INSIGHT OF GCC

LINKER

ABSTRACT

This document gives overview of linker. How linker works internally

Contents

1. Objective

This document gives details of linker, how it works by taking GNU gcc compiler. In this document we have explained linking process by taking examples written in C language and gcc 4.8.5 version.

This paper gives details of following:

- program executable and execution process
- what is Linker
- how linking is done by linker
- static and dynamic linking
- relocations
- PIC(Position Independent Code)
- GOT(Global Offset Table)
- PLT(Process Linkage Table)

2. Introduction

Linker plays a crucial role in generating an executable, which can be executed on a particular hardware. Before explaining linking it is required to know what is a program executable and what an execution process is.

In computing, an executable file or executable program, causes a computer to perform indicated tasks according to encoded instructions. Performing these indicated tasks is an execution process.

Prior to execution, a program must first be written in high level language. This is generally done in source code, which is then compiled at compile time and statically linked at link time to an executable. This executable is then invoked, most often by an operating system, which loads the program into memory (load time), performs linking, and then begins execution by moving control to the entry point of the program. At this point execution begins and the program enters run time. The program then runs until it ends, either normal termination or a crash.

Need of program executable:

Computers are a balanced mix of software and hardware. Hardware is just a piece of mechanical device and its functions are being controlled by a compatible software. Hardware understands instructions in the form of electronic charge, which is the counterpart of binary language in software programming. Binary language has only two alphabets, 0 and 1. To instruct, the hardware codes must be written in binary format, which is simply a series of 1s and 0s. So programs written in high level language must be converted into machine level language which is binary language (0 and 1).

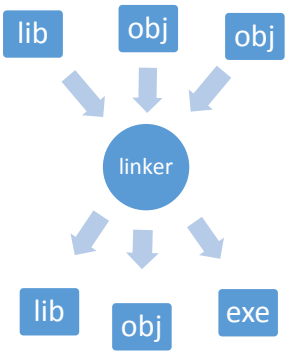
Program execution tools:

Different tools of program executions are given in below diagram

1. **Preprocessor** is the first pass of any C compilation. It processes include-files, conditional compilation instructions and macros.
2. **Compiler** is the second pass. It takes the output of the preprocessor, and the source code, and generates assembler source code.
3. **Assembler** is the third stage of compilation. It takes the assembly source code and produces an assembly listing with offsets. The assembler output is stored in an object file.
4. **Linker** is the final stage of compilation. It takes one or more object files or libraries as input and combines them to produce a single (usually executable) file. In doing so, it resolves references to external symbols, assigns final addresses to procedures/functions and variables, and revises code and data to reflect new addresses (a process called relocation).
5. **Loader:** At the end there should be a single executable file which can be loaded into the memory by loader.

This document explains the **Linker** part in detail.

3. Linker



In computing, a **Linker** is a computer program that takes one or more object files generated by a compiler or libraries and combines them into a single executable file, library file, or another object file.

Why Linker is required?

During compilation, if the compiler could not find the definition for a function, it would just assume that the function is defined in another file. The linker will look at multiple files and try to find references for the functions that were not mentioned. Hence, linker provide following advantages:

- Large programs can be divided into separate modules (small programs) which can be compiled separately and then can be linked by the linker to form an executable.

- Can build libraries of common functions e.g., Math library, standard C library etc. which can be used by different applications(set of programs).

Tasks performed by Linker:

- Searches the program to find library routines used by program, e.g. printf (), math routines etc.
- Determines the memory locations that code from each module will occupy and relocates its instructions by adjusting absolute references (relocation).
- Resolves references among files (variables or functions may be defined in other files that is resolved by linker).

4. Types of Linking

Static Linking

During static linking the linker copies only the routines that are actually needed into the executable image.

Advantage: But static linking is faster and more portable because it does not require the presence of the library on the system where it runs. So executable can be copied to any other machine, and will run correctly.

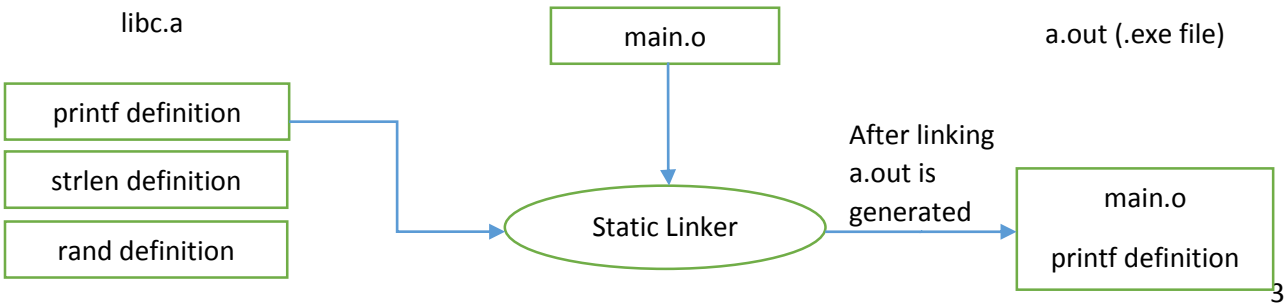
Disadvantage: This of course takes more space on the disk and in memory than dynamic linking.

This is explained by using an example below.

File: main.c

```
#include <stdio.h>
int main()
{
printf("welcome");
return 0;
}
```

After compiling this program from gcc compiler an object file main.o is created. The file main.o calls the function printf, which is present in some library libc.a. The linker figures out this function reside in libc.a and puts the whole thing together in exe file. This is now known as **static linking**.



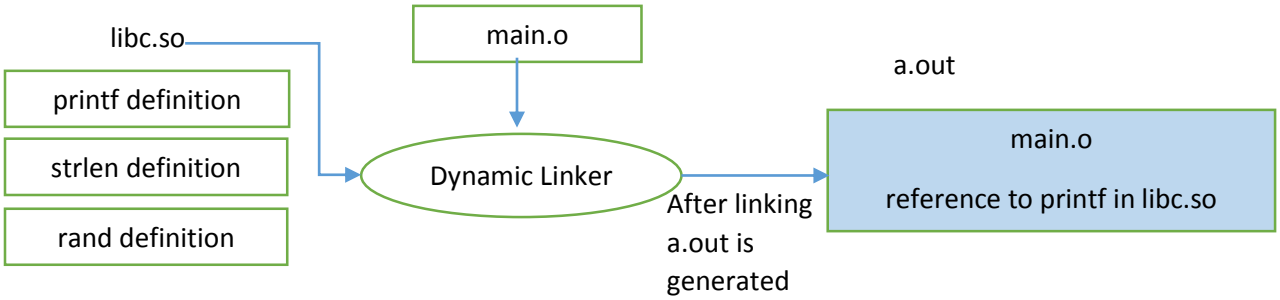
Dynamic Linking

The dynamic linking process starts when a program is actually executing. During linking process the linker does not copy the definitions of a function/routine into the executable. Rather, it makes a note that the program depends upon a certain library. In this process the name of the shared library is placed in the final executable file while the actual linking takes place at run time when both executable file and library are placed in different locations of memory.

Advantage:

- The main advantage to using dynamically linked libraries is that the size of executable programs is dramatically reduced because each program does not have to store redundant copies of the library functions that it uses.
- Also, when DLL functions are updated, programs that use them will automatically obtain their benefits.

Disadvantage: It is slower as compared to static linking.



Note that dynamic linker does not copy libc.so in the final executable image (which is a.out in above example) it just note that libc.so library is needed. When execution of program starts it uses this library at run time.

Dynstr section of a.out

```
Contents of section .dynstr:
 400318 006c6962 632e736f 2e360070 72696e74  .libc.so.6.printf
```

Dynstr section contains string which is used in dynamic linking. Above line tells linker that printf function is present in the library libc.so

Static linking	**Dynamic linking**
gcc -static print.c // by default it will perform dynamic linking to force linker to perform static linking use -static option it will statically link with static library libc.a	gcc print.c // by default it will perform dynamic linking it will dynamically linked with shared library libc.so
objdump –ds a.out **Disassemble of main function** 0000000000400dc0 <main>: // some code 400dce: e8 cd 09 00 00 callq 4017a0 <IO_printf> // some code	objdump –ds a.out **Disassemble of main function** 0000000000400530 <main>: //some code 40053e: e8 cd fe ff callq 400410 <printf@plt> //some code

Disassemble of printf function 00000000004017a0 <_IO_printf>: // code of printf function	Disassemble of PLT 0000000000400410 <printf@plt>: // plt will contain only reference of printf function and printf function will be linked dynamically
Size of a.out 852465 bytes // large size due to definition of printf and other functions	**Size of a.out** 8505 bytes// small size because it only contains reference of printf not definition

5. Internal working of Linker

Relocation and Symbol Resolution

When an assembler generates an object file, it does not know where the code and data will ultimately be stored in memory. Assembler does not know locations of any externally defined functions or global variables referenced in the object file either. In this case a "relocation entry" is generated when the assembler encounters a reference to an object (function/variable) whose definition is not known to assembler. In this case assembler assumes that its definition may present in other object files or library files. So assembler just enters dummy address (0x0) and generate relocation table entry in relocation table. So it is the job of linker to resolve all references and modify the object program so that it can be loaded at an address different from the location originally specified

Each relocatable object file contains a relocation table which gives information to linker where and how relocations need to perform.

- **Relocation**. Compilers and assemblers generate the object code for each input module with a starting address of zero. Relocation is the process of assigning load addresses to different parts of the program by merging all sections of the same type into one section. The code and data section also are adjusted so they point to the correct runtime addresses.
- **Symbol Resolution**. A program is made up of multiple subprograms; reference of one subprogram to another is made through symbols. A linker's job is to resolve the reference by noting the symbol's location and patching the caller's object code.

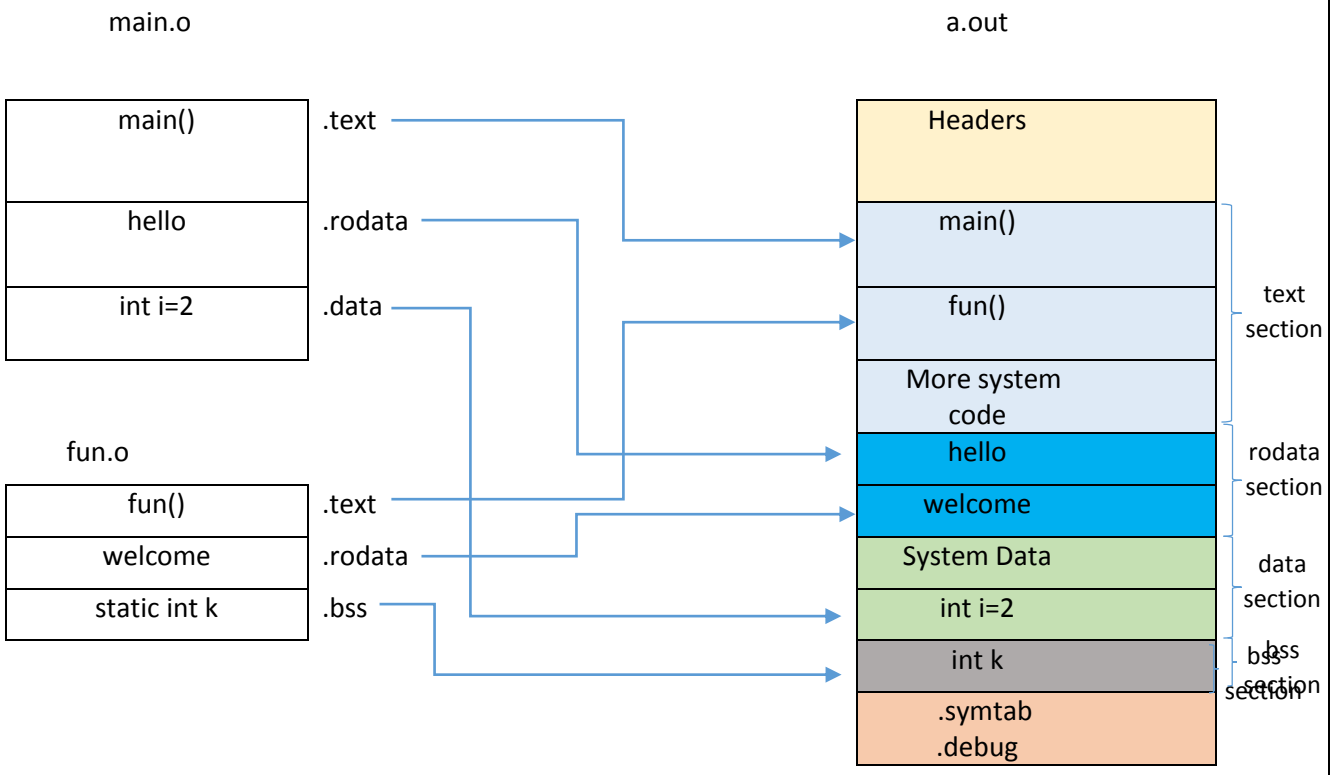
Example: Suppose we have two files main.c and fun.c. These are compiled individually to generate main.o and fun.o.

main.c

```
#include<stdio.h>
int i=2;
int main()
{
printf("hello");
fun();
return 0;
}
```

fun.c

```
#include<stdio.h>
void fun()
{
static int k;
int j;
printf("welcome");
}
```



As shown in above figure linker merge two object files to generate an executable. It performs following tasks

- it merges text sections of both main.o and fun.o files
- defined variables of both object files placed in the data section (variable i in above example)
- undefined variables (global/static) of both files are placed in the bss section (variable k in above example)
- It will not store temporary variables in any section. Temporary variable are pushed onto the stack when corresponding function containing that variable is pushed on to the stack (variable j in above example)

Relocation Table:

Assembler will create relocation table as follows:

$gcc –c main.c // creates main.o file

$readelf –r main.o // give following relocation information

```
#include<stdioh>
int i=2;
int main() {
printf("hello");
fun();
return 0; }
```

Relocation section '.rela.text' at offset 0x270 contains 3 entries:

Offset	Info	Type	Sym. Value	Sym. Name + Addend
00000000000a	000a00000002	R_X86_64_PC32	0000000000000000	fun - 4
00000000000f	00050000000a	R_X86_64_32	0000000000000000	.rodata + 0
000000000019	000b00000002	R_X86_64_PC32	0000000000000000	printf - 4

1 Offset: This member gives the location at which to apply the relocation action

2 info: This member gives both the symbol table index of corresponding entry

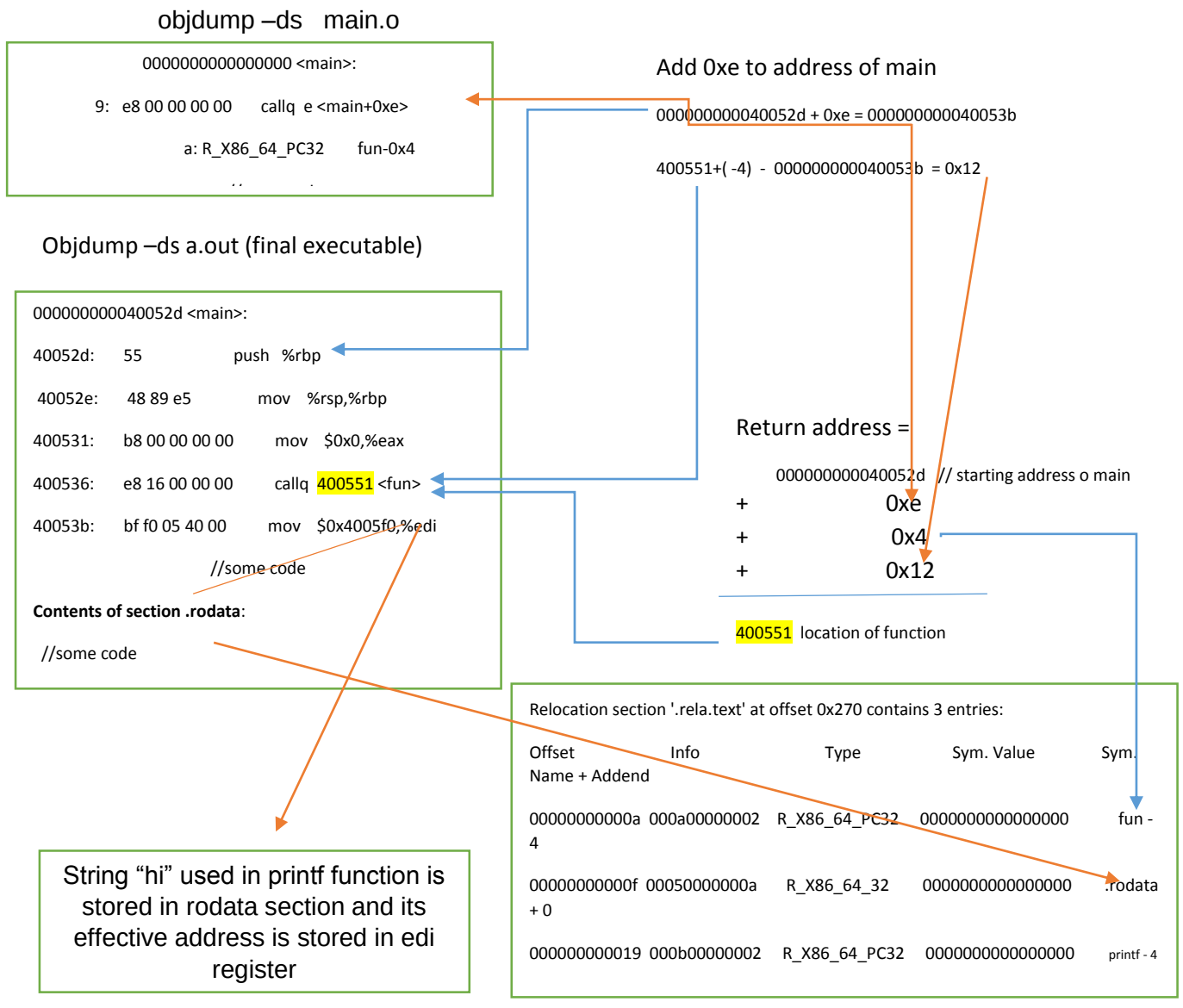
3 Type: This member gives type of relocation that needs to apply.

4 Sym. Value: This member gives value of symbol in symbol table

5 Sym. Name + Addend: This member specifies a constant addend used to compute the value to be stored into the relocatable field.

Two relocation types are used in this program:

1 R_X86_64_32: Used for absolute addressing. Uses the value encoded in the instruction. This relocation is for hello string in printf function which is stored in the rodata section
2 R_X86_64_PC32: Relocate a reference that uses a 32 bit PC (program counter) relative address.
Effective address = PC + instruction encoded address.
In above example this relocation type is used to calculate actual address of function fun() and printf() in final executable.



6. PIC

Need:

Linker when linking the executable, can fully resolve all internal symbol references (to functions and data) to fixed and final locations. The linker does some relocations of its own but eventually the output it produces contains no additional relocations. As long as the executable needs no shared libraries it needs no relocations. But if it does use shared libraries (as do the vast majority of Linux applications), symbols taken from these shared libraries need to be relocated, because of how shared libraries are loaded.

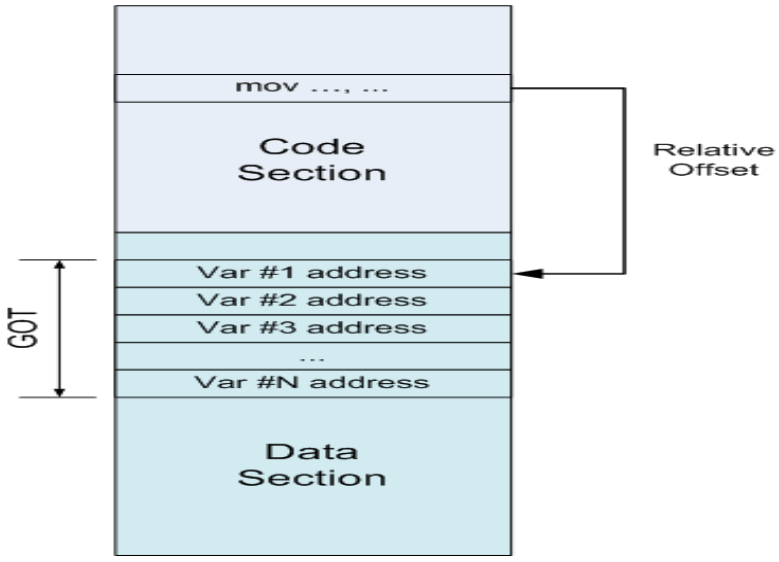
Unlike executables when shared libraries are being built, the linker can't assume a known load address for their code. The reason for this is simple. Each program can use any number of shared libraries, and there's simply no way to know in advance where any given shared library will be loaded in the process's virtual memory. To solve this problem PIC is used.

Introduction:

Position-independent code can be executed at any memory address without modification. The idea behind PIC is simple - add an additional level of indirection to all global data (with the help of got) and function (with the help of plt) references in the code. It's possible to make the text section of the shared library truly position independent, in the sense that it can be easily mapped into different memory addresses without needing to change one bit.

7. The Global Offset Table (GOT)

The implementation of position-independent code is accomplished by means of a "global offset table", or in short GOT.A GOT is simply a table of addresses (mainly addresses of global variables), residing in the data section. Suppose some instruction in the code section wants to refer to a variable. Instead of referring to it directly by absolute address (which would require a relocation), it refers to an entry in the GOT. Since the GOT is in a known place in the data section, this reference is relative and known to the linker. The GOT entry, in turn, will contain the absolute address of the variable.



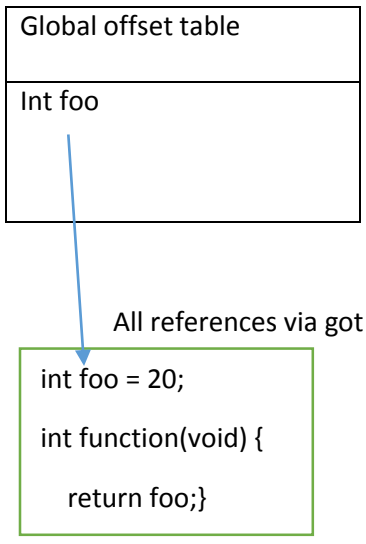
So, we've gotten rid of a relocation in the code section by redirecting variable references through the GOT. But we've also created a relocation in the data section. In this way t's possible to make the text section of the shared library truly position independent and it is much easier to handle relocations in data section than text section. In this way we can achieve position independent code (PIC) mechanism

```
int foo = 20;

int function(void) {

    return foo;

}
```

In this example we have foo as a global variable. So if we compile this got.c file with –fpic option and check relocation table then it will give following relocation table

```
Objdump –r got.o

Relocation section '.rela.text' at offset 0x260 contains 1 entries:

 Offset          Info           Type           Sym. Value    Sym. Name + Addend

000000000000a  000800000009 R_X86_64_GOTPCREL 0000000000000000 foo - 4
```

Note P_X86_64_GOTPCREL relocation. This says "replace the data of global variable foo with the *global offset table* (GOT) entry of foo



All references via got

```
int foo = 20;

int function(void) {

    return foo;}
```

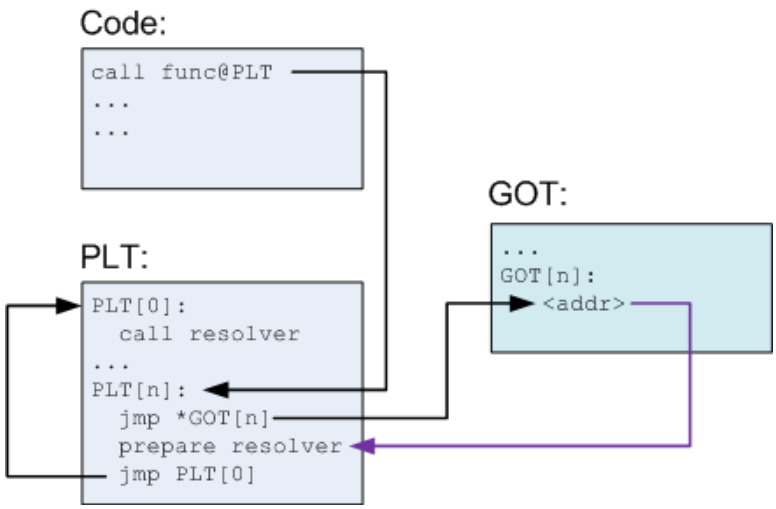
Disassembly of got.so file

```
// contents of function

00000000000006b5 <function>:

6b9:   48 8b 05 20 09 20 00    mov    0x200920(%rip),%rax        # 200fe0 <_DYNAMIC+0x1d0>
```

```
Contents of section .got:

 200fd0 00000000 00000000 00000000 00000000

 200fe0 00000000 00000000 00000000 00000000

 200ff0 00000000 00000000 00000000 00000000
```

8. The Procedure Linkage Table (PLT)

The PLT is part of the executable text section, consisting of a set of entries (one for each external function the shared library calls). Each PLT entry is a short chunk of executable code. Instead of calling the function directly, the code calls an entry in the PLT, which then takes care to call the actual function. Each PLT entry also has a corresponding entry in the GOT which contains the actual offset to the function, but only when the dynamic loader resolves it.

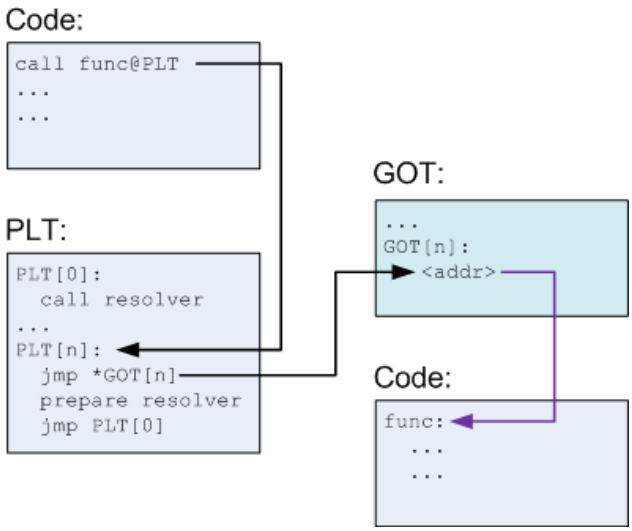


Explanation:

- In the code, a function `func` is called. The compiler translates it to a call to `func@plt`, which is some N-th entry in the PLT.
- The PLT consists of a special first entry, followed by a bunch of identically structured entries, one for each function needing resolution.
- Each PLT entry consists of these parts:
  - A jump to a location which is specified in a corresponding GOT entry
  - Preparation of arguments for a "resolver" routine
  - Call to the resolver routine, which resides in the first entry of the PLT

- The first PLT entry is a call to a resolver routine, which is located in the dynamic loader itself. This routine resolves the actual address of the function. This is explained below in detail.
- Before the function's actual address has been resolved, the Nth GOT entry just points to after the jump. This is why this arrow in the diagram is colored differently - it's not an actual jump, just a pointer.

  After the first call, the diagram looks a bit differently:



  What happens when func is called for the first time is this:

- PLT[n] is called and jumps to the address pointed to in GOT[n].
- This address points into PLT[n] itself, to the preparation of arguments for the resolver.
- The resolver is then called.
- The resolver performs resolution of the actual address of func, places its actual address into GOT[n] and calls func.

Note that GOT[n] now points to the actual func instead of back into the PLT. So, when func is called again:

- PLT[n] is called and jumps to the address pointed to in GOT[n].
- GOT[n] points to func, so this just transfers control to func.

In other words, now func is being actually called, without going through the resolver, at the cost of one additional jump. That's all there is to it, really. This mechanism allows lazy resolution of functions, and no resolution at all for functions that aren't actually called.

It also leaves the code/text section of the library completely position independent, since the only place where an absolute address is used is the GOT, which resides in the data section and will be relocated by the dynamic loader. Even the PLT itself is PIC, so it can live in the read-only text section.

The resolver is simply a chunk of low-level code in the loader that does symbol resolution. The arguments prepared for it in each PLT entry, along with a suitable relocation entry, help it to know about the symbol that needs resolution and about the GOT entry to update.

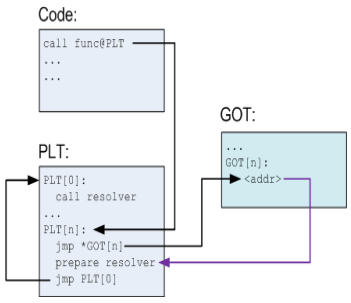
```
int var = 42;

int ml_util_func(int a)

{

   return a + 1;

}

int ml_func(int a, int b)

{

   int c = b + ml_util_func(a);

   var += c;

   return b + var;

}
```

gcc -c -shared -fpic sample.c // create object file

gcc -shared -fPIC -o sa.so sample.o // create shared library

// note that -fpic option is used to create position

// independent code



$objdump –ds sa.so

Text section of ml_func function

```
0000000000000714 <ml_func>:

 714:   55                 push   %rbp

 715:   48 89 e5           mov    %rsp,%rbp

 718:   48 83 ec 20        sub    $0x20,%rsp

 71c:   89 7d ec           mov    %edi,-0x14(%rbp)

 71f:   89 75 e8           mov    %esi,-0x18(%rbp)

 722:   8b 45 ec           mov    -0x14(%rbp),%eax

 725:   89 c7              mov    %eax,%edi

 727:   e8 d4 fe ff ff     callq  600 <ml_util_func@plt>
```

corresponding plt entries

```
0000000000000600 <ml_util_func@plt>:

 600:   ff 25 1a 0a 20 00      jmpq   *0x200a1a(%rip)  # 201020<_GLOBAL_OFFSET_TABLE_+0x20>

 606:   68 01 00 00 00         pushq  $0x1

 60b:   e9 d0 ff ff ff         jmpq   5e0 <_init+0x20>
```

Corresponding got entries

```
Contents of section .got.plt:

 201000 100e2000 00000000 00000000 00000000  .. ............

 201010 00000000 00000000 f6050000 00000000  ................

 201020 06060000 00000000 16060000 00000000  ................
```

The interesting part is the call to ml_util_func@plt. Note also that the address of GOT Here's what ml_ (it's in an executable section called .plt

Recall that each PLT entry consists of three parts:

- A jump to an address specified in GOT (this is the jump to [201020 i.e. 201000+0x20 where 201000 is the start address for GOT as shown above])
- Preparation of arguments for the resolver
- Call to the resolver

Note that corresponding plt entry of function is stored in got at address 201020

To help the dynamic loader do its job, a relocation entry is also added and specifies which place in the GOT to relocate for ml_util_func

```
Relocation section '.rela.plt' at offset 0x578 contains 3 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000201018  000300000007 R_X86_64_JUMP_SLO 0000000000000000 __gmon_start__ + 0
000000201020  000a00000007 R_X86_64_JUMP_SLO 0000000000000705 ml_util_func + 0
000000201028  000600000007 R_X86_64_JUMP_SLO 0000000000000000 __cxa_finalize + 0
~
```

The second last line means that the dynamic loader should place the value (address) of symbol ml_util_func into 0x201020 (which, recall, is the GOT entry for this function).

Difference between Position independent code and Position dependent code

```
void add(int a,int b)
{
int c;
c=a+b;
}
void main()
{
int i=3,j=4;
add(i,j);
}
```

a.c

Position independent code

position dependent code

a.so

a.out

```
00000000000006cc <main>:
 6cc:   55                 push   %rbp
 6cd:   48 89 e5           mov    %rsp,%rbp
 6d0:   48 83 ec 10        sub    $0x10,%rsp
//some code
 6ec:   e8 af fe ff ff     callq  5a0 <add@plt>
//some code
```

```
0000000000400504 <main>:
  400504:     55                 push   %rbp
  400505:     48 89 e5           mov    %rsp,%rbp
  400508:     48 83 ec 10        sub    $0x10,%rsp
  //  some code
  400524:     e8 c4 ff ff ff     callq  4004ed <add>
//some code
```

sec nt) and a.so file (position

From the above figs we can conclude following points

1 In a.so file address of function is stored in plt section so this code can be executed at any position (by mapping address in plt rather than giving absolute address in code section)

2 In a.out file absolute address of function add is given. So a.out file cannot be executed from any other location because of absolute address of function add.

Hence, we conclude to the end of linking process. The more we go in deep the more we learn.

8 References

1 http://www.mindfruit.co.uk/2012/06/interpreting-readelf-r-in-this-case.html
2 http://docs.oracle.com/cd/E23824_01/html/819-0690/chapter6-54839.html

9 Appendix

Some important relocation types:

(R_*_GOT32	reference to a global symbol: create an entry in the GOT and let the run-time system deposit the symbol's address into the GOT for us)
(R_*_PLT32)	Same as got In addition, relative calls to subroutine can be used.
R_*_32	absolute, to a symbol
(R_*_PC32)	relative, from "here" to a symbol used for branches and function calls
(R_*_COPY	reach into a library to a symbol and copy down the data into our own .bss space
R_*_GLOB_DAT	pointer to global data
(R_*_JMP_SLOT)	pointer to library function
(R_*_RELATIVE)	There will be times when local data structures need to hold absolute pointers to local data. Put the module-relative address of the symbol in the library; at run-time, add the module-load address to it

Where * is architecture specific name

E: g if architecture is x86-64 then relocation type will be R_X86-64_GOT32 and so on