Best way to invoke gdb from inside program to print its stacktrace?

Asked 14 years, 5 months ago Modified 6 years, 9 months ago Viewed 48k times

63

Using a function like this:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

void print_trace() {
    char pid_buf[30];
    sprintf(pid_buf, "--pid=%d", getpid());
    char name_buf[512];
    name_buf[readlink("/proc/self/exe", name_buf, 511)]=0;
    int child_pid = fork();
    if (!child_pid) {
        dup2(2,1); // redirect output to stderr
        fprintf(stdout,"stack trace for %s pid=%s\n",name_buf,pid_buf);
        execlp("gdb", "gdb", "--batch", "-n", "-ex", "thread", "-ex", "bt",
name_buf, pid_buf, NULL);
        abort(); /* If gdb failed to start */
    } else {
        waitpid(child_pid,NULL,0);
    }
}
```

I see the details of print_trace in the output.

What are other ways to do it?

c linux gdb stack-trace

Share

Improve this question

Follow

edited Feb 27, 2018 at 21:02

community wiki
13 revs, 4 users 97%
Vi.

1 Is there a problem with it? Something it doesn't do? – Adam Shiemke Jun 30, 2010 at 17:37

1 @Adam Shiemke Some problems listed. May be gdb can be invoked in more proper way. May be I need something special to support multithreading. May be there's way to make it portable or there is special "libstacktrace.so". Discussion. – Vi. Jun 30, 2010 at 18:04

2 You can use the -ex option more than once. – Derek Ledbetter Jun 30, 2010 at 18:52

1 @Derek Ledbetter, OK, applying. – Vi. Jun 30, 2010 at 19:03

1 As a side note, there's a wrapper library for attaching GDB: libdebugme. – yugr Aug 25, 2018 at 12:33

4 Answers

Sorted by: Highest score (default)

102

You mentioned on my other answer (now deleted) that you also want to see line numbers. I'm not sure how to do that when invoking gdb from inside your application.

But I'm going to share with you a couple of ways to print a simple stacktrace with function names and their respective line numbers **without using gdb**. Most of them came from a *very nice* article from Linux Journal:

- **Method #1:**

> The first method is to disseminate it with print and log messages in order to pinpoint the execution path. In a complex program, this option can become cumbersome and tedious even if, with the help of some GCC-specific macros, it can be simplified a bit. Consider, for example, a debug macro such as:

```
#define TRACE_MSG fprintf(stderr, __FUNCTION__      \
                          "() [%s:%d] here I am\n", \
                          __FILE__, __LINE__)
```

> You can propagate this macro quickly throughout your program by cutting and pasting it. When you do not need it anymore, switch it off simply by defining it to no-op.

- **Method #2:** (It doesn't say anything about line numbers, but I do on method 4)

> A nicer way to get a stack backtrace, however, is to use some of the specific support functions provided by glibc. The key one is backtrace(), which navigates the stack frames from the calling point to the beginning of the program and provides an array of return addresses. You then can map each address to the body of a particular function in your code by having a look at the object file with the nm command. Or, you can do it a simpler way--use backtrace_symbols(). This function transforms a list of return addresses, as returned by backtrace(), into a list of strings, each containing the function name offset within the function and the return address. The list of strings is

> allocated from your heap space (as if you called malloc()), so you should free() it as soon as you are done with it.

I encourage you to read it since the page has source code examples. In order to convert an address to a function name you must compile your application with the **-rdynamic** option.

- **Method #3:** (A better way of doing method 2)

> An even more useful application for this technique is putting a stack backtrace inside a signal handler and having the latter catch all the "bad" signals your program can receive (SIGSEGV, SIGBUS, SIGILL, SIGFPE and the like). This way, if your program unfortunately crashes and you were not running it with a debugger, you can get a stack trace and know where the fault happened. This technique also can be used to understand where your program is looping in case it stops responding

An implementation of this technique is available here.

- **Method #4:**

A small improvement I've done on method #3 to print line numbers. This could be copied to work on method #2 also.

Basically, I followed a tip that uses **addr2line** to

> convert addresses into file names and line numbers.

The source code below prints line numbers for all local functions. If a function from another library is called, you might see a couple of `??:0` instead of file names.

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

void bt_sighandler(int sig, struct sigcontext ctx) {

  void *trace[16];
  char **messages = (char **)NULL;
  int i, trace_size = 0;

  if (sig == SIGSEGV)
    printf("Got signal %d, faulty address is %p, "
           "from %p\n", sig, ctx.cr2, ctx.eip);
  else
```

```
    printf("Got signal %d\n", sig);

  trace_size = backtrace(trace, 16);
  /* overwrite sigaction with caller's address */
  trace[1] = (void *)ctx.eip;
  messages = backtrace_symbols(trace, trace_size);
  /* skip first stack frame (points here) */
  printf("[bt] Execution path:\n");
  for (i=1; i<trace_size; ++i)
  {
    printf("[bt] #%d %s\n", i, messages[i]);

    /* find first occurence of '(' or ' ' in message[i] and assume
     * everything before that is the file name. (Don't go beyond 0 though
     * (string terminator)*/
    size_t p = 0;
    while(messages[i][p] != '(' && messages[i][p] != ' '
            && messages[i][p] != 0)
        ++p;

    char syscom[256];
    sprintf(syscom,"addr2line %p -e %.*s", trace[i], p, messages[i]);
        //last parameter is the file name of the symbol
    system(syscom);
  }

  exit(0);
}


int func_a(int a, char b) {

  char *p = (char *)0xdeadbeef;

  a = a + b;
  *p = 10;  /* CRASH here!! */

  return 2*a;
}


int func_b() {

  int res, a = 5;

  res = 5 + func_a(a, 't');

  return res;
}


int main() {

  /* Install our signal handler */
  struct sigaction sa;

  sa.sa_handler = (void *)bt_sighandler;
  sigemptyset(&sa.sa_mask);
  sa.sa_flags = SA_RESTART;

  sigaction(SIGSEGV, &sa, NULL);
  sigaction(SIGUSR1, &sa, NULL);
```

```
  /* ... add any other signal here */

  /* Do something */
  printf("%d\n", func_b());
}
```

This code should be compiled as: `gcc sighandler.c -o sighandler -rdynamic`

The program outputs:

```
Got signal 11, faulty address is 0xdeadbeef, from 0x8048975
[bt] Execution path:
[bt] #1 ./sighandler(func_a+0x1d) [0x8048975]
/home/karl/workspace/stacktrace/sighandler.c:44
[bt] #2 ./sighandler(func_b+0x20) [0x804899f]
/home/karl/workspace/stacktrace/sighandler.c:54
[bt] #3 ./sighandler(main+0x6c) [0x8048a16]
/home/karl/workspace/stacktrace/sighandler.c:74
[bt] #4 /lib/tls/i686/cmov/libc.so.6(__libc_start_main+0xe6) [0x3fdbd6]
??:0
[bt] #5 ./sighandler() [0x8048781]
??:0
```

Update 2012/04/28 for recent linux kernel versions, the above `sigaction` signature is obsolete. Also I improved it a bit by grabbing the executable name from this answer. Here is an up to date version:

```
char* exe = 0;

int initialiseExecutableName()
{
    char link[1024];
    exe = new char[1024];
    snprintf(link,sizeof link,"/proc/%d/exe",getpid());
    if(readlink(link,exe,sizeof link)==-1) {
        fprintf(stderr,"ERRORRRRR\n");
        exit(1);
    }
    printf("Executable name initialised: %s\n",exe);
}

const char* getExecutableName()
{
    if (exe == 0)
        initialiseExecutableName();
    return exe;
}

/* get REG_EIP from ucontext.h */
#define __USE_GNU
#include <ucontext.h>

void bt_sighandler(int sig, siginfo_t *info,
                   void *secret) {
```

```
  void *trace[16];
  char **messages = (char **)NULL;
  int i, trace_size = 0;
  ucontext_t *uc = (ucontext_t *)secret;

  /* Do something useful with siginfo_t */
  if (sig == SIGSEGV)
    printf("Got signal %d, faulty address is %p, "
           "from %p\n", sig, info->si_addr,
           uc->uc_mcontext.gregs[REG_EIP]);
  else
    printf("Got signal %d\n", sig);

  trace_size = backtrace(trace, 16);
  /* overwrite sigaction with caller's address */
  trace[1] = (void *) uc->uc_mcontext.gregs[REG_EIP];

  messages = backtrace_symbols(trace, trace_size);
  /* skip first stack frame (points here) */
  printf("[bt] Execution path:\n");
  for (i=1; i<trace_size; ++i)
  {
    printf("[bt] %s\n", messages[i]);

    /* find first occurence of '(' or ' ' in message[i] and assume
     * everything before that is the file name. (Don't go beyond 0 though
     * (string terminator)*/
    size_t p = 0;
    while(messages[i][p] != '(' && messages[i][p] != ' '
            && messages[i][p] != 0)
        ++p;

    char syscom[256];
    sprintf(syscom,"addr2line %p -e %.*s", trace[i] , p, messages[i] );
           //last parameter is the filename of the symbol
    system(syscom);

  }
  exit(0);
}
```

and initialise like this:

```
int main() {

  /* Install our signal handler */
  struct sigaction sa;

  sa.sa_sigaction = (void *)bt_sighandler;
  sigemptyset (&sa.sa_mask);
  sa.sa_flags = SA_RESTART | SA_SIGINFO;

  sigaction(SIGSEGV, &sa, NULL);
  sigaction(SIGUSR1, &sa, NULL);
  /* ... add any other signal here */

  /* Do something */
  printf("%d\n", func_b());
```

```
}
```

Share

Improve this answer

Follow

edited Jul 17, 2017 at 7:32

community wiki
8 revs, 4 users 75%
karlphillip

"Method #1" -> There is my other question on SO about how to "propogate" it automatically, but with no useful answers. – Vi. Jan 6, 2011 at 15:02

1 Methods #2 - #4 -> Already tried - it works: vi-server.org/vi/simple_sampling_profiler.html But backtrace/addr2line approach have limitation: 1. often addr2line cannot figure out the line (while gdb can), 2. gdb can iterate threads: "thread apply all bt". – Vi. Jan 6, 2011 at 15:03

@Vi This guy nailed it: stackoverflow.com/questions/4636456/stack-trace-for-c-using-gcc/… – karlphillip Jan 19, 2011 at 18:01

@karlphillip: I found another way to put file and line numbers to the stacktrace. Use libbfd (sourceware.org/binutils/docs-2.21/bfd/…) as they did in refdbg: refdbg.cvs.sourceforge.net/viewvc/refdbg/refdbg/… I did not tried it myself yet. – rve Nov 2, 2011 at 14:19

2 In addition to using `-rdynamic`, also check that your build system doesn't add `-fvisibility=hidden` option! (as it will completely discard the effect of `-rdynamic`) – Dima Litvinov May 13, 2020 at 0:43

8

If you're using Linux, the standard C library includes a function called `backtrace`, which populates an array with frames' return addresses, and another function called `backtrace_symbols`, which will take the addresses from `backtrace` and look up the corresponding function names. These are documented in the GNU C Library manual.

Those won't show argument values, source lines, and the like, and they only apply to the calling thread. However, they should be a lot faster (and perhaps less flaky) than running GDB that way, so they have their place.

Share

Improve this answer

Follow

answered Aug 2, 2010 at 19:13

community wiki
Jim Blandy

3 Actually snippet that I insert into program firstly outputs backtrace with backtrace_symbols and then starts gdb to output fully annotated stack traces for all threads. If gdb fails, I still has the `backtrace`'s stacktrace. – Vi. Aug 4, 2010 at 16:51

7

nobar posted a fantastic answer. In short;

> So you want a **stand-alone function that prints a stack trace** with all of the features that **gdb** stack traces have and that doesn't terminate your application. The answer is to automate the launch of gdb in a non-interactive mode to perform just the tasks that you want.
>
> This is done by executing gdb in a child process, using fork(), and scripting it to display a stack-trace while your application waits for it to complete. This can be performed without the use of a core-dump and without aborting the application.

I believe that this is what you are looking for, @Vi

Share

Improve this answer

Follow

edited Jun 20, 2020 at 9:12

community wiki
3 revs, 2 users 87%
karlphillip

2 Look at the sample code in the question. It is that method. I'm looking for other, less heavyweight ways. The main problem of addr2line-quality things that it often cannot display line number where gdb can. – Vi. Jan 20, 2011 at 0:05

1 @Vi It is stated in his answer that he got the base code from your question in this thread. However, if you look more closely you will see that there are some differences. Have you tried it? – karlphillip Jan 20, 2011 at 0:43

Isn't `abort()` simpler?

That way if it happens in the field the customer can send you the core file (I don't know many users who are involved enough in *my* application to want me to force them to debug it).

Share Improve this answer Follow

answered Jun 30, 2010 at 18:30

Motti

114k ● 56 ● 194 ● 273

4 I don't need to abort. I need a stack trace. Program can continue after printing it. And I like the verbosity of "bt full" – Vi. Jun 30, 2010 at 19:07

1 Also print_trace() way is rather unintrusive. If `gdb` in not found the program can just continue without printing a stacktrace. – Vi. Jun 30, 2010 at 19:11

@Vi, OK sorry I wasn't any help :o/ – Motti Jul 1, 2010 at 6:04