

Lecture 5.2

Loader Design Options and dynamic linking

Md Mahbub E Noor

Assistant Professor, Department of CSE
University of Barishal

```
LD_DEBUG=bindings ./count_so (real)
binding file ./count_so
  to ./libcount.so: symbol `total'
binding file ./count_so
  to ./libcount.so: symbol `counta'
binding file ./count_so
  to libc.so.6: symbol `printf'
total = 3
```

What you will learn today



Explain automatic library search, and why the order of -l matters.



Compare a linking loader, a linkage editor and dynamic linking.



Build, find and debug a shared library with ldd, LD_DEBUG and strace.



Trace a call through the PLT and GOT, with real gdb values.

Today's plan (60 min)

0-5

Recap of 5.1

5-15

Library search, options

15-25

Linkage editors

25-50

Dynamic linking, PLT/GOT

50-60

Big picture, checkpoint

Lab 5.2: finish the linking loader (TODO 4-5), then the Linux part.

Quick recap of Lecture 5.1

Q1

What does Pass 1 of the linking loader build?

ESTAB: every section name and every exported (D record) name with its real address.

Q2

M00000105+COUNTA: what does Pass 2 do?

Looks up COUNTA in ESTAB (4011) and adds it to the 5 half-bytes at 0001 of this section.

Q3

Before linking, **call counta** in main.o is **e8 00 00 00 00**. Who fills it?

ld, the linker: **e8 29 00 00 00** = the distance to counta. Today: who fills it when counta is in a shared library?

PART 1

Machine-independent loader features

Searching libraries automatically, and options that change what gets linked.

Automatic library search: take only what is missing

- 1 After Pass 1, look at ESTAB: which referenced names are still **undefined**?
- 2 Search the library's directory for a member that **defines** one of them.
- 3 Load that member like any other section; it may need new names: repeat.
- 4 Names still undefined at the end: error.

```
● ● ● a static library = an archive of .o files (real)
$ ar rcs libcount.a counta.o data.o
$ ar t libcount.a
counta.o
data.o
$ gcc -no-pie -o count_lib main.o -L. -lcount
$ ./count_lib
total = 3
```

Same as C: you never write `printf.o` on the gcc command line. gcc adds the C library for you, and only what your program needs is taken from it. `-L.` says where to look, `-lcount` means the file `libcount.a` (or `libcount.so`).

Book: Beck §3.3.1 (automatic library search)

Why the order of -l matters, and the famous -lm

● ● ● library before the object file (real)

```
$ gcc -no-pie -o x -L. -lcount main.o
main.c:(.text+0x9): undefined reference
to `counta'
main.c:(.text+0xf): undefined reference
to `total'
```

● ● ● root.c uses sqrt (real)

```
$ gcc -o root root.c
root.c:(.text+0x37): undefined reference
to `sqrt'
$ gcc -o root root.c -lm
$ ./root
sqrt(2) = 1.414214
```

One pass, left to right

ld visits files in command-line order. When it reaches libcount.a, main.o has not been read yet, so **nothing is undefined** and no member is taken. Then main.o needs counta: too late.

The rule

Put libraries **after** the files that use them: `gcc main.o -lcount`.

Why -lm?

sqrt lives in the math library libm, which gcc does not add by itself. `#include <math.h>` only declares sqrt; it does not link it.

Loader options: telling the loader what to do

Beck's command	What it asks	With gcc / ld (all used today)
INCLUDE prog(lib)	load this program from this library	name the .o or .a on the command line
LIBRARY MYLIB	search this library too	-L. -lcount
(load map)	print where everything went	-Wl, -Map=count.map
DELETE / CHANGE	drop a section / rename a symbol	ld has options for these; not needed in this course
NOCALL STDDEV	do not load these routines, even if called	no common gcc equivalent
(no Beck name)	link everything now, or at run time?	-static or the default (shared)
(no Beck name)	where to find shared libraries	-Wl, -rpath, '\$ORIGIN'

-Wl,... passes an option through gcc to ld. When you type `gcc main.o -lcount`, gcc runs ld with a long list of options for you: `gcc -v` shows it.

Book: Beck §3.3.2 (loader options)

PART 2

Linking loader or linkage editor?

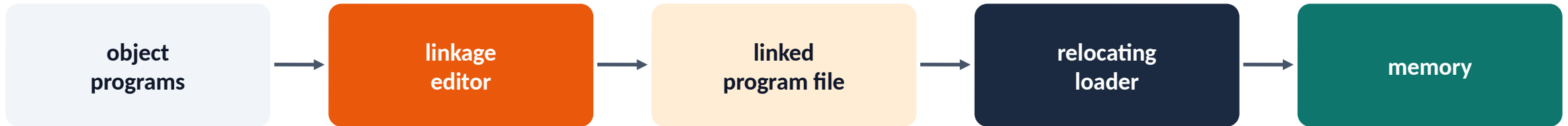
Link every time you load, or link once and save the result.

Two ways to organise linking and loading

Linking loader (Lecture 5.1): link and load together, every run



Linkage editor: link once, save the result; load it many times



SAME AS C

ld is a linkage editor

`gcc main.o counta.o data.o -o count` links once and writes the file `count`. You run it a thousand times without linking again.

DIFFERENT FROM C

But not the whole story

`count` still needs `libc.so.6`: that part is linked **every run** by `ld.so`. Linux mixes both designs (next part).

Book: Beck §3.4.1 (linkage editors)

Link everything in? The price of static linking

```
● ● ● same three C files, two ways (real)
$ gcc -no-pie -static -o count_static main.c counta.c
data.c
$ gcc -no-pie -o count_dyn main.c counta.c data.c
$ size count_static count_dyn
   text    data     bss  filename
667949  23360  22440 count_static
   1182     556      12 count_dyn
$ ls -l → 785504 count_static
         16064 count_dyn
```

785 KB

static file

16 KB

dynamic file

1868

symbols (nm) in the
static file

Our own code is $0x36 + 0x44 = 122$ bytes. The rest of the 668 KB of text is **printf** and **everything printf** needs, copied in from **libc.a**.

Every static program carries its own copy of **printf**: 100 programs = 100 copies on disk and in memory.

A bug fixed in **libc**? Every static program must be **linked again**.

But: no library can go missing at run time, and start-up is fast.

PART 3

Dynamic linking

Link the library when the program starts, or even later: ld.so, GOT and PLT.

Make a shared library and use it

```
demo folder (real)

$ gcc -fPIC -c counta.c -o counta_pic.o
$ gcc -fPIC -c data.c -o data_pic.o
$ gcc -shared -o libcount.so counta_pic.o data_pic.o
$ gcc -o count_so main.o -L. -lcount
$ ./count_so
./count_so: error while loading shared libraries:
libcount.so: cannot open shared object file
exit status 127
$ LD_LIBRARY_PATH=. ./count_so
total = 3
$ gcc -o count_rp main.o -L. -lcount -Wl,-rpath,'$ORIGIN'
$ ./count_rp
total = 3
```

-fPIC

Position-independent code: the library may be loaded at **any** address, so its code must not contain absolute addresses.

Linking ≠ copying

The link step only checks that counta exists and writes "I need libcount.so" into count_so. The code is not copied.

The error

At **run time** ld.so must find the file. The current folder is not searched. LD_LIBRARY_PATH or a RUNPATH (\$ORIGIN = the program's own folder) fixes it.

Who links at run time? The dynamic linker ld.so

```
● ● ● readelf -l count_dyn (excerpt, real)
```

```
Entry point 0x401050
```

```
INTERP [Requesting program interpreter:  
         /lib64/ld-linux-x86-64.so.2]
```

```
LOAD ... (4 segments)
```

```
● ● ● ldd and a direct start (real)
```

```
$ LD_LIBRARY_PATH=. ldd ./count_so  
linux-vdso.so.1 (0x00007f7fa716d000)  
libcount.so => ./libcount.so (0x00007f7fa715b000)  
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (...)  
/lib64/ld-linux-x86-64.so.2 (0x00007f7fa716f000)  
$ /lib64/ld-linux-x86-64.so.2 ./count_dyn  
total = 3
```

1. The kernel

loads count_dyn's segments, sees INTERP, and also loads ld.so. It starts **ld.so**, not your program (Lecture 8.2).

2. ld.so

reads the NEEDED list, finds and maps libcount.so and libc.so.6, relocates them, binds the names. A linking loader!

3. Your program

ld.so jumps to the entry point (`_start`), which calls main.

Book: Beck §3.4.2 (dynamic linking)

Watching ld.so search and load (real traces)

```
LD_DEBUG=libs LD_LIBRARY_PATH=. ./count_so
find library=libcount.so [0]; searching
search path=./glibc-hwcaps/x86-64-v4:
...-v3:...-v2:. (LD_LIBRARY_PATH)
trying file=./glibc-hwcaps/x86-64-v4/...
trying file=./libcount.so
find library=libc.so.6 [0]; searching
trying file=./libc.so.6
search cache=/etc/ld.so.cache
trying file=/lib/x86_64-linux-gnu/libc.so.6
calling init: ./libcount.so
initialize program: ./count_so
transferring control: ./count_so
```

```
strace -e trace=openat,mmap (excerpt)
openat("./libcount.so", O_RDONLY) = 3
mmap(NULL, 16416, PROT_READ, ..., 3, 0)
mmap(..., 4096, PROT_READ|PROT_EXEC, ..., 3, 0x1000)
mmap(..., 8192, PROT_READ|PROT_WRITE, ..., 3, 0x2000)
openat("/etc/ld.so.cache") = 3
openat("/lib/x86_64-linux-gnu/libc.so.6") = 3
```

The search order: LD_LIBRARY_PATH, then the RUNPATH in the program, then the cache /etc/ld.so.cache (built by ldconfig), then /lib and /usr/lib. **mmap** maps the file's pieces into memory: code read+execute, data read+write.

Relocation at load time, at a random address

```
● ● ● ldd twice: libcount.so's base (real)
libcount.so => ./libcount.so (0x00007efd940cb000)
libcount.so => ./libcount.so (0x00007f6b4c702000)
```

base + offset

$0x7f041e7b30f9 - 10f9 = 0x7f041e7b2000$, and $0x7f041e7b6018 - 4018 =$ the same base. One base per library, offsets from the file.

```
● ● ● dyn.c: dlopen + dlsym, run twice (real)
counta is at 0x7f041e7b30f9, total is at 0x7f041e7b6018
counta is at 0x7f3e190450f9, total is at 0x7f3e19048018
$ nm -D libcount.so
00000000000010f9 T counta
0000000000004018 B total
```

This is PROGADDR

ASLR picks a new random base each run, like a relocating loader that gets a different PROGADDR every time. Attackers cannot guess where code is.

Same as 5.1

Only the base changes; the last three hex digits (0f9, 018) never do, because bases are multiples of the page size, 1000 hex (4096).

Binding names: ld.so's Pass 2

```
LD_DEBUG=bindings LD_LIBRARY_PATH=. ./count_so (real)
binding file ./libcount.so to ./libcount.so: normal symbol `text'
binding file ./libcount.so to ./libcount.so: normal symbol `len'
binding file ./count_so to ./libcount.so: normal symbol `total'
binding file ./count_so to ./libcount.so: normal symbol `counta'
binding file ./count_so to libc.so.6: normal symbol `printf'
```

The big difference

Our linking loader wrote addresses **into the code** (+JSUB 4011). A shared library's code is shared by many processes and mapped **read-only**, so nobody may write into it.

Solution: the code reads every outside address from a small **writable** table in each process: the Global Offset Table (GOT). ld.so fills the GOT; the code never changes.

Linking loader (5.1)	ld.so
ESTAB built from D records	the dynamic symbol tables of all loaded files (nm -D)
R record: TOTAL, TEXT	undefined dynamic symbols (nm -D: U)
M record +TOTAL	dynamic relocations (readelf -r count_so)
patch the code	fill a table of addresses: the GOT

Calls through the PLT: one stub per function

```
objdump -d lazy (real)
401162: e8 e9 fe ff ff  call 401050 <counta@plt>
...
0000000000401050 <counta@plt>:
401050: f3 0f 1e fa      endbr64
401054: ff 25 a6 2f 00 00  jmp *0x2fa6(%rip) # 404000
```

```
readelf -r lazy (.rela.plt, real)
404000 R_X86_64_JUMP_SLOT counta + 0
404008 R_X86_64_JUMP_SLOT printf@GLIBC_2.2.5 + 0
```

1 main calls **counta@plt**, a stub inside the program. ld could fill that call: the stub is in the same file.

2 The stub jumps to the address stored in the **GOT slot 404000**.

3 ld.so fills slot 404000 (a JUMP_SLOT relocation = an M record for the GOT).

C link: you saw `call puts@PLT` in Lecture 1's assembly. The GOT is an array of function pointers, and the stub does `goto *got[3];`.

Watch the first call to counta

Lazy binding: the first call to counta through PLT and GOT

STARTCALL 1CALL 2

```
main (lazy.C, -z lazy)
int a = counta();
401162: call 401050 <counta@plt>

int b = counta();
40116a: call 401050 <counta@plt>
```

call 1

PLT (code in the program)

```
.plt (PLT0)
401020: push [403ff0]    GOT[1]
401026: jmp  *[403ff8]   GOT[2]
401030: endbr64
401034: push $0x0       reloc 0
401039: jmp  401020     to PLT0
.plt.sec
401050: endbr64         counta@plt
401054: jmp  *[404000]  GOT[3]
```

GOT (.got.plt, writable data)

403fe8 GOT[0]	0000000000403de8	_DYNAMIC
403ff0 GOT[1]	00007ffff7ffe2e0	link_map
403ff8 GOT[2]	00007ffff7fda2f0	resolver
404000 GOT[3]	00007ffff7fb70f9	counta
404008 GOT[4]	0000000000401040	printf

Who writes what

PLT: read-only code, made by the linker (ld)

GOT: data, filled in at run time by ld.so

ld.so (/lib64/ld-linux-x86-64.so.2)

```
7ffff7fda2f0 _dl_runtime_resolve_xsavec

reads reloc 0 in .rela.plt:
JUMP_SLOT counta at 404000
looks up counta, writes GOT[3]
```

libcount.so (loaded at 7ffff7fb6000)

```
7ffff7fb70f9 counta
= base 7ffff7fb6000 + 10f9
(nm -D libcount.so: counta at 10f9)
```

The resolver looks up counta, finds it at 7ffff7fb70f9 in libcount.so, writes that into GOT[3] and jumps there. A relocation done at run time, once.

Watch for

1. At start GOT[3] points **back into the PLT** (401030).
2. push \$0x0: which relocation to fix.
3. PLT0 jumps into **ld.so** through GOT[2].
4. ld.so **writes GOT[3]**: it turns orange.
5. The second call skips ld.so completely.

Click the video to play or pause. About 1½ minutes, no sound.

The evidence: gdb reads the GOT slot

```
gdb -batch -x got.gdb ./lazy (real)

Breakpoint 1, main () at lazy.c:6
before 1st call: GOT[counta] at 0x404000 = 0x401030
Breakpoint 2, main () at lazy.c:7
after 1st call : GOT[counta] at 0x404000 = 0x7ffff7fb70f9
counta in section .text of ../libcount.so
Breakpoint 3, main () at lazy.c:8
after 2nd call : GOT[counta] at 0x404000 = 0x7ffff7fb70f9
```

```
built with -Wl,-z,now instead (real)

BIND_NOW, before 1st call: GOT[counta] = 0x7ffff7fb70f9
```

Lazy binding (-z lazy)

The slot is filled at the **first call**. Functions that are never called are never looked up: faster start for programs that use few of many library functions.

Immediate binding (-z now)

ld.so fills **every** slot before main. Ubuntu's gcc does this for normal (PIE) programs: `readelf -d count_so` shows `FLAGS BIND_NOW`.

Why now is the default

After start-up the GOT can be made read-only (RELRO), so an attacker cannot overwrite it to redirect calls.

Beck's dynamic linking and Linux: the same idea

1

The program asks the OS for a routine **by name** (a service request), instead of calling an address.

2

First time: the OS's dynamic loader finds the routine in a library and loads it.

3

It records the routine's address in a table and passes control to it.

4

Next time: the call goes **directly** through the table. No loading, no searching.

SAME AS C

Same as lazy binding

Beck's table is the GOT; the request by name is the PLT stub + push \$0x0 (relocation number = which name). First call slow, later calls one jump.

DIFFERENT FROM C

Different in detail

Beck's dynamic loader is part of the **operating system**. On Linux, ld.so is an ordinary user-space program loaded into every process; the kernel only maps files. And dlopen lets a program load a library it did not know about when it was linked (next slide).

Book: Beck §3.4.2 (dynamic linking)

Loading a library on request: dlopen and dlsym

● ● ● dyn.c (demo folder)

```
void *h = dlopen("./libcount.so", RTLD_NOW);
if (!h) { printf("...%s\n", dlerror()); return 1; }
int (*f)(void) = (int (*)(void)) dlsym(h, "counta");
int *t = (int *) dlsym(h, "total");
*t = f();
printf("total = %d\n", *t);
dlclose(h);
```

● ● ● gcc -o dyn dyn.c && ./dyn (real)

```
counta is at 0x7f041e7b30f9, total is at 0x7f041e7b6018
total = 3
```

dlopen = run the loader

Load the file and its dependencies, relocate it, bind its names. Your program becomes the one asking, at a time you choose.

dlsym = look up ESTAB

Give a name, get an address. Then call it through a **function pointer**, as in 1st-semester C.

Where it is used

Plug-ins: a program that loads modules chosen by the user, like a browser or Python loading a C extension.

Static or dynamic linking? The trade-offs

Question	Static (-static)	Dynamic (default)
File size (count)	785 504 bytes	16 064 bytes
Copies of printf in memory, 50 programs	50	1, shared
libc security fix	relink every program	update libc.so.6 once
Start-up	fast: nothing to find	ld.so must find, map and bind libraries
Can fail at run time?	no missing libraries	yes: "cannot open shared object file"
Each call to printf	direct call	through the PLT/GOT: one extra jump
Runs on another Linux?	usually yes	needs compatible library versions

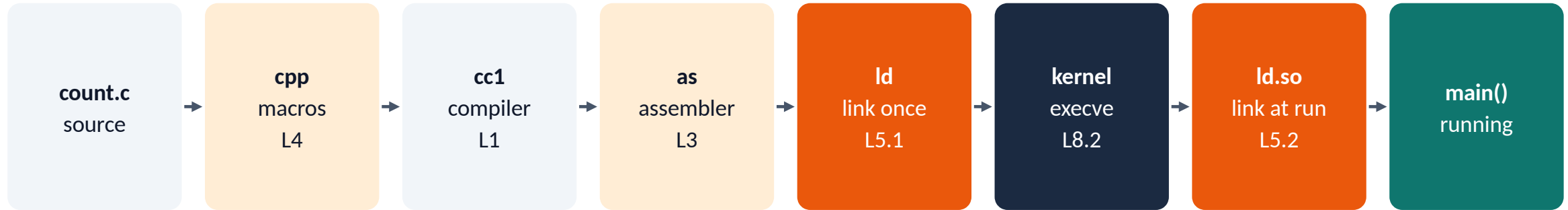
No winner: desktops and servers use dynamic linking; small tools for rescue disks and containers are often static. Know the trade-off, then choose.

PART 4

From count.c to a running process

Every tool of Part A so far, in one chain.

The whole chain, and where each lecture fits



SIC/XE world (Beck)	Linux world (real tools)
macro processor: NAMTAB, DEFTAB, ARGTAB	cpp (gcc -E)
two-pass assembler: SYMTAB, M records	as: symbol table, relocation entries
linking loader: ESTAB, CSADDR	ld: symbol table, load map (-WL,-Map)
relocating loader: + PROGADDR	kernel + ld.so: random base (ASLR)
bootstrap loader at address 0	firmware → boot loader → kernel

LAB 5.2

Finish the linking loader, then explore Linux

TODO 4 and 5 in sicload.c; then libraries, ld.so and the GOT on the lab PC.

Lab 5.2 tasks, step by step

1

TODO 4: Pass 1

Fill ESTAB from H and D records. `./sicload -l 4000 main.obj counta.obj data.obj` must print the load map of Lecture 5.1.

2

TODO 5: Pass 2

Apply M records with `±symbol` from ESTAB. `diff` your output with `expected_link4000.txt`: nothing.

3

Errors

Leave out `data.obj`, then give `data.obj` twice. You must get "undefined external symbol TOTAL" and "duplicate external symbol DATA".

4

Libraries

In `demo/`: build `libcount.a` and `libcount.so`; reproduce the -l order error and the 'cannot open shared object file' error, and fix both.

5

Watch ld.so

Run with `LD_DEBUG=libs` and `LD_DEBUG=bindings`, and with `strace`. Find where `libcount.so` is opened and mapped.

6

The GOT

`gdb -batch -x got.gdb ./lazy`: write down `GOT[counta]` before and after the first call; check `base + 10f9`.

Submit: `sicload.c`, your load map, and a short report with the GOT values from task 6.

Messages you will meet, and what they mean

Message (real)	Who says it	When	Fix
undefined reference to `sqrt'	ld	linking	add -lm after the source file
undefined reference to `counta'	ld	linking	add the .o, or put -lcount after main.o
multiple definition of `text'	ld	linking	remove one definition (or make one static)
cannot open shared object file	ld.so	starting	LD_LIBRARY_PATH=. or -Wl,-rpath,'\$ORIGIN'
dlopen failed: ... No such file	your program	running	check the path given to dlopen
undefined external symbol TOTAL	sicload	Pass 2	link data.obj too
duplicate external symbol DATA	sicload	Pass 1	give each file once

Read the "who" column: ld errors mean "rebuild"; ld.so errors mean "the program is fine, the environment is not".

Checkpoint: test yourself

Q1

Why does `gcc -L. -lcount main.o` fail but `gcc main.o -L. -lcount` work?

→ `ld` searches the library when it meets it; nothing is undefined yet

Q2

`count_so` links fine but will not start. Which program complains?

→ `ld.so`: it cannot find `libcount.so` at run time

Q3

Why can `ld.so` not patch the call instruction, as our linking loader did?

→ library code is shared and read-only; it fills the GOT

Q4

`GOT[counta]` before the first call (lazy) points where?

→ back into the PLT (0x401030), to call the resolver

Q5

`counta` is at 0x7f3e190450f9, offset 10f9. Base of `libcount.so`?

→ 0x7f3e19044000

Key ideas from today



Library search takes only the members that define missing names; ld searches left to right.



A linkage editor (ld) links once into a file; a linking loader links at every load.



Static linking copies the library in (785 KB); dynamic linking shares one copy (16 KB).



ld.so is a linking loader that runs at start-up: find, map, relocate, bind.



Shared code is read-only, so calls go through the PLT and the GOT; ld.so fills the GOT.



dlopen/dlsym: load and look up a library yourself, then call through a function pointer.

Homework 5.2 (due before Lecture 6)

1. Library search

libcount.a holds count.o and data.o. For `gcc main.o -L. -lcount`, list which members ld takes and why, step by step, like Beck's library search.

2. Linking loader

Our sicload has no library search. Describe in 6–8 lines how you would add it, using ESTAB and a library file that lists "name → object file".

3. The GOT

In gdb, GOT[printf] at 0x404008 was 0x401040 before printf ran and 0x7fff7c60100 after. libc's printf offset (nm -D) is 0x60100. Where was libc loaded? Where does 0x401040 point?

4. dlopen

Write dynm.c: load libm.so.6 with dlopen, find sqrt with dlsym, and print sqrt(2). Which gcc options does it need on our lab PC?

5. Trade-off

A lab has 40 PCs running the same 30 tools. Static or dynamic? Give two reasons, and one situation where you would choose the other.

Next lecture and reading

NEXT: LECTURE 6

Object file formats: ELF

One format for .o files, executables and shared libraries: the ELF header, sections and segments, symbol tables and relocation entries — the real-world H, T, D, R, M and E records.

Main text

L. L. Beck, **System Software**, 3rd ed. Today: §3.3–3.4 (and §3.5 for real loaders).

C side

R. E. Bryant & D. R. O'Hallaron, **CS:APP**, 3rd ed., §7.6–7.12 (symbol resolution, relocation, shared libraries, PIC).

Manual pages

`man ld.so`, `man ldd`, `man dlopen`: on the lab PC, offline.

Lab files

`lab5_files.zip` (same as 5.1): `demo/` has `run_all.sh`, `got.gdb` and every C file.