# Software Security

## *Connecting the Dots*

Mohamed Sabt

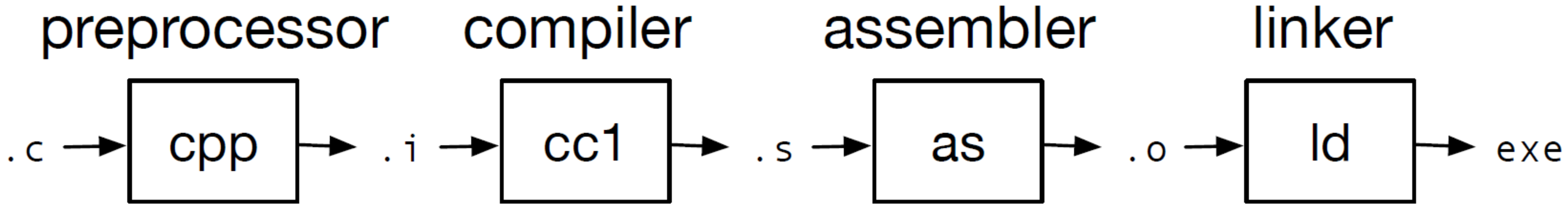
Univ Rennes, CNRS, IRISA

2022 / 2023

# Part I

## *Introduction*

# Compilation Steps



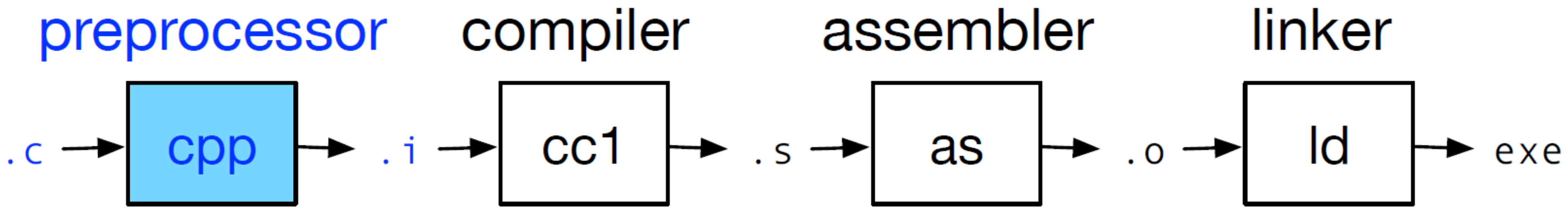

- C code first gets compiled into assembly code.
- Assembly code is then converted into machine code.

# Simple Program

💿 A simple C program: return47.c

```
#define FOURTYSEVEN 47
int main(void) {
	return FOURTYSEVEN;
}
```

# Preprocessor

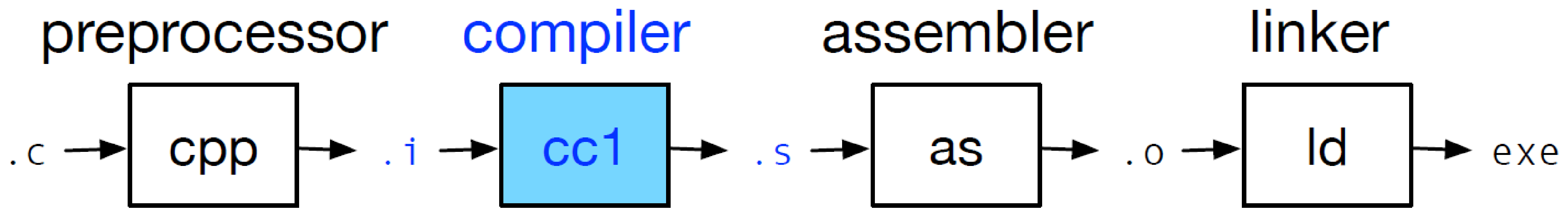



- Resolves Macros (#define)
- Add additional source code (#include)
- Handles other directives like #pragma and #if

- Example:

```
gcc  -E return47.c
int main(void) {
        return 47;
}
```

# Compiler

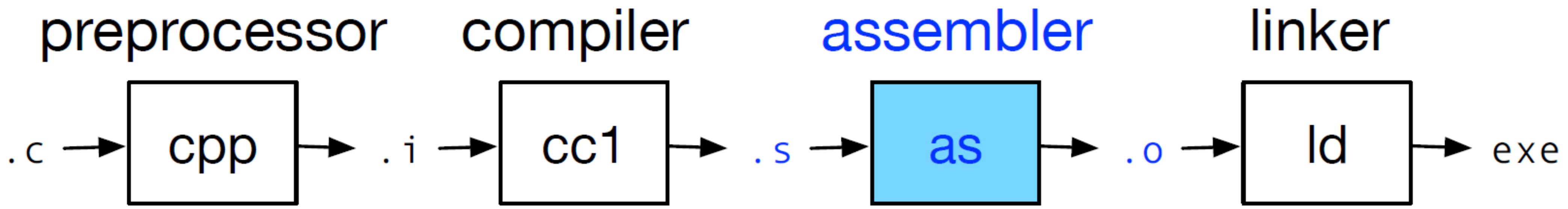



- Compilation into assembly code.

- Example:

```
gcc  -S return47.c
main:
        movl $47, %eax
        ret
```

# Assembler




- Conversion into machine code.

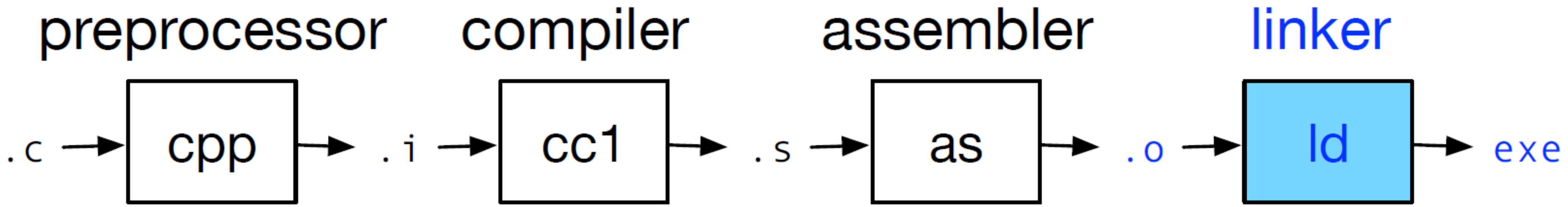
- Example:

gcc -c return47.c

0000000000000000 <main>:

```
    0:    b8 2f 00 00              mov    $0x2f,%eax
    5:    c3                       retq
```

# Linker



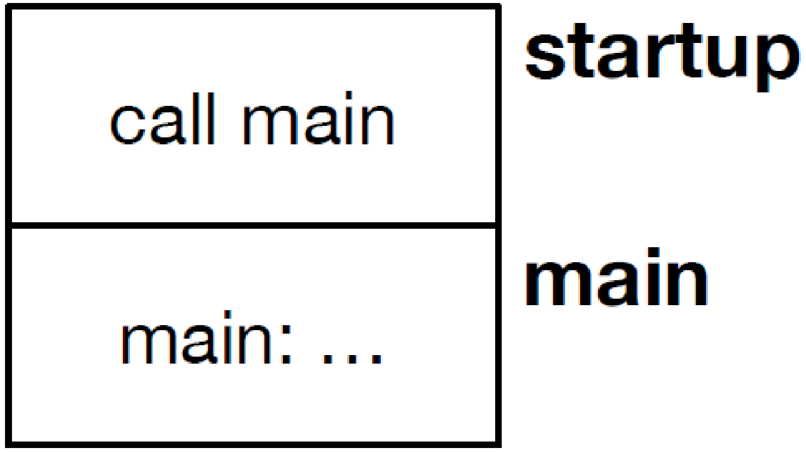

- Adds start up code.
- May combine multiple object files.

- Example:

gcc  return47.c

./a.out

echo $?



# Hello World – Code Source

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

int main(void) {
	printf("Hello world!\n");
	return EXIT_SUCCESS;
}
```

# Hello World – Assembly Code

```
.LC0:
        .string     "Hello world!"
        .text
        .globl      main
        .type       main, @function
main:
        subq        $8, %rsp
        movl        $.LC0, %edi
        call        puts
        movl        $0, %eax
        addq        $8, %rsp
        ret
```

# Hello World – Machine Code (Disassembled)

- objdump –t hello_world.o

0000000000000000 g F .text 0000000000000018 main

0000000000000000 *UND* 0000000000000000 puts

- Function “puts” is labeled as undefined (*UND*).
- Linker resolves this.

# Part II

*Static Linking (1)*

# Example C Program

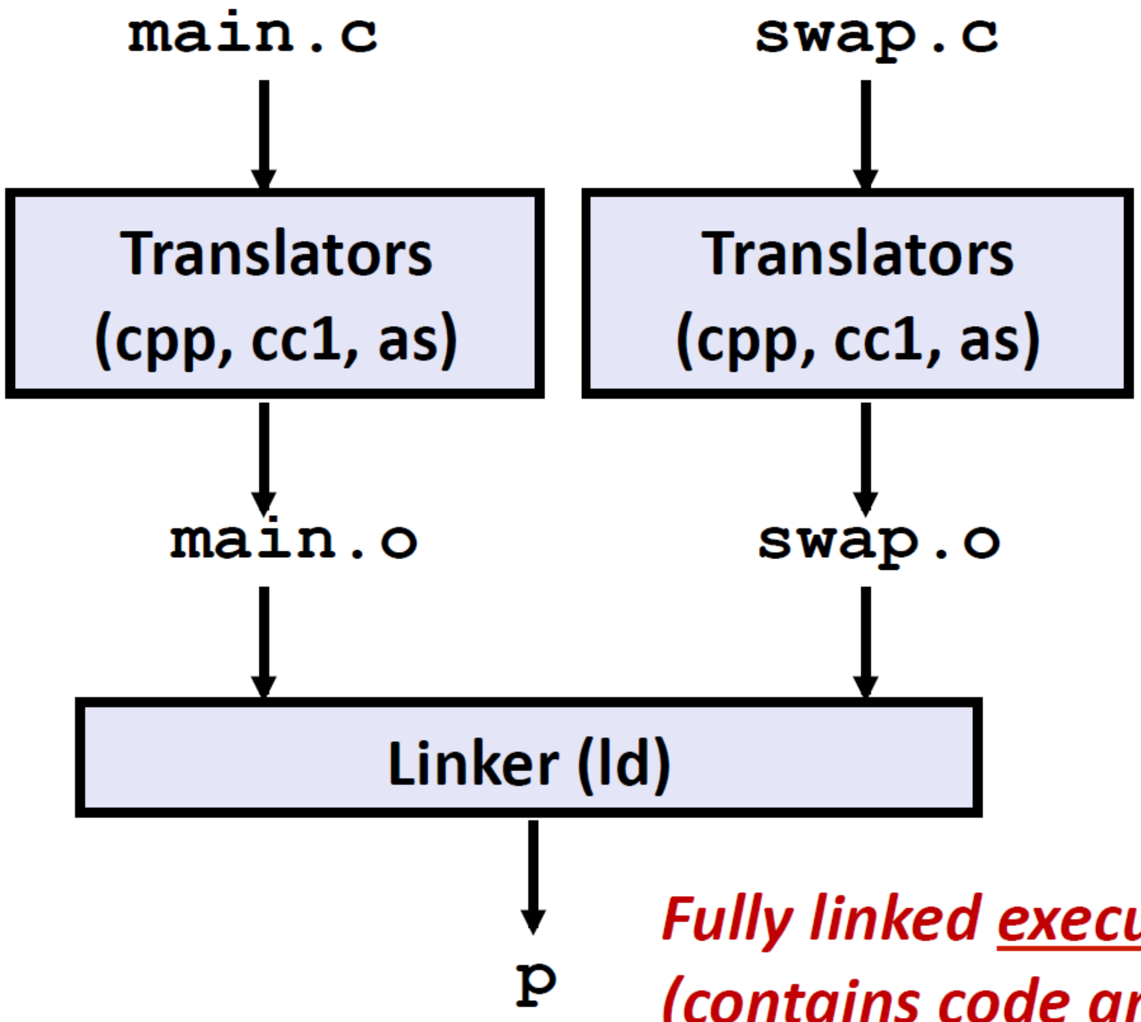
**main.c**

```
int array[2] = {1, 2};
void swap(int*, int*);
int main(void) {
	swap(array, array + 1);
	return 0;
}
```

**swap.c**

```
void swap(int * a, int * b) {
	int temp = *a;
	*a = *b;
	*b = temp;
}
```

# Static Linking




*Source files*

*Separately compiled relocatable object files*

*Fully linked executable object file (contains code and data for all functions defined in `main.c` and `swap.c`)*

# What is a Linker?

- A System Software that combines two or more separate object programs and supplies the information needed to allow references between them.

- In short, linker performs the final step to convert .obj files into executable or machine readable file (.exe).

# 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 1: 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 Are Linkers For?

**Step 1. Symbol resolution**

- Programs define and reference symbols (variables and functions)
  - void swap(int *, int*) {…}  // define symbol swap
  - swap(a, b);                 // reference symbol swap
  - int * xp = &x;              // define symbol xp, reference x

- Symbol definitions are stored (by compiler) in symbol table
  - Symbol table is an array of structs.
  - Each entry includes name, size, and location of symbol.

- Linker associates each symbol reference with exactly one symbol definition.

# Exercises

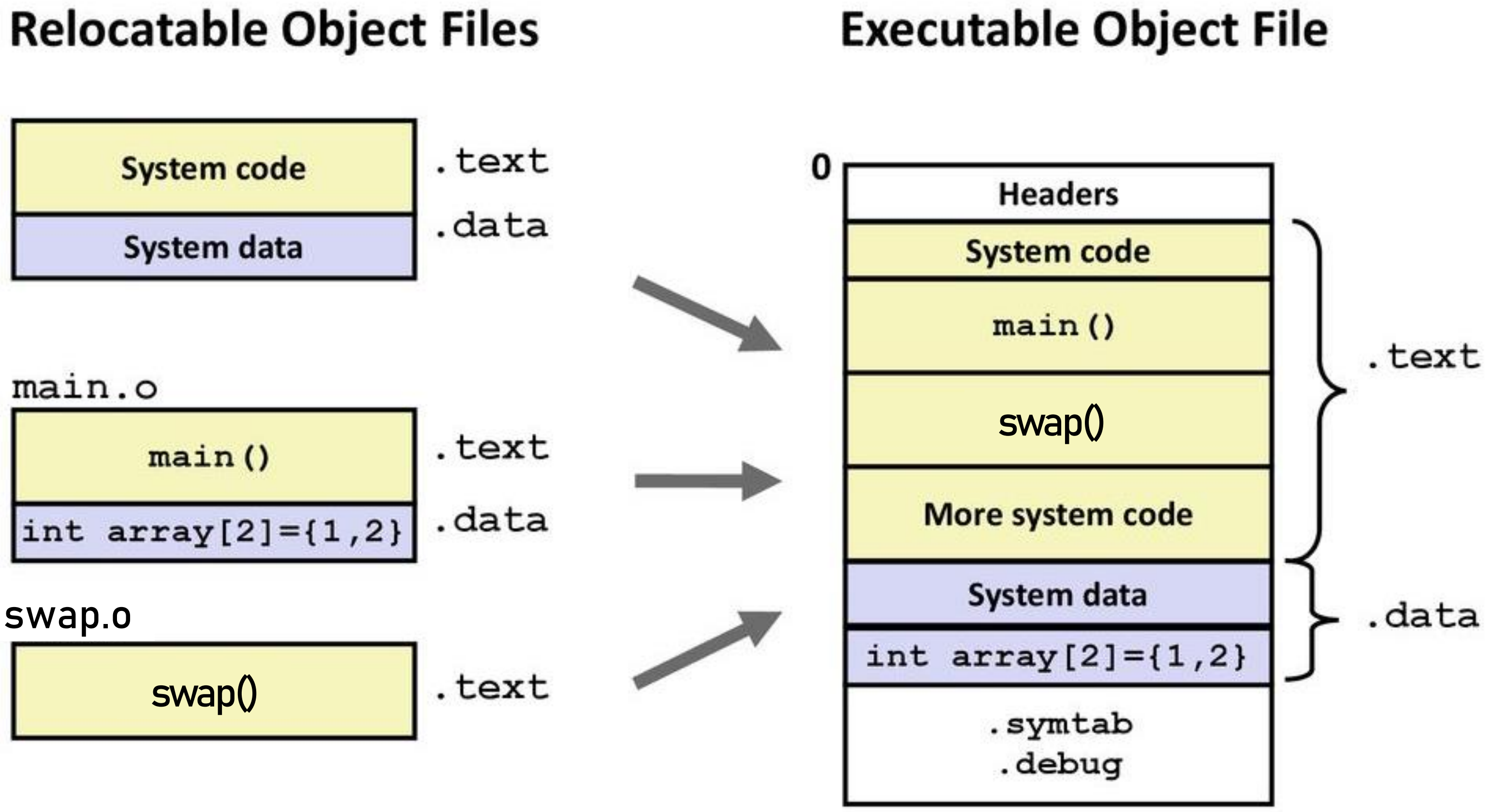
**True / False**

- The (External) Symbol Table does contain variables with automatic storage duration.

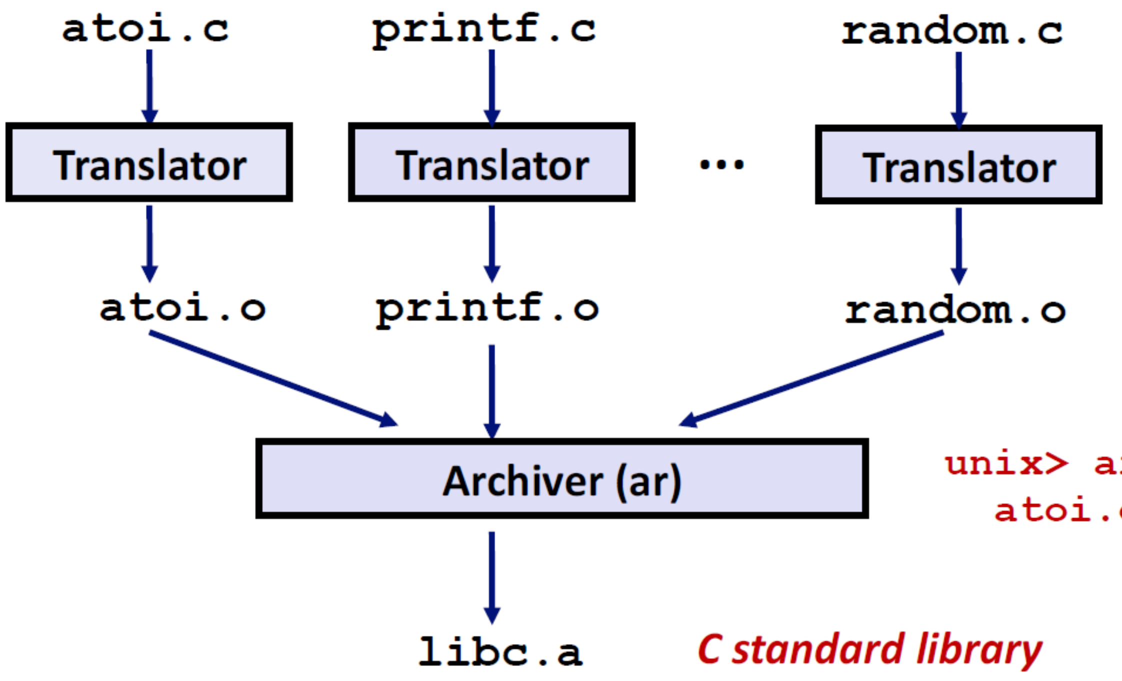
- The (External) Symbol Table does contain variables with static storage duration, but internal linking.

# What Do Linkers Do? (cont)

**Step 2. Relocation**



# Creating Static Libraries




*C standard library*

```
unix> 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 MB archive of almost 1500 object files.
  - I/O, memory allocation, signal handling, string handling, etc.
- libm.a (The C math library)
  - 2 MB archive of almost 500 object files.
  - Floating-point math (sin, cos, tan, exp, etc.)

```
msabt@sabtmoha:~$ ar -t /usr/lib/x86_64-linux-gnu/libm.a
e_exp2l.o
e_exp2.o
e_exp-avx.o
e_exp-fma4.o
e_expf.o
e_expl.o
e_exp.o
e_fmodf.o
e_fmodl.o
e_fmod.o
```

```
msabt@sabtmoha:~$ ar -t /usr/lib/x86_64-linux-gnu/libc.a
prctl.o
pread64_chk.o
pread64.o
pread_chk.o
pread.o
preadv64.o
preadv.o
printf_chk.o
printf_fphex.o
printf_fp.o
```

# Using Static Libraries

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

# Part III

*Static Linking (2)*

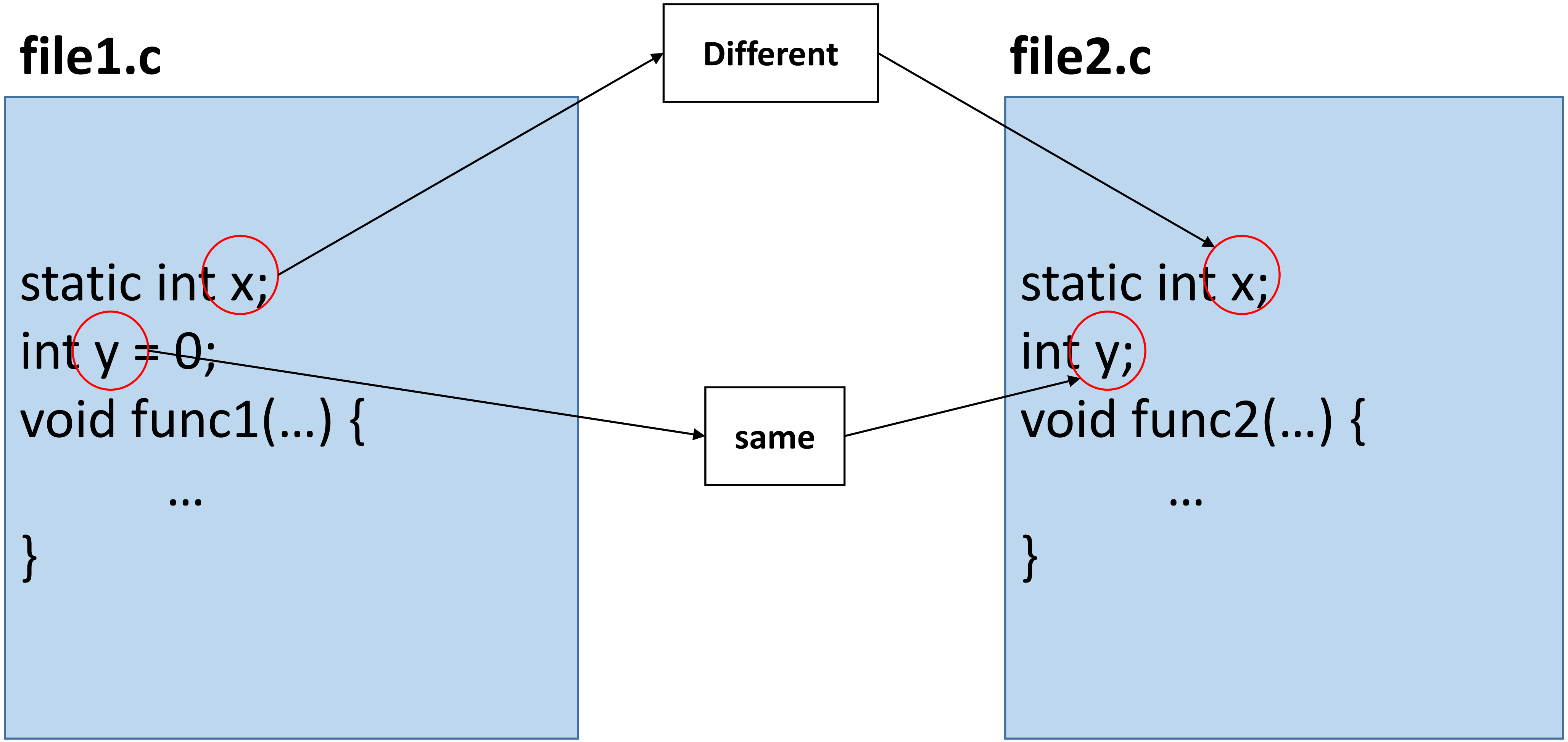
# Global Variables

- Initialized vs. Uninitialized
  - Initialized allocated in data section.
  - Uninitialized allocated in bss section.
    - Implicitly initialized to zero.

- File scope vs. program scope
  - Static global variables known only within file that declares them.
    - Two of same name in different files are different.

  - Non-static global variables potentially shared across all files.
    - Two of same name in different files are same.

# Scope


file1.c
static int x;
int y = 0;
void func1(…) {
…
}
Different
same
file2.c
static int x;
int y;
void func2(…) {
…
}

# Reconciling Program Scope

**file1.c**

```
static int x;
int y = 0;
void func1(…) {
        …
}
```

**complete definition**

**file2.c**

```
static int x;
int y;
void func2(…) {
        …
}
```

**Tentative definition**

# Linker Symbols

- **Global symbols**
  - Symbols defined by module “m” that can be referenced by other modules.
  - Examples: non-static functions and non-static global variables.

- **External symbols**
  - Global symbols that are referenced by a module “m”, but defined by some other module.

- **Local symbols**
  - Symbols that are defined and referenced exclusively by module “m”.
  - Examples: static functions and variables.

# Resolving Symbols

**main.c**

Global

```c
int array[2] = {1, 2};
void swap(int*, int*);
int main(void) {
        swap(array, array + 1);
        return 0;
}
```

External

**swap.c**

Global

```c
void swap(int * a, int * b) {
        int temp = *a;
        *a = *b;
        *b = temp;
}
```

Linker knows nothing

# Strong and Weak Symbols

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

**p1.c**

```c
int foo=5;      // strong

p1() {          // strong
}
```

**p2.c**

```c
int foo;        // weak

p2() {          // strong
}
```

# 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 symbol, 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*

# Exercise 1/3

**file1.c**

```
static int x;
int y = 0;
void func1(...) {
        ...
}
```

**file2.c**

```
static int x;
int y = 1;
void func2(...) {
        ...
}
```

# Exercise 2/3

**file1.c**

```
static int x;
int y = 1;
void func1(…) {
        …
}
```

**file2.c**

```
static int x;
int y = 1;
void func2(…) {
        …
}
```

# Exercise 3/3

**file1.c**

```
static int x;
extern int y;
void func1(...) {
        ...
}
```

**file2.c**

```
static int x;
int y = 1;
void func2(...) {
        ...
}
```

# Default Values

**file1.c**

```
static int x;
__attribute__((weak))int y = 1;
void func1(…) {
        …
}
```

**file2.c**

```
static int x;
int y = 2;
void func1(…) {
        …
}
```

# Global Symbols in Assembly

**file1.c**

```
int y = 1;
Int * yp = &y;
int func(void);
void main(void) {
        int x = *yp + 1;
        return func();
}
```

**file1.s**

```
        .file   "file1.c"
        .text
        .globl  y
        .data
        .align 4
        .type   y, @object
        .size   y, 4
y:
        .long   1
        .globl  yp
        .section        .data.rel.local,"aw"
        .align 8
        .type   yp, @object
        .size   yp, 8
yp:
        .quad   y
        .text
        .globl  main
        .type   main, @function
main:
```

# Assembly Directives

- **.file:** supplies information to be placed in the object file and the executable.

- **.data:** indicates that what follows goes in the data section.

- **.globl:** indicates that the defined symbol will be used by other modules, and this should be made known to the linker.

- **.type:** indicates how the symbol is used. Two possibilities are function and object.

- **.size (.align):** indicates the size (alignment) should be associated with the given symbol.

# New Programs (Relocation)

**file1.c**

```
void doAlmostNothing(void);
void main(void) {
	doAlmostNothing();
	return 0;
}
```

**nothing.c**

```
static void doNothingStatic(void){}
void doNothing(void){}
void doAlmostNothing(void) {
	doNothingStatic();
	doNothing();
}
```

# Symbols in nothing.o

```
$objdump -d nothing.o

nothing.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <doNothingStatic>:
   0:	55                   	push   %rbp
   1:	48 89 e5             	mov    %rsp,%rbp
   4:	90                   	nop
   5:	5d                   	pop    %rbp
   6:	c3                   	retq

0000000000000007 <doNothing>:
   7:	55                   	push   %rbp
   8:	48 89 e5             	mov    %rsp,%rbp
   b:	90                   	nop
   c:	5d                   	pop    %rbp
   d:	c3                   	retq
```

```
000000000000000e <doAlmostNothing>:
   e:	55                   	push   %rbp
   f:	48 89 e5             	mov    %rsp,%rbp
  12:	b8 00 00 00 00       	mov    $0x0,%eax
  17:	e8 e4 ff ff ff       	callq  0 <doNothingStatic>
  1c:	b8 00 00 00 00       	mov    $0x0,%eax
  21:	e8 00 00 00 00       	callq  26 <doAlmostNothing+0x18>
  26:	90                   	nop
  27:	5d                   	pop    %rbp
  28:	c3                   	retq
```

# Relocations in nothing.o

```
$readelf -r nothing.o

Relocation section '.rela.text' at offset 0x250 contains 1 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000000022  000900000002 R_X86_64_PC32     0000000000000007 doNothing - 4
```

- What this says to the link editor is: “Be careful, what is at offset 22 has to be replaced by an address that can be calculated in the way described by the relocation type X86_64_PC32.
- The type of relocation tells the linker how to calculate the effective address. In this case S + A - P where:
  - S: The value of the symbol whose index resides in the relocation entry.
  - A: The addend used to compute the value of the relocatable field.
  - P: The section offset or address of the storage unit being relocated

# Doing Relocation

- Linker is provided instructions for updating object files.

- Lots of ways addresses can appear in machine code.
- Three in common use on x86-64
  - 32-bit absolute addresses (.text)
  - 64-bit absolute addresses (.data)
  - 32-bit PC-relative addresses (both .text and .data)

# Final Binary

```
0000000000000680 <doNothingStatic>:
 680:   55                      push   %rbp
 681:   48 89 e5                mov    %rsp,%rbp
 684:   90                      nop
 685:   5d                      pop    %rbp
 686:   c3                      retq

0000000000000687 <doNothing>:
 687:   55                      push   %rbp
 688:   48 89 e5                mov    %rsp,%rbp
 68b:   90                      nop
 68c:   5d                      pop    %rbp
 68d:   c3                      retq
```

```
000000000000068e <doAlmostNothing>:
 68e:   55                      push   %rbp
 68f:   48 89 e5                mov    %rsp,%rbp
 692:   b8 00 00 00 00          mov    $0x0,%eax
 697:   e8 e4 ff ff ff          callq  680 <doNothingStatic>
 69c:   b8 00 00 00 00          mov    $0x0,%eax
 6a1:   e8 e1 ff ff ff          callq  687 <doNothing>
 6a6:   90                      nop
 6a7:   5d                      pop    %rbp
 6a8:   c3                      retq
 6a9:   0f 1f 80 00 00 00 00    nopl   0x0(%rax)
```

# Doing Relocation

- The call to doNothingStatic has not changed. The call to doNothingStatic was already relative jump from the next instruction to execute.

- The linker calculated that call to doNothing was a jump to 0x6a6 + 0xffffffe1 = 0x687.

# Doing Relocation

- The .text section of nothing.o starts at 0x680.
- The linker knows from the relocation section that
  - it will have to change the value at 0x6a2 = (Try to guess?)
  - So that it jumps towards 0x687 = (try to guess)

- The relative jump is
  - 0x687 – 0x6a6 = ???? = S + A – P
  - Guess S, A, and P.

# Part IV

*More on ELF Linking*

# The Under-Estimated Task 1/2

- Unlike compilation, the linking process remains largely invisible and poorly studied, for programmers and researchers alike.

```
void* _int_malloc(mstate av, size_t bytes)
  { ... }
void* __libc_malloc(size_t bytes)
  { ...
    void *mem = _int_malloc(av, sz);
    ...
  }
void* __libc_calloc(size_t bytes)
  { ...
    void *mem = _int_malloc(av, sz);
    ...
  }
strong_alias (__libc_malloc, __malloc) /* These expand to */
strong_alias (__libc_malloc, malloc)   /* asm directives   */
strong_alias (__libc_calloc, __calloc) /* and/or compiler  */
weak_alias (__libc_calloc, calloc)     /* attributes       */
```

# The Under-Estimated Task 2/2

- We can see that __libc_malloc and __libc_calloc using a third internal helper function called _int_malloc().
  - Those two functions are not guaranteed to always call the helper.
  - By defining calloc as alternative ‘weak’ name, user code may optionally supply its own calloc (overriding the local ‘weak’ alias)

- These strong_alias and weak_alias directives step outside the bounds of the C language:
  - they are macro-expanded to assembler directives controlling the object file sent to the linker.

- Conventional source-language semantics do not attempt to address the questions posed in the previous paragraph, but in practice linker features are used to control name binding and symbol visibility.

# Linker Speak

- Much software is not written merely in a programming language like C, but also in ‘linker speak’.

- « Linker Speak » include
  - the linker command-line,
  - metadata contained within object files,
  - assembler and compiler directives that generate that.

- Linking is not simply a matter of separate compilation. Systems code and application code alike.

# Linker Speak – Arguments

- These are command-line options supplied when invoking the linker.

- In dynamic linking, environment variables serve an analogous purpose.

# Linker Speak – Scripts

- Most linkers embed a script language.
  - Although programmers rarely see it, every link job is controlled by a unique control script.

- Purposes:
  - It used to control how sections in input files are mapped in the output file.
  - It also provides means for controlling the program entry point, describing regions of memory and their flags, alignment, and so on.

- The user may supply their own script, overriding the built-in default.

# Linker Speak – Metadata

- Object files also contain metadata.

- These metadata have corresponding forms in
  - Assemblers (directive, or pseudo-operations)
  - Compilers (attributes)

# Use-Cases – Encapsulation

- Source-language encapsulation features, such as static modifier, may map directly to linker features, such as ELF's local symbols.

- Linkers expose three other encapsulation facilities that are not supported by the language:
  - ELF symbol visibility attributes,
  - Archives (static libraries),
  - Dynamic export control.

- Compiler options can hide symbols (-fvisibility=hidden) by default.

# Use-Cases – Built-Time Substitution

- Link-time mechanisms may be used to substitute one definition for another.

- Multiple definitions are allowed if all are marked ‘weak’;
  - An ordinary strong definition takes precedence, but otherwise the first weak definition is chosen.

- The semantics of archives are such that a C program can supply its own malloc.o while still linking with the remainder of the C library libc.a.

# Use-Cases – Load-Time Substitution

- Dynamic linkers offer another substitution feature: LD_PRELOAD
  - This environment variable can supply a named library, whose definitions take precedence over those on all other libraries (but not in the executable).

- This works only when the program is dynamically linked.

# Use-Cases – Interposition

- Interposition can be seen as substitution where the prior definition is re-used by the substituted one.
  - The linker’s --wrap option.

- Linking with --wrap *func* redirects *func* to __wrap_func, which may call the real func to reach the original definition.

- The semantics of --wrap affect only reference to undefined symbols.

# Use-Cases – Optionality

- Weak symbols allow codebases to reference optional features.
  - Unresolved weak symbols are specified to take the value 0, so the absence of a definition can be identified.

# Use-Cases – Aliases

- In most programming languages, a definition has exactly one name.
- At link time, however, the same range of bytes may have multiple symbol names:
  - Each denoting the same address but with different metadata.

- __attribute__(alias(“”))

# Use-Cases – Versioning

- Shared libraries must allow old clients to be executed against a newer library binary.

- To prevent interface changes from breaking old clients, modern dynamic linkers support symbol versioning
  - allowing multiple versions of an interface to be exposed by a single backward-compatible binary.