# Linking

CSC 235 - Computer Organization

# References

- Slides adapted from CMU

# Overview

- Linking
  - Motivation
  - What it does
  - How it works
  - Dynamic linking
- Case Study Library interpositioning
- Understanding linking can help you avoid nasty errors and make you a better programmer

# Example C Program

- main.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;
}
```

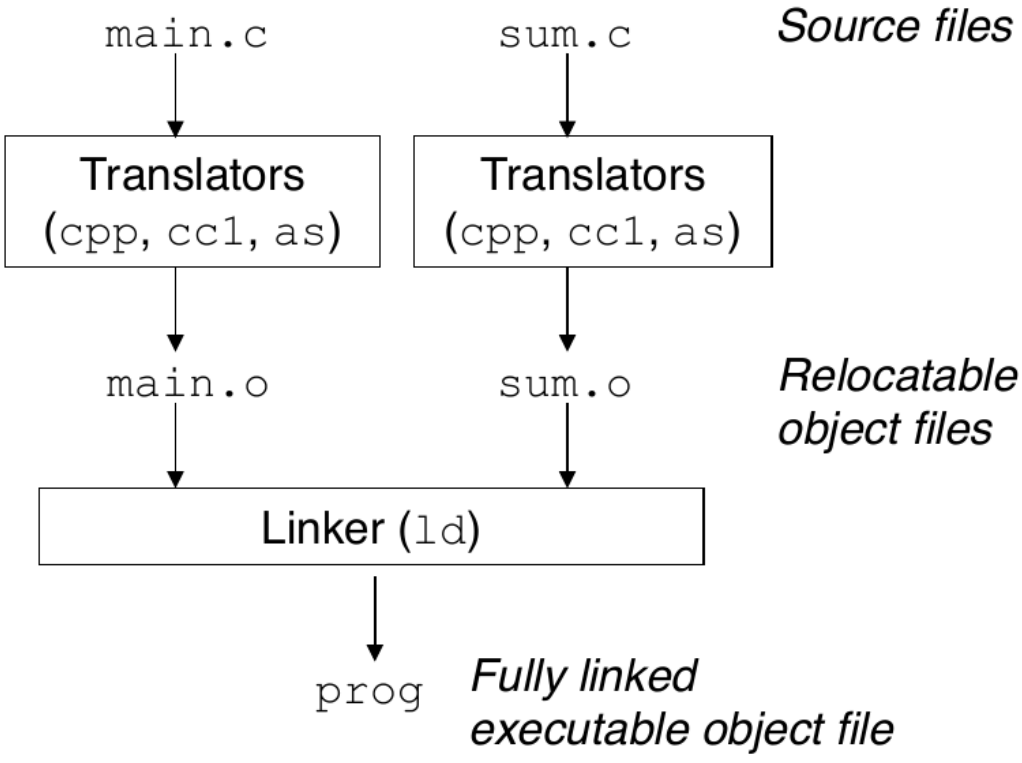
- sum.c

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

# 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)
    - For example, math library, standard C library

# Why Linkers?

- Reason 2: Efficiency
- Time: Separate compilation
  - Change one source file, compile, and then relink
  - No need to recompile other source files
  - Can compile multiple files concurrently
- Space: Libraries
  - Common functions can be aggregated into a single file
  - Option 1: Static linking
    - Executable files and running memory images contain only the library code they actually use
  - Option 2: Dynamically linking
    - Executable files contain no library code
    - During execution, 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* (global variables and functions)
  - Symbol definitions are stored in the object file (by assembler) in a *symbol table*
    - Symbol table is an array of entries
    - Each entry includes name, size, and location of symbol
  - During symbol resolution step, the linker associates each symbol reference with exactly one symbol definition

# Symbols in Example C Program

- Definitions
  - `main`
  - `sum`
  - `array`
- References
  - call to `sum`

# What Do Linkers Do?

- 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

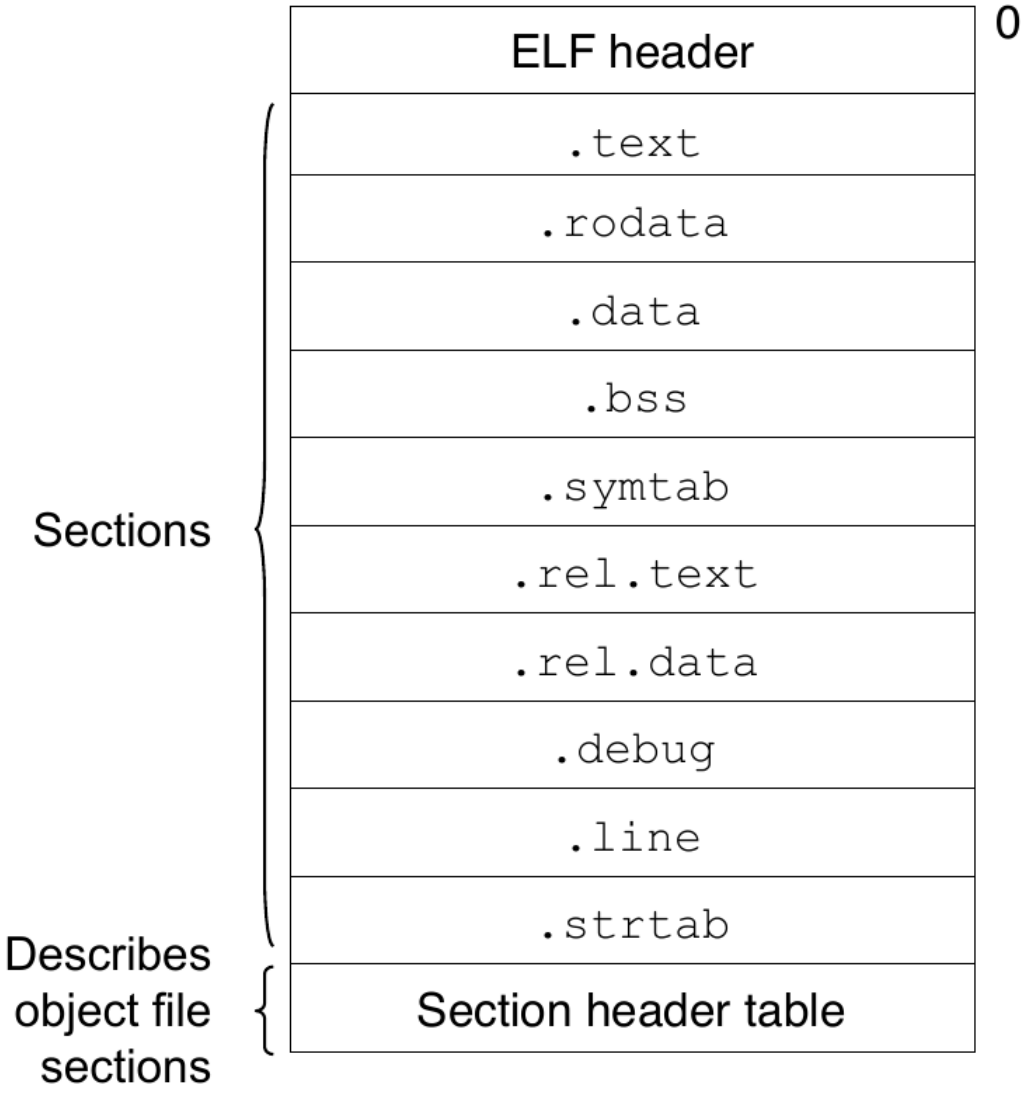
# Three Kinds of Object Files (Modules)

- 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
- 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

# 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 Object File Format

- ELF header
  - Word size, byte ordering, file type, machine type, etc.
- Segment header table
  - Page size, virtual address memory segments (sections), segment sizes
- `.text` section
  - Code
- `.rodata` section
  - Read only data: jump tables, string constants, 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 Object File Format (Continued)

- `.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
  - Relocatable info for `.data` section
  - Addresses of pointer data that will need to be modified in the merged executable
- `.debug` section
  - Info for symbolic debugging
- 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
  - For example, 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*
  - For example, C functions and global variables defined with the `static` attribute
  - Local linker symbols are not local program variables

# Symbol Identification

- Which names will be in the symbol table of `symbols.c`?

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

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

- Can find this with the command: `readelf -s symbols.o`

# Local Symbols

- Local non-static C variables versus local static C variables
  - local non-static C variables: stored on the stack
  - local static variables: stored in either `.bss`, or `.data`
- Example: each local `x` gets a unique name, for example `x.1721`

```
static in x = 15;
inf f() {
    static int x = 17;
    return x++;
}
int g() {
    static int x = 19;
    return x += 14;
}
int h() {
    return x += 27;
}
```

# How the Linker Resolves Duplicate Symbol Definitions

- Program symbols are either *strong* or *weak*
  - Strong: procedures and initialized globals
  - Weak: uninitialized globals or declared with `extern` specifier
- Examples

```
int foo = 5; // strong

p1() {       // strong
}

int foo;     // weak

p2() {       // strong
}
```

# Linker 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`

# 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 a linker error if not defined in some file

# Type Mismatch Error

- The linker does not do type checking
- `main.c`

```
long int x; // weak symbol

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

- `variable.c`

```
double x = 3.14; // global strong symbol
```

# Use of extern Example

- c1.c

```
#include "global.h"

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

- global.h

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

# Use of extern Example (Continued)

- c2.c

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

int g;

int main (int argc, char* argv[]) {
    int t = f();
    printf("Calling f yields %d\n", t);
    return 0;
}
```

# Relocation Entries

- main.c

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

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

# Relocation Entries

- 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
```

# Libraries: Packaging a Set of Functions

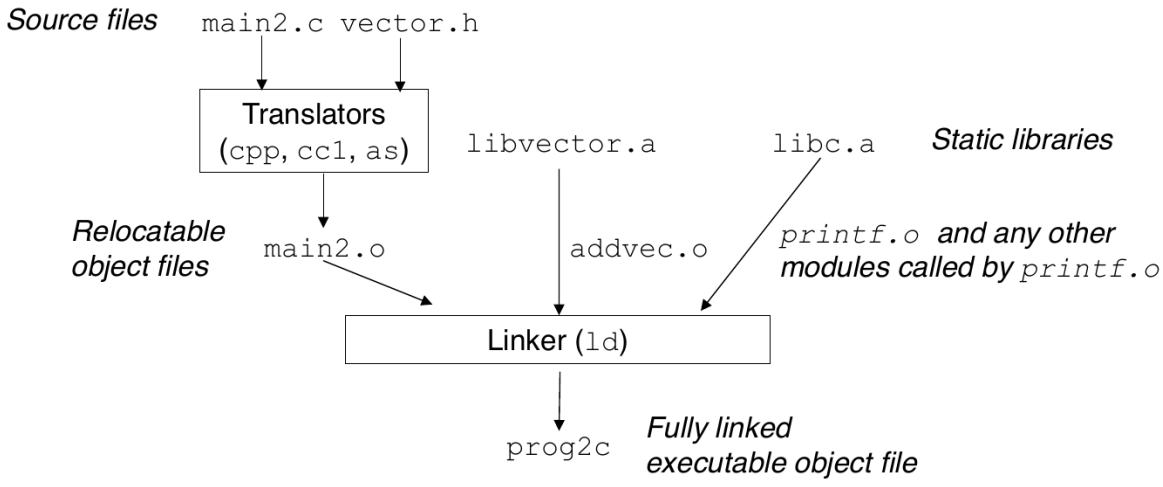
- How to package functions commonly used by programmers?
  - Math, I/O, memory management, string manipulation, etc.
- Awkward given the linker framework so far
  - Option 1: Put all functions into a single source file
    - Programmers link big object file to their programs
    - Not efficient in space and time
  - 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

# One 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, then link it into the executable

# Creating Static Libraries

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




- Creating static libraries
  - Example: `ar rs libstuff.a a.o b.o ... x.o`
- Linking with static libraries
  - Example: `gcc -static main.o -L. -lstuff`

# 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, date 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, etc.)
- The `ar` command can be used to view contents:
  - Example: `ar -t /usr/lib/libc.a`

# Using Static Libraries

- The 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 of the current unresolved references
  - As each new `.o` or `.a` file is encountered, try to resolve each unresolved reference in the list against the symbols defined in the `.o` or `.a` file
  - If any entries in the unresolved list at the end of scan, then error

# Using Static Libraries

- Problem:
  - The command line order matters
  - So, put libraries at the end of the command line

```
linux> gcc -static -o prog -L. -lstuff main.o
main.o: In function `main':
main.c:(.text+0x19): undefined reference to `thing'
collect2: error: ld returned 1 exit status
```

# Another 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 explicilty relink
- 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

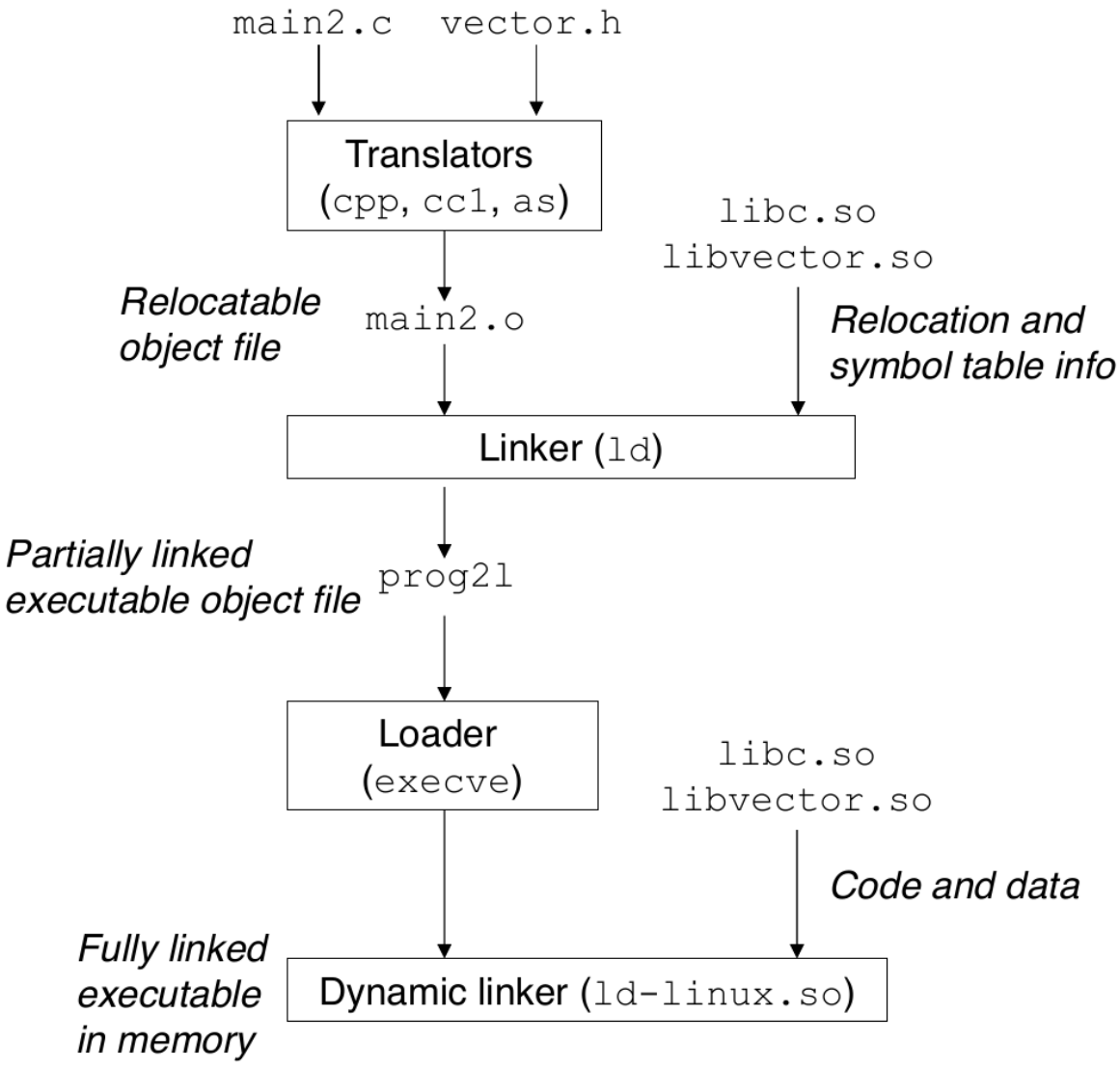
- Dynamic linking can occur when the executable is first loaded and run (load time linking)
  - Common case for Linux, handled automatically by the dynamic linker
  - Standard C library (`libc.so`) usually dynamically linked
- Dynamic linking can occur after the program has begun (run time linking)
  - In Linux this is done by calls to the `dlopen` interface
- Shared libraries routines can be shared by multiple processes
  - More on this when we learn about virtual memory

# What Dynamic Libraries Require?

- `.interp` section
  - Specifies the dynamic linker to use (for example `ld-linux.so`)
- `.dynamic` section
  - Specifies the names, etc. of the dynamic libraries to use
- Where are the libraries found?
  - Use `ldd` to find out
- Example

```
linux> ldd prog
linux-vdso.so.1 =>  (0x00007ffcf2998000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f9
/lib64/ld-linux-x86-64.so.2 (0x00007f99adcef000)
```

# 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];

...
```

# Dynamic Linking at Run Time (continued)

```
...
int main(int argc, char** argv)
{
    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);
    }
    ...
```

# Dynamic Linking at Run Time (Continued)

```
...

/* 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);
}
...
```

# Dynamic Linking at Run Time (Continued)

```
    ...
    /* 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;
}
```

# 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 (when a program is executing)
- Understanding linking can help you avoid nasty errors and make you a better programmer

# Case Study: Library Interpositioning

- Documented in Section 7.13 of the textbook
- 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
  - Example: Confinement (sandboxing)
- Debugging
  - Example: intercept POSIX functions
- Monitoring and profiling
  - Example: `malloc` tracing
- Error checking
  - Example: custom versions of `malloc`/`free` for careful error checking

# Example Program

```
#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);
}
```

# Example Program

- 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 library `malloc` and `free` functions at compile time, link time and load/run time

# Compile Time Interpositioning

- myalloc.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;
}
...
```

# Compile Time Interpositioning

- `myalloc.c` continued

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

# Compile Time Interpositioning

- `malloc.h`

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

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

- Compiling

```
linux> gcc -Wall -DCOMPILETIME -c mymalloc.c
linux> gcc -Wall -I. -o intc int.c mymalloc.o
```

# 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)
{
    /* Call libc malloc */
    void *ptr = __real_malloc(size);
    printf("malloc(%d) = %p\n", (int)size, ptr);
    return ptr;
}
...
```

# Link Time Interpositioning

```
...

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

# Link Time Interpositioning

- Compiling

```
linux> gcc -Wall -DLINKTIME -c mymalloc.c
linux> gcc -Wall -c int.c
linux> gcc -Wall -Wl,--wrap,malloc -Wl,--wrap,free
  -o intl int.o mymalloc.o
```

- The "`-Wl`" flag passes argument to the linker, replacing each comma with a space
- The "`--wrap,malloc`" arg instructs the 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

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

...
```

# Load/Run Time Interpositioning

```
/* malloc wrapper function */
void *malloc(size_t size)
{
    void *(*mallocp)(size_t size);
    char *error;

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

# Load/Run Time Interpositioning

```
/* free wrapper function */
void free(void *ptr)
{
    void (*freep)(void *) = NULL;
    char *error;
    if (!ptr) return;
    /* Get address of libc free */
    freep = dlsym(RTLD_NEXT, "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

- Compiling

```
linux> gcc -Wall -DRUNTIME -shared -fpic
       -o mymalloc.so mymalloc.c -ldl
linux> gcc -Wall -o intr int.c
```

- The `LD_PRELOAD` environment variable tells the dynamic linker to resolve unresolved references by looking in `myalloc.so` first
- Type into (some) shells as:

```
env LD_PRELOAD=./mymalloc.so ./intr 10 100 1000)
```

# Interpositioning Recap

- Compile time
  - Apparent calls to `malloc`/`free` get macro-expanded into calls to `myalloc`/‘myfree
  - Simple approach; must have access to source and recompile
- Link time
  - Use linker trick to have special name resolutions
- 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

# Linking Recap

- 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 with and without source code