wait(2) — Linux manual page

NAME top

wait, waitpid, waitid - wait for process to change state

SYNOPSIS top

```
#include <sys/wait.h>

pid_t wait(int *wstatus);
pid_t waitpid(pid_t pid, int *wstatus, int options);

int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
                /* This is the glibc and POSIX interface; see
                   NOTES for information on the raw system call. */
```

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

```
waitid():
    Since glibc 2.26:
        _XOPEN_SOURCE >= 500 || _POSIX_C_SOURCE >= 200809L
    Glibc 2.25 and earlier:
        _XOPEN_SOURCE
            || /* Since glibc 2.12: */ _POSIX_C_SOURCE >= 200809L
            || /* Glibc <= 2.19: */ _BSD_SOURCE
```

DESCRIPTION top

All of these system calls are used to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed. A state change is considered to be: the child terminated; the child was stopped by a signal; or the child was resumed by a signal. In the case of a terminated child, performing a wait allows the system to release the resources associated with the child; if a wait is not performed, then the terminated child remains in a "zombie" state (see NOTES below).

If a child has already changed state, then these calls return

immediately. Otherwise, they block until either a child changes state or a signal handler interrupts the call (assuming that system calls are not automatically restarted using the **SA_RESTART** flag of sigaction(2)). In the remainder of this page, a child whose state has changed and which has not yet been waited upon by one of these system calls is termed *waitable*.

wait() and waitpid()

The **wait**() system call suspends execution of the calling thread until one of its children terminates. The call *wait(&wstatus)* is equivalent to:

```
waitpid(-1, &wstatus, 0);
```

The **waitpid**() system call suspends execution of the calling thread until a child specified by *pid* argument has changed state. By default, **waitpid**() waits only for terminated children, but this behavior is modifiable via the *options* argument, as described below.

The value of *pid* can be:

`< -1` meaning wait for any child process whose process group ID is equal to the absolute value of *pid*.

`-1` meaning wait for any child process.

`0` meaning wait for any child process whose process group ID is equal to that of the calling process at the time of the call to **waitpid**().

`> 0` meaning wait for the child whose process ID is equal to the value of *pid*.

The value of *options* is an OR of zero or more of the following constants:

WNOHANG
: return immediately if no child has exited.

WUNTRACED
: also return if a child has stopped (but not traced via ptrace(2)). Status for *traced* children which have stopped is provided even if this option is not specified.

WCONTINUED (since Linux 2.6.10)
: also return if a stopped child has been resumed by delivery of **SIGCONT**.

(For Linux-only options, see below.)

If *wstatus* is not NULL, **wait**() and **waitpid**() store status information in the *int* to which it points. This integer can be inspected with the following macros (which take the integer itself as an argument, not a pointer to it, as is done in **wait**() and **waitpid**()!):

WIFEXITED(*wstatus***)**
returns true if the child terminated normally, that is, by calling exit(3) or _exit(2), or by returning from main().

WEXITSTATUS(*wstatus***)**
returns the exit status of the child. This consists of the least significant 8 bits of the *status* argument that the child specified in a call to exit(3) or _exit(2) or as the argument for a return statement in main(). This macro should be employed only if **WIFEXITED** returned true.

WIFSIGNALED(*wstatus***)**
returns true if the child process was terminated by a signal.

WTERMSIG(*wstatus***)**
returns the number of the signal that caused the child process to terminate. This macro should be employed only if **WIFSIGNALED** returned true.

WCOREDUMP(*wstatus***)**
returns true if the child produced a core dump (see core(5)). This macro should be employed only if **WIFSIGNALED** returned true.

This macro is not specified in POSIX.1-2001 and is not available on some UNIX implementations (e.g., AIX, SunOS). Therefore, enclose its use inside *#ifdef WCOREDUMP ... #endif*.

WIFSTOPPED(*wstatus***)**
returns true if the child process was stopped by delivery of a signal; this is possible only if the call was done using **WUNTRACED** or when the child is being traced (see ptrace(2)).

WSTOPSIG(*wstatus***)**
returns the number of the signal which caused the child to stop. This macro should be employed only if **WIFSTOPPED** returned true.

WIFCONTINUED(*wstatus***)**
(since Linux 2.6.10) returns true if the child process was resumed by delivery of **SIGCONT**.

waitid()
The **waitid**() system call (available since Linux 2.6.9) provides more precise control over which child state changes to wait for.

The *idtype* and *id* arguments select the child(ren) to wait for, as follows:

idtype == **P_PID**
Wait for the child whose process ID matches *id*.

idtype == **P_PIDFD** (since Linux 5.4)
Wait for the child referred to by the PID file descriptor specified in *id*. (See pidfd_open(2) for further information on PID file descriptors.)

idtype == **P_PGID**
Wait for any child whose process group ID matches *id*. Since Linux 5.4, if *id* is zero, then wait for any child that is in the same process group as the caller's process group at the time of the call.

idtype == **P_ALL**
Wait for any child; *id* is ignored.

The child state changes to wait for are specified by ORing one or more of the following flags in *options*:

WEXITED
Wait for children that have terminated.

WSTOPPED
Wait for children that have been stopped by delivery of a signal.

WCONTINUED
Wait for (previously stopped) children that have been resumed by delivery of **SIGCONT**.

The following flags may additionally be ORed in *options*:

WNOHANG
As for **waitpid**().

WNOWAIT
Leave the child in a waitable state; a later wait call can be used to again retrieve the child status information.

Upon successful return, **waitid**() fills in the following fields of the *siginfo_t* structure pointed to by *infop*:

si_pid The process ID of the child.

si_uid The real user ID of the child. (This field is not set on most other implementations.)

si_signo
Always set to **SIGCHLD**.

si_status
Either the exit status of the child, as given to _exit(2) (or exit(3)), or the signal that caused the child to terminate, stop, or continue. The *si_code* field can be used to determine how to interpret this field.

si_code
Set to one of: **CLD_EXITED** (child called _exit(2));

CLD_KILLED (child killed by signal); **CLD_DUMPED** (child killed by signal, and dumped core); **CLD_STOPPED** (child stopped by signal); **CLD_TRAPPED** (traced child has trapped); or **CLD_CONTINUED** (child continued by **SIGCONT**).

If **WNOHANG** was specified in *options* and there were no children in a waitable state, then **waitid**() returns 0 immediately and the state of the *siginfo_t* structure pointed to by *infop* depends on the implementation. To (portably) distinguish this case from that where a child was in a waitable state, zero out the *si_pid* field before the call and check for a nonzero value in this field after the call returns.

POSIX.1-2008 Technical Corrigendum 1 (2013) adds the requirement that when **WNOHANG** is specified in *options* and there were no children in a waitable state, then **waitid**() should zero out the *si_pid* and *si_signo* fields of the structure. On Linux and other implementations that adhere to this requirement, it is not necessary to zero out the *si_pid* field before calling **waitid**(). However, not all implementations follow the POSIX.1 specification on this point.

RETURN VALUE top

wait(): on success, returns the process ID of the terminated child; on failure, -1 is returned.

waitpid(): on success, returns the process ID of the child whose state has changed; if **WNOHANG** was specified and one or more child(ren) specified by *pid* exist, but have not yet changed state, then 0 is returned. On failure, -1 is returned.

waitid(): returns 0 on success or if **WNOHANG** was specified and no child(ren) specified by *id* has yet changed state; on failure, -1 is returned.

On failure, each of these calls sets *errno* to indicate the error.

ERRORS top

EAGAIN The PID file descriptor specified in *id* is nonblocking and the process that it refers to has not terminated.

ECHILD (for **wait**()) The calling process does not have any unwaited-for children.

ECHILD (for **waitpid**() or **waitid**()) The process specified by *pid* (**waitpid**()) or *idtype* and *id* (**waitid**()) does not exist or is not a child of the calling process. (This can happen for one's own child if the action for **SIGCHLD** is set to **SIG_IGN**. See also the *Linux Notes* section about threads.)

EINTR **WNOHANG** was not set and an unblocked signal or a **SIGCHLD** was caught; see signal(7).

EINVAL The *options* argument was invalid.

ESRCH (for **wait**() or **waitpid**()) *pid* is equal to **INT_MIN**.

CONFORMING TO top

SVr4, 4.3BSD, POSIX.1-2001.

NOTES top

A child that terminates, but has not been waited for becomes a "zombie". The kernel maintains a minimal set of information about the zombie process (PID, termination status, resource usage information) in order to allow the parent to later perform a wait to obtain information about the child. As long as a zombie is not removed from the system via a wait, it will consume a slot in the kernel process table, and if this table fills, it will not be possible to create further processes. If a parent process terminates, then its "zombie" children (if any) are adopted by init(1), (or by the nearest "subreaper" process as defined through the use of the prctl(2) **PR_SET_CHILD_SUBREAPER** operation); init(1) automatically performs a wait to remove the zombies.

POSIX.1-2001 specifies that if the disposition of **SIGCHLD** is set to **SIG_IGN** or the **SA_NOCLDWAIT** flag is set for **SIGCHLD** (see sigaction(2)), then children that terminate do not become zombies and a call to **wait**() or **waitpid**() will block until all children have terminated, and then fail with *errno* set to **ECHILD**. (The original POSIX standard left the behavior of setting **SIGCHLD** to **SIG_IGN** unspecified. Note that even though the default disposition of **SIGCHLD** is "ignore", explicitly setting the disposition to **SIG_IGN** results in different treatment of zombie process children.)

Linux 2.6 conforms to the POSIX requirements. However, Linux 2.4 (and earlier) does not: if a **wait**() or **waitpid**() call is made while **SIGCHLD** is being ignored, the call behaves just as though **SIGCHLD** were not being ignored, that is, the call blocks until the next child terminates and then returns the process ID and status of that child.

Linux notes

In the Linux kernel, a kernel-scheduled thread is not a distinct construct from a process. Instead, a thread is simply a process that is created using the Linux-unique clone(2) system call; other routines such as the portable pthread_create(3) call are implemented using clone(2). Before Linux 2.4, a thread was just a special case of a process, and as a consequence one thread could not wait on the children of another thread, even when the latter belongs to the same thread group. However, POSIX prescribes such functionality, and since Linux 2.4 a thread can, and by default will, wait on children of other threads in the

same thread group.

The following Linux-specific *options* are for use with children created using clone(2); they can also, since Linux 4.7, be used with **waitid**():

__WCLONE
Wait for "clone" children only. If omitted, then wait for "non-clone" children only. (A "clone" child is one which delivers no signal, or a signal other than **SIGCHLD** to its parent upon termination.) This option is ignored if **__WALL** is also specified.

__WALL (since Linux 2.4)
Wait for all children, regardless of type ("clone" or "non-clone").

__WNOTHREAD (since Linux 2.4)
Do not wait for children of other threads in the same thread group. This was the default before Linux 2.4.

Since Linux 4.7, the **__WALL** flag is automatically implied if the child is being ptraced.

C library/kernel differences

wait() is actually a library function that (in glibc) is implemented as a call to wait4(2).

On some architectures, there is no **waitpid**() system call; instead, this interface is implemented via a C library wrapper function that calls wait4(2).

The raw **waitid**() system call takes a fifth argument, of type *struct rusage **. If this argument is non-NULL, then it is used to return resource usage information about the child, in the same manner as wait4(2). See getrusage(2) for details.

BUGS top

According to POSIX.1-2008, an application calling **waitid**() must ensure that *infop* points to a *siginfo_t* structure (i.e., that it is a non-null pointer). On Linux, if *infop* is NULL, **waitid**() succeeds, and returns the process ID of the waited-for child. Applications should avoid relying on this inconsistent, nonstandard, and unnecessary feature.

EXAMPLES top

The following program demonstrates the use of fork(2) and **waitpid**(). The program creates a child process. If no command-line argument is supplied to the program, then the child suspends its execution using pause(2), to allow the user to send signals to the child. Otherwise, if a command-line argument is supplied, then the child exits immediately, using the integer supplied on

the command line as the exit status. The parent process executes a loop that monitors the child using **waitpid**(), and uses the W*() macros described above to analyze the wait status value.

The following shell session demonstrates the use of the program:

```
$ ./a.out &
Child PID is 32360
[1] 32359
$ kill -STOP 32360
stopped by signal 19
$ kill -CONT 32360
continued
$ kill -TERM 32360
killed by signal 15
[1]+  Done                    ./a.out
$
```

Program source

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

int
main(int argc, char *argv[])
{
    pid_t cpid, w;
    int wstatus;

    cpid = fork();
    if (cpid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (cpid == 0) {            /* Code executed by child */
        printf("Child PID is %jd\n", (intmax_t) getpid());
        if (argc == 1)
            pause();                    /* Wait for signals */
        _exit(atoi(argv[1]));

    } else {                    /* Code executed by parent */
        do {
            w = waitpid(cpid, &wstatus, WUNTRACED | WCONTINUED);
            if (w == -1) {
                perror("waitpid");
                exit(EXIT_FAILURE);
            }

            if (WIFEXITED(wstatus)) {
                printf("exited, status=%d\n", WEXITSTATUS(wstatus));
            } else if (WIFSIGNALED(wstatus)) {
                printf("killed by signal %d\n", WTERMSIG(wstatus));
```

```
                } else if (WIFSTOPPED(wstatus)) {
                    printf("stopped by signal %d\n", WSTOPSIG(wstatus));
                } else if (WIFCONTINUED(wstatus)) {
                    printf("continued\n");
                }
            } while (!WIFEXITED(wstatus) && !WIFSIGNALED(wstatus));
            exit(EXIT_SUCCESS);
        }
    }
```

SEE ALSO top

_exit(2), clone(2), fork(2), kill(2), ptrace(2), sigaction(2), signal(2), wait4(2), pthread_create(3), core(5), credentials(7), signal(7)

COLOPHON top

This page is part of release 5.13 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/.

Linux **2021-08-27** **WAIT(2)**

Pages that refer to this page: intro(1), watch(1), clone(2), _exit(2), fork(2), getrusage(2), kill(2), pidfd_open(2), prctl(2), ptrace(2), reboot(2), seccomp(2), seccomp_unotify(2), sigaction(2), syscalls(2), times(2), vfork(2), wait4(2), clock(3), exit(3), __pmprocessexec(3), __pmprocesspipe(3), pmrecord(3), posix_spawn(3), pthread_exit(3), sd-event(3), sd_event_add_child(3), sd_event_add_inotify(3), system(3), proc(5), systemd.exec(5), credentials(7), man-pages(7), pthreads(7), signal(7), signal-safety(7), system_data_types(7), user_namespaces(7)