



# Linking (Cont)

B&O Readings: 7

CSE 361: Introduction to Systems Software

**Instructor:**

I-Ting Angelina Lee

Note: these slides were originally created in parts by Markus Püschel at Carnegie Mellon University

# Question from Last Lecture

- **User defined type names are not part of the symbol table.**

- Ex:

```
typedef struct {  
    int x;  
    double y;  
} foo_t;
```

- The name `foo_t` is only visible within the file that declared it.
- If the type needs to be shared across different C files, declare it in the header file and include the header file.
- Watch out for multiple inclusion!
- Good practice: use of `#include` guard

```
#ifndef FOO_H  
#define FOO_H  
...  
#endif //FOO_H
```

# Packaging Commonly Used Functions

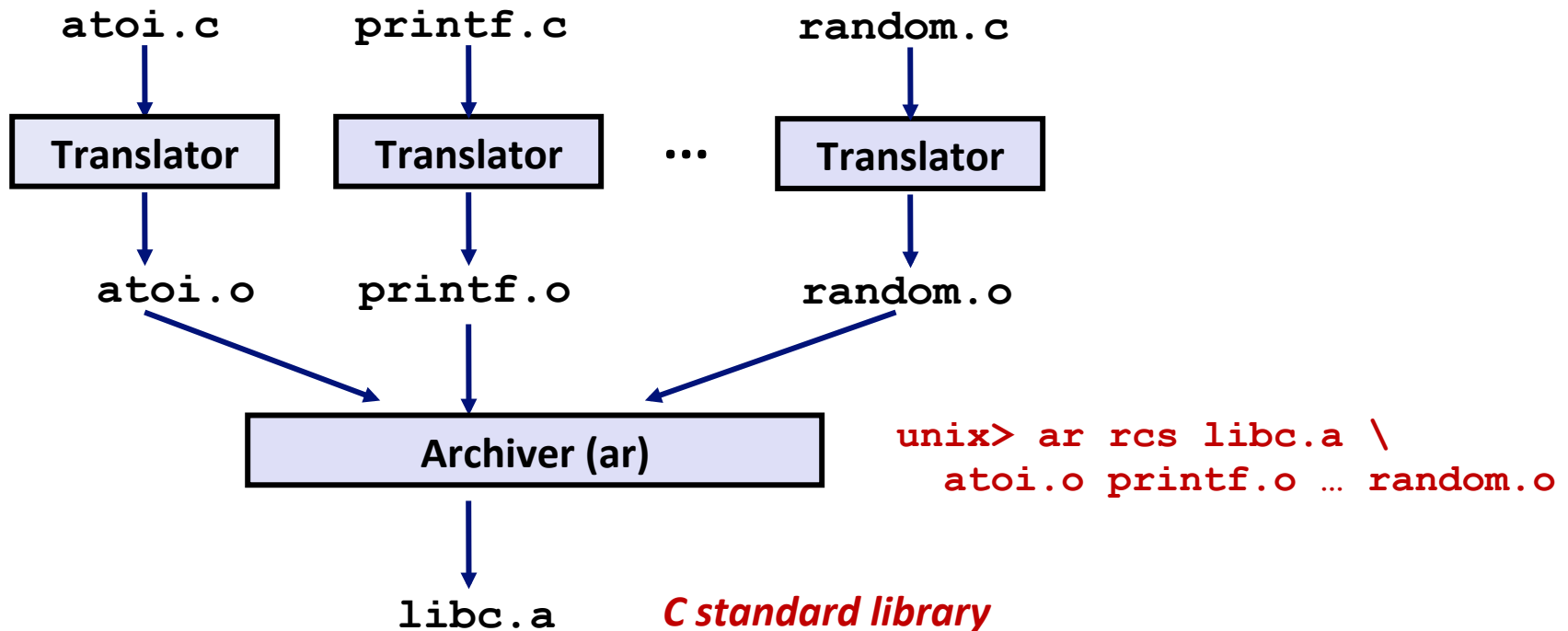
- **How to package functions commonly used by programmers?**
  - Math, I/O, memory management, string manipulation, etc.
- **Awkward, given the linker framework so far:**
  - **Option 1:** Put all functions into a single source file
    - Programmers link big object file into their programs
    - Space 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



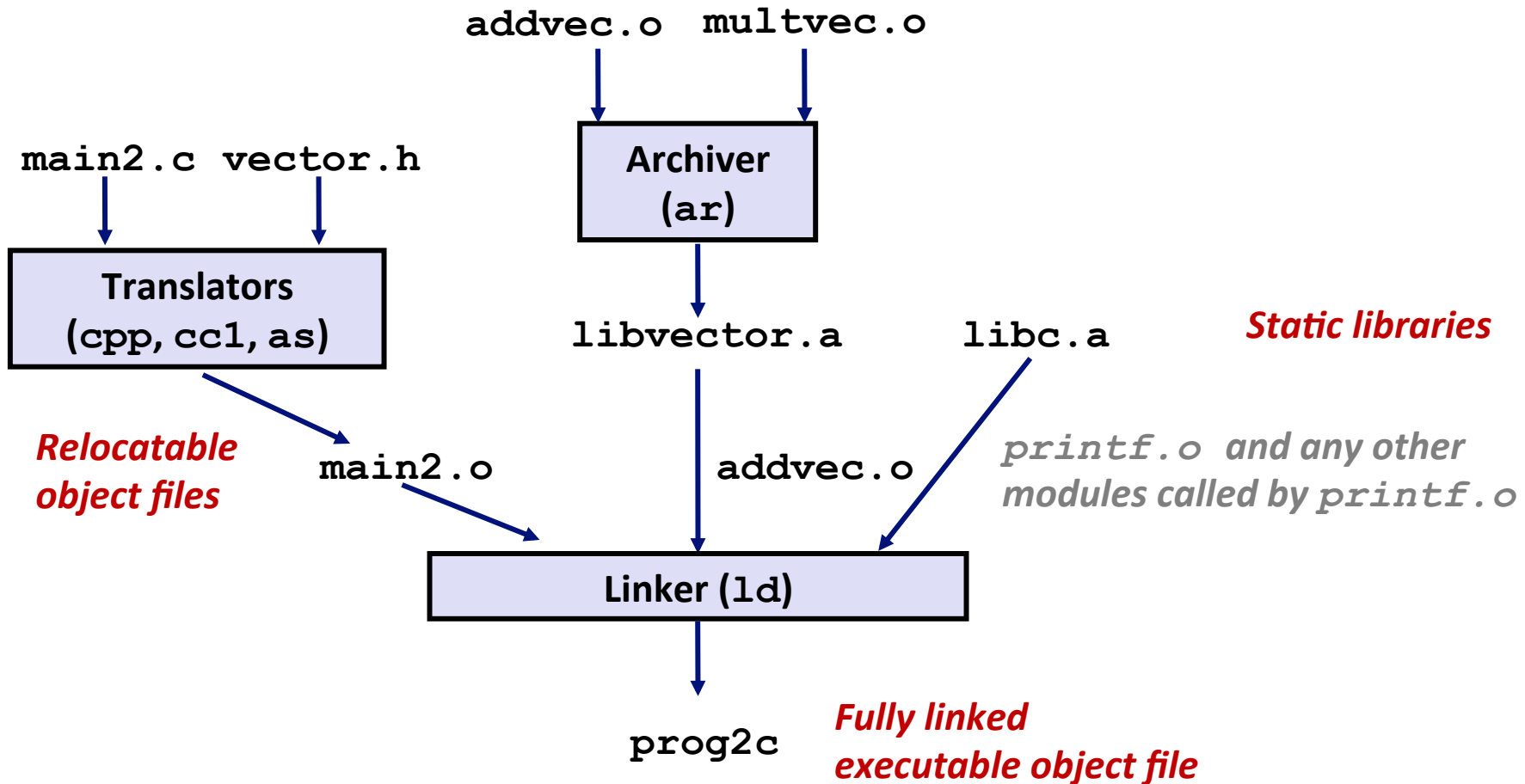
```
void addvec(int *x, int *y,
            int *z, int n) {
    int i;

    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 and maintains a set of E (object files to add to executable), D (defined symbols), and U (undefined symbols).
- 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*.
- For each new `.o`, always add to E (and their defined symbols to D).
- For each new `.a`, add only the definition that resolves the symbol in U into D.
- 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. main2.o -lvector
unix> gcc -L. -lvector main2.o
libtest.o: In function `main':
libtest.o(.text+0x4): undefined reference to `addvec'
```

# Puzzle: Linking with Static Libraries

- Say I have a file `p.o` that uses function `Foo`, defined in `libx.a`.
- Then, `libx.a` uses function `Bar`, defined in `liby.a`.
- `libx.a` uses function `P` defined in `p.o`.
- Finally, `liby.a` uses function `Quack` defined in `libx.a`.
  
- How should I link everything together so that my code compile correctly?



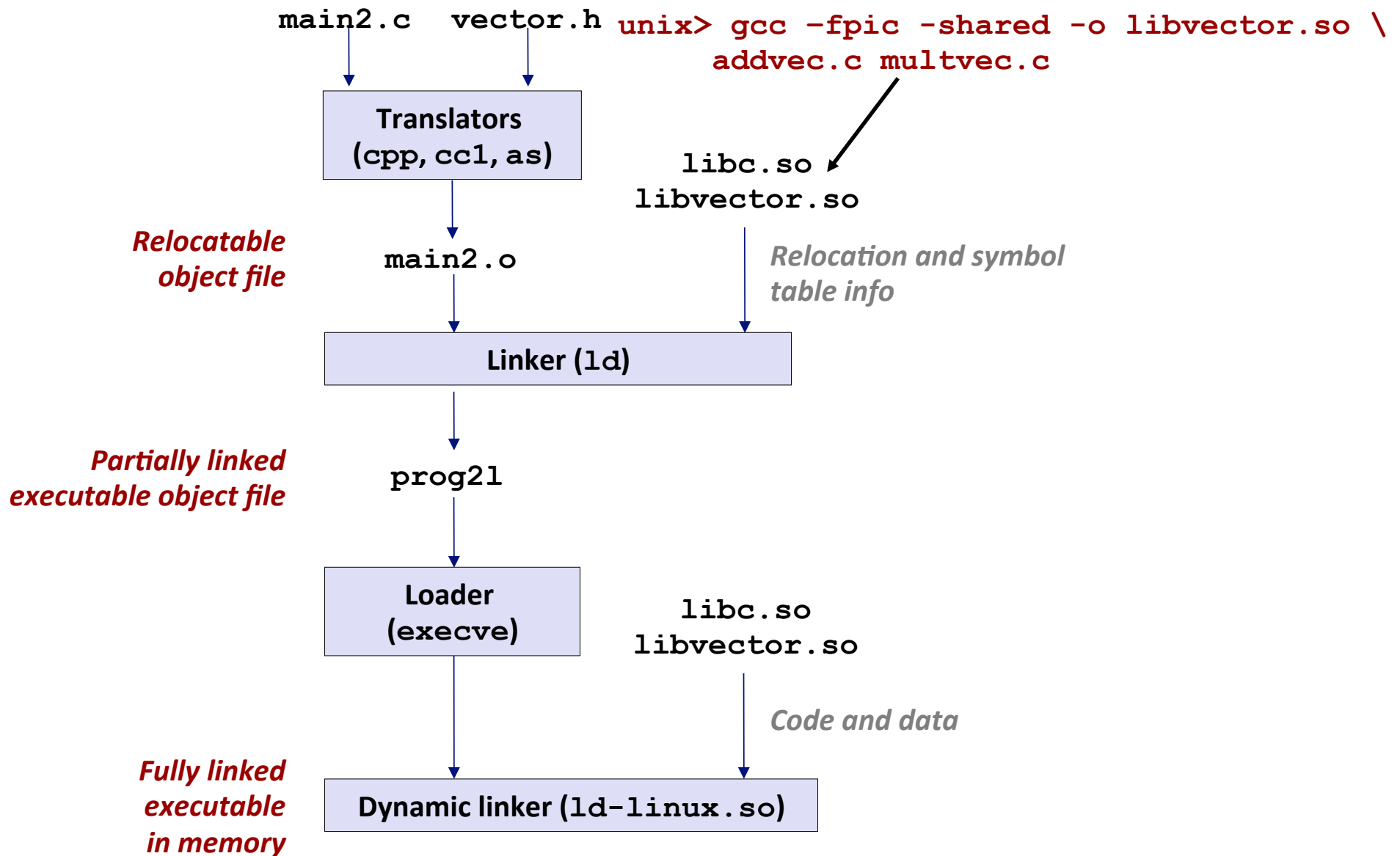
# Modern Solution: Shared Libraries

- **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

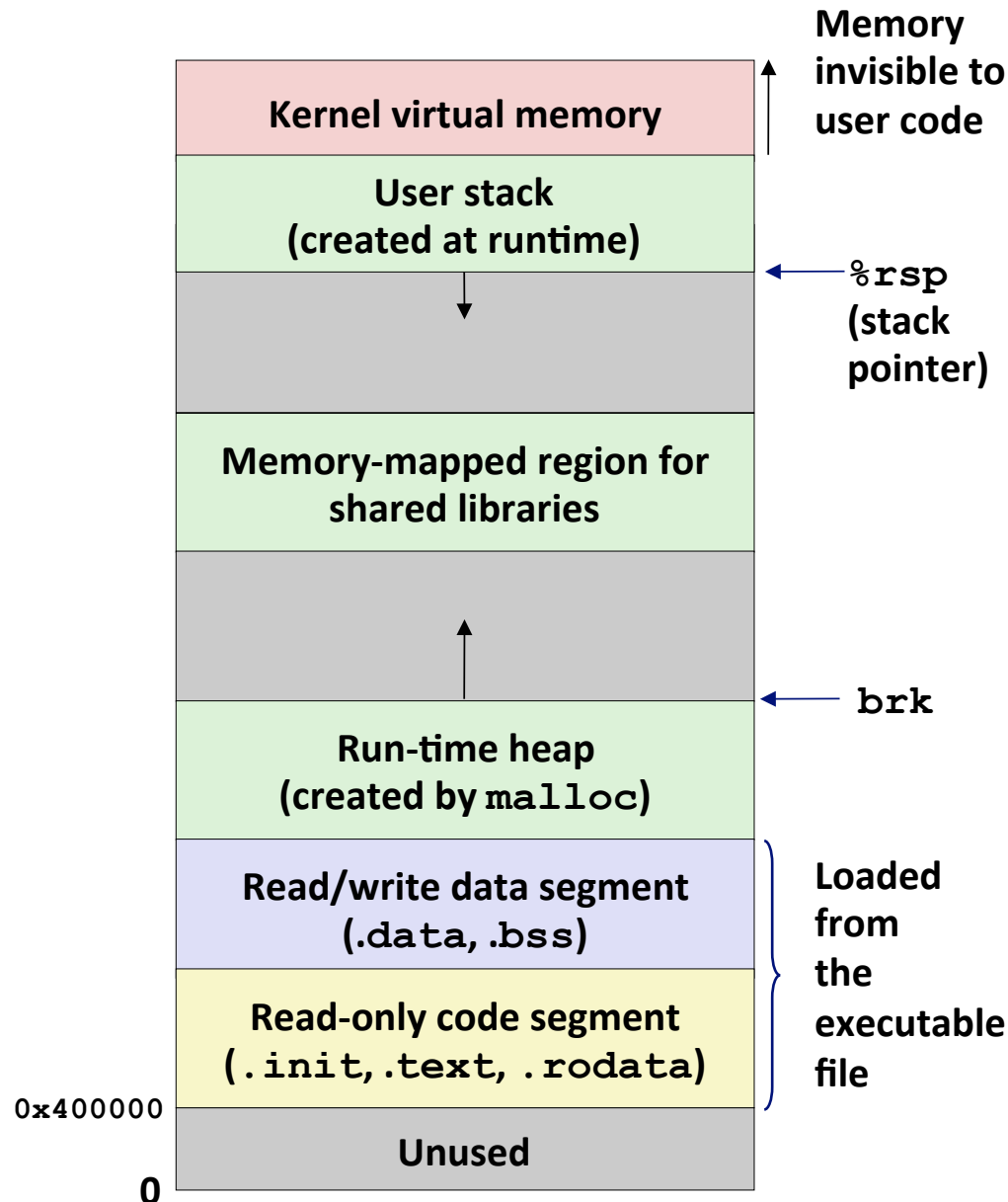
# Shared Libraries (cont.)

- **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.
    - 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



# Loading Dynamically Linked Libraries



- **Code:** only one physical copy loaded in physical memory (though they may have different virtual address).
- **Data:** each process gets its own copy.
- **Code needs to be "position independent,"** i.e., can be loaded without relocations.
- **PIC achieved with a level of indirection.**

# 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);
    }
}
```

*d11.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;
}
```

*d11.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.**

# Today

- 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, writev, 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 -DCOMPLETE TIME -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

# Library Interpositioning Puzzle

- `main.c` calls standard `malloc` and `free`
- `libmymalloc.so` implements user-defined `malloc` and `free` with the same interface as ones defined in `libc.so`
- `wrapmalloc.c` implements `__wrap_malloc` and `__wrap_free` as shown before.

```
linux> gcc -Wall -g -Wl,--wrap,malloc -Wl,--wrap,free \  
-o prog main.c wrapmalloc.c libmymalloc.so -ldl
```

```
linux> LD_LIBRARY_PATH=<path to libmalloc.so> ./prog
```

- What's the order of invocation?

