第19回

標準的デバッガGDBのすべて

岸 哲夫

メモリ領域のデバッグ補助ツールvalgrind

本連載ではgdbを使ったデバッグ手法について解説しているが，gdb以外にも便利なデバッグ・ツールが存在する．今回は，メモリ領域のデバッグ補助ツールvalgrindを使ってメモリ・リークなどの検出を行う．　（編集部）

内部のデータ，またプログラムの動作を解析することは，デバッグの助けになります．デバッグ時にすべてのソース・コードを通過させることで予期せぬ動きを阻止し，実行時異常終了を防止できます．

今回はgdbと同じくデバッグ補助ツールであるvalgrindを使ったデバッグの手法について解説します．

1. スタックについてのまとめ

プログラムが予期せぬ停止をした場合，どんな処理をして，どこで止まったのかがわかれば，デバッグが楽になると思います．

組み込み機器の場合，ハードウェア的な動きで推測可能なものもありますが，それでもデバッグは重要です．どの関数が呼び出されて，どのデータが加工されて，どの関数で停止したかを理解するだけで，その後の対処がスムーズになるでしょう．

実行中のプログラムが関数をコールするとき，プログラムがどのようにコールされたのかという情報が，スタック・フレームと呼ばれるデータ・ブロックに保存されます．

スタック・フレームは，コール時の引き数とコールされた関数のローカル変数の内容を含んでいます．すべてのスタック・フレームはスタックと呼ばれるメモリ領域に配置されます．

● スタック・フレームの表示

図1にgdbでスタック・フレーム情報を表示した場合の実行結果を示します．⑥のbtコマンドでは，ここまでに呼び出された関数の履歴が表示されています．

このように，関数が呼ばれるたびにスタック領域が使われるので，当然増加します．再帰呼び出しを何度も行うと，場合によってはスタックを限界まで消費してオーバフローします．

GCC（C++も含む）の場合，自動変数はスタック領域に，newやmalloc()で変数を確保するとヒープ領域に確保されます．細かい指定もできますが，通常は各実行プロセスごとに実メモリを取れるだけ仮想メモリを取得します．そこ

```
[root@localhost ~]# gdb test1          ← ① gdb起動
GNU gdb Red Hat Linux (6.5-8.fc6rh)
Copyright (C) 2006 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License,
and you are
welcome to change it and/or distribute copies of it under certain
conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty"
for details.
This GDB was configured as "i386-redhat-linux-gnu"...Using host
libthread_db library "/lib/libthread_db.so.1".

(gdb) list                             ← ② ソース・リストの表示
1       #include <stdio.h>
2       #include <stdlib.h>
3       static char *wk;
4       void test(void);
5       void test2(void);
6       void test3(void);
7       void test4(void);
8       int main(void)
9       {
10          test();
11      }
12      void test(void)
13      {
14          wk = (char*)malloc(10);
15          test2();
16      }
17      void test2(void)
18      {
19          test3();
20      }
21      void test3(void)
22      {
23              test4();
24      }
25      void test4(void)
26      {
27              free(wk);
28      }
(gdb) b 27                             ← ③ ブレーク・ポイントの設定
Breakpoint 1 at 0x80483e1: file test1.c, line 27.
(gdb) run                              ← ④ 実行
Starting program: /root/test1

Breakpoint 1, test4 () at test1.c:27
27              free(wk);
(gdb) frame                            ← ⑤ スタック・フレームの情報
#0  test4 () at test1.c:27
27              free(wk);
 (gdb) bt                              ← ⑥ バック・トレースの情報
#0  test4 () at test1.c:27
#1  0x080483d9 in test3 () at test1.c:23
#2  0x080483cc in test2 () at test1.c:19
#3  0x080483bf in test () at test1.c:15
#4  0x0804839a in main () at test1.c:10
(gdb)
```

図1　スタック・フレーム情報

KEYWORD —— GDB，デバッガ，クロス・コンパイル，コンパイル・デバッグ，PowerPC，ARM