ABIs, linkers and other animals

Stephen Kell

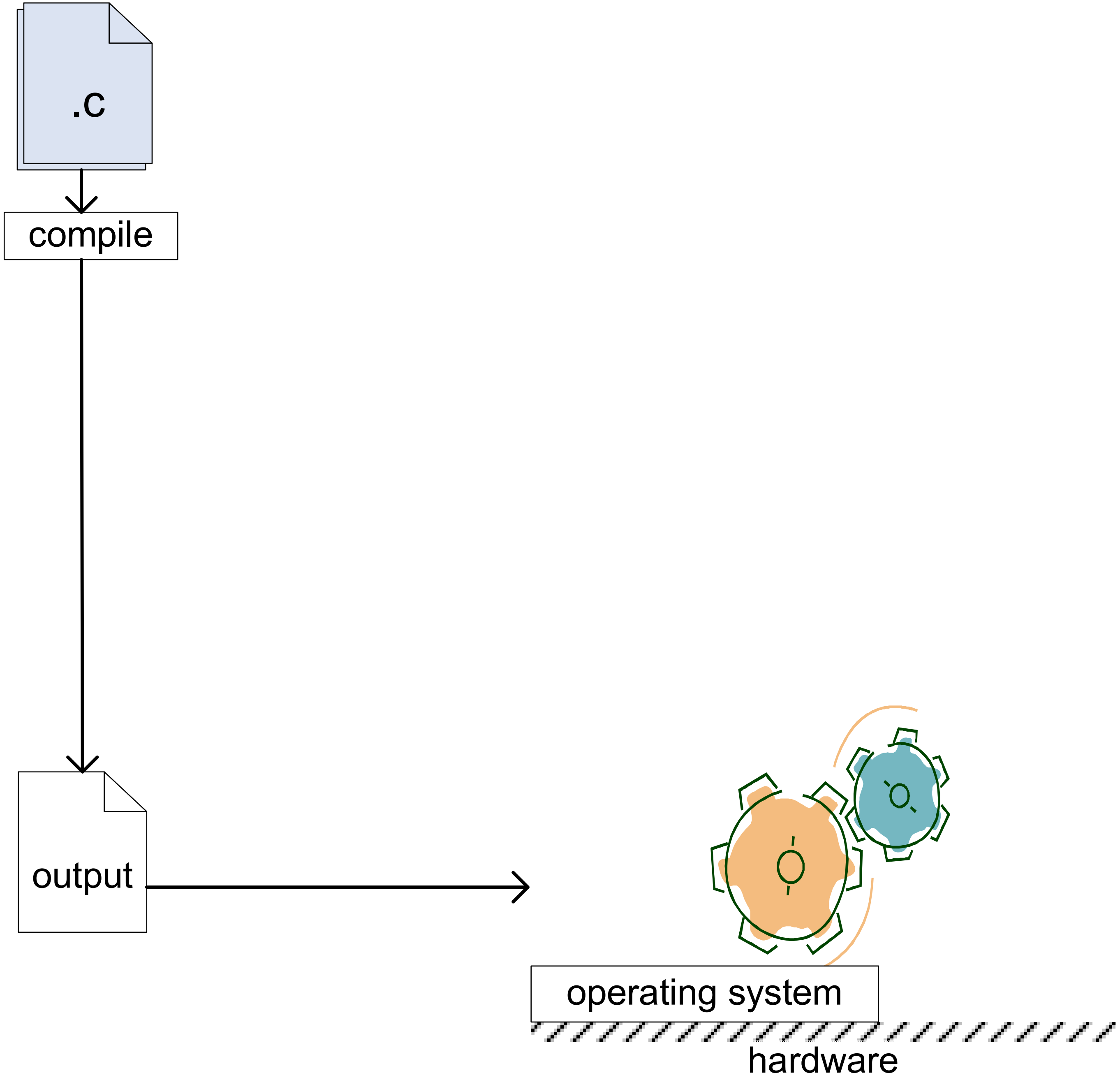
stephen.kell@cl.cam.ac.uk

Computer Laboratory
University of Cambridge

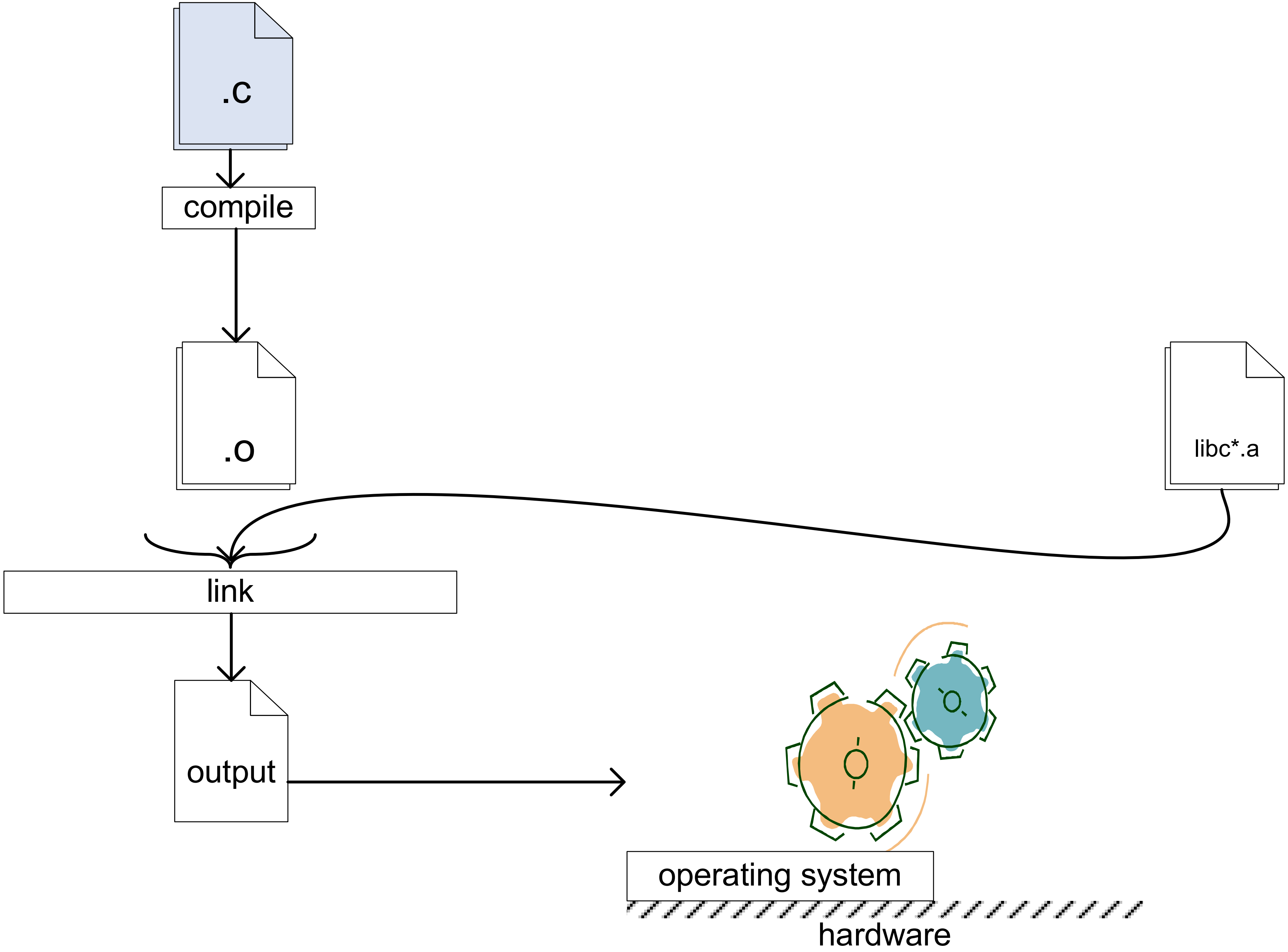
Subject of this talk

- introduce murky artifacts to those unfamiliar
  - ABIs
  - linkers
  - debuggers (a little)
- REMS-flavoured ideas about what to do with them

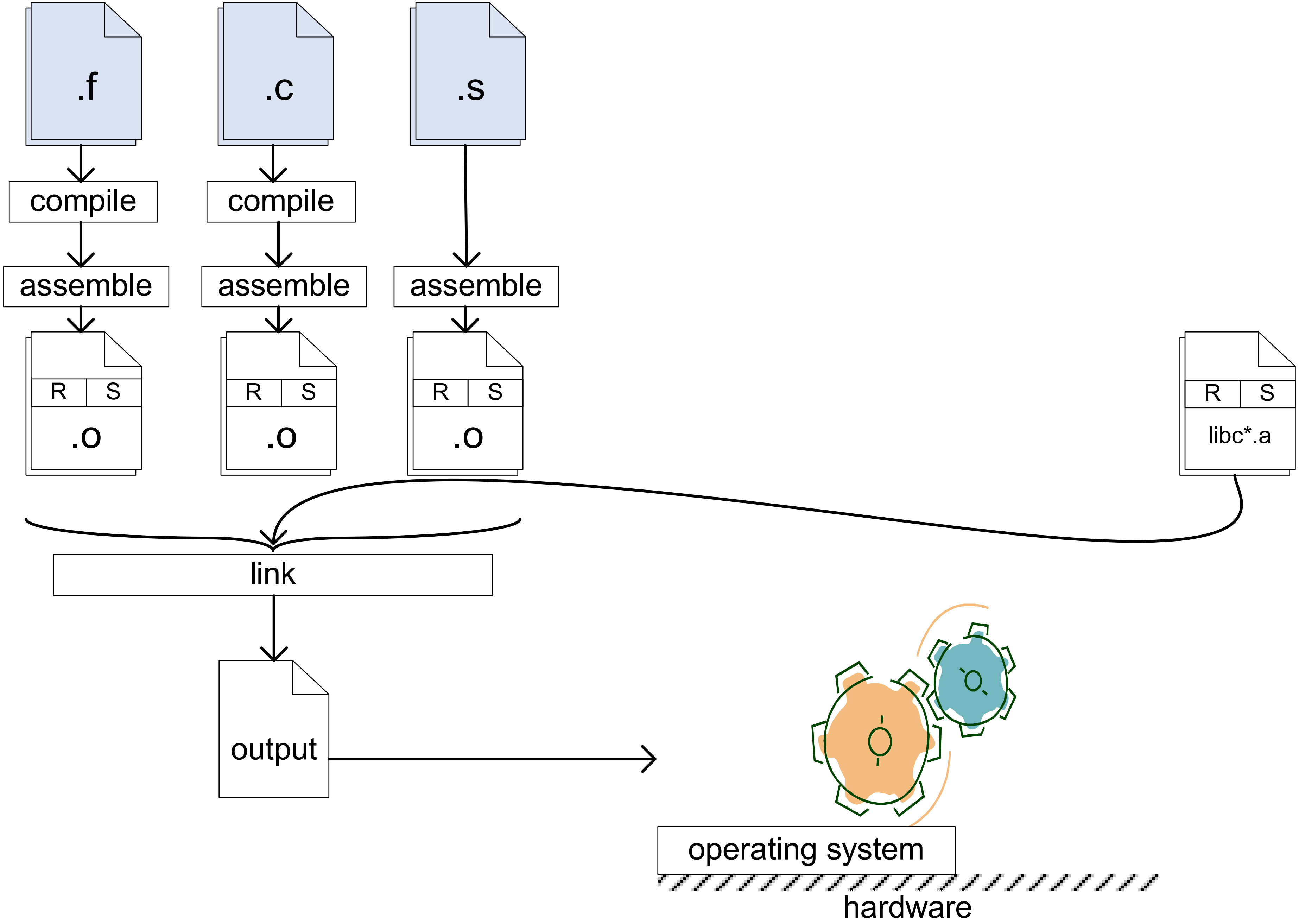
A simplified picture



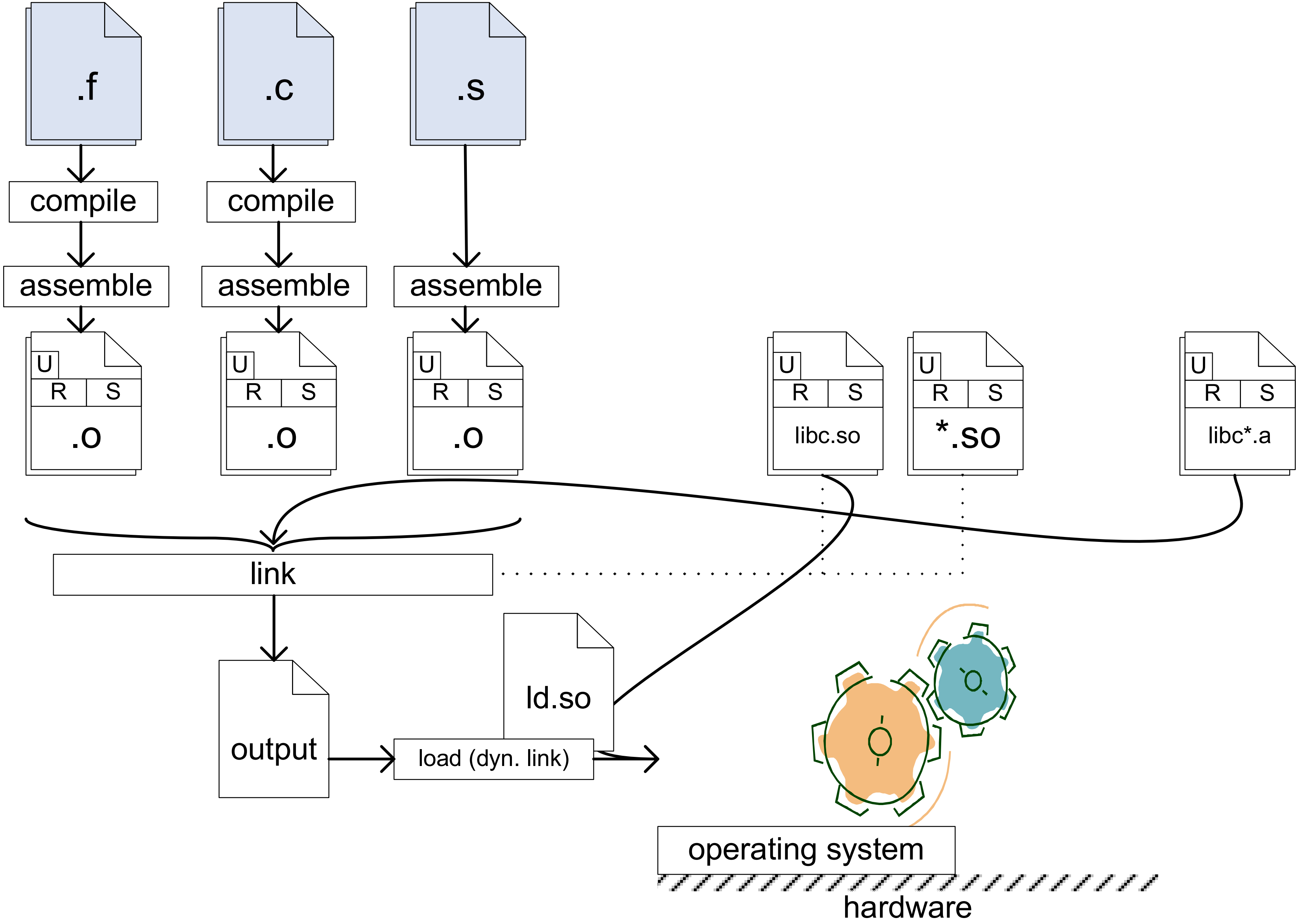
A somewhat more realistic picture



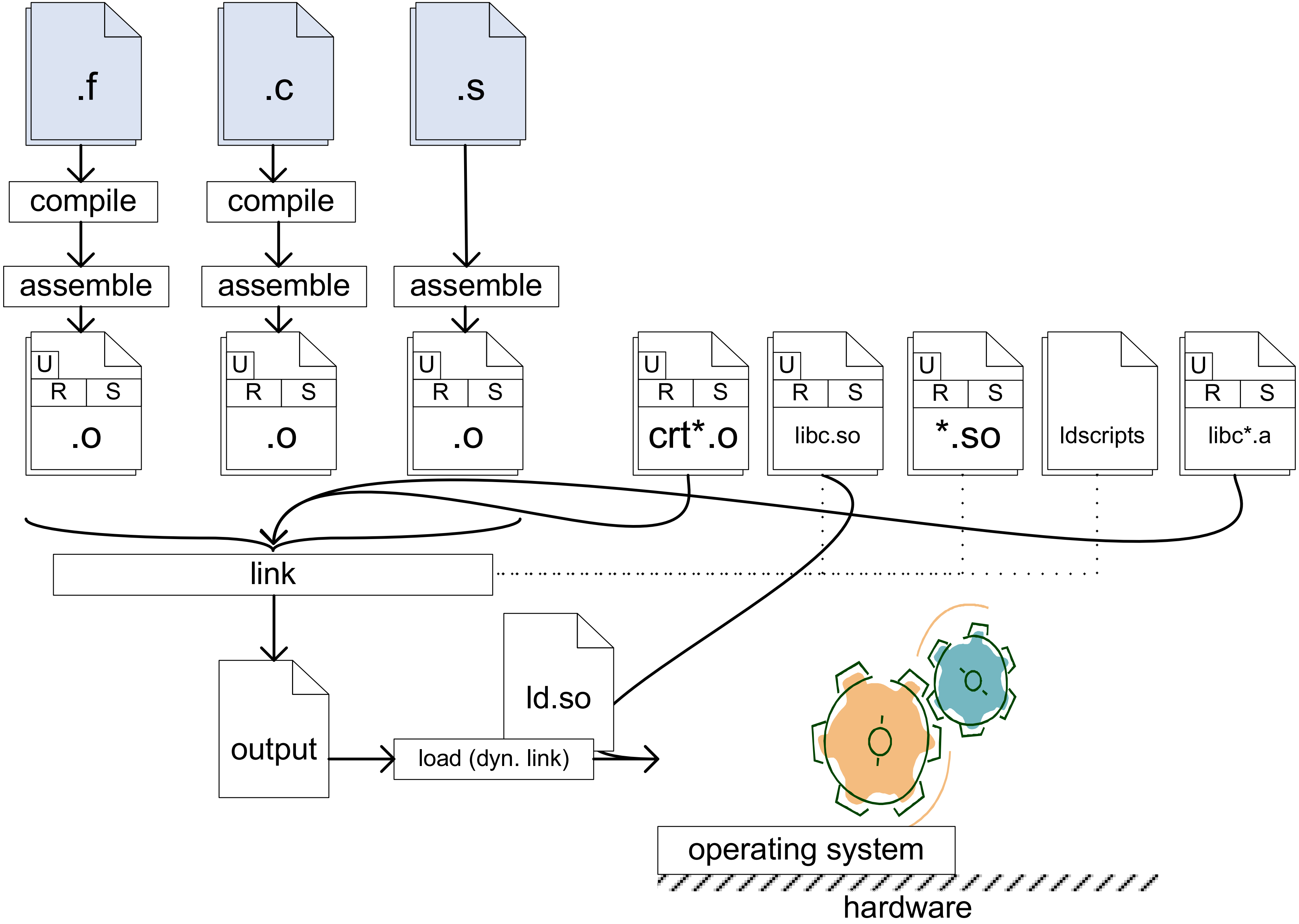
A more realistic picture



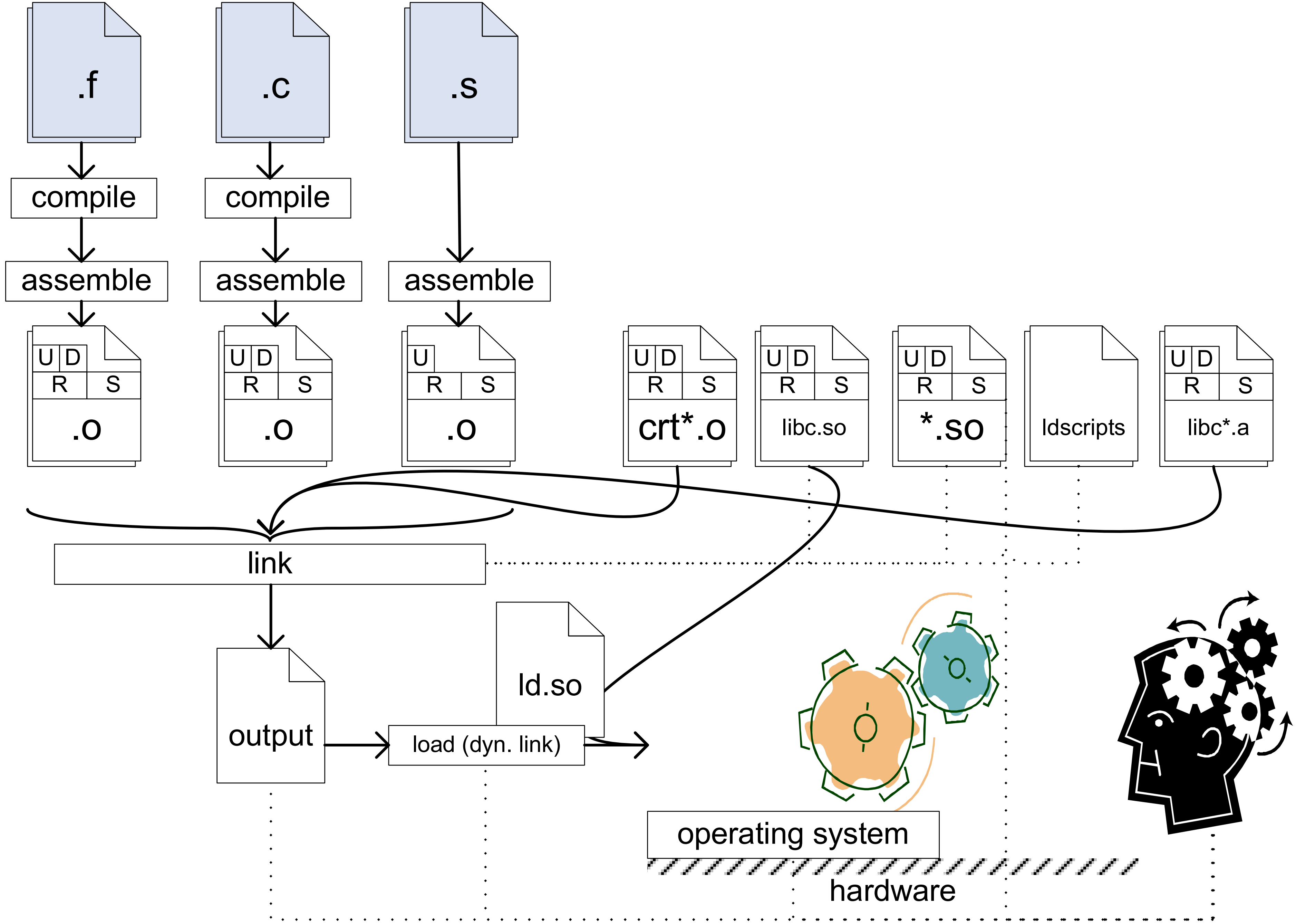
A yet more realistic picture



A yet more, more realistic picture still



A yet more, more realistic picture still, still



Where we're going

- ABIs – the compile-and-link-time part
- linking (static, dynamic)
- ABIs – the load-and-run-time part
- ABIs – cross-language issues
- debugging

Where C leaves off

J.3 Implementation-defined behavior

. . .

J.3.4 Characters

– The number of bits in a byte.

. . .

J.3.5 Integers

– Whether signed integer types are represented using sign and magnitude, two's complement, or ones's complement

. . .

J.3.9 Structures, unions, enumerations, and bit-fields

– The order of allocation of bit-fields within a unit.

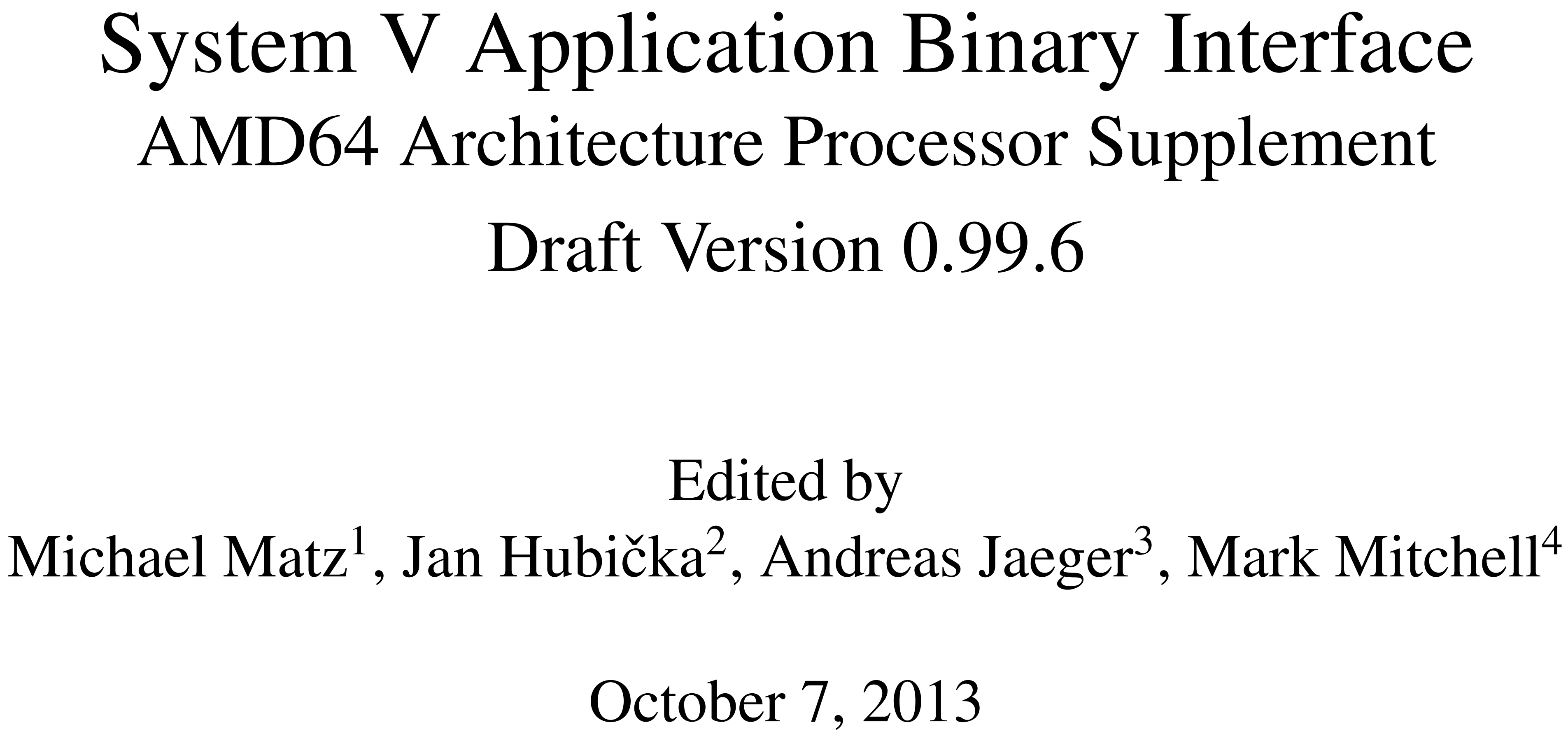
– The alignment of non-bit-field members of structures.

This should present no problem *unless binary data written by one implementation is read by another.*

Things to agree on

- data representation
- register meanings
- calling sequence
- process start-up and shutdown
- object file format & semantics
- system call mechanism
- threading primitive mechanisms
- stack unwinding primitive mechanisms
- hardware exceptions & their delivery
- address-space layout…

You’re going to need an ABI



System V Application Binary Interface

AMD64 Architecture Processor Supplement

Draft Version 0.99.6

Edited by

Michael Matz[1], Jan Hubička[2], Andreas Jaeger[3], Mark Mitchell[4]

October 7, 2013

What’s an ABI?

Application Binary Interface

- conventions for “near-the-metal” interfacing
- usually per-ISA, per-OS-family…
- covers user–user and user–kernel code interactions
- not quite dual to “API”
  - ABIs quantify over a universe of software
- also per-language; usually
  - “the ABI” covers only assembly + C
  - (C++ also has a de facto standard ABI)

Look inside!

Contents

1 Introduction

2 Software Installation

3 Low Level System Information

- 3.1 Machine Interface
- 3.2 Function Calling Sequence
- 3.3 Operating System Interface
- 3.4 Process Initialization

…

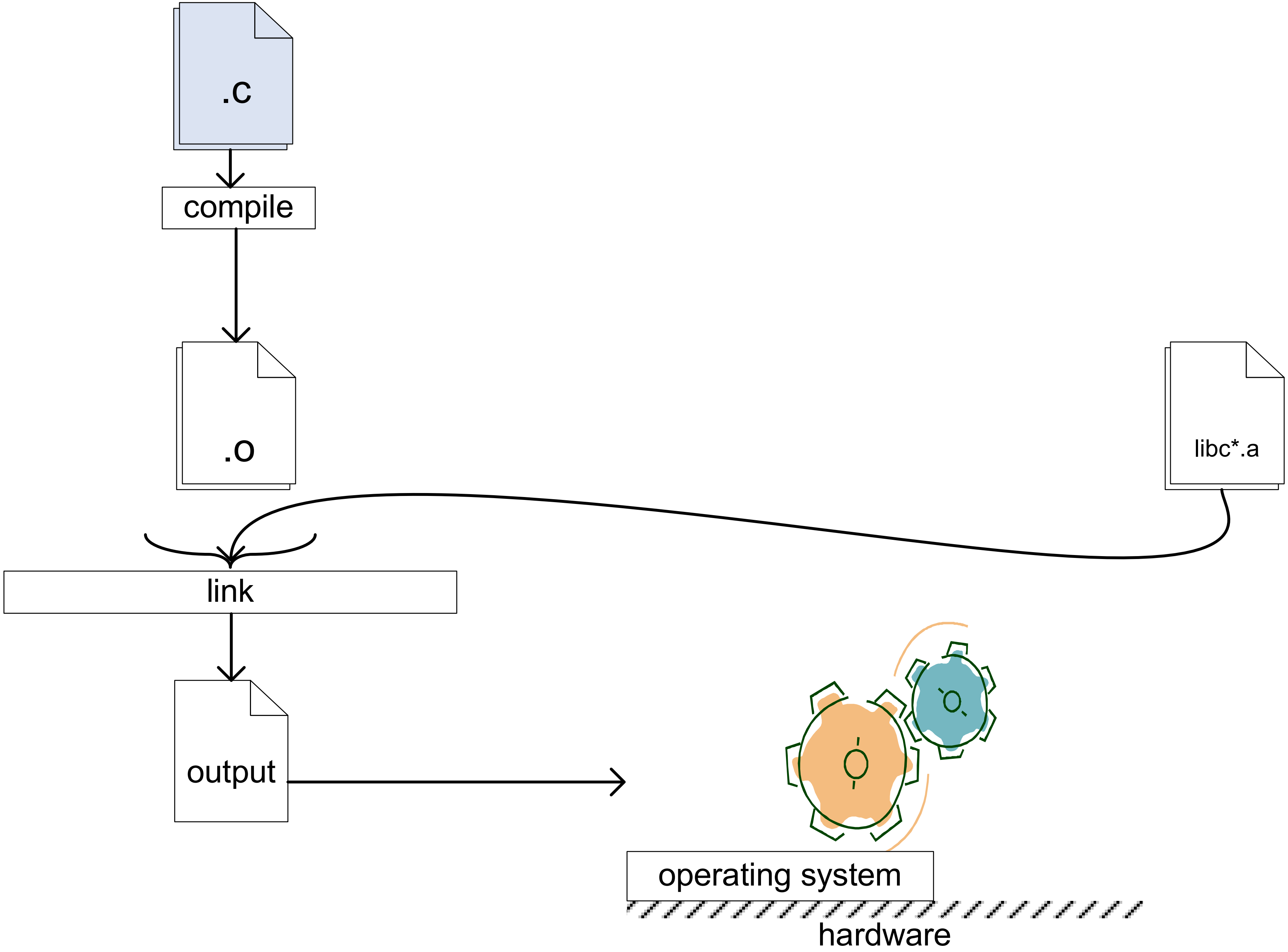
4 Object Files

5 Program Loading and Dynamic Linking

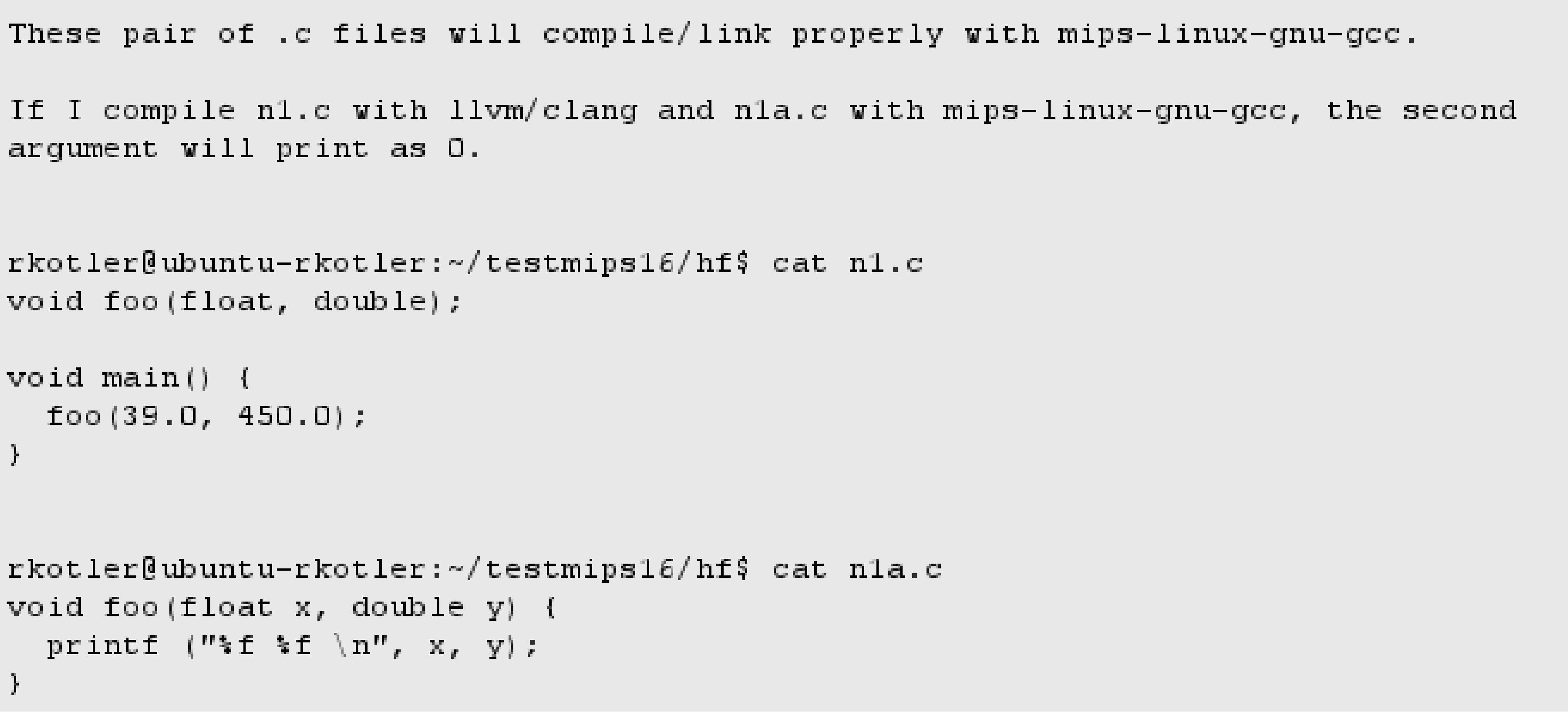
6 Libraries

- 6.1 C Library
- 6.2 Unwind Library Interface

Recall: a simple linking scenario



How it goes wrong: the compiler author’s fault (1)



These pair of .c files will compile/link properly with mips-linux-gnu-gcc.

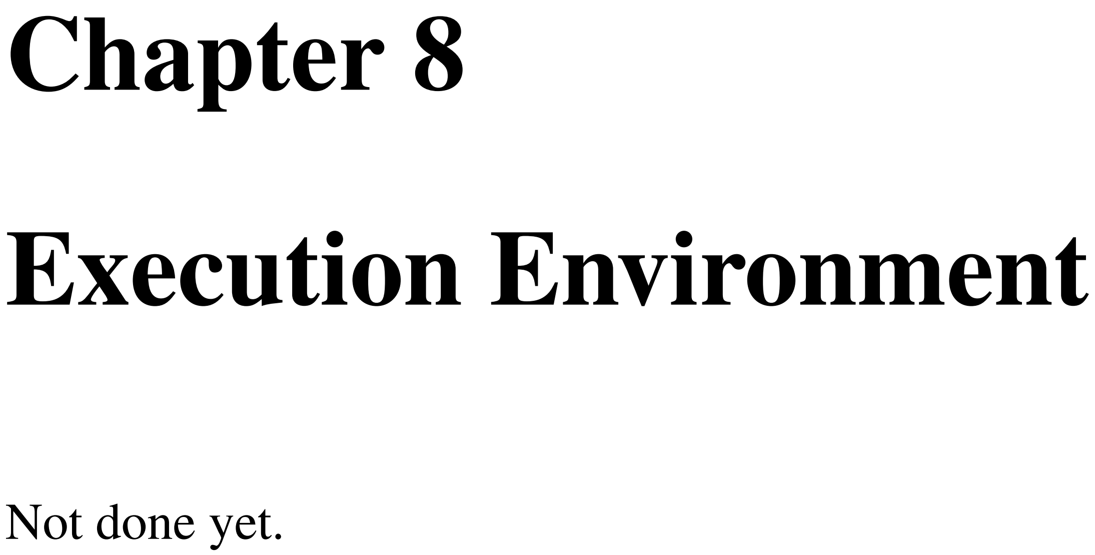
If I compile n1.c with llvm/clang and n1a.c with mips-linux-gnu-gcc, the second argument will print as 0.

```
rkotler@ubuntu-rkotler:~/testmips16/hf$ cat n1.c
void foo(float, double);

void main() {
  foo(39.0, 450.0);
}


rkotler@ubuntu-rkotler:~/testmips16/hf$ cat n1a.c
void foo(float x, double y) {
  printf ("%f %f \n", x, y);
}
```

How it goes wrong: the compiler author’s fault (2)

```
diff --git a/lib/CodeGen/TargetInfo.cpp b/lib/CodeGen/TargetInfo.cpp
--- a/lib/CodeGen/TargetInfo.cpp
+++ b/lib/CodeGen/TargetInfo.cpp
@@ -4020,7 +4020,8 @@
MipsABIInfo::classifyArgumentType(QualType Ty, uint64_t &Offset) const {
    if (Ty->isPromotableIntegerType())
      return ABIArgInfo::getExtend();

-   return ABIArgInfo::getDirect(0, 0, getPaddingType(Align, OrigOffset));
+   return ABIArgInfo::getDirect(0, 0,
+                                IsO32 ? 0 : getPaddingType(Align, OrigOffset));
 }
```

How it goes wrong: the ABI specifier’s fault



Chapter 8

Execution Environment

Not done yet.

Wanted: a formal, complete, precise ABI spec [or subset. . .].

- less obvious omissions aboud
- e.g. x86-64 two’s complement ints

How it goes wrong: the user-level programmer’s fault (1)

```
extern int putchar(int c);
```

Beginner’s mistake!

- `putchar` is a macro in many C libraries
- C APIs are A*P*Is; you *must* do

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

- don’t confuse source with binary!
- more troubling example of this later (interposition)

How it goes wrong: the user-level programmer's fault (2)

```
/* f1.c */
int myfunc(off_t o) {
   /* ... */
}
/* f2.c */
#define _GNU_SOURCE
...
int i = myfunc(o); // off_t has different definition !
```

Ouch. Tools that might help:

- a link-time ABI checker
- what ABI properties are guaranteed by this C file?
- example properties: layout of struct X, size of Y …
  - without headers! (but…)
- environment synthesis…

Linking (1): anatomy of an ELF

```
$ cc -c -o hello.o hello.c && readelf -WS hello.o
[Nr] Name              Type       Addr Off  Size Flg
[ 1] .text             PROGBITS   0    040  020  AX
[ 2] .rela.text        RELA       0    5a0  030
[ 3] .data             PROGBITS   0    060  000  WA
[ 4] .bss              NOBITS     0    060  000  WA
[ 5] .rodata           PROGBITS   0    060  00e   A
[ 6] .comment          PROGBITS   0    06e  02b  MS
[ 7] .note.GNU-stack PROGBITS   0    099  000
[ 8] .eh_frame         PROGBITS   0    0a0  038   A
[ 9] .rela.eh_frame    RELA       0    5d0  018
[10] .shstrtab         STRTAB     0    0d8  061
[11] .symtab           SYMTAB     0    480  108
[12] .strtab           STRTAB     0    588  013
```

This is a *relocatable* ELF…

Linking (2): anatomy of an ELF continued

```
$ readelf -Ws hello.o | egrep -v 'SECTION|FILE'
Symbol table '.symtab' contains 11 entries:
   Num:    Value Size Type  Bind   Vis        Ndx Name
     0: 00000000    0 NOTYPE  LOCAL  DEFAULT  UND
     9: 00000000   24 FUNC    GLOBAL DEFAULT    1 main
    10: 00000000    0 NOTYPE  GLOBAL DEFAULT  UND puts
```

Concepts:

- section: chunk of bytes; “slides as a unit”
  - some have special meaning to the linker
- symbol: a named location in the (eventual) program
- relocation: bytes encoding a reference (pointer)
  - … needing to be fixed up

Linking (2): relocation, relocation, relocation

```
$ objdump -rdS hello.o
...
int main(int argc, char **argv)
{
   0:    48 83 ec 08                sub    $0x8,%rsp
        printf("Hello, world!\n");
   4:    bf 00 00 00 00             mov    $0x0,%edi
                        5: R_X86_64_32  .rodata.str1.1
   9:    e8 00 00 00 00             callq  e <main+0xe>
                        a: R_X86_64_PC32        puts-0x4
        return 0;
}
   e:    b8 00 00 00 00             mov    $0x0,%eax
  13:    48 83 c4 08                add    $0x8,%rsp
  17:    c3                         retq
```

ABIs [loosely] specify many kinds of relocation

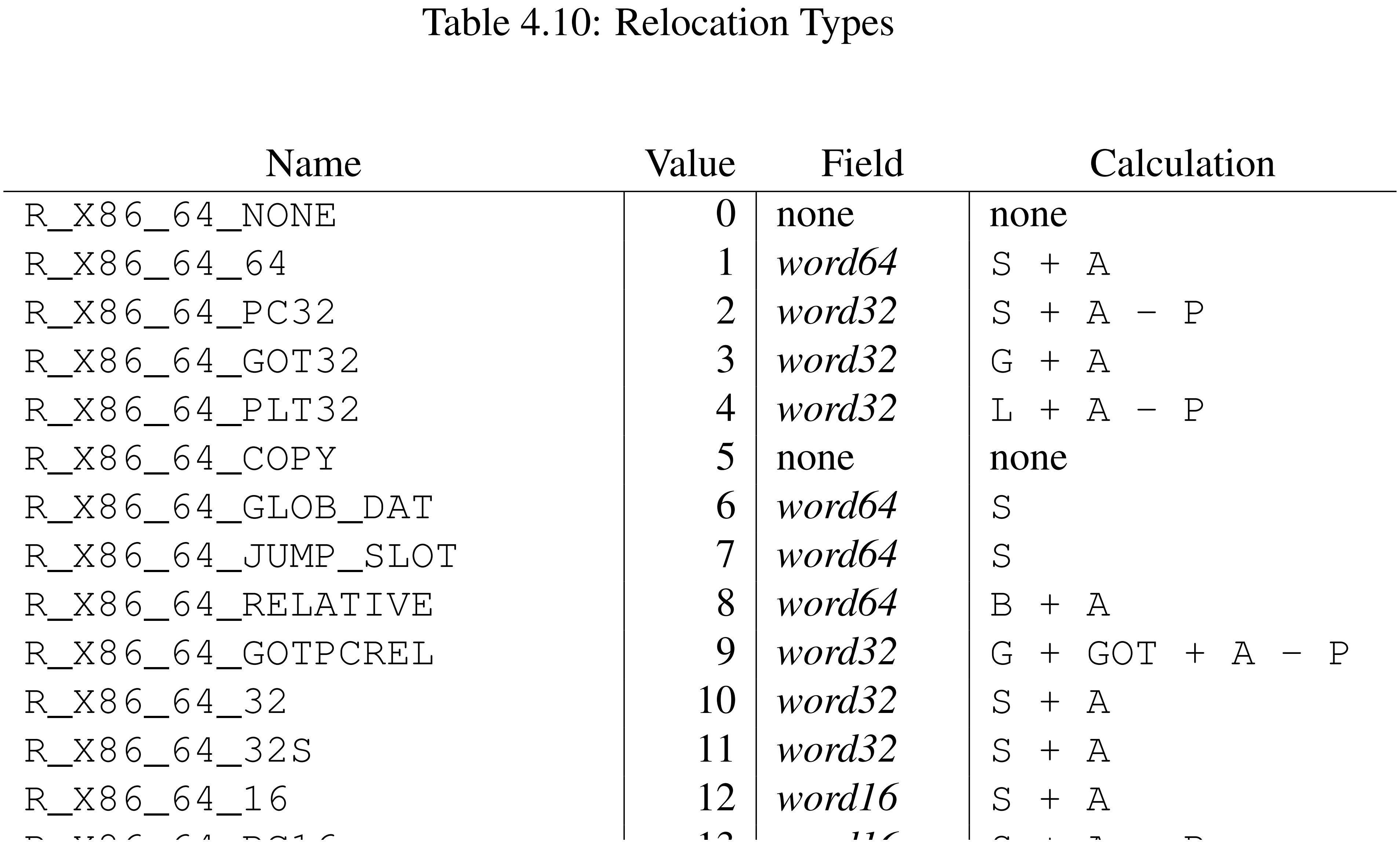


Table 4.10: Relocation Types

Name	Value	Field	Calculation
R_X86_64_NONE	0	none	none
R_X86_64_64	1	*word64*	S + A
R_X86_64_PC32	2	*word32*	S + A - P
R_X86_64_GOT32	3	*word32*	G + A
R_X86_64_PLT32	4	*word32*	L + A - P
R_X86_64_COPY	5	none	none
R_X86_64_GLOB_DAT	6	*word64*	S
R_X86_64_JUMP_SLOT	7	*word64*	S
R_X86_64_RELATIVE	8	*word64*	B + A
R_X86_64_GOTPCREL	9	*word32*	G + GOT + A - P
R_X86_64_32	10	*word32*	S + A
R_X86_64_32S	11	*word32*	S + A
R_X86_64_16	12	*word16*	S + A
R_X86_64_PC16	13	*word16*	S + A - P

Hey—you got your code in my program!

```
$ cc -o hello hello.o && readelf -WS hello
  [Nr] Name      Type     Address  Off    Size   ES Flg
...
  [ 5] .dynsym   DYNSYM   004002b8 0002b8 000060 18   A
...
  [ 9] .rela.dyn RELA     00400380 000380 000018 18   A
...
  [13] .text     PROGBITS 00400440 000440 0001a4 00  AX
...
  [15] .rodata   PROGBITS 004005f0 0005f0 000012 00   A
...
  [24] .data     PROGBITS 00601030 001030 000010 00  WA
  [25] .bss      NOBITS   00601040 001040 000008 00  WA
```

Gained 0x164 bytes text, 4 rodata, 16 data, 8 bss

crt*.o and libgcc files

```
$ cc -### -o hello hello.o      # + simplified somewhat!
/usr/lib/gcc/x86_64-linux-gnu/4.7/collect2
  -m elf_x86_64
  --hash-style=gnu
  -dynamic-linker /lib64/ld-linux-x86-64.so.2
  -o hello
  /usr/lib/x86_64-linux-gnu/crt1.o
  /usr/lib/x86_64-linux-gnu/crti.o
  /usr/lib/gcc/x86_64-linux-gnu/4.7/crtbegin.o
  hello.o
  -lgcc
  -lgcc_s
  -lc
  /usr/lib/gcc/x86_64-linux-gnu/4.7/crtend.o
  /usr/lib/x86_64-linux-gnu/crtn.o
```

Is that everything, then?

```
$ cat /usr/lib/x86_64-linux-gnu/libc.so
/* GNU ld script
   Use the shared library, but some functions are only in
   the static library, so try that secondarily.  */
OUTPUT_FORMAT(elf64-x86-64)
GROUP ( /lib/x86_64-linux-gnu/libc.so.6
  /usr/lib/x86_64-linux-gnu/libc_nonshared.a
  AS_NEEDED ( /lib/x86_64-linux-gnu/ld-linux-x86-64.so.2 ) )
```

What’s in the startup files, libgcc, …?

Process initialization

- what happens between _start and main()
- initialize C library state
  - environ (from auxv), malloc() (global data)
  - transactional memory stuff
- hooks for some tools (__gmon_start__)
- call user-defined constructor functions

Process shutdown similarly…

libgcc: out-of-line impls of compiler intrinsics

libc_nonshared.a: a few C library functions

What linkers do (1)

Combine like-named sections, in a variety of ways

- concatenate
- merge
- merge + sort
- discard all but one

Resolve references, as they go

- i.e. fixup relocation sites
- by *resolving symbols* in input objects
- … accounting for symbol *binding* and *visibility*
- but must retain interposability!

What linkers do (2)

Organise the address space according to a “code model”

- models constrain compiler w.r.t. addressing modes
- e.g. x86-64 defines Kernel, Small, Medium, Large
  - + position-independent (PIC) variants of S, M and L
- some models require support structures
  - generated by the linker!
  - guided by compiler-generated relocation records

Code models enable shared libraries to be “shared” (or not!)

Actually sharing shared libraries

```
$ cc -shared -o libhello.so hello.o
/usr/bin/ld: hello.o: relocation R_X86_64_32 against `.rodata.str1.1’
can not be used when making a shared object; recompile with -fPIC
```

Embedding addresses makes code non-shareable!

```
$ cc -O -c -fPIC -o hello.o hello.c && objdump -rdS hello.o
0000000000000000 <main>:
   0:   48 83 ec 08             sub    $0x8,%rsp
   4:   48 8d 3d 00 00 00 00    lea    0x0(%rip),%rdi
                        7: R_X86_64_PC32        .LC0-0x4
   b:   e8 00 00 00 00          callq  10 <main+0x10>
                        c: R_X86_64_PLT32       puts-0x4
  10:   b8 00 00 00 00          mov    $0x0,%eax
  15:   48 83 c4 08             add    $0x8,%rsp
  19:   c3                      retq
```

It’s not over yet…

```
$ cc -shared -o libhello.so hello.o && objdump -rdS libhello.
(snip!)
00000000000006c0 <main>:
 6c0:    48 83 ec 08                sub    $0x8,%rsp
 6c4:    48 8d 3d 1a 00 00 00       lea    0x1a(%rip),%rdi
 6cb:    e8 e0 fe ff ff             callq  5b0 <puts@plt>
 ...
```

Q. What’s this PLT thing?

```
00000000000005b0 <puts@plt>:
 5b0: ff 25 62 0a 20 00 jmpq ⋆0x200a62(%rip) # .got.plt+0x18
 5b6: 68 00 00 00 00    pushq $0x0
 5bb: e9 e0 ff ff ff    jmpq  5a0 <_init+0x28>
```

A. a tortuous (lazy) position-independent linking device…

Take-home about code models

Compiler and linker collaborate on

- what code & relocations the compiler generates
- how the linker transforms them
- proof-of-pudding: the desired sizing & shareability
- … without unnecessary performance penalty

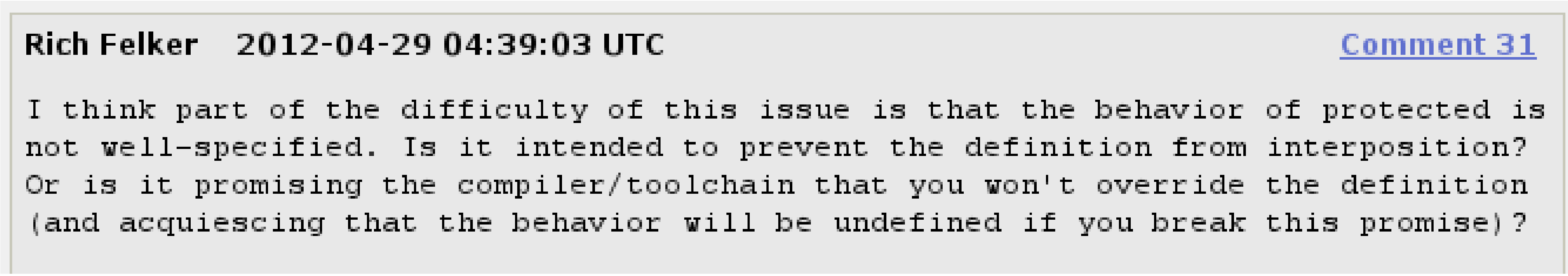
Bugs tend to be in the compiler. There May Be Bugs here.

- wanted: from formal ISA (+ ABI) spec, proof that…
  - code is correct …
  - … w.r.t. ABI's binding & interposability semantics
  - + is no more indirected than necessary

An interesting bug

ELF “protected” symbol visibility bug in gcc (#19520)

- 9 years old and counting!
- test case: do these two function pointers compare equal?
- note: this is a compiler bug, not a linker bug



Rich Felker 2012-04-29 04:39:03 UTC Comment 31

I think part of the difficulty of this issue is that the behavior of protected is not well-specified. Is it intended to prevent the definition from interposition? Or is it promising the compiler/toolchain that you won't override the definition (and acquiescing that the behavior will be undefined if you break this promise)?

Section combining is configured by a linker script

```
/* Default linker script, for normal executables */
OUTPUT_FORMAT("elf64-x86-64", "elf64-x86-64",
              "elf64-x86-64")
OUTPUT_ARCH(i386:x86-64)
ENTRY(_start)
SEARCH_DIR("/usr/x86_64-linux-gnu/lib64"); SEARCH_DIR("=/usr/
SECTIONS
  /* Read-only sections, merged into text segment: */
  PROVIDE (__executable_start = SEGMENT_START("text-segment",
  .interp         : { *(.interp) }
  .note.gnu.build-id : { *(.note.gnu.build-id) }
  .hash           : { *(.hash) }
  .gnu.hash       : { *(.gnu.hash) }
  .dynsym         : { *(.dynsym) }
  .dynstr         : { *(.dynstr) }
```

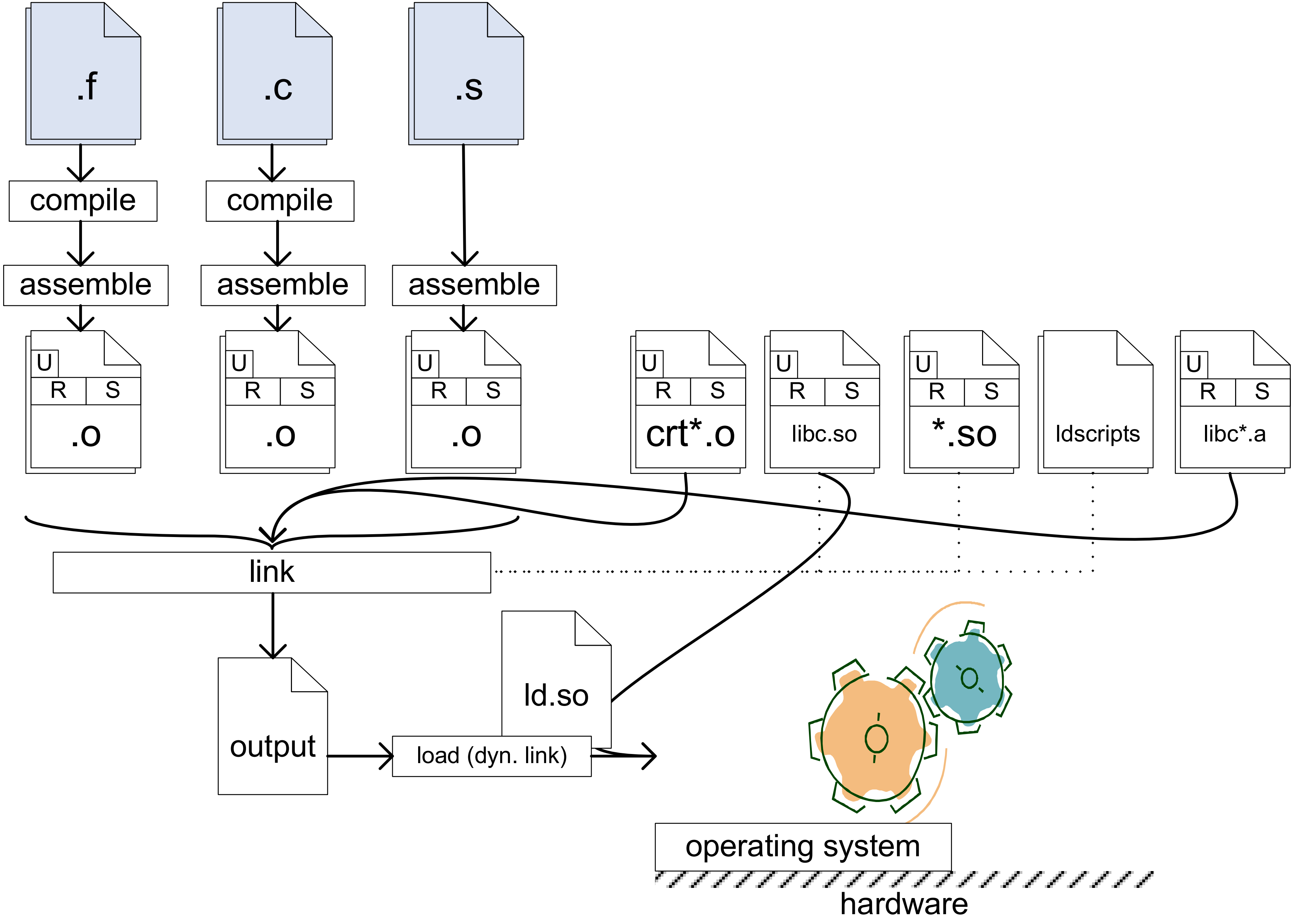
The implementation is the specification

Linkers are full of not-written-downs

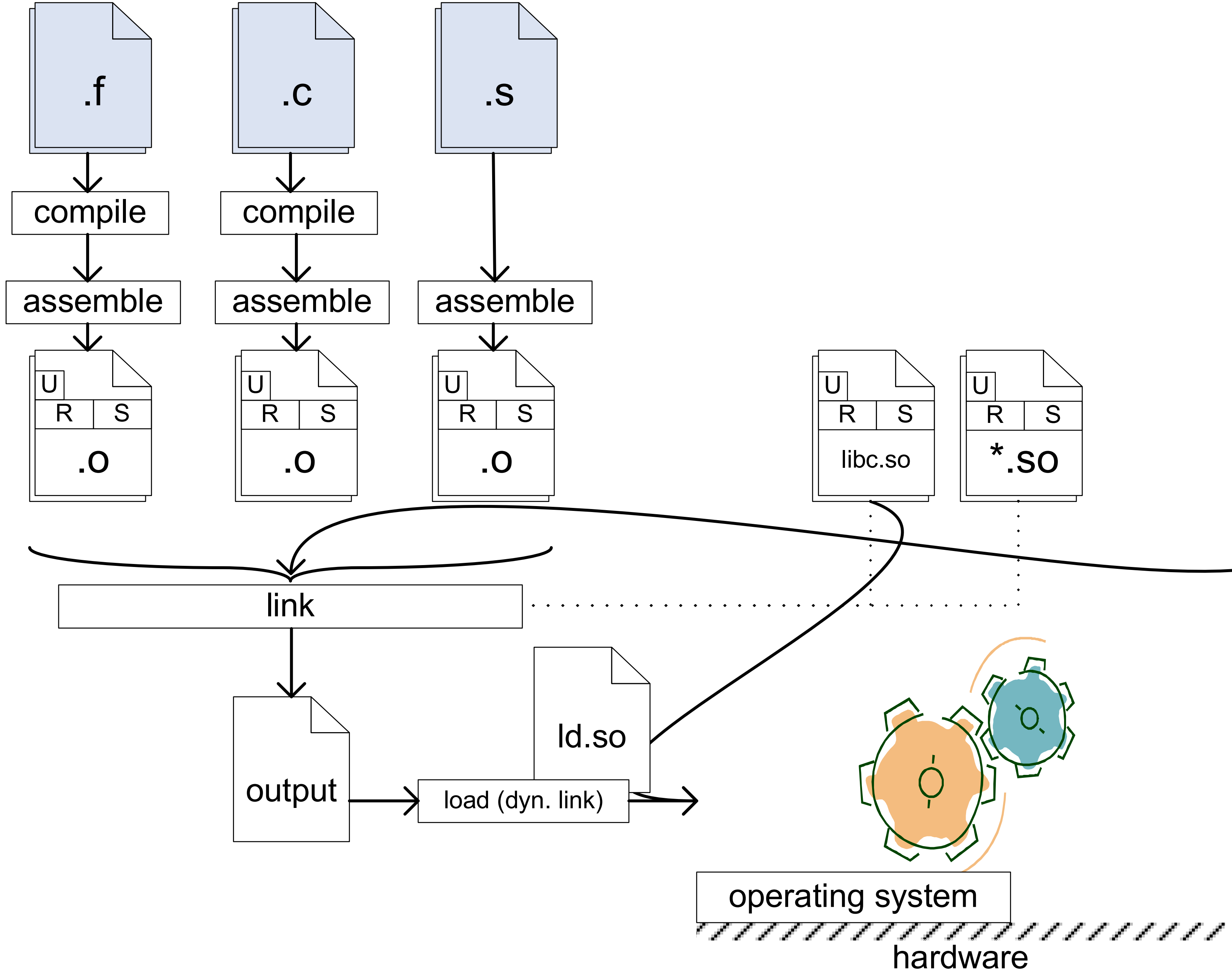
- script language is vaguely standardised
- encode many ABI details, but also
- section names map to meanings, many *not* ABI-defined
  - vendor extensions “for all vendors we can think of”
  - things the ABI left undefined, e.g. debugging
- symbol versioning is not standardised
  - works via user-supplied scripts

Despite this, bugs are *relatively* few…

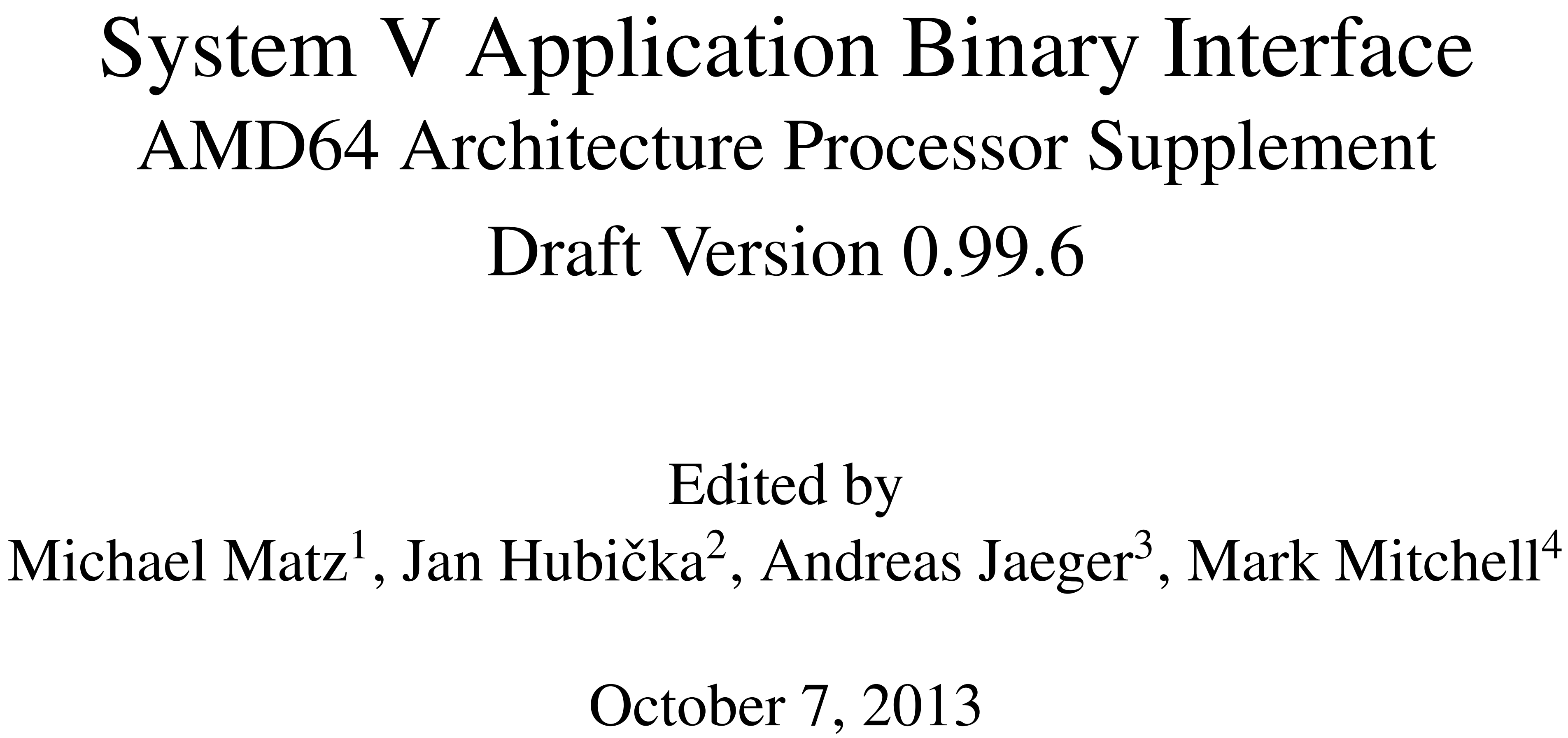
Recap (1)


.f
.c
.s
compile
compile
assemble
assemble
assemble
U
R
S
.o
U
R
S
.o
U
R
S
.o
U
R
S
crt*.o
U
R
S
libc.so
U
R
S
*.so
ldscripts
U
R
S
libc*.a
link
output
ld.so
load (dyn. link)
operating system
hardware

Recap (2)


.f
.c
.s
compile
compile
assemble
assemble
assemble
U
R
S
.o
U
R
S
.o
U
R
S
.o
U
R
S
libc.so
U
R
S
*.so
U
R
S
libc*.a
link
output
ld.so
load (dyn. link)
operating system
hardware

Recap (3)



System V Application Binary Interface

AMD64 Architecture Processor Supplement

Draft Version 0.99.6

Edited by

Michael Matz[1], Jan Hubička[2], Andreas Jaeger[3], Mark Mitchell[4]

October 7, 2013

Recap (4)

```
$ cc -o hello hello.o && readelf -WS hello
  [Nr] Name      Type     Address  Off    Size   ES Flg
...
  [ 5] .dynsym   DYNSYM   004002b8 0002b8 000060 18   A
...
  [ 9] .rela.dyn RELA     00400380 000380 000018 18   A
...
  [13] .text     PROGBITS 00400440 000440 0001a4 00  AX
...
  [15] .rodata   PROGBITS 004005f0 0005f0 000012 00   A
...
  [24] .data     PROGBITS 00601030 001030 000010 00  WA
  [25] .bss      NOBITS   00601040 001040 000008 00  WA
```

Different kinds of linking

Relocatable-to-relocatable linking

- make a bigger `.o` out of one or more `.o`s
- comparatively rare
- done by “static” a.k.a. “compile-time” linker

“Final” linking

- produce a loadable object (shared lib or executable)
- assign address space, discard some relocations…
- also done by “compile-time” linker

Dynamic linking, dynamic loading

- by “dynamic linker”, “loader”, “run-time linker”…
- map binaries into memory, fix up, initialize

Dynamic linking as interpretation

```
$ ./hello
Hello, world!
$ readelf -WS hello  | grep interp
  [ 1] .interp   PROGBITS  00400238 000238 00001c 00   A
$ hexdump -c hello -s $(( 0x238 )) -n $(( 0x1c ))
0000238   / l i b 6 4 / l d - l i n u x -
0000248   x 8 6 - 6 4 . s o . 2 \0
$ /lib64/ld-linux-x86-64.so.2
Usage: ld.so [OPTION]... EXECUTABLE-FILE [ARGS-FOR-PROGRAM...
You have invoked `ld.so', the helper program for shared libra
(snip)
$ /lib64/ld-linux-x86-64.so.2 ./hello
Hello, world!
```

Loading a program with shared libraries

Another round of linking

- “dynamic linking”, “run-time linking”
- more strictly specified by the ABI, cf. static linking
- e.g. x86-64 prescribes relocations-with-addends

Otherwise similar to “compile-time” (sic) linking, *except*. . .

- choose a load address for each object
- dependency search (+ transitive closure)

```
$ ldd hello
    linux-vdso.so.1 =>  (0x00007fff0c768000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f460
    /lib64/ld-linux-x86-64.so.2 (0x00007f46011d4000)
```

ELF as a module system

- modules specify dependencies
- symbols form a def–use relation
- … and have visibility attributes (twice over)
- modules specify initialization and finalization logic
- globally-visible ELF symbol definitions are *interposable*
  - enables executable to override library, e.g. malloc()
  - enables preloaded libraries to override other libs (LD_PRELOAD)
- → mixin layers-style composition model (Smaragdakis)
- every (d-l'd) ELF process includes an "ELF runtime"…

The ELF runtime

Safe assumptions are compile time

- each shared object has a “load address”
- symbols mark locations of interest (etext, edata, end)
- structures necessitated by code model (GOT, PLT)

libdl is the run-time interface

- dlopen(filename, mode) loads+links a library
- dlsym(handle, symname) looks up a symbol in it
- think: plugin systems

Per-implementation extensions fill some gaps

- e.g. walking the link map

Interposition and forwarding (1)

Symbol interposition adds value: can override libraries

- fakeroot, tsocks, aoss, padsp

. . . and also for diagnostic-style tools

- catchsegv, ltrace, early versions of Valgrind

. . . and more elaborate things (blcr, . . .).

Interposition and forwarding (2)

Basic idea: `$ LD_PRELOAD=libmylib.so my-command`

```
int (*orig_stat)(const char *path, struct stat *buf);
void init() { orig_stat = dlsym(RTLD_NEXT, "stat"); // fails!
}
int stat(const char *path, struct stat *buf)
{
    fprintf(stderr, "stat() called\n");
    return orig_stat(path, buf);
}
```

This doesn’t work!

- binary interfaces are implementation details!

A real bug

```
--- a/alsa/alsa-oss.c
+++ b/alsa/alsa-oss.c
@@ -69,6 +69,7 @@
 static int (*_open)(const char *file, int oflag, ...);
+static int (*___open_2)(const char *file, int oflag);
 static int (*_open64)(const char *file, int oflag, ...);
@@ -819,6 +840,7 @@
        _open64 = dlsym(RTLD_NEXT, "open64");
+       ___open_2 = dlsym(RTLD_NEXT, "__open_2");
        _close = dlsym(RTLD_NEXT, "close");
@@ -312,6 +313,25 @@
 DECL_OPEN(open, _open)
 DECL_OPEN(open64, _open64)
+int __open_2(const char *file, int oflag)
+{
+       mode_t mode = 0;
```

ABIs for language pluralism (1): the SysV-AMD64 exception ABI

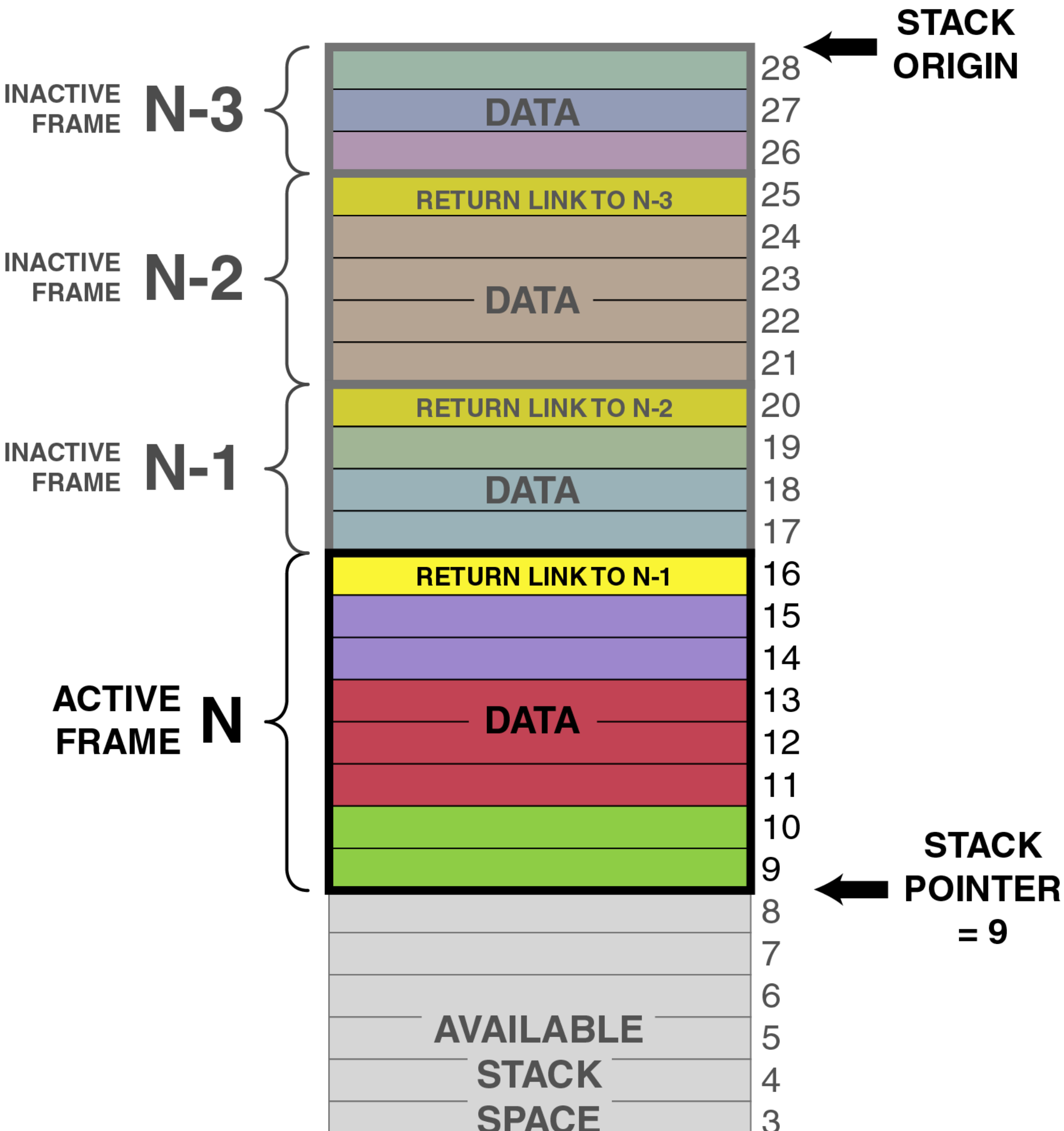
An elaborate ABI exists for cross-language exceptions

- throw through foreign frames
- can catch even foreign exceptions
- clean up each frame appropriately (e.g. C++ destructors)
- supported by: most major C, C++, Fortran, Ada impls
- not: most Java impls, OCaml (though. . . ?), . . .

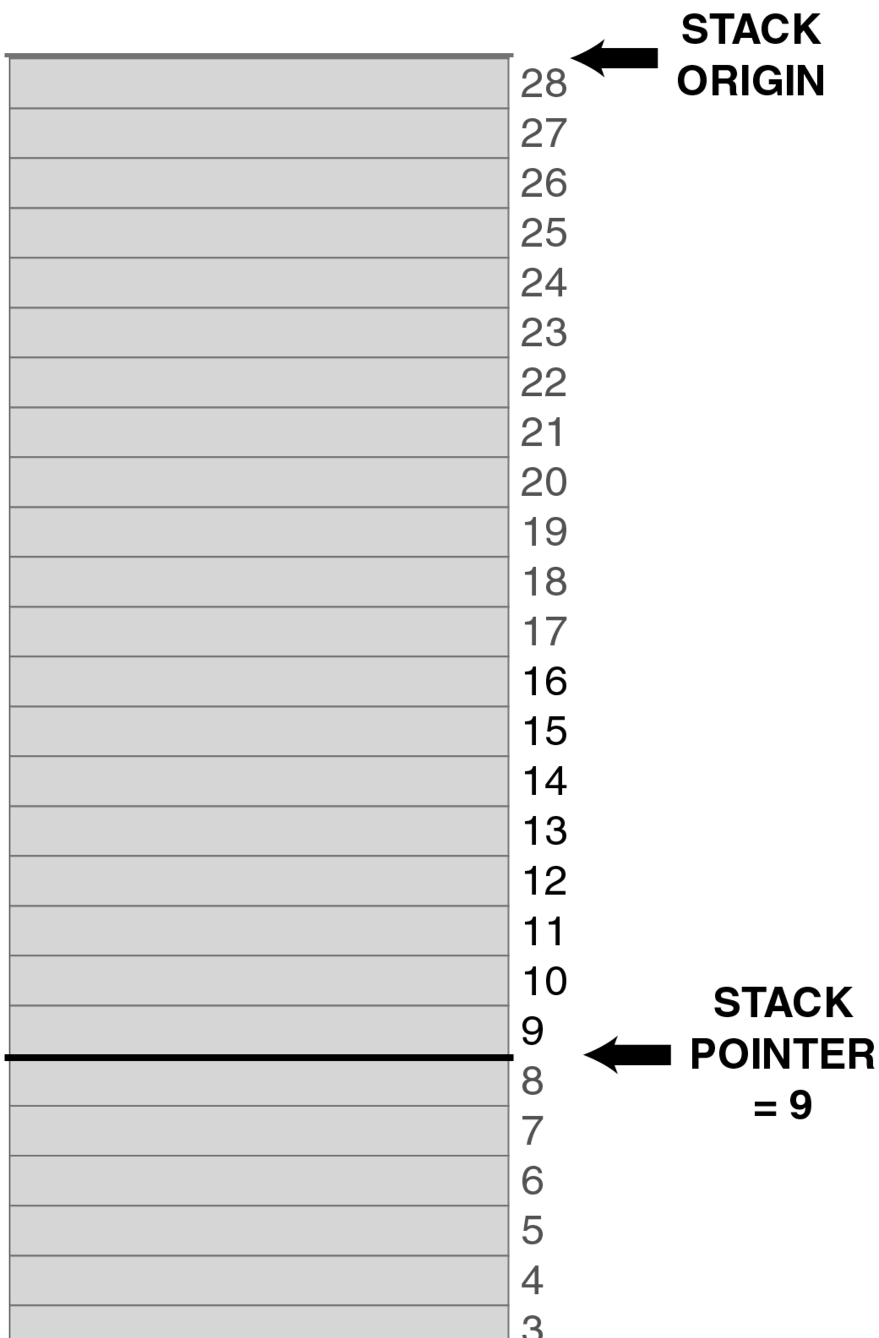
A few elements:

- common format for unwind information
- per-language “personality routine” + data area
- two-phase algorithm (first look, then go)

Unwind information (0)



Unwind information ($\frac{1}{2}$)



Unwind information (1)

```
$ readelf -wF hello.o
(snip)
0018 0014 001c FDE cie=0000 pc=0000..0018  # hint: main()
   LOC              CFA      ra
0000000000000000 rsp+8    c-8
0000000000000004 rsp+16   c-8
0000000000000017 rsp+8    c-8
```

All because the function does

```
   0:   48 83 ec 08       sub    $0x8,%rsp
   4:   bf 00 00 00 00    mov    $0x0,%edi     # "Hello...
   9:   e8 00 00 00 00    callq  e <main+0xe>  # puts
   e:   b8 00 00 00 00    mov    $0x0,%eax
  13:   48 83 c4 08       add    $0x8,%rsp
  17:   c3                retq
```

Unwind information (2)

```
$ readelf -wf hello.o
0000 0014 0000 CIE
  Version:               1
  (snip)
  DW_CFA_def_cfa: r7 (rsp) ofs 8
  DW_CFA_offset: r16 (rip) at cfa-8
  DW_CFA_nop
  DW_CFA_nop

0018 0014 001c FDE cie=0000 pc=0000..0018
  DW_CFA_advance_loc: 4 to 0004
  DW_CFA_def_cfa_offset: 16
  DW_CFA_advance_loc: 19 to 0017
  DW_CFA_def_cfa_offset: 8
  DW_CFA_nop
```

ABIs across languages

“Platform” ABIs cover C and assembly

- … maybe Fortran too

Other languages tend to layer over C

- … hence (transitively) over host ABI!
- a C++ ABI is well established (Itanium)
- Objective-C comparable (has “older, old, new” ABIs)
- JNI is a binary interface (but not used VM-internally)

ABIs and FFIs

∃ big similarities between ABIs and FFIs

- both concerned with separate compilation
- FFIs more directional (more tyrannical)
- … usually for no good reason (ask me)

∃ case for tooling them the same way

- avoid manually repeating interfaces once per language
- allow co-development
- (ask me)

Cross-language thoughts: ABI pluralism

Enforcing a single ABI for all languages is unlikely. But

- describing [families of] ABIs is very desirable
- 'compatibility' ABIs exist (-fpcc-struct-return)

Wanted:

- tools to make it easy to target an ABI
- tools to specify ABI extensions

If we can describe ABIs, we can synthesise glue code!

- tools to do the synthesis
- tools to specify ABI *non-extensions*
  - don't program against them, but synthesis is okay

Extending ABIs to would-be sophisticates

ABIs + garbage collection is an unaddressed issue

- need pointer maps, safepoints, …

Cross-language ABIs need a clever object layout model

- don’t assume headers; don’t assume contiguity!

Most VMs are too stupid at present…

- ABI-based compilers are more sophisticated
  - ELF also has fancy object model
  - recall gcc bug!
- (ask me about “fragments” versus “objects”…)

Implementing debugging: two approaches

- “VM-style” vs “ABI-style”

VM: provide debug server in runtime

- expedient but prescriptive
- no multi-language debugging

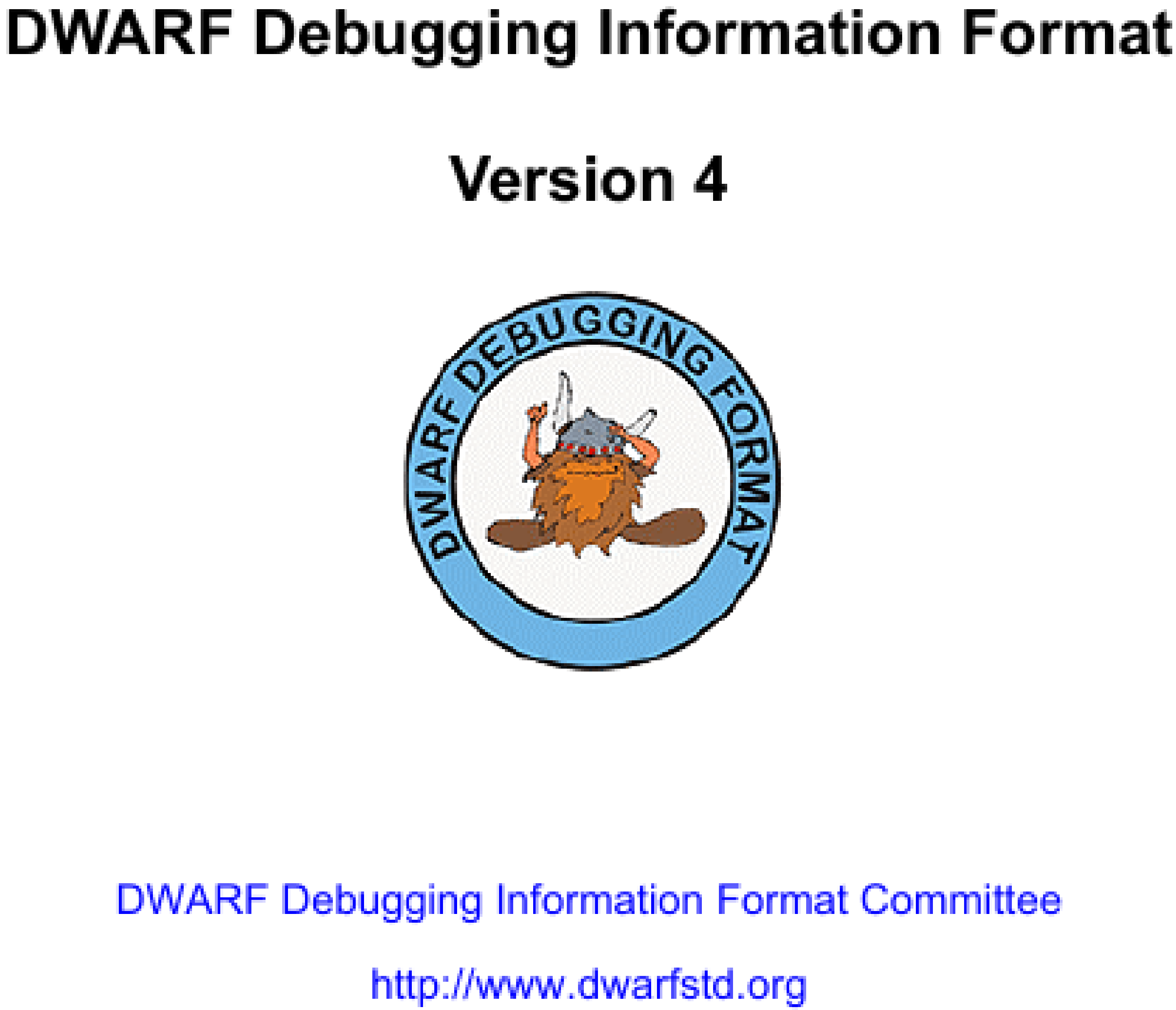
ABI: separate debugger from runtime

- compiler documents its work in metadata
- … “debugging information” (DWARF is my favourite)
- OS has simple control interface (ptrace() + signals)
- some burden for compiler authors
- naturally multi-language

What the ABI says about debugging…

This section defines the Debug With Arbitrary Record Format (DWARF) debugging format for the AMD64 processor family. The AMD64 ABI does not define a debug format. However, all systems that do implement DWARF on AMD64 shall use the following definitions.

DWARF



DWARF Debugging Information Format

Version 4

DWARF Debugging Information Format Committee

http://www.dwarfstd.org

DWARF in a nutshell

Three main kinds of info

- info: how to decode values (objects, stack frames. . .)
- line: how to map binary locations to source locations
- frame: how to reconstruct register values up a callchain

All embedded as sections in ELF file

- .debug_info, .debug_frame, .debug_line
- + some subservient sections. . .

Each defines its own (different) abstract machine!

DWARF info section

```
$ cc -g -o hello hello.c && readelf -wi hello | column
 <b>:TAG_compile_unit                <7ae>:TAG_pointer_type
      AT_language  : 1 (ANSI C)            AT_byte_size: 8
      AT_name      : hello.c               AT_type     : <0x2af>
      AT_low_pc    : 0x4004f4        <76c>:TAG_subprogram
      AT_high_pc   : 0x400514              AT_name     : main
 <c5>: TAG_base_type                       AT_type     : <0xc5>
        AT_byte_size : 4                   AT_low_pc   : 0x4004f4
        AT_encoding  : 5 (signed)          AT_high_pc  : 0x400514
        AT_name      : int           <791>: TAG_formal_parameter
 <2af>:TAG_pointer_type                     AT_name     : argc
        AT_byte_size: 8                     AT_type     : <0xc5>
        AT_type     : <0x2b5>               AT_location : fbreg - 20
 <2b5>:TAG_base_type                 <79f>: TAG_formal_parameter
        AT_byte_size: 1                     AT_name     : argv
        AT_encoding : 6 (char)              AT_type     : <0x7ae>
        AT_name     : char                  AT_location : fbreg - 32
```

DWARF is…

- very expressive
  - out of necessity!
  - has to capture details of *optimised* code
- a huge, bloated spec
  - grown different limbs at different times
  - too many ways of saying the same thing
  - too many abstract machines!
- never implemented *completely* (e.g. gdb)
- not a complete solution…

Big expressiveness wins big prizes

- use as a binary interface definition language
  - (dwarfidl – part of Cake)
- use for sanity-checking compiler output
  - did I generate the code I expected?
- use in various tools, not just debuggers
  - gprof, Valgrind, . . .
- re-use frame info for exception handling (passim.)

Wanted:

- tools making it easier to generate correct DWARF
- tools making it easier to generate complete DWARF
- extensions to DWARF e.g. for interpreted languages

DWARF helps you decode a process's *state*...

... what about *control* of the debugged program?

- process start/stop/interrupt
  - Unix signals: tracer can trap on tracee's signals
- breakpoints
  - trap instrs + single-step or breakpoint shuffle
- watchpoints
  - hardware watchpoint registers and/or software emul
- library loading
  - secret breakpoint + R_DEBUG protocol (on ELF)
- thread control, exception events...

It's all *very* ad-hoc, arch-dependent, nasty...

Further reading

- System V ABI specs & processor supplements
- ELF spec (+ PE, Ma{so}ch-O if you must)
- man pages: gcc, clang, ld, ld.so, dlopen
- Ian Lance Taylor's blog (airs.com/blog)
- readelf and objdump output of your favourite programs

Thanks for listening. Questions?

Using ELF

Mmost ELF features accessed using assembler directives

- .symver, .pushsection/.popsection
- use C's __asm__

But also

- compiler options (e.g. -fvisibility)
- and linker options (e.g. -Bsymbolic)
- and linker scripts (e.g. symbol versioning)!

Reliability problems in the murky bits

Q. Are there reliability / interoperability issues here?

a. YES!

an x86-64 one exhibited when using libffi:
https://sourceware.org/ml/libffi-discuss/2013/msg00013.html

a MIPS one
https://dmz-portal.mips.com/bugz/show_bug.cgi?id=805

an ARM (hardfloat) one
http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=704111

a simple C++ one:
http://lists.cs.uiuc.edu/pipermail/llvmdev/2010-February/02

(and these are just the relatively simple case of def/use across compilers)