

COMP201

Computer Systems & Programming

Lecture #33 – Linking



**KOÇ
UNIVERSITY**

Aykut Erdem // Koç University // Fall 2020

Good news, everyone!

- I've finalized your midterm exam grades
- Extra office hour today between 19:00-20:00
- The unofficial end-term course feedback form is available.
- The official course feedback form is also available on KU Mobil.



Recap

- What is optimization?
- GCC Optimization
- Limitations of GCC Optimization
- Caching revisited

Plan for Today

- Linking
- Case study: Library interpositioning

Disclaimer: Slides for this lecture were borrowed from
—Randal E. Bryant and David R. O'Hallaron's CMU 15-213 class

Lecture Plan

- Linking
- Case study: Library interpositioning

Example C Program

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

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

int main()
{
    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;
}
```

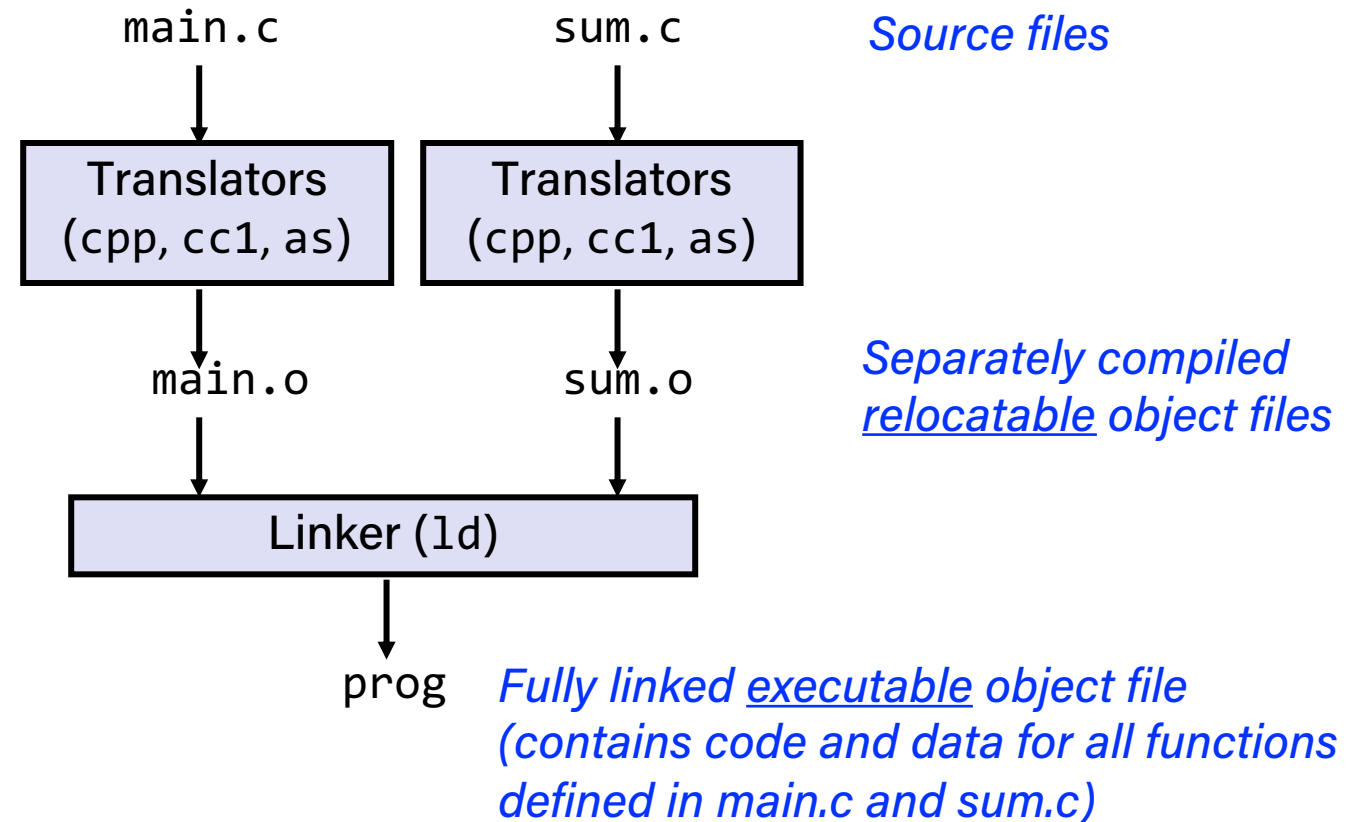
sum.c

Static Linking

Programs are translated and linked using a *compiler driver*:

```
linux> gcc -Og -o prog main.c sum.c
```

```
linux> ./prog
```



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 then relink.
 - No need to recompile other source files.
- Space: Libraries
 - Common functions can be aggregated into a single file...
 - Yet executable files and running memory images contain only code for the functions they actually use.

What Do Linkers Do?

Step 1: Symbol resolution

- Programs define and reference *symbols* (global variables and functions):
 - `void swap() {...} /* define symbol swap */`
 - `swap(); /* reference symbol swap */`
 - `int *xp = &x; /* define symbol xp, reference x */`
- Symbol definitions are stored in object file (by assembler) in *symbol table*.
 - Symbol table is an array of `structs`
 - Each entry includes name, size, and location of symbol.
- During symbol resolution step, the linker associates each symbol reference with exactly one symbol definition.

What Do Linkers Do? (cont)

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.

Let's look at these two steps in more detail....

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)**

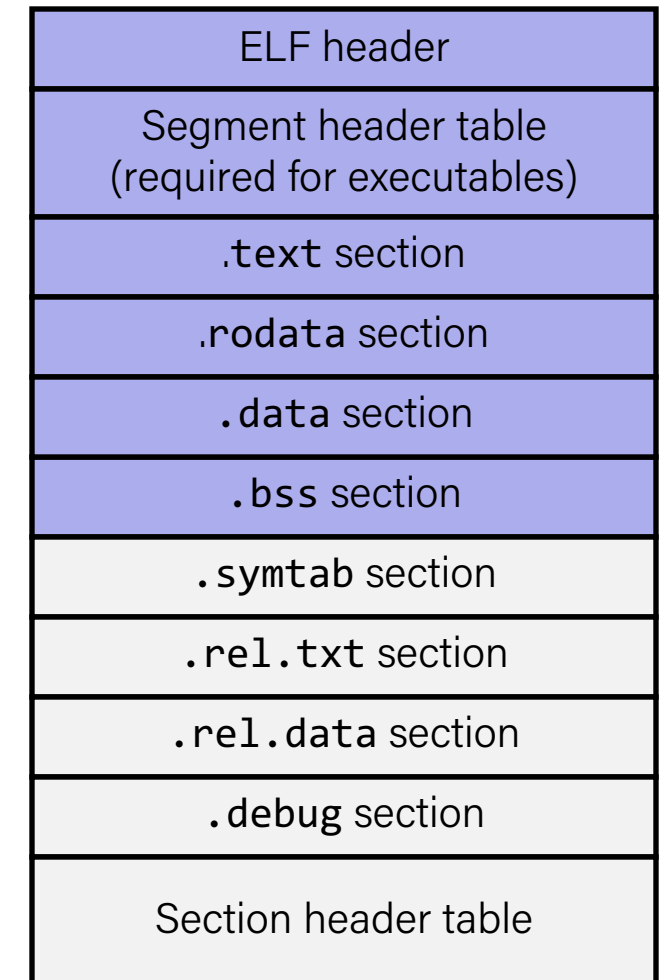
- Special type of relocatable object file that can be loaded into memory and linked dynamically, at either load time or run-time.
- Called *Dynamic Link Libraries* (DLLs) by Windows

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

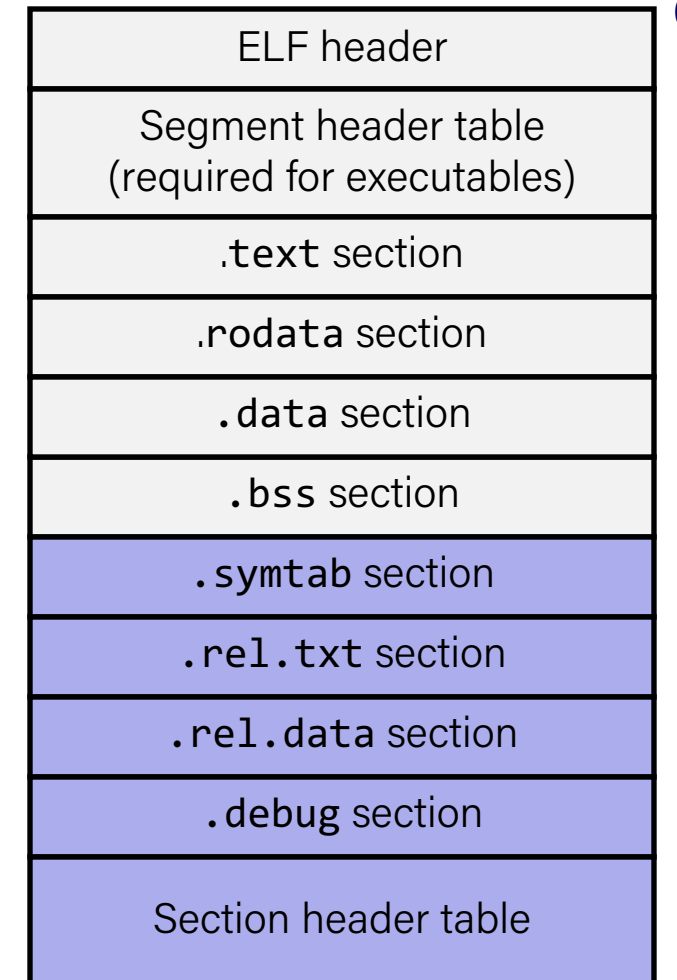
ELF Object File Format

- **Elf header**
 - Word size, byte ordering, file type (.o, exec, .so), machine type, etc.
- **Segment header table**
 - Page size, virtual addresses memory segments (sections), segment sizes.
- **.text section**
 - Code
- **.rodata section**
 - Read only data: jump tables, ...
- **.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 Object File Format (cont.)

- **.symtab section**
 - Symbol table
 - Procedure and static variable names
 - Section names and locations
- **.rel.text section**
 - Relocation info for .text section
 - Addresses of instructions that will need to be modified in the executable
 - Instructions for modifying.
- **.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)
- **Section header table**
 - Offsets and sizes of each section



Linker Symbols

- **Global symbols**

- 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 that are referenced by module m but defined by some other module.

- **Local symbols**

- Symbols that are 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

...that's defined here

Referencing
a global...

```
int sum(int *a, int n);  
  
int array[2] = {1, 2};  
  
int main()  
{  
    int val = sum(array, 2);  
    return val;  
}
```

main.c

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

Linker knows
nothing of `i` or `s`

...that's defined here

Local Symbols

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

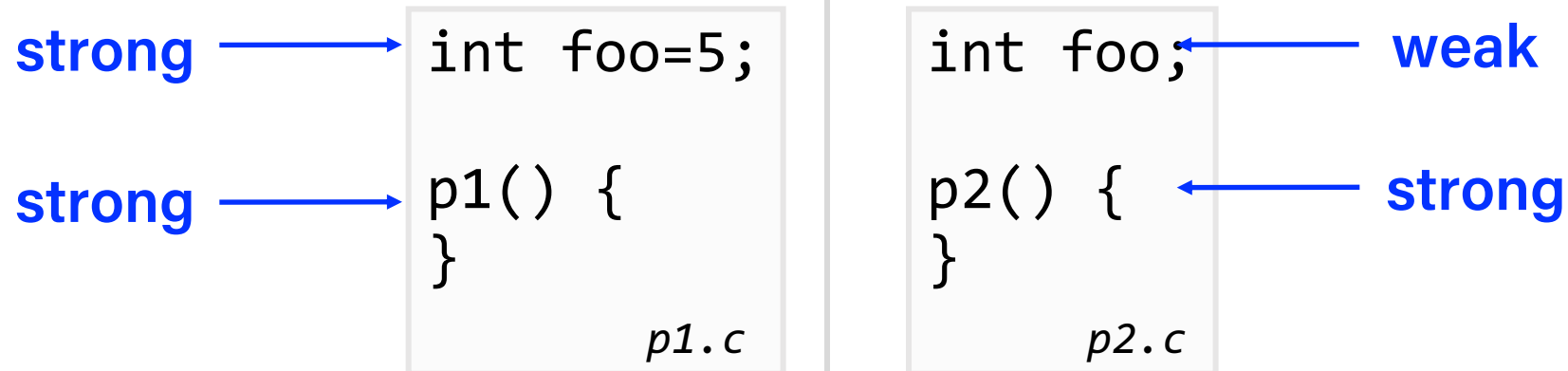
```
int f()
{
    static int x = 0;
    return x;
}

int g()
{
    static int x = 1;
    return x;
}
```

- Compiler allocates space in `.data` for each definition of `x`
- Creates local symbols in the symbol table with unique names, e.g., `x.1` and `x.2`.

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: Given a strong symbol and multiple weak symbols, choose the strong symbol**
 - References to the weak symbol 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 1

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

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

Writes to x in p2 might overwrite y!
Evil!

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

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

Writes to x in p2 will overwrite y!
Nasty!

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

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

References to x will refer to the same initialized variable.

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

Linker Puzzles 2

- Which definitions do the references `main` or `x` refer to?

```
int main()  
{}
```

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

REF(main.1) → DEF(main.1)

REF(main.2) → DEF(main.1)

```
void main()  
{}
```

```
int main = 1;  
p2() {}
```

Error! Each module defines a strong symbol `main`

```
int x;  
void main()  
{}
```

```
double x=1.0;  
int p2()  
{}
```

REF(x.1) → DEF(x.2)

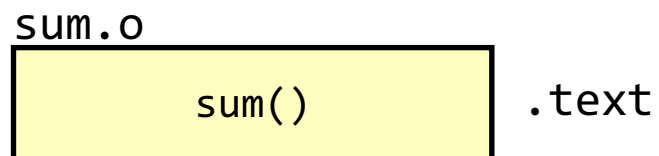
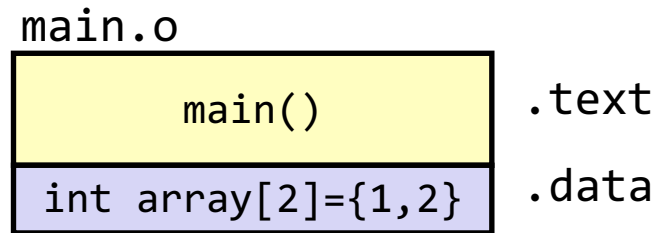
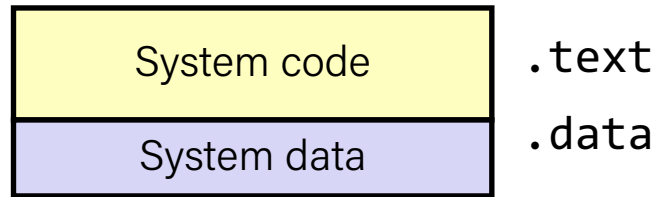
REF(x.2) → DEF(x.2)

Global Variables

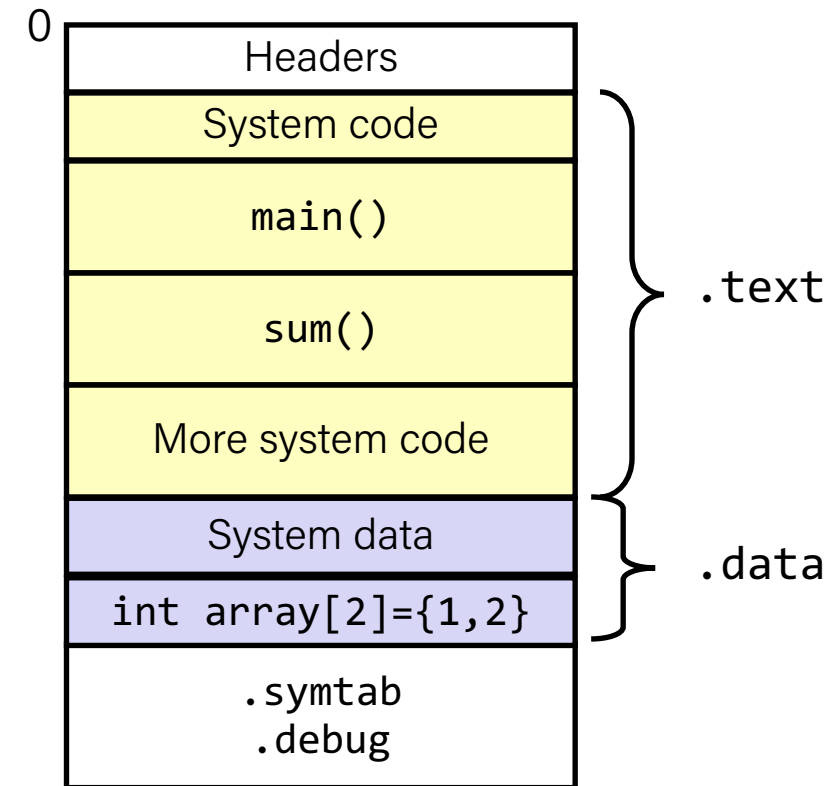
- Avoid if you can
- Otherwise
 - Use `static` if you can
 - Initialize if you define a global variable
 - Use `extern` if you reference an external global variable

Step 2: Relocation

Relocatable Object Files



Executable Object File



Relocation Entries

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

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

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
                                # Relocation entry
                                a: R_X86_64_32 array

 e:  e8 00 00 00 00            callq  13 <main+0x13>           # sum()
                                # Relocation entry
                                f: R_X86_64_PC32 sum-0x4

13:  48 83 c4 08                add    $0x8,%rsp
17:  c3                        retq

                                main.o
```

Relocated .text section

objdump -dx 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
```

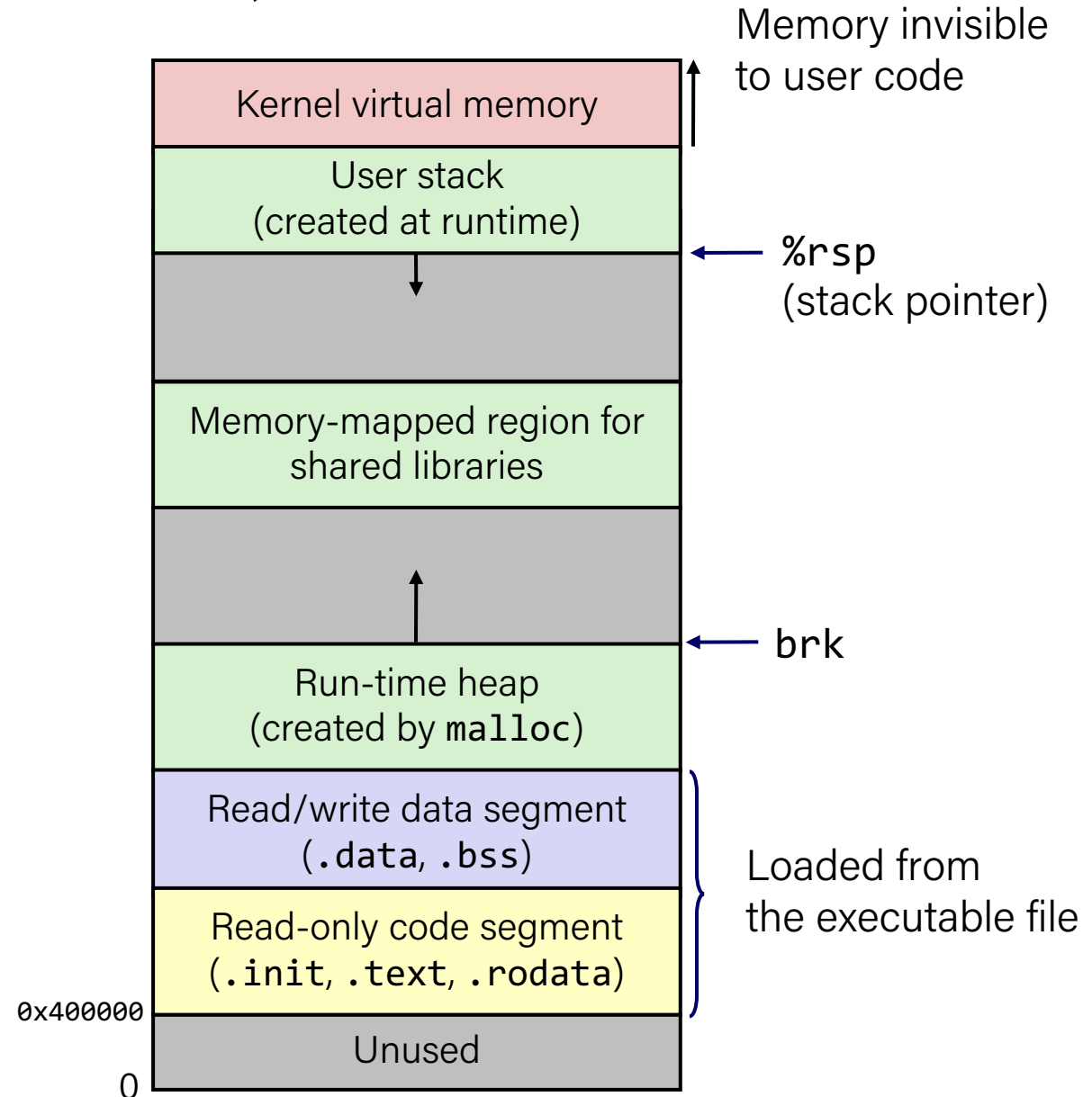
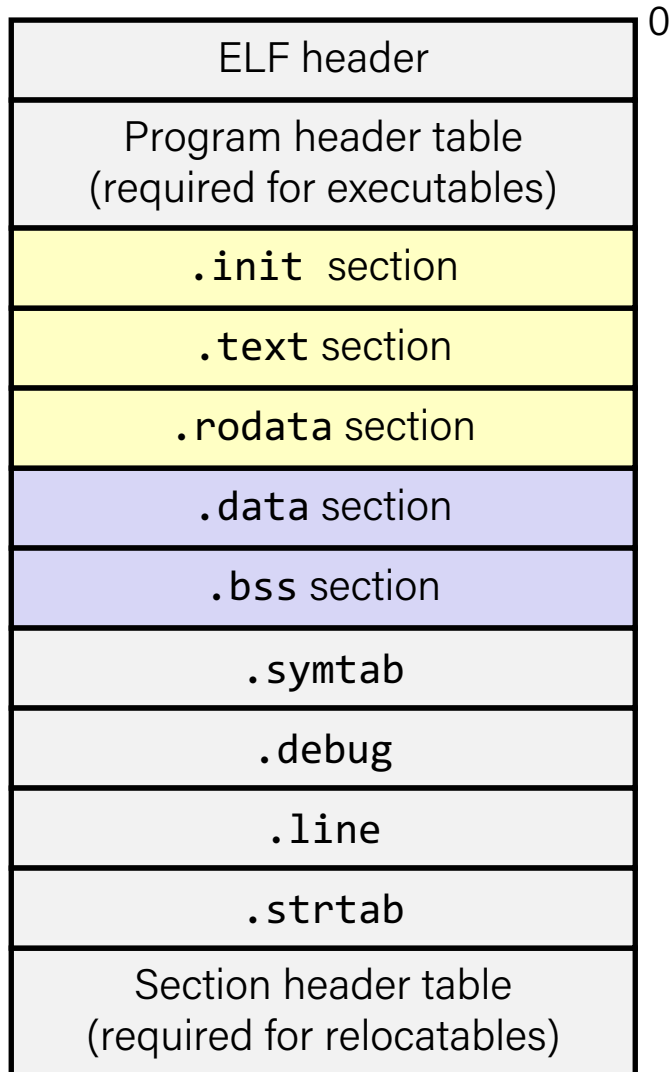
00000000004004e8 <sum>:

```
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
```

Using PC-relative addressing for sum(): $0x4004e8 = 0x4004e3 + 0x5$

Loading Executable Object Files

Executable Object File



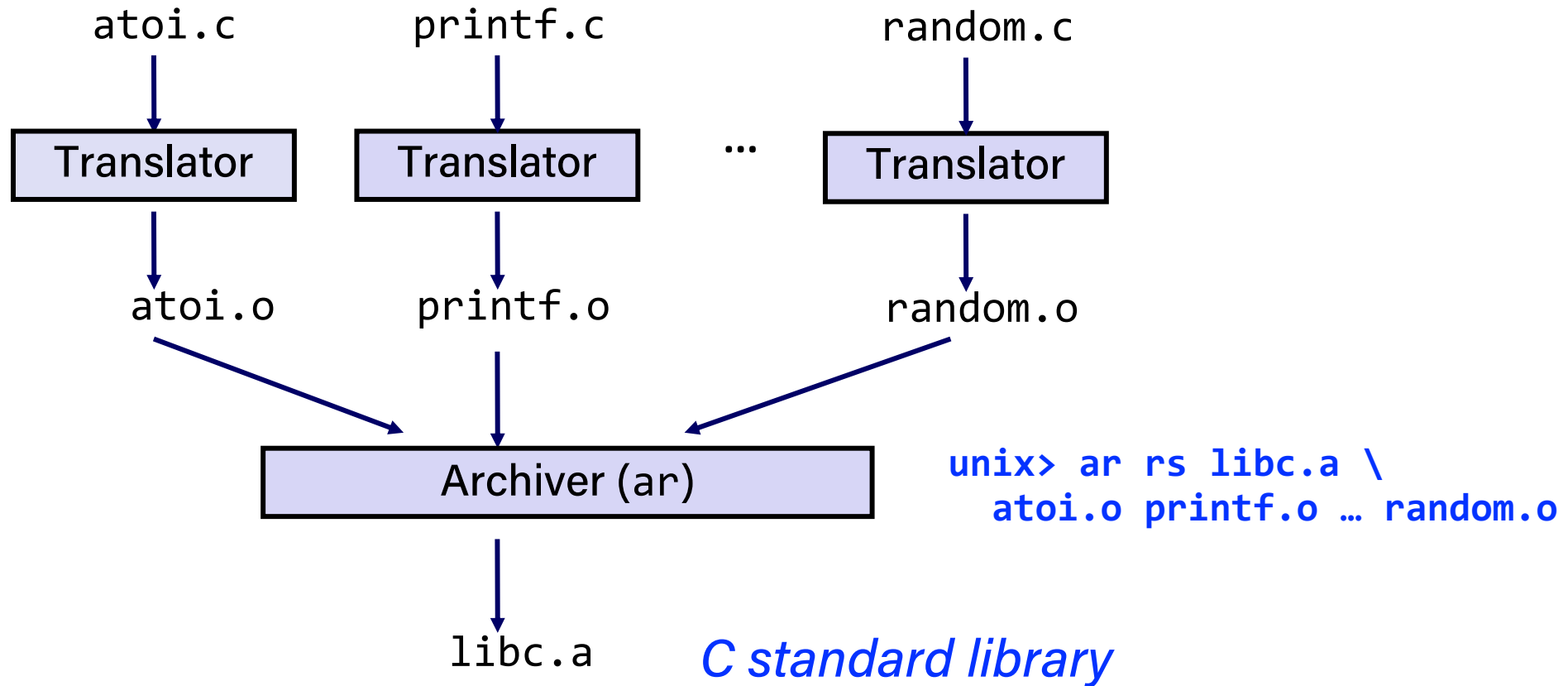
Packaging Commonly Used Functions

- 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 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 (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



- 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 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

```
#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;
}
```

main2.c

libvector.a



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

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

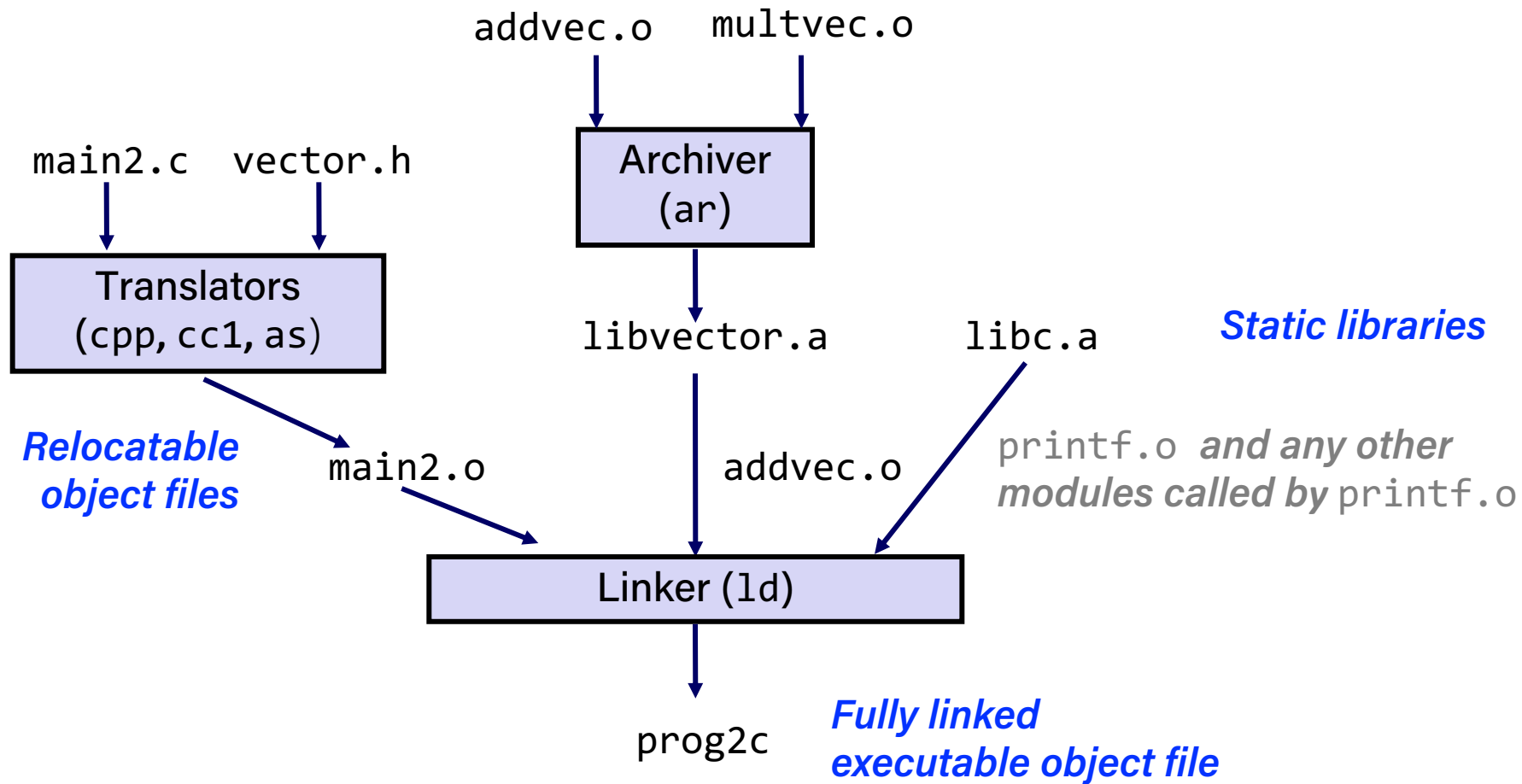
addvec.c

```
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



"c" for "compile-time"

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.

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

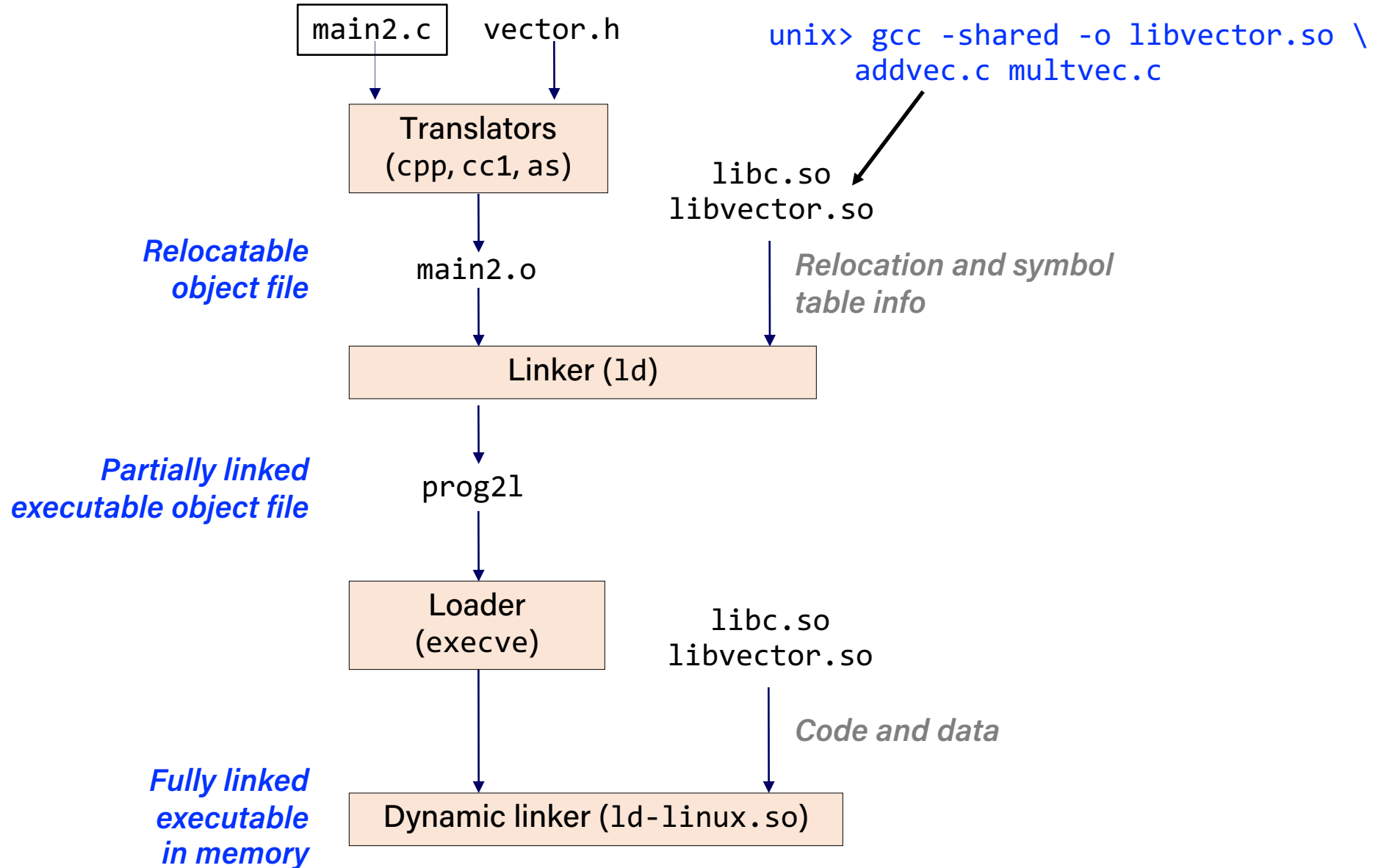
Modern Solution: Shared Libraries

- **Static libraries have the following disadvantages:**
 - Duplication in the stored executables (every function needs libc)
 - Duplication in the running executables
 - Minor bug fixes of system libraries require each application to explicitly relink
- **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, .so files

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 (`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

Dynamic Linking at Load-time



Dynamic Linking at Run-time

```
#include <stdio.h>
#include <stdlib.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 library that contains addvec() */
    handle = dlopen("./libvector.so", RTLD_LAZY);
    if (!handle) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }
}
```

dll.c

Dynamic Linking at Run-time

...

```
/* 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;
```

```
}
```

dll.c

Linking Summary

- **Linking is a technique that 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 Plan

- Linking
- Case study: Library interpositioning

Case Study: Library Interpositioning

- **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)
- Behind the scenes encryption

- **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`, `writew`, `pwrite`)

Source: Facebook engineering blog post at

<https://code.facebook.com/posts/313033472212144/debugging-file-corruption-on-ios/>

Some Interpositioning Applications

- **Monitoring and Profiling**

- Count number of calls to functions
- Characterize call sites and arguments to functions
- Malloc tracing
 - Detecting memory leaks
 - **Generating address traces**

Example program

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

int main()
{
    int *p = malloc(32);
    free(p);
    return(0);
}                                     int.c
```

- Goal: trace the addresses and sizes of the allocated and freed blocks, without breaking the program, and without modifying the source code.
- Three solutions: interpose on the **lib malloc** and **free** functions at compile time, link time, and load/run time.

Compile-time Interpositioning

```
#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
```

mymalloc.c

Compile-time Interpositioning

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

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

malloc.h

```
linux> make intc
gcc -Wall -DCOMPILETIME -c mymalloc.c
gcc -Wall -I. -o intc int.c mymalloc.o
linux> make runc
./intc
malloc(32)=0x1edc010
free(0x1edc010)
linux>
```

Link-time Interpositioning

```
#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
```

mymalloc.c

Link-time Interpositioning

```
linux> 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
linux> make runl
./intl
malloc(32) = 0x1aa0010
free(0x1aa0010)
linux>
```

- The “-Wl” flag passes argument to linker, replacing each comma with a space.
- The “--wrap,malloc” arg instructs linker to resolve references in a special way:
 - Refs to malloc should be resolved as __wrap_malloc
 - Refs to __real_malloc should be resolved as malloc

Load/Run-time Interpositioning

```
#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;
}
```

mymalloc.c

Load/Run-time Interpositioning

```
/* free wrapper function */
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
```

mymalloc.c

Load/Run-time Interpositioning

```
linux> make intr
gcc -Wall -DRUNTIME -shared -fpic -o mymalloc.so mymalloc.c -ldl
gcc -Wall -o intr int.c
linux> make runr
(LD_PRELOAD="./mymalloc.so" ./intr)
malloc(32) = 0xe60010
free(0xe60010)
linux>
```

- The `LD_PRELOAD` environment variable tells the dynamic linker to resolve unresolved refs (e.g., to `malloc`) by looking in `mymalloc.so` first.

Interpositioning Recap

- **Compile Time**

- Apparent calls to `malloc/free` get macro-expanded into calls to `mymalloc/myfree`

- **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

Recap

- Linking
- Case study: Library interpositioning

- **Next time:** Managing the heap