# Jserv's blog



## July 30, 2008

### Who Call Me (更新版)

兩週前在亞洲大學講 [快快樂樂學 GNU Debugger (gdb) Part I + II] 時，與一位同業的工程師談到，不透過 gdb 而能自我建立 backtrace，也就是在必要時建立函式呼叫的階層資訊，當時即提到 TimHsu 兄四年前的作品 [Who Call Me]，實在是極好的參考資訊。可惜，TimHsu 兄文中所使用的工具與 API 稍微過時，所以，取得他的同意下，筆者將該文改寫並更新，本文以 GNU/Linux 在 IA32 平台的運作為主。

**Who Call Me?**

原作：徐千洋 (TimHsu) 於 March 30, 2004
改作：Jim Huang (jserv) 於 July 30, 2008

---

誰呼叫我？使用過 gdb 對 core 檔作 bt (backtrace) 嗎？所謂 backtrace 是用來回溯檢查函式呼叫的關聯性，以便得知執行時期有哪個函式呼叫的動作，尤其是在許多錯綜複雜的龐大程式碼中，backtrace 是相當有用的 debug 技巧，而這個題目則是用來討論如何在程式執行中作 backtrace。

在實作這個技術前，有兩個關鍵點要先解決：

- 如何取得此 function 返回位址
- 如何依據返回位址查知函式名稱

關於第一點，須先瞭解堆疊 (stack) 和函式呼叫的處理關係。堆疊是一個後進先出 (LIFO, Last-In-First-Out) 的資料結構，當呼叫某個函式時，相關的暫存器 (register) 就會被存入堆疊。而當函式返回時，便會從堆疊裡取出返回位址，以便回到原來呼叫的下一個指令繼續執行。以 x86 暫存器組來說，其中 EIP 是 Instruction Pointer 之意，用以指出 CPU 將要執行指令的位址；ESP 暫存器則是用來指向目前堆疊的位址。

我們先寫個小程式來觀察： (test.c)

```
1 void test()
2 {
3 }
4
5 int main()
6 {
7     test();
8 }
```

透過 gdb 分析其執行行為：

```
jserv@venux:~/whocallme$ gcc -o test test.c
jserv@venux:~/whocallme$ gdb ./test
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...
```

```
(gdb) b test
Breakpoint 1 at 0x8048397
(gdb) r
Starting program: /home/jserv/whocallme/test

Breakpoint 1, 0x08048397 in test ()
Current language:  auto; currently asm
(gdb) info reg
eax            0xbff05fb4    -1074765900
ecx            0xbff05f30    -1074766032
edx            0x1    1
ebx            0xb7fceff4    -1208160268
esp            0xbff05f08    0xbff05f08
ebp            0xbff05f08    0xbff05f08
esi            0x80483d0     134513616
edi            0x80482e0     134513376
eip            0x8048397     0x8048397 <test+3>
eflags         0x282 [ SF IF ]
cs             0x73   115
ss             0x7b   123
ds             0x7b   123
es             0x7b   123
```

```
fs             0x0    0
gs             0x33   51
(gdb) disas test
Dump of assembler code for function test:
0x08048394 <test+0>:    push   %ebp
0x08048395 <test+1>:    mov    %esp,%ebp
0x08048397 <test+3>:    pop    %ebp
0x08048398 <test+4>:    ret
End of assembler dump.
(gdb)
```

由上可見，ebp 暫存器值為 0xbff05f08，也就是原來的堆疊位址。以 IA32 來說，函式呼叫 (對應機械指令為 "call") 的過程，CPU 會將返回位址存入堆疊，因此可從 ebp 暫存器的位址裡面，找到我們需要的返回位址。繼續透過 gdb 觀察：

```
(gdb) p/x *0xbff05f08
$1 = 0xbff05f18
```

別忘了，一進入此函式時，機械指令 "push $ebp" 已被執行 (詳見 test 函式反組譯的結果，也就是位址 0x08048394)，因此堆疊位址已被減 4，故，若要取得正確的值，需要再將位址加回 4，才可，也就是：

```
(gdb) p/x *(0xbff05f08+4)
$2 = 0x80483af
```

此值應該就是 test() 正確的返回位址，繼續透過 gdb 檢查看看：

```
(gdb) disas main
Dump of assembler code for function main:
0x08048399 <main+0>:    lea    0x4(%esp),%ecx
0x0804839d <main+4>:    and    $0xfffffff0,%esp
```

```
0x080483a0 <main+7>:pushl  -0x4(%ecx)
0x080483a3 <main+10>:      push   %ebp
0x080483a4 <main+11>:      mov    %esp,%ebp
0x080483a6 <main+13>:      push   %ecx
0x080483a7 <main+14>:      sub    $0x4,%esp
0x080483aa <main+17>:      call   0x8048394 <test>
0x080483af <main+22>:      add    $0x4,%esp
0x080483b2 <main+25>:      pop    %ecx
0x080483b3 <main+26>:      pop    %ebp
0x080483b4 <main+27>:      lea    -0x4(%ecx),%esp
0x080483b7 <main+30>:      ret
End of assembler dump.
(gdb)
```

果然在 "call <test>" 完後的下個指令，位址就是位於 0x80483af，這也就是 test() 返回位址。接下來，我們將前述的程式，透過 inline assembly 印出一些有用的訊息： (test-1.c)

```
1 #include <stdio.h>
2 void test()
3 {
4     unsigned int *stack;
5     asm ("movl %%ebp, %0\n"
6          : "=g"(stack));
7     printf("Return address = 0x%x\n", *(stack + 1));
8 }
9
```

```
10 int main()
11 {
12     test();
13 }
```

編譯並執行：

```
jserv@venux:~/whocallme$ gcc -o test-1 test-1.c
jserv@venux:~/whocallme$ ./test-1
Return address = 0x80483fd
```

再次，透過 gdb 來驗證 test() 函式的返回位址與機械指令 "call" 的關聯：

```
$ gdb ./test-1
(gdb) disas main
Dump of assembler code for function main:
0x080483e7 <main+0>: lea    0x4(%esp),%ecx
0x080483eb <main+4>: and    $0xfffffff0,%esp
0x080483ee <main+7>: pushl  -0x4(%ecx)
0x080483f1 <main+10>:       push   %ebp
0x080483f2 <main+11>:       mov    %esp,%ebp
0x080483f4 <main+13>:       push   %ecx
0x080483f5 <main+14>:       sub    $0x4,%esp
0x080483f8 <main+17>:       call   0x80483c4 <test>
0x080483fd <main+22>:       add    $0x4,%esp
0x08048400 <main+25>:       pop    %ecx
0x08048401 <main+26>:       pop    %ebp
```

```
0x08048402 <main+27>:      lea    -0x4(%ecx),%esp
0x08048405 <main+30>:      ret
End of assembler dump.
(gdb)
```

果然如此，所以我們已對本文一開始提出「如何取得此 function 返回位址」的問題，有了初步的解答，再來，就要思索，該如何依據記憶體位址，查知所處的函式名稱。

我們可透過 GNU binutils 的 objdump 工具程式分析 ELF 執行檔的重要資訊，首先觀察執行檔的符號表：

```
jserv@venux:~/whocallme$ objdump -t ./test-1 | awk '{print $1" "$3" "$6}'|grep "F"
08048340 F __do_global_dtors_aux
080483a0 F frame_dummy
080484e8 O __FRAME_END__
08048480 F __do_global_ctors_aux
08048410 F __libc_csu_fini
08048310 F _start
080484ac F _fini
08048420 F __libc_csu_init
080483c4 F test
0804847a F .hidden
080483e7 F main
08048298 F _init
```

既然 "objdump -t" 可印出程式的函式名稱和記憶體位址，不就是我們預期的動作嗎？所以，我們將重心擺在該工具程式背後的原理。objdump 是利用 BFD Library (Binary File Descriptor Library) 來實作的，底下的小程式也利用 BFD Library 來讀取符號表 (bfd.c)。注意：在 Debian/Ubuntu 下，需安裝套件 "binutils-dev"，方可編譯。

```
1  #include <stdio.h>
2  #include <bfd.h>
3
4  int main(int argc, char *argv[])
5  {
6      bfd *abfd;
7      long storage_needed;
8      asymbol **symbol_table;
9      long number_of_symbols;
10     long i;
11     char **matching;
12     sec_ptr section;
13     char *symbol_name;
14     long symbol_offset, section_vma, symbol_address;
15
16     if (argc < 2)
17         return 0;
18     printf("Open %s\n", argv[1]);
19     bfd_init();
20     abfd = bfd_openr(argv[1], NULL);
21     if (abfd == (bfd *) 0) {
22         bfd_perror("bfd_openr");
23         return -1;
```

```
24      }
25
26      if (!bfd_check_format_matches(abfd, bfd_object, &matching)) {
27          return -1;
28      }
29
30      if (!(bfd_get_file_flags (abfd) & HAS_SYMS)) {
31          printf("ERROR flag!\n");
32          return -1;
33      }
34
35      /* 取得符號表大小 */
36      storage_needed = bfd_get_symtab_upper_bound(abfd);
37      if (storage_needed < 0)
38          return -1;
39
40      symbol_table = (asymbol **) xmalloc(storage_needed);
41      /* 將符號表讀進所配置的記憶體裡(symbol_table), 並傳回符號表個數 */
42      number_of_symbols = bfd_canonicalize_symtab(abfd, symbol_table);
43      if (number_of_symbols < 0)
44          return -1;
45      for (i = 0; i < number_of_symbols; i++)
46      {
47          /* 檢查此符號是否為函式 */
48          if (symbol_table[i]->flags & (BSF_FUNCTION | BSF_GLOBAL)) {
49              /* 反查此函式所處的區段(section) 及
```

```
50              區段位址(section_vma) */
51            section = symbol_table[i]->section;
52            section_vma = bfd_get_section_vma(abfd, section);
53            /* 取得此函式的名稱(symbol_name)、
54               偏移位址(symbol_offset) */
55            symbol_name = symbol_table[i]->name;
56            symbol_offset = symbol_table[i]->value;
57            /* 將此函式的偏移位址加上區段位址，則為此函式在執行時
58               的記憶體位址(symbol_address */
59            symbol_address = section_vma + symbol_offset;
60            /* 檢查此函式是否處在程式本文區段 */
61            if (section->flags & SEC_CODE)
62                printf("<%s> 0x%x 0x%x 0x%x\n",
63                       symbol_name,
64                       section_vma,
65                       symbol_offset,
                         symbol_address);
          }
      }
      bfd_close(abfd);
  }
```

編譯並執行：

```
jserv@venux:~/whocallme$ gcc -o bfd bfd.c -lbfd
jserv@venux:~/whocallme$ ./bfd test-1
Open test-1
<__do_global_dtors_aux> 0x8048310 0x30 0x8048340
```

```
<frame_dummy> 0x8048310 0x90 0x80483a0
<__do_global_ctors_aux> 0x8048310 0x170 0x8048480
<__libc_csu_fini> 0x8048310 0x100 0x8048410
<_start> 0x8048310 0x0 0x8048310
<_fini> 0x80484ac 0x0 0x80484ac
<__libc_csu_init> 0x8048310 0x110 0x8048420
<test> 0x8048310 0xb4 0x80483c4
<__i686.get_pc_thunk.bx> 0x8048310 0x16a 0x804847a
<main> 0x8048310 0xd7 0x80483e7
<_init> 0x8048298 0x0 0x8048298
```

觀察由 objdump 工具程式與我們撰寫的小程式 bfd，對於 test() 函式的位址，有著相同的輸出，也就是 0x80483c4。現在，我們依照函式名稱及記憶體位址作對照表，即可立即查詢.，不過這其中仍有個小問題，就是，雖然知道個別函式的起始位址，但並不知道函式的結束位址，也不知道各函式程式內容的大小。要解決這個小問題，就必須在建立對照表時，先作排序，將位址越高的函式排在越後面，並將下一個函式的起始位址當作結束位址。於是筆者建立於前面的 bfd.c 程式，提出新的工具程式 (bfd_dumpfun.c)

```
1  /* bfd_dumpfun.c (GPL)
2   *
3   * Usage: ./bfd_dumpfun [binary]
4   * Note: Dump functions infomation of ELF-binary with BFD Library.
5   *
6   * by TimHsu(timhsu@info.sayya.org) 2004/03/31
7   * Modified by Jim Huang <jserv.tw@gmail.com>, 2008/07/22
8   *   - Bump bfd APIs and build fixes.
9   */
10
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <bfd.h>

typedef struct function_table FUN_TABLE;

/* 宣告一個包含函式名稱和位址的結構 */
struct function_table
{
    char name[80];
    unsigned long addr;
};

static FUN_TABLE *fun_table;
static int table_count = 0;   /* 函式個數 */

static int compare_function(const void *a, const void *b)
{
    FUN_TABLE *aa = (FUN_TABLE *) a, *bb = (FUN_TABLE *) b;
    if (aa->addr > bb->addr)
        return 1;
    else if (aa->addr < bb->addr)
        return -1;
    return 0;
}
```

```
37
38 /* 增加一個函式資料至對照表 */
39 static void add_function_table(char *name, unsigned long address)
40 {
41     strncpy(fun_table[table_count].name, name, 80);
42     fun_table[table_count].addr = address;
43     table_count++;
44 }
45
46 static void dump_function_table(void)
47 {
48     int i;
49     for (i = 0; i < table_count; i++)
50     {
51         printf("%-30s 0x%x\n",
52                fun_table[i].name, fun_table[i].addr);
53     }
54 }
55
56 int main(int argc, char *argv[])
57 {
58     bfd *abfd;
59     asection *text;
60     long storage_needed;
61     asymbol **symbol_table;
62     long number_of_symbols;
```

```
63      long i;
64      char **matching;
65      sec_ptr section;
66      char *symbol_name;
67      long symbol_offset, section_vma, symbol_address;
68
69      if (argc < 2)
70          return 0;
71
72      printf("Open %s\n", argv[1]);
73      bfd_init();
74      abfd = bfd_openr(argv[1], NULL);
75      if (abfd == (bfd *) 0) {
76          bfd_perror("bfd_openr");
77          return -1;
78      }
79
80      if (!bfd_check_format_matches(abfd, bfd_object, &matching)) {
81          return -1;
82      }
83
84      if (!(bfd_get_file_flags (abfd) & HAS_SYMS)) {
85          printf("ERROR flag!\n");
86          return -1;
87      }
88      if ((storage_needed = bfd_get_symtab_upper_bound(abfd)) < 0)
```

```
89          return -1;
90
91      symbol_table = (asymbol **) xmalloc(storage_needed);
92      number_of_symbols = bfd_canonicalize_symtab(abfd, symbol_table);
93      if (number_of_symbols < 0)
94          return -1;
95
96      fun_table = (FUN_TABLE **)malloc(sizeof(FUN_TABLE) * number_of_symbols);
97      bzero(fun_table, sizeof(FUN_TABLE)*number_of_symbols);
98
99      for (i = 0; i < number_of_symbols; i++)
100     {
101         if (symbol_table[i]->flags & (BSF_FUNCTION | BSF_GLOBAL)) {
102             section = symbol_table[i]->section;
103             section_vma = bfd_get_section_vma(abfd, section);
104             symbol_name = symbol_table[i]->name;
105             symbol_offset = symbol_table[i]->value;
106             symbol_address = section_vma + symbol_offset;
107             if (section->flags & SEC_CODE) {
108                 add_function_table(symbol_name,
109                                    symbol_address);
            }
        }
    }
    bfd_close(abfd);
    /* 將函式對照表作排序 */
```

```
        qsort(fun_table, table_count, sizeof(FUN_TABLE), compare_function);
        dump_function_table();
    }
```

編譯並執行：

```
jserv@venux:~/whocallme$ gcc -o bfd_dumpfun bfd_dumpfun.c -lbfd
jserv@venux:~/whocallme$ ./bfd_dumpfun ./test-1
Open ./test-1
_init                          0x8048298
_start                         0x8048310
__do_global_dtors_aux           0x8048340
frame_dummy                    0x80483a0
test                           0x80483c4
main                           0x80483e7
__libc_csu_fini                 0x8048410
__libc_csu_init                 0x8048420
__i686.get_pc_thunk.bx          0x804847a
__do_global_ctors_aux           0x8048480
_fini                          0x80484ac
```

現在，我們已將技術的關鍵點都處理好，為能實用化，最好是作成函式庫，得以日後隨時呼叫。我們的函式庫包含兩部份： whocallme.[ch]，首先是標頭檔部份： (whocallme.h)

```
1 #include <stdio.h>
2
3 #define       FUNCTION_NAME_MAXLEN        80
```

```
4
5 #define who_call_me() \
6     do { \
7         unsigned int *stack; \
8         asm ("movl %%ebp, %0\n"  \
9              : "=g"(stack));  \
10        fprintf(stderr, \
11                "<whocallme>: function <%s> call me <%s>!\n", \
12                find_function_by_addr(*(stack+1)), who_am_i()); \
13    } while(0)
14
15 extern int init_function_table(char *);
```

留意到巨集定義中的自訂函式 who_am_i()，目的自然就是取得執行中的函式名稱，整個實做如下： (whocallme.c)

```
1  /* whocallme.c (GPL)
2   *
3   * A runtime backtrace of function.
4   *
5   * by Timhsu(timhsu@chroot.org) 2004/03/31
6   * Modified by Jim Huang <jserv.tw@gmail.com>, 2008/07/22
7   *   - Bump bfd APIs.
8   *   - Eliminate compiler errors.
9   */
10
11 #include <stdio.h>
12 #include <stdlib.h>
13 #include <string.h>
```

```
14 #include <bfd.h>
15 #include "whocallme.h"
16
17 /* forward declarations */
18 char *find_function_by_addr(unsigned long addr);
19
20 typedef struct function_table FUN_TABLE;
21 /* 宣告一個包含函式名稱和位址的結構 */
22 struct function_table
23 {
24     char name[FUNCTION_NAME_MAXLEN];
25     unsigned long addr;
26 };
27
28 static FUN_TABLE *fun_table;
29 static int table_count = 0; /* 函式個數 */
30
31 static int compare_function(const void *a, const void *b)
32 {
33     FUN_TABLE *aa = (FUN_TABLE *) a;
34     FUN_TABLE *bb = (FUN_TABLE *) b;
35     if (aa->addr > bb->addr)
36         return 1;
37     else if (aa->addr < bb->addr)
38         return -1;
39     return 0;
```

```
}

/* 增加一個函式資料至對照表 */
static void add_function_table(char *name, unsigned long address)
{
    strncpy(fun_table[table_count].name, name, FUNCTION_NAME_MAXLEN);
    fun_table[table_count].addr = address;
    table_count++;
}

/* 取得目前正在執行的函式名稱 */
char * who_am_i(void)
{
    unsigned long *stack; \
    asm ("movl %%ebp, %0\n"  \
         : "=g"(stack));
    return find_function_by_addr(*(stack + 1));
}

/* 依照位址取得函式名稱 */
char *find_function_by_addr(unsigned long addr)
{
    int i;
    for (i = 0; i < table_count; i++)
    {
        if (addr > fun_table[i].addr)
```

```
66          {
67              if (addr < fun_table[i + 1].addr)
68                  return fun_table[i].name;
69          }
70      }
71      return NULL;
72  }
73
74  /* 初始化函式對照表 */
75  int init_function_table(char *file)
76  {
77      bfd *abfd;
78      long storage_needed;
79      asymbol **symbol_table;
80      long number_of_symbols;
81      long i;
82      char **matching;
83      sec_ptr section;
84      char *symbol_name;
85      long symbol_offset, section_vma, symbol_address;
86
87      bfd_init();
88      abfd = bfd_openr(file, NULL);
89      if (abfd == (bfd *) 0) {
90          bfd_perror("bfd_openr");
91          return -1;
```

```
92      }
93
94      if (!bfd_check_format_matches(abfd, bfd_object, &matching)) {
95          return -1;
96      }
97
98      if (!(bfd_get_file_flags (abfd) & HAS_SYMS)) {
99          printf("ERROR flag!\n");
100         return -1;
101     }
102
103     /* 取得符號表大小 */
104     storage_needed = bfd_get_symtab_upper_bound(abfd);
105     if (storage_needed < 0)
106         return -1;
107
108     symbol_table = (asymbol **) malloc(storage_needed);
109     /* 將符號表讀進所配置的記憶體裡(symbol_table), 並傳回符號表個數 */
110     number_of_symbols = bfd_canonicalize_symtab(abfd, symbol_table);
111     if (number_of_symbols < 0)
112         return -1;
113
114     /* 配置空間給函式對照表 */
115     fun_table = (FUN_TABLE *) malloc(sizeof(FUN_TABLE) * number_of_symbols);
116     bzero(fun_table, sizeof(FUN_TABLE)*number_of_symbols);
117
```

```
118     for (i = 0; i < number_of_symbols; i++)
119     {
120         /* 檢查此符號是否為函式 */
121         if (symbol_table[i]->flags & (BSF_FUNCTION | BSF_GLOBAL)) {
122             /* 反查此函式所處的區段(section) 及區段位址(section_vma) */
123             section = symbol_table[i]->section;
124             section_vma = bfd_get_section_vma(abfd, section);
125 
126             /* 取得此函式的名稱(symbol_name), 偏移位址(symbol_offset) */
127             symbol_name = (char *) symbol_table[i]->name;
128             symbol_offset = symbol_table[i]->value;
129 
130             /* 將此函式的偏移位址加上區段位址，則為此函式
131              * 在執行時的記憶體位址 (symbol_address) */
132             symbol_address = section_vma + symbol_offset;
133 
134             /* 檢查此函式是否處在程式本文區段 */
135             if (section->flags & SEC_CODE) {
                    /* 將此函式名稱和位址加入至對照表 */
                    add_function_table(symbol_name,
                                       symbol_address);
                }
            }
        }
        free(symbol_table);
        bfd_close(abfd);
```

```
        /* 將函式對照表作排序 */
        qsort(fun_table, table_count, sizeof(FUN_TABLE), compare_function);
        return 0;
    }
```

建構此函式庫方式如下：

```
jserv@venux:~/whocallme$ gcc -c whocallme.c
jserv@venux:~/whocallme$ ar -q libwhocallme.a whocallme.o
```

寫個簡短的測試程式，看看執行的效果： (test-2.c)

```
1  #include "whocallme.h"
2
3  void test()
4  {
5      who_call_me();
6  }
7  void test_a()
8  {
9      test_b();
10     test_c();
11 }
12 void test_b()
13 {
14     test();
15 }
16 void test_c()
17 {
```

```
18     who_call_me();
19 }
20 int main(int argc, char *argv[])
21 {
22     init_function_table(argv[0]);
23     test();
24     test_a();
25     test_b();
26     test_c();
27 }
```

編譯並執行：

```
jserv@venux:~/whocallme$ gcc -o test-2 test-2.c -lbfd -L. -lwhocallme
jserv@venux:~/whocallme$ ./test-2
: function <main> call me <test>!
: function <test_b> call me <test>!
: function <test_a> call me <test_c>!
: function <test_b> call me <test>!
: function <main> call me <test_c>!
```

下載本文的範例程式： [whocallme.tar.bz2]

由 jserv 發表於 July 30, 2008 02:28 PM

迴響

如在此使用 GCC builtin function: __builtin_return_address 與__builtin_frame_address，或可少掉很多 architecture-dependent dirty work

由 I-Jui Sung 發表於 July 30, 2008 03:01 PM

發生 segfault 的時候怎樣取得 backtrace 呢?

由 jwang 發表於 July 30, 2008 05:42 PM

@ijsung,
很好的建議，感謝提醒

@jwang,
將 backtrace 的實做加入 SIGSEGV 的 signal handler 即可

由 jserv 發表於 July 30, 2008 05:49 PM

為什麼要用 do while(0) 去把 marco 包著呢?
一直看不明白

由 3322 發表於 July 31, 2008 03:38 AM

@3322,
因為考量區域變數 unsigned int *stack; 不該展開後，在同一個 scope 相互污染，再來，這是避免 dangling-else 的慣用技巧

由 jserv 發表於 July 31, 2008 04:17 AM

很有用的工具。
請問可以列出完整的 backtrace，而不是只有上一層嗎？

由 Vincent 發表於 July 31, 2008 11:25 AM

@Vincent,
依序將 stack frame 往上移動即可，當然，可用遞迴的技巧

由 jserv 發表於 July 31, 2008 11:41 AM

若是用 gnu C Lib 的話有包好的:
http://www.gnu.org/software/libc/manual/html_node/Backtraces.html#Backtraces
搭配"-rdynamic"可印出 function name.(沒有的話只能得到 address)

由 lbr 發表於 August 6, 2008 02:25 PM

改了一點小東東
利用 -finstrument-functions 和 whocallme 的 symbal table lookup 來印出 caller 和 calle name ..:P

http://cmchao.pixnet.net/blog/post/21519972/

由 cmchao 發表於 August 25, 2008 10:16 AM

上面取得返回位址不用這麼麻煩還要 inline asm 吧.
這樣就好了:

```
#include
void
test()
{
    unsigned int *stack;

    printf("Return address = %p\n", (&stack)[2]);
}
```

```
int
main()
{
    test();
}
```

由 吳俊緯 發表於 September 30, 2008 03:14 PM

之前 jserv 寫了篇很有趣的文章(Who Call Me ?)，利用一些小技巧去取 caller 和 callee 之間的 information。不過每個 trace 的地方都還要手動塞 code，如果要在往上看再去看 stack frame，我對 x86 實在是很不熟。

今天在寫 profiling 的東東想到是不是可以用 instrumentation(註一)的方式來得到同樣的資訊，把 whocallme 的 call 塞到每個 function 裡。原本以為-pg(gprof)可以做得到，因為 gprof 本來就是利用偷塞一些 code 來統計資訊，但 gcc 似乎不允許你塞自己的 function。

那有其它的方式嗎？果然 gcc support -finstrument-functions 這個選項，會在你所有 function 前後各塞入一個 function，讓 user 可以蒐集或輸出一些資訊

```
void __cyg_profile_func_enter (void *this_fn, void *call_site);
void __cyg_profile_func_exit  (void *this_fn, void *call_site);
```

使用方式呢？這種技巧 jserv 去年已經介紹過了，請看這裡。那現在就是把他們組合起來啦，底下是 patch

```
--- ../whocallme/whocallme.c    2008-07-30 13:57:30.000000000 +0800
+++ ../whocallme2/whocallme.c   2008-08-20 17:59:23.000000000 +0800
@@ -60,7 +60,7 @@
{
    int i;
    for (i = 0; i < table_count; i++) {
-       if (addr > fun_table[i].addr) {
+       if (addr >= fun_table[i].addr) {
            if (addr < fun_table[i + 1].addr)
                return fun_table[i].name;
        }
@@ -134,3 +134,12 @@
```

```
    qsort(fun_table, table_count, sizeof(FUN_TABLE), compare_function);
    return 0;
}
+
+#define DUMP(func, call) \
+        printf("%s: func = %s, called by = %s\n", __FUNCTION__, func, call)
+
+void __attribute__((__no_instrument_function__))__cyg_profile_func_enter(void *this_func, void *call_site)
+{
+        DUMP(find_function_by_addr((unsigned long)this_func),
+        find_function_by_addr((unsigned long)call_site));
+}
```

在編譯你自已的程式時加入 -finstrument-functions(編譯 whocallme.c 的時候不用，不然每個 function 都要再補上 no_instrucment_function 的 attribute

，然後輸出就會長成這樣，

```
__cyg_profile_func_enter: func = test, called by = main
__cyg_profile_func_enter: func = test_a, called by = main
__cyg_profile_func_enter: func = test_b, called by = test_a
__cyg_profile_func_enter: func = test, called by = test_b
__cyg_profile_func_enter: func = test_c, called by = test_a
__cyg_profile_func_enter: func = test_b, called by = main
__cyg_profile_func_enter: func = test, called by = test_b
__cyg_profile_func_enter: func = test_c, called by = main
```