[CSED211] Introduction to Computer Software Systems

Lecture 12: Linking

Prof. Jisung Park

2023.11.13

Lecture Agenda: Linking

- Linking
  - Motivation
  - What It Does
  - How It Works
  - Dynamic Linking
- Case Study: Library Interpositioning

C Program Example

main.c

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

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

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

sum.c

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

Static Linking

- Programs are translated and linked using a compiler driver

```
$ gcc -Og -o prog main.c sum.c
$ ./prog
```

`main.c` `sum.c` **Source Files**

Static Linking

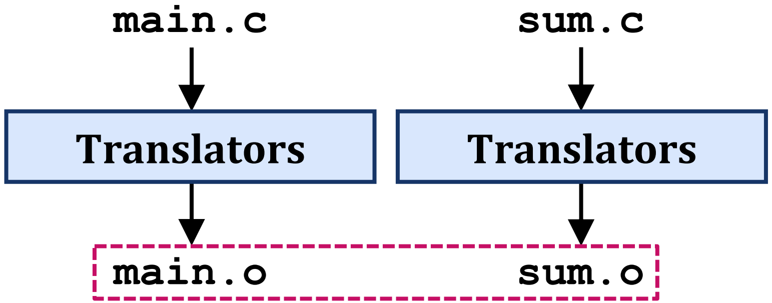
- Programs are translated and linked using a compiler driver

```
$ gcc -Og -o prog main.c sum.c
$ ./prog
```

Static Linking

- Programs are translated and linked using a compiler driver

```
$ gcc -Og -o prog main.c sum.c
$ ./prog
```



Source Files

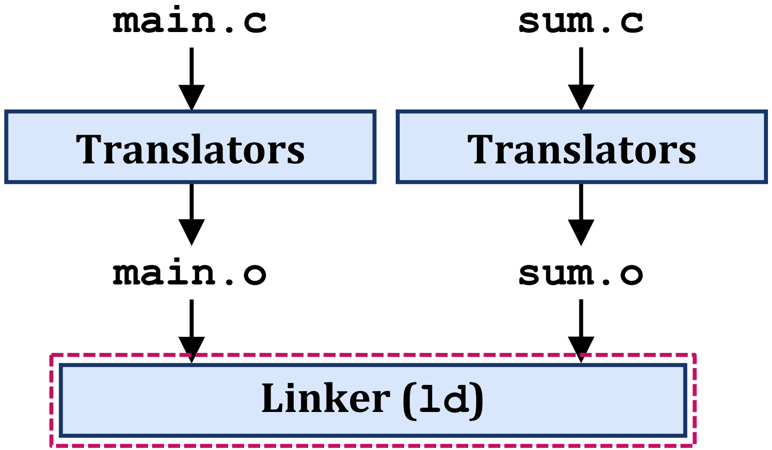
Preprocessing **(`cpp`),** ***compile*** **(`cc1`),** ***assembly*** **(`as`)**

Separately-Compiled Relocatable Object Files

Static Linking

- Programs are translated and linked using a compiler driver

```
$ gcc -Og -o prog main.c sum.c
$ ./prog
```



Source Files

***Preprocessing* (`cpp`), *compile* (`cc1`), *assembly* (`as`)**

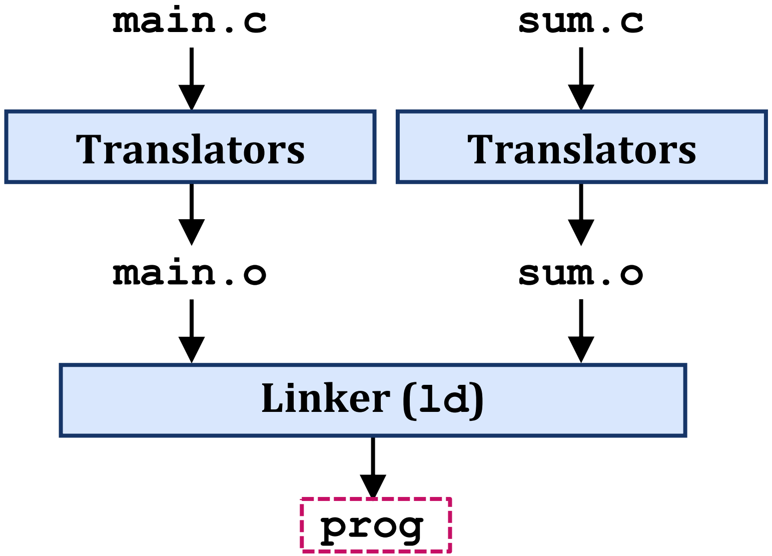
Separately-Compiled Relocatable Object Files

***Merges code and data for all functions defined in multiple modules* (`main.c` and `sum.c`)**

Static Linking

- Programs are translated and linked using a compiler driver

```
$ gcc -Og -o prog main.c sum.c
$ ./prog
```



Source Files

Preprocessing (`cpp`), *compile* (`cc1`), *assembly* (`as`)

Separately-Compiled Relocatable Object Files

Merges code and data for all functions defined in multiple modules (`main.c` and `sum.c`)

Fully-Linked Executable Object File

Why Linkers?

- Reason 1: modularity
    - Program can be written as a collection of smaller source files, rather than one monolithic mass
    - Can build libraries of common functions (more on this later)
        - e.g., math library, standard C library

Why Linkers? (Cont.)

- Reason 2: Efficiency
  - Time: separate compilation
    - Change one source file, compile, and the relink
    - No need to recompile other source files
    - Can compile multiple files concurrently
  - Space: libraries
    - Common functions can be aggregated into a single file in two ways
    - Option 1: static linking
      - Executable files and running memory images contain only the library code they actually use
    - Option 2: dynamic linking
      - Executable files contain no library code
      - During execution, a single copy of library code can be shared across all executing processes

What Do Linkers Do?

- Step 1. symbol resolution
  - Programs define and reference symbols (variables and functions)

```
void swap(){…}; /* define symbol swap */
swap();         /* reference symbol swap */
int *xp = &x;   /* define symbol xp, reference x */
```

  - Compiler stores symbol definitions in symbol table
    - An array of structs, each includes the name, size, and location of a symbol
  - Linker associates each symbol reference with exactly one symbol definition

Symbol Resolution: C Program Example

Definitions

main.c

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

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

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

sum.c

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

Reference

What Do Linkers Do?

- Step 1. symbol resolution
  - Programs define and reference symbols (variables and functions)

```
void swap(){…}; /* define symbol swap */
swap();         /* reference symbol swap */
int *xp = &x;   /* define symbol xp, reference x */
```

  - Compiler stores symbol definitions in symbol table
    - An array of structs, each includes the name, size, and location of a symbol
  - Linker associates each symbol reference with exactly one symbol definition

- Step 2. relocation
  - Merges separate code and data sections into single sections
  - Relocates symbols from their relative locations in the `.o` files to their final absolute memory locations in the executable
  - Updates all references to these symbols to reflect their new positions

Three Kinds of Object Files (Modules)

- Relocatable object file (`.o` file)
  - Contains code and data in a form that can be combined with other relocatable object files to form executable object file
  - Each `.o` file is produced from exactly one source (`.c`) file
- Executable object file (`a.out` file)
  - Contains code and data in a form that can be copied directly into memory and then executed
- Shared object file (`.so` file)
  - Special type of relocatable object file that can be loaded into memory and linked dynamically, at either loading time or run-time
  - Called Dynamic Link Libraries (DLLs) in Windows

Executable and Linkable Format (ELF)

- Standard binary format for object files

- One unified format for
  - Relocatable object files (`.o`)
  - Executable object files (`a.out`)
  - Shared object files (`.so`)

- Generic name: ELF binaries

ELF Object File Format

- ELF header: word size, byte ordering, file type, machine type, etc.
- Segment header table: page size, virtual addresses memory segments (sections), segment sizes
- `.text` section: code
- `.rodata` section: read-only data, e.g., jump tables, etc.
- `.data` section: initialized global variables
- `.bss` section
  - Uninitialized global variables
  - Block Started by Symbol
  - Better Save Space
  - Has section header but occupies no space

ELF header
Segment header table
.text section
.rodata section
.data section
.bss section
.symtab section
.rel.txt section
.rel.data section
.debug section
Section header table

ELF Object File Format (Cont.)

- **`.symtab`** section
  - Symbol table
  - Procedure and static variable names
  - Section names and locations
- **`.rel.text`** section
  - Relocation information for **`.text`** section
  - Addresses of instructions that will need to be modified in the executable
  - Instructions for modifying
- **`.rel.dat`** section
  - Relocation information for **`.data`** section
  - Addresses of pointer data that will need to be modified in the merged executable

ELF header
Segment header table
.text section
.rodata section
.data section
.bss section
.symtab section
.rel.txt section
.rel.data section
.debug section
Section header table

ELF Object File Format (Cont.)

- `.debug` section
  - Info for symbolic debugging (`gcc -g`)
- Section header table
  - Offsets and sizes of each section

ELF header
Segment header table
.text section
.rodata section
.data section
.bss section
.symtab section
.rel.txt section
.rel.data section
.debug section
Section header table

Linker Symbols

- Global symbols
  - Defined by module M that can be referenced by other modules
  - e.g., non-static C functions and non-static global variables

- External symbols
  - Global symbols referenced by module M but defined by another module

- Local symbols
  - Defined and referenced exclusively by module M
  - e.g., C functions and global variables defined with the static attribute
  - Local linker symbols are not local program variables

Step 1: Symbol Resolution

Global Symbol Definitions

`main.c`

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

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

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

Global Symbol References

`sum.c`

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

Local Variable Definitions

Linker knows nothing of them

Symbol Identification

- Which of the following names will be in the symbol table of **`symbols.o`**?
  - `time`
  - `foo`
  - `a`
  - `argc`
  - `argv`
  - `b`
  - `main`
  - `printf`
  - Any others?

`symbols.c`

```c
int time;

int foo(int a){
  int b = a + 1;
  return b;
}

int main(int argc, char** argv){
  printf("%d\n", foo(5));
  return 0;
}
```

Symbol Identification

- Which of the following names will be in the symbol table of `symbols.o`?
  - `time`
  - `foo`
  - `a`
  - `argc`
  - `argv`
  - `b`
  - `main`
  - `printf`
  - Any others?

- Can find this with `readelf`

```
$ readelf -s symbols.o
```

symbols.c

```c
int time;

int foo(int a){
  int b = a + 1;
  return b;
}

int main(int argc, char** argv){
  printf("%d\n", foo(5));
  return 0;
}
```

Local Symbols

- Local non-static C variables vs. Local static C variables
  - Local non-static: stored on the stack
  - Local static: stored in either `.bss` or `.data`

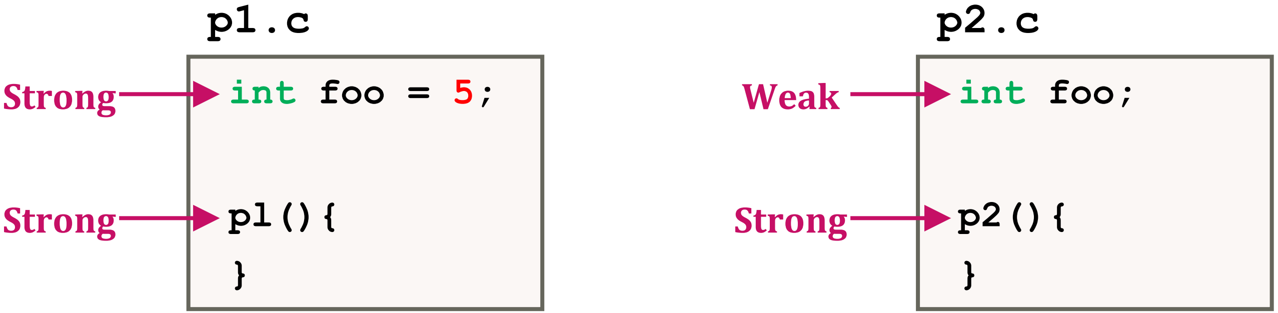
Complier allocates space in **`.data`**
for each definition of **`x`**

Creates local static symbols in the symbol table
with unique names, e.g., **`x`*****,*** **`x.1721`*****, and*** **`x.1724`**

```c
static int x = 15;

int f(){
  static int x = 17;
  return x++;
}

int g(){
  static int x = 19;
  return x += 14;
}

int h(){
  return x += 27;
}
```

How Linker Resolves Duplicate Symbol Definitions

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



Linker's Symbol Rules

- Rule 1: multiple strong symbols are not allowed
  - Each item can be defined only once
  - Otherwise, linker error
- Rule 2: a strong symbol and multiple weak symbols → the strong symbol
  - References to the weak symbol resolve to the strong symbol
- Rule 3: multiple weak symbols → pick an arbitrary one
  - Can override this with `gcc -fno-common`

Linker Puzzles

p1.c	p2.c	
`int x;` `p1(){}`	`p1(){}`	**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, y;` `p1(){}`	`double x;` `p2(){}`	Writes to `x` in `p2.c` might overwrite `y`!
`int x = 7, y = 5;` `p1(){}`	`double x;` `p2(){}`	Writes to `x` in `p2.c` will overwrite `y`!
`int x = 7;` `p1(){}`	`int x;` `p2(){}`	References to `x` will refer to the initialized variable (in `p1.c`)

Nightmare scenario: two identical weak structs compiled with different alignment rules.

Type Mismatch Example

- Compiles without any errors or warnings, but what gets printed?

mismatch-main.c

```c
long int x; /* Weak symbol */

int main(int argc, char** argv){
  printf("%ld\n", x);
  return 0;
}
```

mismatch-variable.c

```c
/* Global strong symbol */
double x = 3.14;
```

Global Variables

- Avoid 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
    - Treated as weak symbol
    - But also causes linker error if not defined in some file

Use of extern in .h Files

c1.c

```
#include “global.h”

int f(){
  return g + 1;
}
```

global.h

```
extern int g;
int f();
```

c2.c

```
#include <stdio.h>
#include “global.h”

int main(int argc, char** argv){
  if(init)
    // do something, e.g., g=31;
  int t = f();
  printf(“Calling f yields %d\n", t);
  return 0;
}
```

Use of extern in .h Files

c1.c

```
#include "global.h"

int f(){
  return g + 1;
}
```

global.h

```
#ifdef INITIALIZE
  int g = 23;
  static int init = 1;
#else
  extern int g;
  static int init = 0;
#endif
```

c2.c

```
#define INITIALIZE
#include <stdio.h>
#include "global.h"

int main(int argc, char** argv){
  if(init)
    // do something, e.g., g=31;
  int t = f();
  printf("Calling f yields %d\n", t);
  return 0;
}
```

Use of extern in .h Files

c1.c

```c
int g;
static int init = 0;
int f(){
  return g + 1;
}
```

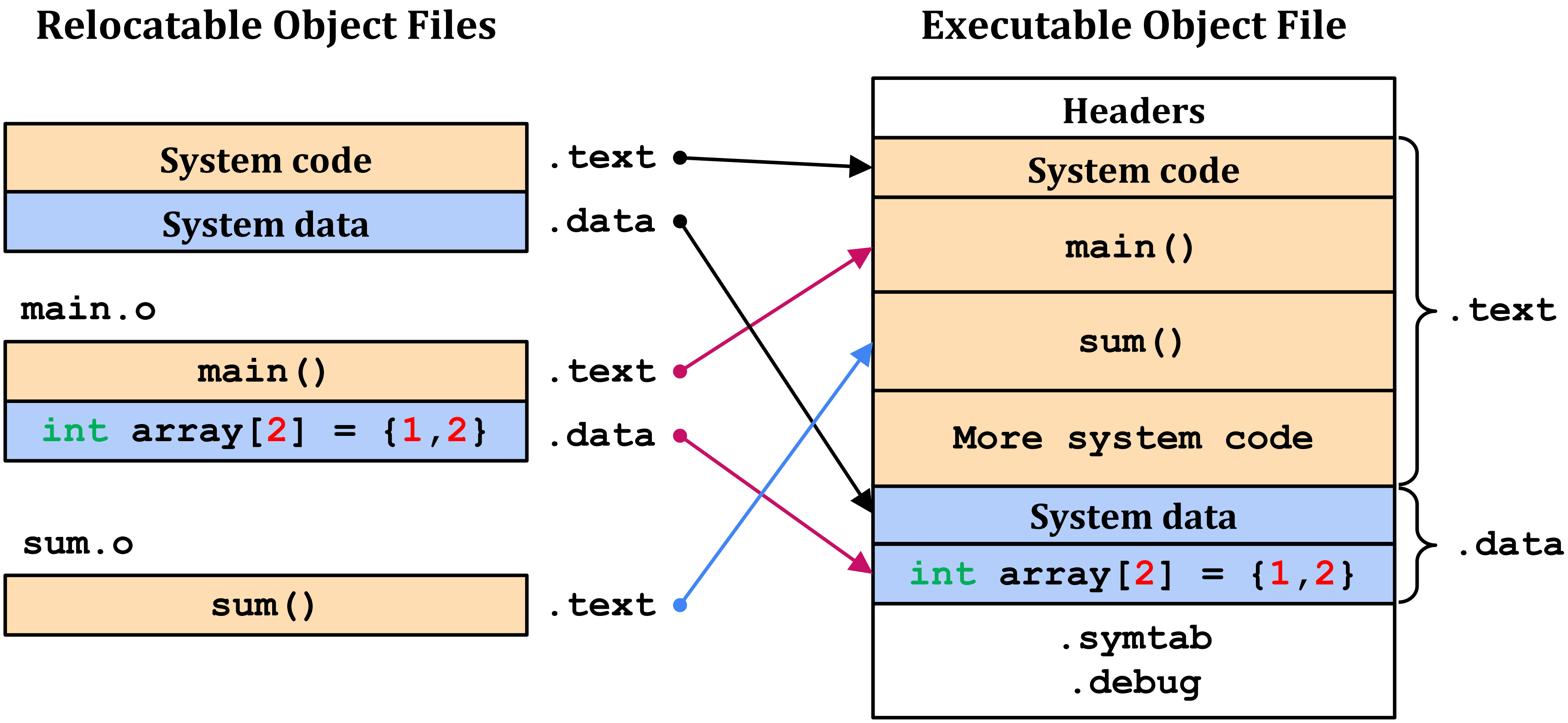
global.h

```c
#ifdef INITIALIZE
  int g = 23;
  static int init = 1;
#else
  extern int g;
  static int init = 0;
#endif
```

c2.c

```c
#define INITIALIZE
#include <stdio.h>
int g = 23;
static int init = 1;

int main(int argc, char** argv){
  if(init)
    // do something, e.g., g=31;
  int t = f();
  printf("Calling f yields %d\n", t);
  return 0;
}
```

Step 2: Relocation



Relocation Entries

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

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

```
0000000000000000 <main>:                                    main.o
  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
```

Relocated `.text` Section

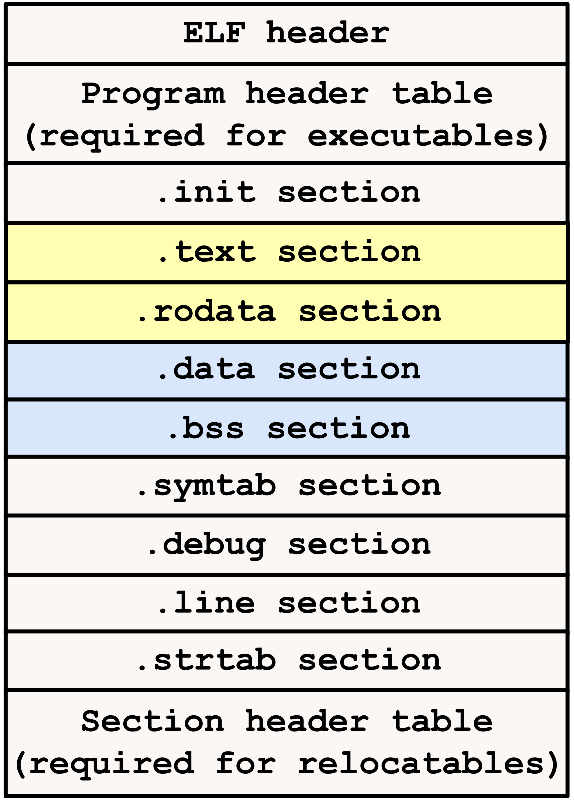
```
00000000004004d0 <main>:
  4004d0: 48 83 ec 08       sub    $0x8,%rsp
  4004d4: be 02 00 00 00    mov    $0x2,%esi
  4004d9: bf 18 10 60 00    mov    $0x601018,%edi  # %edi = &array
  4004de: e8 05 00 00 00    callq  4004e8 <sum>    # sum()
  4004e3: 48 83 c4 08       add    $0x8,%rsp
  4004e7: c3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

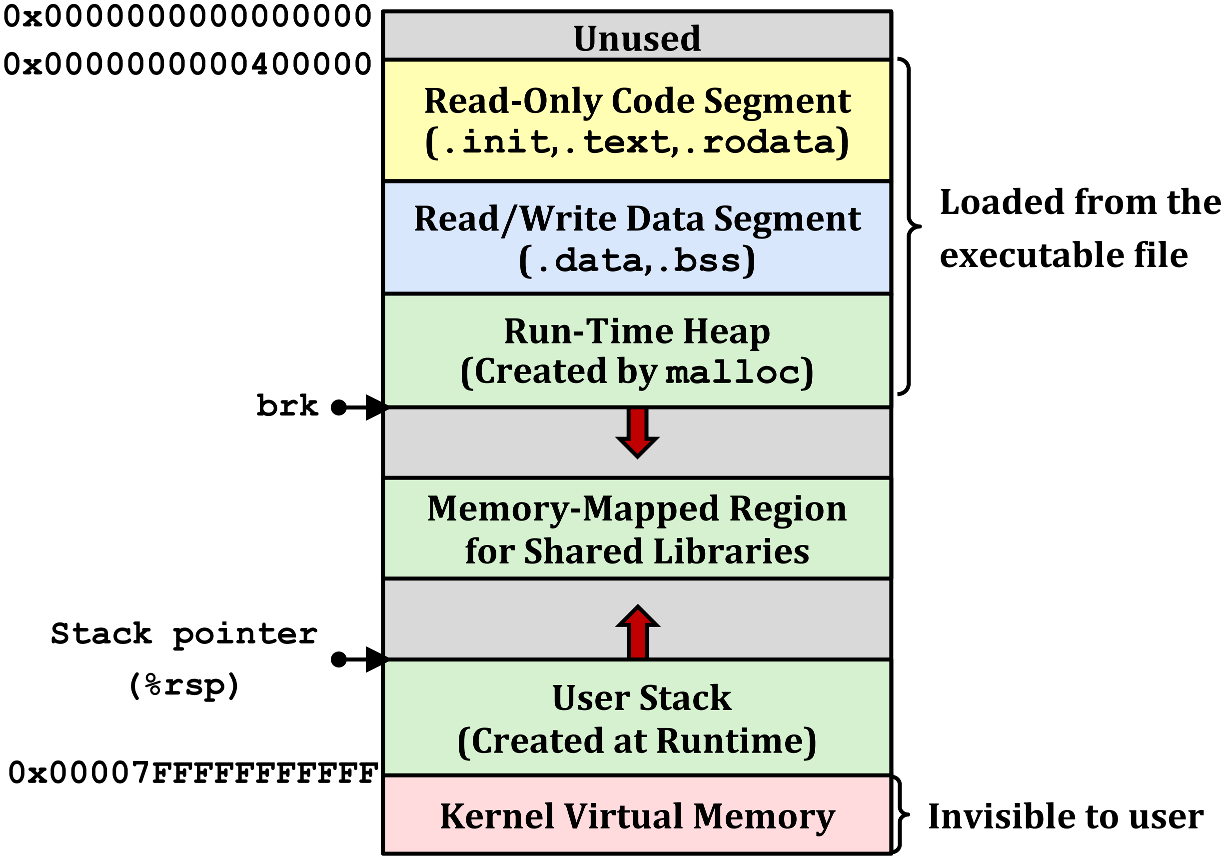
Using PC-relative addressing for `sum()`:
`0x4004e8 = 0x4004e3 + 0x05`

Source: `objdump -dx prog`

Loading Executable Object Files

Executable Object File





Packaging Commonly Used Functions

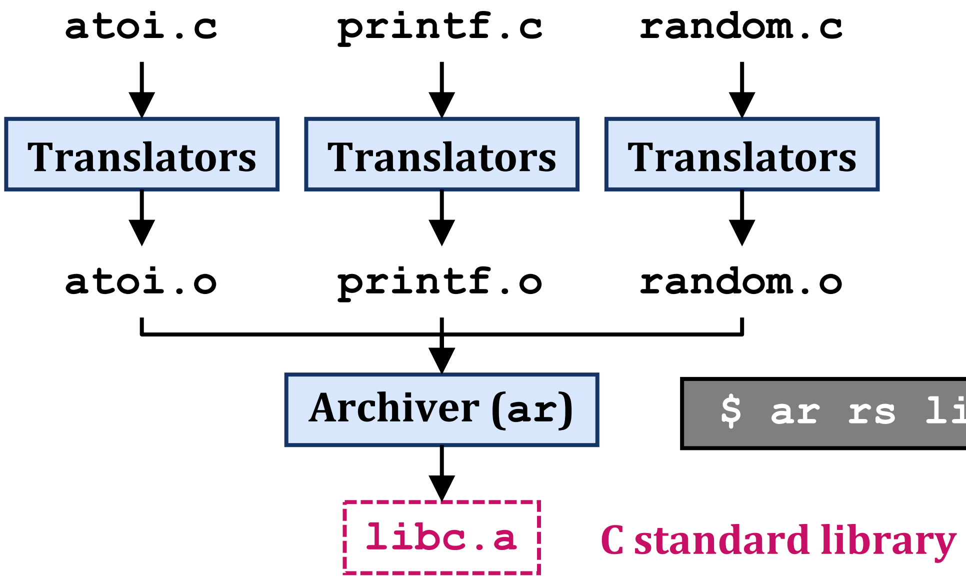
- How to package functions commonly used by programmers?
  - e.g., math, I/O, memory management, string manipulation, etc.

- Awkward, given the linker framework so far
  - Option 1: put all functions into a single source file
    - Programmers link big object file into their programs
    - Space and time inefficient
  - Option 2: put each function in a separate source file
    - Programmers explicitly link appropriate binaries into their programs
    - More efficient, but burdensome on the programmer

Old-fashioned Solution: Static Libraries

- Static libraries (`.a` archive files)
  - Concatenate related relocatable object files into a single file with an index
    - Which is called an archive
  - Enhance linker so that it tries to resolve 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 1,496 object files
  - I/O, memory allocation, signal handling, string handling, data and time, random numbers, integer math, etc.

- **`libm.a`** (the C math library)
  - 2 MB archive of 444 object files
  - Floating point math (sin, cos, tan, log, exp, sqrt,etc.)

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

```
$ ar -t 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

main2.c

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

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

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

```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];
}
```

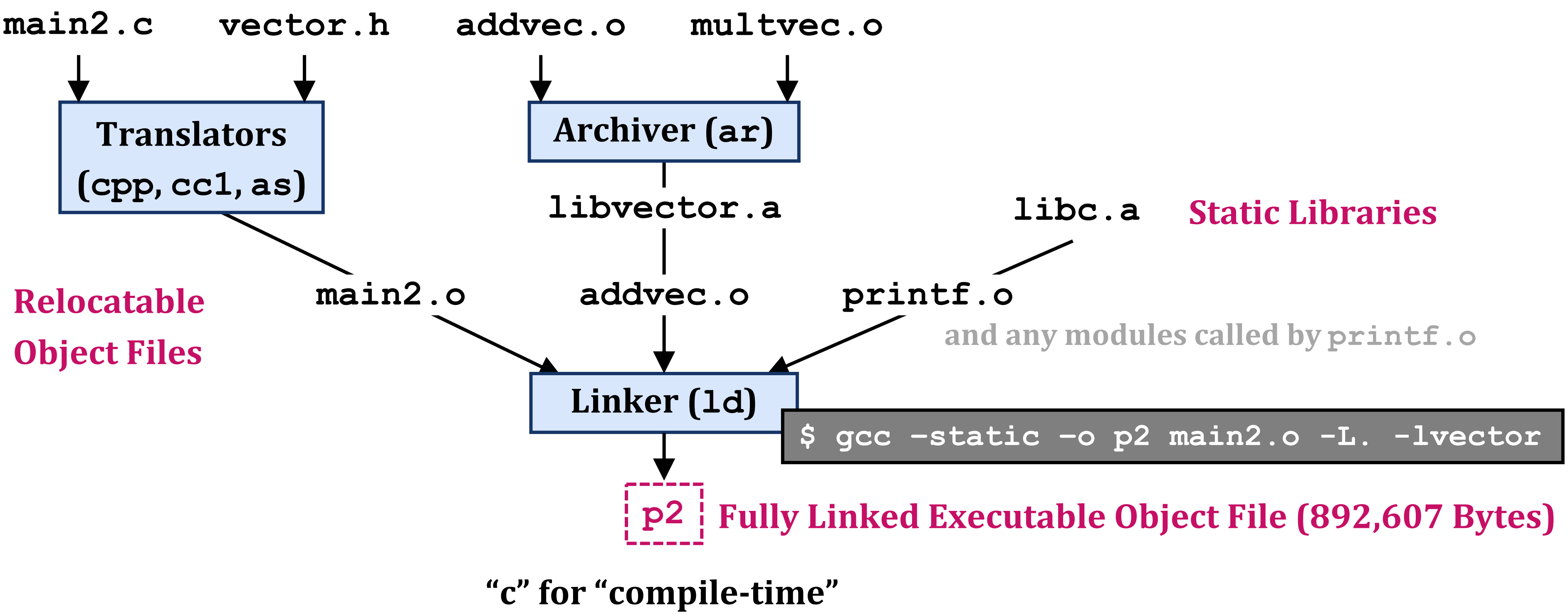
addvec.c

```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

libvector.a

Linking with Static Libraries



Using Static Libraries

- Linker's algorithm for resolving external references
  - Scan `.o` files and `.a` files in the command line order
  - During the scan, keep a list of the current unresolved references
  - As each new `.o` or `.a` file, `obj`, is encountered, try to resolve each unresolved reference in the list against the symbols defined in `obj`
  - If any entries in the unresolved list at end of scan, then error

- Problem
  - Command line order matters
  - Moral: put libraries at the end of the command line

```
$ gcc -L. libtest.o -lmine
$ gcc -L. -lmine libtest.o
libtest.o: In function `main':
libtest.o(.text+0x4): undefined reference to `libfun'
```

Modern Solution: Shared Libraries

- Static libraries have the following disadvantages:
  - Duplication in the stored executables (every function need std libc)
  - Duplication in the running executables
  - Minor bug fixes of system libraries require each application to explicitly relink
    - Rebuild everything with glibc?
    - https://security.googleblog.com/2016/02/cve-2015-7547-glibc-getaddrinfo-stack.html

- Modern solution: shared libraries
  - 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: dynamic link libraries (DLLs) in Windows and `.so` files in Linux

Shared Libraries (Cont.)

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

- Dynamic linking can also occur after program has begun
  - Run-time linking
  - In Linux, this is done by calls to the **`dlopen()`** interface
    - Distributing software
    - High-performance web servers
    - Runtime library interpositioning

- Shared library routines can be shared by multiple processes
  - More on this when we learn about virtual memory

What Dynamic Libraries Are Required?

- **`.interp`** section
  - Specifies the dynamic linker to use (i.e., **`ld-linux.so`**)

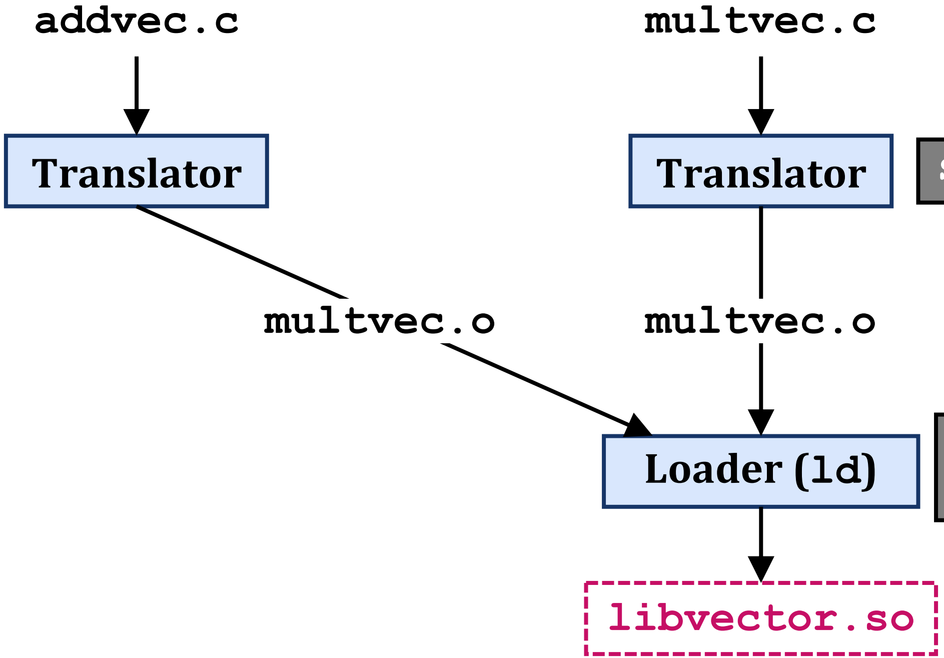
- **`.dynamic`** section
  - Specifies the names, etc. of the dynamic libraries to use
  - Follow an example of **`prog`**

```
(NEEDED)     Shared library: [libm.so.6]
```

- Where are the libraries found?
  - 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)
```

Dynamic Linking at Load-time



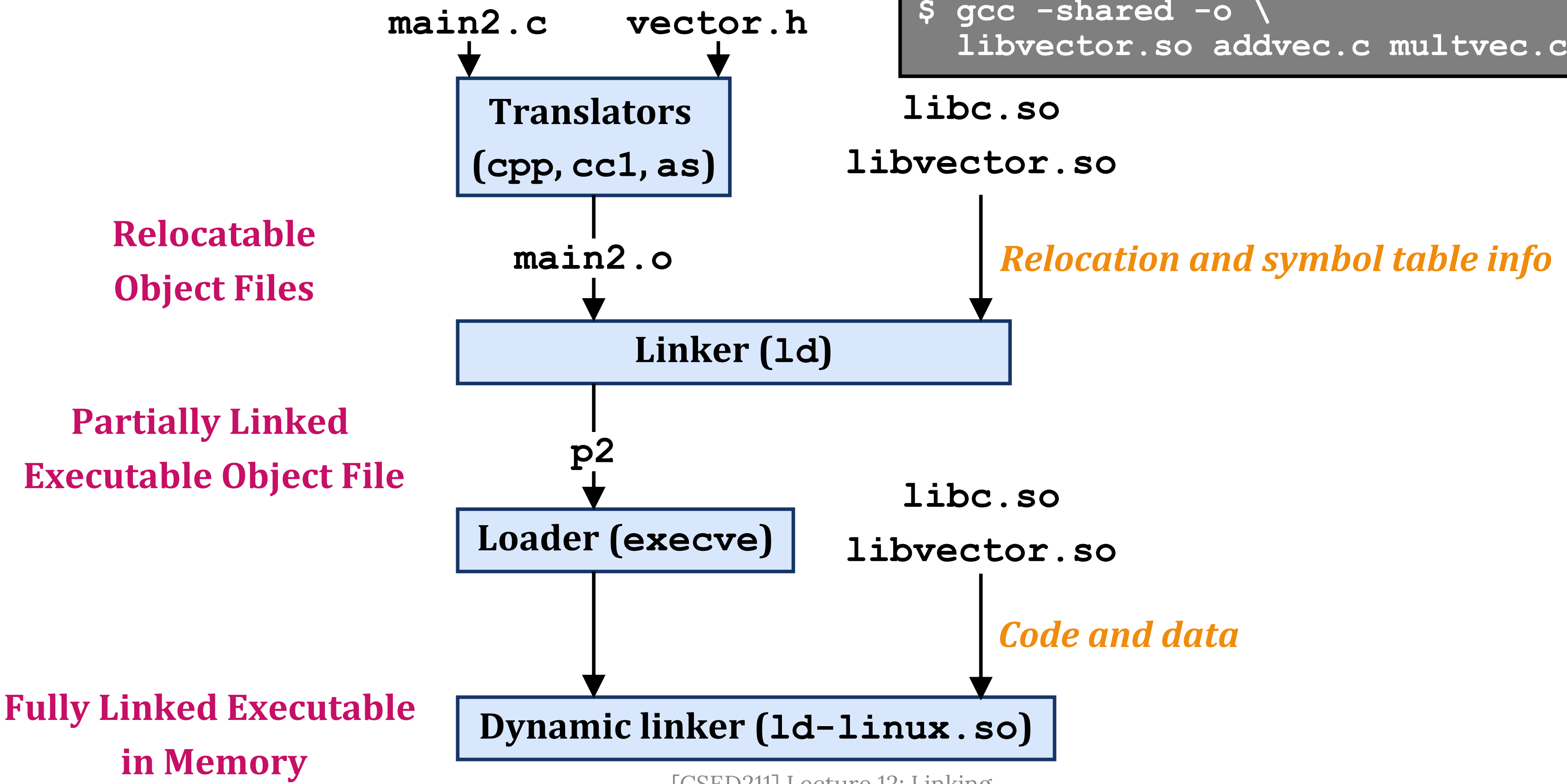

```
$ gcc -Og -c addvec.c multvec.c -fpic
```

```
$ gcc -shared -o \
  libvector.so addvec.o multvec.o
```

Dynamic vector library

Dynamic Linking at Load-Time

```
$ gcc -shared -o \
  libvector.so addvec.c multvec.c
```



Dynamic Linking at Run-Time

dll.c

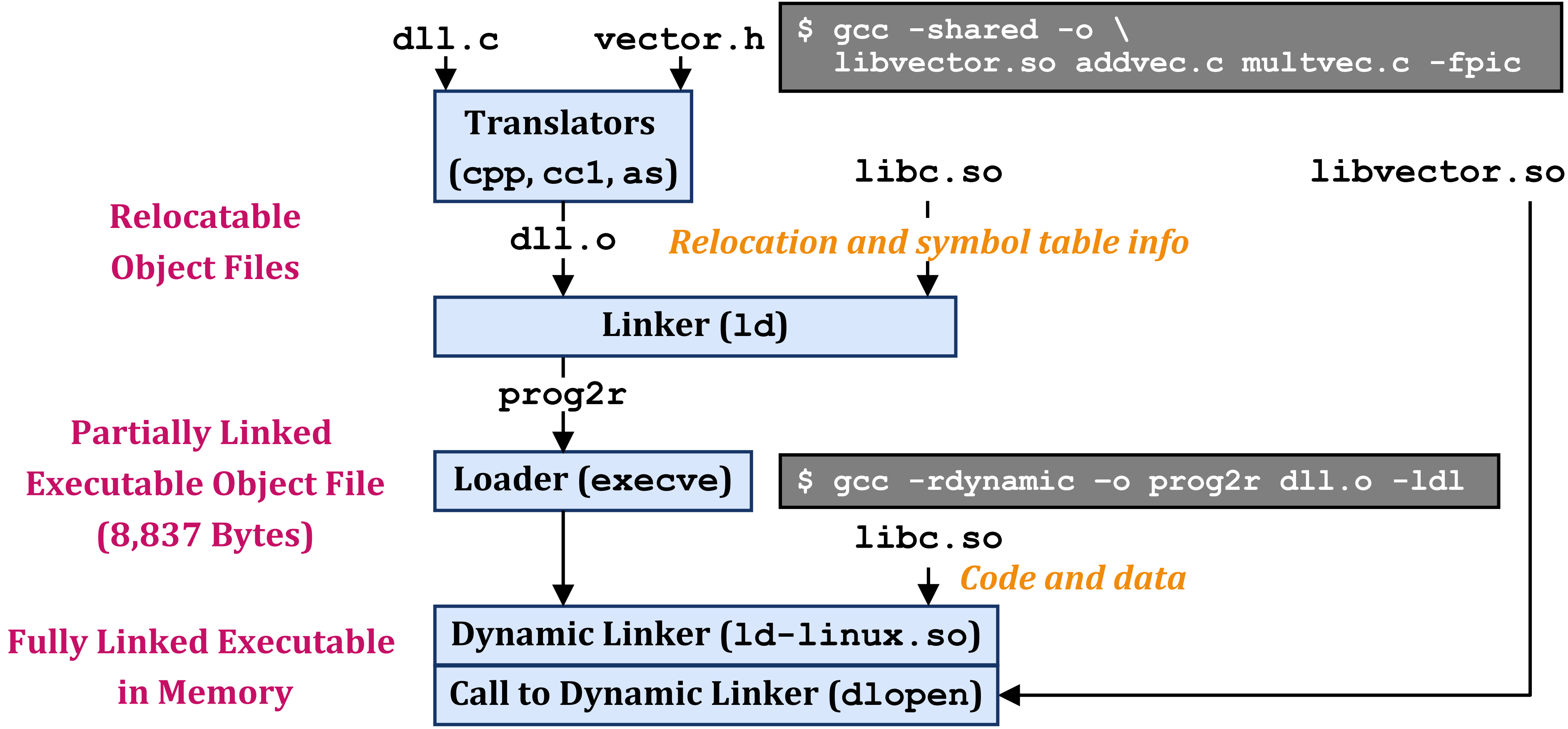
```
#include <stdio.h>
#include <dlfcn.h>

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

int main(){
  void* handle;
  void (*addvec)(int *, int *, int *, int);
  char* error;

  /* dynamically load the shared lib that contains addvec() */
  handle = dlopen("./libvector.so", RTLD_LAZY);
  if(!handle){
    fprintf(stderr, "%s\n", dlerror());
    exit(1);
  }
```

Dynamic Linking at Run-Time

dll.c (Cont.)

```
  ...
  /* get a pointer to the addvec() function we just loaded */
  addvec = dlsym(handle, "addvec");
  if((error = dlerror()) != NULL) {
    fprintf(stderr, "%s\n", error);
    exit(1);
  }
  /* Now we can call addvec() just like any other function */
  addvec(x, y, z, 2);
  printf("z = [%d %d]\n", z[0], z[1]);
  /* unload the shared library */
  if (dlclose(handle) < 0) {
    fprintf(stderr, "%s\n", dlerror());
    exit(1);
  }
  return 0;
}
```

Dynamic Linking at Run-Time



Linking Summary

- Linking allows programs to be constructed from multiple object files

- Linking can happen at different times in a program's lifetime
  - Compile time (when a program is compiled)
  - Load time (when a program is loaded into memory)
  - Run time (while a program is executing)

- Understanding linking can help you avoid nasty errors and make you a better programmer

Lecture Agenda: Linking

- Linking
  - Motivation
  - What It Does
  - How It Works
  - Dynamic Linking

- Case Study: Library Interpositioning

Case Study: Library Interpositioning

- Powerful linking technique that allows programmers to intercept calls to arbitrary functions

- Interpositioning can occur at:
  - Compile time: when the source code is compiled
  - Link time: when the relocatable object files are statically linked to form an executable object file
  - Load/run time: when an executable object file is loaded into memory, dynamically linked, and then executed

Some Interpositioning Applications

- Security
  - Confinement (sandboxing)
    - Interpose calls to `libc` functions
  - Behind the scenes encryption
    - Automatically encrypt otherwise unencrypted network connections

- Debugging
  - In 2014, two Facebook engineers debugged a treacherous 1-year old bug in their iPhone app using interpositioning
  - Code in the SPDY networking stack was writing to the wrong location
  - Solved by intercepting calls to Posix write functions (`write`, `writev`, `pwrite`)

Source: Facebook engineering blog post at
https://code.facebook.com/posts/313033472212144/debugging-file-corruption-on-ios/

Some Interpositioning Applications (Cont.)

- Monitoring and profiling
  - Count number of calls to functions
  - Characterize call sites and arguments to functions
  - `malloc` tracing
    - Detecting memory leaks
    - Generating address traces

- Error checking
  - C Programming Lab used customized versions of `malloc/free` to do careful error checking

Example program

- Goal: trace the addresses and sizes of the allocated and freed blocks
  - w/o breaking the program
  - w/o modifying the source code
- Three solutions: interpose on the library `malloc` and `free` functions at
  - Compile time
  - Link time
  - Load/run time

int.c

```
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>

int main(int argc, char** argv){
    int i;
    for (i = 1; i < argc; i++) {
        void* p = malloc(atoi(argv[i]));
        free(p);
    }
    return 0;
}
```

Compile-Time Interpositioning

mymalloc.c

```
#ifdef COMPILETIME
#include <stdio.h>
#include <malloc.h>

/* malloc wrapper function */
void* mymalloc(size_t size){
    void* ptr = malloc(size);
    printf("malloc(%d)=%p\n", (int)size, ptr);
    return ptr;
}

/* free wrapper function */
void myfree(void* ptr){
    free(ptr);
    printf("free(%p)\n", ptr);
}
#endif
```

Compile-Time Interpositioning

```
#define malloc(size) mymalloc(size)                          mymalloc.h
#define free(ptr) myfree(ptr)

void* mymalloc(size_t size);
void myfree(void* ptr);
```

```
$ make intc
gcc -Wall -DCOMPILETIME -c mymalloc.c
gcc -Wall -I. -o intc int.c mymalloc.o
$ make runc
./intc 10 100 1000
malloc(10)=0x1ba7010
free(0x1ba7010)
malloc(100)=0x1ba7030
free(0x1ba7030)
malloc(1000)=0x1ba70a0
free(0x1ba70a0)
```

Search for <malloc.h> leads to /usr/include/malloc.h

Search for <malloc.h> leads to

Link-Time Interpositioning

mymalloc.c

```
#ifdef LINKTIME
#include <stdio.h>

void* __real_malloc(size_t size);
void __real_free(void* ptr);

/* malloc wrapper function */
void* __wrap_malloc(size_t size){
    void* ptr = __real_malloc(size); /* Call libc malloc */
    printf("malloc(%d) = %p\n", (int) size, ptr);
    return ptr;
}
/* free wrapper function */
void __wrap_free(void* ptr){
    __real_free(ptr); /* Call libc free */
    printf("free(%p)\n", ptr);
}
#endif
```

Link-Time Interpositioning

```
$ make intl
gcc -Wall -DLINKTIME -c mymalloc.c
gcc -Wall -c int.c
gcc -Wall -Wl,--wrap,malloc -Wl,--wrap,free -o intl int.o mymalloc.o
$ make runl
./intl 10 100 1000
malloc(10) = 0x91a010
free(0x91a010)
...
```

Search for `<malloc.h>` leads to `/usr/include/malloc.h`

- The **`-Wl`** flag passes argument to linker, replacing each comma with a space
- The **`--wrap,malloc`** argument instructs linker to resolve references in a special way:
  - References to malloc should be resolved as **`__wrap_malloc`**
  - References to **`__real_malloc`** should be resolved as **`malloc`**

Load/Run-Time Interpositioning

mymalloc.c

```
#ifdef RUNTIME
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

/* malloc wrapper function */
void* malloc(size_t size){
  void* (*mallocp)(size_t size);
  char* error;
  mallocp = dlsym(RTLD_NEXT, "malloc"); /* Get addr of libc malloc */
  if((error = dlerror()) != NULL){
    fputs(error, stderr);
    exit(1);
  }
  char* ptr = mallocp(size); /* Call libc malloc */
  printf("malloc(%d) = %p\n", (int) size, ptr);
  return ptr;
}
```

Observe no ‘`#include <malloc.h>`’

Load/Run-Time Interpositioning

```
/* free wrapper function */                              mymalloc.c (Cont.)
void free(void* ptr)
{
  void (*freep)(void*) = NULL;
  char* error;
  if(!ptr)
    return;
  freep = dlsym(RTLD_NEXT, "free"); /* Get address of libc free */
  if((error = dlerror()) != NULL) {
    fputs(error, stderr);
    exit(1);
  }
  freep(ptr); /* Call libc free */
  printf("free(%p)\n", ptr);
}
#endif
```

Load/Run-Time Interpositioning

```
$ make intr
gcc -Wall -DRUNTIME -shared -fpic -o mymalloc.so mymalloc.c -ldl
gcc -Wall -o intr int.c
$ make runr
(LD_PRELOAD="./mymalloc.so" ./intr 10 100 1000)
malloc(10) = 0x91a010
free(0x91a010)
...
```

Search for `<malloc.h>` leads to `/usr/include/malloc.h`

- The **`LD_PRELOAD`** environment variable tells the dynamic linker to resolve unresolved refs (e.g., to **`malloc`**) by looking in **`mymalloc.so`** first
- Type into (some) shells as:

```
(setenv LD_PRELOAD "./mymalloc.so"; ./intr 10 100 1000)
```

Interpositioning Summary

- Compile time
  - Apparent calls to **`malloc/free`** get macro-expanded into calls to **`mymalloc/myfree`**
  - Simple approach that must have access to source & recompile

- Link time
  - Use linker trick to have special name resolutions
    - **`malloc`** → **`__wrap_malloc`**
    - **`__real_malloc`** → **`malloc`**

- Load/Run time
  - Implement custom version of **`malloc/free`** that use dynamic linking to load library **`malloc/free`** under different names
  - Can use with any dynamically linked binary

```
(setenv LD_PRELOAD "./mymalloc.so"; gcc -c int.c)
```

Linking Summary

- Usually: just happens, no big deal
- Sometimes: strange errors
  - Bad symbol resolution
  - Ordering dependence of linked `.o`, `.a`, and `.so` files
- For power users: interpositioning to trace programs w/ or w/o source

[CSED211] Introduction to Computer Software Systems

Lecture 12: Linking

Prof. Jisung Park

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY 1986 포항공과대학교

CAOS COMPUTER ARCHITECTURE & OPERATING SYSTEMS LABORATORY

2023.11.13