Dynamic Libraries

Systems Programming

Michael Sonntag
Institute of Networks and Security

JOHANNES KEPLER
UNIVERSITÄT LINZ

INSTITUTE
OF NETWORKS
AND SECURITY

Why dynamic libraries?

- Dynamic library = DLL (Windows), Shared Object (Linux)
- Share code – without giving away the source
- Reuse code
- Avoid multiple copies of the same code in memory
- Fix bug at one central place
  - Smaller patches
  - Bug is removed for all programs using that library
    - Otherwise each program would have to be replaced separately

- But: “DLL hell”
  - Programs need libraries in different versions
    - Prg A needs v1.0 and Prg B needs v1.1; both should be installed on a single computer and work simultaneously
  - Upgrading can lead to the case that other programs do not work anymore (incompatible changes, especially in the API)

helloworld-lib.s – Using a DLL

```
#PURPOSE:  This program writes the message "hello world" and exits

.section .data
helloworld:     .ascii "hello world\n\0"

.section .text
.globl _start
_start:
       movq $helloworld,%rdi     # Store address in first parameter
       xorq %rax,%rax            # Clear RAX (no floating point parameters
                                 # this is a vararg function)
                                 # We didn't use the stack, so it should
                                 # remain 16-Byte aligned from _start
       call printf

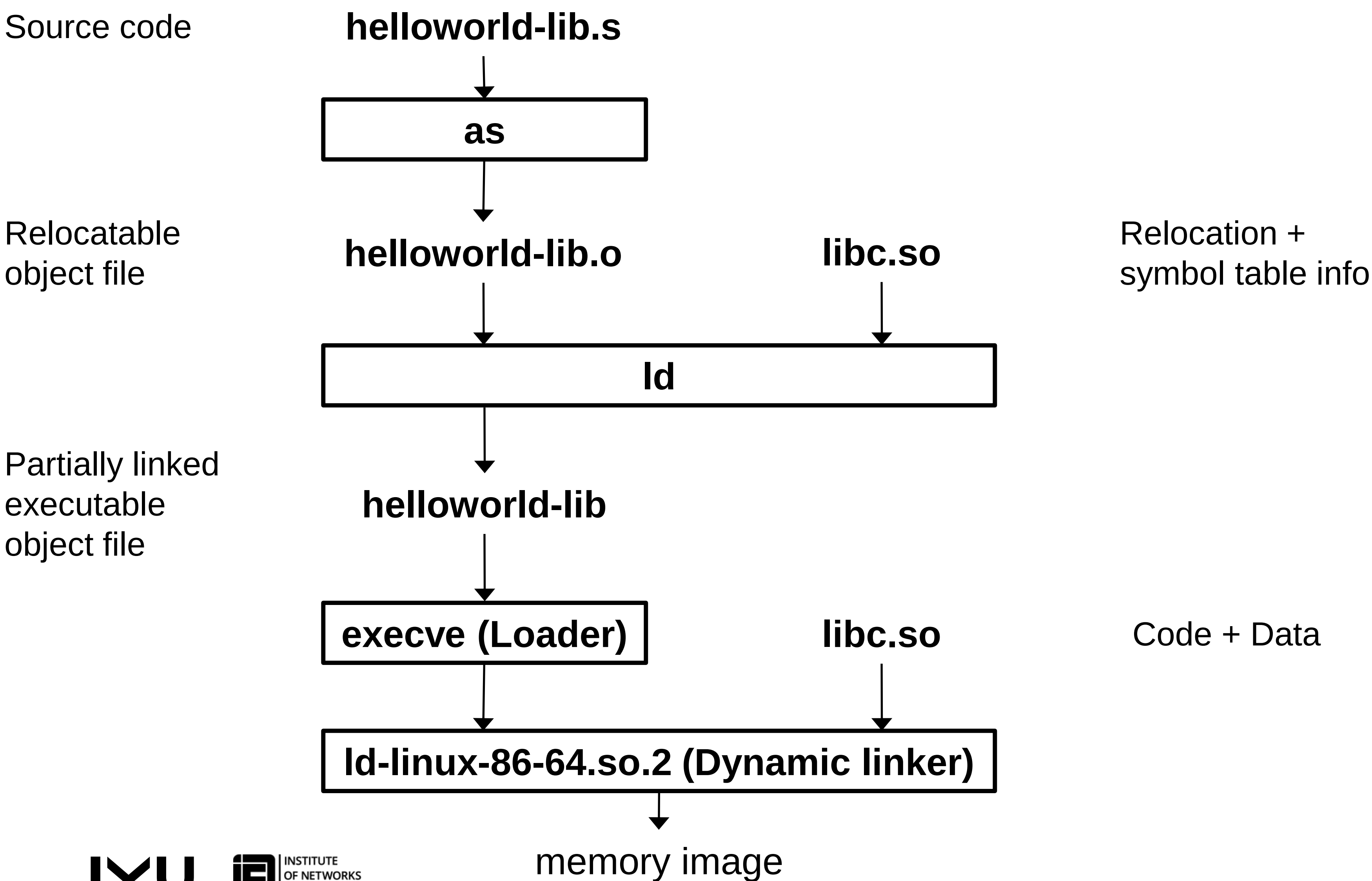
       movq $0,%rdi              # Terminate program
       call exit
```

- Prints the classic „hello world“ and then terminates
  - Both (printing and terminating) are not done directly through the OS anymore, but using the C library

Notes on helloworld-lib.s

- Building and running the program
  1) **`as helloworld-lib.s -o helloworld-lib.o`**
  2) **`ld -dynamic-linker /lib64/ld-linux-x86-64.so.2 -o helloworld-lib helloworld-lib.o -lc`**
     - **`-dynamic-linker /lib64/ld-linux-x86-64.so.2`** allows linking to dynamic libraries
       - This is the part that ensures that the needed libraries are searched, loaded, and adapted when the program is started
     - **`-lc`** is necessary to link to the C library (**`libc.so`** on GNU/Linux)
       - **`printf`**
       - **`exit`**
  3) **`./helloworld-lib`**

Dynamic linking



Library dependencies

- Printing shared dependencies: **`ldd`**
- **`ldd helloworld-lib`**

```
linux-vdso.so.1 =>  (0x00007ffd5bafa000)
libc.so.6 => /lib64/libc.so.6 (0x00007f22b3222000)
/lib64/ld-linux-x86-64.so.2 (0x00007f22b35f7000)
```

- Notes:
  - Addresses may differ on your machine
  - linux-vdso.so.1: Not an actual library, but part of the kernel “injected” into every user-space application
    - To avoid full kernel interface for some very frequently used functions that have to be extremely fast (switch to OS is “expensive”)
    - clock_gettime, getcpu, gettimeofday, time
  - libc.so.6: Basic C library → Used by practically **ANY** program!
    - “.so” → Shared library (DLL)
  - ld-linux-x86-64.so.2: Library for dynamic loading of libraries
    - Dynamic linking (=resolving addresses etc)

factorial-lib.s – Writing a DLL

```
.section .text
.globl factorial
.type factorial,@function
factorial:
        pushq %rbp          # Standard function stuff
        movq  %rsp,%rbp
...
do_recursive:
        pushq %rdi          # Save original value (function
                            # might overwrite it - caller save!)
        decq  %rdi          # Decrease the value
        call  factorial@PLT # Recursively call factorial
                            # Add '@PLT' to force generation of PIC
                            # (Position Independent Code), which
                            # is necessary for shared libraries.
        popq  %rdi          # %rax has the return value, so we reload our
                            # parameter into %rdi
        imulq %rdi,%rax
end_factorial:
        movq  %rbp,%rsp     # Standard function return stuff
        popq  %rbp
        ret                 # Return from the function
```

Notes on factorial-lib.s - PIC

- We have to use a Procedure Linkage Table (PLT) and a Global Offset Table (GOT) – fortunately even in Assembler this is done for us!
  - **`call factorial@PLT`**
- The reason is simple: As a shared library may end up in memory anywhere, it **must** contain **only** Position Independent Code (PIC)
  - No absolute addresses allowed, only relative ones!
    - Or indirect calls via a table different for every process
  - The **`call`** instruction however does not know where "factorial" will end up, so it must be filled in by the loader
    - At every location it occurs (very inefficient)
    - Or once in a table - the Procedure Linkage Table (PLT)
  - Note: In the library (recursive call) this would be easy (relative call) as we know where we start ourselves (in relation), but for the other programs (=factorial-main) this is not possible!
    - No "@PLT" → the library alone assembles perfectly fine, but the linker complains as both PLT and GOT are missing in the object file!

Notes on factorial-lib.s - PIC

- We can disassemble the generated code:
- `objdump –disassemble libfactorial.so`

```
0000000000000220 <factorial@plt-0x10>:
 220:     ff 35 e2 0d 20 00     pushq  0x200de2(%rip)  # 201008 <_GLOBAL_OFFSET_TABLE_+0x8>
 226:     ff 25 e4 0d 20 00     jmpq   *0x200de4(%rip) # 201010 <_GLOBAL_OFFSET_TABLE_+0x10>
 22c:     0f 1f 40 00           nopl   0x0(%rax)
0000000000000230 <factorial@plt>:
 230:     ff 25 e2 0d 20 00     jmpq   *0x200de2(%rip) # 201018 <_GLOBAL_OFFSET_TABLE_+0x18>
 236:     68 00 00 00 00        pushq  $0x0
 23b:     e9 e0 ff ff ff        jmpq   220 <factorial@plt-0x10>
```

- ```
  266:   e8 c5 ff ff ff        callq  230 <factorial@plt>
  ```

- The second is the part for actually calling the function (first line)
  - Second+third line (236, 23b + first block 220-22c) are for lazy binding
    - Lazy binding: Addresses filled in when needed, not at program start
  - `jmpq *????`: Indirect jump to the address specified in the GOT at offset 0x18 (=where the loader will fill in the absolute address where factorial ended up in the memory)
    - The GOT is found at offset 0x201000 (=200de2+236, current RIP offset)
    - = we don't know the absolute address, but we do know "how far away"

32-bit x86 Position Independent Code - It's that bad: http://ewontfix.com/18/
Some Assembly Required*: http://www.mindfruit.co.uk/2012/06/relocations-relocations.html
  ```

factorial-main.s

```
.section .data

.section .text
.globl _start
_start:
        movq  $4,%rdi       # The factorial takes one argument - the
                            # number we want a factorial of.  So, it
                            # gets put into RDI
        call  factorial     # Run the factorial function
        movq  %rax,%rdi     # Factorial returns the answer in %rax, but
                            # we want it in %rdi to send it as our exit
                            # status
        movq  $60,%rax      # Call the kernel's exit function
        syscall
```

- **Exactly** the same as before!

Building a dynamic library

1. Assemble dynamic library
   - `as factorial-lib.s -o factorial-lib.o`
2. Build dynamic library
   - `ld -shared factorial-lib.o -o libfactorial.so`
3. Assemble main code
   - `as factorial-main.s -o factorial-main.o`
4. Link against library and build executable
   - `ld -L . -dynamic-linker /lib/ld-linux-x86-64.so.2 -o factorial-main -lfactorial factorial-main.o`
     - "`-L .`" tells the linker to look for libraries in the current directory

- Library naming: Must always be called "lib"<name of library>".so"
  - Reference in linker: <name of library> only!
    - See C library: -lc and libc.so (libc.so.6 → Versioning)

Using a dynamic library

- Program is built, but cannot run (yet)
  - `./factorial-main: error while loading shared libraries: libfactorial.so: cannot open shared object file: No such file or directory`
  - Tell the dynamic linker that it should also search for libraries in the current directory
    1. `LD_LIBRARY_PATH=.`
    2. `export LD_LIBRARY_PATH`
    - **NEVER ever do this on production systems – This is a huge security problem!**
      - Secure alternative: install library into an OS library directory (but which needs root/administrator permissions)
        - E.g. /usr/lib64 (or /ursr/local/lib64)

JOHANNES KEPLER
UNIVERSITÄT LINZ

INSTITUTE
OF NETWORKS
AND SECURITY

https://www.ins.jku.at

THANK YOU FOR YOUR ATTENTION!

Slides by: Michael Sonntag
michael.sonntag@ins.jku.at
+43 (732) 2468 - 4137
S3 235 (Science park 3, 2nd floor)

JOHANNES KEPLER
UNIVERSITÄT LINZ
Altenberger Straße 69
4040 Linz, Österreich
www.jku.at

JKU