Linking (Cont.)

B&O Readings: 7

CSE 361: Introduction to Systems Software

Instructor:
I-Ting Angelina Lee

Today

- Linking
- **Static vs Dynamic libraries**
- Case study: Library interpositioning

Packaging Commonly Used Functions

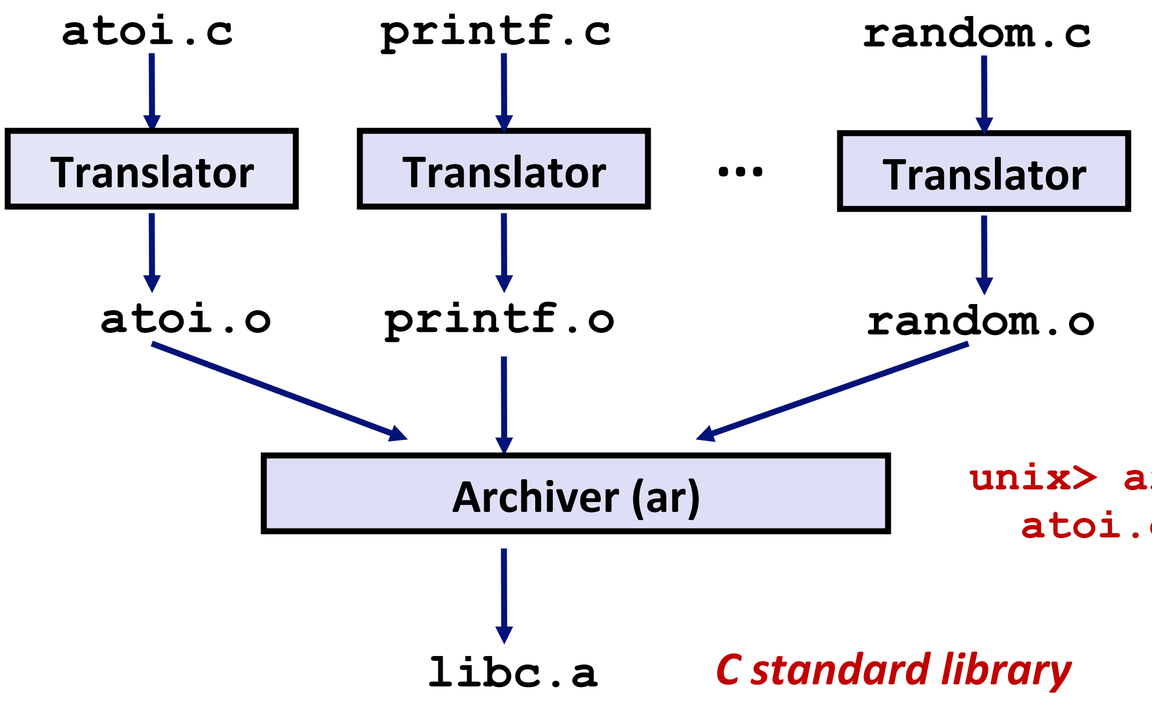
- **How to package functions commonly used by programmers?**
  - Math, I/O, memory management, string manipulation, etc.

- **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



```
unix> ar rcs libc.a \
  atoi.o printf.o … random.o
```

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

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];
}
```

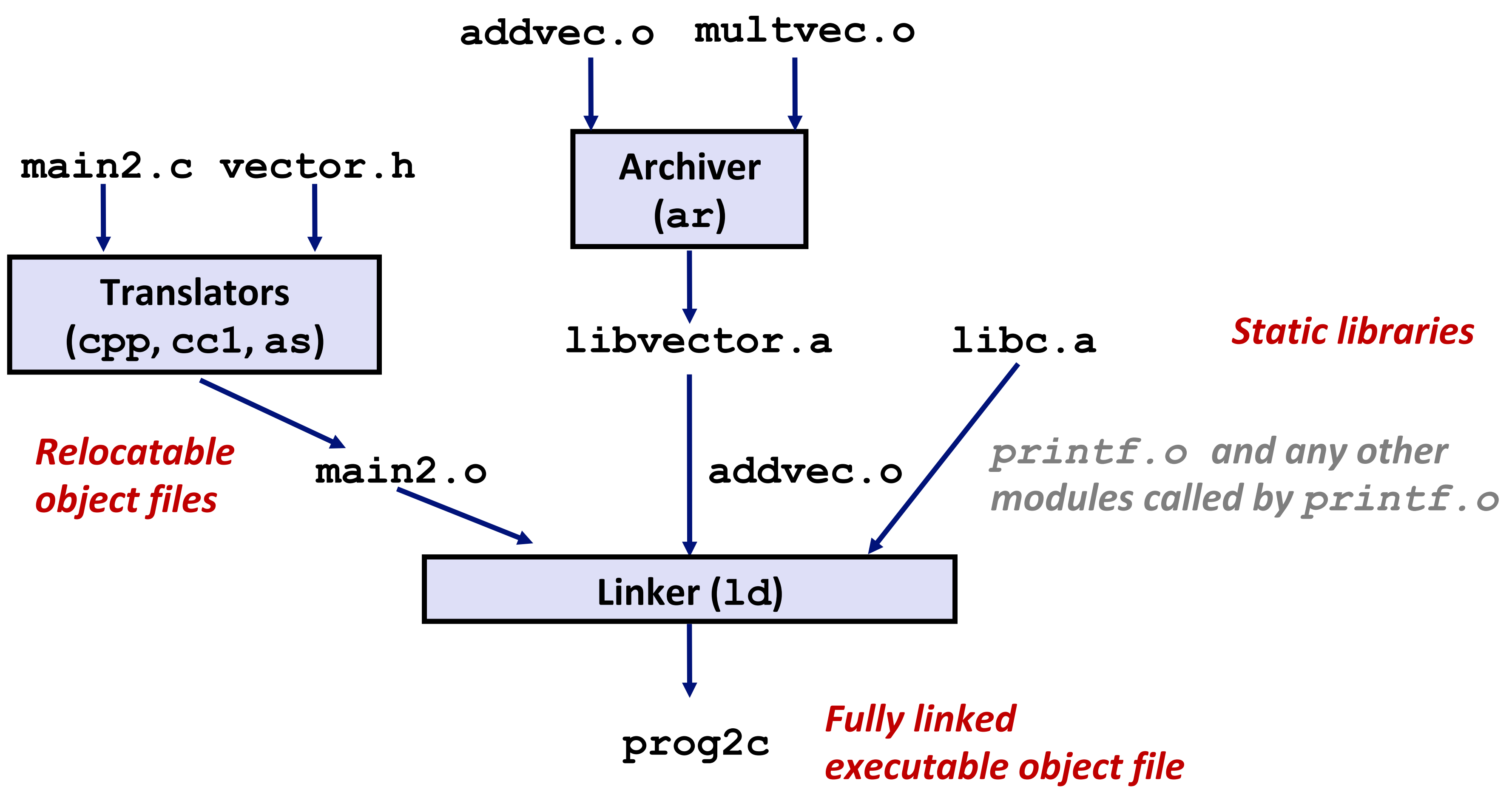
addvec.c

```
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 an object file p.o that uses function Foo, defined in libx.a.
- Then, Foo in libx.a uses function Bar, defined in liby.a.
- Foo in libx.a also uses function P defined in p.o.
- Finally, Bar in liby.a uses function Quack defined in libx.a.

- How should I link everything together so that my code compile correctly?

Puzzle: Linking with Static Libraries

- Say I have an object file p.o that uses function Foo, defined in libx.a.
- Then, Foo in libx.a uses function Bar, defined in liby.a.
- Foo in libx.a also uses function P defined in p.o.
- Finally, Bar in liby.a uses function Quack defined in libx.a.

- How should I link everything together so that my code compile correctly?

```
unix> gcc  -o prog p.o libx.a liby.a libx.a
```

Needs Foo
P included
by default

Provides Foo
Needs Bar

Provides Bar
Needs Quack

Provides Quack

Modern Solution: Shared Libraries

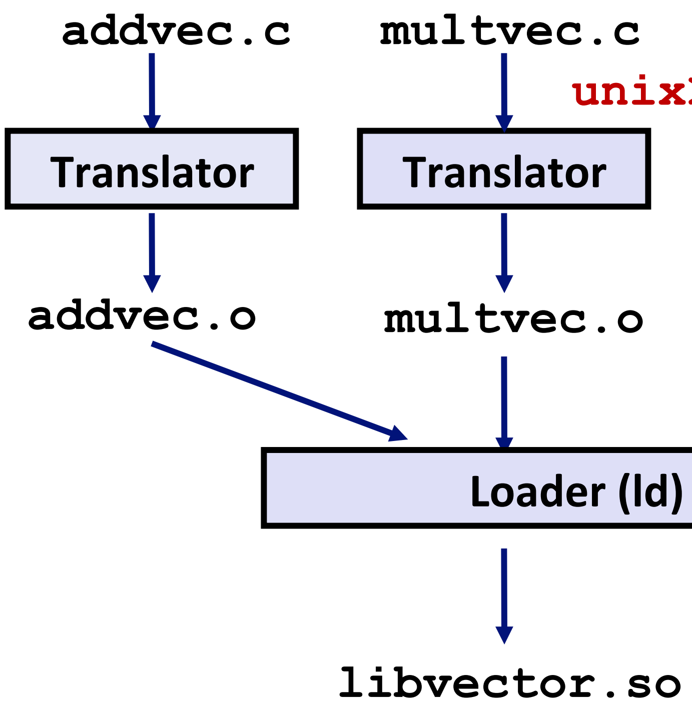
- **Static libraries have the following disadvantages:**
  - Duplication in the stored executables (every function needs libc)
  - Duplication in the running executables
  - Minor bug fixes of system libraries require each application to explicitly relink

- **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.)

- **A shared library: an object module that can be loaded at an *arbitrary memory address* and linked with a program in memory.**
- **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.
- **Shared library routines can be shared by multiple processes.**
  - More on this when we learn about virtual memory

Dynamic Library Example

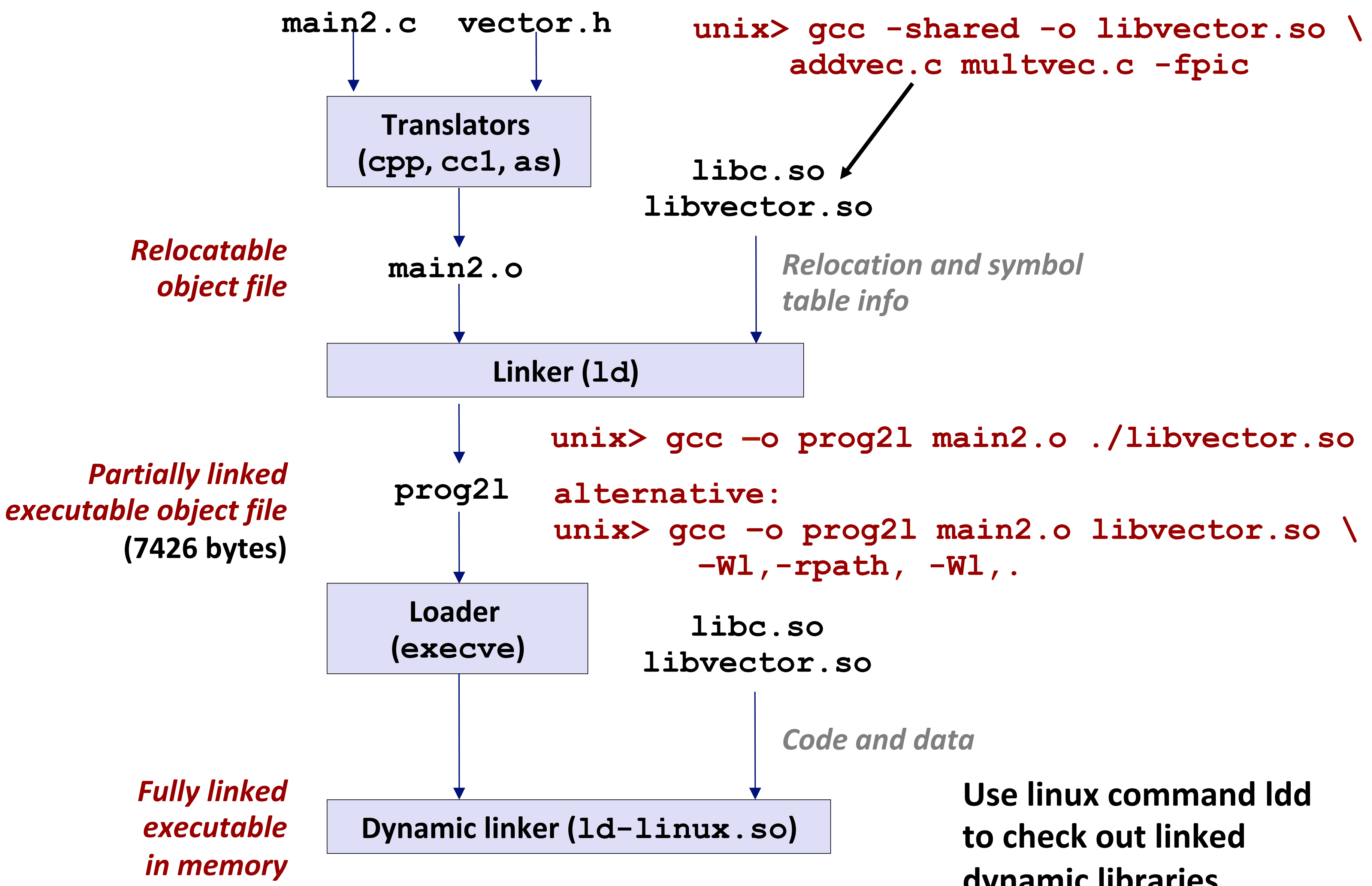


```
unix> gcc –Og –c addvec.c multvec.c -fpic
```

```
unix> gcc -shared -o libvector.so \
        addvec.o multvec.o
```

Dynamic vector library

Dynamic Linking at Load-time


main2.c
vector.h
Translators (cpp, cc1, as)
Relocatable object file
main2.o
libc.so
libvector.so
Relocation and symbol table info
Linker (ld)
Partially linked executable object file (7426 bytes)
prog21
Loader (execve)
libc.so
libvector.so
Code and data
Fully linked executable in memory
Dynamic linker (ld-linux.so)

```
unix> gcc -shared -o libvector.so \
        addvec.c multvec.c -fpic
```

```
unix> gcc –o prog21 main2.o ./libvector.so
```

```
alternative:
unix> gcc -o prog21 main2.o libvector.so \
          -Wl,-rpath, -Wl,.
```

Use linux command ldd to check out linked dynamic libraries.

How Dynamic Linker Resolves References to Dynamic Libraries

- **Problem: once the executable is loaded, you can't modify it (code segment not writeable), so we can't do relocation for references to shared libraries. However, before loading, we don't know how to resolve these references.**
- **Solution: a level of indirection!**
  - For a given executable, regardless where it's loaded at runtime, the distance between its data segment and code segment is a runtime constant.
  - The compiler generates GOT (Global Offset Table) in the data segment (writable) for these unknown references. The distance between an instruction to an entry in GOT is a runtime constant. The entries are updated once the shared libraries get loaded. The code in executable simply refers to entries in GOT.
  - For calls to functions implemented by shared libraries, additional PLT (Procedure linkage table) is used to allow for lazy update of GOT.

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.**

Today

- Linking
- Static vs Dynamic Libraries
- **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

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

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

Example program

```c
#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

```c
#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

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?

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 libmymalloc.so> ./prog
```

- What's the order of invocation?

`main.c`'s call to `malloc` gets converted into `__wrap_malloc`; `__wrap_malloc` calls `malloc` implemented by `libmymalloc.so`. Call to `free` resolves similarly.