
Linking

Philipp Koehn

18 April 2018



Hello World



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

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

Compilation



2

- Compile

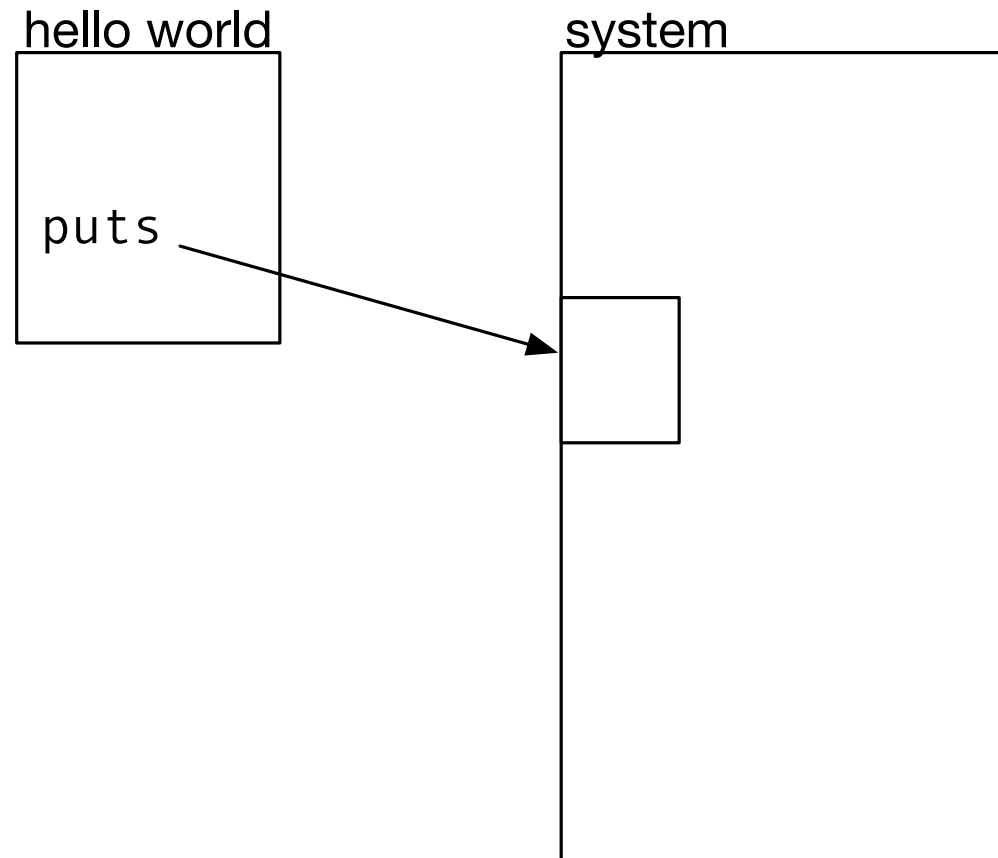
```
linux> gcc -Og hello-world.c
```

- Resulting program

```
linux> ls -l a.out  
-rwxr-xr-x.  1 phi users 8512 Nov 16 03:57 a.out
```

- That's pretty small!

Dynamic Linking



Static Linking



- Compile with `--static`
- Results in very large file
- Includes the entire library!

Benefits of Dynamic Linking



- Makes code smaller
 - needs less disk space
 - needs less RAM
- Library is not part of the compiled program
 - ⇒ when it gets updated, no need to recompile

Example: Code in 2 Files



main.c

```
int sum(int *a, int n);

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

int main() {
    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;
}
```

Compile and Run

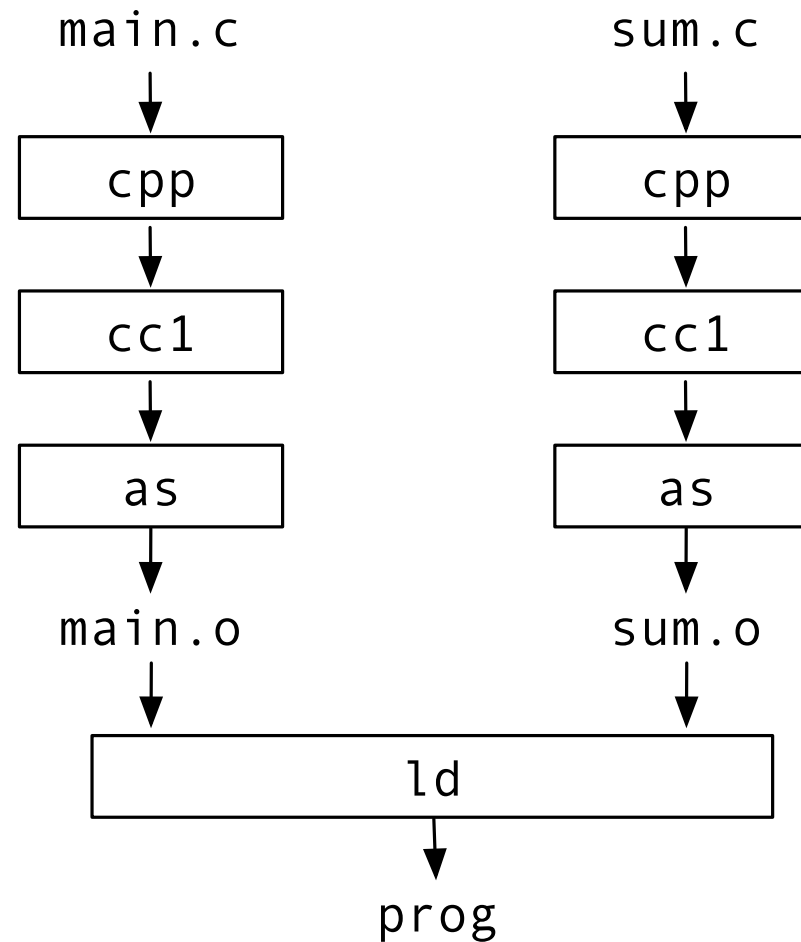


```
linux> gcc -Og -o prog main.c sum.c
linux> ./prog
linux> echo $?
3
```


Static Linking



8



Static Linking



9

- Symbol resolution
 - object files define and reference symbols
(functions, global variables, static variables)
 - need to connect symbol to exactly one definition

- Symbol resolution
 - object files define and reference symbols (functions, global variables, static variables)
 - need to connect symbol to exactly one definition
- Relocation
 - assemblers generate object files that starts at address 0
 - when combining multiple object files, code must be shifted
 - all reference to memory addresses must be adjusted
 - assembler stores meta information in object file
 - linker is guided by relocation entries

Object Files

- Relocatable object file
 - binary code
 - meta information that allows symbol resolution and relocation
- Executable object file
 - binary code
 - can be copied into memory and executed
- Shared object file
 - binary code
 - can be loaded into memory
 - can be linked dynamically

Relocatable Object Files

11



- Executable and Linkable Format (ELF)
 - header
 - sections with different type of data
 - section header table

ELF header
.text
.rodata
.data
.bss
.symtab
.rel.text
.re.data
.debug
.line
.strtab
Section header table

- .text** machine code of compiled program
- .rodata** read-only data (e.g., strings in printf statements)
- .data** initialized global and static C variables
- .bss** uninitialized global and static C variables
- .symtab** symbol table
- .rel.text** list of locations in .text section (machine code)
to be modified when object is relocated
- .rel.data** same for .data
- .debug** debugging symbol table
(only compiled with -g)
- .line** mapping between line number and machine code
(only compiled with -g)
- .strtab** string table for .symtab and .debug

Symbols

- Global symbols that can be used by other objects
- Global symbols of other objects (not defined here)
- Local symbols only used in object defined with "static" attribute
- Note: non-static local variable are not exposed

ELF Symbol Table Entry

Name	Pointer to string of symbol name
Type	Function or data type
Binding	Indicates local or global
Section	Index of which section it belongs to
Value	Section offset
Size	Size in bytes

Example

```
linux> readelf -a main.o
```

```
Section Headers:
```

```
[ 1] .text
```

```
[ 3] .data
```

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
8:	00000000000000000000	24	FUNC	GLOBAL	DEFAULT	1	main
9:	00000000000000000000	8	OBJECT	GLOBAL	DEFAULT	3	array
10:	00000000000000000000	0	NOTYPE	GLOBAL	DEFAULT	UND	sum

- main is a function (FUNC) in section .text (1)
- array is an object (OBJECT) in section .data (3)
- sum is undefined (UND)

- Linker must resolve all symbols to connect references to addresses
- Local symbols are contained to their object, each has a unique name
- Symbols in an object file may be undefined (listed as UND in symbol table)
⇒ these must be defined in other objects
- If not found, linker complains:

```
linux> gcc -Og main.c  
/tmp/ccZzl3Pp.o: In function 'main':  
main.c:(.text+0xf): undefined reference to 'sum'  
collect2: error: ld returned 1 exit status
```

Static Libraries

- Goal: link various standard functions statically
→ binary without dependency
- Plan A
 - put everything into big libc.o
 - link it to the application object file
 - ... but that adds too big of a file
- Plan B
 - have separate object files printf.o, scanf.o, ...
 - link only the ones that are needed
 - ... but that requires a lot of tedious bookkeeping by programmer

Static Libraries

- Solution: archives
 - Combine object files `printf.o`, `scanf.o`, ... into archive `libc.a`
 - Let linker pick out the ones that are needed
- ```
linux> gcc main.c /usr/lib/libc.a
```

- Solution: archives
- Combine object files `printf.o`, `scanf.o`, ... into archive `libc.a`
- Let linker pick out the ones that are needed

```
linux> gcc main.c /usr/lib/libc.a
```

- You can build your own libraries

```
linux> ar rcs libmy.a my1.o my2.o my3.o
```

- Multiple object files
- Merge all sections, e.g., all .data sections together
- Assign run time memory addresses for each symbol
- Modify each symbol reference
- This is aided by relocation entries

# Relocation Entry

|               |                                        |
|---------------|----------------------------------------|
| <b>Offset</b> | Offset of reference within object      |
| <b>Type</b>   | Relocation type                        |
| <b>Symbol</b> | Symbol table index                     |
| <b>Added</b>  | Constant part of relocation expression |

Type may be  
absolute 32 bit address or  
address relative to program counter

# Relocating Symbol Addresses

- main.o

|     |                |       |                |
|-----|----------------|-------|----------------|
| 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     |
| e:  | e8 00 00 00 00 | callq | 13 <main+0x13> |
| 13: | 48 83 c4 08    | add   | \$0x8,%rsp     |
| 17: | c3             | retq  |                |

- Relocation entries

- a: R\_X86\_64\_32 array
- f: R\_X86\_64\_PC32 sum-0x4

- At line 9: reference to array
- At line e: reference to sum function (undefined in object)





000000000000000000 <sum>:

|     |                |        |                    |
|-----|----------------|--------|--------------------|
| 0:  | b8 00 00 00 00 | mov    | \$0x0,%eax         |
| 5:  | ba 00 00 00 00 | mov    | \$0x0,%edx         |
| a:  | eb 09          | jmp    | 15 <sum+0x15>      |
| c:  | 48 63 ca       | movslq | %edx,%rcx          |
| f:  | 03 04 8f       | add    | (%rdi,%rcx,4),%eax |
| 12: | 83 c2 01       | add    | \$0x1,%edx         |
| 15: | 39 f2          | cmp    | %esi,%edx          |
| 17: | 7c f3          | j1     | c <sum+0xc>        |
| 19: | f3 c3          | repz   | retq               |

## main.o + sum.o → prog

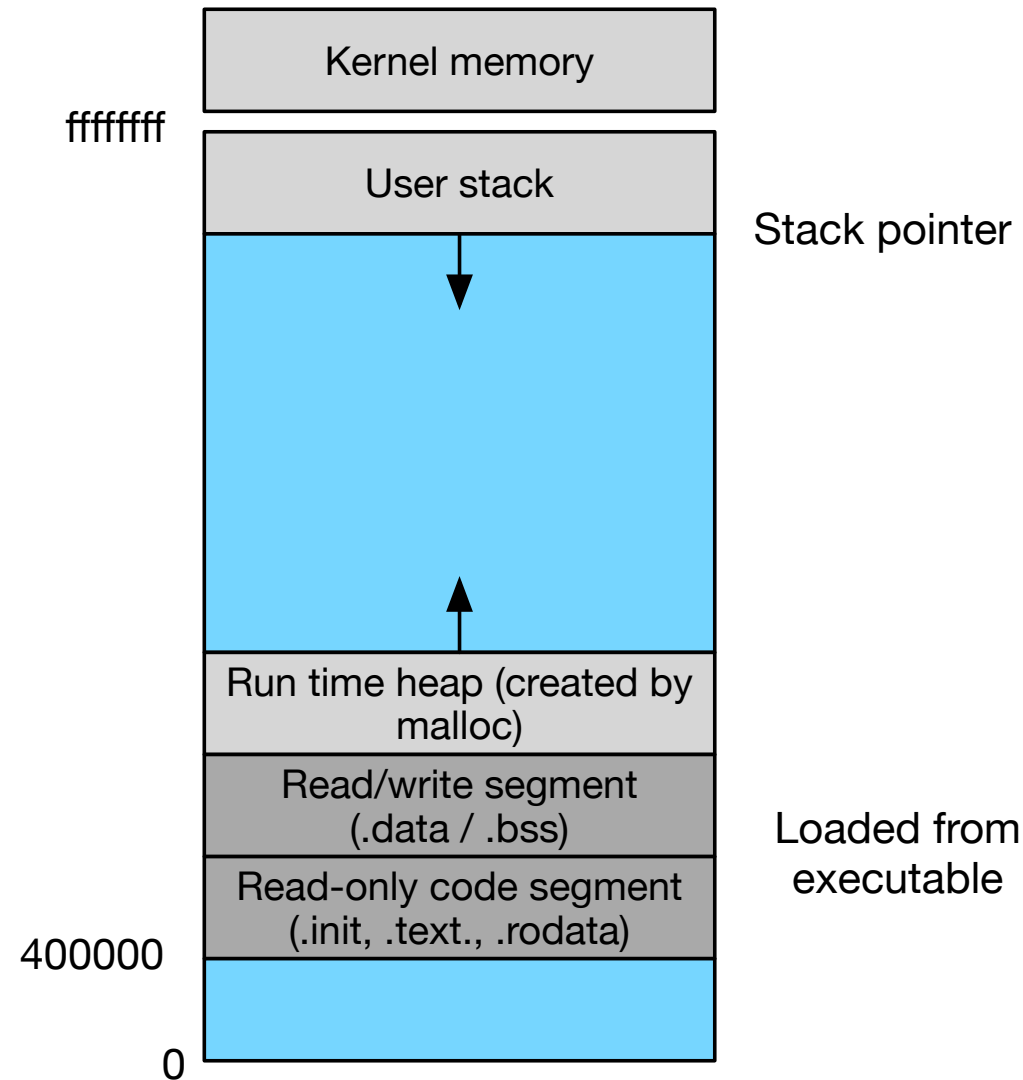
23



```
00000000004004f6 <main>:
 4004f6: 48 83 ec 08 sub $0x8,%rsp
 4004fa: be 02 00 00 00 mov $0x2,%esi
 4004ff: bf 30 10 60 00 mov $0x601030,%edi
 400504: e8 05 00 00 00 callq 40050e <sum>
 400509: 48 83 c4 08 add $0x8,%rsp
 40050d: c3 retq

000000000040050e <sum>:
 40050e: b8 00 00 00 00 mov $0x0,%eax
 400513: ba 00 00 00 00 mov $0x0,%edx
 400518: eb 09 jmp 400523 <sum+0x15>
 40051a: 48 63 ca movslq %edx,%rcx
 40051d: 03 04 8f add (%rdi,%rcx,4),%eax
 400520: 83 c2 01 add $0x1,%edx
 400523: 39 f2 cmp %esi,%edx
 400525: 7c f3 jl 40051a <sum+0xc>
 400527: f3 c3 repz retq
 400529: 0f 1f 80 00 00 00 00 nopl 0x0(%rax)
```

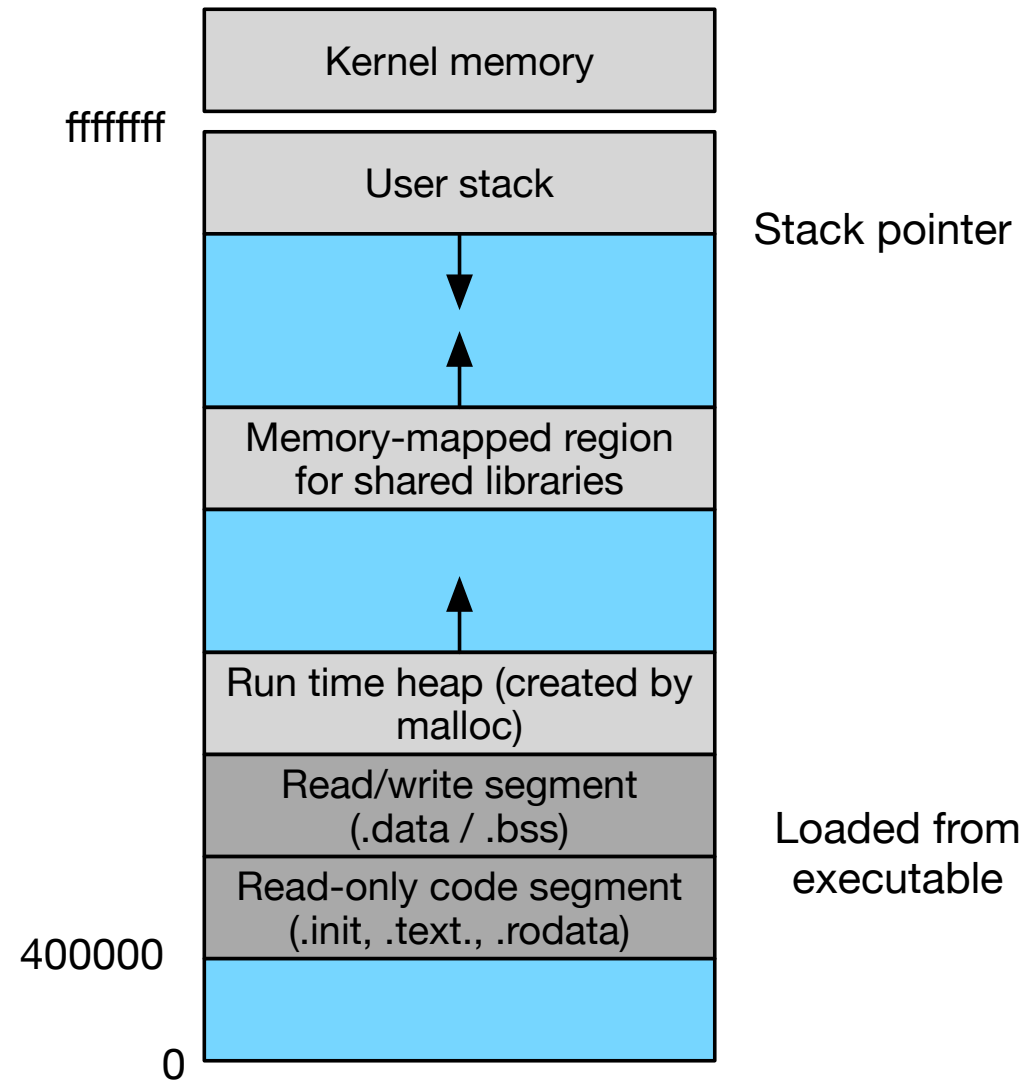
# Loading Executable Object Files



# Dynamic Linking Shared Libraries

- Once program is executed, loader calls dynamic linker
- Dynamic linker "loads" shared library
- Nothing is actually loaded
- Memory mapping: pretend its in memory  
(operation system deals with mapping of RAM address)

# Dynamic Linking Shared Libraries



# Addresses in Shared Libraries

- Multiple processes use same shared library
- Idea: put it into a dedicated place in memory
- But
  - there may be many libraries
  - we may run out of address space  
(or at least waste it)
- Instead: compile into position-independent code

# Position-Independent Code

- No matter where the libraries is loaded into memory  
→ distances between addresses are the same
- Global offset table
  - table in data segment (relative position is known)
  - contains absolute addresses of functions and variables
  - gets filled with correct values by dynamic linker
- Uses instruction point register (%rip)

# Example

- Global offset table (in data segment)

|   |                     |
|---|---------------------|
| 0 | address of symbol a |
| 1 | address of symbol b |
| 2 | ...                 |

- Code

```
mov 0x2008b9(%rip), %rax
addl $1, (%rax)
```

- Distance between code line and GOT entry 1 is 0x2008b9 bytes
- First line of code loads actual address of variable
- Second line increases it by 1



# Tools for Manipulating Object Files

30



**AR** Creates static libraries, and inserts, deletes, and extracts members

**STRINGS** Lists all printable strings

**STRIP** Deletes symbol table information

**NM** Lists symbols defined in symbol table

**READELF** Displays complete structure

**OBJDUMP** Displays all information, useful to disassemble code