Rocky Enterprise Linux 9.2 Manual Pages on command 'backtrace_symbols.3'

$ man backtrace_symbols.3

BACKTRACE(3) Linux Programmer's Manual BACKTRACE(3)

NAME

backtrace, backtrace_symbols, backtrace_symbols_fd - support for appli?

cation self-debugging

SYNOPSIS

#include <execinfo.h>

int backtrace(void **buffer, int size);

char **backtrace_symbols(void *const *buffer, int size);

void backtrace_symbols_fd(void *const *buffer, int size, int fd);

DESCRIPTION

backtrace() returns a backtrace for the calling program, in the array pointed to by buffer. A backtrace is the series of currently active function calls for the program. Each item in the array pointed to by buffer is of type void *, and is the return address from the corre? sponding stack frame. The size argument specifies the maximum number of addresses that can be stored in buffer. If the backtrace is larger than size, then the addresses corresponding to the size most recent function calls are returned; to obtain the complete backtrace, make

sure that buffer and size are large enough.

Given the set of addresses returned by backtrace() in buffer, back? trace_symbols() translates the addresses into an array of strings that describe the addresses symbolically. The size argument specifies the number of addresses in buffer. The symbolic representation of each ad? dress consists of the function name (if this can be determined), a hexadecimal offset into the function, and the actual return address (in hexadecimal). The address of the array of string pointers is returned as the function result of backtrace_symbols(). This array is mal? loc(3)ed by backtrace_symbols(), and must be freed by the caller. (The strings pointed to by the array of pointers need not and should not be freed.)

backtrace_symbols_fd() takes the same buffer and size arguments as backtrace_symbols(), but instead of returning an array of strings to the caller, it writes the strings, one per line, to the file descriptor fd. backtrace_symbols_fd() does not call malloc(3), and so can be em? ployed in situations where the latter function might fail, but see NOTES.

RETURN VALUE

backtrace() returns the number of addresses returned in buffer, which is not greater than size. If the return value is less than size, then the full backtrace was stored; if it is equal to size, then it may have been truncated, in which case the addresses of the oldest stack frames are not returned.

On success, backtrace_symbols() returns a pointer to the array mal? loc(3)ed by the call; on error, NULL is returned.

VERSIONS

backtrace(), backtrace_symbols(), and backtrace_symbols_fd() are pro? vided in glibc since version 2.1.

ATTRIBUTES

For an explanation of the terms used in this section, see at? tributes(7).

?Interface ? Attribute ? Value ?

??

?backtrace(), ? Thread safety ? MT-Safe ?

?backtrace_symbols(), ? ? ?

?backtrace_symbols_fd() ? ? ?

??

CONFORMING TO

These functions are GNU extensions.

NOTES

These functions make some assumptions about how a function's return ad?
dress is stored on the stack. Note the following:

* Omission of the frame pointers (as implied by any of gcc(1)'s non?
zero optimization levels) may cause these assumptions to be vio?
lated.

* Inlined functions do not have stack frames.

* Tail-call optimization causes one stack frame to replace another.

* backtrace() and backtrace_symbols_fd() don't call malloc() explic?
itly, but they are part of libgcc, which gets loaded dynamically
when first used. Dynamic loading usually triggers a call to mal?
loc(3). If you need certain calls to these two functions to not al?
locate memory (in signal handlers, for example), you need to make
sure libgcc is loaded beforehand.

The symbol names may be unavailable without the use of special linker
options. For systems using the GNU linker, it is necessary to use the
-rdynamic linker option. Note that names of "static" functions are not
exposed, and won't be available in the backtrace.

EXAMPLES

The program below demonstrates the use of backtrace() and back?
trace_symbols(). The following shell session shows what we might see
when running the program:

```
$ cc -rdynamic prog.c -o prog
$ ./prog 3
backtrace() returned 8 addresses
```

```
./prog(myfunc3+0x5c) [0x80487f0]
./prog [0x8048871]
./prog(myfunc+0x21) [0x8048894]
./prog(myfunc+0x1a) [0x804888d]
./prog(myfunc+0x1a) [0x804888d]
./prog(main+0x65) [0x80488fb]
/lib/libc.so.6(__libc_start_main+0xdc) [0xb7e38f9c]
./prog [0x8048711]
```

Program source

```
#include <execinfo.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define BT_BUF_SIZE 100
void
myfunc3(void)
{
    int nptrs;
    void *buffer[BT_BUF_SIZE];
    char **strings;
    nptrs = backtrace(buffer, BT_BUF_SIZE);
    printf("backtrace() returned %d addresses\n", nptrs);
    /* The call backtrace_symbols_fd(buffer, nptrs, STDOUT_FILENO)
       would produce similar output to the following: */
    strings = backtrace_symbols(buffer, nptrs);
    if (strings == NULL) {
        perror("backtrace_symbols");
        exit(EXIT_FAILURE);
    }
    for (int j = 0; j < nptrs; j++)
        printf("%s\n", strings[j]);
    free(strings);
}
```

```
static void   /* "static" means don't export the symbol... */
myfunc2(void)
{
    myfunc3();
}
void
myfunc(int ncalls)
{
    if (ncalls > 1)
        myfunc(ncalls - 1);
    else
        myfunc2();
}
int
main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "%s num-calls\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    myfunc(atoi(argv[1]));
    exit(EXIT_SUCCESS);
}
```

SEE ALSO

addr2line(1), gcc(1), gdb(1), ld(1), dlopen(3), malloc(3)

COLOPHON

This page is part of release 5.10 of the Linux man-pages project. A description of the project, information about reporting bugs, and the latest version of this page, can be found at https://www.kernel.org/doc/man-pages/.