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

compile

link

output

ABIs, linkers… – p.4/66
A more realistic picture
A yet more realistic picture
A yet more, more realistic picture still

.\(f\) \rightarrow \text{compile} \rightarrow \text{assemble} \rightarrow URS.O

.\(c\) \rightarrow \text{compile} \rightarrow \text{assemble} \rightarrow URS.O

.\(s\) \rightarrow \text{assemble} \rightarrow URS.O

\text{link} \rightarrow URS.crt* O \rightarrow URS.libc.so

.\(o\) \rightarrow URS.libc*.a

\text{ldscripts}

output \rightarrow \text{ld.so} \rightarrow \text{load (dyn. link)}

\text{operating system}

\text{hardware}

ABIs, linkers... – p.7/66
A yet more, more realistic picture still, still

- .f
  - compile
  - assemble
  - .o
- .c
  - compile
  - assemble
  - .o
- .s
  - assemble
  - .o

- crt*.o
- libc.so
- *.SO
- ldscripts
- libc*.a

link

load (dyn. link)

output

operating system

hardware

ABIs, linkers... – p.8/66
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
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...
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)
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

- .c file
  - Compile
  - .o file
  - Link
  - Output
  - Operating system
  - Hardware
  - libc*.a

ABIs, linkers... – p.15/66
These pair of .c files will compile/link properly with mips-linux-gnu-gcc.

If I compile n1.c with llvm/clang and nla.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 nla.c
void foo(float x, double y) {
  printf("%f %f \n", x, y);
}
```
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));
}
Chapter 8

Execution Environment

Not done yet.

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

- less obvious omissions about
- 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 APIs; 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)

```c
/* 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...
$ cc -c -o hello.o hello.c && readelf -WS hello.o

<table>
<thead>
<tr>
<th>Nr</th>
<th>Name</th>
<th>Type</th>
<th>Addr</th>
<th>Off</th>
<th>Size</th>
<th>Flg</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>.text</td>
<td>PROGBITS</td>
<td>0</td>
<td>040</td>
<td>020</td>
<td>AX</td>
</tr>
<tr>
<td>2</td>
<td>.rela.text</td>
<td>RELA</td>
<td>0</td>
<td>5a0</td>
<td>030</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>.data</td>
<td>PROGBITS</td>
<td>0</td>
<td>060</td>
<td>000</td>
<td>WA</td>
</tr>
<tr>
<td>4</td>
<td>.bss</td>
<td>NOBITS</td>
<td>0</td>
<td>060</td>
<td>000</td>
<td>WA</td>
</tr>
<tr>
<td>5</td>
<td>.rodata</td>
<td>PROGBITS</td>
<td>0</td>
<td>060</td>
<td>00e</td>
<td>A</td>
</tr>
<tr>
<td>6</td>
<td>.comment</td>
<td>PROGBITS</td>
<td>0</td>
<td>06e</td>
<td>02b</td>
<td>MS</td>
</tr>
<tr>
<td>7</td>
<td>.note.GNU-stack</td>
<td>PROGBITS</td>
<td>0</td>
<td>099</td>
<td>000</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>.eh_frame</td>
<td>PROGBITS</td>
<td>0</td>
<td>0a0</td>
<td>038</td>
<td>A</td>
</tr>
<tr>
<td>9</td>
<td>.rela.eh_frame</td>
<td>RELA</td>
<td>0</td>
<td>5d0</td>
<td>018</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>.shstrtab</td>
<td>STRTAB</td>
<td>0</td>
<td>0d8</td>
<td>061</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>.symtab</td>
<td>SYMTAB</td>
<td>0</td>
<td>480</td>
<td>108</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>.strtab</td>
<td>STRTAB</td>
<td>0</td>
<td>588</td>
<td>013</td>
<td></td>
</tr>
</tbody>
</table>

This is a *relocatable* ELF...
$ readelf -Ws hello.o | egrep -v 'SECTION|FILE'
Symbol table ".symtab" contains 11 entries:

<table>
<thead>
<tr>
<th>Num</th>
<th>Value</th>
<th>Size</th>
<th>Type</th>
<th>Bind</th>
<th>Vis</th>
<th>Ndx</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>NOTYPE</td>
<td>LOCAL</td>
<td>DEFAULT</td>
<td>UND</td>
<td>main</td>
</tr>
<tr>
<td>9</td>
<td>24</td>
<td>FUNC</td>
<td>GLOBAL</td>
<td>DEFAULT</td>
<td></td>
<td></td>
<td>puts</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>NOTYPE</td>
<td>GLOBAL</td>
<td>DEFAULT</td>
<td>UND</td>
<td></td>
</tr>
</tbody>
</table>

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
$ objdump -rdS hello.o

...!

int main(int argc, char **argv)
{
    sub $0x8,%rsp
    printf("Hello, world!\n");
    mov $0x0,%edi

    callq e <main+0xe>

    return 0;
}

mov $0x0,%eax
add $0x8,%rsp
retq

ABIs, linkers... – p.23/66
ABIs [loosely] specify many kinds of relocation

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Field</th>
<th>Calculation</th>
</tr>
</thead>
<tbody>
<tr>
<td>R_X86_64_NONE</td>
<td>0</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>R_X86_64_64</td>
<td>1</td>
<td>word64</td>
<td>S + A</td>
</tr>
<tr>
<td>R_X86_64_PC32</td>
<td>2</td>
<td>word32</td>
<td>S + A - P</td>
</tr>
<tr>
<td>R_X86_64_GOT32</td>
<td>3</td>
<td>word32</td>
<td>G + A</td>
</tr>
<tr>
<td>R_X86_64_PLT32</td>
<td>4</td>
<td>word32</td>
<td>L + A - P</td>
</tr>
<tr>
<td>R_X86_64_COPY</td>
<td>5</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>R_X86_64_GLOB_DAT</td>
<td>6</td>
<td>word64</td>
<td>S</td>
</tr>
<tr>
<td>R_X86_64_JUMP_SLOT</td>
<td>7</td>
<td>word64</td>
<td>S</td>
</tr>
<tr>
<td>R_X86_64_RELATIVE</td>
<td>8</td>
<td>word64</td>
<td>B + A</td>
</tr>
<tr>
<td>R_X86_64_GOTPCREL</td>
<td>9</td>
<td>word32</td>
<td>G + GOT + A - P</td>
</tr>
<tr>
<td>R_X86_64_32</td>
<td>10</td>
<td>word32</td>
<td>S + A</td>
</tr>
<tr>
<td>R_X86_64_32S</td>
<td>11</td>
<td>word32</td>
<td>S + A</td>
</tr>
<tr>
<td>R_X86_64_16</td>
<td>12</td>
<td>word16</td>
<td>S + A</td>
</tr>
<tr>
<td>R_X86_64_PC16</td>
<td>13</td>
<td>word16</td>
<td>S + A</td>
</tr>
</tbody>
</table>
Hey—you got your code in my program!

```bash
$ 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

ABIs, linkers… – p.25/66
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
-1gcc
-1gcc_s
-1c
/usr/lib/gcc/x86_64-linux-gnu/4.7/crtend.o
/usr/lib/x86_64-linux-gnu/crtn.o
```

ABIs, linkers… – p.26/66
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 )

ABIs, linkers… – p.27/66
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!

ABIs, linkers… – p.29/66
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 00  callq    10 <main+0x10>

c:   R_X86_64_PLT32 puts-0x4

10:  b8 00 00 00 00 00  mov      $0x0,%eax

15:  48 83 c4 08  add      $0x8,%rsp

19:  c3  retq

ABIs, linkers... – p.31/66
It's not over yet...

```
$ cc -shared -o libhhello.so hello.o && objdump -rdS libhhello.so 
(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...

ABIs, linkers... – p.32/66
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

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/local

SECTIONS

    /* Read-only sections, merged into text segment: */

    PROVIDE (__executable_start = SEGMENT_START("text-segment", 0x400000

    .interp             : { *(.interp) }
    .note.gnu.build-id : { *(.note.gnu.build-id) }
    .hash               : { *(.hash) }
    .gnu.hash           : { *(.gnu.hash) }
    .dynsym             : { *(.dynsym) }
    .dynstr             : { *(.dynstr) }

ABIs, linkers… – p.35/66
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…

ABIs, linkers… – p.36/66
Recap (2)

- Compile:
  - .f → .o
  - .c → .o
  - .s → .o

- Assemble:
  - .o → .o

- Link:
  - .o → .o
  - libc.so

- Output:
  - load (dyn. link)

- Load (dyn. link)

- Operating system

- Hardware

ABIs, linkers… – p.38/66
$ cc -o hello hello.o && readelf -WS hello

<table>
<thead>
<tr>
<th>Nr</th>
<th>Name</th>
<th>Type</th>
<th>Address</th>
<th>Off</th>
<th>Size</th>
<th>ES</th>
<th>Flg</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>.dynsym</td>
<td>DYNSYM</td>
<td>004002b8</td>
<td>0002b8</td>
<td>000060</td>
<td>18</td>
<td>A</td>
</tr>
<tr>
<td>9</td>
<td>.rela.dyn</td>
<td>RELA</td>
<td>00400380</td>
<td>000380</td>
<td>000018</td>
<td>18</td>
<td>A</td>
</tr>
<tr>
<td>13</td>
<td>.text</td>
<td>PROGBITS</td>
<td>00400440</td>
<td>000440</td>
<td>0001a4</td>
<td>00</td>
<td>AX</td>
</tr>
<tr>
<td>15</td>
<td>.rodata</td>
<td>PROGBITS</td>
<td>004005f0</td>
<td>0005f0</td>
<td>000012</td>
<td>00</td>
<td>A</td>
</tr>
<tr>
<td>24</td>
<td>.data</td>
<td>PROGBITS</td>
<td>00601030</td>
<td>001030</td>
<td>000010</td>
<td>00</td>
<td>WA</td>
</tr>
<tr>
<td>25</td>
<td>.bss</td>
<td>NOBITS</td>
<td>00601040</td>
<td>001040</td>
<td>000008</td>
<td>00</td>
<td>WA</td>
</tr>
</tbody>
</table>
Different kinds of linking

Relocatable-to-relocatable linking
- make a bigger .o out of one or more .Os
- 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

ABIs, linkers… – p.41/66
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 /lib64/ld-linux-x86-64.so.2
0000248 x86-64.so.2
$ /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 library (snip)
$ /lib64/ld-linux-x86-64.so.2 ./hello
Hello, world!

ABIs, linkers… – p.42/66
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 (0x00007f4600e0a000)
/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
- dltsymp(handle, symname) looks up a symbol in it
- think: plugin systems

Per-implementation extensions fill some gaps
- e.g. walking the link map
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, ...).
Basic idea: $ LD_PRELOAD=libmylib.so my-command

```c
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, linkers... – p.48/66
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 \( \left( \frac{1}{2} \right) \)

STACK ORIGIN

STACK POINTER

= 9

ABIs, linkers... – p.51/66
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
00000000000000017  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

ABIs, linkers... – p.52/66
$ 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
“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

ABIs, linkers… – p.58/66
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 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!

ABIs, linkers... – p.61/66
$ cc -g -o hello hello.c && readelf -wi hello | column

<b>:TAG_compile_unit
   AT_language : 1 (ANSI C)
   AT_name : hello.c
   AT_low_pc : 0x4004f4
   AT_high_pc : 0x400514

<7ae>:TAG_pointer_type
   AT_byte_size: 8
   AT_type : <0x2af>
   AT_name : main

<c5>: TAG_base_type
   AT_byte_size : 4
   AT_encoding : 5 (signed)
   AT_name : int

<2af>:TAG_pointer_type
   AT_byte_size: 8
   AT_type : <0x2b5>
   AT_name : argc

<2b5>:TAG_base_type
   AT_byte_size: 1
   AT_encoding : 6 (char)
   AT_name : char

<791>: TAG_formal_parameter
   AT_name : argv
   AT_type : <0x7ae>
   AT_location : fbreg - 32

<79f>: TAG_formal_parameter
   AT_name : char
   AT_type : <0x7ae>
   AT_location : fbreg - 32

ABIs, linkers... – p.62/66
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
  ABIs, linkers… – p.64/66
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…

ABIs, linkers… – p.65/66
Further reading

- System V ABI specs & processor supplements
- ELF spec (+ PE, Mach-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

Most 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)!
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:  

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