

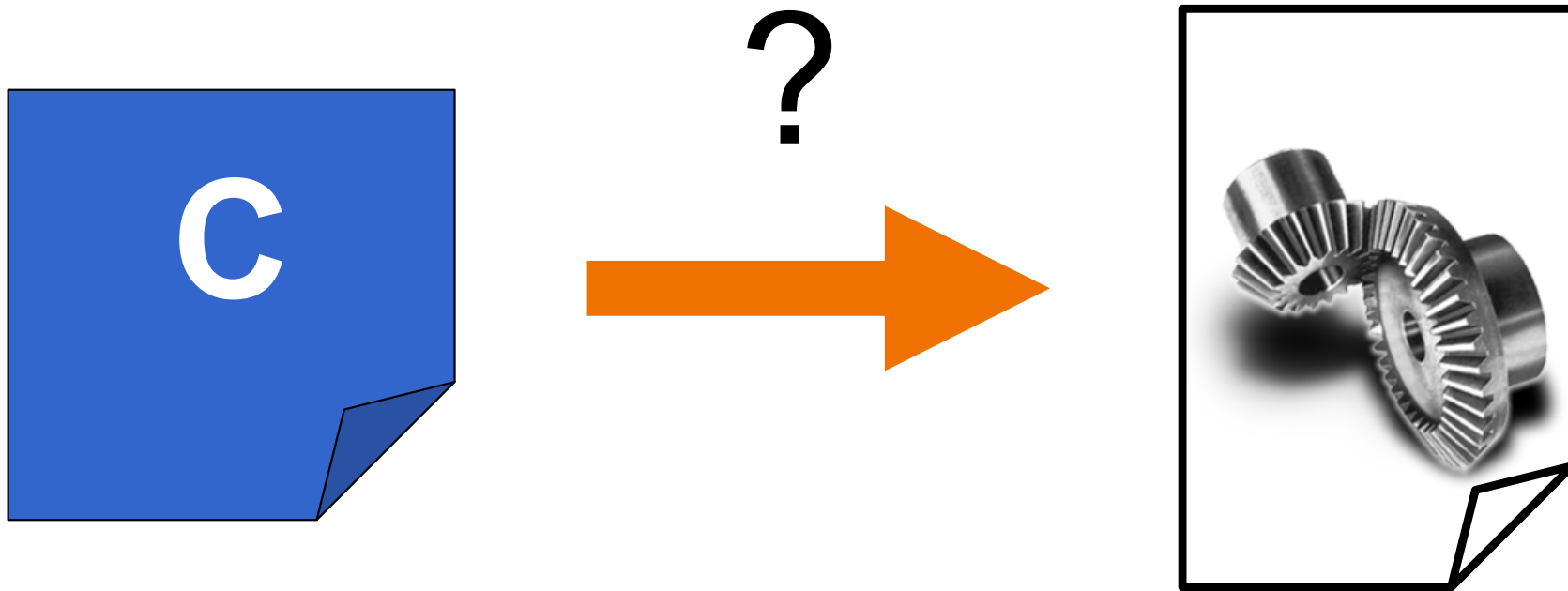
Introduction aux Systèmes d'Exploitation

Unit 10: Linkers and Loaders and Dynamic Libraries

François Taïani



Source code to executable?



- Obviously some machine code produced at some point
 - ➔ But many steps involved

Steps of Executable Creation

■ Preprocessing

- macros, include directives, (#xxxx statements)
- output: “pure” C code

■ Compilation (lexing, parsing, semantic analysis)

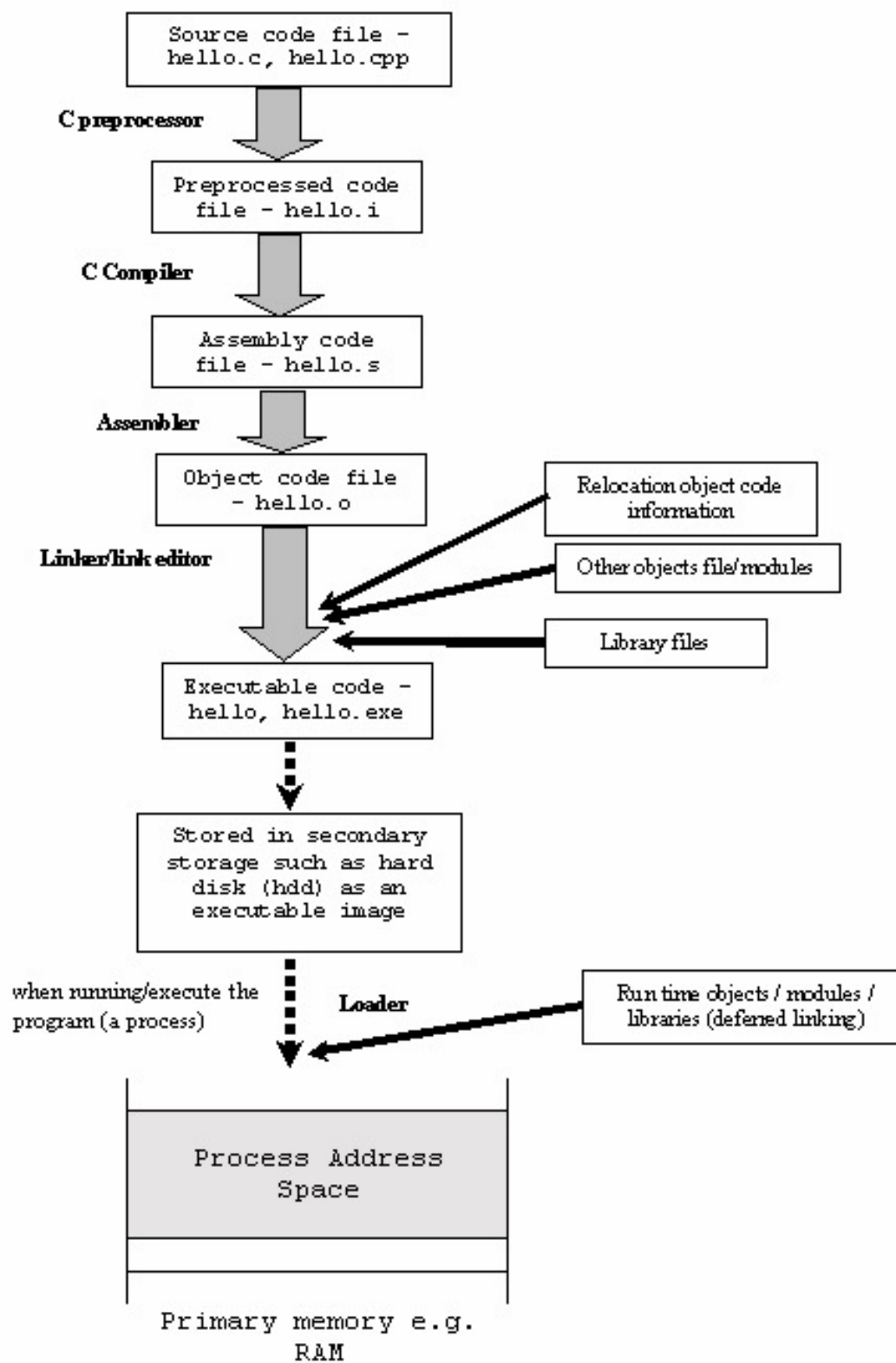
- transform C into actual assembler source code
- not machine code: still human readable
- dependent on machine architecture (!), x86, ARM, ...

■ Assembly

- creates actual machine code, stored in object file (.o)

■ Linking

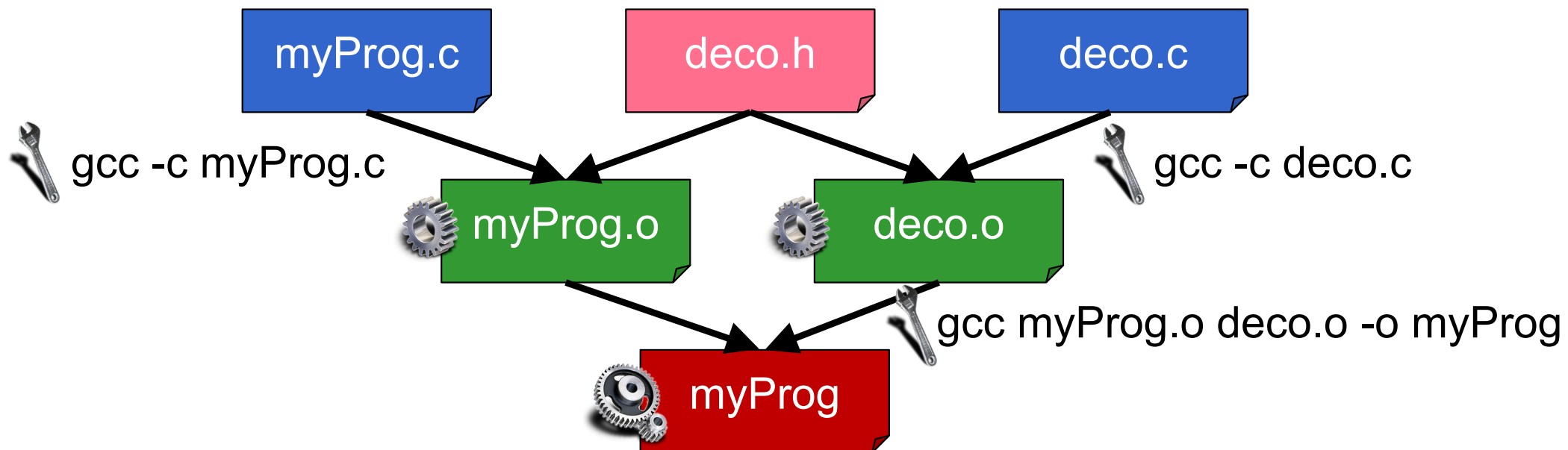
- combining several object files together



Example

```
#include "deco.h"

int main(int argc, char *argv[])
{
    if (argc < 2) return 0 ;
    print_message_with_decoration(argv[1]);
}
```



What is in an Object file?

- When program compiled **in several parts**:
 - myProg.c calls a functions in deco.c
 - myProg.o needs a jump to code in deco.o
- In mProg.o: **actual address** of jumps cannot be decided
 - a) put on 'stand-by' until known
 - b) but needs a way to remember address is not resolved
 - c) needs to be resolved to build executable
- a) and b) achieved with symbol tables
- c) is the job of the linker! (ld on Linux, called by gcc)

What is in an Object file?

- Main formats (all originated from Unix world):
 - ELF: Executable and Linking Format (Linux)
 - COFF: Common Object-File Format (Windows)
 - Mach-O: Mac OS X (Mach Kernel)
- Object file:
 - machine code of program (known as “**text**” section)
 - data (global constant) (aka “**data**” section)
 - how much space for uninitialised data (“**bss**”)
 - symbol tables (where is function x)
 - relocation information (what to modify when linking)

Looking at object files

- Option 1: binary or hexadecimal dump of o file
 - you need to know your ELF format extremely well
 - e.g. hexdump myProg.o
- Option 2: use tools!
 - to look at symbols: nm (for '*name* list')
 - e.g. nm myProg.o

```
$ nm myProg.o
00000000000000000000 U _GLOBAL_OFFSET_TABLE_
00000000000000000000 T main
U print_message_with_decoration
$ nm deco.o
U _GLOBAL_OFFSET_TABLE_
U printf
00000000000000000000 T print_message_with_decoration
U puts
```

What nm tells you

- ‘U’ undefined symbol
 - the linker will need to find it somewhere
- ‘T’ external text symbol
 - a function implemented in this object file
 - available externally (from other o file, or OS)
- ‘t’ internal text symbol
 - same as above, but not available externally
- ‘D’ external data symbol (‘d’ internal data)
 - in initialised data section
- *etc.*

What nm tells you

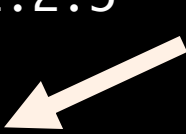
- First column: value of symbol
 - where the symbol is located in the object image
- With -l, --line-numbers: where in code source
 - requires -g option, or information not put in object file
- Note on example how
 - `print_message_with_decoration` Undefined in `myProg.o`
 - `print_message_with_decoration` defined (T) in `deco.o`

What happens to object files?

- Either linked into an **executable**
 - `gcc -g deco.o myProg.o -o myProg`
- Or into a **library**
 - **static**: `ar -rs libdeco.a deco.o`
 - **dynamic**: `gcc -fPIC -shared deco.o -o libdeco.so`
 - note: usually several object files in a library (here 1)
- You can apply **nm** to all these files!

Still some undefined symbols!

```
$ nm myProg
0000000000004038 B __bss_start
0000000000004038 b completed.0
                  w __cxa_finalize@GLIBC_2.2.5
0000000000004028 D __data_start
0000000000004028 W data_start
...
0000000000004000 d _GLOBAL_OFFSET_TABLE_
...
                  U __libc_start_main@GLIBC_2.2.5
0000000000001184 T main
                  U printf@GLIBC_2.2.5
0000000000001145 T print_message_with_decoration
                  U puts@GLIBC_2.2.5
...
```



now defined

Still some undefined symbols!

```
$ nm myProg
0000000000004038 B __bss_start
0000000000004038 b completed.0
                w __cxa_finalize@GLIBC_2.2.5
0000000000004028 D __data_start
0000000000004028 W data_start
...
0000000000004000 d _GLOBAL_OFFSET_TABLE_
...
                U __libc_start_main@GLIBC_2.2.5
0000000000001184 T main
                U printf@GLIBC_2.2.5
0000000000001145 T print_message_with_decoration
                U puts@GLIBC_2.2.5
...
```

Positions have changed. No longer zero



Still some undefined symbols!

```
$ nm myProg
0000000000004038 B __bss_start
0000000000004038 b completed.0
                                w __cxa_finalize@GLIBC_2.2.5
0000000000004028 D __data_start
0000000000004028 W data_start
...
0000000000004000 d _GLOBAL_OFFSET_TABLE_
...
                                U __libc_start_main@GLIBC_2.2.5
0000000000001184 T main
                                U printf@GLIBC_2.2.5
0000000000001145 T print_message_with_decoration
                                U puts@GLIBC_2.2.5
...
```

gcc legwork

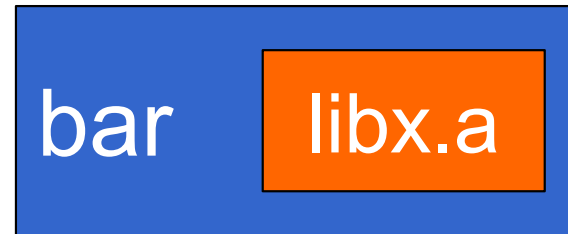
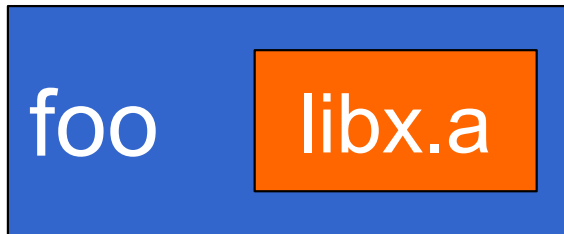
shared library calls

The diagram illustrates the relationship between the symbols listed in the nm output and their origins. A large white curly brace on the right side of the output, spanning from the first line to the line containing `U __libc_start_main@GLIBC_2.2.5`, is labeled "gcc legwork". Two white arrows point from the text "shared library calls" at the bottom right to the two undefined symbols: `U printf@GLIBC_2.2.5` and `U puts@GLIBC_2.2.5`.

Why shared libraries?

■ Static lib

- an archive of object file
- a copy included in each executable



Replicated on the hard disk

Why shared libraries?

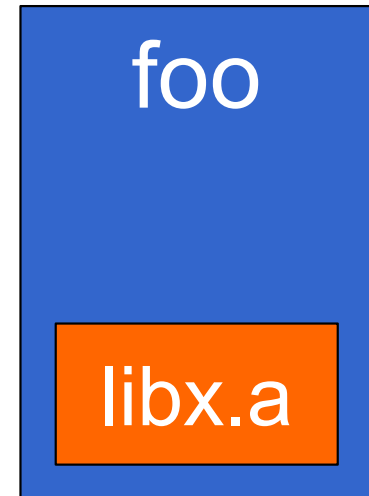
■ Static lib

- an archive of object file
- a copy included in each executable

■ Waste of space

- some libraries in all executable: e.g. libc
- physical memory often bottleneck

Replicated in memory

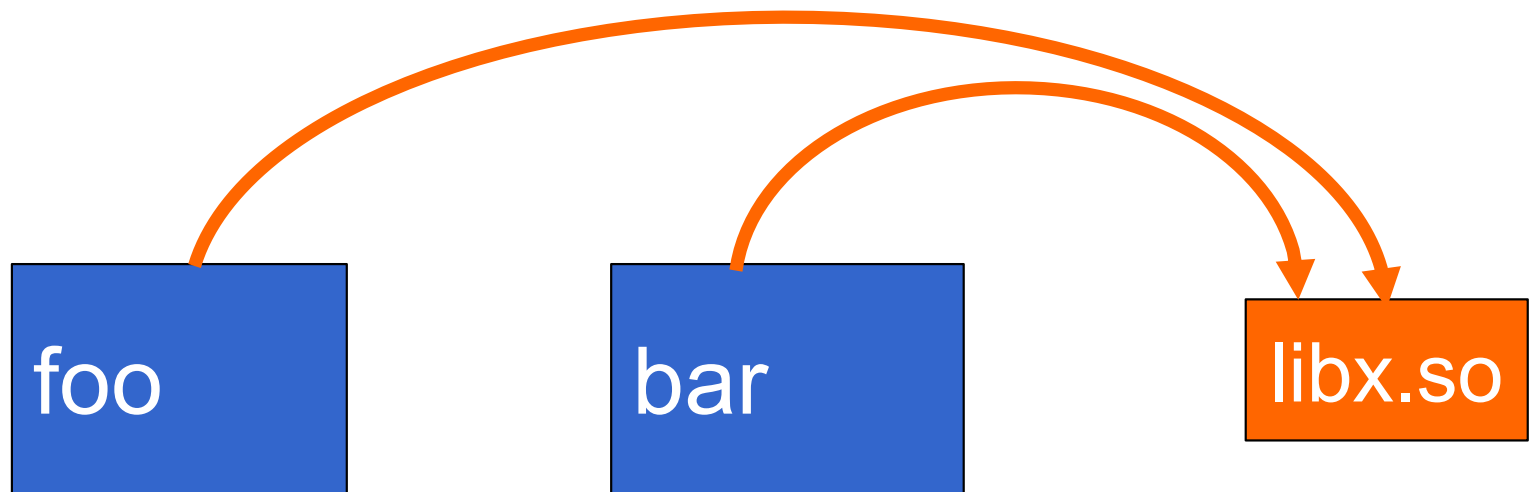


Shared Libraries

- Idea: only load once, use many times
- Supported on all modern OSs
 - Linux / Unix: *.so (for shared object)
 - Windows: *.dll (dynamically linked library)
 - Mac OS: *.dylib (for dynamic library)
- Downside
 - more bookkeeping/complexity by OS
 - require HW support (MMU) for efficient implementation
 - often not available on highly constrained systems

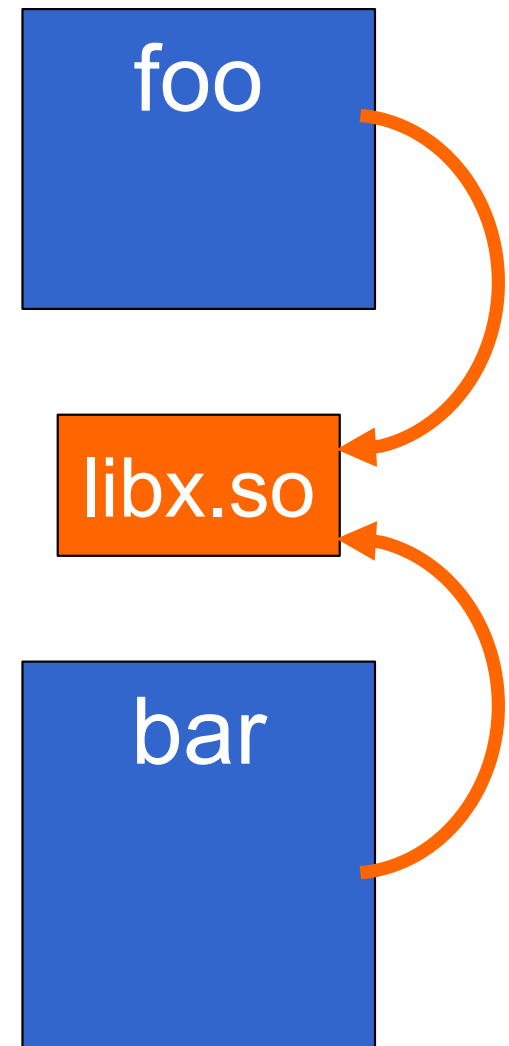
Shared Libraries: Principle

- On the hard drive



Shared Libraries: Principle

- In memory (simplified)
- Additional complexity
 - each process isolated in memory
foo cannot see' s bar memory
 - only code (text) of libx shared
 - each process: own libx data section
 - processes use “virtual” memory:
no access to real physical memory
 - as a result: libx can have different
“virtual” addresses for each processes
- All of this handled by OS and MMU
 - in depth treatment in OS module



Example of shared libraries

- e.g. in /lib: system libraries

```
$ ls *.so
ld-2.31.so                libmemusage.so           librt-2.31.so
liba52-0.7.4.so           libm.so                  librt.so
libanl-2.31.so            libmvec-2.31.so          libSegFault.so
libanl.so                libmvec.so              libshotwell-authenticator.so
libbfd-2.35.2-system.so   libns-9.16.15-Debian.so libshotwell-plugin-common.so
libbind9-9.16.15-Debian.so libnsl-2.31.so            libshotwell-plugin-dev-1.0.so
libBrokenLocale-2.31.so   libnsl.so               libsmime3.so
libBrokenLocale.so       libnspr4.so              libSM.so
libc-2.31.so             libnss3.so               libssl3.so
libcrypt.so             libnss_compat-2.31.so    libthread_db-1.0.so
libc.so                  libnss_compat.so        libthread_db.so
libdb-5.3.so             libnss_dns-2.31.so       libtirpc.so
libdl-2.31.so            libnss_dns.so          libutil-2.31.so
libdl.so                libnss_files-2.31.so     libutil.so
...
```

- note how minor versions handled with symbolic links allow implementation to change without recompiling
- locations vary depending on distribution, OS, etc.

Looking inside a shared library

■ `nm -D libncurses.so.5 | less`

```
$ nm -D libncurses.so.6 | head -20
          U acs_map@NCURSES6_TINFO_5.0.19991023
000000000000d380 T addch@@NCURSES6_5.0.19991023
000000000000d3a0 T addchnstr@@NCURSES6_5.0.19991023
000000000000d3c0 T addchstr@@NCURSES6_5.0.19991023
000000000000d3e0 T addnstr@@NCURSES6_5.0.19991023
000000000000d400 T addstr@@NCURSES6_5.0.19991023
000000000001f310 T assume_default_colors@@NCURSES6_5.1.20000708
000000000001f200 T assume_default_colors_sp@@NCURSES6_5.8.20110226
000000000000d480 T attr_get@@NCURSES6_5.0.19991023
000000000000d4c0 T attr_off@@NCURSES6_5.0.19991023
```

Dyn libraries of executable

- Can be done with ldd command

→ e.g. ldd xgalaga

```
$ ldd $(which xgalaga)
linux-vdso.so.1 (0x00007ffdeb793000)
libXxf86vm.so.1 => /lib/x86_64-linux-gnu/libXxf86vm.so.1 (0x00007f54fb436000)
libX11.so.6 => /lib/x86_64-linux-gnu/libX11.so.6 (0x00007f54fb2f3000)
libXpm.so.4 => /lib/x86_64-linux-gnu/libXpm.so.4 (0x00007f54fb0e1000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f54faf1c000)
libXext.so.6 => /lib/x86_64-linux-gnu/libXext.so.6 (0x00007f54faf07000)
libxcb.so.1 => /lib/x86_64-linux-gnu/libxcb.so.1 (0x00007f54faedc000)
libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f54faed4000)
/lib64/ld-linux-x86-64.so.2 (0x00007f54fb6c2000)
libXau.so.6 => /lib/x86_64-linux-gnu/libXau.so.6 (0x00007f54faecf000)
libXdmcp.so.6 => /lib/x86_64-linux-gnu/libXdmcp.so.6 (0x00007f54facc9000)
libbsd.so.0 => /lib/x86_64-linux-gnu/libbsd.so.0 (0x00007f54facb2000)
libmd.so.0 => /lib/x86_64-linux-gnu/libmd.so.0 (0x00007f54faca5000)
```

Using dynamic libs

- During building process at (static) linking phase

```
#include <ncurses.h>
```

```
int main()  
{  
    initscr();                /* Start curses mode          */  
    printw("Hello World !!!"); /* Print Hello World         */  
    refresh();                /* Print it on to the real screen */  
    getch();                  /* Wait for user input         */  
    endwin();                 /* End curses mode             */  
  
    return 0;  
}
```

→ gcc -c ncurses_example.c

→ gcc ncurses_example.o -lncurses

Notes

- To compile previous program
 - ncurses.h header needed
 - can be obtained with `sudo apt-get install libncurses-dev`
 - why: C compiler needed to know signature of methods
 - but headers not installed by default
- Dev header usually called something-dev
 - e.g. `apt-cache search ncurses-dev`
- **`gcc ncurses_example.o -lncurses`**
 - this is the linking stage
 - note how ncurses library passed to gcc

Looking at the result

- `nm -u a.out` # only undefined symbols

```
w __cxa_finalize@GLIBC_2.2.5
U endwin@NCURSES6_5.0.19991023
w __gmon_start__
U initscr@NCURSES6_5.0.19991023
w _ITM_deregisterTMCloneTable
w _ITM_registerTMCloneTable
U __libc_start_main@GLIBC_2.2.5
U printf@NCURSES6_5.0.19991023
U wgetch@NCURSES6_5.0.19991023
U wrefresh@NCURSES6_5.0.19991023
```

- `ldd a.out`

```
linux-vdso.so.1 (0x00007ffc1a9cc000)
libncurses.so.6 => /lib/x86_64-linux-gnu/libncurses.so.6 (...)
libtinfo.so.6 => /lib/x86_64-linux-gnu/libtinfo.so.6 (0x00007f1...)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f179bff1000)
libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f179bfef000)
/lib64/ld-linux-x86-64.so.2 (0x00007f179c22b000)
```

What happens during execution?

- Executable still contains undefined symbols
 - these must be resolved for executable to work
 - this is the work of the **dynamic linker / loader** ld-linux.so
 - both a shared lib & an executable
- When launching a program that uses shared libs **ld**
 - finds the shared libraries needed by the program
 - prepares the program to run
 - then runs it

Looking at the result

- launch a.out
- ctr + Z to suspend it
- **pmap -d <PID>** to view memory mapping of process
 - ➔ alternative cat /proc/PID/maps

Result

Address	Kbytes	Mode	Offset	Device	Mapping
000055e433e25000	4	r----	0000000000000000	000:00025	ncurse_example
000055e433e26000	4	r-x--	0000000000000100	000:00025	ncurse_example
000055e433e27000	4	r----	0000000000000200	000:00025	ncurse_example
000055e433e28000	4	r----	0000000000000200	000:00025	ncurse_example
000055e433e29000	4	rw---	0000000000000300	000:00025	ncurse_example
000055e434e3c000	132	rw---	0000000000000000	000:00000	[anon]
00007f31a0c87000	8	rw---	0000000000000000	000:00000	[anon]
...					
00007f31a0c8f000	148	r----	0000000000000000	008:00001	libc-2.31.so
00007f31a0cb4000	1324	r-x--	0000000000002500	008:00001	libc-2.31.so
...					
00007f31a0e83000	32	r----	0000000000000000	008:00001	libncurses.so.6.2
00007f31a0e8b000	100	r-x--	0000000000000800	008:00001	libncurses.so.6.2
00007f31a0ea4000	24	r----	0000000000002100	008:00001	libncurses.so.6.2
00007f31a0eaa000	4	-----	0000000000002700	008:00001	libncurses.so.6.2
00007f31a0eab000	4	r----	0000000000002700	008:00001	libncurses.so.6.2
00007f31a0eac000	4	rw---	0000000000002800	008:00001	libncurses.so.6.2
00007f31a0ead000	8	rw---	0000000000000000	000:00000	[anon]
00007f31a0ec4000	4	r----	0000000000000000	008:00001	ld-2.31.so
00007f31a0ec5000	128	r-x--	0000000000000100	008:00001	ld-2.31.so
00007f31a0ee5000	32	r----	0000000000002100	008:00001	ld-2.31.so
00007f31a0eee000	4	r----	0000000000002900	008:00001	ld-2.31.so
00007f31a0eef000	4	rw---	0000000000002a00	008:00001	ld-2.31.so
00007f31a0ef0000	4	rw---	0000000000000000	000:00000	[anon]
00007ffc8fab3000	132	rw---	0000000000000000	000:00000	[stack]
00007ffc8fb93000	16	r----	0000000000000000	000:00000	[anon]
00007ffc8fb97000	8	r-x--	0000000000000000	000:00000	[anon]
mapped: 2692K writeable/private: 332K shared: 0K					

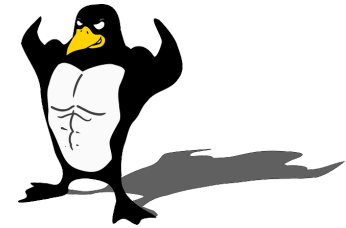
Expected learning outcome

- At the end of this session you should:
 - Be able to explain the main phases of generating an executable from code
 - Be able to justify the need for symbol tables in object files, and explain how they are used by the linker
 - Know how to view the symbols contained in an object file, library, or executable
 - Be able to analyse the addresses present in the disassembly output of a basic program
 - ~~→ Be able to explain the main workings of dlopen and dlsym~~

Expected Learning outcome

- You should be able to explain the difference between shared and static libraries, and the benefit of using shared library
- You should be able to use and compile a program that relies on a shared library
- You should be able to understand the role, and be able to use in simple case the command line tools `ldd`, `nm -D`, and `pmap`

Going deeper



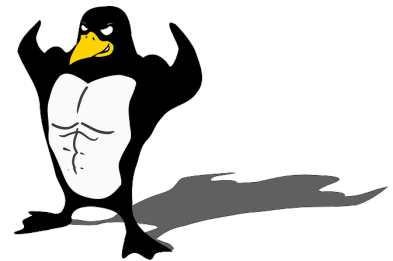
- `objdump -Intel -d myProg | less`
 - ➔ notice how `main`'s code calls `print_message_with_decoration`
- Contrast with `objdump -Intel -d myProg.o | less`
- If you are *really* interested
 - ➔ Disassemble everything: `objdump -S -Intel -Dx myProg`
 - ➔ Look at the **@plt** symbols and their use of the **GOT** table
 - ➔ **PLT**=Procedure Linkage Table, **GOT**=Global Offset Table
 - ➔ Explanations :
<https://eli.thegreenplace.net/2011/11/03/position-independent-code-pic-in-shared-libraries#the-procedure-linkage-table-plt>

References

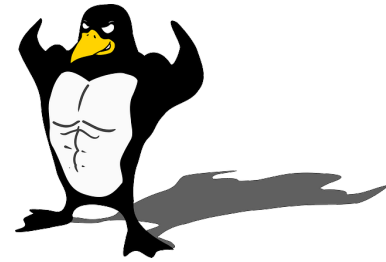
- **Linkers and Loaders** by John R. Levine
→ <http://linker.iecc.com/> (book available for free)
- **Compiler, Assembler, Linker & Loader: A Brief Story**
→ <http://www.tenouk.com/ModuleW.html>
- **Program Library HOWTO**, by David A. Wheeler (2003)
→ <http://tldp.org/HOWTO/Program-Library-HOWTO/>
- **Linux Commands For Shared Library Management**
→ <http://www.cyberciti.biz/tips/linux-shared-library-management.html>
- **Position Independent Code (PIC) in shared libraries**
→ <https://eli.thegreenplace.net/2011/11/03/position-independent-code-pic-in-shared-libraries>

Extra: Manipulating dynamic libs

- Dynamic libraries can be manipulated at runtime
 - Advanced use
- Two main functions:
 - `dlopen` : loads a shared library not declared at compilation
 - `dlsym` : find a symbol in a shared library
- Usage
 - Plug-ins (see labs)
 - Wrappers (intercepting calls to standard libraries)
 - Reflective programming (see `java.lang.reflect`)



Example



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

```
void lib_init() {
    printf("-> my_dynamic_library has just been loaded.\n");
}
```

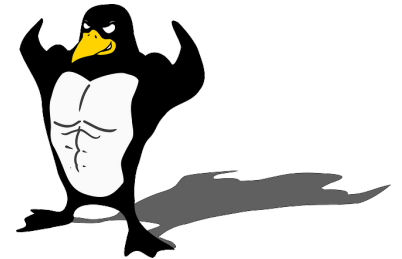
```
void lib_fini() {
    printf("-> my_dynamic_library about to be unloaded. Bye!\n");
}
```

```
void print_welcome_message() {
    printf("-> This is a new dawn; this is a new day!\n");
}
```

my_dynamic_library.c

Compiling the library

- `gcc -fPIC -shared -Wl,-init,lib_init -Wl,-fini,lib_fini my_dynamic_library.c -o libmy_dynamic_library.so`
 - ➔ **-fPIC** : position independent code (needed for dyn libs)
 - ➔ **-shared** : tell linker to generate a shared object (.so)
 - ➔ **-Wl** : what follows is passed as option to the linker
 - ➔ **-Wl,-init,lib_init** : lib_init executed when lib first loaded
 - ➔ **-Wl,-fini,lib_fini** : lib_fini executed when lib unloaded



- Notes:
 - ➔ **-Wl,-init** and **-Wl,-fini** are optional
 - ➔ Usually in two steps (object files '.o', and then '.so')

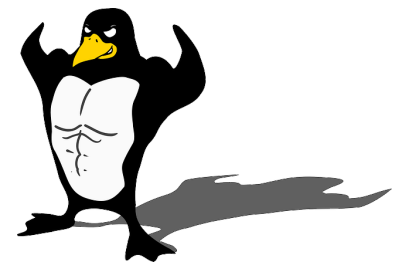
Using the library with dlopen

```
void* (*func_pointer)() ; // a pointer to a function

printf("** %s has started.\n", argv[0]);

void *handle = dlopen("./libmy_dynamic_library.so", RTLD_LAZY);
func_pointer = dlsym(handle, "print_welcome_message");
func_pointer();
dlclose(handle); // unloading 'libmy_dynamic_library.so'

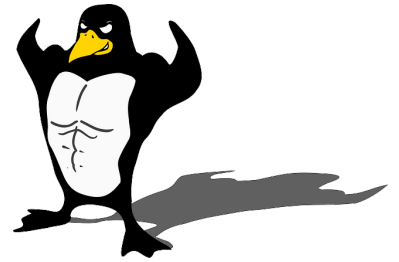
printf("** %s is about to end.\n", argv[0]);
```



■ Note:

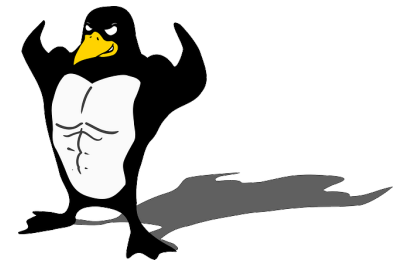
➔ error checking code removed for readability

Explanations



- `void* (*func_pointer)();`
 - ➔ A pointer to a function whose signature is `void* f()`
- `dlopen("./libmy_dynamic_library.so", RTLD_LAZY);`
 - ➔ Loads `./libmy_dynamic_library.so`
 - ➔ `RTLD_LAZY` : lazy binding symbol ➔ address
- `dlsym(handle, "print_welcome_message");`
 - ➔ Returns address of symbol `"print_welcome_message"`
- `func_pointer();`
 - ➔ Invokes function pointed by the variable `func_pointer`
- `dlclose(handle);`
 - ➔ `./libmy_dynamic_library.so` gets unloaded
 - ➔ Memory is freed

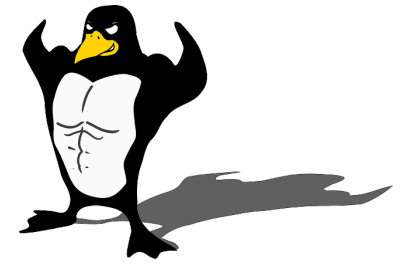
Compilation



- `gcc -ldl my_dynamic_program.c -o my_dynamic_program`
 - ➔ **-ldl** : use the library `libdl.so` (dynamic linker functions)

```
$ ldd my_dynamic_program
    linux-vdso.so.1 => (0x00007fff55556000)
    libdl.so.2 => /lib/libdl.so.2 (0x00007fdfd242e000)
    libc.so.6 => /lib/libc.so.6 (0x00007fdfd20cc000)
    /lib64/ld-linux-x86-64.so.2 (0x00007fdfd2644000)
$ nm -u my_dynamic_program
     w _Jv_RegisterClasses
     w __gmon_start__
     U __libc_start_main@@GLIBC_2.2.5
     U dlclose@@GLIBC_2.2.5
     U dLError@@GLIBC_2.2.5
     U dlopen@@GLIBC_2.2.5
     U dlsym@@GLIBC_2.2.5
     U exit@@GLIBC_2.2.5
     U fprintf@@GLIBC_2.2.5
     U printf@@GLIBC_2.2.5
```

Execution



```
$ ./my_dynamic_program
** ./my_dynamic_program has started.
-> my_dynamic_library has just been loaded.
-> This is a new dawn; this is a new day!
-> my_dynamic_library about to be unloaded. Bye!
** ./my_dynamic_program is about to end.
```

- 'my_dynamic_library'
 - loaded **after** the program has started
 - unloaded **before** the program's end
 - some code of the library executed in between