DWARF (debugging data format)

DAVIDE CHIAPPETTA · ಸೋಮವಾರ, ಸೆಪ್ಟೆಂಬರ್ 19, 2016 ·

How source debuggers work?

Application binaries are a result of compile and build operations performed on a single or a set of source files. Program Source files contain functions, data variables of various types (local, global, register, static), and abstract data objects, all written and neatly indented with nested control structures as per high level programming language syntax (C/C++). Compilers translate code in each source file into machine instructions (1's and 0's) as per target processors Instruction set Architecture and bury that code into object files. Further, Linkers integrate compiled object files with other pre-compiled objects files (libraries, runtime binaries) to create end application binary image called executable.

Source debuggers are tools used to trace execution of an application executable binary. Most amazing feature of a source debugger is its ability to list source code of the program being debugged; it can show the line or expression in the source code that resulted in a particular machine code instruction of a running program loaded in memory. This helps the programmer to analyze a program's behavior in the high-level terms like source-level flow control constructs, procedure calls, named variables, etc, instead of machine instructions and memory locations. Source-level debugging also makes it possible to step through execution a line at a time and set source-level breakpoints.

Lets explore how source debuggers like gnu gdb work ? So how does a debugger know where to stop when you ask it to break at the entry to some function? How does it manage to find what to show you when you ask it for the value of a variable? The answer is – debug information. All modern compilers are designed to generate Debug information together with the machine code of the source file. *It is a representation of the relationship between the executable program and the original source code*. This information is encoded as per a pre-defined format and stored alongside the machine code. Many such formats were invented over the years for different platforms and executable files (aim of this article isn't to survey the history of these formats, but rather to show how they work). Gnu compiler and ELF executable on Linux/ UNIX platforms use DWARF, which is widely used today as default debugging information format.

Word of Advice : Does an Application/ Kernel programmer need to know Dwarf?

Obvious answer to this question is a big **NO.** It is purely subject matter for developers involved in implementation of a Debugger tool. A normal Application developer using debugger tools would never need to learn or dig into binary files for debug information. This in no way adds any edge to your debugging skills nor adds any new skills into your armory. However, if you are a developer using debuggers for years and curious about how debuggers work read this document for an outline into debug information. If you are a beginner to

systems programming or fresher's learning programming I would suggest not to waste your time as you can safely ignore this.

ELF -DWARF sections

Gnu compiler generates debug information which is organized into various sections of the ELF object file. Let's use the following source file for compiling and observing DWARF sections

```
sample.c
*****************************************************************

#include <stdio.h>
#include <stdlib.h>
int add(int x, int y)
{
        return x + y;
}
int main()
{
        int a = 10, b = 20;
        int result;
        int (*fp) (int, int);
        fp = add;
        result = (*fp) (a, b);
        printf(" %dn", result);
}
```

```
*****************************************************************
gcc -c -g sample.c -o sample.o
objdump -h sample.o | more
*****************************************************************


sample.o      file format pe-i386    (on Windows)
sample.o:     file format elf32-i386 (on Linux)


Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         0000005f  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000000  00000000  00000000  00000094  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  00000094  2**2
                  ALLOC
  3 .debug_abbrev 000000a2  00000000  00000000  00000094  2**0
                  CONTENTS, READONLY, DEBUGGING
  4 .debug_info   00000114  00000000  00000000  00000136  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING
  5 .debug_line   00000040  00000000  00000000  0000024a  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING
  6 .rodata       00000005  00000000  00000000  0000028a  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  7 .debug_loc    00000070  00000000  00000000  0000028f  2**0
                  CONTENTS, READONLY, DEBUGGING
  8 .debug_pubnames 00000023  00000000  00000000  000002ff  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING
  9 .debug_pubtypes 00000012  00000000  00000000  00000322  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING
10 .debug_aranges 00000020  00000000  00000000  00000334  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING
11 .debug_str    000000b0  00000000  00000000  00000354  2**0
```

```
                  CONTENTS, READONLY, DEBUGGING
12 .comment      0000002b  00000000  00000000  00000404  2**0
                  CONTENTS, READONLY
13 .note.GNU-stack 00000000  00000000  00000000  0000042f  2**0
                  CONTENTS, READONLY
14 .debug_frame  00000054  00000000  00000000  00000430  2**2
                  CONTENTS, RELOC, READONLY, DEBUGGING
```

All of the sections with naming *debug_xxx* are debugging information sections. Information in these sections is interpreted by source debugger like gdb. Each *debug_* section holds specific information like

```
.debug_info               core DWARF data containing DIEs
.debug_line               Line Number Program
.debug_frame              Call Frame Information
.debug_macinfo            lookup table for global objects and functions
.debug_pubnames           lookup table for global objects and functions
.debug_pubtypes           lookup table for global types
.debug_loc                Macro descriptions
.debug_abbrev             Abbreviations used in the .debug_info section
.debug_aranges            mapping between memory address and compilation
.debug_ranges             Address ranges referenced by DIEs
.debug_str                String table used by .debug_info
```

Debugging Information Entry (DIE)

Dwarf format organizes debug data in all of the above sections using special objects (program descriptive entities) called Debugging Information Entry (DIE). Each DIE has a tag filed whose value specifies its type, and a set of attributes. DIEs are interlinked via sibling and child links, and values of attributes can point at other DIEs. Now let's dig into ELF file to view how a DIE looks like. We will begin our exploration with *.debug_info* section of the ELF file since core DIE's are listed in it.

```
objdump --dwarf=info . /sample
```

Above operation shows long list of DIE's. Let's limit ourselves to relevant information

```
./sample:     file format elf32-i386
Contents of the .debug_info section:
  Compilation Unit @ offset 0x0:
   Length:        0x110 (32-bit)
   Version:       2
   Abbrev Offset: 0
   Pointer Size:  4
<0><b>: Abbrev Number: 1 (DW_TAG_compile_unit)
    < c>   DW_AT_producer    : (indirect string, offset: 0xe): GNU C 4.5.2
    <10>   DW_AT_language    : 1      (ANSI C)
    <11>   DW_AT_name        : (indirect string, offset: 0x44): sample.c
    <15>   DW_AT_comp_dir    : (indirect string, offset: 0x71): /root
    <19>   DW_AT_low_pc      : 0x80483c4
    <1d>   DW_AT_high_pc     : 0x8048423
    <21>   DW_AT_stmt_list   : 0x0
```

Each source file in the application is referred in dwarf terminology as a “compilation unit”. Dwarf data for each compilation unit (source file) starts with a compilation unit DIE. Above dump shows the first DIE’s and tag value “*DW_TAG_compile_unit “.* This DIE provides general information about compilation unit like source file name (DW_AT_name : (indirect string, offset: 0x44): sample.c), high level programming language used to write source file(DW_AT_language : 1 (ANSI C)) , directory of the source file(DW_AT_comp_dir : (indirect string, offset: 0x71): /root) , compiler and producer of dwarf data(DW_AT_producer : (indirect string, offset: 0xe): GNU C 4.5.2) , start virtual address of the compilation unit (DW_AT_low_pc : 0x80483c4), end virtual address of the unit (DW_AT_high_pc : 0x8048423). Compilation Unit DIE is the parent for all the other DIE’s that describe elements of source file. Generally, the list of DIE’s that follow will describe data types, followed by global data, then the functions that make up the source file. The DIEs for variables and functions are in the same order in which they appear in the source file.

How does debugger locate Function Information ?

While using source debuggers we often instruct debugger to insert or place break point at some function, expecting the debugger to pause program execution at functions. To be able to perform this task, debugger must to map between a function name in the high-level code and the address in the machine code where the instructions for this function begin. For this mapping information debuggers rely on DIE’s that describes specified function. DIE’s describing functions in a compilation unit are assigned tag value ***“DW_TAG_subprogram”*** subprogram as per dwarf terminology is a function. In our sample application source we have two functions (main, add), dwarf should generate a “*DW_TAG_subprogram”* DIE’s for each function, these DIE attributes would define function mapping information that debugger needs for resolving machine code addresses with function name. Each “DW_TAG_subprogam” DIE contains

1. Function scope
2. function name
3. source file or compilation unit in which function is located
4. line no in the source file where the function starts
5. Functions return type
6. Start address of the fucntion
7. End address of the function
8. Frame information of the function.

```
objdump --dwarf=info ./sample | grep "DW_TAG_subprogram"
************************************************************


<1><72>: Abbrev Number: 4 (DW_TAG_subprogram)
<1><a8>: Abbrev Number: 6 (DW_TAG_subprogram)
<1><72>: Abbrev Number: 4 (DW_TAG_subprogram)
    <73>   DW_AT_external    : 1
    <74>   DW_AT_name        : add
    <78>   DW_AT_decl_file   : 1
    <79>   DW_AT_decl_line   : 4
    <7a>   DW_AT_prototyped  : 1
```

```
    <7b>   DW_AT_type        : <0x4f>
    <7f>   DW_AT_low_pc      : 0x80483c4
    <83>   DW_AT_high_pc     : 0x80483d1
    <87>   DW_AT_frame_base  : 0x0    (location list)
    <8b>   DW_AT_sibling     : <0xa8>
<1><a8>: Abbrev Number: 6 (DW_TAG_subprogram)
    <a9>   DW_AT_external    : 1
    <aa>   DW_AT_name        : (indirect string, offset: 0x1a): main
    <ae>   DW_AT_decl_file   : 1
    <af>   DW_AT_decl_line   : 8
    <b0>   DW_AT_type        : <0x4f>
    <b4>   DW_AT_low_pc      : 0x80483d2
    <b8>   DW_AT_high_pc     : 0x8048423
    <bc>   DW_AT_frame_base  : 0x38   (location list)
    <c0>   DW_AT_sibling     : <0xf8>
```

We now have accessed DIE description of function's *main* and *add***.** Let's analyze attribute information of add fucntions DIE.

Function scope: DW_AT_external : 1 (scope external)

Function name: DW_AT_name : add

Source file or compilation unit in which function is located: DW_AT_decl_file : 1 (indicates 1st compilation unit which is sample.c)

Line no in the source file where the function starts: DW_AT_decl_line : 4 (indicates line no 4 in source file)

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


int add(int x, int y)
{
        return x + y;
}
int main()
{
        int a = 10, b = 20;
        int result;
        int (*fp) (int, int);
        fp = add;
        result = (*fp) (a, b);
        printf(" %dn", result);
}
```

Function's source line no matched with DIE description of line no. let's continue with rest of the attribute values

Functions return type: DW_AT_type : <0x4f>

As we have already understood that values of attributes can point to other DIE , here is an example of it.

Value DW_AT_type : <0x4f> indicates that return type description is stored in other DIE at offset 0x4f.

```
<4f>: Abbrev Number: 3 (DW_TAG_base_type)
    <50>   DW_AT_byte_size   : 4
    <51>   DW_AT_encoding    : 5      (signed)
    <52>   DW_AT_name        : int
```

This DIE describes data type and composition of return type of the function add, as per DIE attribute values return type is *signed int* of size 4 bytes.

Start address of the function : DW_AT_low_pc : 0x80483c4

End address of the function: DW_AT_high_pc : 0x80483d1

Above values indicate start and end virtual address of the machine instructions of *add* function, we can verify that with binary dump of the function

```
080483c4 <add>:
80483c4:      55                    push   %ebp
80483c5:      89 e5                 mov    %esp,%ebp
80483c7:      8b 45 0c              mov    0xc(%ebp),%eax
80483ca:      8b 55 08              mov    0x8(%ebp),%edx
80483cd:      8d 04 02              lea    (%edx,%eax,1),%eax
80483d0:      5d                    pop    %ebp
80483d1:      c3                    ret
```

How does debugger find program data (variables...) Information?

When the program hits assigned break point in a function, debugger pauses the program execution, at this time we can instruct debugger to show or print values of variables, by using debugger commands like *print* or *display* followed by variable name (ex: print a) How does debugger know where to find memory location of the variable ? Variables can be located in global storage, on the stack, and even in registers.The debugging information has to be able to reflect all these variations, and indeed DWARF does. As an example let's take a look at complete DIE information set for *main* function.

```
<1><a8>: Abbrev Number: 6 (DW_TAG_subprogram)
    <a9>   DW_AT_external    : 1
    <aa>   DW_AT_name        : (indirect string, offset: 0x1a): main
    <ae>   DW_AT_decl_file   : 1
    <af>   DW_AT_decl_line   : 8
    <b0>   DW_AT_type        : <0x4f>
    <b4>   DW_AT_low_pc      : 0x80483d2
    <b8>   DW_AT_high_pc     : 0x8048423
    <bc>   DW_AT_frame_base  : 0x38   (location list)
    <c0>   DW_AT_sibling     : <0xf8>
<2><c4>: Abbrev Number: 7 (DW_TAG_variable)
    <c5>   DW_AT_name        : a
    <c7>   DW_AT_decl_file   : 1
    <c8>   DW_AT_decl_line   : 10
    <c9>   DW_AT_type        : <0x4f>
    <cd>   DW_AT_location    : 2 byte block: 74 1c   (DW_OP_breg4 (esp): 28)
<2><d0>: Abbrev Number: 7 (DW_TAG_variable)
    <d1>   DW_AT_name        : b
    <d3>   DW_AT_decl_file   : 1
    <d4>   DW_AT_decl_line   : 10
    <d5>   DW_AT_type        : <0x4f>
    <d9>   DW_AT_location    : 2 byte block: 74 18   (DW_OP_breg4 (esp): 24)
```

```
<2><dc>: Abbrev Number: 8 (DW_TAG_variable)
    <dd>          : (indirect string, offset: 0x6a): result
    <e1>   DW_AT_decl_file   : 1
    <e2>   DW_AT_decl_line   : 11
    <e3>   DW_AT_type        : <0x4f>
    <e7>   DW_AT_location    : 2 byte block: 74 10   (DW_OP_breg4 (esp): 16)
<2><ea>: Abbrev Number: 7 (DW_TAG_variable)
    <eb>   DW_AT_name        : fp
    <ee>   DW_AT_decl_file   : 1
    <ef>   DW_AT_decl_line   : 12
    <f0>   DW_AT_type        : <0x10d>
    <f4>   DW_AT_location    : 2 byte block: 74 14   (DW_OP_breg4 (esp): 20)
```

Note the first number inside the angle brackets in each entry. This is the nesting level – in this example entries with <2> are children of the entry with <1>. *main* function has three integer variables *a,b and result* each of these variables are described with DW_TAG_variable nested DIE's (0xc4, 0xd0, 0xdc). *main* function also has a function pointer *fp* described in DIE 0xea . Variable DIE attributes specify variable name (DW_AT_name), declaration line no in source function (DW_AT_decl_line), pointer to address of DIE describing variables data type (DW_AT_type) and relative location of the variable within function's frame (DW_AT_location).

To locate the variable in the memory image of the executing process, the debugger will look at the DW_AT_location attribute of DIE. For *a* its value is DW_OP_fbreg4 (esp):28. This means that the variable is stored at offset 28 from the top in the frame of containing function. The DW_AT_frame_base attribute of *main* has the value 0x38(location list), which means that this value actually has to be looked up in the location list section. Let's look at it:

```
objdump --dwarf=loc sample
sample:     file format elf32-i386
Contents of the .debug_loc section:
    Offset   Begin    End      Expression
    00000000 080483c4 080483c5 (DW_OP_breg4 (esp): 4)
    00000000 080483c5 080483c7 (DW_OP_breg4 (esp): 8)
    00000000 080483c7 080483d1 (DW_OP_breg5 (ebp): 8)
    00000000 080483d1 080483d2 (DW_OP_breg4 (esp): 4)
   00000000 <End of list>
    00000038 080483d2 080483d3 (DW_OP_breg4 (esp): 4)
    00000038 080483d3 080483d5 (DW_OP_breg4 (esp): 8)
    00000038 080483d5 08048422 (DW_OP_breg5 (ebp): 8)
    00000038 08048422 08048423 (DW_OP_breg4 (esp): 4)
    00000038 <End of list>
```

Offset column 0x38 values are the entries for main function variables. Each entry here describes possible frame base address with respect to where debugger may be paused by break point within function instructions; it specifies the current frame base from which offsets to variables are to be computed as an offset from a register. For x86, bpreg4 refers to esp and bpreg5 refers to ebp. Before analyzing further lets look at disassemble dump for *main* function

```
080483d2 <main>:
80483d2:      55                      push   %ebp
80483d3:      89 e5                   mov    %esp,%ebp
80483d5:      83 e4 f0                and    $0xfffffff0,%esp
80483d8:      83 ec 20                sub    $0x20,%esp
80483db:      c7 44 24 1c 0a 00 00    movl   $0xa,0x1c(%esp)
```

```
80483e2:      00
 80483e3:      c7 44 24 18 14 00 00   movl   $0x14,0x18(%esp)
80483ea:      00
 80483eb:      c7 44 24 14 c4 83 04   movl   $0x80483c4,0x14(%esp)
80483f2:      08
 80483f3:      8b 44 24 18            mov    0x18(%esp),%eax
80483f7:      89 44 24 04            mov    %eax,0x4(%esp)
80483fb:      8b 44 24 1c            mov    0x1c(%esp),%eax
80483ff:      89 04 24               mov    %eax,(%esp)
8048402:      8b 44 24 14            mov    0x14(%esp),%eax
8048406:      ff d0                         call   *%eax
8048408:      89 44 24 10            mov    %eax,0x10(%esp)
804840c:      b8 f0 84 04 08         mov    $0x80484f0,%eax
8048411:      8b 54 24 10            mov    0x10(%esp),%edx
8048415:      89 54 24 04            mov    %edx,0x4(%esp)
8048419:      89 04 24               mov    %eax,(%esp)
804841c:      e8 d3 fe ff ff         call   80482f4 <printf@plt>
8048421:      c9                     leave
 8048422:      c3                     ret
```

First two instructions deal with function's preamble, function's stack frame base pointer is determined after pre-amble instructions are executed. Ebp remains constant throughout function's execution and esp keeps changing with data being pushed and popped from the stack frame. From the above dump instructions at offset *80483db* and *80483e3* are assigning values 10 and 20 to variables *a*, and *b*. These variables are being accessed their offset in stack frame relative to location of current esp(variable a: 0x1c(%esp), variable b: 0x18(%esp)). Now let's assume that break point was set after initializing *a* and *b* variables and program paused, and we have run *print command* to view conents of *a* or *b* variables. Debugger would access 3rd record of the main function's dwarf debug.loc table since our break point falls between *080483d5 – 08048422 region of the function code***.**

```
00000038 080483d2 080483d3 (DW_OP_breg4 (esp): 4)
00000038 080483d3 080483d5 (DW_OP_breg4 (esp): 8)
00000038 080483d5 08048422 (DW_OP_breg5 (ebp): 8)--- 3rd record
00000038 08048422 08048423 (DW_OP_breg4 (esp): 4)
```

Now as per records debugger will locate a with esp + 28 , b with esp +24 and so on...

Looking up line number information

We can set breakpoints mentioning line no's Lets now look at how debuggers resolve line no's to machine instruction's? DWARF encodes a full mapping between lines in the C source code and machine code addresses in the executable. This information is contained in the .debug_line section and can be extracted using objdump

```
objdump --dwarf=decodedline ./sample
./sample:     file format elf32-i386
Decoded dump of debug contents of section .debug_line:
CU: sample.c:
File name                            Line number    Starting address
sample.c                                   5          0x80483c4
sample.c                                   6          0x80483c7
sample.c                                   7          0x80483d0
sample.c                                   9          0x80483d2
sample.c                                  10          0x80483db
sample.c                                  14          0x80483eb
sample.c                                  15          0x80483f3
```

```
sample.c                                16          0x804840c
sample.c                                17          0x8048421
```

This dump shows machine instruction line no's for our program. It is quite obvious that line no's for non-executable statements of the source file need not be tracked by dwarf .we can map the above dump to our source code for analysis.

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 int add(int x, int y)
 5 {
 6         return x + y;
 7 }
 8 int main()
 9 {
10         int a = 10, b = 20;
11         int result;
12         int (*fp) (int, int);
13
 14         fp = add;
15         result = (*fp) (a, b);
16         printf(" %dn", result);
17 }
```

From the above it should be clear of what debugger does it is instructed to set breakpoint at entry into function **add**, it would insert break point at line no 6 and pause after pre-amble of function add is executed.

What's next?

If you are into implementation of debugging tools or involved in writing programs/tools that simulate debugger facilities/ read binary files , you may be interested in specific programming libraries *libbfd* or *libdwarf***.**

Binary File Descriptor library (BFD)* or *libbfd as it is called provides ready to use functions to read into ELF and other popular binary files. BFD works by presenting a common abstract view of object files. An object file has a "header" with descriptive info; a variable number of "sections" that each has a name, some attributes, and a block of data; a symbol table; relocation entries; and so forth. Gnu binutils package tools like objdump, readelf and others have been written using these libraries.

Libdwarf is a C library intended to simplify reading (and writing) applications built with DWARF2, DWARF3 debug information.

Dwarfdump is an application written using libdwarf to print dwarf information in a human readable format. It is also open sourced and is copyrighted GPL. It provides an example of using libdwarf to read.

https://en.wikipedia.org/wiki/DWARF
http://www.dwarfstd.org
http://wiki.dwarfstd.org/index.php?title=Libdwarf_And_Dwarfdump
http://www.skyfree.org/linux/references/bfd.pdf

21 3 ಹಂಚಿಕೆಗಳು

ಹಂಚು