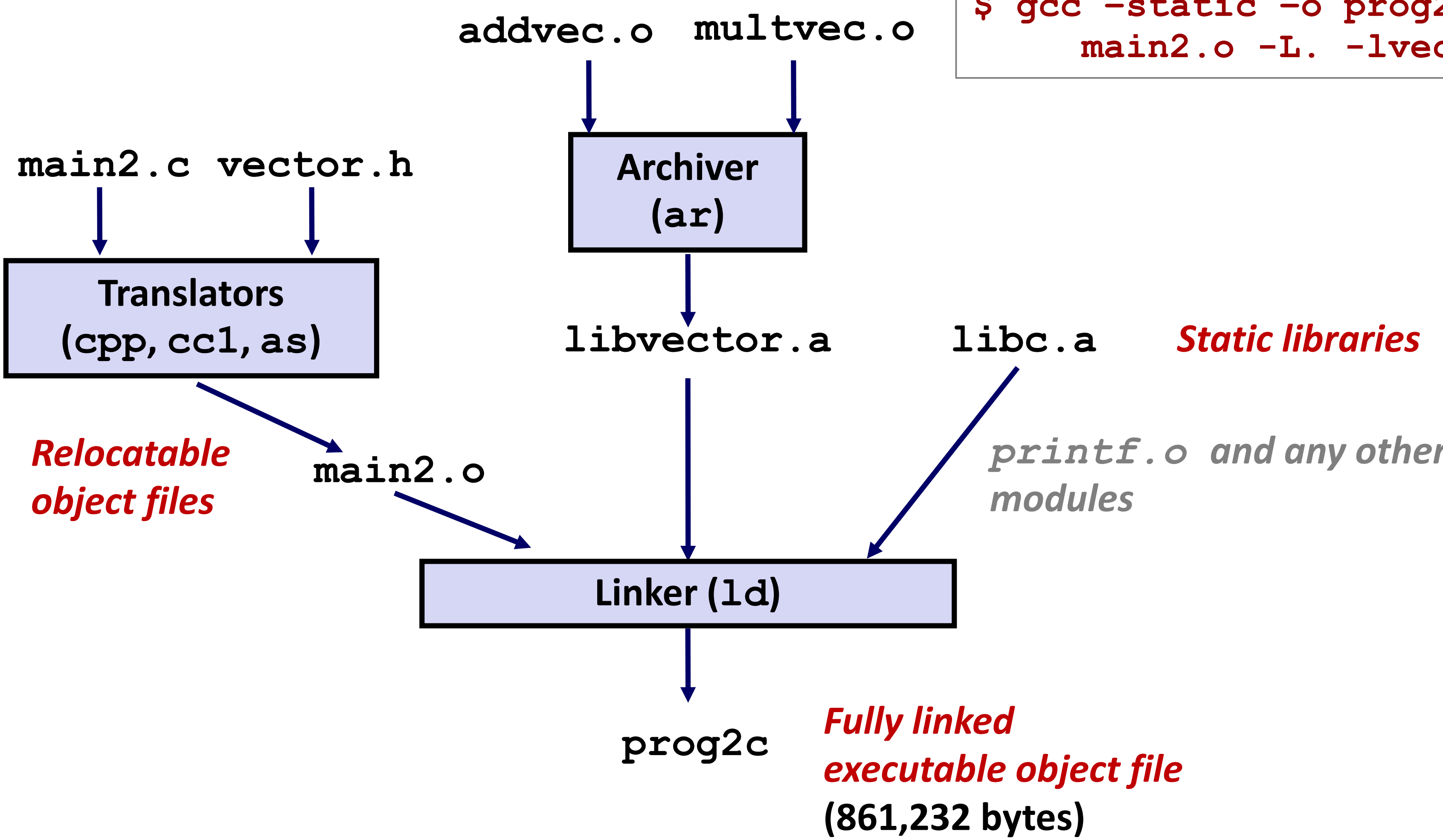
```c
void multvec(int *x, int *y,
             int *z, int n)
{
    int i;

    for (i = 0; i < n; i++)
        z[i] = x[i] * y[i];
}
```

multvec.c

Linking with Static Libraries

```
$ gcc -static -o prog2c \
    main2.o -L. -lvector
```



"c" for "compile-time"

Shared Libraries

- **Limitations of Static libraries**
  - Duplication in the linked executables (every program needs libc)
  - Duplication in the running executables
    - This may not be a problem: de-duplication?
  - Minor bug fixes of system libraries require each application to explicitly relink
    - Stack overflow in glibc's getaddrinfo() (CVE-2015-7547)
      - If linking glibc statically, an executable has to be rebuilt
      - https://security.googleblog.com/2016/02/cve-2015-7547-glibc-getaddrinfo-stack.html

- **Shared libraries (or dynamic library, or dynamic linking)**
  - Object files that contain code and data that are loaded and linked into an application *dynamically,* at either *load-time* or *run-time*
  - Also called: DLLs, `.so` files

Shared Libraries (cont.)

- **Load-time linking: Dynamic linking can occur when executable is first loaded and run**
  - Common case for Linux, handled automatically by the dynamic linker (`ld-linux.so`)
  - Standard C library (`libc.so`) is usually dynamically linked

- **Run-time linking: Dynamic linking can also occur after program has begun**
  - In Linux, this is done by calls to the `dlopen()` interface
  - Lazy loading with lazy binding

What dynamic libraries are required?

- Use "`ldd`" to find out:

```
$ ldd prog
  linux-vdso.so.1 =>  (0x00007ffcf2998000)
  libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f99ad927000)
  /lib64/ld-linux-x86-64.so.2 (0x00007f99adcef000)
```