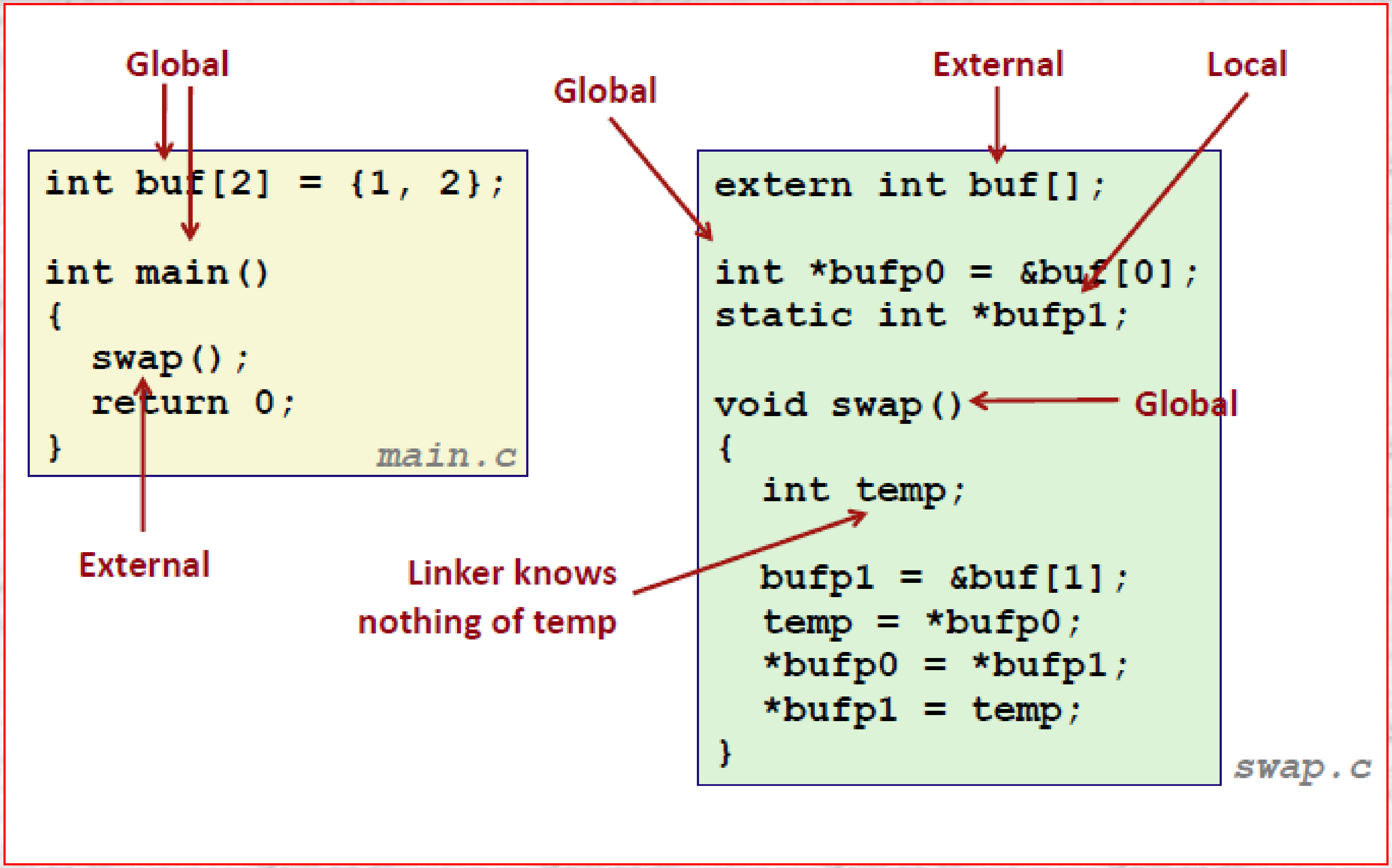
Resolving Symbols



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

int main()
{
  swap();
  return 0;
}
```

main.c

```
extern int buf[];

int *bufp0 = &buf[0];
static int *bufp1;

void swap()
{
  int temp;

  bufp1 = &buf[1];
  temp = *bufp0;
  *bufp0 = *bufp1;
  *bufp1 = temp;
}
```

swap.c

Relocating Code and Data

main.c

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

int main()
{
  swap();
  return 0;
}
```

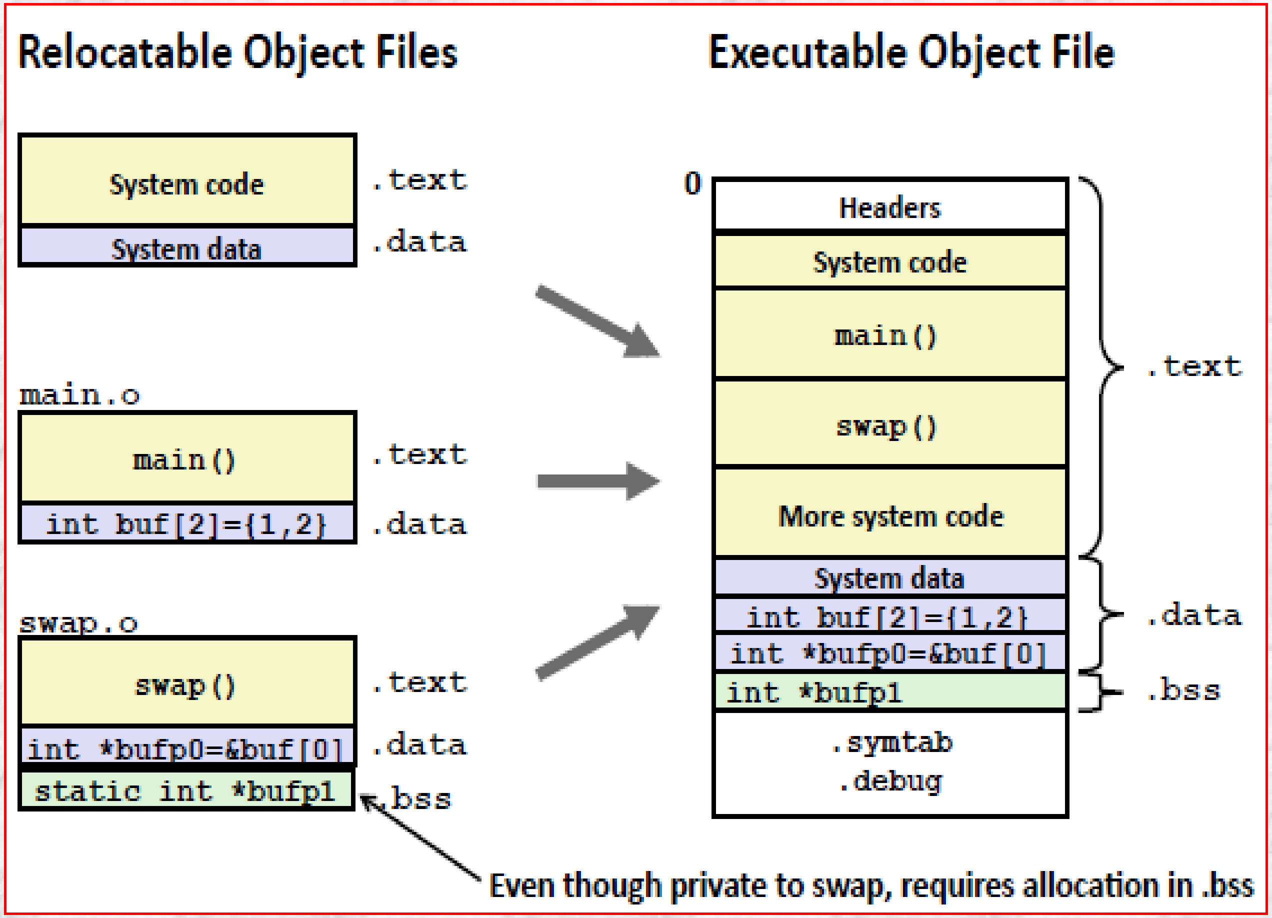
swap.c

```
extern int buf[];

int *bufp0 = &buf[0];
static int *bufp1;

void swap()
{
  int temp;

  bufp1 = &buf[1];
  temp = *bufp0;
  *bufp0 = *bufp1;
  *bufp1 = temp;
}
```



Relocatable Object Files

System code .text
System data .data

main.o
main() .text
int buf[2]={1,2} .data

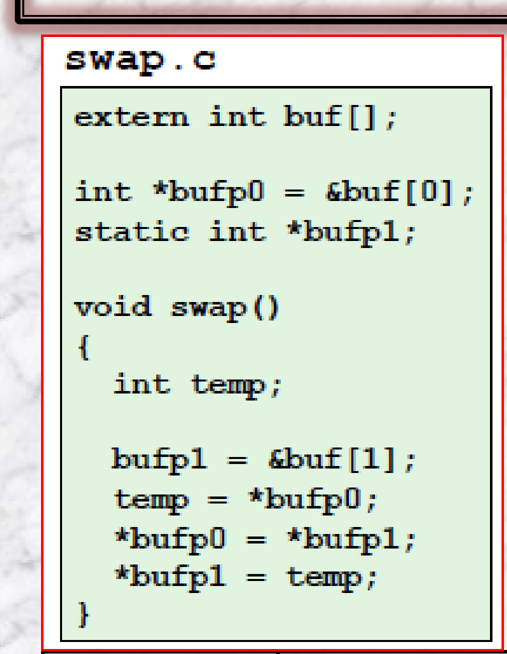
swap.o
swap() .text
int *bufp0=&buf[0] .data
static int *bufp1 .bss

Executable Object File

0
Headers
System code
main()
swap()
More system code
.text
System data
int buf[2]={1,2}
int *bufp0=&buf[0]
.data
int *bufp1
.bss
.symtab
.debug

Even though private to swap, requires allocation in .bss

Practice problem 7.1 (pg 662)



swap.c

```
extern int buf[];

int *bufp0 = &buf[0];
static int *bufp1;

void swap()
{
  int temp;

  bufp1 = &buf[1];
  temp = *bufp0;
  *bufp0 = *bufp1;
  *bufp1 = temp;
}
```

- SYMBOL TABLE →.symtab
- Info about functions and global variables that are defined and referenced in a program
  - Does not contain entries for local variables

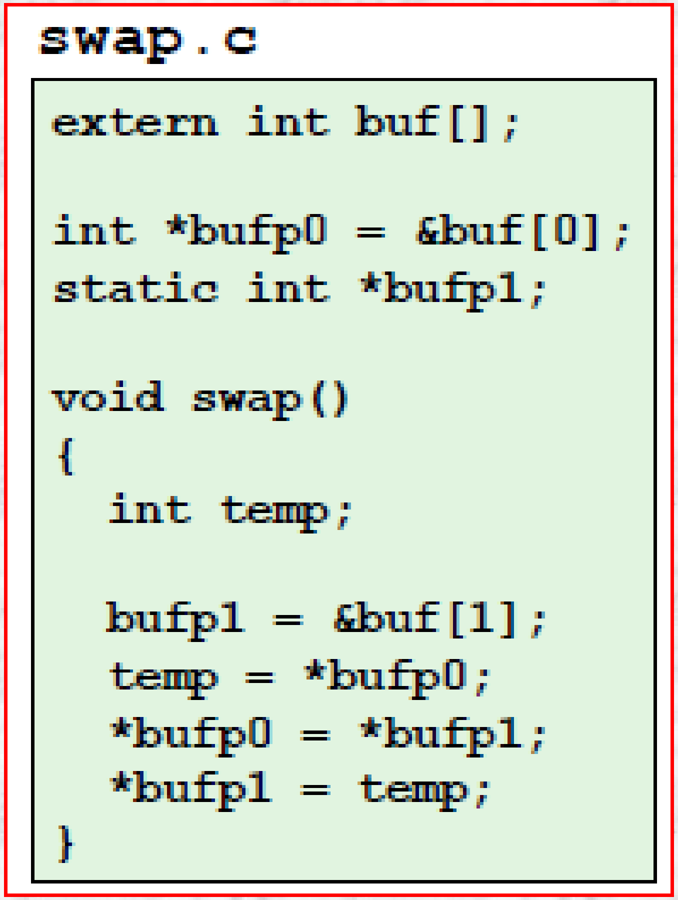
- Understanding the relationship between linker symbols and C variables/functions… notice that the C local variable temp does NOT have a symbol table entry. Why? It goes on the stack!

Symbol	swap.o .symtab entry?	symbol type	module where defined	section
buf	yes	extern	main.o	.data
bufp0	yes	global	swap.o	.data
bufp1	yes	global	swap.o	.bss
swap	yes	global	swap.o	.text
temp	no	---	---	---

Swap relocatable symbol table

```
% objdump -r -d -t swap.o
swap.o:     file format elf32-i386
SYMBOL TABLE:
00000000 l    df *ABS*       00000000 swap.c
00000000 l    d  .text       00000000 .text
00000000 l    d  .data       00000000 .data
00000000 l    d  .bss        00000000 .bss
00000000 l    O .bss         00000004 bufp1
00000000 g    O .data        00000004 bufp0
00000000      *UND*          00000000 buf
00000000 g     F .text       00000035 swap
```

O = object	F = function
d = debug	f = file
l = local	g = global



swap.c

```
extern int buf[];

int *bufp0 = &buf[0];
static int *bufp1;

void swap()
{
  int temp;

  bufp1 = &buf[1];
  temp = *bufp0;
  *bufp0 = *bufp1;
  *bufp1 = temp;
}
```

Symbols and Symbol Tables

- Local linker symbols != local program variables
  - .symtab does not contain any symbols that correspond to local non-static program variables.
  - These are managed at run time on the stack and are not of interest to the linker
  - However… local procedure variables that are defined with the C static attribute (EXCEPTION) are not managed on the stack
    - The compiler allocates space in .data or .bss for each and created a local linker symbol in the symbol table with a unique name

FYI: Static variable’s lifetime extends across the entire run of the program where local variables are allocated and deallocated on the stack

SYMBOL TABLES

- Built by assemblers using symbols exported by the compiler into the .s file
- An ELF symbol table is contained in the .symtab section
- It contains an array of entries where each entry contains:
  - Symbol’s value i.e. address
  - Flag bits (l=local, g = global, F=function, etc)
    - Characters and spaces – up to 7 bits
  - Section or
    - *ABS* (absolute – not in any section)
    - *UND* if referenced but not defined
  - Alignment or size
  - Symbol’s name

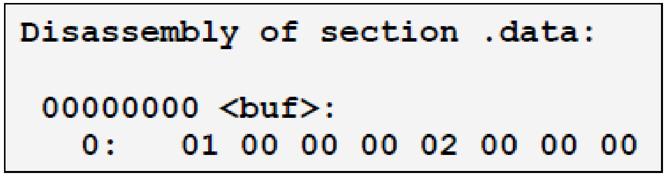
Relocation

- Relocation merges the input modules and assigns run-time addresses to each symbol
- When an assembler generates an object module, it does not know where the code and data will ultimately be stored in memory or the locations of any externally defined functions or global variables referenced by the module
- A “relocation entry” is generated when the assembler encounters a reference to an object who ultimate location is unknown
- 2 types
  - R_386_PC32
  - R_386_32

2 Relocation types

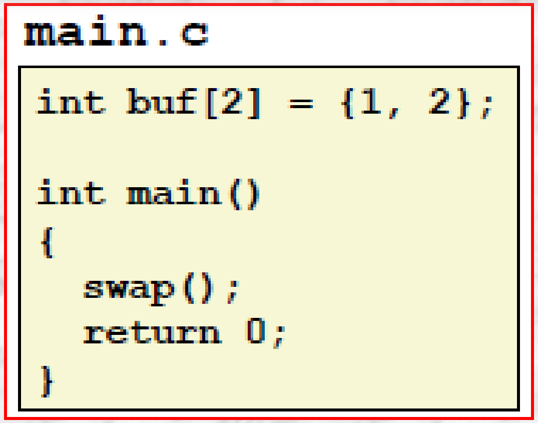
- R_386_PC32
  - relocate a reference that uses a 32-bit PC-relative address.
  - Effective address = PC + instruction encoded addr
- R_386_32
  - Absolute addressing
  - Uses the value encoded in the instruction

Relocation Info



```
Disassembly of section .data:

 00000000 <buf>:
    0:    01 00 00 00 02 00 00 00
```



main.c

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

int main()
{
   swap();
   return 0;
}
```

```
% gcc –c –m32 main.c
% objdump -r -tdata main.o

SYMBOL TABLE:
00000000 l    df *ABS*    00000000 main.c
00000000 l    d  .text    00000000 .text
00000000 l    d  .data    00000000 .data
00000000 l    d  .bss     00000000 .bss
00000000 g     O .data    00000008 buf
00000000 g     F .text    00000014 main
00000000         *UND*    00000000 swap
```

NOTE: relocation entries and instructions are stored in different sections of the object file → .rel.txt vs .text

```
% gcc -o main -m32 main.c
/tmp/ccEVrEUg.o:
In function `main':
main.c:(.text+0x7): undefined
reference to `swap'
collect2:
ld returned 1 exit status
```

```
Disassembly of section .text:
00000000 <main>:
  0:      55              push   %ebp
  1:      89 e5           mov    %esp,%ebp
  3:      83 e4 f0        and    $0xfffffff0,%esp
  6:      e8 fc ff ff ff  call   7 <main+0x7>
                  7: R_386_PC32    swap
  b:      b8 00 00 00 00  mov    $0x0,%eax
 10:      89 ec           mov    %ebp,%esp
 12:      5d              pop    %ebp
 13:      c3              ret
```

0x80483b4

0x80483c8

call swap

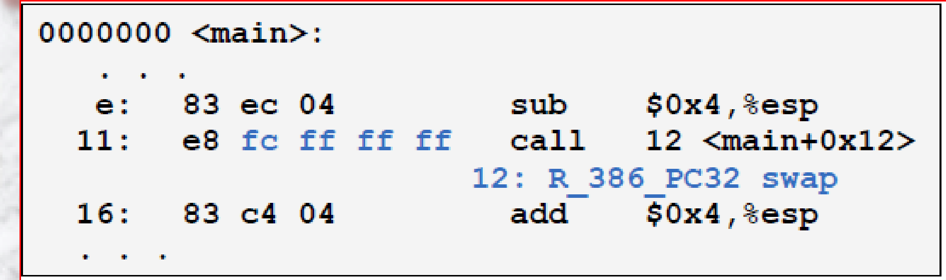
Relocation entry

.text + .offset
0x80483b4 + 0x7
= 0x80483bb = ref addr

Call instruction @ offset 0x6, opcode e8, 32-bit ref (-4)
Offset = 0x7 Symbol = swap
Type = R_386_PC32

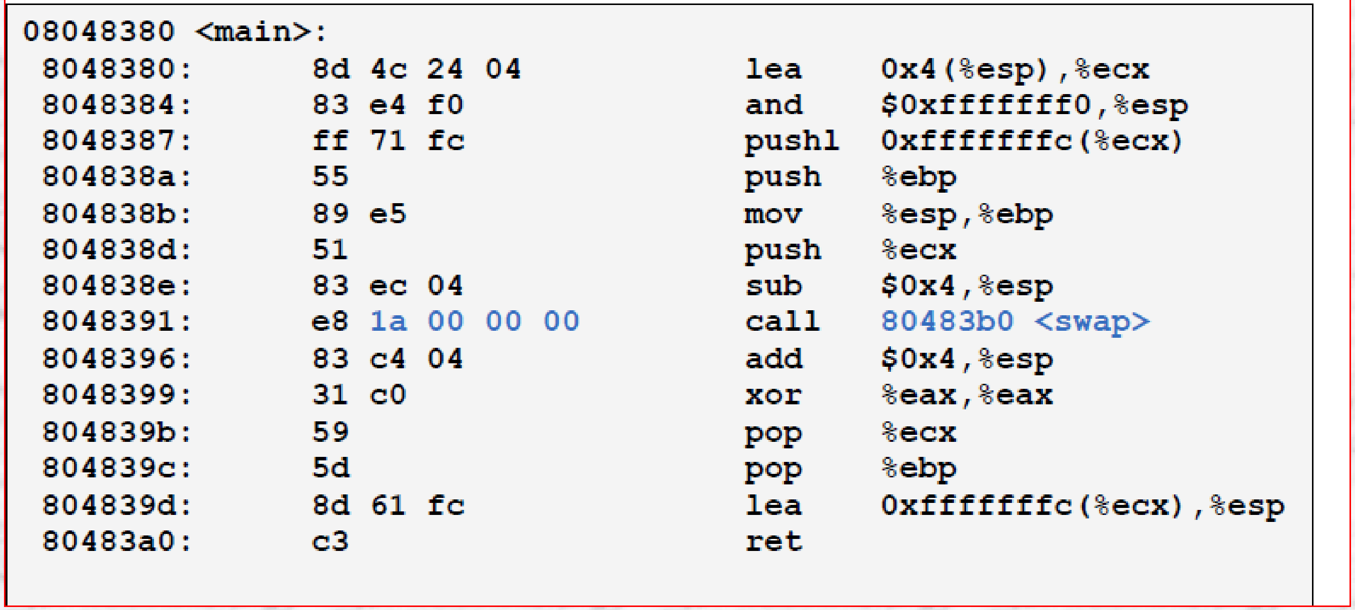
.symbol swap + 32-bit ref – ref addr
0x80483c8 + -4 - 0x80483bb = 0x9

Executable before/after relocation



```
0000000 <main>:
   . . .
   e:   83 ec 04             sub    $0x4,%esp
  11:   e8 fc ff ff ff       call   12 <main+0x12>
                        12: R_386_PC32 swap
  16:   83 c4 04             add    $0x4,%esp
  . . .
```

Return address
0x8048396 + 0x1a
= 0x80483b0
Location of swap function



```
08048380 <main>:
 8048380:       8d 4c 24 04             lea    0x4(%esp),%ecx
 8048384:       83 e4 f0                and    $0xfffffff0,%esp
 8048387:       ff 71 fc                pushl  0xfffffffc(%ecx)
 804838a:       55                      push   %ebp
 804838b:       89 e5                   mov    %esp,%ebp
 804838d:       51                      push   %ecx
 804838e:       83 ec 04                sub    $0x4,%esp
 8048391:       e8 1a 00 00 00          call   80483b0 <swap>
 8048396:       83 c4 04                add    $0x4,%esp
 8048399:       31 c0                   xor    %eax,%eax
 804839b:       59                      pop    %ecx
 804839c:       5d                      pop    %ebp
 804839d:       8d 61 fc                lea    0xfffffffc(%ecx),%esp
 80483a0:       c3                      ret
```

R_386_PC32 relocate a reference that uses a 32-bit PC-relative address.

Effective address = PC + instruction encoded address

.text + .offset = 0x8048380 + 0x12 = 0x8048392 → reference address

Call + 32-bit ref – reference address = 0x80483b0 + -4 – 0x8048392 = 0x1a