NAME

sigaltstack − set and/or get signal stack context

SYNOPSIS

#include <signal.h>

int sigaltstack(const stack_t ***_ss_, stack_t ***_oss_**);**

Feature Test Macro Requirements for glibc (see **feature_test_macros**(7)):

sigaltstack(): _BSD_SOURCE || _XOPEN_SOURCE >= 500

DESCRIPTION

sigaltstack() allows a process to define a new alternate signal stack and/or retrieve the state of an existing alternate signal stack. An alternate signal stack is used during the execution of a signal handler if the establishment of that handler (see **sigaction**(2)) requested it.

The normal sequence of events for using an alternate signal stack is the following:

1. Allocate an area of memory to be used for the alternate signal stack.
2. Use **sigaltstack**() to inform the system of the existence and location of the alternate signal stack.
3. When establishing a signal handler using **sigaction**(2), inform the system that the signal handler should be executed on the alternate signal stack by specifying the **SA_ONSTACK** flag.

The *ss* argument is used to specify a new alternate signal stack, while the *oss* argument is used to retrieve information about the currently established signal stack. If we are interested in performing just one of these tasks then the other argument can be specified as NULL. Each of these arguments is a structure of the following type:

```
typedef struct {
    void  *ss_sp;     /* Base address of stack */
    int    ss_flags;  /* Flags */
    size_t ss_size;   /* Number of bytes in stack */
} stack_t;
```

To establish a new alternate signal stack, *ss.ss_flags* is set to zero, and *ss.ss_sp* and *ss.ss_size* specify the starting address and size of the stack. The constant **SIGSTKSZ** is defined to be large enough to cover the usual size requirements for an alternate signal stack, and the constant **MINSIGSTKSZ** defines the minimum size required to execute a signal handler.

When a signal handler is invoked on the alternate stack, the kernel automatically aligns the address given in *ss.ss_sp* to a suitable address boundary for the underlying hardware architecture.

To disable an existing stack, specify *ss.ss_flags* as **SS_DISABLE**. In this case, the remaining fields in *ss* are ignored.

If *oss* is not NULL, then it is used to return information about the alternate signal stack which was in effect prior to the call to **sigaltstack**(). The *oss.ss_sp* and *oss.ss_size* fields return the starting address and size of that stack. The *oss.ss_flags* may return either of the following values:

SS_ONSTACK
The process is currently executing on the alternate signal stack. (Note that it is not possible to change the alternate signal stack if the process is currently executing on it.)

SS_DISABLE
The alternate signal stack is currently disabled.

RETURN VALUE

sigaltstack() returns 0 on success, or −1 on failure with *errno* set to indicate the error.

ERRORS

EFAULT
Either *ss* or *oss* is not NULL and points to an area outside of the process's address space.

EINVAL
ss is not NULL and the *ss_flags* field contains a non-zero value other than **SS_DISABLE**.

ENOMEM
The specified size of the new alternate signal stack (*ss.ss_size*) was less than **MINSTKSZ**.

EPERM
An attempt was made to change the alternate signal stack while it was active (i.e., the process was already executing on the current alternate signal stack).

CONFORMING TO

SUSv2, SVr4, POSIX.1-2001.

NOTES

The most common usage of an alternate signal stack is to handle the **SIGSEGV** signal that is generated if the space available for the normal process stack is exhausted: in this case, a signal handler for **SIGSEGV** cannot be invoked on the process stack; if we wish to handle it, we must use an alternate signal stack.

Establishing an alternate signal stack is useful if a process expects that it may exhaust its standard stack. This may occur, for example, because the stack grows so large that it encounters the upwardly growing heap, or it reaches a limit established by a call to **setrlimit(RLIMIT_STACK, &rlim)**. If the standard stack is exhausted, the kernel sends the process a **SIGSEGV** signal. In these circumstances the only way to catch this signal is on an alternate signal stack.

On most hardware architectures supported by Linux, stacks grow downwards. **sigaltstack**() automatically takes account of the direction of stack growth.

Functions called from a signal handler executing on an alternate signal stack will also use the alternate signal stack. (This also applies to any handlers invoked for other signals while the process is executing on the alternate signal stack.) Unlike the standard stack, the system does not automatically extend the alternate signal stack. Exceeding the allocated size of the alternate signal stack will lead to unpredictable results.

A successful call to **execve**(2) removes any existing alternate signal stack. A child process created via **fork**() inherits a copy of its parent's alternate signal stack settings.

sigaltstack() supersedes the older **sigstack**() call. For backwards compatibility, glibc also provides **sigstack**(). All new applications should be written using **sigaltstack**().

History

4.2BSD had a **sigstack**() system call. It used a slightly different struct, and had the major disadvantage that the caller had to know the direction of stack growth.

EXAMPLE

The following code segment demonstrates the use of **sigaltstack**():

```
stack_t ss;

ss.ss_sp = malloc(SIGSTKSZ);
if (ss.ss_sp == NULL)
    /* Handle error */;
ss.ss_size = SIGSTKSZ;
ss.ss_flags = 0;
if (sigaltstack(&ss, NULL) == −1)
    /* Handle error */;
```

SEE ALSO

execve(2), **setrlimit**(2), **sigaction**(2), **siglongjmp**(3), **sigsetjmp**(3), **signal**(7)

COLOPHON

This page is part of release 3.22 of the Linux *man-pages* project. A description of the project, and information about reporting bugs, can be found at http://www.kernel.org/doc/man-pages/.