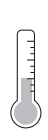
Profiler・Debugger Hacks

HACK #87-92

本章將介紹與 profiler、debugger 有關的技巧。電腦的性能隨著 CPU、記憶體性能提昇的支持而有飛躍的進展，儘管如此，受到龐大流量束縛的網站營運，以及資料庫、動畫處理等應用，不管有多少性能都嫌不夠。寫出高速的程式，仍然是程式設計者的重要課題之一。另外，寫出沒有臭蟲的程式，說是程式設計者最重要的課題也不為過。

本章首先將介紹開發高速程式時可以幫上忙的各種 profiler (效能量測器) 的基本使用方式與機制，接著會介紹 debugger (除錯器) 使用上的方技巧。



HACK #87

以 gprof 量測執行效能

profiling 指的是調查程式是在哪個處理部份花時間。本 hack 將介紹以 gprof 進行 profiling 的方法。

一般的程式，會在效能上造成瓶頸的，常常只有很小一部份。為了找出程式最慢的部份，而調查程式在哪個部份花了多少時間、找出程式的 profile (輪廓)，就是 profiling。本 hack 要介紹以 gprof 進行 profiling 的方法。

用法

試著調查下面這個簡單程式的 profile 看看。

```
void slow() {
  int i;
  for (i = 0; i < 2000000; i++);
}

void f() {
  int i;
  for (i = 0; i < 1000; i++) slow();
}
```

```
void g() {
  int i;
  for (i = 0; i < 4000; i++) slow();
}

int main() {
  f();
  g();
}
```

像下面這樣編譯，然後使用 gprof。

```
% gcc -O -g -pg bench.c
% ./a.out
% gprof a.out
```

請注意 gcc 要加上 -pg 選項，讓程式輸出 profile 資訊。執行 ./a.out 之後，就會產生 gmon.out 檔案、輸出 profile 資訊。gprof a.out 的輸出摘錄如下。

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
100.00     25.86    25.86     5000     0.01     0.01  slow
  0.00     25.86     0.00        1     0.00     5.17  f
  0.00     25.86     0.00        1     0.00    20.69  g

index % time    self  children    called     name
                5.17    0.00    1000/5000        f [4]
               20.69    0.00    4000/5000        g [3]
[1]    100.0   25.86    0.00    5000            slow [1]
-----------------------------------------------
                                                 <spontaneous>
[2]    100.0    0.00   25.86                     main [2]
                0.00   20.69       1/1              g [3]
                0.00    5.17       1/1              f [4]
-----------------------------------------------
                0.00   20.69       1/1           main [2]
[3]     80.0    0.00   20.69       1             g    [3]
               20.69    0.00    4000/5000        slow [1]
-----------------------------------------------
                0.00    5.17       1/1           main [2]
[4]     20.0    0.00    5.17       1             f    [4]
                5.17    0.00    1000/5000        slow [1]
-----------------------------------------------
```

最初的表格是函式需要的時間。slow 函式消費了大多數時間 (self seconds)、g 呼叫 slow 的次數是 f 的四倍，所以總消費時間 (total s/call) 也差不多是四倍。

第二個表格 (call graph) 可以看出函式之間的呼叫關係。可以看出 main 呼叫了 f 與 g、f 與 g 呼叫了 slow。

gprof 加上 -l 就可以看出逐行需要的時間。這需要編譯時以 -g 選項加上除錯資訊。使用 -l -A -x 就會顯示原始碼、並且一起顯示每行需要的時間。若只想看到特定函式的 call graph，可以用 gprof -F slow 之類的寫法指定。

請注意，在使用 gcc -pg 的時候，所有處理內容都會嵌入遞增計數器的處理，因而導致效能低於一般執行時的狀況。

原理

gcc -pg 產生的 binary 會留下三種 profile 資訊。預設會儲存到 gmon.out 檔案內，不過可以透過 GMON_OUT_PREFIX 環境變數改變輸出的檔名。這份 profile 資訊的細節在「GNU gprof - Implementation」(http://sourceware.org/binutils/docs/gprof/Implementation.html) 有詳細的說明。

首先，第一個是以 timer 每隔 10 ms 檢查一次 program counter 的位置。這是以「[HACK #31] 在 main() 之前呼叫函式」介紹的方法，在 main 之前呼叫 setitimer(2) 函式，每隔 10 ms 產生 SIGPROF。SIGPROF 呼叫的 handler (glibc 的 sysdeps/posix/sprofil.c 的 profil_count) 會遞增計數器。實際上是使用 glibc 的 profil(3) 實現這個動作。

第二個是函式的 call graph 資訊。這是以類似「[HACK #77] 附掛在函式的 enter/exit」介紹的方法，在進入函式時呼叫 mcount 函式。mcount 是各架構不同的一小段組合語言碼 (glibc 的 sysdeps/i386/i386-mcount.S 之類)，裡面在取得呼叫者的 PC 與呼叫後的 PC 之後，會呼叫各架構通用的 __mcount_internal，而後以 PC 資訊建立 call graph 資訊，紀錄正確的函式呼叫次數。

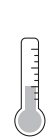
第三個是使用老舊 GCC 版本時需要的資訊，在各個基本區塊 (if 區塊與 while 區塊等等) 逐一紀錄進入區塊的次數。這些計數器是由編譯器建立的。這些計數器能提供完整涵蓋的資訊，但會讓執行變得非常慢，導致時間方面的 profiling 變得非常不正確。現在已經不會紀錄這些資訊，想知道涵蓋範圍的時候，使用 gcov 即可。

執行 gprof 指令後，會開啟紀錄了這些資訊的檔案，取得正確的 call graph 與函式呼叫次數資訊，另外也能根據取樣結果推算出大致需要的時間。另外還會以類似「[HACK #67] 以 libbfd 取得 symbol 一覽」介紹的方法從執行檔取得位址、對應的符號名稱，整合在一起顯示。

小結

本 hack 介紹了以 gprof 進行效能量測的方法，以及它的運作原理。

— *Shinichiro Hamaji*



HACK
#88

以 sysprof 輕鬆量測系統效能

使用 sysprof 即可非常輕鬆地取得簡單的 profile。

本 hack 將介紹取得系統整體 profile 的 sysprof 程式。

安裝

本節執筆時 sysprof (`http://www.daimi.au.dk/~sandmann/sysprof/`) 最新的版本是 1.0.2，採用 GPL2 授權。

sysprof 是以 Linux kernel module 以及 GUI 的使用者程式組合在一起運作的程式。目前只支援 x86 與 x86-64。

首先需要打開 Linux kernel 的 profiling 支援。若使用中的 kernel 沒有支援相關功能的話，2.6.11 之後的 kernel 請打開 `CONFIG_PROFILING` 再重新編譯。順便把 `CONFIG_OPROFILE` 一起打開的話「[HACK #89] 以 oprofile 詳細量測系統效能」也就能一起使用。另外 GUI 的部份需要 GTK+ 2.6.0 之後的版本與 libglade 2.5.1 之後的版本。

完成上述工作後，只要解開 `sysprof` 的壓縮檔執行 `./configure; make; make install` 即可安裝完成。之後再以 `modprobe sysprof-module` 等指令載入模組，就做好準備了。

用法

sysprof 的用法幾乎是不用說明地簡單。想取得特定程式的 profile 時，只要執行 sysprof 按下 Start 按鈕，執行想量測的程式，執行完畢之後再按 Profile 按鈕就可以了。這邊跟「[HACK #87] 以 gprof 量測執行效能」一樣，採用下面這個程式作為範例。

```
void slow() {
  int i;
  for (i = 0; i < 2000000; i++);
}

void f() {
  int i;
  for (i = 0; i < 1000; i++) slow();
}

void g() {
  int i;
  for (i = 0; i < 4000; i++) slow();
}

int main() {
  f();
  g();
}
```

不必特別新增編譯選項。除錯 symbol 沒有也沒關係。只要不把 symbol 給 strip 掉就 OK。

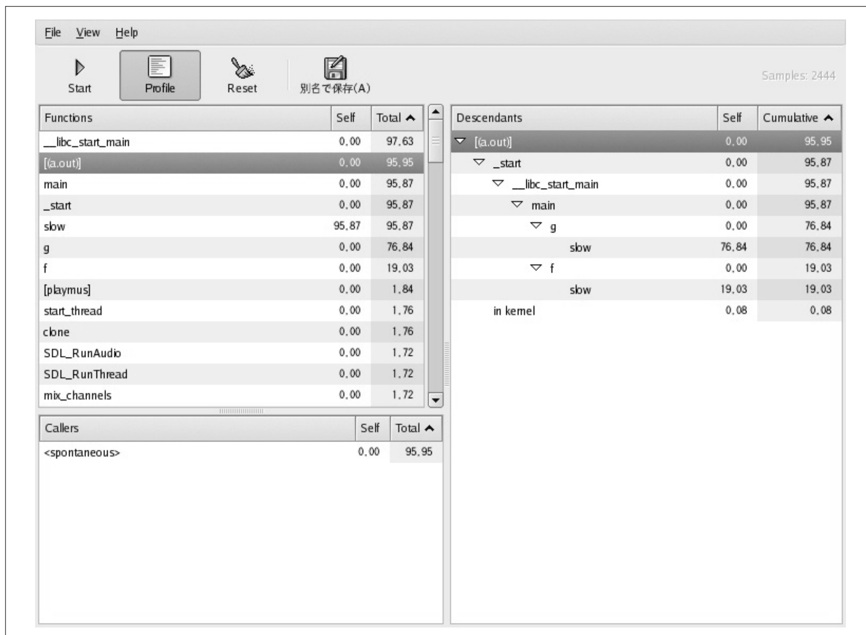


圖 6-1. sysprof 的執行畫面

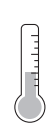
取到這個範例程式的 profile 時的畫面如圖 6-1 所示。這邊看起來也是 f 與 g 函式的比例大概在 2:8。另外因為這是系統整體的 profiler，而筆者當時正好也在聽音樂，所以 playmus 這個程式也出現在清單裡。這個 profile 可以用 xml 格式儲存下來。

sysprof 執行過程中會以很高的比例進行取樣、留下當時 program counter 的位置，儘管不見得會得到完全正確的結果，若只想尋找程式瓶頸的話，應該已十分具有實用價值了。

小結

本 hack 介紹了使用 sysprof 取得系統整體 profile 的方法。使用 sysprof 可以非常輕鬆地取得簡單的 profile。

— *Shinichiro Hamaji*



HACK #89

以 oprofile 詳細量測系統效能

本 hack 將介紹取得系統 profile 的 oprofile 程式。oprofile 支援許多 CPU，可獲得比 sysprof 更詳細的 CPU 資訊。

「[HACK #88] 以 sysprof 輕鬆量測系統效能」介紹的 sysprof 是用途受限、但操作起來非常簡單的程式。本 hack 介紹的 oprofile 與其相對，需要比較複雜的操作，但是可以在許多 CPU 上運作，可透過 CPU 的各種功能取得各式各樣的系統 profile。

安裝

本節執筆時 oprofile 的版本是 0.9.1。oprofile 是以 GPL2 授權發佈。oprofile 與 sysprof 一樣，是 kernel module 與使用者程式協調運作的軟體。根據文件說明，Linux 2.2 支援 x86、Linux 2.4 新增支援 IA-64、Linux 2.6 新增支援 Alpha、MIPS、ARM、x86-64、sparc64、ppc64，同時也對 PA-RISC 與 s390 提供有限的支援。

筆者在 Mobile Celeron 1.7 GHz、Pentium 4 2.53 GHz、Xeon 3.2 GHz 雙處理器三種環境下測試了 oprofile。Xeon 豐富的 CPU 資訊全部都可以使用、Pentium 4 只能取得簡單的資訊、Mobile Celeron 則無法取得 profile。根據 oprofile 網站說明，似乎在某些筆記型電腦上面無法順利運作的樣子。本 hack 是在 Xeon 機器上進行確認的。

與 sysprof 的 hack ([HACK #88]) 介紹的一樣，kernel 的設定要打開 CONFIG_PROFILING 與 CONFIG_OPROFILE，然後 oprofile 跟平常一樣 ./configure; make; make install 就安裝完成了。另外，若想對 kernel 本身以 oprofile 進行測定的話，也順便打開 CONFIG_FRAME_POINTER 選項以便取得 kernel 的 call graph 吧。

基本的用法

oprofile 在取得 profile 的過程是以 opcontrol 控制、閱覽取得的 profile 時是使用 opreport 與 opannotate。它也提供了 oprof_start 這個使用 Qt 的 GUI 介面，但筆者嘗試的時候，用起來並不方便，所以本 hack 只解釋 CUI 的指令。

首先，完成基本的設定。最低限度是像下面這樣，設定 kernel 的位置 (請注意，在指定 kernel 的時候，不能使用壓縮過的 vmlinuz) 以及要紀錄的 call graph 數量。

```
% opcontrol --vmlinux=/usr/src/linux-2.6.15.4/vmlinux --callgraph=20
```

這些設定會紀錄在 $HOME/.oprofile/daemonrc，設定一次之後就不必再變動了。以下指令即可啟動 oprofiled daemon process。

```
% opcontrol --start
```

執行下列指令即可儲存 profile 資料。

```
% opcontrol --dump
```

想清空所有 profile 計數器的時候可使用 --reset 選項、想停止的時候可以用 --stop、想結束 daemon 的時候可以用 --shutdown。

與 gprof、sysprof 的 hacks 一樣 ([HACK #87]、[HACK #88])，嘗試取得下面這段程式碼的 profile 看看吧。

```
void slow() {
  int i;
  for (i = 0; i < 2000000; i++);
}
void f() {
  int i;
  for (i = 0; i < 1000; i++) slow();
}
```

```
void g() {
  int i;
  for (i = 0; i < 4000; i++) slow();
}

int main() {
  f();
  g();
}
```

在 oprofiled 啟動的狀態，執行下面這些動作。

```
% gcc -O -g bench.c
% opcontrol --reset
% ./a.out
% opcontrol --dump
```

這樣就應該會把 profile 存成檔案。用 opreport 看看 profile 吧。它支援各種選項，這邊先以 -c 看看 call graph。

```
% opreport -c

samples  %        image name           app name           symbol name
-------------------------------------------------------------------------------
  5        0.0051  a.out                 a.out              main
  19409   19.9324  a.out                 a.out              f
  77960   80.0624  a.out                 a.out              g
  97374   53.1753  a.out                 a.out              slow
  97374   100.000  a.out                 a.out              slow [self]
-------------------------------------------------------------------------------
  12937    7.0648  libruby.so.1.8.3      libruby.so.1.8.3   (no symbols)
  12937   100.000  libruby.so.1.8.3      libruby.so.1.8.3   (no symbols) [self]
... 略 ...
```

跟前面一樣，大致可以看出 f:g 使用的時間約略成 2:8 的比例。由於使用了 CPU 的功能，可以看出取樣數比 sysprof 多很多。另外它與 sysprof 一樣是取得系統整體 profile 的軟體，所以下面看得到 libruby.so.1.8.3 之類來自其他 process 的資訊。

接著用 opannotate 看看原始碼層級的 profile 資訊吧。opannotate 加上 -a 選項就會顯示組合語言碼、加上 -s 選項就會顯示原始碼。顯示原始碼的時候需要除錯資訊。兩個都加上的時候應該最容易瞭解。

```
% opannotate -a -s
... 只列出 slow 函式的部份 ...
08048348 <slow>: /* slow total:   97374  5.6869 */
               :void slow() {
               : 8048348:        push   %ebp
               : 8048349:        mov    %esp,%ebp
               : 804834b:        mov    $0x1e8480,%eax
               :    int i;
               :    for (i = 0; i < 2000000; i++);
 97361  5.6861 : 8048350:        dec    %eax
     7 4.1e-04 : 8048351:        jne    8048350 <slow+0x8>
               :}
     1 5.8e-05 : 8048353:        leave
     5 2.9e-04 : 8048354:        ret
```

偵測 Cache Miss

使用 oprofile 的時候，不僅有 profile 資訊，也能偵測各式各樣事件的發生次數。可以偵測的事件清單以 `oprofile --list-events` 即可列出。想偵測 cache miss 的時候，偵測 `BSQ_CACHE_REFERENCE` 即可。增加調查事件的方式舉例如下。

```
% opcontrol --event=BSQ_CACHE_REFERENCE:100000:0x100
% opcontrol --shutdown
% opcontrol --start
```

100000 這個數字是取樣頻率，`0x100` 則是單位遮罩，用來選擇要紀錄的事件用的。指定 0x100 應該會偵測到讀取時 L2 cache miss 才對。這個數值在「IA-32 Intel Architecture Software Developer's Manual Volume 3B: System Programming Guide, Part 2」的 Appendix A.4 Table A.11 可以查到。這次調查的程式碼是下面這段簡單的範例。

```
#include <stdlib.h>
#define NUM 10000
int cache_miss() {
    int* a[NUM];
    int i, j, sum;
    for (i = 0; i < NUM; i++) a[i] = (int*)malloc(sizeof(int)*NUM);
    for (i = 0; i < NUM; i++)
        for (j = 0; j < NUM; j++)
//          sum += a[i][j];
            sum += a[j][i];
    return sum;
```

```
}
int main() {
    return cache_miss();
}
```

這是把二維陣列內容全部加起來的程式碼，因為最內圈取的是 a[j][i]，所以取用的順序是 a、a+10000、a+20000、...、a+1、a+10001、...，應該比註解掉的 a[i][j] 版本更容易引發 cache miss。

```
% opcontrol --reset
% ./a.out
% opcontrol --dump
% opreport -c

samples  %       image name              app name         symbol name
-------------------------------------------------------------------------------
  231      100.000  a.out                    a.out           main
  231      94.6721  a.out                    a.out           cache_miss
  231      100.000  a.out                    a.out           cache_miss [self]
```

似乎總共引起 231 次 cache miss。接著把註解的部份交換後編譯執行，得到的結果如下。

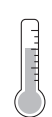
```
  1        100.000  a.out                    a.out            main
  1        10.0000  a.out                    a.out            cache_miss
  1        100.000  a.out                    a.out            cache_miss [self]
```

Cache miss 的次數只有一次，變得非常少。可以說是預料中的結果。

小結

本 hack 介紹了取得系統 profile 的軟體 oprofile。

— *Shinichiro Hamaji*



HACK
#90

以 GDB 操作執行中的 process

本 hack 將積極活用 GDB 擁有的 ptrace(2) 系統呼叫功能，介紹幾個操作正在執行的 process 的例子。

GDB 可作為 ptrace(2) 系統呼叫的便利前端介面來用。特別是「可以在運算式中呼叫目標 process 之內的函式」這個強大的功能，與 ptrace 的附掛功能組合在一起運用，即可簡單地直接干涉正在執行的 process。

活用範例

在此介紹幾個單純的活用例。以下是在 x86 的 Debian GNU/Linux 系統上進行實驗。若使用的 GDB 是 6.4 之前的版本，就可能有一些例子無法正常運作。

```
% ps x | grep firefox
 3616 ?        Rl    19:40 /usr/lib/firefox/firefox-bin -a firefox
% gdb -q -p 3616
(gdb) p chdir("/")
[Switching to Thread -1221168480 (LWP 3616)]
$1 = 0
(gdb) detach
Detaching from program: /usr/lib/firefox/firefox-bin, process 3616
(gdb)
```

這個例子是在 `gdb -p` 附掛的目標 process 之內取用 `chdir(2)` 系統呼叫。當 daemon 或 X client 之類在背景運作的 process 佔用掛載點下層目錄，導致 `device is busy` 而無法 `umount` 的時候很有用。

檔案操作也能比較簡單地達成。首先啟動 `cat`，在另一個 shell 像下面這樣啟動 GDB 玩弄一下 cat process 看看。

```
% gdb -q -p $(pidof cat)
(gdb) p write(1, "hoge", 4)
$1 = 4
(gdb) p open("/etc/passwd", 0)   0 是 O_RDONLY
$2 = 3
(gdb) p dup2(3, 0)
$3 = 0
(gdb) c
Continuing.

Program exited normally.
(gdb)
```

如果 `cat` process 輸出了 `hoge` 字串與 `/etc/passwd` 檔案內容的話就是成功了。如果鬥志與根性夠的話，用 socket 建立新的 TCP 連線也不是問題。

也可以呼叫 `execlp(2)` 半途換成其他程式執行。

```
(gdb) p execlp("ls", "ls", "/", 0)

Program exited normally.
```

```
The program being debugged stopped while in a function called from GDB.
When the function (execlp) is done executing, GDB will silently
stop (instead of continuing to evaluate the expression containing
the function call).
(gdb)
```

應該會執行 "ls /" 指令。最後的訊息是 GDB 警告控制權沒有返回預期的地方。再舉一些例子，假如目標 process 是有跟 libdl 連結的程式，那麼就能呼叫 dlopen(3) 連結其他共享 object 並使用。如果沒有與 libdl 連結的話，就不能使用 dlopen(3)，而 GDB 也沒有連結器的功能，可能就必須依賴 [HACK #86] 的 livepatch 這類具備連結器功能的程式了。

應用例：外部指令 cd

為什麼 cd 不是外部指令而是 shell 內建指令，是 Unix 經常被問到的問題之一。答案應該是「其他程式無法改變目前程式的工作目錄」，不過這裡作為本 hack 的應用範例，硬是實作外部指令的 cd 看看。

以 GDB 搭配 script 語言即可簡單實作完成。

```
#!/usr/bin/ruby
# Usage:
#   ext-cd directory

require 'tempfile'

dir = ARGV.shift

t = Tempfile.new("ext-cd.gdb")
t.puts <<"End"
attach #{Process.ppid}
call chdir(#{dir.dump})
End
t.close

# shut up gdb.
STDIN.reopen("/dev/null")
STDOUT.reopen("/dev/null")
STDERR.reopen("/dev/null")

system("gdb", "-batch", "-n", "-x", t.path)
```

執行結果範例如下。

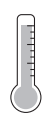
```
$ /bin/pwd
/tmp
$ ./ext-cd /
$ /bin/pwd
/
```

題外話，Solaris 有 `/bin/cd`，不過這個指令的行為不是像 `ext-cd` 這樣。單純只是在 `cd` process 裡面 `chdir` 而已。這是因為 POSIX 標準要求所有指令都要能 exec(2)，理由是在 POSIX 訂定時的議論過程中，不管是作成外部指令之後多麼沒用的指令，都還是有人提出用得上的例子，導致一直得不到結論，結果是決定所有指令都要準備外部的版本。

小結

本 hack 介紹了幾個積極活用 GDB 擁有的 `ptrace(2)` 系統呼叫功能、操作執行中的 process 的例子。使用 GDB 的話，Ruby 之類的 scripting 語言也能簡單寫出這類程式。

— *Takeshi Yaegashi, Akira Tanaka*



HACK #91 使用硬體的除錯功能

解說 x86 擁有的硬體除錯支援功能使用法。

處理器之中有支援除錯功能的硬體，比如說 x86 架構準備了八個除錯 register (DR0 ～ DR7)。本 hack 將解說 Linux 的 process 活用這些功能的方法。

x86 的除錯 Registers

x86 的除錯功能在「IA-32 Intel Architecture Software Developer's Manual, Volume 3B」的「CHAPTER 18 Debugging and Performance Monitoring」有完整的解說。

簡單來說，DR0-DR3 這四個 registers 指定的記憶體線性位址範圍，若是處理器有讀寫的話，會引發 `INT 1` 的除錯例外。剩下的四個 registers 是保留用途、或是控制除錯功能用的。

Linux/x86 可以為各 process 獨立設定除錯 registers 的內容，引發 INT 1 的 process 會收到 `SIGTRAP`。GDB 就是利用這個功能實現硬體監看點。

改寫自己的除錯 Registers

除錯 registers 的讀寫動作只允許 kernel mode 進行，所以 process 想改寫除錯 registers 的時候必須使用 ptrace 系統呼叫。瀏覽 kernel header 檔 asm-i386/user.h 可發現 struct user 定義了 int u_debugreg[8]; 的成員。

GDB 讀寫目標 process 的除錯 registers 的時候，這樣就可以了，但在 process 想讀寫自己的除錯 registers 的時候，process 沒辦法直接對自己 ptrace，因此要使用 fork 在子 process 裡面 ptrace。

以下是設定除錯 registers 內容的函式實作範例。

```
#include <asm/user.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/ptrace.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

void
set_debugregs(unsigned long *v)
{
  if (!fork()) {
    int i, *p = ((struct user *)0)->u_debugreg;
    pid_t ppid = getppid();
    ptrace(PTRACE_ATTACH, ppid, NULL, NULL);
    waitpid(ppid, NULL, 0);
    for (i = 0; i < 8; i++, p++, v++) {
      if (i == 4 || i == 5)
        continue;
      if (ptrace(PTRACE_POKEUSER, ppid, p, *v) < 0)
        fprintf(stderr,
                "ptrace failed: dr%d = %08lx\n", i, *v);
    }
    ptrace(PTRACE_DETACH, ppid, NULL, NULL);
    exit(0);
  }
  wait(NULL);
}
```

像這種設定新值的函式寫起來比較簡單，不過要讀取現有值的時候，要得到子 process 以 ptrace(PTRACE_PEEKUSER) 讀到的結果，就必須以某些方式進行跨 process 的通訊了。

除錯 Registers 的活用例

用上面的 set_debugregs() 函式，寫簡單的測試程式實驗看看吧。

```
#include <signal.h>
#include <asm/ucontext.h>

int tmp, data0, data1;
void func(void) {}

static unsigned long regs[] = {
  (unsigned long)&data0,
  (unsigned long)&data1,
  (unsigned long)func,
  0, /* 未使用 */
  0, /* 保留 */
  0, /* 保留 */
  0, /* 無法讀取 */
  0x00fd013f, /* Trap 條件
                DR0: 寫入, DR1: 讀寫, DR2: 執行, DR3: 未使用 */
};

#define TRY(x) do { fputs("Trying " #x "\n", stderr); x; } while (0)

static void
trap_handler(int n, siginfo_t *si, struct ucontext *uc)
{
  fprintf(stderr, " Trapped at 0x%08lx\n", uc->uc_mcontext.eip);
  /* 碰到 DR2 的時候, 關閉中斷點, 以避免無窮迴圈 */
  if (uc->uc_mcontext.eip == regs[2]) {
    regs[7] = 0;
    set_debugregs(regs);
  }
}

int
main(void)
{
  struct sigaction sa = {
    .sa_sigaction = (void *)trap_handler,
    .sa_flags = SA_RESTART | SA_SIGINFO,
  };

  sigemptyset(&sa.sa_mask);
  sigaction(SIGTRAP, &sa, NULL);
```

```
    set_debugregs(regs);

    TRY(tmp = data0);
    TRY(tmp = data1);
    TRY(data0 = 1);
    TRY(data1 = 1);
    TRY(func());

    return 0;
}
```

編譯並執行的狀況如下所示。

```
% gcc -g -Wall -O2 debugregs.c
% ./a.out
Trying tmp = data0
Trying tmp = data1
  Trapped at 0x080487de
Trying data0 = 1
  Trapped at 0x08048818
Trying data1 = 1
  Trapped at 0x08048844
Trying func()
  Trapped at 0x080486b0
```

像這樣使用 x86 的除錯 registers，即可感知一般 page 單位的記憶體保護機制不可能辦到的，微細記憶體範圍的操作情形。

若在 GDB 之內執行這個程式，引發 SIGTRAP 的時候，控制權會轉移到 GDB。與「[HACK #92] 在 C 程式中設定中斷點」介紹的技巧一樣，想在程式裡動態設定監看點的時候可以使用。

小結

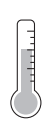
在此說明了 x86 擁有的硬體除錯功能的使用方式。可以監視的最多只有 4 words，是很少量的記憶體範圍，但是善加使用的話，仍然能幫上不少忙。

參考文獻

- 《IA-32 Intel(R) Architecture Software Developer's Manual, Volume 3B》的「CHAPTER 18 Debugging and Performance Monitoring」(http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html)

- 《図解 32 ビットマイクロコンピュータ 80486 の使い方》(圖解 32-bit 微電腦 80486 的使用法 ; 暫譯) 的「13 章 デバッグサポート (除錯支援)」(W.B. スルヤント著、オーム社)

— *Takeshi Yaegashi*



HACK #92 在 C 程式中設定中斷點

介紹 C 程式除錯時設定中斷點的方法。

C 程式除錯的時候，GDB 之類的除錯器能幫上不少忙。通常，中斷點是在除錯器之內設定的，本 hack 則要介紹在除錯對象的 C 程式之中設定中斷點的方法。

Linux 只要先 `#include <signal.h>`，之後在任何地方插入下列指令就 OK 。

```
raise(SIGTRAP);
```

使用 `raise()` 函式引發 `SIGTRAP` signal。或者是限制在 x86 環境下的話，下面的寫法也 OK。

```
__asm__ __volatile__("int3");
```

這邊為了引發 `SIGTRAP` 而嵌入了 `int3` (0xcc) 指令。這個方法跟呼叫 `raise()` 的狀況不一樣，不會呼叫函式，有不會擾亂 call stack 的好處。GDB 在設定軟體中斷點的時候，也是在對應位置寫入 `int3` 指令，作法很類似 (不過 GDB 在寫入 `int3` 之前需要把原本的指令保存下來)。

把內含上面這段程式碼的程式編譯、然後以 GDB 執行 binary，就會在執行到這個指令的時候中斷執行，將控制權轉到 GDB。

檢查是否透過除錯器執行

程式在除錯器上執行的時候，`SIGTRAP` 會被除錯器處理，利用這個性質，即可檢查程式是否透過除錯器執行。以下的程式透過除錯器執行時會顯示 `being debugged`，反之則會顯示 `not being debugged`。

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

int being_debugged = 1;
```

```
void signal_handler(int signum) {
  being_debugged = 0;
}

int main() {
  signal(SIGTRAP, signal_handler);
  __asm__ __volatile__("int3");
  if (being_debugged) {
    printf("being debugged\n");
  } else {
    printf("not being debugged\n");
  }
  return(0);
}
```

執行結果如下所示。

```
% gcc test.c
% ./a.out
not being debugged
% gdb ./a.out
... 略 ...
(gdb) run
Starting program: /tmp/a.out
Program received signal SIGTRAP, Trace/breakpoint trap.
0x080483a9 in main ()
(gdb) c
Continuing.
being debugged
```

這是惡意程式防止自己透過除錯器執行時使用的技巧之一。詳情請參閱文獻《Security Warrior》。

小結

想進行上述動作的情形應該不多，但是想在執行過程之中的特定時機中斷執行、用除錯器檢查狀態，或是想檢查 macro 之類難以用除錯器設定中斷點的地方時，這就應該是很方便的作法。

參考文獻

- 《Security Warrior》(Cyrus Peikari, Anton Chuvakin 著 , O'Reilly 出版)

— *Satoru Takabayashi*