Shared Libraries in Windows, in Linux, and yes - in C++

OFEK SHILON

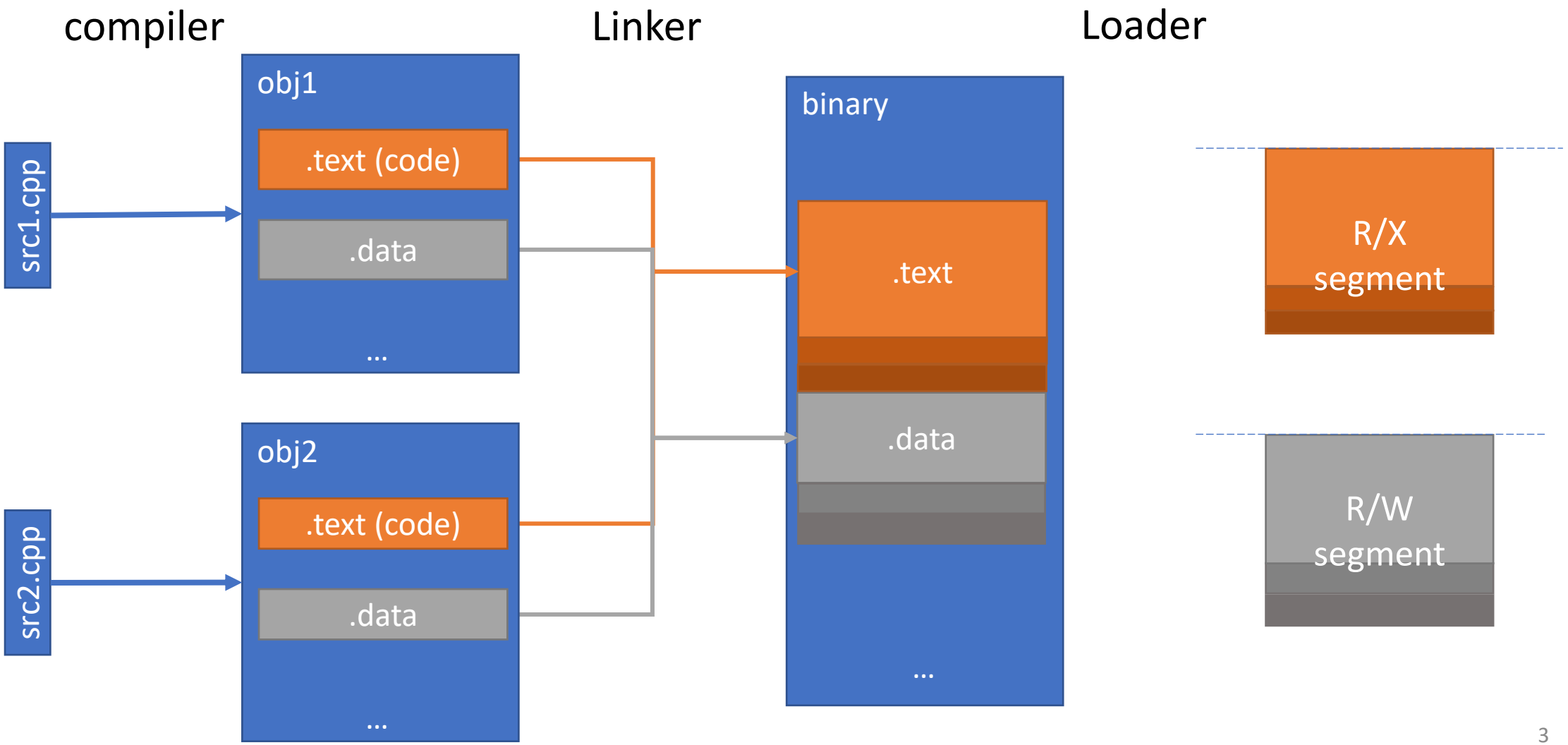
Cppcon
The C++ Conference

2023
October 01 - 06

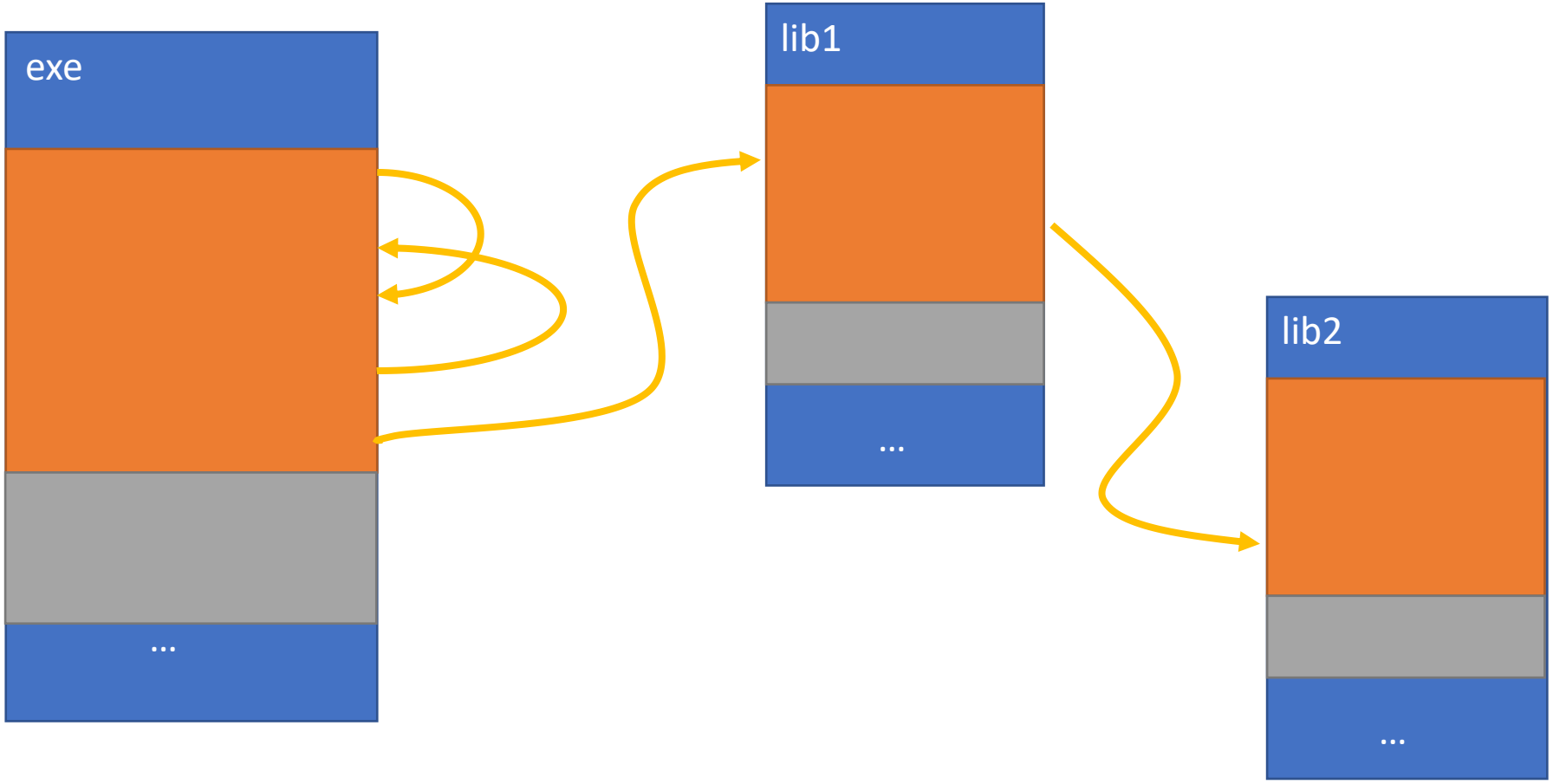
Terminology

- Shared Library:
  - Shared Object, Dynamic Object, Dynamic Shared Object (DSO), Dynamic Load Library (DLL), Dynamic Shared Library
- Binary:
  - Executable / Shared Library , Component, Module
- Symbol:
  - Function / Global variable
- “Linux”:
  - Unix-like systems
  - Mostly MacOS too (1 important distinction to come)

Intro to Linking in 3 slides, #1



Intro to Linking in 3 slides, #2



Intro to Linking in 3 slides, #3

code

```
foo();
...
```

Intro to Linking in 3 slides, #3

code

```
0x1000:    call 0x00000000
           ...
```

.reloc

“Find `foo` and write its address at 0x1001”

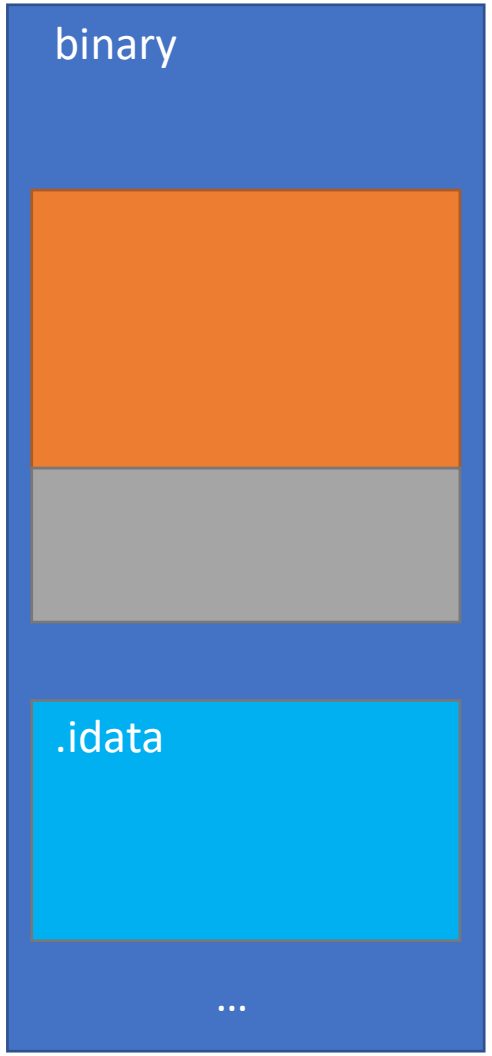
Intro to Linking in 3 slides, #3

Cross-binary calls are typically indirect –
carry a virtual-call overhead

Address-list section:
“IAT” in Windows,
“GOT” in Linux

Windows

Import data section



.idata contents

Directory Table

Directory entry:

- **DLL Name**
- ImportLookupTable offset
- ImportAddressTable offset
- …

Import Lookup Table*

Imported symbol names/ordinals,
for loader usage

…
…
…

* Actually in section .rdata

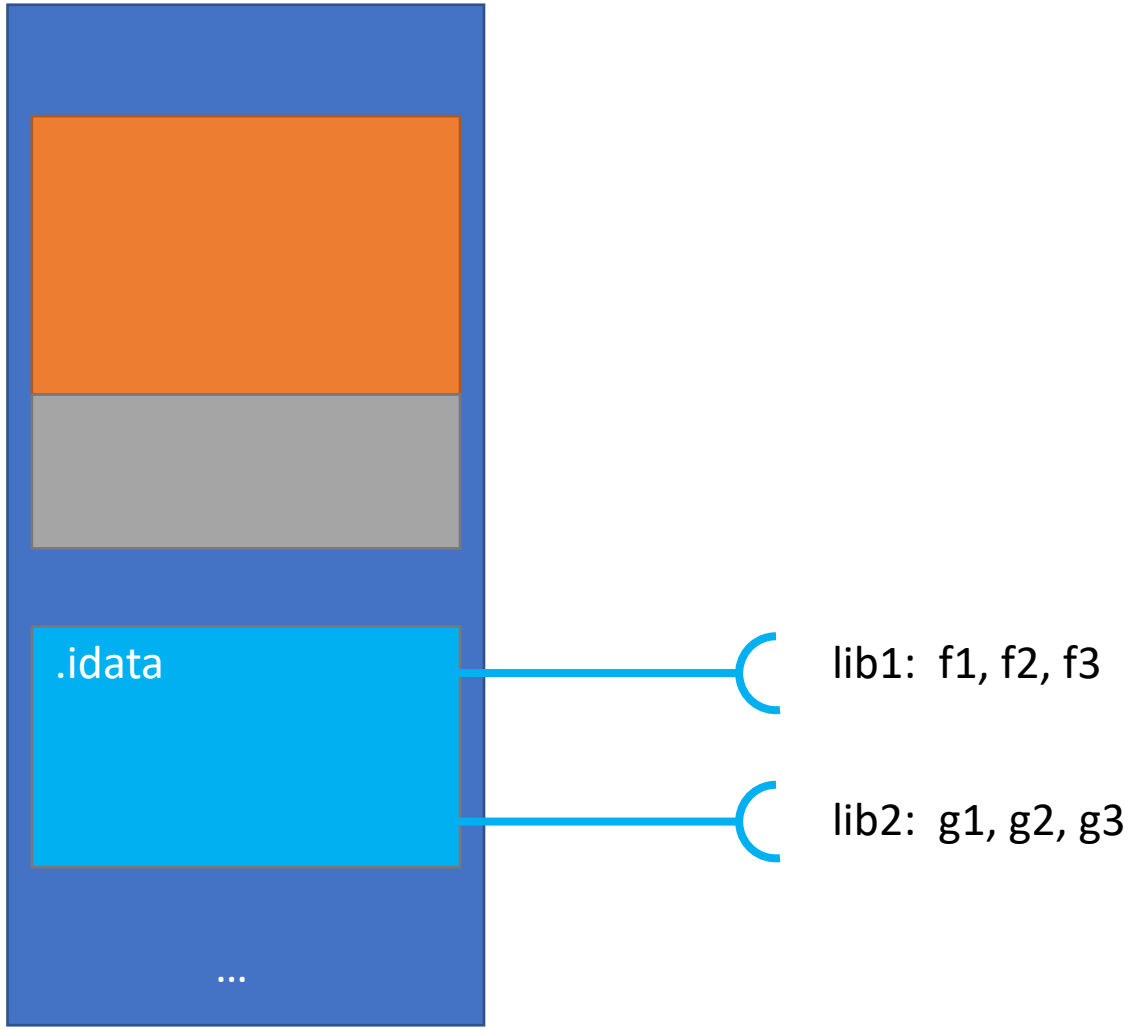
Import Address Table (IAT)

List of import addresses
as resolved by the loader

…
…
…

https://learn.microsoft.com/en-us/windows/win32/debug/pe-format#the-idata-section

Windows Schematic Interface



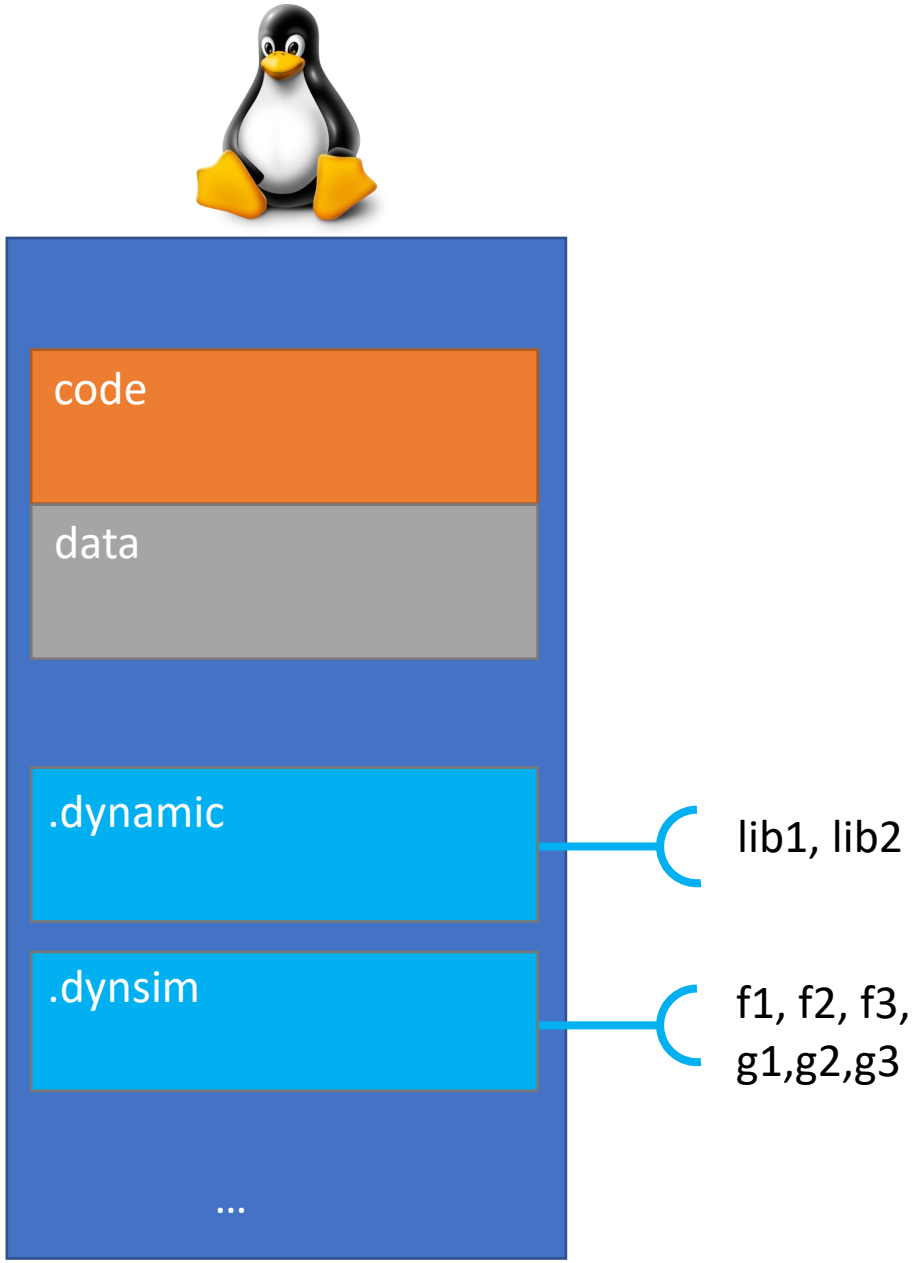
Linux

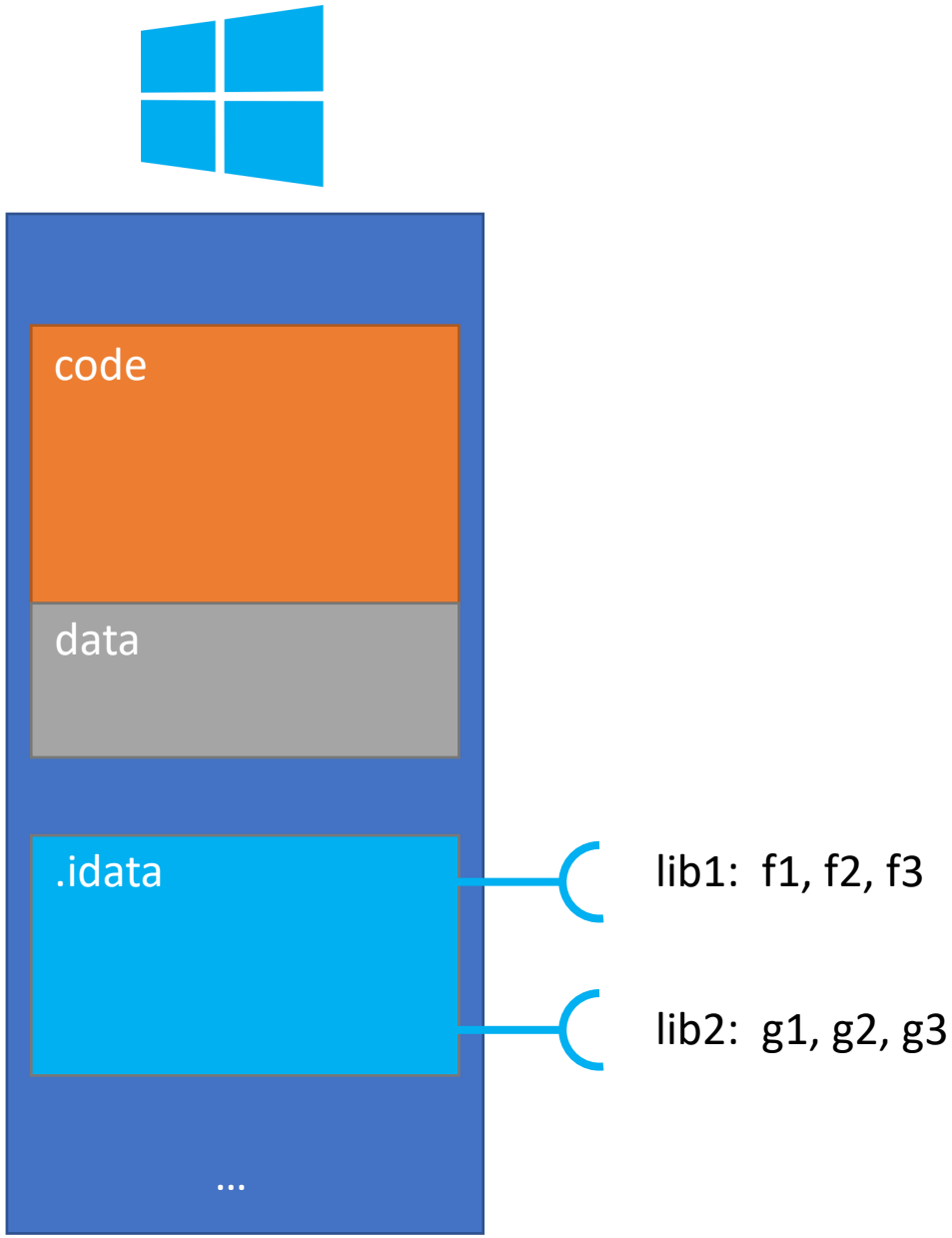
Linux import sections

- .dynamic /.dynsym : separate buckets of lib names and symbol names

```
Dynamic section at offset 0x21a58 contains 28 entries:
  Tag        Type                         Name/Value
 0x0000000000000001 (NEEDED)             Shared library: [libselinux.so.1]
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
```

```
Symbol table '.dynsym' contains 139 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __ctype_toupper_loc@GLIBC_2.3 (2)
     2: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND getenv@GLIBC_2.2.5 (3)
     3: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND sigprocmask@GLIBC_2.2.5 (3)
     4: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __snprintf_chk@GLIBC_2.3.4 (4)
```


code
data
.dynamic
.dynsim
...
lib1, lib2
f1, f2, f3,
g1,g2,g3


code
data
.idata
...
lib1: f1, f2, f3
lib2: g1, g2, g3

Interposition

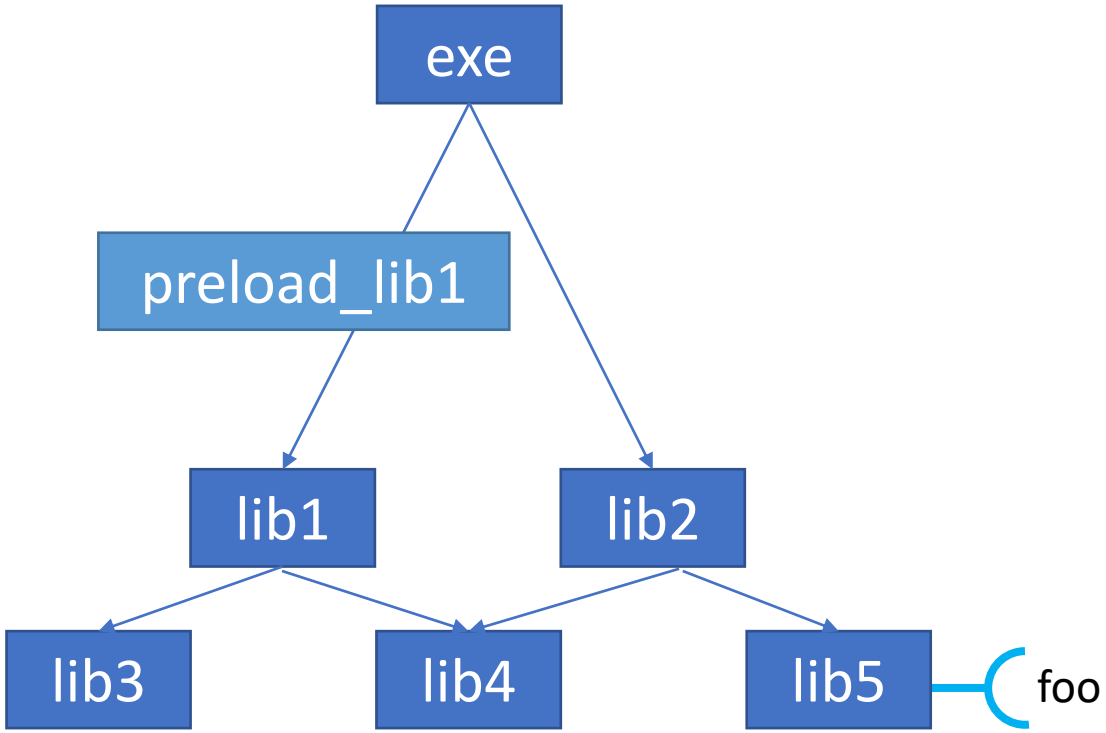
- *Overriding a symbol in one binary from another.*
- Fundamental ABI design pillar!

- Alleged motivation:
  - Mimic the behavior of static libs: first definition shadows the later ones.
  - Not true: in static libs duplicate definitions can cause ODR violation errors.
- Speculated motivation:
  - In the elden days the canonical example of shared-lib was **libc**.
    Users may reasonably wish to override implementations.

Library Search Order

- Breadth-first
- **exe before current lib** (By default)
- To search current lib first:
  - `-Bsymbolic*,`
  - `--dynamic-list*`

- LD_PRELOAD:
  - After exe, before any lib



C++ Implication #1: Can a shared-library symbol be overridden from an executable?

- Windows:
  - No.
- Linux:
  - Yes.
- Mac:
  - Yes, but requires non-default linker switches (eg `-flat_namespace`)

C++: new

[replacement.functions]: A C++ program may provide the definition for any of the following dynamic memory allocation function signatures declared in header <new> :

- operator new(std::size_t)
- operator new(std::size_t, std::align_val_t)
- …

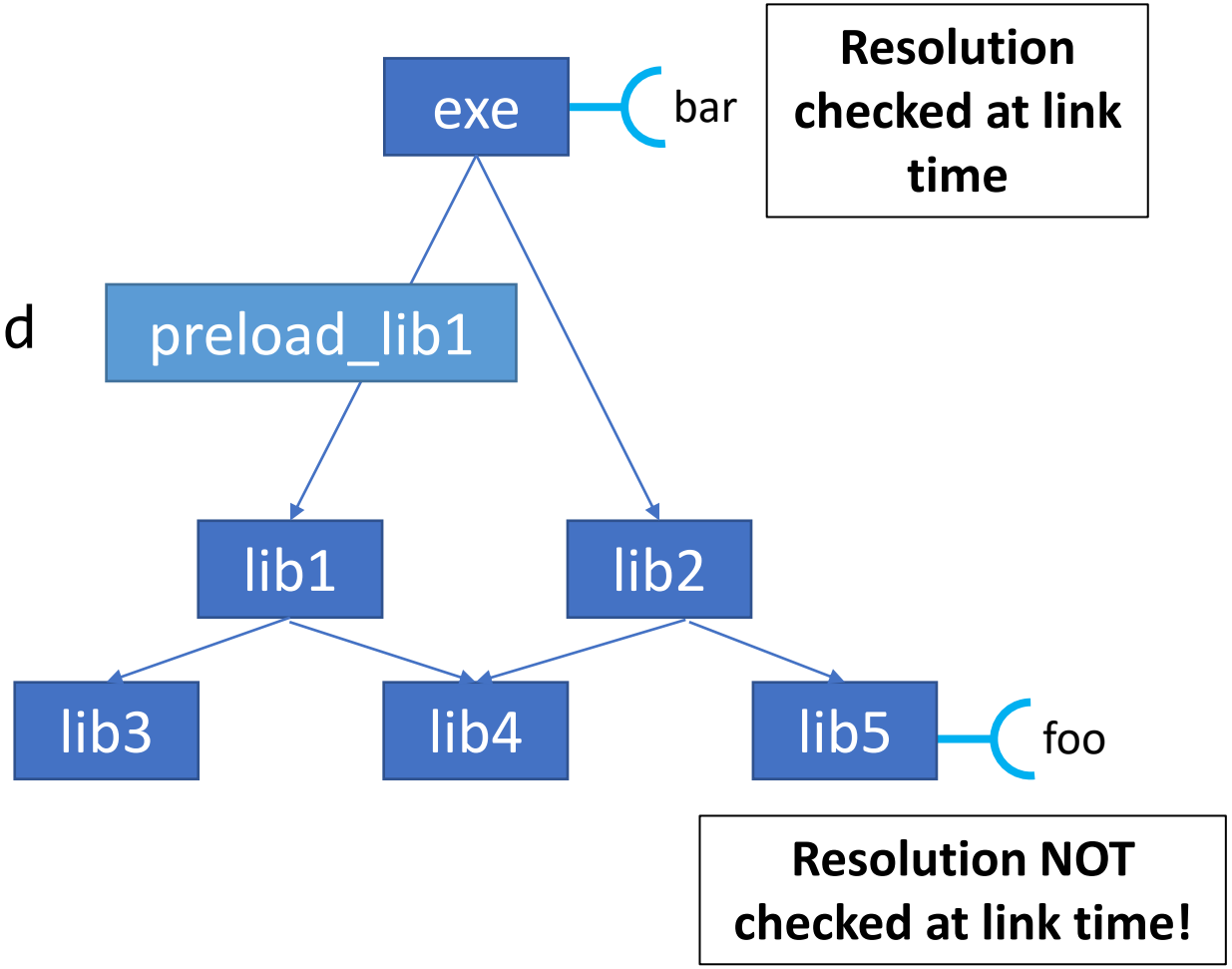
The program's definitions are used instead of the default versions supplied by the implementation …

- *Windows doesn't (can't) do that.*

- Suggestion: drop this clause, add instead library hooks like "set_new_verride()", similar to the existing `set_terminate()` and others.
  - Credit: Thiago Macieira, https://lists.isocpp.org/std-proposals/2023/07/7240.php

Symbol Resolution Time

- Default:
  `--allow-shlib-undefined`
- Can be controlled with –
  - `--no-allow-shlib-undefined` (on the exe)
    - Operates recursively on ld, not on gold/lld.
  - `-z defs` (on the shlib)
  - `--no-undefined` (on the shlib)



C++ Implication #2: How to form a process-wide singleton?

- Singleton: a single object usable by all the code in the process, from all binaries.
- Windows:
  - Usual singleton design patterns would create a *per-binary* singleton
  - Export the singleton variable from a shared-lib, link all your binaries against it.
- Linux:
  - Just put it in the executable

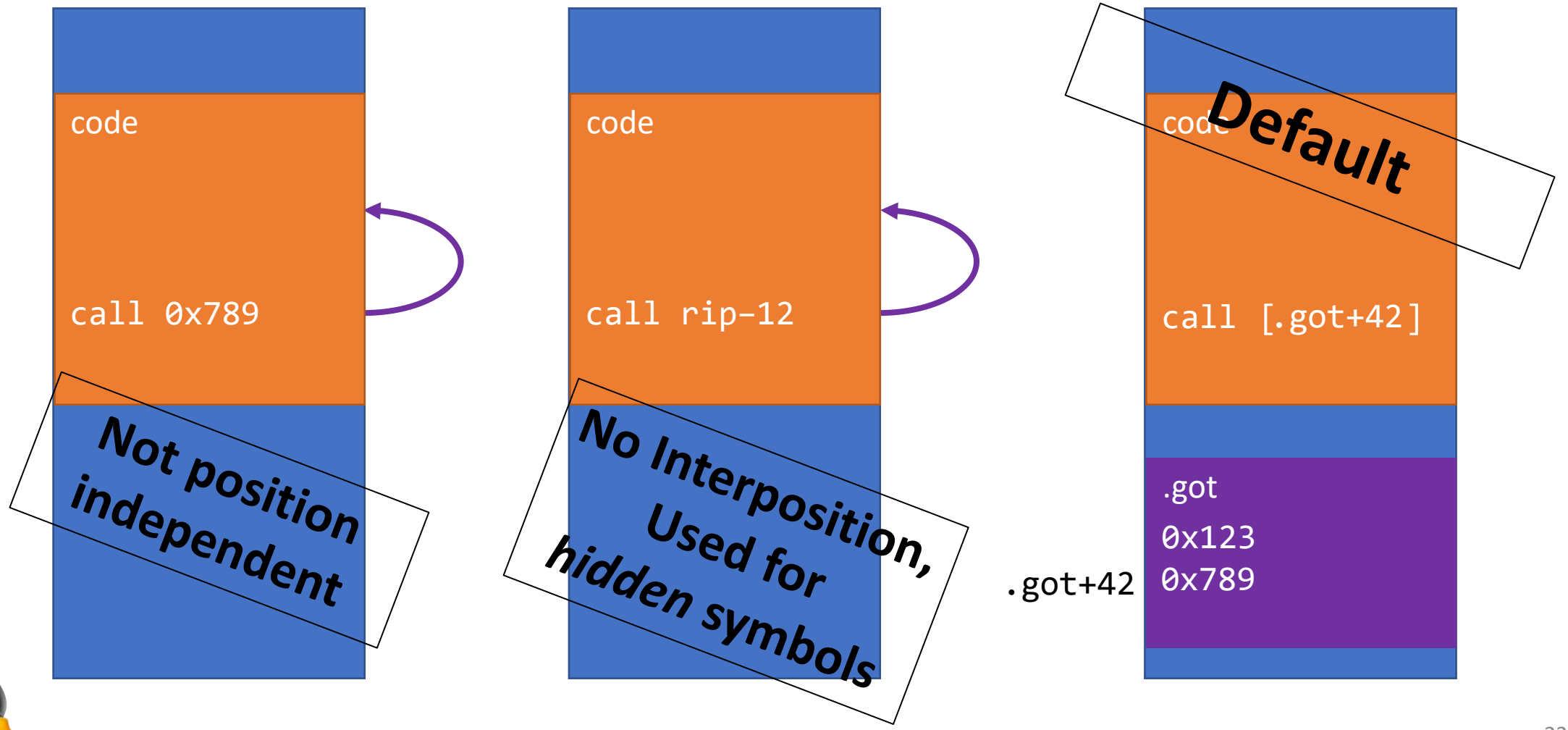
C++ Implication #3: Can you have circular library dependencies?

- Linux:
  - Yes
- Windows
  - No.
  - Well, you'd have to hack hard.

- The Linux design provides some flexibility, but ..
  - "This [allowed-shlib-undefined] is an unfortunate default for -shared. Changing it may be disruptive today. Mach-O and PE/COFF have many problems but this may be a place where they got right."

Fangrui Song, LLD maintainer
https://maskray.me/blog/2021-06-13-dependency-related-linker-options

Position Independent Code: GOT



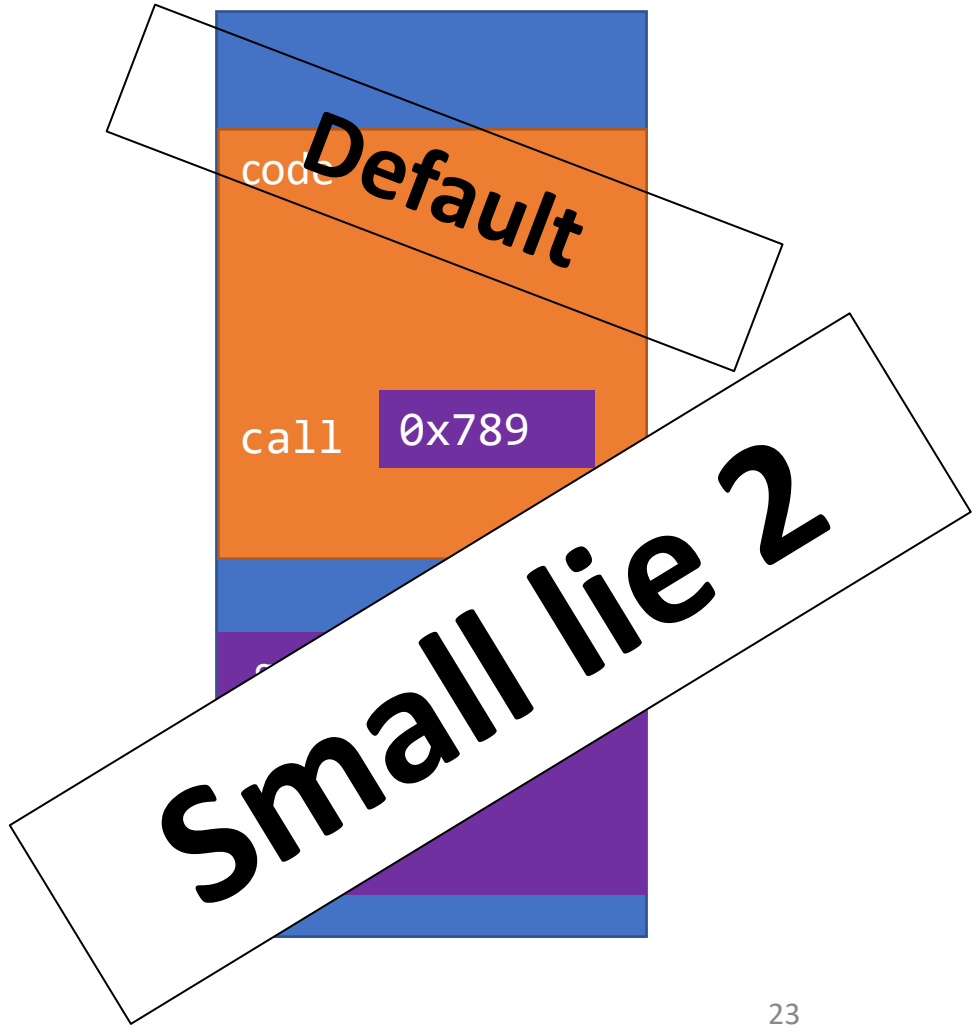
Position Independent Code: GOT

- *For all (default visibility) calls from a shared-lib!*
- This code -

```
void f() { ... }
void g() { f(); }
```

  Will (probably) be inlined if put in an exe, *but not if put in a shared library.**

- All inter-procedural optimizations are inhibited *



- * clang does it anyway…

Position Independent Code - switches

- ***ALL*** shared lib code must be position independent
  - Yet `-fPIC` is optional , not even implicit for `-shared`.
  - If you try to link a shared lib from obj files not built with fPIC *and* using global vars:

```
error: relocation R_X86_64_PC32 against symbol `global' can not be
used when making a shared object; recompile with -fPIC
```

- -fpic vs –fpie:
  - EXE symbols are relocatable but NOT interposable. Interprocedural optimizations apply.

Lazy Binding

(a.k.a Delayed Loading)

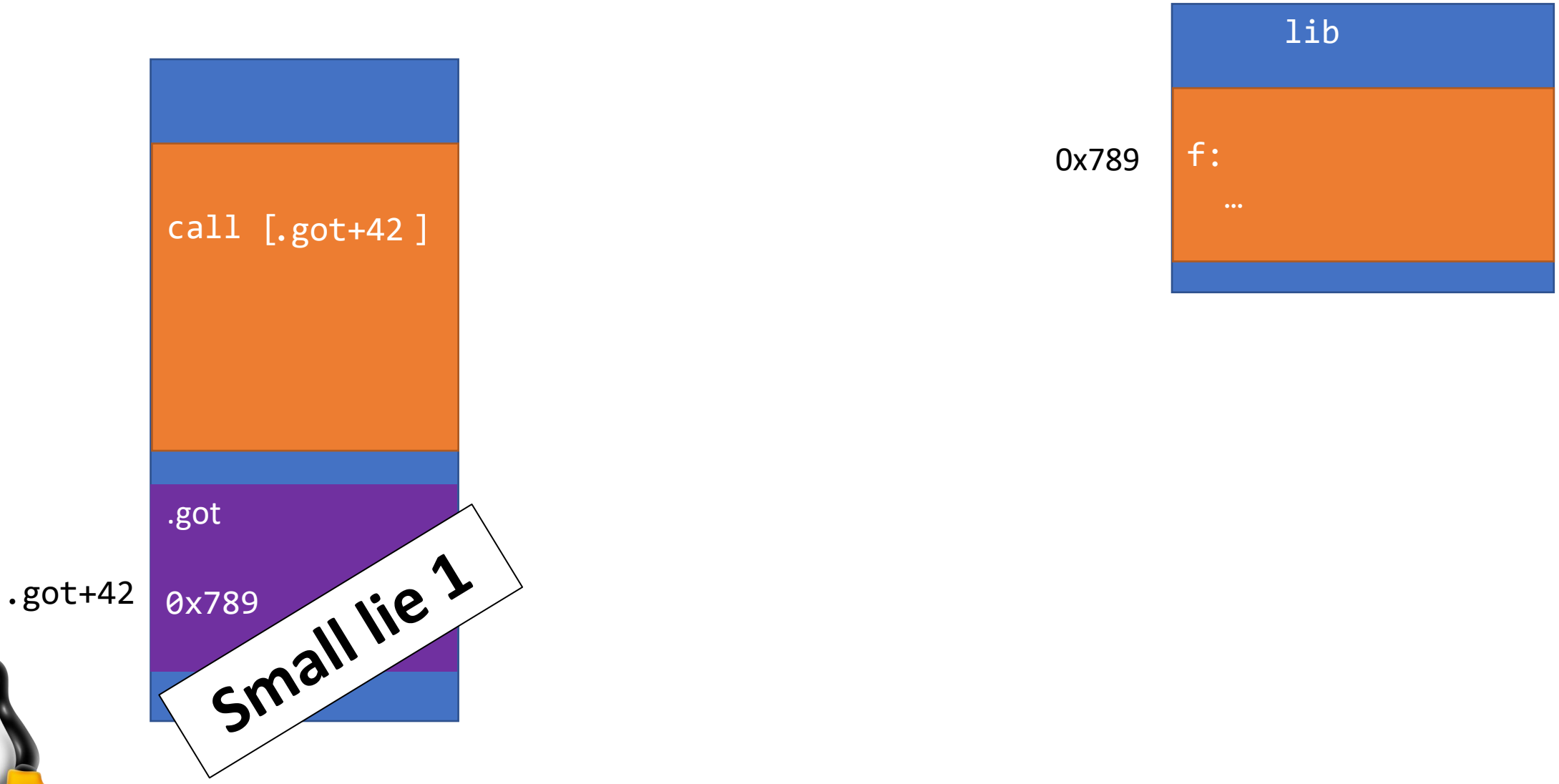
Lazy Bind by Default?

Yes. *

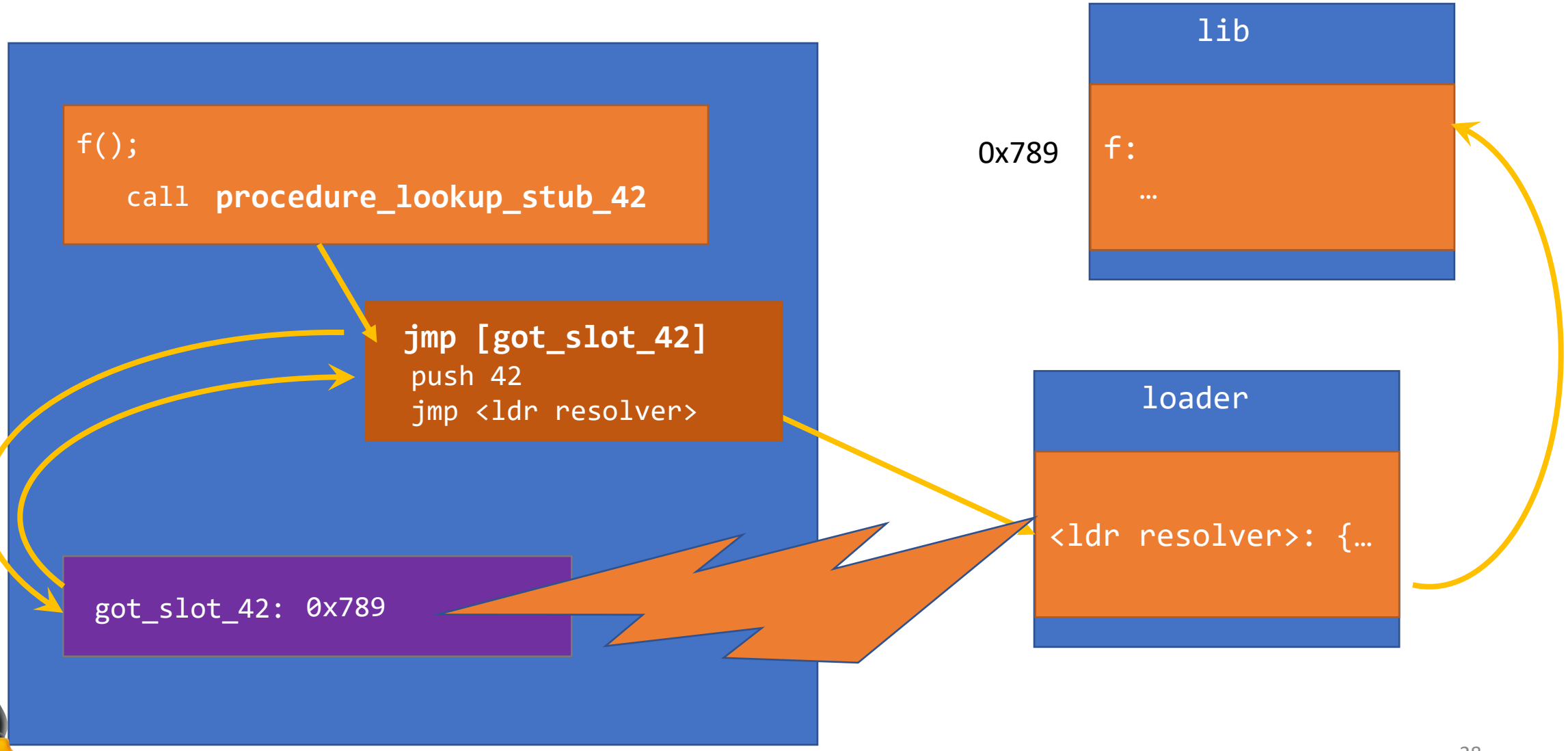
Controllable with linker switches
`-no-plt, -z now,`
function attribute `noplt`
or env var `LD_BIND_NOW`

No.

Controllable with linker switch
`/DELAYLOAD:<your_dll.dll>`


lib
0x789
f:
…
call [.got+42]
.got
.got+42
0x789
Small lie 1

Lazy Binding Design



Lazy Binding Design

```
f();
   call  procedure_lookup_stub_42
```

```
jmp [got_slot_42]
 push 42
 jmp <ldr resolver>
```

```
got_slot_42: 0x789
```

0x789

```
lib

f:
  …
```

```
loader

<ldr resolver>: {…
```

Lazy Binding Design

```
f();
   call  procedure
```

```
…
 jmp [got_slot_41]
  push 42
  jmp <ldr resolver>

 jmp [got_slot_42]
  push 42
  jmp <ldr resolver>

 jmp [got_slot_43]
  push 42
  jmp <
…
```

```
got_slot_42: 0x78
```

Procedure … p Table

Almost Full Truth

PLT: some extra details

- The 1st PLT entry is special and used in all PLT calls, add a module descriptor.
- **`.got`** section used for eager binding (usually global vars), **`.got.plt`** used for lazy binding (functions).
- On modern x86/x64 compilers you'd see *another* level of indirection, through the section `.plt.sec`
  - Added because support for intel security features CET/MPX* didn't fit in 16-byte `.plt` entry scheme.
    (* in a nutshell: branches marked with `bnd` prefix must land on an `endbr64` instruction)
  - Was another section really necessary? Probably a design error…
    https://reviews.llvm.org/D59780#1468080

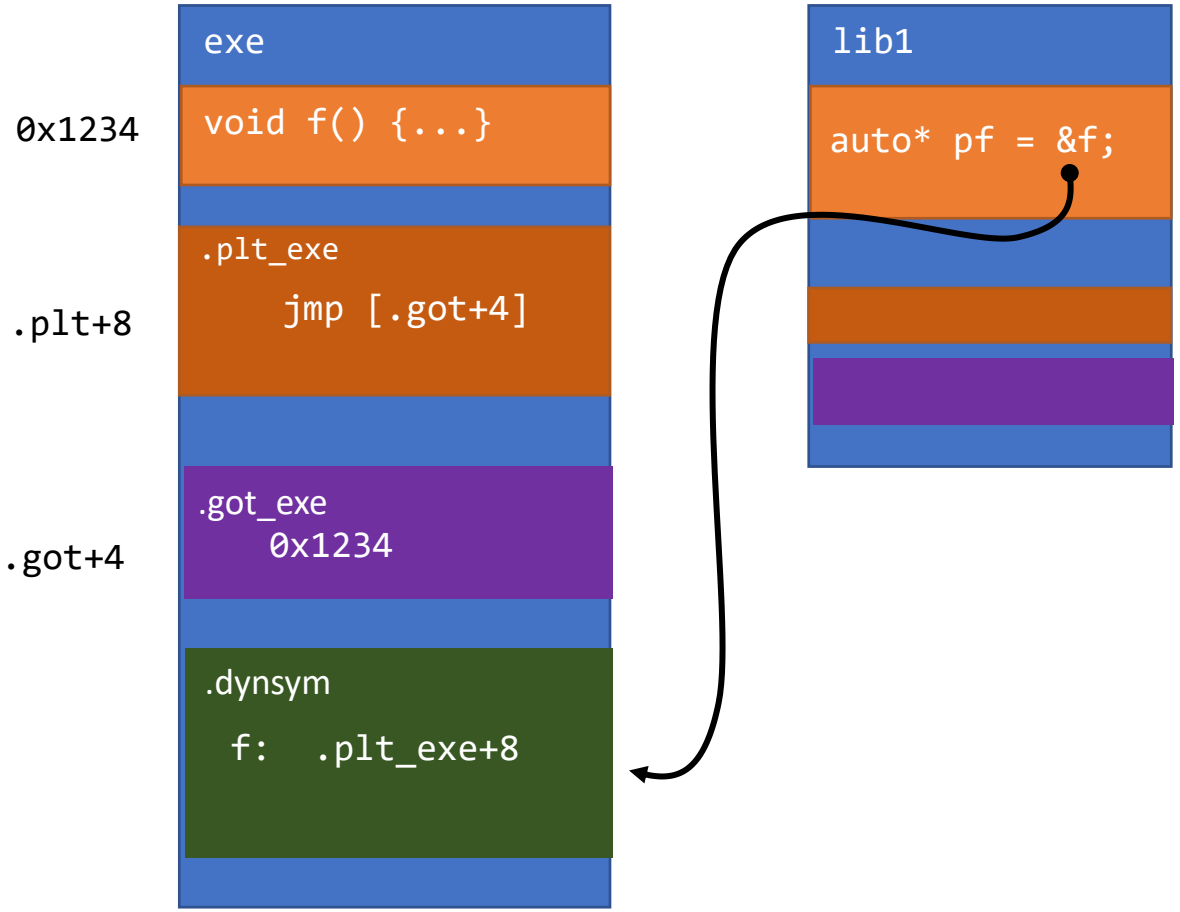
C++ Implication #4: Comparing Func Ptrs

- C++ standard, [expr.eq]§3.2: "… if the pointers are both null, **both point to the same function**, or both represent the same address (6.8.2), they compare equal."
- Actual calls are made to a PLT entry.
- Different among libraries!

C++ Implication #4: Comparing Func Ptrs

From the SystemV ABI doc:

To allow comparisons of function addresses to work as expected, if an executable file references a function defined in a shared object, the link editor will place the address of the procedure linkage table entry for that function in its associated symbol table entry. This will result in symbol table entries with section index of SHN_UNDEF but a type of STT_FUNC and a non-zero st_value. A reference to the address of a function from within a shared library will be satisfied by such a definition in the executable



https://gitlab.com/x86-psABIs/x86-64-ABI/-/jobs/artifacts/master/raw/x86-64-ABI/abi.pdf?job=build

C++ Implication #4: Comparing Func Ptrs

- Not comprehensive:
  - Different for –fpie
  - Different for comparison between pointers both taken at shared libs

- Anyway, Linux tries hard. And improves even today:
  - https://gcc.gnu.org/bugzilla/show_bug.cgi?id=100593#c13
  - (fix from 2023)

Windows doesn’t even try

```
#include <iostream>
using std::cout;

__declspec(dllimport) void delayedLoadFunc();

int main()
{
    cout << &delayedLoadFunc << "\n";
    delayedLoadFunc();
    cout << &delayedLoadFunc << "\n";
}
```

```
Microsoft Visual Studio Debu
00007FF79875149C
00007FFBB25211D1

C:\Users\o00836696\source\repos\testDela
Press any key to close this window . . .
```

Resolution stub

Real function

https://developercommunity.visualstudio.com/t/DELAYLOADs-implementation-breaks-the-C/10458529

Intermediate Summary

- GOT is needed for Interposition
- PLT is needed for Lazy-Binding

  - Both are questionable goals..!

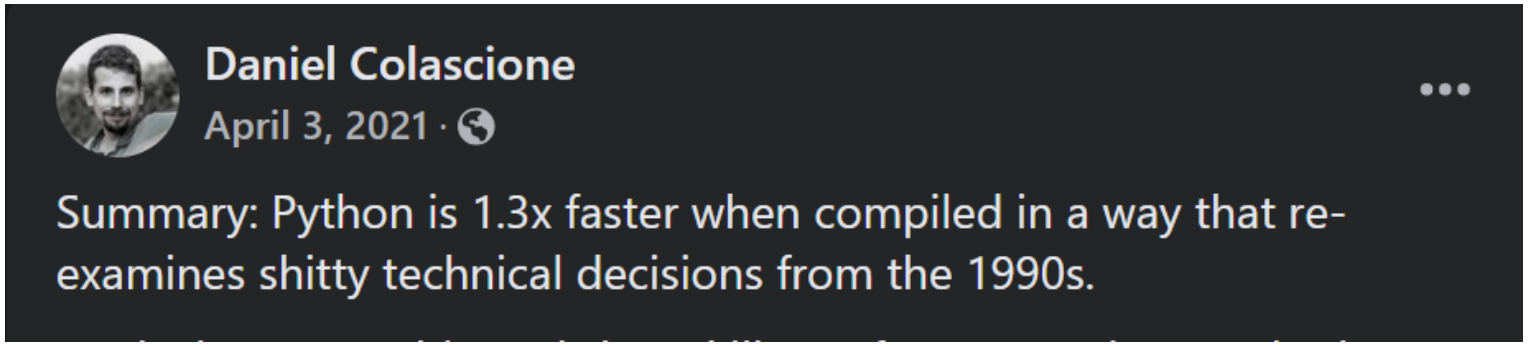
Interposition, in retrospect

- Uneasy coexistence with C++ ODR,
- Rarely used,
- Takes a toll on *ALL* calls from shared libs, even those that are never interposed.

- Doubtful this would have been the design choice taken today.
  - https://maskray.me/blog/2021-05-16-elf-interposition-and-bsymbolic

Interposition, in retrospect

- “Outside of the GNU ELF world, many dynamic linking implementations have shifted to a direct binding and non-interposition by default.”
  - https://gcc.gnu.org/bugzilla/show_bug.cgi?id=100593#c13
- CPython got x1.3 faster by building with `-fno-semantic-interposition`
  - https://github.com/python/cpython/issues/83161



Lazy Bind by Default?

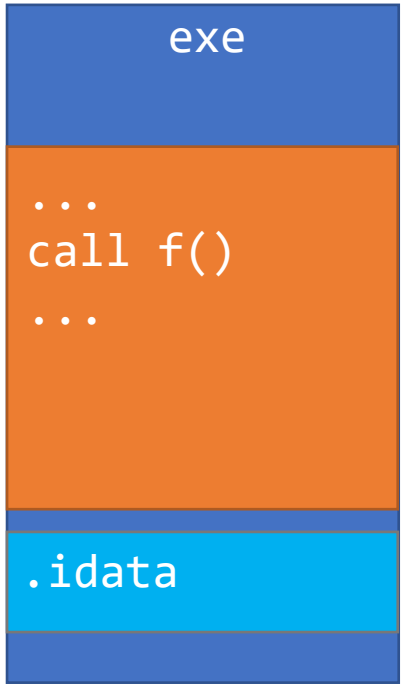
- Lazy binding mandates having the GOT writable throughout the program execution
  - Glaring attack surface!
- As of Today:
  - `ld` still defaults to `-z lazy`
  - Many distros (Debian, Ubuntu, Fedore) configure compilers to build with `-z relro : “relocate+read only”`
  - Other languages (zig, rust) do too.
- Could consider `-no-plt` too
  - Not always possible: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=100593#c13

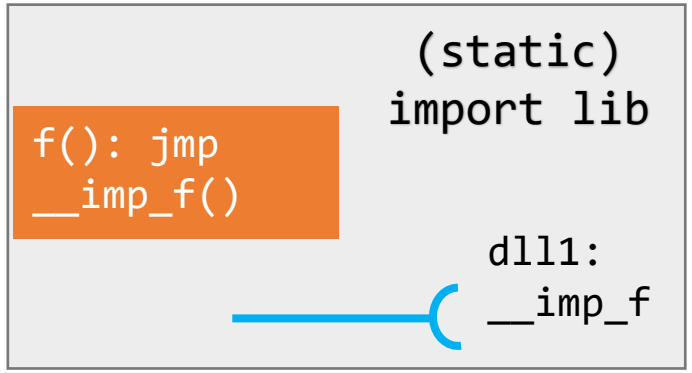
Yes. *

Symbol Visibility

Symbol Visibility - Windows

- `__declspec(dllexport)` – add symbol to .edata
- `__declspec(dllimport)` – does *not* add to .idata!
  - That's achieved by statically linking against an *import library*
  - Minor optimization that happens in Release anyway
- Most symbols are neither.



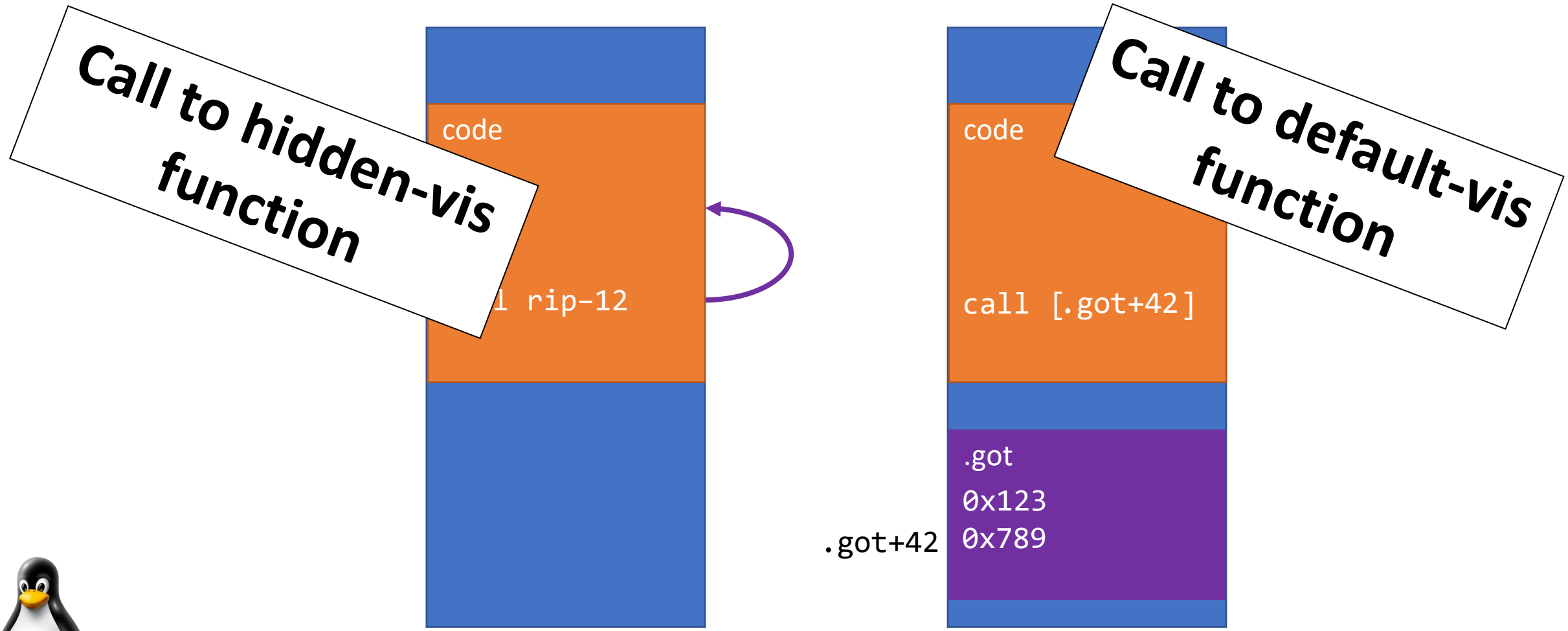


Symbol Visibility - Linux

- An ELF binary has a symbol table: section `.dynsym`
- Everything in the symbol table is accessible by other binaries
  - Potentially “exported”,
- Symbols can be marked UNDEFINED
  - Necessarily *“imported”* by the loader from other binaries.

- Not the end of the story – symbols have *visibility* :
  - Default / Protected / Hidden / ~~internal~~
- `

Symbol Visibility – Who Cares?



Symbol Visibility - Linux

	Default	Protected	Hidden
Available to other binaries? (“exported”)	V	V	X
Subject to interposition? (potentially “imported”)	V	X	X
Has a GOT entry?	V	V	X
Appears in .dynsym?	V	V	X

Symbol Visibility - Linux

- “Using this feature [-fvisibility=hidden] can very substantially improve linking and load times of shared object libraries, produce more optimized code, provide near-perfect API export and prevent symbol clashes. It is strongly recommended that you use this in any shared objects you distribute.”

$ man gcc

- Intervention hooks:
  - `-fvisibility=hidden,`
  - `-fvisibility-inlines-hidden`
  - `-fvisibility-ms-compat`
  - `__attribute__ ((visibility ("hidden")))`

Symbol Visibility vs Symbol Binding

- Binding (global/weak/local):
  - local binding →
    - static / anon-namespace symbols
  - weak binding →
    - Can be preempted even from a different file in the same binary. “Global” without ODR (eg, template instantiation).
  - Per file,
  - Used by the linker.
- Visibility (default/protected/hidden) :
  - Per binary,
  - Used by the Loader.

Recommendations for Shared Libraries on Linux

- Opt out of interposition!
  - Build with `–fvisibility=hidden, –Bsymbolic`
  - Mark only exported functions as `__attribute__((visibility(“protected”)))`

- Easier fallbacks:
  - `-fvisibility-inlines-hidden`
  - `-fvisibility-ms-compat`
  - `-fno-semantic-interposition`

- **Basically, link more like windows.**

C++ and Shared Libs – Preemptive Comment

Shared libraries are out of scope for C++ ?

“… Similarly, x86-64 instruction encoding is outside the scope of the C++ Standard. Does that mean a program that uses x86-64 instructions is not covered by the Standard? Of course not. It just means that that’s a low-level concern (relative to the semantics of C++), and it's up to the platform to make a conforming C++ implementation on top of that stuff. “

Arthur O’Dwyer, on the iso mailing list

https://lists.isocpp.org/std-proposals/2023/07/7239.php

Takeaways

Shared Libs - Takeaways

- Linux and Windows designs are fundamentally different.
- The Linux design is burdened by archaic design goals.
- Practical recommendations: `-fvisibility=hidden, -Bsymbolic`
- “Shared Libs are outside the scope of the standard” – is not a valid bailout!
  - Some fixes to the C++ standard are due.

- Linkers are messy, creative and wonderful.
  And a *highly* active dev front.

Ofek Shilon

Senior Developer
@Toga Networks
(a Huawei Company)

ofekshilon@gmail.com

OfekShilon

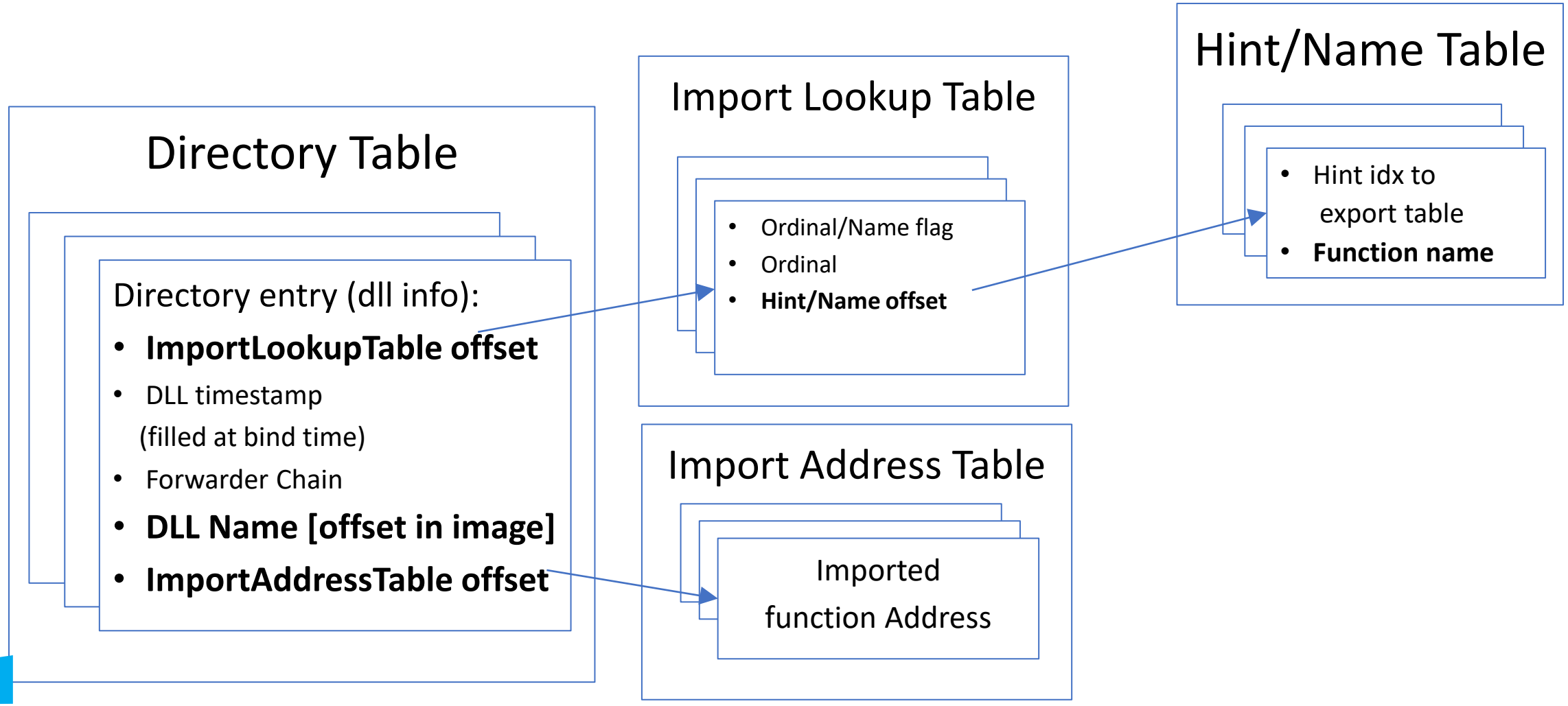
More Rabbit Holes for the Curious

- Relocation details
- Weak linkage
- Versioning (both of libraries and of API)
- Granularity
  - COMDAT, -ffunction-sections, -fdata-sections
- Optimizations
  - Identical Code Folding: /OPT:ICF, -icf=all, -icf=safe
  - Dead Code Elimination: /OPT:REF, -fvtable-gc, --gc-section
- Linker scripts

Resources

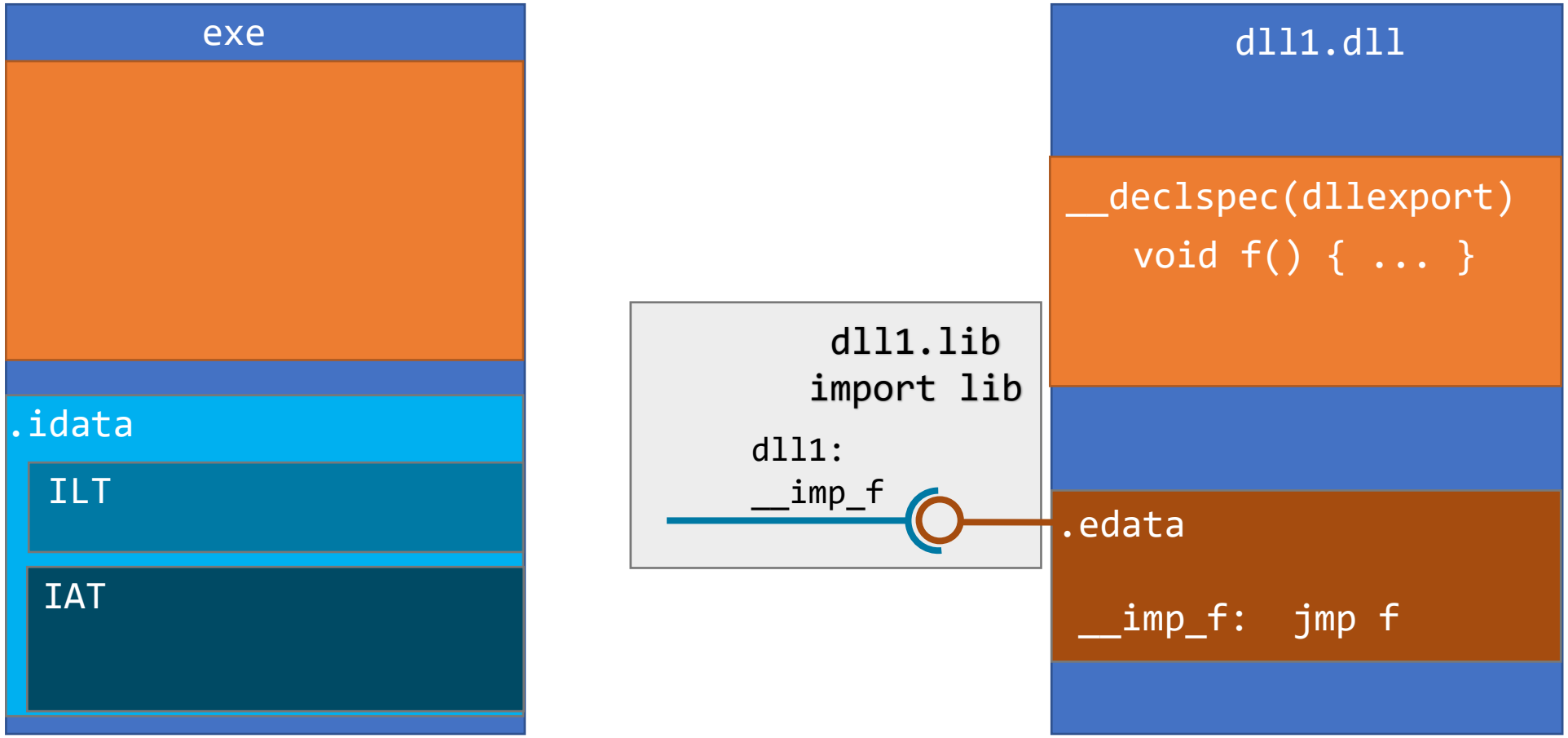
- Ulrich Drepper: “How to write shared libraries”
  - http://library.bagrintsev.me/CPP/dsohowto.pdf
- Eli Benderski:
  - https://eli.thegreenplace.net/2011/08/25/load-time-relocation-of-shared-libraries/
  - https://eli.thegreenplace.net/2011/11/03/position-independent-code-pic-in-shared-libraries/
- Fangrui Song, LLVM/LLD contributor:
  - https://maskray.me/blog/
- SystemV ABI spec and mailing list
  - https://groups.google.com/g/x86-64-abi
- Solaris documentation
  - https://docs.oracle.com/cd/E23824_01/html/819-0690/toc.html
- Ian Lance Taylor, author of GOLD:
  - https://lwn.net/Articles/276782/
- John Levine, “Linkers and Loaders” book:
  - https://www.amazon.com/Linkers-Kaufmann-Software-Engineering-Programming/dp/1558604960
- Michael Kerrisk online workshop:
  - https://www.man7.org/training/shlib/index.html

.idata section layout

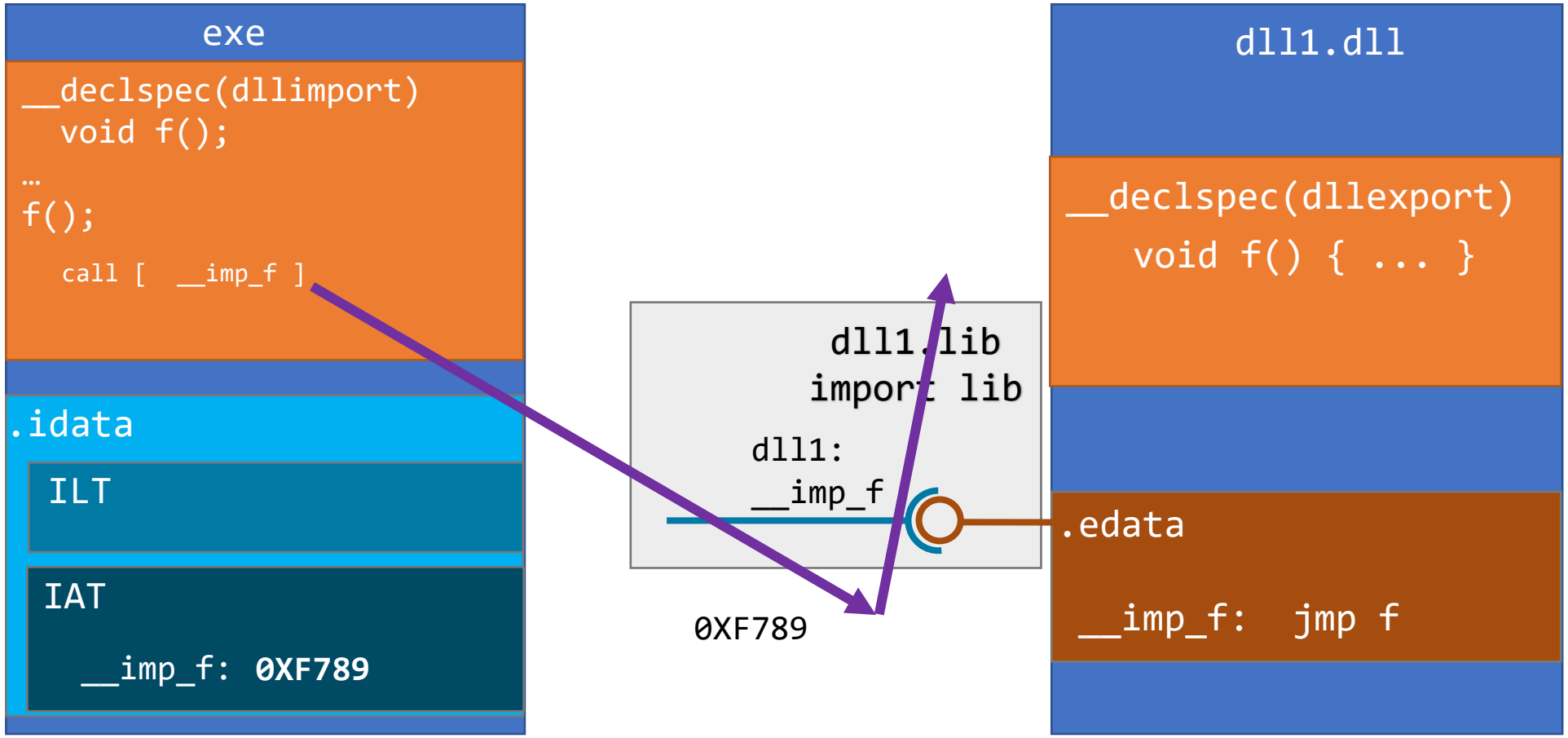


https://learn.microsoft.com/en-us/windows/win32/debug/pe-format#the-idata-section

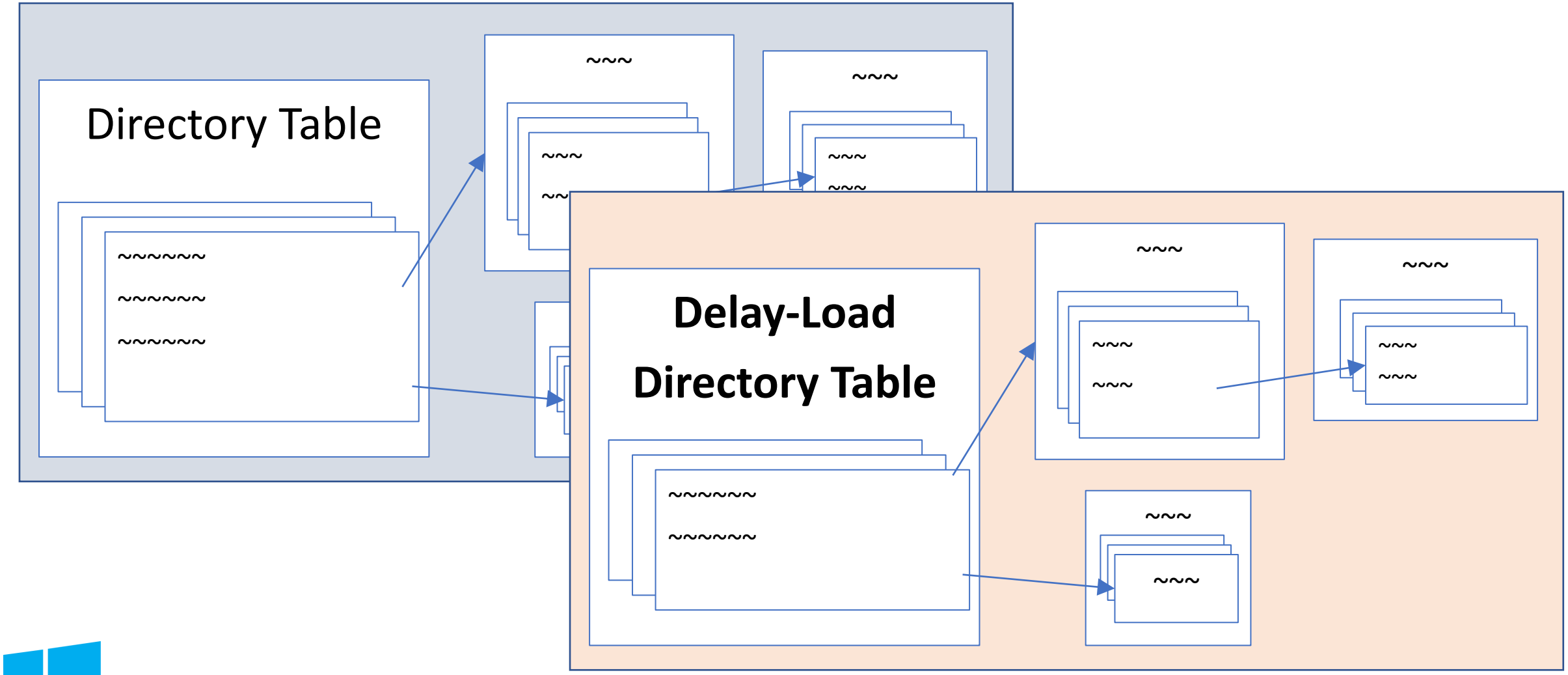
Import Library



Import Library



Windows .idata section - appendix

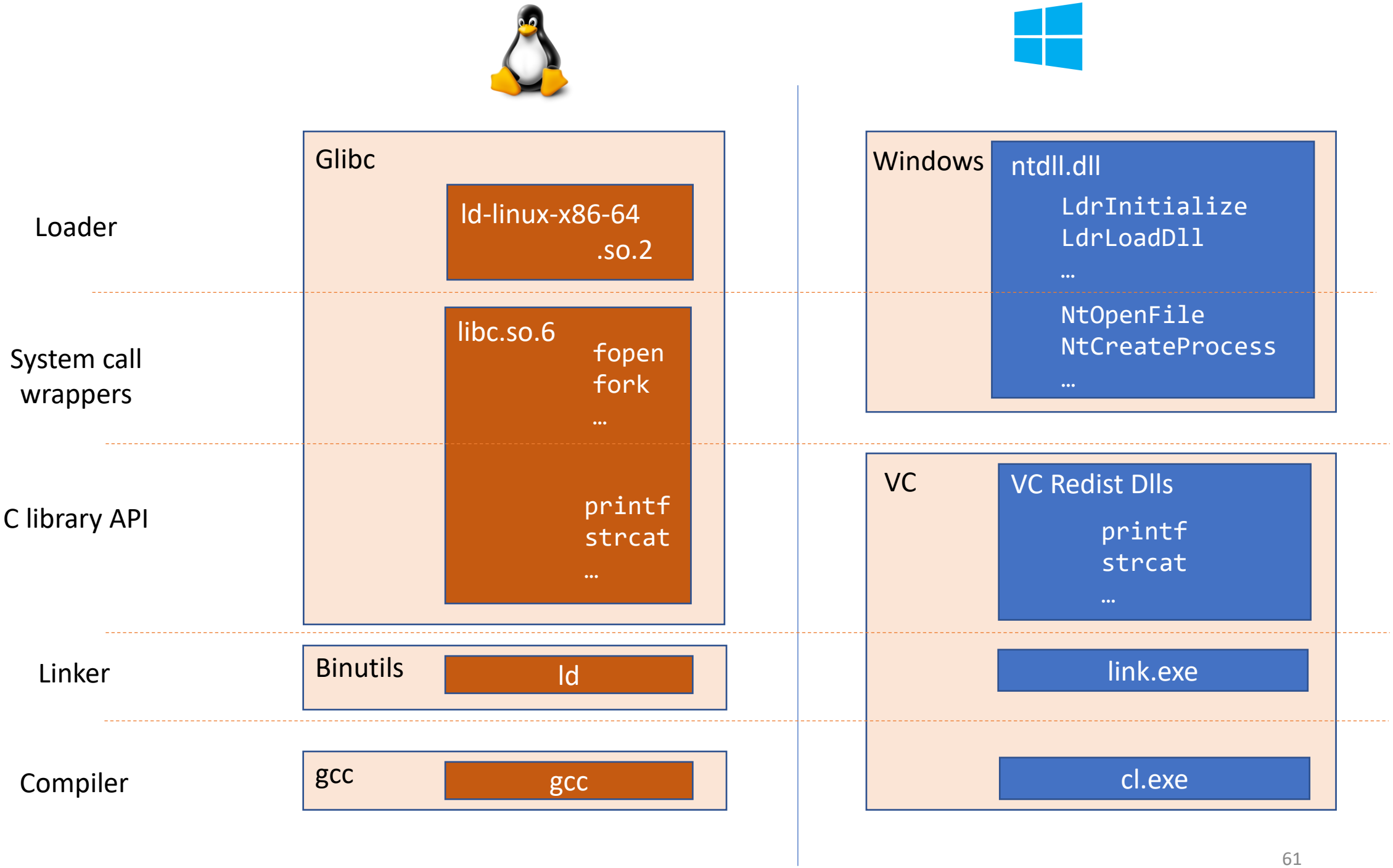


https://learn.microsoft.com/en-us/windows/win32/debug/pe-format#the-idata-section

Loader

- A.k.a Dynamic Linker, a.k.a Interpreter, a.k.a Image Loader (esp in windows)
- Runs in user mode – operates on regular process address space

- In Linux: not invoked for executables linked with -static
- Not even with –static-pie!! https://gcc.gnu.org/legacy-ml/gcc/2015-06/msg00008.html
  - "… motivation for doing the relocations in the start file, rather than with an external program
    interpreter, is both to reduce runtime cost on very small systems, and to make deployment easier."
- So no LD_PRELOAD there


Loader
System call wrappers
C library API
Linker
Compiler
Glibc
ld-linux-x86-64 .so.2
libc.so.6
fopen
fork
…
printf
strcat
…
Binutils
ld
gcc
gcc
Windows
ntdll.dll
LdrInitialize
LdrLoadDll
…
NtOpenFile
NtCreateProcess
…
VC
VC Redist Dlls
printf
strcat
…
link.exe
cl.exe

Selecting a different loader

```
$ gcc –v whatever.cpp
...
COLLECT_GCC_OPTIONS='-v' '-mtune=generic' '-march=x86-64'
 /usr/lib/gcc/x86_64-linux-gnu/9/collect2 -plugin /usr/lib/gcc/x86_64-linux-gnu/9/liblto_plugin.so -
plugin-opt=/usr/lib/gcc/x86_64-linux-gnu/9/lto-wrapper -plugin-opt=-fresolution=/tmp/ccfheLJQ.res -
plugin-opt=-pass-through=-lgcc -plugin-opt=-pass-through=-lgcc_s -plugin-opt=-pass-through=-lc -
plugin-opt=-pass-through=-lgcc -plugin-opt=-pass-through=-lgcc_s --build-id --eh-frame-hdr -m
elf_x86_64 --hash-style=gnu --as-needed -dynamic-linker /lib64/ld-linux-x86-64.so.2 -pie -z now -z
relro /usr/lib/gcc/x86_64-linux-gnu/9/../../../x86_64-linux-gnu/Scrt1.o /usr/lib/gcc/x86_64-linux-
gnu/9/../../../x86_64-linux-gnu/crti.o /usr/lib/gcc/x86_64-linux-gnu/9/crtbeginS.o -
L/usr/lib/gcc/x86_64-linux-gnu/9 -L/usr/lib/gcc/x86_64-linux-gnu/9/../../../x86_64-linux-gnu -
L/usr/lib/gcc/x86_64-linux-gnu/9/../../../../lib -L/lib/x86_64-linux-gnu -L/lib/../lib -
L/usr/lib/x86_64-linux-gnu -L/usr/lib/../lib -L/usr/lib/gcc/x86_64-linux-gnu/9/../../..
/tmp/ccijfiIQ.o -lgcc --push-state --as-needed -lgcc_s --pop-state -lc -lgcc --push-state --as-
needed -lgcc_s --pop-state /usr/lib/gcc/x86_64-linux-gnu/9/crtendS.o /usr/lib/gcc/x86_64-linux-
gnu/9/../../../x86_64-linux-gnu/crtn.o
...
```

Selecting a different loader

```
$ readelf --program-headers /usr/bin/ls
...
Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x00000000000002d8 0x00000000000002d8  R      0x8
  INTERP         0x0000000000000318 0x0000000000000318 0x0000000000000318
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
...
```

Observing the Loader in action - Linux

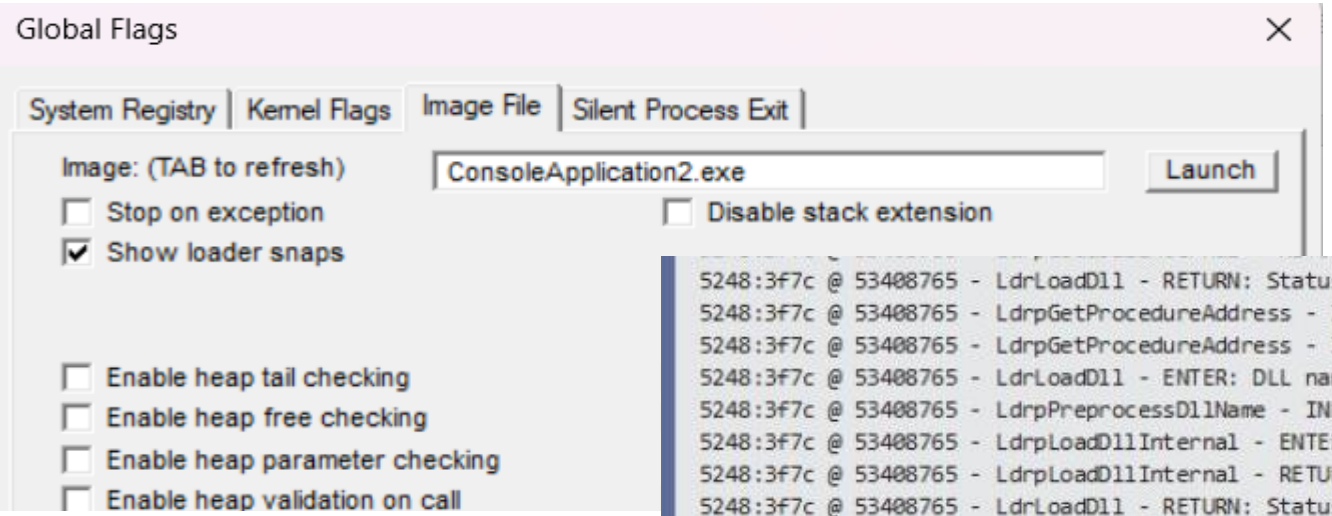
```
$ LD_DEBUG=help cat
Valid options for the LD_DEBUG environment variable are:

  libs        display library search paths
  reloc       display relocation processing
  files       display progress for input file
  symbols     display symbol table processing
  bindings    display information about symbol binding
  versions    display version dependencies
  scopes      display scope information
  all         all previous options combined
  statistics  display relocation statistics
  unused      determined unused DSOs
  help        display this help message and exit
```

Observing the Loader in action - Linux

```
$ LD_AUDIT=1
Check!!
```

Observing the Loader in action - Windows




```
5248:3f7c @ 53408765 - LdrLoadDll - RETURN: Status: 0x00000000
5248:3f7c @ 53408765 - LdrpGetProcedureAddress - INFO: Locating procedure "CompareStringEx" by name
5248:3f7c @ 53408765 - LdrpGetProcedureAddress - INFO: Locating procedure "EnumSystemLocalesEx" by name
5248:3f7c @ 53408765 - LdrLoadDll - ENTER: DLL name: api-ms-win-core-datetime-l1-1-1
5248:3f7c @ 53408765 - LdrpPreprocessDllName - INFO: DLL api-ms-win-core-datetime-l1-1-1 was redirected to C:\WIN
5248:3f7c @ 53408765 - LdrpLoadDllInternal - ENTER: DLL name: C:\WINDOWS\SYSTEM32\kernelbase.dll
5248:3f7c @ 53408765 - LdrpLoadDllInternal - RETURN: Status: 0x00000000
5248:3f7c @ 53408765 - LdrLoadDll - RETURN: Status: 0x00000000
5248:3f7c @ 53408765 - LdrpGetProcedureAddress - INFO: Locating procedure "GetDateFormatEx" by name
5248:3f7c @ 53408765 - LdrpGetProcedureAddress - INFO: Locating procedure "GetLocaleInfoEx" by name
5248:3f7c @ 53408765 - LdrpGetProcedureAddress - INFO: Locating procedure "GetTimeFormatEx" by name
5248:3f7c @ 53408765 - LdrpGetProcedureAddress - INFO: Locating procedure "GetUserDefaultLocaleName" by name
5248:3f7c @ 53408765 - LdrpGetProcedureAddress - INFO: Locating procedure "IsValidLocaleName" by name
5248:3f7c @ 53408765 - LdrLoadDll - ENTER: DLL name: api-ms-win-core-localization-obsolete-l1-2-0
5248:3f7c @ 53408765 - LdrpPreprocessDllName - INFO: DLL api-ms-win-core-localization-obsolete-l1-2-0 was redirec
5248:3f7c @ 53408765 - LdrpLoadDllInternal - ENTER: DLL name: C:\WINDOWS\SYSTEM32\kernelbase.dll
5248:3f7c @ 53408765 - LdrpLoadDllInternal - RETURN: Status: 0x00000000
5248:3f7c @ 53408765 - LdrLoadDll - RETURN: Status: 0x00000000
5248:3f7c @ 53408765 - LdrpGetProcedureAddress - INFO: Locating procedure "LCIDToLocaleName" by name
5248:3f7c @ 53408765 - LdrpGetProcedureAddress - INFO: Locating procedure "LocaleNameToLCID" by name
```

https://ofekshilon.com/2013/06/22/entry-point-not-found-and-other-dll-loading-problems/

Shared lib vs Static Lib

- Both are mechanisms for code sharing
- Static:
  1. Smaller single binary footprint (only used sections are imported)
  2. Better code locality
  3. Efficient calls
- Dynamic:
  1. Shared binary copies across processes (Linus disagrees)
  2. Save overall disk space (Linus disagrees)
  3. Separate lib versioning
     - Linus Torvalds disagrees with 1-3: https://lore.kernel.org/lkml/CAHk-=whs8QZf3YnifdLv57+FhBi5_WeNTG1B-suOES=RcUSmQg@mail.gmail.com/
  4. Separate (smaller) linkage

Vague Linkage

- C++ constructs not clearly tied to a single translation unit:
  - Out-of-line copies of Inline Functions
  - String constants (shared across obj files)
  - VTables
  - type_info objects
  - Template Instantiations

Indirect Call Overhead

- Recent (2020) work on de-virtualization in clang:
  - "Modeling the Invariance of Virtual Pointers in LLVM" Piotr Padlewski, Krzysztof Pszeniczny, Richard Smith , https://arxiv.org/pdf/2003.04228.pdf
- "Our benchmarks show an average of 0.8% performance improvement on real-world C++ programs, with more than 30% speedup in some cases. "