
*Full credit is given to the above companies including the OS that this PDF file was generated!*

## *Red Hat Enterprise Linux Release 9.2 Manual Pages on '_Exit.3p' command*

***$ man _Exit.3p***

_EXIT(3P) POSIX Programmer's Manual _EXIT(3P)

PROLOG

This manual page is part of the POSIX Programmer's Manual. The Linux implementation of this interface may differ (consult the corresponding Linux manual page for details of Linux behavior), or the interface may not be implemented on Linux.

NAME

_Exit, _exit ? terminate a process

SYNOPSIS

```
#include <stdlib.h>
void _Exit(int status);
#include <unistd.h>
void _exit(int status);
```

DESCRIPTION

For _Exit(): The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1?2017 defers to the ISO C standard.

The value of status may be 0, EXIT_SUCCESS, EXIT_FAILURE, or any other value, though only the least significant 8 bits (that is, status & 0377) shall be available from wait() and waitpid(); the full value shall be available from waitid() and in the siginfo_t passed to a sig? nal handler for SIGCHLD.

The _Exit() and _exit() functions shall be functionally equivalent.

The _Exit() and _exit() functions shall not call functions registered with atexit() nor any registered signal handlers. Open streams shall not be flushed. Whether open streams are closed (without flushing) is implementation-defined. Finally, the calling process shall be termi? nated with the consequences described below.

Consequences of Process Termination

Process termination caused by any reason shall have the following con? sequences:

Note: These consequences are all extensions to the ISO C standard and are not further CX shaded. However, functionality relat? ing to the XSI option is shaded.

* All of the file descriptors, directory streams, conversion descrip? tors, and message catalog descriptors open in the calling process shall be closed.

* If the parent process of the calling process has set its SA_NOCLD? WAIT flag or has set the action for the SIGCHLD signal to SIG_IGN:

  -- The process' status information (see Section 2.13, Status In? formation), if any, shall be discarded.

  -- The lifetime of the calling process shall end immediately. If SA_NOCLDWAIT is set, it is implementation-defined whether a SIGCHLD signal is sent to the parent process.

  -- If a thread in the parent process of the calling process is blocked in wait(), waitpid(), or waitid(), and the parent process has no remaining child processes in the set of waited- for children, the wait(), waitid(), or waitpid() function shall fail and set errno to [ECHILD].

  Otherwise:

  -- Status information (see Section 2.13, Status Information) shall be generated.

  -- The calling process shall be transformed into a zombie process. Its status information shall be made available to the parent process until the process' lifetime ends.

-- The process' lifetime shall end once its parent obtains the process' status information via a currently-blocked or future call to wait(), waitid() (without WNOWAIT), or waitpid().

-- If one or more threads in the parent process of the calling process is blocked in a call to wait(), waitid(), or waitpid() awaiting termination of the process, one (or, if any are call? ing waitid() with WNOWAIT, possibly more) of these threads shall obtain the process' status information as specified in Section 2.13, Status Information and become unblocked.

-- A SIGCHLD shall be sent to the parent process.

* Termination of a process does not directly terminate its children. The sending of a SIGHUP signal as described below indirectly termi? nates children in some circumstances.

* The parent process ID of all of the existing child processes and zombie processes of the calling process shall be set to the process ID of an implementation-defined system process. That is, these pro? cesses shall be inherited by a special system process.

* Each attached shared-memory segment is detached and the value of shm_nattch (see shmget()) in the data structure associated with its shared memory ID shall be decremented by 1.

* For each semaphore for which the calling process has set a semadj value (see semop()), that value shall be added to the semval of the specified semaphore.

* If the process is a controlling process, the SIGHUP signal shall be sent to each process in the foreground process group of the con? trolling terminal belonging to the calling process.

* If the process is a controlling process, the controlling terminal associated with the session shall be disassociated from the ses? sion, allowing it to be acquired by a new controlling process.

* If the exit of the process causes a process group to become or? phaned, and if any member of the newly-orphaned process group is stopped, then a SIGHUP signal followed by a SIGCONT signal shall be sent to each process in the newly-orphaned process group.

* All open named semaphores in the calling process shall be closed as if by appropriate calls to sem_close().
* Any memory locks established by the process via calls to mlockall() or mlock() shall be removed. If locked pages in the address space of the calling process are also mapped into the address spaces of other processes and are locked by those processes, the locks estab? lished by the other processes shall be unaffected by the call by this process to _Exit() or _exit().
* Memory mappings that were created in the process shall be unmapped before the process is destroyed.
* Any blocks of typed memory that were mapped in the calling process shall be unmapped, as if munmap() was implicitly called to unmap them.
* All open message queue descriptors in the calling process shall be closed as if by appropriate calls to mq_close().
* Any outstanding cancelable asynchronous I/O operations may be can? celed. Those asynchronous I/O operations that are not canceled shall complete as if the _Exit() or _exit() operation had not yet occurred, but any associated signal notifications shall be sup? pressed. The _Exit() or _exit() operation may block awaiting such I/O completion. Whether any I/O is canceled, and which I/O may be canceled upon _Exit() or _exit(), is implementation-defined.
* Threads terminated by a call to _Exit() or _exit() shall not invoke their cancellation cleanup handlers or per-thread data destructors.
* If the calling process is a trace controller process, any trace streams that were created by the calling process shall be shut down as described by the posix_trace_shutdown() function, and mapping of trace event names to trace event type identifiers of any process built for these trace streams may be deallocated.

RETURN VALUE

These functions do not return.

ERRORS

No errors are defined.

The following sections are informative.

## EXAMPLES

None.

## APPLICATION USAGE

Normally applications should use exit() rather than _Exit() or _exit().

## RATIONALE

### Process Termination

Early proposals drew a distinction between normal and abnormal process termination. Abnormal termination was caused only by certain signals and resulted in implementation-defined ``actions'', as discussed below.

Subsequent proposals distinguished three types of termination: normal termination (as in the current specification), simple abnormal termina? tion, and abnormal termination with actions. Again the distinction be? tween the two types of abnormal termination was that they were caused by different signals and that implementation-defined actions would re? sult in the latter case. Given that these actions were completely im? plementation-defined, the early proposals were only saying when the ac? tions could occur and how their occurrence could be detected, but not what they were. This was of little or no use to conforming applica? tions, and thus the distinction is not made in this volume of POSIX.1?2017.

The implementation-defined actions usually include, in most historical implementations, the creation of a file named core in the current work? ing directory of the process. This file contains an image of the memory of the process, together with descriptive information about the process, perhaps sufficient to reconstruct the state of the process at the receipt of the signal.

There is a potential security problem in creating a core file if the process was set-user-ID and the current user is not the owner of the program, if the process was set-group-ID and none of the user's groups match the group of the program, or if the user does not have permission to write in the current directory. In this situation, an implementation either should not create a core file or should make it unreadable by

the user.

Despite the silence of this volume of POSIX.1?2017 on this feature, ap? plications are advised not to create files named core because of poten? tial conflicts in many implementations. Some implementations use a name other than core for the file; for example, by appending the process ID to the filename.

Terminating a Process

It is important that the consequences of process termination as de? scribed occur regardless of whether the process called _exit() (perhaps indirectly through exit()) or instead was terminated due to a signal or for some other reason. Note that in the specific case of exit() this means that the status argument to exit() is treated in the same way as the status argument to _exit().

A language other than C may have other termination primitives than the C-language exit() function, and programs written in such a language should use its native termination primitives, but those should have as part of their function the behavior of _exit() as described. Implemen? tations in languages other than C are outside the scope of this version of this volume of POSIX.1?2017, however.

As required by the ISO C standard, using return from main() has the same behavior (other than with respect to language scope issues) as calling exit() with the returned value. Reaching the end of the main() function has the same behavior as calling exit(0).

A value of zero (or EXIT_SUCCESS, which is required to be zero) for the argument status conventionally indicates successful termination. This corresponds to the specification for exit() in the ISO C standard. The convention is followed by utilities such as make and various shells, which interpret a zero status from a child process as success. For this reason, applications should not call exit(0) or _exit(0) when they ter? minate unsuccessfully; for example, in signal-catching functions.

Historically, the implementation-defined process that inherits children whose parents have terminated without waiting on them is called init and has a process ID of 1.

The sending of a SIGHUP to the foreground process group when a control? ling process terminates corresponds to somewhat different historical implementations. In System V, the kernel sends a SIGHUP on termination of (essentially) a controlling process. In 4.2 BSD, the kernel does not send SIGHUP in a case like this, but the termination of a controlling process is usually noticed by a system daemon, which arranges to send a SIGHUP to the foreground process group with the vhangup() function. However, in 4.2 BSD, due to the behavior of the shells that support job control, the controlling process is usually a shell with no other pro? cesses in its process group. Thus, a change to make _exit() behave this way in such systems should not cause problems with existing applica? tions.

The termination of a process may cause a process group to become or? phaned in either of two ways. The connection of a process group to its parent(s) outside of the group depends on both the parents and their children. Thus, a process group may be orphaned by the termination of the last connecting parent process outside of the group or by the ter? mination of the last direct descendant of the parent process(es). In either case, if the termination of a process causes a process group to become orphaned, processes within the group are disconnected from their job control shell, which no longer has any information on the existence of the process group. Stopped processes within the group would languish forever. In order to avoid this problem, newly orphaned process groups that contain stopped processes are sent a SIGHUP signal and a SIGCONT signal to indicate that they have been disconnected from their session. The SIGHUP signal causes the process group members to terminate unless they are catching or ignoring SIGHUP. Under most circumstances, all of the members of the process group are stopped if any of them are stopped.

The action of sending a SIGHUP and a SIGCONT signal to members of a newly orphaned process group is similar to the action of 4.2 BSD, which sends SIGHUP and SIGCONT to each stopped child of an exiting process. If such children exit in response to the SIGHUP, any additional descen?

dants receive similar treatment at that time. In this volume of POSIX.1?2017, the signals are sent to the entire process group at the same time. Also, in this volume of POSIX.1?2017, but not in 4.2 BSD, stopped processes may be orphaned, but may be members of a process group that is not orphaned; therefore, the action taken at _exit() must consider processes other than child processes.

It is possible for a process group to be orphaned by a call to setpgid() or setsid(), as well as by process termination. This volume of POSIX.1?2017 does not require sending SIGHUP and SIGCONT in those cases, because, unlike process termination, those cases are not caused accidentally by applications that are unaware of job control. An imple? mentation can choose to send SIGHUP and SIGCONT in those cases as an extension; such an extension must be documented as required in <sig? nal.h>.

The ISO/IEC 9899:1999 standard adds the _Exit() function that results in immediate program termination without triggering signals or atexit()-registered functions. In POSIX.1?2008, this is equivalent to the _exit() function.

## FUTURE DIRECTIONS

None.

## SEE ALSO

atexit(), exit(), mlock(), mlockall(), mq_close(), munmap(), posix_trace_create(), sem_close(), semop(), setpgid(), setsid(), shmget(), wait(), waitid()

The Base Definitions volume of POSIX.1?2017, <stdlib.h>, <unistd.h>

## COPYRIGHT

is the referee document. The original Standard can be obtained online at http://www.opengroup.org/unix/online.html .

Any typographical or formatting errors that appear in this page are most likely to have been introduced during the conversion of the source files to man page format. To report such errors, see https://www.ker?nel.org/doc/man-pages/reporting_bugs.html .