

## 4章

# 実践アプリケーションデバッグ

## Hack #26-32

この章では、ユーザアプリケーションの実践的なデバッグ方法について記しています。スタックオーバーフローによるセグメンテーションフォルト (SIGSEGV)、バックトレースが正しく表示されない、配列の不正アクセスによるスタック破壊、ウォッチポイントを活用した不正メモリアクセスの検知、`malloc()/free()` での障害、アプリケーションのストールなどさまざまな事例によるデバッグ方法を記しています。



### HACK #26 SIGSEGV でアプリケーションが異常終了した スタックオーバーフローによるセグメンテーションフォルトのデバッグ

アプリケーションプログラムが不正なメモリアクセスなどをした場合、SIGSEGV という例外を発生し異常終了します。SIGSEGV が発生する場合は、(1) NULL ポインタによるアクセス、(2) ポインタ破壊などによる不正アドレスへのアクセス、(3) スタックオーバーフローなどにより、確保したアドレス領域を越えてのアクセス、などがあります。

ここでは、スタックオーバーフローにより SIGSEGV が発生した場合のデバッグ方法について解説します。

以下にセグメンテーションフォルトを発生させる例を示します。

```
$ ruby1.8 -e 'eval("1+" * 100000 + "1")'
Segmentation fault
```

```
"1+" "1+" "1+" "1+" ... "1+" "1+"
100000 連結

"1+1+1+1+...1+1"
↑をRubyのプログラムとして eval (評価する)
```

図 4-1 eval("1+" \* 100000 + "1") プログラム



`eval("1" * 100000 + "1")`というのは、"1+"という文字列を100000個連結したものに、"1"という文字列を連結した文字列をRubyのプログラムとして評価 (eval) するプログラムです。

コアファイルを生成するように設定します。

```
$ ulimit -c unlimited
$ ruby1.8 -e 'eval("1" * 100000 + "1")'
Segmentation fault (core dumped)
$ ls core
core
```

アプリケーションプログラムがコアファイルを生成したので、デバッガで分析してみましょう。

```
$ gdb ruby1.8 core
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
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"...
(no debugging symbols found)
中略
Loaded symbols for /lib/ld-linux.so.2
(no debugging symbols found)
Core was generated by `ruby1.8 -e eval("1" * 100000 + "1")'.
Program terminated with signal 11, Segmentation fault.
[New process 24488]
#0  0xb7e22cb7 in ?? () from /usr/lib/libruby1.8.so.1.8
```

システムにインストールされているアプリケーション (この場合は `ruby1.8`) にはデバッグ情報が付加されていないので、シンボル情報が表示できません。しかし、スタックフレームの情報などから、ある程度原因を推定できる場合があります。

`bt` (backtrace) コマンドでスタックフレームを取得してみます。そうすると、大量にスタックフレームが表示されました。これは再帰的に関数が呼ばれていることを示しています。そこで最初のいくつかだけを表示することにします。この場合 10 個取得することにします。

bt <数字> という形式です。

```
(gdb) bt 10
#0 0xb7e22cb7 in ?? () from /usr/lib/libruby1.8.so.1.8
#1 0xb7e22f3a in ?? () from /usr/lib/libruby1.8.so.1.8
#2 0xb7e22f3a in ?? () from /usr/lib/libruby1.8.so.1.8
#3 0xb7e22f3a in ?? () from /usr/lib/libruby1.8.so.1.8
#4 0xb7e22f3a in ?? () from /usr/lib/libruby1.8.so.1.8
#5 0xb7e22f3a in ?? () from /usr/lib/libruby1.8.so.1.8
#6 0xb7e22f3a in ?? () from /usr/lib/libruby1.8.so.1.8
#7 0xb7e22f3a in ?? () from /usr/lib/libruby1.8.so.1.8
#8 0xb7e22f3a in ?? () from /usr/lib/libruby1.8.so.1.8
#9 0xb7e22f3a in ?? () from /usr/lib/libruby1.8.so.1.8
```

スタックフレームを眺めてみると、0xb7e22f3a というアドレスから何度も呼ばれていることがわかります。これは再帰的に関数が呼ばれスタックオーバーフローでアプリケーションが異常終了したことが疑われます。

## ソースコードレベルのデバッグ

それでは gdb でソースコードを追跡してみましょう。あらかじめ、gcc の -g オプション付きでビルドしたアプリケーション (ruby) を gdb で起動します。

```
(gdb) run -e 'eval("1" * 100000 + "1")'
Starting program: /home/hyoshiok/work/ruby_trunk/ruby/ruby -e 'eval("1" * 100000 + "1")'
[Thread debugging using libthread_db enabled]
[New Thread 0xb7d3d6b0 (LWP 24646)]
[New Thread 0xb7f24b90 (LWP 24649)]

Program received signal SIGSEGV, Segmentation fault.
[Switching to Thread 0xb7d3d6b0 (LWP 24646)]
iseq_compile_each (iseq=0x8efa9c0, ret=0xbf64b138, node=0x931dd74, popped=0) at compile.c:2883
```

アプリケーションが SIGSEGV を引き起こしました。gdb の場合、シグナルを受け取るとあらかじめ定義された動作を起こします。SIGSEGV の場合は当該の箇所です自動的に停止してくれます。

info signal で gdb が処理するシグナルの一覧が表示できます。

ソースコードは下記のところです。emacs で gdb を起動すれば自動的に表示してくれますので便利です。

```
/**
 * compile each node
 *
 * self: InstructionSequence
 * node: Ruby compiled node
 * popped: This node will be popped
 */
static int
iseq_compile_each(rb_iseq_t *iseq, LINK_ANCHOR *ret, NODE * node, int popped)
{ /* ここで停止する */
  enum node_type type;

  if (node == 0) {
    if (!popped) {
      debugs("node: NODE_NIL(implicit)\n");
      ADD_INSN(ret, iseq->compile_data->last_line, putnil);
    }
    return COMPILER_OK;
  }
}
```

スタックフレームを `bt` コマンドで取ります。せいぜい5つまで取れば十分でしょう。

```
(gdb) bt 5
#0  iseq_compile_each (iseq=0x8efa9c0, ret=0xbf64b138, node=0x931dd74, popped=0) at compile.c:2883
#1  0x0811154d in iseq_compile_each (iseq=0x8efa9c0, ret=0xbf64b278, node=0x931dd38, popped=0) at
compile.c:3954
#2  0x0811154d in iseq_compile_each (iseq=0x8efa9c0, ret=0xbf64b3b8, node=0x931dcfc, popped=0) at
compile.c:3954
#3  0x0811154d in iseq_compile_each (iseq=0x8efa9c0, ret=0xbf64b4f8, node=0x931dcc0, popped=0) at
compile.c:3954
#4  0x0811154d in iseq_compile_each (iseq=0x8efa9c0, ret=0xbf64b638, node=0x931dc84, popped=0) at
compile.c:3954
(More stack frames follow...)
```

そうすると、`0x0811154d` というアドレスから何度も呼ばれていることがわかります。スタックフレームを1つ上へのぼってみましょう。 `up` コマンドを利用します。

```
(gdb) up
#1  0x0811154d in iseq_compile_each (iseq=0x8efa9c0, ret=0xbf64b278, node=0x931dd38, popped=0) at
compile.c:3954
```

当該アドレスのソースコードは以下の `COMPILE()` のところです。

```
        else {
            ADD_LABEL(ret, label);
        }
        break;
    }
}
#endif

/* reciever */
if (type == NODE_CALL) {
    COMPILE(recv, "recv", node->nd_recv); /* ここから呼んでいる */
}
else if (type == NODE_FCALL || type == NODE_VCALL) {
    ADD_CALL_RECEIVER(recv, nd_line(node));
}

/* args */
if (nd_type(node) != NODE_VCALL) {
    argc = setup_args(iseq, args, node->nd_args, &flag);
}
```

ソースコードを見ると、`COMPILE` は下記のようにマクロの定義になっていて `iseq_compile_each()` 関数を再帰的に呼んでいることがわかります。

```
/* compile node */
#define COMPILE(anchor, desc, node) \
    (debug_compile("== " desc "\n", \
        iseq_compile_each(iseq, anchor, node, 0)))
```

ソースコードを分析した結果、何度も再帰的に関数を呼んだためスタックオーバーフローを引き起こしたということがわかります。

## スタックオーバーフローで SIGSEGV への対応

一般的に言ってシグナルを捕獲したら、そのためのシグナルハンドラを用意して、何かしらの作業をすればいいわけです。しかしスタックオーバーフローで SIGSEGV を発生させた場合は、スタック領域があふれ不正アクセスになったため、シグナルハンドラを起動するスタックすら確保できないので、そのままでは対処できません。そのため、スタックオーバーフローを捕捉するために代替シグナルスタックを設定する必要があります。それには

sigaltstack(2) を使います。

man page に掲載されている例は以下のとおりです。

```
stack_t ss;

ss.ss_sp = malloc(SIGSTKSZ);
if (ss.ss_sp == NULL)
    /* ハンドルエラー */;
ss.ss_size = SIGSTKSZ;
ss.ss_flags = 0;
if (sigaltstack(&ss, NULL) == -1)
    /* ハンドルエラー */;
```

さて、それを参考に以下のようなパッチを作成してみました。

```
$ svn diff signal.c
Index: signal.c
=====
--- signal.c      (リビジョン 20086)
+++ signal.c      (作業コピー)
@@ -47,6 +47,10 @@
 # define NSIG (_SIGMAX + 1)    /* For QNX */
 #endif

+#ifdef SIGSEGV
+static int is_altstack_defined = 0;
+#endif
+
static const struct signals {
    const char *signm;
    int signo;
@@ -410,6 +414,28 @@
 typedef RETSIGTYPE (*sighandler_t)(int);

#ifdef POSIX_SIGNAL
+#define ALT_STACK_SIZE (4*1024)
+#ifdef SIGSEGV
+/* alternate stack for SIGSEGV */
+static void register_sigaltstack() {
+    stack_t newSS, oldSS;
```

```
+
+ if(is_altstack_defined)
+   return;
+
+ newSS.ss_sp = malloc(ALT_STACK_SIZE);
+ if(newSS.ss_sp == NULL)
+   /* should handle error */
+   rb_bug("register_sigaltstack. malloc error\n");
+ newSS.ss_size = ALT_STACK_SIZE;
+ newSS.ss_flags = 0;
+
+ if (sigaltstack(&newSS, &oldSS) < 0)
+   rb_bug("register_sigaltstack. error\n");
+ is_altstack_defined = 1;
+}
+
+static sighandler_t
ruby_signal(int signum, sighandler_t handler)
{
@@ -432,7 +458,12 @@
    if (signum == SIGCHLD && handler == SIG_IGN)
        sigact.sa_flags |= SA_NOCLDWAIT;
    #endif
-   sigaction(signum, &sigact, &old);
+
+   #ifdef SA_ONSTACK
+   if (signum == SIGSEGV)
+       sigact.sa_flags |= SA_ONSTACK;
+   #endif
+   if (sigaction(signum, &sigact, &old) < 0)
+       rb_bug("sigaction error.\n");
+   return old.sa_handler;
+}

@@ -663,6 +694,7 @@
#ifdef SIGSEGV
    case SIGSEGV:
        func = sigsegv;
+       register_sigaltstack();
        break;
```

```
#endif
#ifdef SIGPIPE
@@ -1070,6 +1102,7 @@
    install_sighandler(SIGBUS, sigbus);
#endif
#ifdef SIGSEGV
+ register_sigaltstack();
    install_sighandler(SIGSEGV, sigsegv);
#endif
}
```

このパッチを当てたプログラムを実行した結果は下記のとおりです。

```
$ ./ruby -e 'eval("1+" * 100000 + "1")'
-e:1: [BUG] Segmentation fault
ruby 1.9.0 (2008-11-01 revision 20086) [i686-linux]

-- control frame -----
c:0004 p:---- s:0010 b:0010 l:000009 d:000009 CFUNC :eval
c:0003 p:0017 s:0006 b:0006 l:000005 d:000005 TOP -e:1
c:0002 p:---- s:0004 b:0004 l:000003 d:000003 FINISH :inherited
c:0001 p:0000 s:0002 b:0002 l:000001 d:000001 TOP <dummy toplevel>:17
-----
-e:1:in `eval': stack level too deep (SystemStackError)
    from -e:1:in `<main>'
```

いずれにせよセグメンテーションフォルト (SIGSEGV) で異常終了するのですが、何も追加情報を出力しないで終了するのと違って、異常終了のヒントを出してくれるので、アプリケーションをデバッグするときの助けになります。

なお、<http://redmine.ruby-lang.org/repositories/revision/ruby-19?rev=20293> において、このパッチを元に修正が ruby に加えられました。

## まとめ

スタックオーバーフローのため SIGSEGV で異常終了した場合のデバッグ方法について記しました。

## 参考文献

- 『BINARY HACKS』の「sigaltstack でスタックオーバーフローに対処する」[HACK

#76] (pp. 291-300)

— Hiro Yoshioka

HACK  
#27

## バックトレースが正しく表示されない

マルチスレッドアプリケーションで、スレッド間競合によりスタックが破壊されたケースを題材に説明します。

### 概要

スタック破壊によって、問題の解析が困難となることがあります。特に、バックトレース情報が得られなくなることで、問題現象に至るルートを追いかけていくことになります。また、スタック破壊が存在することで、バックトレース情報は完全ではないと言えます。デバッグでのバックトレースは万能ではないことを覚えておきましょう。

### 問題内容

とあるスレッド間通信を行うプログラムにバグがあり、`core` が生成されました。

### バックトレース確認

デバッグで解析を行う際に、とりあえずバックトレース、というのが定石です。しかし、再現プログラム実行によって生成された `core` ファイルにおいて、バックトレースを見てみましたが、何がコールされているのかさっぱりわかりません。`nanosleep()` を実行中に `SIGSEGV` となったようですが、`th_req()` から `nanosleep()` に至るルートはどうなっているのでしょうか？

```
(gdb) bt
#0 0x0000003b4869ac80 in nanosleep () from /lib64/libc.so.6
#1 0x000ee1c2000ee1c1 in ?? ()
#2 0x000ee1c4000ee1c3 in ?? ()
#3 0x000ee1c6000ee1c5 in ?? ()
#4 0x000ee1c8000ee1c7 in ?? ()
#5 0x000ee1ca000ee1c9 in ?? ()
#6 0x000ee1cb000ee1ca in ?? ()
#7 0x000ee1cd000ee1cc in ?? ()
#8 0x0000000000000002 in ?? ()
#9 0x0000000001877c90 in ?? ()
#10 0x000000004162f130 in ?? ()
#11 0x000000000400d02 in th_req (p=0x1877c90) at bug.c:167
```