Systems 3

Libraries

Marcel Waldvogel

(Handout)

Department of Computer and Information Science
University of Konstanz

Winter 2019/2020

These slides are based on previous lectures held by Alexander Holupirek, Roman Byshko, and especially Stefan Klinger.

Chapter Goals

- How to manage big programs?
- How to split/structure them into modules?
- How modules can be separated/made to interact?
- How to compile big programs (efficiently)?
- What happens behind the scenes?
- How are programs loaded and run?
- The use of header files.
- The use (and dangers) of macros.
- Portable code and conditional compilation.

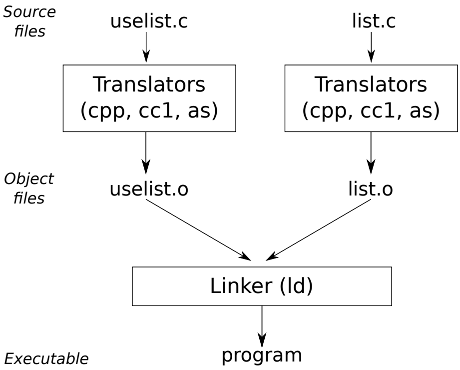
Linking

Some of the examples in this chapter are taken from the excellent book

- Randal E. Bryant, David O'Hallaron. *Computer Systems, A Programmer's Perspective*. 2003, Pearson Education, Prentice Hall. ISBN 0-13-178456-0.

Introduction

- We have already seen how separate compilation works (Lecture on 'Big Programs').
- The compiler driver gcc(1) employs a bunch of different tools for this task:
  - preprocessor cpp(1) — removes comments, applies macros.
  - compiler cc1 — compiles into assembler code.
  - assembler as(1) — translates into binary object file.
  - linker ld(1) — **links together the compiled object files**.



- We will have a closer look at linking now…

Object files

Object files contain chunks of data, (almost) ready to be copied to memory for execution.

- program code, *i.e.*, CPU instructions compiled from your program, and
- constant data (*e.g.*, string literals),

There are three kinds of object files:

- **Executable** object files can be executed directly, *cf.* page 15.
  - Generated by the linker, not by the compiler!
- **Relocatable** object files can be **linked** with other relocatable object files, to form an executable.
  - Symbols may change their position (*cf.* page 23), hence the name.
- **Shared** object files are relocatable object files that can be loaded into memory at runtime, and be shared amongst processes (*cf.* page 39).

The functions, global variables, and `static` variables defined in an object file, can be referred to by name: The **symbols**.

Linker Symbols

Relocatable object files come with a **symbol table**, that lists all the symbols an object file exposes.

- **Global** symbols are defined in the object file, and may be referenced from other object files (no modifier).
- **External** symbols are referenced by the object file, but not defined. *I.e.*, the definition must be provided in another object file (`extern`).
- **Local** symbols are defined and referenced only from within the object file (`static` or compiler-generated).

Note *Local symbols* have nothing to do with function-local variables in a C-program. Unless `static`, they are never visible in the symbol table. (Compare debugger symbols.)

Example

```c
extern int buf[];
int *bufp0 = &buf[0];
int *bufp1;

void swap(void)
{
    int temp;
    static int count = 42;

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

    count++;
}
```

```
$ gcc -c swap.c
$ readelf -s swap.o                    # cf. readelf(1)
Symbol table '.symtab' contains 18 entries:
   Num: Size Type    Bind   Ndx Name
#...
     5:    4 OBJECT  LOCAL    3 count.1597
#...
    14:    8 OBJECT  GLOBAL   3 bufp0
    15:    0 NOTYPE  GLOBAL UND buf
    16:    8 OBJECT  GLOBAL COM bufp1
    17:   74 FUNC    GLOBAL   1 swap
```

(some lines and columns have been removed)

- The local symbol `count` (has its name extended to avoid name clashes) uses 4 bytes, and will be stored in section 3

  (Sections are explained on slide 19)

- Object `bufp0` uses 8 bytes in section 3, function `swap` uses 74B in section 1.
- `buf` is `UND`efined, *i.e.*, referenced by this module, but we have no idea where it will be in the compiled program.
- `COM`mon objects, like `bufp1` are uninitialized, and not yet allocated.

Symbol resolution

- For each **local symbol**, the compiler guarantees exactly one definition. The name is modified to be unique (*e.g.* `count` above).
- If *the compiler* **finds no definition**, it expects it to come from another module, and leaves it to the linker, (*e.g.* `buf` above).
- When **the linker** resolves *global* symbols, several conditions can occur:
  - **No definition** is found in the symbol table of any input object file.
  - **Multiple definitions** are found in different object files, choose one.

Example No `main` function, and `buf` undefined.

```
$ gcc swap.o # without -c, try to build an executable
.../lib/crt1.o: In function ‘_start':
(.text+0x20): undefined reference to ‘main'
swap.o: In function ‘swap':
.../swap.c:12: undefined reference to ‘buf'
swap.o:(.data+0x0): undefined reference to ‘buf'
collect2: error: ld returned 1 exit status
```

- The linker tries to link with `crt1.o`, wich refers to the `main` function.

Choosing one among multiple definitions

- The linker distinguishes weak and strong symbols:
  - Functions and initialized global variables are **strong symbols**.
  - Uninitialized global variables are **weak symbols**.
- If a conflict arises, the strategy is as follows:
  - Multiple strong symbols → raise error.
  - One strong, and multiple weak symbols → choose the strong one.
  - Multiple weak symbols → choose an arbitrary one.

Example Two strong symbols

```
$ cc -c foo1.c bar1.c          # compilation is fine
$ cc foo1.o bar1.o
bar1.o: In function 'main':
.../bar1.c:2: multiple definition of 'main'
foo1.o:.../foo1.c:2: first defined here
collect2: error: ld returned 1 exit status
```

foo1.c

```c
int main(void)
{
    return 0;
}
```

bar1.c

```c
int main(void)
{
    return 0;
}
```

Question What will happen here?

foo2.c

```c
#include <stdio.h>

void f(void);
int x = 12345;

int main(void)
{
    f();
    printf("x = %d\n", x);

    return 0;
}
```

bar2.c

```c
int x = 54321;

void f(void)
{
    x++;
}
```

Example One strong, and one weak symbol.

foo3.c

```c
#include <stdio.h>

void f(void);

int x = 12345;  /* strong */

int main(void)
{
    f();
    printf("x = %d\n", x);

    return 0;
}
```

bar3.c

```c
int x;  /* weak */

void f(void)
{
    x = 54321;
}
```

The result is probably expected:

```
$ gcc -c foo3.c bar3.c
$ gcc foo3.o bar3.o
$ ./a.out
x = 54321
```

- Maybe check out the symbol tables? `readelf -s {foo,bar}3.o`

Example Two weak symbols.

foo4.c

```c
#include <stdio.h>

void f(void);

int x;  /* weak */

int main(void)
{
    x = 12345;
    f();
    printf("x = %d\n", x);

    return 0;
}
```

bar4.c

```c
int x;  /* weak */

void f(void)
{
    x = 54321;
}
```

Again, no surprise:

```
$ gcc -c foo4.c bar4.c
$ gcc foo4.o bar4.o
$ ./a.out
x = 54321
```

Note The linker has unified both variables `x`, and makes references to both symbols address the same space in memory.

Question What about this one?

foo5.c

```
#include <stdio.h>

void f(void);

int x = 12345;  /* strong */
int y = 54321;  /* strong */

int main(void)
{
    f();
    printf("x = %d, y = %d\n", x, y);

    return 0;
}
```

bar5.c

```
double x;  /* weak */

void f(void)
{
    x = 1;
}
```

Explain this:

```
$ gcc -c foo5.c bar5.c
$ gcc foo5.o bar5.o
/usr/bin/ld: Warning: alignment 4
of symbol ‘x' in foo5.o is smaller
than 8 in bar5.o
$ ./a.out
x = 0, y = 1072693248
```

Notes

- Not long ago, the linker would give **no warning** at all.
- It is extremely difficult to **debug** such code.

What else?

- After resolving symbols, the linker knows which definition belongs to each symbol.
- The linker does not know about the type, only about the size.

Recall

- Machine code does not use variable names any more.
- The compiler produced code that accesses variables and functions only by their **memory addresses**.

$\Rightarrow$ How does this go together with separate compilation and symbol resolution?

The program in memory

How does a program start?

- When a program is run, it is **copied into memory** by the **loader**.
    - Copy **text segment**, *i.e.*, the actual machine code,
    - copy **initialized data**,
    - **initialize** uninitialized data,
    - *etc.*
- We want to minimize the amount of data to be copied!
    - Only load parts that are **actually required**,
    - and only load them **when** they are needed.
- Wa want to save memory!
    - Do not load the same code into memory multiple times.
    - **Share** already loaded code between processes.
- Avoid expensive **transformations**
    - Store program on disk in a **format** that allows fast setup of the process image.

Virtual memory

- VM is a mapping from the process' **virtual address space** into the machine's physical address space (organized in **pages**).
- The VM system may flag pages as, *e.g.*, **read only**, **executable**, or **private**, *cf.* `mmap`(2).
- A physical page may reside on disk, until **loaded on demand**.
  - So we compile the memory layout into the executable file,
  - the loader just **maps the file** into the process' virtual address space, and
  - the VM system gets the pages into memory when actually referenced.
- Multiple running instances of a program share their text (machine code) through a *read only* mapping to **the same physical** address space.

Note To achieve all this, the structure of the program file depends on the process' memory layout!

Example

- The mapping of virtual memory can be observed in `/proc/$PID/maps`.
- I have several `xterm`s running, one with PID 23172.

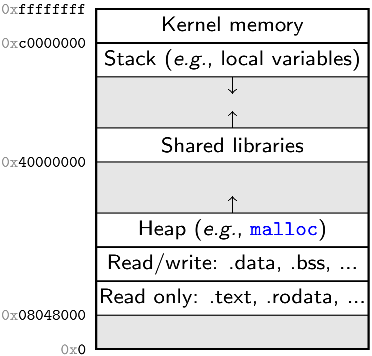
```
$ cat /proc/23172/maps                                        # this is on a 64bit machine
# address                   perms offset   dev   inode    pathname
00400000-00472000            r-xp 00000000 00:10 987773   /usr/bin/xterm
00672000-00673000            r--p 00072000 00:10 987773   /usr/bin/xterm
00673000-0067c000            rw-p 00073000 00:10 987773   /usr/bin/xterm
0067c000-0067e000            rw-p 00000000 00:00 0
006f7000-00718000            rw-p 00000000 00:00 0        [heap]
# ...
7f05d802c000-7f05d81cc000 r-xp 00000000 00:10 953240   /usr/lib/libc-2.18.so
# ...
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]
```

- The executable pages are readonly,
- the writable area is private (*i.e.*, copy-on-write).
- Other instances of `xterm` have the same mapping for the binary `xterm`,
- but may have others for the shared libraries.

Process memory layout

When running, a process has the following **virtual memory layout**.

(This is for 32bit Linux)



- Kernel memory (1GiB) is not accessible by the process.
  - **Shared** among all processes.
- The stack maintains local variables and function calls.
- Shared libraries may even be added at runtime.
- The heap contains allocated memory.
- `.data` and `.bss` store global variables.
- `.text` and `.rodata` are marked `ro`, so can be shared with other processes.

Object file layout

- There are various formats to store binary programs.
- Linux uses ELF, the **Executable and Linking Format**.
- COFF and `a.out` are others, the latter coined the name used by `gcc` for default binaries (in ELF on Linux!).
- All formats have the concept of **sections** in common.
- A section is the unit of organization in a binary.

Some section names (but there are many more)

`.text` The program code, *i.e.*, processor instructions.

`.rodata` Read-only data, *e.g.*, string literals.

`.data` *Initialized global* variables.

`.bss` *Uninitialized global* variables.

`.symtab` The symbol table, displayed with `readelf -s`.

fruits.c

```c
char pear[40];
static double peach;

int mango = 13;
static long melon = 2001;

int main(void)
{
    int i = 3, j, *ip;

    ip = malloc(sizeof(i));
    pear[5] = i;
    peach = 2.0 * mango;

    return 0;
}
```

fruits.o

…
size needed for **.bss** section
.data section initialized global and static variables
.text section machine instructions
…

A typical relocatable object file

This is what the compiler produces out of the individual **C files**.

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

- The **ELF header** describes word size, endian, object file type, machine type, offset and format of the section header table, and other information.
- The **section header table** describes the locations of the various sections.
- Try

```
$ gcc -c -m32 swap.c
$ readelf -S swap.o
```

A typical executable object file

That is what we want to have in the **final binary program**.

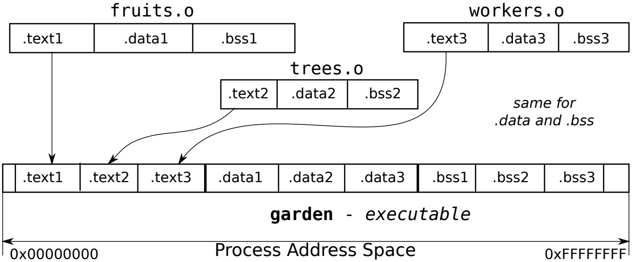
	ELF header
	Segment header table
Code segment mapped read-only	`.init`
	`.text`
	`.rodata`
Data segment mapped read/write	`.data`
	`.bss`
Symbol table and debug info are not loaded into memory	`.symtab`
	`.debug`
	`.line`
	`.strtab`
	Section header table

- The **segment header table** describes the mapping of contiguous file sections into memory.
- Try this

```
$ gcc -m32 -static swap.c \
> main.c
$ readelf -l a.out
```

Relocation

- So after resolving the symbols, the linker needs to put all the code from the individual object files' sections into the final program's sections:



- This process is called **Relocation**.

Relocation involves **two tasks**:

1. Relocating **sections** and symbol **definitions**.
   - Merge sections of the same type.
   - Assign **run-time memory addresses** to the new aggregate sections, the input sections, and each symbol defined in the input.

   ⇒ After this step, every global symbol has a known run-time memory address.
2. Relocating symbol **references** everywhere in the code.
   - Modification of each reference in `.text` and `.data`, so that they point to correct location.

Relocation entries

- The assembler does not know where data and code will be stored ultimately,
- nor does it know addresses of the external objects.

⇒ In such situations a *relocation entry* is generated by the assembler.

Relocation entries

```c
typedef struct {
  int offset;       /* Offset of the reference to relocate. */
  int symbol: 24,   /* Symbol the reference should point to. */
      type:8;       /* Relocation type. */
} Elf32_Rel;
```

- ELF defines 11 different relocation types
- We will look at R_386_32 only.
  This is used to relocate 32bit absolute addresses.

Example: Relocation at work

- Function `f` simply assigns `0xbeef` to the global variable `x`.
- The final memory location of `x` is not yet known.
- Relocation will fix the runtime address of `x`.

foo6.c

```c
int x = 0xdead;

void f(void)
{
    x = 0xbeef;
}
```

- First we **compile** `foo.c` into a *relocatable object file*:

```
$ gcc -m32 -c foo6.c
```

- Have a look at the `.data` section:

```
$ objdump -d -j.data foo6.o
Disassembly of section .data:
00000000 <x>:
   0:   ad de 00 00                                        ....
```

  - Variable `x` appears at address `0x0` in the `.data` section.
  - The value `0xdead` is stored there.

- Have a look at the .text section:

```
1 $ objdump -d -j.text foo6.o
2 Disassembly of section .text:
3 00000000 <f>:
4    0:   55                          push   %ebp
5    1:   89 e5                       mov    %esp,%ebp
6    3:   c7 05 00 00 00 00 ef        movl   $0xbeef,0x0
7    a:   be 00 00
8    d:   5d                          pop    %ebp
9    e:   c3                          ret
```

  - In line 6, the value 0xbeef is copied to address 0x0.
  - This address 0x0 appears at **offset 5** in the .text section.

- These are the relocation entries:

```
$ objdump -r -j.text foo6.o
RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000005 R_386_32          x
```

  - So on relocation of symbol x, the absolute 32bit address at **offset 5** in the .text section must be updated.

- Then we **link** `foo.o` with something that uses `f`.

```
$ gcc -m32 foo6.o bar6.c          /* main in bar6.c simply calls f in foo6.c */
```

- Have a look at the `.data` section **after relocation**:

```
$ objdump -d -j.data a.out
Disassembly of section .data:
08049698 <x>:
 8049698:       ad de 00 00                                        ....
```

  - Variable `x` has been moved to address `0x8049698` in the `.data` section.
  - The value `0xdead` is stored there.

- Have a look at the `.text` section **after relocation**:

```
$ objdump -d -j.text a.out | grep -C3 beef
080483cd <f>:
 80483cd:       55                          push   %ebp
 80483ce:       89 e5                       mov    %esp,%ebp
 80483d0:       c7 05 98 96 04 08 ef        movl   $0xbeef,0x8049698
 80483d7:       be 00 00
 80483da:       5d                          pop    %ebp
 80483db:       c3                          ret
```

  - The reference to variable `x` has been **updated to address** `0x8049698`.

Example: Relocation in the `.data` section

- Sometimes, updating references in the `.text` section is not enough.
- Here we have a global variable `xp` initialized with an address!
- The compiler **cannot even fix a value** for `xp`!

`foo7.c`

```c
int x = 0xdead;
int *xp = &x;

void f(void)
{
    *xp = 0xbeef;
}
```

- Again, we compile and link our object files:

```
$ gcc -m32 -c foo7.c
$ gcc -m32 foo7.o bar6.c
```

- Relocation in the `.data` section.

```
$ objdump -d -j.data foo7.o
Disassembly of section .data:
00000000 <x>:
   0:   ad de 00 00                                         ....
00000004 <xp>:
   4:   00 00 00 00     # This value has to be updated on relocation of x!
$ objdump -r -j.data foo7.o    # Note: in .data this time!
RELOCATION RECORDS FOR [.data]:
OFFSET   TYPE              VALUE
00000004 R_386_32          x
$ objdump -d -j.data a.out
Disassembly of section .data:
08049698 <x>:
 8049698:       ad de 00 00                                         ....
0804969c <xp>:
 804969c:       98 96 04 08                                         ....
```

- Relocation in the `.text` section:

```
$ objdump -d -j.text foo7.o
Disassembly of section .text:
00000000 <f>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   a1 00 00 00 00          mov    0x0,%eax
   8:   c7 00 ef be 00 00       movl   $0xbeef,(%eax)
   e:   5d                      pop    %ebp
   f:   c3                      ret
$ objdump -r -j.text foo7.o
RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000004 R_386_32          xp
$ objdump -d -j.text a.out | grep -C3 beef
 80483cd:       55                      push   %ebp
 80483ce:       89 e5                   mov    %esp,%ebp
 80483d0:       a1 9c 96 04 08          mov    0x804969c,%eax
 80483d5:       c7 00 ef be 00 00       movl   $0xbeef,(%eax)
 80483db:       5d                      pop    %ebp
 80483dc:       c3                      ret
```

- Recall: `0x804969c` is the address of the variable `xp`!

Static libraries

- A static library is a **collection of relocatable object files**.
  - Since the term "object file" is not correct in this context, the members of a library are referred to as **object modules** instead.
- Linking with a library means to link with all the **required** object files.

Why use libraries at all?

- Why not put all library functions into one relocatable object file?
  - An object module is added to a program in its **entirety, or not at all**.
  - The potentially **large object file** would be added to every binary using one of the functions. ⇒ Waste of space.
- Why not copy only **required functions** from an object file?
  - Sections like `.text` and `.data` are merely binary blocks to the linker.
- Why not **explicitly link** all the required object files with the binary?
  - Tedious with object modules at the granulatiry of individual functions!

```
$ gcc -o main main.c printf.o atoi.o read.o write.o ...
```

Making a static library

- A static library is an **archive** of relocatable object modules

`ar rcs archive [member...]` Create archive, containing the members.

`nm archive` List symbols from object files or archives.

- The `ar`(1) command provides various means to modify an archive.
  - `r` Add the members to archive, **replace** existing with the same name.
  - `c` **Create** the archive if it does not exist.
  - `s` Write an object-file **index** into the archive.
  - ... many others
- `nm`(1) displays the symbols defined in a module, or a library.

- By the way: See `info binutils` for an overview of tools for manipulating ELF binaries.

Example

addvec.c

```c
#include "addvec.h"

void addvec(int n, int *x, int *y,
    int *z)
{
    for (int i = 0; i < n; i++)
        z[i] = x[i] + y[i];
}
```

dotproduct.c

```c
#include "dotproduct.h"

int dotproduct(int n, int *x, int *y)
{
    int r = 0;
    for (int i = 0; i < n; i++)
        r += x[i] * y[i];
    return r;
}
```

- The header files just contain the respective function prototype.

```
$ gcc -c addvec.c                    # usually, all this is arranged for in a Makefile
$ gcc -c dotproduct.c
$ ar rcs libvector.a addvec.o dotproduct.o
$ nm libvector.a  # a shiny new library
addvec.o:
0000000000000000 T addvec
dotproduct.o:
0000000000000000 T dotproduct
```

main.c

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

int main(void)
{

        int x[3] = { 1, 0, 0 }, y[3] = { 0, 1, 1 }, z[3] = { 1, 1, 0 };

        printf("<x,y> = %d\n", dotproduct(3, x, y));
        printf("<x,z> = %d\n", dotproduct(3, x, z));

        return 0;
}
```

- The `-static` flag tells `gcc` to build a **statically linked** binary.

```
$ gcc -c main.c
$ gcc -static -omain main.o libvector.a
$ ./main
<x,y> = 0
<x,z> = 1
```

Linker flags for static libraries

- Typically, libraries do not reside in the directory where they are used.
  - With `-lname` the library `libname.a` is searched for in the library search path.
  - With `-Ldir`, a directory is added to the **library search path**.
  - The library search path is searched in the order of the `-L` options.

```
$ gcc -static -omain main.o -L. -lvector      # link with the libvector.a library
```

- The **order** in which libraries are given on the **command line** is significant, and counter-intuitive:
  - The library providing a symbol must appear **after** the object using it.

```
$ gcc -static -omain libvector.a main.o
main.o: In function ‘main':
/home/marcel/Repos//bspk/pk/lect/src/lib/main.c:10: undefined reference
to ‘dotproduct'
/home/marcel/Repos//bspk/pk/lect/src/lib/main.c:12: undefined reference
to ‘dotproduct'
collect2: error: ld returned 1 exit status
$ gcc -static -omain -L. -lvector main.o
# the same error message
```

Input : Files passed to the linker on the command line
Output: A statically linked binary
Data : The set of object modules O **to be linked** to the binary, the set of referenced but yet **unresolved symbols** U, and the set of already **defined symbols** D

$O \leftarrow \emptyset$; $U \leftarrow \emptyset$; $D \leftarrow \emptyset$
foreach input file f given on the command line **do**
  if f is an object file **then**
    $O \leftarrow O \cup \{f\}$; $D \leftarrow D \cup \mathrm{global}\, f$; $U \leftarrow (U \setminus D) \cup \mathrm{external}\, f$
  else if f is an archive **then**
    repeat
      foreach object module m which is a member of f **do**
        if $U \cap \mathrm{global}\, m \neq \emptyset$ **then**
          $O \leftarrow O \cup \{m\}$; $D \leftarrow D \cup \mathrm{global}\, m$; $U \leftarrow (U \setminus D) \cup \mathrm{external}\, m$
    until U and D do not change anymore
if $U \neq \emptyset$ **then**
  Fail with error message: Undefined references to all symbols in U
else
  Relocate object modules in O and build executable.

($\mathrm{global}\, m$ = symbols defined in m; $\mathrm{external}\, m$ = symbols not defined in, but referenced by m)

- Review the previous example: Why does this fail?

```
$ gcc -static -omain libvector.a main.o     # Wrong!
```

  - U is empty when `libvector.a` is visited,
  - so no object modules are added to O.
  - When `main.o` is checked, `dotproduct` is added to U, but `libvector.a` is not visited again.

- Sometimes, it is necessary to specify a library multiple times on the command line. Example (pseudocode!):

```
main.o
    main() { foo(); }
```

```
libfoobar.a
    foo.o
        foo() { ding(); }
    bar.o
        bar() { dong(); }
```

```
libdingdong.a
    ding.o
        ding() { bar(); }
    dong.o
        dong() { foo(); }
```

```
$ gcc -static -omain main.o -L. -lfoobar -ldingdong -lfoobar -ldingdong
```

Shared libraries

- Shared libraries are linked to the program **not until runtime**.
- **Different programs** can use the same shared library.
- The tool `ldd`(1) lists the **dynamic dependencies** of a thus linked binary.[1]

Example

- The binary created in the previous section is quite big:

```
$ gcc -static -o main main.o -L. -lvector
$ ls -l main
-rwx------ 1 marcel users 826k Feb  3 18:37 main
$ ldd main
        not a dynamic executable
```

- This is, because the `-static` flag enforces static linking, including way more than only `libvector`.

[1] Security risk: Runs program!

- Without `-static`, **dynamic linking** is used where possible:

```
$ gcc -o main main.o -L. -lvector
$ ls -l main
-rwx------ 1 marcel users 8.7k Feb  3 18:44 main
$ ldd main
        linux-vdso.so.1 (0x00007fffc013a000)
        libc.so.6 => /usr/lib/libc.so.6 (0x00007ff560b8c000)
        /lib64/ld-linux-x86-64.so.2 (0x00007ff560f36000)
```

  - `linux-vdso.so.1` (Virtual Dynamic Shared Objects) is a part of the kernel, providing **fast system calls**. It is not a shared library in the usual sense.
  - `libc.so.6` is the **standard C library** on Linux systems.
  - `ld-linux-x86-64.so.2` contains the ELF **dynamic linker and loader**.

  Most Programs (unless compiled `-static`) will depend on these.
- Obviously, `libvector.a` is **not shared**, but still statically linked.
  ⇒ How can we change this?

Making a shared library

- The individual object modules have to be compiled with `-fPIC`.
  - This generates **position-independent code**, which allows relocation later on, *cf.* page 46.
- Instead of using `ar`(1), the object files are linked together into a **single shared object file** with `.so` suffix.

Example To build a shared `libvector` instead of a static one:

```
$ gcc -c -fPIC addvec.c
$ gcc -c -fPIC dotproduct.c
$ gcc -shared -o libvector.so addvec.o dotproduct.o
$ nm libvector.so     # my first shared library
0000000000000665 T addvec
# ...
00000000000006d5 T dotproduct
# ...
```

Using a shared library

- The shared library is used just like a static library:

```
$ gcc -c main.c
$ gcc -o main main.o -L. -lvector
```

- The generated binary now **depends** on `libvector.so`:

```
$ ldd main
        linux-vdso.so.1 (0x00007fffe8eaa000)
        libvector.so => not found
        libc.so.6 => /usr/lib/libc.so.6 (0x00007f2cbc343000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f2cbc6ed000)
$ ./main
./main: error while loading shared libraries: libvector.so: cannot open
shared object file: No such file or directory
```

- The dynamic linker `ld-linux`(8) only searches a **default path**:
  - Falling back to `/lib`, and `/usr/lib`, but see the manual!
  - The search path can be extended (prefixed) with `$LD_LIBRARY_PATH`.

```
$ LD_LIBRARY_PATH=. ./main
<x,y> = 0
<x,z> = 1
```

Choosing a shared library at runtime

- Applications can **decide on which shared libraries to load** at runtime.

```
#include <dlfcn.h>

void *dlopen(const char *filename, int flag);
void *dlsym(void *handle, const char *symbol);
int dlclose(void *handle);
char *dlerror(void);
```

- `dlopen`(3) **loads** a shared library, and returns a handle to it.
  - See the manual for how the library is **searched**.
  - The flag indicates when/how to **resolve symbols**:
    - `RTLD_NOW` Before `dlopen` returns, or
    - `RTLD_LAZY` when the called function is needed.
    - ... further flags are available
- `dlsym`(3) returns a **pointer to the symbol** named.
- `dlclose`(3) **unloads** a shared library if it is not used anymore.
- `dlopen`(3) and `dlsym`(3) return `NULL` on failure. `dlerror`(3) returns a string describing the most recent error.

Example

```c
typedef int (*dotproduct_t)(int, int *, int *);

int main(int argc, char *argv[])
{
    dotproduct_t dotproduct;
    void *handle;

    if (argc < 2) errx(1, "use: sick <lib>");

    handle = dlopen(argv[1], RTLD_NOW);
    if (handle == NULL) errx(1, "dlopen: %s", dlerror());

    /* dotproduct = (dotproduct_t)dlsym(handle, "dotproduct"); */
    *(void **)(&dotproduct) = dlsym(handle, "dotproduct");
    if (dotproduct == NULL) errx(1, "dlsym: %s", dlerror());

    int x[3] = { 1, 0, 0 }, y[3] = { 0, 1, 1 }, z[3] = { 1, 1, 0 };
    printf("<x,y> = %d\n", dotproduct(3, x, y));
    printf("<x,z> = %d\n", dotproduct(3, x, z));

    dlclose(handle);
    return 0;
}
```

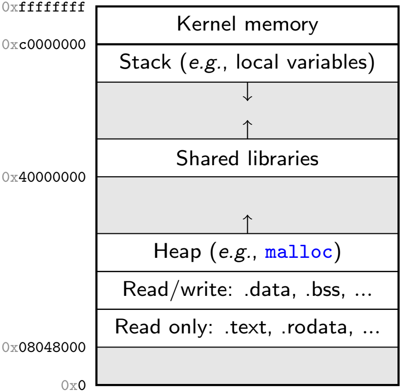
```
$ ./sick ./libvector.so
<x,y> = 0
<x,z> = 1
$ ./sick ./libfake.so
<x,y> = 42
<x,z> = 42
```

- Some projects use this mechanism to provide a **plugin interface**.
- If the program is compiled with `-rdynamic`, then a loaded library can use the program's global symbols.

Position-independent code

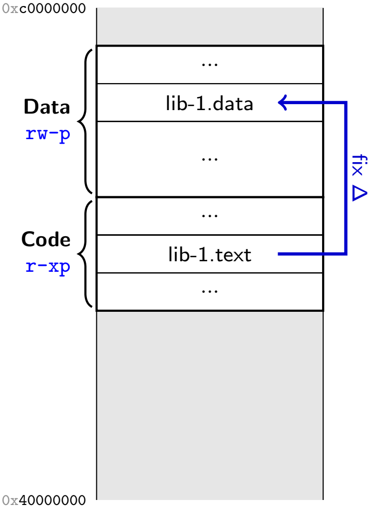
- **Different programs** should use a shared library simultaneously.
- Mapping is likely to happen to **different virtual memory regions**.
- Simple **relocation breaks sharing**, since it modifies `.text`.

How can we solve this?



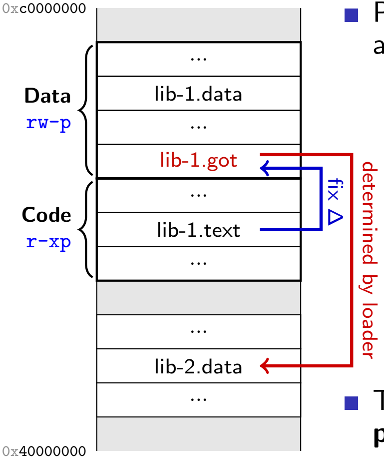
- Recall process memory layout:
  - Shared libraries' `.text` and `.data` is **not merged** with the main program's.
  - Instead, each shared library is loaded **somewhere above** `0x40000000`.
- **Position-independent code** is compiled in a way that allows it to be executed at any address, without prior relocation.

Memory layout of a shared library



- The data segment of a shared library is mapped **directly after** the code segment.
  - For each access to a local symbol, the **distance from instruction to variable** is fix!
  - This Δ is known at **compile time**.
  - Variable access is implemented by **offset from the program counter**, instead of absolute addresses.
- How can we access variables in **other modules**?

Memory layout of a shared library



- PIC adds **one level of indirection** to access external symbols.
  - A **global offset table** (GOT) is added at the **start of the data segment** of every module.
  - At compile time, references to variables are replaced by **indirect references** via the GOT.
  - The **dynamic loader fills the GOT** with the correct addresses (relocation) at runtime.
- Thus, relocation **happens in the private data segment**!
  - The code segment can be shared.

Final Remarks

- Relocation for **function calls** also uses the GOT.
  - A more sophisticated algorithm, called **lazy binding**, reduces the overhead after the first function call.
- Shared libraries come with a **runtime overhead** for accessing any external symbols.
- Using shared libraries requires **expensive setup of all GOTs** when loading a program.
- Shared libraries **increase code sharing** (and page sharing) more than static libraries.
  - Static library code **cannot be shared between different programs**, only between different instances of the same program.

- An in-depth discussion about shared libraries can be found here:
  - Ulrich Drepper. *How To Write Shared Libraries.* December 2011, http://www.akkadia.org/drepper/dsohowto.pdf.