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

$ man longjmp.3

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

NAME

setjmp, sigsetjmp, longjmp, siglongjmp - performing a nonlocal goto

SYNOPSIS

#include <setjmp.h>

int setjmp(jmp_buf env);

int sigsetjmp(sigjmp_buf env, int savesigs);

void longjmp(jmp_buf env, int val);

void siglongjmp(sigjmp_buf env, int val);

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

setjmp(): see NOTES.

sigsetjmp(): _POSIX_C_SOURCE

DESCRIPTION

The functions described on this page are used for performing "nonlocal gotos": transferring execution from one function to a predetermined lo? cation in another function. The setjmp() function dynamically estab? lishes the target to which control will later be transferred, and longjmp() performs the transfer of execution.

The setjmp() function saves various information about the calling envi?
ronment (typically, the stack pointer, the instruction pointer, possi?
bly the values of other registers and the signal mask) in the buffer
env for later use by longjmp(). In this case, setjmp() returns 0.

The longjmp() function uses the information saved in env to transfer
control back to the point where setjmp() was called and to restore
("rewind") the stack to its state at the time of the setjmp() call. In
addition, and depending on the implementation (see NOTES), the values
of some other registers and the process signal mask may be restored to
their state at the time of the setjmp() call.

Following a successful longjmp(), execution continues as if setjmp()
had returned for a second time. This "fake" return can be distin?
guished from a true setjmp() call because the "fake" return returns the
value provided in val. If the programmer mistakenly passes the value 0
in val, the "fake" return will instead return 1.

sigsetjmp() and siglongjmp()

sigsetjmp() and siglongjmp() also perform nonlocal gotos, but provide
predictable handling of the process signal mask.

If, and only if, the savesigs argument provided to sigsetjmp() is non?
zero, the process's current signal mask is saved in env and will be re?
stored if a siglongjmp() is later performed with this env.

RETURN VALUE

setjmp() and sigsetjmp() return 0 when called directly; on the "fake"
return that occurs after longjmp() or siglongjmp(), the nonzero value
specified in val is returned.

The longjmp() or siglongjmp() functions do not return.

ATTRIBUTES

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

Interface	Attribute	Value
setjmp(), sigsetjmp()	Thread safety	MT-Safe

??

?longjmp(), siglongjmp() ? Thread safety ? MT-Safe ?

??

CONFORMING TO

setjmp(), longjmp(): POSIX.1-2001, POSIX.1-2008, C89, C99.

sigsetjmp(), siglongjmp(): POSIX.1-2001, POSIX.1-2008.

NOTES

POSIX does not specify whether setjmp() will save the signal mask (to be later restored during longjmp()). In System V it will not. In 4.3BSD it will, and there is a function _setjmp() that will not. The behavior under Linux depends on the glibc version and the setting of feature test macros. On Linux with glibc versions before 2.19, setjmp() follows the System V behavior by default, but the BSD behavior is provided if the _BSD_SOURCE feature test macro is explicitly defined and none of _POSIX_SOURCE, _POSIX_C_SOURCE, _XOPEN_SOURCE, _GNU_SOURCE, or _SVID_SOURCE is defined. Since glibc 2.19, <setjmp.h> exposes only the System V version of setjmp(). Programs that need the BSD semantics should replace calls to setjmp() with calls to sigsetjmp() with a non? zero savesigs argument.

setjmp() and longjmp() can be useful for dealing with errors inside deeply nested function calls or to allow a signal handler to pass con? trol to a specific point in the program, rather than returning to the point where the handler interrupted the main program. In the latter case, if you want to portably save and restore signal masks, use sigsetjmp() and siglongjmp(). See also the discussion of program read? ability below.

The compiler may optimize variables into registers, and longjmp() may restore the values of other registers in addition to the stack pointer and program counter. Consequently, the values of automatic variables are unspecified after a call to longjmp() if they meet all the follow? ing criteria:

? they are local to the function that made the corresponding setjmp() call;

? their values are changed between the calls to setjmp() and longjmp(); and

? they are not declared as volatile.

Analogous remarks apply for siglongjmp().

Nonlocal gotos and program readability

While it can be abused, the traditional C "goto" statement at least has the benefit that lexical cues (the goto statement and the target label) allow the programmer to easily perceive the flow of control. Nonlocal gotos provide no such cues: multiple setjmp() calls might employ the same jmp_buf variable so that the content of the variable may change over the lifetime of the application. Consequently, the programmer may be forced to perform detailed reading of the code to determine the dy? namic target of a particular longjmp() call. (To make the programmer's life easier, each setjmp() call should employ a unique jmp_buf vari? able.)

Adding further difficulty, the setjmp() and longjmp() calls may not even be in the same source code module.

In summary, nonlocal gotos can make programs harder to understand and maintain, and an alternative should be used if possible.

Caveats

If the function which called setjmp() returns before longjmp() is called, the behavior is undefined. Some kind of subtle or unsubtle chaos is sure to result.

If, in a multithreaded program, a longjmp() call employs an env buffer that was initialized by a call to setjmp() in a different thread, the behavior is undefined.

POSIX.1-2008 Technical Corrigendum 2 adds longjmp() and siglongjmp() to the list of async-signal-safe functions. However, the standard recom? mends avoiding the use of these functions from signal handlers and goes on to point out that if these functions are called from a signal han? dler that interrupted a call to a non-async-signal-safe function (or some equivalent, such as the steps equivalent to exit(3) that occur upon a return from the initial call to main()), the behavior is unde?

fined if the program subsequently makes a call to a non-async-signal-safe function. The only way of avoiding undefined behavior is to ensure one of the following:

* After long jumping from the signal handler, the program does not call any non-async-signal-safe functions and does not return from the initial call to main().
* Any signal whose handler performs a long jump must be blocked during every call to a non-async-signal-safe function and no non-async-signal-safe functions are called after returning from the initial call to main().

SEE ALSO

signal(7), signal-safety(7)

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/.