系統程式設計 System Programming 2021

Hw01 使用 GDB 進行偵錯

電機四 406415089 宋 O 天

<前置作業>

先看看 makefile 裡面有沒有下-g 參數，確認有

```
SHELL = /bin/bash
CC = gcc
CFLAGS = -g
SRC = $(wildcard *.c)
EXE = $(patsubst %.c, %, $(SRC))
```

/bin ：這是放例如：ls, mv, rm, mkdir, rmdir, gzip, tar, telnet, 及 ftp 等等常用的執行檔的地方而通常這個檔案的內容與 /usr/bin 是一樣的（有時候甚至會使用連結檔哩）

輸入 `make all` 指令之後發現 bin/bash 裡沒有 gcc

```
nash@SleepyCat:~/Desktop/system-programming/ch02$ make all
gcc -g  rdtsc.c -o rdtsc
/bin/bash: gcc: command not found
make: *** [makefile:10: rdtsc] Error 127
```

利用 `ls /usr/bin | grep gcc` 指令確實找不到 gcc

```
nash@SleepyCat:~/Desktop/system-programming/ch02$ ls /usr/bin | grep gcc
nash@SleepyCat:~/Desktop/system-programming/ch02$
```

<Code>

安裝完 gcc 後就可以 make all，並可以利用 `gcc -v ,gdb -v` 查看 gcc、gdb 版本

接著用 vim 開啟 rdtsc.c 看一下，看到如下內容：

rdtsc 將暫存器 eax 裡的值放入 lo，暫存器 edx 的值放入 hi，回傳 uint64_t 的資料型別

```
extern __inline__ uint64_t rdtscp(void)
{
    uint32_t lo, hi;
    // take time stamp counter, rdtscp does serialize by itself, and is much cheaper than using CPUID
    __asm__ __volatile__("rdtscp":"=a"(lo), "=d"(hi));
    return ((uint64_t) lo) | (((uint64_t) hi) << 32);
}
```

rdtsc/rdtscp 是 CPU 指令。用來讀 tsc 暫存器，即 time stamp counter 的值

暫存器在每個時鐘信號到來時加 1，數值的遞增就和 CPU 的主頻相關，主頻為 1MHz 的處理器暫存器每秒就遞增 1,000,000 次。低 32 位存放在 eax 暫存器中，高 32 位存放在 edx 暫存器中

EAX、ECX、EDX、EBX：為 ax,bx,cx,dx 的延伸，各為 32 位元 ... eax, ebx, ecx, edx, esi, edi, ebp, esp 等都是 x86 組合語言中 CPU 上的通用暫存器。

inline 所宣告的 function 並不會有程式本體，會直接展開到呼叫他的地方

extern inline 表示該函式是已宣告過的了，extern 對函式的影響僅僅把函式的隱藏屬性顯式化。

extern 對於非函式的物件是有用的，因為物件宣告時會帶來記憶體的分配，而用 extern 就表示該物件已經宣告過了，不用再分配記憶體。

接著來看 main 函式裡最後顯示的東西：

最後他調用了 system()這一個指令，因為他非常容易出錯，所以前面會加上 assert

```
printf("開始 %lu, 結束 %lu\n",cycles1, cycles2);
printf("rdtscp: tmp++ consumes %lu cycles!\n", cycles2-cycles1);
printf("開始 %lu, 結束 %lu\n",ts_to_long(ts1), ts_to_long(ts2));
printf("clock_gettime: tmp++ consumes %lu nanoseconds!\n", ts_to_long(ts2)-ts_to_long(ts1));
assert(system("cat /proc/cpuinfo | grep 'cpu MHz' | head -1")>=0);
```

/proc :存放系統核心與執行程序的一些資訊

當我下指令 `ls -l | grep sh` 之後，我發現 ubuntu 預設的 shell 是 dash 而不是 bash

```
lrwxrwxrwx 1 root root           4  三   13 22:43 sh -> dash
```

system() executes a command specified in command by calling /bin/sh -c command,
and returns after the command has been completed.
During execution of the command,
SIGCHLD will be blocked, and SIGINT and SIGQUIT will be ignored.

system()函數調用/bin/sh 來執行參數指定的命令，
而/bin/sh 一般是一個軟連接，指向某個具體的 shell，比如常見的 bash
而-c 選項是告訴 shell 從字符串 command 中讀取命令
command 執行期間：
SIGCHLD 是被阻塞的，好比在說” hi 內核，現在不要給我送 SIGCHLD 信號，等我忙完。”
SIGINT 和 SIGQUIT 是被忽略的，意思是進程收到這兩個信號後沒有任何動作。

<Debug 部分>

如同老師上課提到的，我只是把流程跑一遍

要求 1 · 2 · 3 · 4

```
(gdb) r
Starting program: /home/nash/Desktop/system-programming/ch02/rdtsc

Breakpoint 1, main (argc=0, argv=0x0) at rdtsc.c:28
28      {
(gdb) n
29          int tmp=0;
```

接著利用 step 指令跳進去定義的 rdtscp()函式裡，並利用 print 來印出變數

```
36          cycles1 = rdtscp();
(gdb) s
rdtscp () at rdtsc.c:16
16      {
(gdb) s
19          __asm__ __volatile__("rdtscp":"=a"(lo), "=d"(hi));
(gdb) s
20          return ((uint64_t) lo) | (((uint64_t) hi) << 32);
(gdb) s
21      }
(gdb) p lo
$2 = 354361331
(gdb) p hi
$3 = 8290
```

要求 5 · 6 · 7

#0 表示 stack 的頂端，可以發現下指令 bt，配合 up 和 down 之後可以觀察變數值的不同來理解在不同 frame 的時候，變數的狀態到底是甚麼。

```
(gdb) bt
#0  rdtscp () at rdtsc.c:21
#1  0x000055555555527d in main (argc=1, argv=0x7fffffffdfa8) at rdtsc.c:36
(gdb) p tmp
No symbol "tmp" in current context.
(gdb) p lo
$4 = 354361331
(gdb) up
#1  0x000055555555527d in main (argc=1, argv=0x7fffffffdfa8) at rdtsc.c:36
36          cycles1 = rdtscp();
(gdb) p tmp
$5 = 0
(gdb) p lo
No symbol "lo" in current context.
(gdb) up
Initial frame selected; you cannot go up.
(gdb) down
#0  rdtscp () at rdtsc.c:21
21      }
(gdb) p tmp
No symbol "tmp" in current context.
(gdb) p lo
$6 = 354361331
```

下指令 awatch tmp 表示監看 tmp 值的變化，並利用 continue 在中斷後繼續執行程式，最後真的攔截到 tmp 變數的變化，並顯示出 old value and new value。

```
Breakpoint 4, main (argc=1, argv=0x7fffffffdfa8) at rdtsc.c:36
36          cycles1 = rdtscp();
(gdb) c
Continuing.

Hardware access (read/write) watchpoint 5: tmp

Old value = 0
New value = 1
main (argc=1, argv=0x7fffffffdfa8) at rdtsc.c:38
38          cycles2 = rdtscp();
```

照著老師的方式生成錯的檔案，出現 segmentation fault

```
 33      //====wrong code is here====
 34      int *ptr;
 35      printf("%d\n",*ptr);
 36      //====wrong code is here====
Program received signal SIGSEGV, Segmentation fault.
0x0000555555555264 in main (argc=1, argv=0x7fffffffdfa8) at rdtsc.c:35
35              printf("%d\n",*ptr);
(gdb) n
n
Program terminated with signal SIGSEGV, Segmentation fault.
```

SIGSEGV 是當一個行程執行了一個無效的記憶體參照，或發生段錯誤時傳送給它的訊號。

SIGSEGV 的符號常數在標頭檔 signal.h 中定義

記憶體段錯誤，也稱存取權限衝突（access violation）
會出現在當程式企圖存取 CPU 無法定址的記憶體區段時。
當錯誤發生時，硬體會通知作業系統產生了記憶體存取權限衝突的狀況，作業系統通常會產生核心轉儲（core dump）以方便程式員進行除錯。通常該錯誤是由於調用一個位址，而該位址為 NULL 所造成的

參考資料

內聯函數：static inline 和 extern inline 的含義
http://read01.com/MRjLO.html
認識 Linux 檔案屬性及檔案配置
https://linux.vbird.org/linux_basic/redhat6.1/linux_05file.php
如何精确测量一段代码的执行时间
https://www.0xffffff.org/2015/12/06/37-How-to-benchmark-code-execution-times/
【C/C++】Linux 下使用 system()函數一定要謹慎
http://my.oschina.net/kangchunhui/blog/161420