ELF1 1F Weak Symbol

Young W. Lim

2022-11-30 Wed

Outline

Based on

"Study of ELF loading and relocs", 1999
http://netwinder.osuosl.org/users/p/patb/public_html/elf_relocs.html

Compling 32-bit program on 64-bit gcc

- gcc -v
- gcc -m32 t.c
- sudo apt-get install gcc-multilib
- sudo apt-get install g++-multilib
- gcc-multilib
- g++-multilib
- gcc -m32
- objdump -m i386

Weak symbols (1)

- a weak symbol denotes
  a specially annotated symbol
  during linking of ELF object files.
- by default, without any annotation,
  a symbol in an object file is strong.

`https://en.wikipedia.org/wiki/Weak_symbol`

Weak symbols (2)

- during linking, a strong symbol can override a weak symbol of the same name
- in the presence of two strong symbols by the same name, the linker resolves the symbol in favor of the first one found.
  - this behavior allows an executable to override standard library functions, such as `malloc`

`https://en.wikipedia.org/wiki/Weak_symbol`

Weak symbols (3)

- When linking a binary executable,
  a weakly declared symbol
  does not need a definition
- (by default) a declared strong symbol
  without a definition triggers
  an undefined symbol link error.

`https://en.wikipedia.org/wiki/Weak_symbol`

Weak symbols (4)

- Weak symbol references that remain unresolved,
  do not result in a fatal error condition,
  no matter what output file type is being generated.
  - static executable
  - dynamic executable or shared object

https://docs.oracle.com/cd/E19120-01/open.solaris/819-0690/chapter2-11/index.html

Weak symbols in a static executable

- If a static executable is being generated,
  the weak symbol is converted
  to an absolute symbol
  with an assigned value of zero

https://docs.oracle.com/cd/E19120-01/open.solaris/819-0690/chapter2-11/index.html

Weak symbols in a dynamic executable

- if a dynamic executable or shared object is being produced, the symbol is left as an undefined weak reference with an assigned value of zero
- during process execution, the runtime linker searches for this symbol.
- if the runtime linker does not find a match, the reference is bound to an address of zero *instead* of generating a fatal relocation error

https://docs.oracle.com/cd/E19120-01/open.solaris/819-0690/chapter2-11/index.html

Undefined weak referenced symbols

- Historically, these undefined weak referenced symbols have been employed as a mechanism to test for the existence of functionality
- Consider a C code fragment which might have been used in the shared object `libfoo.so.1`
- during execution of the application
  - if the function address tests nonzero, the function is called.
  - if the function address tests zero the function is not called.

`https://docs.oracle.com/cd/E19120-01/open.solaris/819-0690/chapter2-11/index.html`

Syntax for annotating a symbol as weak

- Pragma
  - function declaration

```
#pragma weak power2

int power2(int x);
```

- Attribute
  - function declaration

```
int __attribute__((weak)) power2(int x);
```

or

```
int power2(int x) __attribute__((weak));
```

  - variable declaration

```
extern int __attribute__((weak)) global_var;
```

https://en.wikipedia.org/wiki/Weak_symbol

Static example (1)

files	defined functions	
`main.c`	`main(int, char **);`	
`power_slow.c`	`int power2(int);`	- Weak symbol
	`int power3(int);`	
`power.c`	`int power2(int);`	- Strong symbol

- weak symbol `power2` ... *print slow*

```
cc  main.o power_slow.o          -o slow
```

- strong symbol `power2` ... *print fast*

```
cc  main.o power_slow.o power.o -o fast
```

`https://en.wikipedia.org/wiki/Weak_symbol`

Static example (2)

- build commands

```
cc -g -c -o main.o main.c
cc -g -c -o power_slow.o power_slow.c
cc -g -c -o power.o power.c
cc  main.o power_slow.o          -o slow
cc  main.o power_slow.o power.o -o fast
```

- output

```
$ ./slow 3          // power_slow
slow power2
power3() = 27
$ ./fast 3          // power_slow   power
fast power2
power3() = 27
```

https://en.wikipedia.org/wiki/Weak_symbol

Static example (3)

- build commands

```
cc -g -c -o main.o main.c
cc -g -c -o power_slow.o power_slow.c
cc -g -c -o power.o power.c
cc  main.o power_slow.o          -o slow
cc  main.o power_slow.o power.o -o fast
```

- When removing the weak attribute
  and re-executing the build commands,
  `./fast 3` *fails* with the following error message

```
multiple definition of ‘power2’
```

  - `power2` in `power_slow.c` without the `weak` attribute
  - `power2` in `power.c`

  `./slow 3` still *succeeds*,
  and `./slow 3` has the same output.

`https://en.wikipedia.org/wiki/Weak_symbol`

Setting weak attribute

using __attribute__

```
int __attribute__((weak)) power2(int x);

or

int power2(int x) __attribute__((weak));
```

using pragma

```
#pragma weak power2

int power2(int x);
```

https://en.wikipedia.org/wiki/Weak_symbol

Static example source code (1)

1. main.c

```
#include <stdio.h>
#include <stdlib.h>
#include "power_slow.h"

int main(int argc, char **argv) {
  fprintf(stderr, "power3() = %d\n",
          power3(atoi(argv[1])));
  return 0;
}
```

2. power.c

```
#include <stdio.h>



int power2(int x) {
  fprintf(stderr,
          "fast power2()\n");
  return x*x;
}
```

https://en.wikipedia.org/wiki/Weak_symbol

Static example source code (2)

3. power_slow.h

```
#ifndef POWER2_SLOW_H
#define POWER2_SLOW_H

// #pragma weak power2
// int power2(int x);

// int power2(int x)
//     __attribute__((weak))

int __attribute__((weak))
    power2(int x);

int power3(int x);
#endif
```

4. power_slow.c

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


int power2(int x) {
  fprintf(stderr,
          "slow power2()\n");
  return x*x;
}

int power3(int x) {
    return power2(x)*x;
}
```

https://en.wikipedia.org/wiki/Weak_symbol

Shared example build scripts (1)

- build commands

```
cc -g -c -o main.o main.c
cc -g -fpic -c -o power_slow.po power_slow.c
cc -shared -fpic -o libpowerslow.so power_slow.po
cc  main.o -L‘pwd‘ -Wl,-R‘pwd‘ -lpowerslow -o main

cc -g -DENABLE_DEF -fpic -c -o power_slow.po power_slow.c
cc -shared -fpic -o libpowerslow.so power_slow.po
cc  main.o -L‘pwd‘ -Wl,-R‘pwd‘ -lpowerslow -o main2

cc -g -DNO_USER_HOOK -c -o main.o main.c
cc -g -fpic -c -o power_slow.po power_slow.c
cc -shared -fpic -o libpowerslow.so power_slow.po
cc  main.o -L‘pwd‘ -Wl,-R‘pwd‘ -lpowerslow -o main3

cc -g -DNO_USER_HOOK -c -o main.o main.c
cc -g -DENABLE_DEF -fpic -c -o power_slow.po power_slow.c
cc -shared -fpic -o libpowerslow.so power_slow.po
cc  main.o -L‘pwd‘ -Wl,-R‘pwd‘ -lpowerslow -o main4
```

https://en.wikipedia.org/wiki/Weak_symbol

Shared example build scripts (2)

- build commands

```
1) cc -g                  -c -o main.o main.c
2) cc -g                  -c -o main.o main.c
3) cc -g -DNO_USER_HOOK -c -o main.o main.c
4) cc -g -DNO_USER_HOOK -c -o main.o main.c

1) cc -g                -fpic -c -o power_slow.po power_slow.c
2) cc -g -DENABLE_DEF -fpic -c -o power_slow.po power_slow.c
3) cc -g                -fpic -c -o power_slow.po power_slow.c
4) cc -g -DENABLE_DEF -fpic -c -o power_slow.po power_slow.c

1) cc -shared -fpic -o libpowerslow.so power_slow.po
2) cc -shared -fpic -o libpowerslow.so power_slow.po
3) cc -shared -fpic -o libpowerslow.so power_slow.po
4) cc -shared -fpic -o libpowerslow.so power_slow.po

1) cc  main.o -L`pwd` -Wl,-R`pwd` -lpowerslow -o main
2) cc  main.o -L`pwd` -Wl,-R`pwd` -lpowerslow -o main2
3) cc  main.o -L`pwd` -Wl,-R`pwd` -lpowerslow -o main3
4) cc  main.o -L`pwd` -Wl,-R`pwd` -lpowerslow -o main4
```

https://en.wikipedia.org/wiki/Weak_symbol

NO_USER_HOOK and ENABLE_DEF

1. NO_USER_HOOK

```
// in main.c

#ifndef NO_USER_HOOK
void user_hook(void)
{
  fprintf(stderr,
    "main: user_hook()\n");
}
#endif
```

2. ENABLE_DEF

```
// in power_slow.c

#ifdef ENABLE_DEF
void user_hook(void) {
  fprintf(stderr,
    "power_slow: user_hook()\n");
}
#endif
```

https://en.wikipedia.org/wiki/Weak_symbol

Macro defintions

	main.c	power_slow.c
main		
main2		-DENABLE_DEF
main3	-DNO_USER_HOOK	
main4	-DNO_USER_HOOK	-DENABLE_DEF

https://en.wikipedia.org/wiki/Weak_symbol

Function definitions

	main.c	power_slow.c
main	void user_hook(void)	
main2	void user_hook(void)	void user_hook(void)
main3		
main4		void user_hook(void)

https://en.wikipedia.org/wiki/Weak_symbol

Shared example source code skeleton

1. main.c

```
#ifndef NO_USER_HOOK
  void user_hook(void) { ... }
#endif

int main(int argc, char **argv) {
  power3(...);
  ...
}
```

2. power_slow.c

```
int  __attribute__((weak))
     power2(int x);
int  power3(int x);
void __attribute__((weak))
     user_hook(void);

#ifdef ENABLE_DEF
  void user_hook(void) { ... }
#endif

int power2(int x) {
  if (user_hook) user_hook();
  ...              }

int power3(int x) {
  ... power2(x) ...; }
```

https://en.wikipedia.org/wiki/Weak_symbol

Shared example source code (1)

1. main.c

```
#include <stdio.h>
#include <stdlib.h>
#include "power_slow.h"

#ifndef NO_USER_HOOK  //------------
void user_hook(void)
{
  fprintf(stderr,
    "main: user_hook()\n");
}
#endif  //-------------------------

int main(int argc, char **argv) {
  fprintf(stderr, "power3() = %d\n",
          power3(atoi(argv[1])));
  return 0;
}
```

2. power_slow.h

```
#ifndef POWER2_SLOW_H
#define POWER2_SLOW_H

int __attribute__((weak))
    power2(int x);

int power3(int x);
#endif
```

https://en.wikipedia.org/wiki/Weak_symbol

Shared example soruce code (2)

1. power_slow.c (a)

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

void __attribute__((weak))
     user_hook(void);


#ifdef ENABLE_DEF  //---------------
void user_hook(void) {
  fprintf(stderr,
    "power_slow: user_hook()\n");
}
#endif  //--------------------------
```

2. power_slow.c (b)

```
int power2(int x) {
  // only needed ifndef ENABLE_DEF
  if (user_hook)
    user_hook();
  // call user_hook(), only if
  // defined in power_slow.c

  return x*x;
}


int power3(int x) {
  return power2(x)*x;
}
```

https://en.wikipedia.org/wiki/Weak_symbol

Output results with the weak attribute

- build commands with the weak attribute

```
$ ./main 3
main: user_hook()
power3() = 27

$ ./main2 3
main: user_hook()
power3() = 27

$ ./main3 3
power3() = 27

$ ./main4 3
power_slow: user_hook()
power3() = 27
```

	main.c	power_slow.c
main	user_hook()	
main2	user_hook()	user_hook()
main3		
main4		user_hook()

https://en.wikipedia.org/wiki/Weak_symbol

Shared example skeleton (3-1)

1. main.c for main

```
// no macro

user_hook() // definition

int main() {
  power3(...);
  ...
  return 0;
}

-----------------------
$ ./main 3
main: user_hook()
power3() = 27
```

2. power_slow.c for main

```
// no macro

user_hook(); // weak declaration

int power2(int x) {
  if (user_hook) user_hook();
  return x*x;
}

int power3(int x) {
  return power2(x)*x;
}
```

https://en.wikipedia.org/wiki/Weak_symbol

Shared example skeleton (3-2)

1. main.c for main2

```
// no macro

user_hook() definition

int main() {
  power3(...);
    ... ;
  return 0;
}

-----------------------
$ ./main2 3
main: user_hook()
power3() = 27
```

2. power_slow.c for main2

```
// -DENABLE_DEF

user_hook(); // weak declaration
user_hook() {...}  // definition


int power2(int x) {
  if (user_hook) user_hook();
  return x*x;
}

int power3(int x) {
  return power2(x)*x;
}
```

https://en.wikipedia.org/wiki/Weak_symbol

Shared example skeleton (3-3)

1. main.c for main3

```
// -DNO_USER_HOOK

int main() {
  power3(...);
   ...;
  return 0;
}

-----------------------
$ ./main3 3
power3() = 27
```

2. power_slow.c for main3

```
// no macro

user_hook();  // weak declaration

int power2(int x) {
  if (user_hook)  user_hook();
  return x*x;
}

int power3(int x) {
  return power2(x)*x;
}
```

https://en.wikipedia.org/wiki/Weak_symbol

Shared example skeleton (3-4)

1. main.c for main4

```
// -DNO_USER_HOOK

int main() {
  power3(...);
   ...;
  return 0;
}

-----------------------
$ ./main4 3
power_slow: user_hook()
power3() = 27
```

2. power_slow.c for main4

```
// -DENABLE_DEF

user_hook(); // weak declaration
user_hook() {...}  // definition

int power2(int x) {
  if (user_hook)  user_hook();
  return x*x;
}

int power3(int x) {
  return power2(x)*x;
}
```

https://en.wikipedia.org/wiki/Weak_symbol

Shared example (4-1)

1. main.c for main

```
// no macro
...
#include "power_slow.h"

void user_hook(void) {
  ...
}

int main(int argc, char **argv) {
  ... power3(...) ...;
  return 0;
}

-----------------------
$ ./main 3
main: user_hook()
power3() = 27
```

2. power_slow.c for main

```
// no macro
#include <stdio.h>

void __attribute__((weak))
     user_hook(void);

int power2(int x) {
  // only needed ifndef ENABLE_DEF
  if (user_hook) user_hook();
  return x*x;
}

int power3(int x) {
  return power2(x)*x;
}
```

https://en.wikipedia.org/wiki/Weak_symbol

Shared example (4-2)

1. main.c for main2

```
// no macro
...
#include "power_slow.h"

void user_hook(void) {
  ...
}

int main(int argc, char **argv) {
  ... power3(...) ... ;
  return 0;
}
```

```
$ ./main2 3
main: user_hook()
power3() = 27
```

2. power_slow.c for main2

```
// -DENABLE_DEF
#include <stdio.h>
void __attribute__((weak))
     user_hook(void);

void user_hook(void) {
  ...
}

int power2(int x) {
  // only needed ifndef ENABLE_DEF
  if (user_hook) user_hook();
  return x*x;
}

int power3(int x) {
  return power2(x)*x;
}
```

https://en.wikipedia.org/wiki/Weak_symbol

Shared example (4-3)

1. main.c for main3

```
// -DNO_USER_HOOK
...
#include "power_slow.h"

int main(int argc, char **argv) {
  ... power3(...) ...;
  return 0;
}

-----------------------
$ ./main3 3
power3() = 27
```

2. power_slow.c for main3

```
// no macro
#include <stdio.h>

void __attribute__((weak))
     user_hook(void);

int power2(int x) {
  // only needed ifndef ENABLE_DEF
  if (user_hook)  user_hook();
  return x*x;
}

int power3(int x) {
  return power2(x)*x;
}
```

https://en.wikipedia.org/wiki/Weak_symbol

Shared example (4-4)

1. main.c for main4

```
// -DNO_USER_HOOK
...
#include "power_slow.h"

int main(int argc, char **argv) {
  ... power3(...) ...;
  return 0;
}

-----------------------
$ ./main4 3
power_slow: user_hook()
power3() = 27
```

2. power_slow.c for main4

```
// -DENABLE_DEF
#include <stdio.h>
void __attribute__((weak))
     user_hook(void);
void user_hook(void) {
  ...
}

int power2(int x) {
  // only needed ifndef ENABLE_DEF
  if (user_hook)  user_hook();
  return x*x;
}

int power3(int x) {
  return power2(x)*x;
}
```

https://en.wikipedia.org/wiki/Weak_symbol

Shared example (5)

- removing the weak attribute and re-executing the build commands
  - for `main` and `main2`
    - no build errors
    - the same output : `main user_hook()`
  - for `main3`
    - warning and error messages

      `warning: the address of ‘user_hook’ will always evaluate as ‘true’`

      `libpowerslow.so: undefined reference to ‘user_hook’`
  - for `main4`
    - the same warning

      `warning: the address of ‘user_hook’ will always evaluate as ‘true’`
    - no link error

`https://en.wikipedia.org/wiki/Weak_symbol`

Shared example (6)

- The warning is issued by the compiler
  `warning: the address of ‘user_hook’ will always evaluate as ‘true’`
  because it can *statically determine* that in `if (user_hook)`
  the expression `user_hook` evaluates always to true
  because it contains an ELF jump table entry
- the error message is issued by the linker
  `libpowerslow.so: undefined reference to ‘user_hook’`

`https://en.wikipedia.org/wiki/Weak_symbol`

Shared example (7)

- depending on the compiler and used optimization level,
  the compiler may interpret the conditional
  `if (func)` as always true
- because `func` can be seen as undefined
  from a standards point of view
- instead, a system API can be used
  to check if `func` is defined
  e.g. `dlsym` with `RTLD_DEFAULT`
- `if (func)` check may also fail for other reasons,
  e.g. when `func` contains an ELF jump table entry

`https://en.wikipedia.org/wiki/Weak_symbol`

Shared example (8)

```
#pragma weak func
void func();

void bar() {
  if (func)
      func();
}
```

https://en.wikipedia.org/wiki/Weak_symbol

Output results - without weak

- build commands without the weak attribute

```
$ ./main 3
main: user_hook()
power3() = 27

$ ./main2 3
main: user_hook()
power3() = 27

$ ./main3 3
warning message
error message

$ ./main4 3
warning message
power_slow: user_hook()
power3() = 27
```

	main.c	power_slow.c
main	user_hook()	
main2	user_hook()	user_hook()
main3		
main4		user_hook()

https://en.wikipedia.org/wiki/Weak_symbol

Shared example skeleton - witout weak (5-1)

1. main.c for main

```
// no macro

user_hook() // definition

int main() {
  power3(...);
  ...
  return 0;
}

-----------------------
$ ./main 3
main: user_hook()
power3() = 27
```

2. power_slow.c for main

```
// no macro

// user_hook(); // No weak declaration

int power2(int x) {
  if (user_hook) user_hook();
  return x*x;
}

int power3(int x) {
  return power2(x)*x;
}
```

https://en.wikipedia.org/wiki/Weak_symbol

Shared example skeleton - without weak (5-2)

1. main.c for main2

```
// no macro

user_hook() definition

int main() {
  power3(...);
   ... ;
  return 0;
}

-----------------------
$ ./main2 3
main: user_hook()
power3() = 27
```

2. power_slow.c for main2

```
// -DENABLE_DEF

// user_hook(); // No weak declaration
user_hook() {...}  // definition


int power2(int x) {
  if (user_hook) user_hook();
  return x*x;
}

int power3(int x) {
  return power2(x)*x;
}
```

https://en.wikipedia.org/wiki/Weak_symbol

Shared example skeleton - without weak (5-3)

1. main.c for main3

```
// -DNO_USER_HOOK

int main() {
  power3(...);
   ...;
  return 0;
}
```

```
-----------------------
$ ./main3 3
warning message
error message
```

2. power_slow.c for main3

```
// no macro

// user_hook(); // No weak declaration


int power2(int x) {
  if (user_hook)  user_hook();
  return x*x;
}

int power3(int x) {
  return power2(x)*x;
}
```

https://en.wikipedia.org/wiki/Weak_symbol

Shared example skeleton - without weak (5-4)

1. main.c for main4

```
// -DNO_USER_HOOK

int main() {
  power3(...);
   ...;
  return 0;
}

-----------------------
$ ./main4 3
warning message
power_slow: user_hook()
power3() = 27
```

2. power_slow.c for main4

```
// -DENABLE_DEF

// user_hook(); // No weak declaration
user_hook() {...}  // definition

int power2(int x) {
  if (user_hook)  user_hook();
  return x*x;
}

int power3(int x) {
  return power2(x)*x;
}
```

https://en.wikipedia.org/wiki/Weak_symbol

Weak symbol example code (1)

weak symbols

```
#pragma weak    foo

extern  void    foo(char *);

void bar(char *path)
{
        void (*fptr)(char *);

        if ((fptr = foo) != 0)
                (*fptr)(path);
}
```

- this C code fragment might have been used in the shared object `libfoo.so.1`
- checking the existence of the funciton `foo`
- if exist, `(*fprt)(path)`

https://docs.oracle.com/cd/E19120-01/open.solaris/819-0690/chapter2-11/index.html

Weak symbol example code (2)

- When building an application that references `libfoo.so.1`,
  the link-edit completes successfully
  regardless of whether a definition for the symbol `foo` is found
  - If during execution of the application
    the function address tests nonzero,
    the function is called.
  - However, if the symbol definition is not found,
    the function address tests zero
    and therefore is not called.

`https://docs.oracle.com/cd/E19120-01/open.solaris/819-0690/chapter2-11/index.html`

functionality test (1)

- Compilation systems view this address comparison technique as having undefined semantics, which can result in the test statement being *removed* under optimization.
- In addition, the runtime symbol binding mechanism places other *restrictions* on the use of this technique.
- These *restrictions prevent* a consistent model from being made available for *all* dynamic objects

`https://docs.oracle.com/cd/E19120-01/open.solaris/819-0690/chapter2-11/index.html`

functionality test (2)

- Undefined weak references in this manner are discouraged.
- Instead, you should use `dlsym`
  with the `RTLD_DEFAULT`, or `RTLD_PROBE` handles
  as a means of testing for a symbol's existence.

`https://docs.oracle.com/cd/E19120-01/open.solaris/819-0690/chapter2-11/index.html`

Weak Pragma (1)

- For compatibility with SVR4, GCC supports
  a set of #pragma directives
  for declaring symbols to be weak,
  and defining weak aliases
  1. attribute
     ```
     extern foo(char *) __attribute__((weak)) ;
     ```
  2. pragma
     ```
     #pragma weak foo
     extern foo(char *);
     ```

https://gcc.gnu.org/onlinedocs/gcc-4.4.4/gcc/Weak-Pragmas.html

Weak Pragma (2)

- `#pragma weak symbol`
  - This pragma declares symbol to be weak,
    as if the declaration had the attribute of the same name.
  - The pragma may appear
    before or after the declaration of symbol,
    but must appear
    before either its first use or its definition

  -It is not an error
  for symbol to never be defined at all.

`https://gcc.gnu.org/onlinedocs/gcc-4.4.4/gcc/Weak-Pragmas.html`

Weak Pragma (3)

- `#pragma weak symbol1 = symbol2`
  - This pragma declares `symbol1` to be a weak alias of `symbol2`
  - It is an error if `symbol2` is not defined in the current translation unit.

`https://gcc.gnu.org/onlinedocs/gcc-4.4.4/gcc/Weak-Pragmas.html`

Weak Pragma (4)

- weak symbol declaration
  1. attribute
     ```
     extern foo(char *) __attribute__((weak)) ;
     ```
  2. pragma
     ```
     #pragma weak foo
     extern foo(char *);
     ```
- weak symbol alias
  1. attribute
     ```
     extern foo(char *) __attribute__((weak, alias
     ("__foo"))) ;
     ```
  2. pragma
     ```
     #pragma weak foo = __foo
     ```

https://gcc.gnu.org/onlinedocs/gcc-4.4.4/gcc/Weak-Pragmas.html

Jump table (1)

- A jump table can be
  - an array of pointers to functions
  - an array of machine code jump instructions
- if you have a relatively static set of functions
  (such as system calls or virtual functions for a class)
  then you can create this table once and
  call the functions using a simple index into the array.
- This would mean
  - retrieving the pointer
  - calling a function
  - jumping to the machine code depending on the type of table used.

`https://stackoverflow.com/questions/48017/what-is-a-jump-table`

Jump table (2)

- The benefits of doing this in embedded programming are:
  - indexes are more memory *efficient*
    than machine code or pointers,
    so memory can be saved in constrained environments.
  - for any particular function
    the index will remain stable and
    changing the function merely requires
    changing the function pointer through index
- small overhead for accessing the table,
  but no worse than any other virtual function call

`https://stackoverflow.com/questions/48017/what-is-a-jump-table`

Virtual function (1)

- C has no native syntax for virtual methods
  However, you can still implement virtual methods
  by mimicking the way C++ implements virtual methods.
  C++ stores an additional pointer
  to the function definition in each class
  for each virtual method
  thus, you can simply add a function pointer
  to a struct to simulate virtual methods

`https://stackoverflow.com/questions/6844471/virtual-functions-in-c`

Virtual function (2)

Mimicking a virtual function

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

int f2(int x) {
    printf("%d\n",x);
}

typedef struct mystruct {
    int (*f)(int);
} mystruct;
```

```
int main() {
    mystruct *s;

    s = malloc(sizeof(mystruct));
    s->f=f2;
    s->f(42);
    free(s);
    return 0;
}
```

https://stackoverflow.com/questions/6844471/virtual-functions-in-c

Jump table (3)

- A jump table (branch table), is
  a series of instructions,
  all unconditionally branching
  to another point in code.
- You can think of them as a switch (or select) statement
  where all the cases are filled

`https://stackoverflow.com/questions/48017/what-is-a-jump-table`

Jump table (4)

- Note that there's no return -
  the code that it jumps to will execute the return,
  and it will jump back to wherever `myjump` was called.

```
myjump(int c)
{
   switch(state)
   {
      case 0:
         goto func0label;
      case 1:
         goto func1label;
      case 2:
         goto func2label;
   }
}
```

https://stackoverflow.com/questions/48017/what-is-a-jump-table

Jump table (5)

- Jump tables are used
  - to reduce the stack overhead and
  - to save code space
  - to improve speed
- speed is extremely important in interrupt handlers
- the peripheral that caused the interrupt
  is only identified by a single variable.
- this is similar to the vector table

`https://stackoverflow.com/questions/48017/what-is-a-jump-table`

Jump table (6)

- a branch table is a term used
  to describe an efficient method
  of transferring program control (_branching_)
  to another part of a program or a different program
  that may have been dynamically loaded
  using a table of branch instructions
- the branch table construction is commonly used
  when programming in assembly language
  but may also be generated by a compiler

`https://stackoverflow.com/questions/48017/what-is-a-jump-table`

Jump table (7)

- A branch table consists of a serial list of unconditional branch instructions
- the offset is obtained by multiplying a sequential index by the instruction length of each branch instruction
- machine code instructions for branching have a fixed length
- useful when raw data values can be easily converted to sequential index values

https://stackoverflow.com/questions/48017/what-is-a-jump-table

Jump table (8)

- optionally validating the input data
  to ensure it is acceptable;
- transforming the data into an offset
  into the branch table,
  - multiplying the data (index) by the instruction length
- branching often involves an addition of
  the offset onto the program counter register.
  - branching address is made up of
    - the base of the table
    - the generated offset

`https://stackoverflow.com/questions/48017/what-is-a-jump-table`

ELF Jump table (1)

- As much as possible, ELF dynamic linking *defers* the resolution of jump/call addresses until the last minute.
  - The technique is inspired by the i386 design, and is based on the following two constraints.

`http://netwinder.osuosl.org/users/s/scottb/public_html/notes/Elf-Design-4.html`

ELF Jump table (2)

- two constraints:

  1. The calling technique should not *force*
     a change in the assembly code produced for apps;
     it MAY cause changes in the way assembly code
     is produced for pic code (i.e. libraries)

  1. The technique must be such that
     all executable areas must not be *modified*;
     and any modified areas must not be *executed*.

`http://netwinder.osuosl.org/users/s/scottb/public_html/notes/Elf-Design-4.html`

ELF Jump table (3)

- To *defer* the resolution of jump/call addresses, there are three steps involved in a typical jump:
  1. in the code
  2. through the PLT
  3. using a pointer from the GOT

`http://netwinder.osuosl.org/users/s/scottb/public_html/notes/Elf-Design-4.html`

ELF Jump table (4)

- When the executable or library is *first loaded*,
  the GOT entry *points* to code
  which implements dynamic name resolution
  and code finding
- On the first invocation, the function is located and
  the GOT entry is *replaced*
  by the address of the real functon
- Subsequent calls go through 1)-2)-3) and
  end up calling the real code.

`http://netwinder.osuosl.org/users/s/scottb/public_html/notes/Elf-Design-4.html`

ELF Jump table (5)

1. In the code:

   `b/bl  function_call`

   - this is typical ARM code
     using the 26 bit relative jump or call
   - the target is an entry in the PLT.
   - this call is identical to a normal call.

`http://netwinder.osuosl.org/users/s/scottb/public_html/notes/Elf-Design-4.html`

ELF Jump table (6)

2. In the PLT:
   - The PLT is a synthetic area, created by the linker.
   - It exists in both executables and libraries.
   - It is an array of stubs, one per imported function call.
   - It looks like this:

```
PLT[n+1]:
   ldr     ip, 1f          @load an offset
   add     ip, pc, ip      @add the offset to the pc
   ldr     pc, [ip]        @jump to that address
1:      .word   GOT[n+3] - .
```

http://netwinder.osuosl.org/users/s/scottb/public_html/notes/Elf-Design-4.html

ELF Jump table (7)

- the second line

```
    ldr     ip, 1f          @load an offset
                            @1 is the local label
                            @f (forward), b (backward)

1:      .word   GOT[n+3] - .
```

makes `ip = &GOT[n+3]`,

`GOT[n+3]` contains

- either a pointer to PLT[0] (the fixup trampoline)
- or a pointer to the actual code.

`http://netwinder.osuosl.org/users/s/scottb/public_html/notes/Elf-Design-4.html`

ELF Jump table (8)

- The first PLT entry `PLT[0]` is slightly different,
  and is used to form a trampoline to the fixup code.

```
PLT[0]:
        str     lr, [sp, #-4]!  @push the lr
        ldr     lr, [pc, #16]   @load from 6 words ahead
        add     lr, pc, lr      @form an address
        ldr     pc, [lr, #8]!   @jump to the contents of that addr
```

  - the `lr` is pushed on the stack and used for calculations.
  - the load on the second line `[pc, #16]`
    loads `lr` with `&GOT[3] - . - 20`
  - on the third line `[lr, #8]`, the addition leaves

```
lr = (&GOT[3] - . - 20) + (. + 8)
lr = (&GOT[3] - 12)
```

http://netwinder.osuosl.org/users/s/scottb/public_html/notes/Elf-Design-4.html

ELF Jump table (9)

- The first PLT entry PLT[0]

```
PLT[0]:
        str     lr, [sp, #-4]!  @push the lr
        ldr     lr, [pc, #16]   @load from 6 words ahead
        add     lr, pc, lr      @form an address
        ldr     pc, [lr, #8]!   @jump to the contents of that addr
```

  - on the third line [lr, #8], the addition leaves

```
lr = (&GOT[3] - . - 20) + (. + 8)
lr = (&GOT[3] - 12)
```

  - on the fourth line, the pc and lr are both updated,

```
pc = GOT[2]
lr = &GOT[2]
```

http://netwinder.osuosl.org/users/s/scottb/public_html/notes/Elf-Design-4.html

ELF Jump table (1)

```
#include <stdio.h>

int main() {
    printf("Hello World!\n");
    return 0;
    }
```

- The above code is just a simple program that prints the text "Hello World!" to the screen. Now let us compile and run the program.
- gcc was used to compile the file into an executable and -m32 option was used to compile it as a 32 bit executable, finally the output file was given the name main.

https://www.linkedin.com/pulse/elf-linux-executable-plt-got-tables-mohammad-alhya

ELF Jump table (2)

- Using the file utility we can see that this is a 32 bit ELF executable for linux and that it was dynamically linked. we will get to know the meaning of this later in this article.
- A library is a code that is exposed to be used by other programs running on the system. it allows for modular development of code, separation of components and software reuse which is one of the benefits of using libraries. You do not need to worry about the printf function implementation each time you write a program, simply you can just use a library. for now let us look what libraries are being used by our executable.

https://www.linkedin.com/pulse/elf-linux-executable-plt-got-tables-mohammad-alhya

ELF Jump table (3)

- The ldd command was used to find the libraries our executable is using, one of these libraries is libc which contains the implementation of the printf and many other useful functions.
- Keep in mind that we write programs in high level languages such as C and then we compile programs into a machine code format that the CPU understands. let us look at the main executable machine code.

```
# Command used to produce the dump
# objdump -d main -M intel  | grep main -A20

- -d : decode -M : use intel syntax for assembly language

  https://www.linkedin.com/pulse/elf-linux-executable-plt-got-tables-mohammad-alhya
```

ELF Jump table (4)

- The second column is the machine code of the main function, to the right is the assembly instructions equivalent to these opcodes. You can see from this output that the compiler replaced printf function with puts function which is another function that does the same job, print something out to the screen. the call instruction goes to the address 0x80482e0 ! what is there ? How did the compiler figured out this address ?
- Before wrapping up this section, let us talk a bit about linking. One of the main tasks for A linker is to make the code of library functions (eg printf(), scanf(), sqrt()) available for your program. A linker can accomplish this task in two ways :

https://www.linkedin.com/pulse/elf-linux-executable-plt-got-tables-mohammad-alhya

ELF Jump table (5)

1. By copying the code of the library function to your object code [machine code]. That is static linking.
2. By making some arrangements so that the complete code of library functions is not copied , but made available at run time. That is dynamic Linking.
3. Static linking is the result of the linker making a copy of all used library functions to the executable file. Dynamic linking does not require the code to be copied, it is done by just placing the name of the library in the binary file
4. Linking happens when the program is run, when both the binary file and the library are in memory.

https://www.linkedin.com/pulse/elf-linux-executable-plt-got-tables-mohammad-alhya

ELF Jump table (6) Static Linking

- It is when you link your application to another library at compile time, the library code is part of your application.
- The primary advantage of static linking is the speed:
  - There will be no symbol (a program entity) resolution at runtime.
  - Every piece of the library is part of the binary image (executable).
- Once everything is bundled into your application, you do not have to worry that the system will have the right library and version available.
- Static linking creates large binary files that utilize disk space and main memory.
- Once the library is linked and the program is created, you have no way of changing any of the library code without rebuilding the whole program.

https://www.linkedin.com/pulse/elf-linux-executable-plt-got-tables-mohammad-alhya

ELF Jump table (7) Dynamic Linking

- Dynamic linking defers much of the linking process until a program starts running Performs the linking process "on the fly" as programs are executed in the system.
- Libraries are loaded into memory by programs when they start.
- During compilation of the library, the machine code is stored on your machine.
- When you recompile a program that uses this library, only the new code in the program is compiled.
- Does not recompile the library into the executable file like in static linking.
- The main reason for using dynamic linking of libraries is to free your software from the need to recompile with each new release of library.

https://www.linkedin.com/pulse/elf-linux-executable-plt-got-tables-mohammad-alhya

ELF Jump table (8) Dynamic Linking

- Dynamic linking is the more modern approach, and has the advantage of much smaller executable size.
- Dynamic linking helps overall performance in two ways :
- It saves on disk and virtual memory
- Libraries are only mapped in to the process when needed
- All executable dynamically linked to a particular library share a single copy of the library at run time. Ensure that libraries mapped into memory are shared by all processes using them.

https://www.linkedin.com/pulse/elf-linux-executable-plt-got-tables-mohammad-alhya