How to Write Shared Libraries

Ulrich Drepper

drepper@redhat.com

June 23, 2002

Introduction

Actually, it is Dynamic Shared Objects (DSOs)

Primary motivation: save and share resources

- **better physical memory usage**
- **smaller binaries use less disk space**

ELF makes it easy to create DSOs

Entices people to use DSOs for abstraction

Problems

1. Costs of applications and DSOs different

   DSOs have dynamic cost

2. References in ELF very flexible and powerful

   … but slower than in a.out and COFF

3. People write DSOs just like application code

Solutions

Explain how

- **ELF works (at runtime)**
- **the implementation can be changed to automatically do some of the work**
- **programming affects how ELF code is generated**
- **code can be rewritten**

How does ELF work

Statically linked applications are of no interest

The kernel

1. **maps executable (or DSO) in memory**
2. **locates the dynamic segment (ELF Header → Program Header → Dynamic segment `PT_DYNAMIC`)**
3. **determines loader (`PT_INTERP` entry)**
4. **maps the loader as well (overlay)**
5. **constructs auxiliary vector**
6. **starts the loader program**

How does ELF work II

The Loader/Dynamic Linker

1. relocates itself
2. read information from auxiliary vector
3. builds data structures for the application loaded by the kernel
4. finds, loads, and relocates dependencies (recursively)
5. jumps to start address given in auxiliary vector

Loading DSOs

The Loader/Dynamic Linker

1. loads first block of the object (`ElfXX_Ehdr`)
2. locates program header using `e_phoff` and `e_phnum`
3. finds all loadable segments (`PT_LOAD`)
4. locates dynamic section (`PT_DYNAMIC`)
5. initializes hash table
6. sets up PLT/GOT
7. relocates DSO
8. sort DSOs

Symbol Resolution

The following steps have to be performed for each symbol needed in each of the DSOs:

1. **determine the scope (which DSOs to look in and in which order)**
2. **compute ELF hash sum of symbol**
3. **in first/next DSO in scope**
   - **(a) determine hash bucket**
   - **(b) string comparison with the addressed symbols name**
   - **(c) if necessary, string comparison with version name**
   - **(d) stop if matching**
   - **(e) otherwise continue with next element in hash chain**
4. **if not found, continue with next DSO in scope**

User Influence

- **Number of DSOs**
  - **design decision**
  - **sometimes not in the programmer's hand**
- **Text Relocations (must be avoided)**
- **Number of Symbols**
- **Number of PLT entries**
- **Number of relocations**
  - **i.e., number of GOT entries**

We ignore the first point here

Text Relocations

A relocation against a read-only segment

Requires making the segment writable → cannot be shared anymore

Also prevents prelinking

Solution:

- **compile C/C++/Java/Ada/… code with `-fpic`**
- **use `-fPIC` when necessary**
- **follow PIC programming rules in assembler code**

Number of Symbols

Obvious problem: large symbol table data structure

the dynamic symbol table is present at runtime

Secondary: symbol resolution gets slower due to longer hash chains

`readelf -I` *binary*

Unnecessarily large API: even interface not intended for use can be accessiable

Number of PLT/GOT entries

This means: number of relocations

PLTs are necessary for undefined symbols

- **but using, for instance, both `fgetc` and `getchar` is not necessary**

Internal use of defined functions jump though PLT → symbol resolution and indirect jump

Reducing number of GOT entries (= relocations) first priority

Second priority is converting normal relocations into relative ones (faster to process)

Measuring `ld.so` Performance

Total runtime of the application **measure null program with dependencies**

`ld.so` can measure more and more exact

`env LD_DEBUG=statistics` *program arguments*

```
> env LD_DEBUG=statistics /bin/echo
runtime linker statistics:
  total startup time in dynamic loader: 783596 clock cycles
            time needed for relocation: 398588 clock cycles
                 number of relocations: 132
      number of relocations from cache: 5
           time needed to load objects: 207140 clock cycles


runtime linker statistics:
          final number of relocations: 188
final number of relocations from cache: 5
```

Other Measures

Number of relocations:

`readelf -d` **output contains**

- `DT_RELENT`**: size of one relocation entry**
- `DT_RELSZ`**: size of relocation table**
- `DT_RELCOUNT`**: number of relative relocations (if combining relocations)**
- `DT_PLTRELSZ`**: size of PLT relocation table**

`char` versus `const char`

Often found:

```
char *s = "some string";
```

Compiler warns but still it is ignored

Linker puts read-only strings in “mergeable” sections

```
const char *s = "some string";
const char *t = "string";
```

Only `some string` stored in object file

`const char*` versus `const char[]`

Compile as DSO:

```
const char *s = "some string";
```

Creates one relative relocation and 4 bytes data

Very often `s` need not be a variable

```
const char s[] = "some string";
```

Error Codes and Messages

Often found:

```
static const char *msgs[] = {
   [ERR1] = "message for err1",
   [ERR2] = "message for err2",
   [ERR3] = "message for err3"
};

const char *errstr (int nr) {
  return msgs[nr];
}
```

Good practice, bad implementation!

One relocation per array element, `msgs` **in** `.data`**, not** `.rodata`

Error Codes and Messages II

Replace array of strings with one string:

```
static const char msgstr[] =
   "message for err1\0"
   "message for err2\0"
   "message for err3";
};
static const size_t msgidx[] = {
   0,
   sizeof ("message for err1"),
   sizeof ("message for err1")
   + sizeof ("message for err2")
};
const char *errstr (int nr) {
  return msgstr + msgidx[nr];
}
```

Function Pointers

Very reasonable code in executable:

```
static int a0 (int a) { return a+0; }
static int a1 (int a) { return a+1; }
static int a2 (int a) { return a+2; }

static int (*fps[])(int) = {
   [0] = a0,
   [1] = a1,
   [2] = a2
};
int add (int a,int b) {
   return fps[b] (a);
}
```

3 relocations, `fps` **in** `.data`

Function Pointers II

Better use `switch`:

```
int add (int a, int b) {
   switch (b) {
   case 0:
      return a+0;
   case 1:
      return a+1;
   case 2:
      return a+2;
   }
}
```

All PC-relative jumps, no relocations

Local `gotos`

Not many people use it but it's very effective:

```
int add (int a, int b) {
   static const void *labs[] = {
      &&a0, &&a1, &&a2
   };
   void *targ = labs[b];
   goto *targ;
a0:
   return a+0;
a1:
   return a+1;
a2:
   return a+2;
}
```

3 relocations, `labs` **in** `.data`

Local `goto`s II

```
int add (int a, int b) {
   static const unsigned off[] = {
      &&a0-&&a0, &&a1-&&a0, &&a2-&&a0
   };
   void *targ = &&a0 + off[b];
   goto *targ;
a0:
   return a+0;
a1:
   return a+1;
a2:
   return a+2;
}
```

No relocation, compile-time constant `off` array

Exporting Internal Functions

Internal functions often exported (especially if not in the same source file)

```
int mult (int a, int b) {
   return a * b;
}
int multadd (int a, int b, int c) {
   return mult (a, b) + c;
}
```

- `mult` **call uses ELF name lookup**
- **call is indirect through PLT**

Exporting Internal Functions II

Always use `static` if function is not used outside source file

Sometimes adjusting interfaces to eliminating exported functions is beneficial

```
static int mult (int a, int b) {
   return a * b;
}
int multadd (int a, int b, int c) {
   return mult (a, b) + c;
}
```

Exporting Internal Functions III

If the function is used in another file:

Tell the compiler everything

```
extern int mult (int a, int b)
 __attribute__((visibility("hidden")));

int multadd (int a, int b, int c) {
   return mult (a, b) + c;
}
```

Compiler knows the function is not exported from the DSO (the latter is performed by the linker)

Exporting Internal Functions IV

For most architectures same result using linker maps:

```
$ cat multadd.sym
{ global: multadd; local: mult; };
$ gcc -shared -o multadd.so multadd.c \
  -fPIC \
  -Wl,--version-script,multadd.sym
$ readelf -s multadd.so|grep mult
7: 00000590 10 FUNC LOCAL DEFAULT 10 mult
```

Symbol is not exported but compiler already did its work

Works for IA-32, does not work for SH

Exporting Internal Functions V

`libtool` **provides** `-export-symbols` **option**

```
$ cat multadd.exp
multadd
$ libtool --mode=link gcc -o multadd.so \
  -export-symbols multadd.exp multadd.o
```

But: this only modifies the symbol table

Relocations and indirect jumps remain

Exporting Internal Variables

Similar to functions but marking variables as hidden is always necessary:

C code:

```
extern int a;
a
```

IA-32 PIC code:

```
movl a@GOT(%ebx),%eax
movl (%eax),%eax
```

a hidden, IA-32 PIC code:

```
movl a@GOTOFF(%ebx),%eax
```

Using a linker map *cannot* fix the code the compiler already generated

Calling Exported Functions

Often functions, which must be exported, are used internally

If no interposition is wanted, use alias:

```
int mult (int a, int b) {
   return a * b;
}
extern __typeof (mult) mult_internal
 __attribute__((alias("mult"),
                visibility("hidden"))));
int multadd (int a, int b, int c) {
   return mult_internal (a, b) + c;
}
```

Stable ABIs

DSOs with the same SONAME must be binary compatible:

No documented ABI must change

APIs *could* change

Sooner or later an incompatible change is necessary

Option 1: Create DSO with new SONAME name, leave old file undisturbed

Option 2: Use Symbol Versioning

Stable ABIs II

Advantages of using new SONAME:

1. **Portable**
2. **Same SONAME reference on all platforms**
3. **Symbol versioning available only on Linux and Hurd**

Disadvantages:

1. **DSO nightmare**
2. **Large amount of duplication in > 1 DSO (disk space, memory usage)**
3. **How to phase out old DSO files?**

No reason to not special-case Linux and Hurd!

Stable ABIs III

Before:

```
int ext;
int foo (int a)
{
  ext = some_function (a);
  return 0;
}
```

Now:

```
int foo (int a, int *r)
{
  *r = some_function (a);
  return 0;
}
```

Stable ABIs IV

With symbol versioning:

```
int ext;
int foo_old (int a)
{
  ext = some_function (a);
  return 0;
}
asm (".symver foo_old, foo@ABI_1.0");
int foo_new (int a, int *r)
{
  *r = some_function (a);
  return 0;
}
asm (".symver foo_new, foo@@ABI_2.0");
```

Stable ABIs V

```
$ cat foo.sym
ABI_1.0 {
  global: foo; local: *;
};
ABI_2.0 {
  global: foo;
} ABI_1.0;
$ gcc -shared -fpic -o foo.so foo.c \
  -Wl,--version-script,foo.sym
$ nm foo.so|grep foo
000006d8 T foo@@ABI_2.0
000006a0 T foo@ABI_1.0
000006d8 t foo_new
000006a0 t foo_old
```

Stable ABIs VI

Very small impact on runtime performance

If the two versions differ only slightly, simple stub versions can be versioned, which call the real implementation

Symbol versioning can also help to retire an interface (existing uses allowed, new ones are not)

Using `local: *` helps to avoid nasty surprises

Not all compatibility problems can be handled this way, but many/most