wait(3p) — Linux manual page

PROLOG top

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 top

wait, waitpid — wait for a child process to stop or terminate

SYNOPSIS top

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

pid_t wait(int *stat_loc);
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

DESCRIPTION top

The *wait*() and *waitpid*() functions shall obtain status information (see *Section 2.13*, *Status Information*) pertaining to one of the caller's child processes. The *wait*() function obtains status information for process termination from any child process. The *waitpid*() function obtains status information for process termination, and optionally process stop and/or continue, from a specified subset of the child processes.

The *wait*() function shall cause the calling thread to become blocked until status information generated by child process termination is made available to the thread, or until delivery of a signal whose action is either to execute a signal-catching function or to terminate the process, or an error occurs. If termination status information is available prior to the call to *wait*(), return shall be immediate. If termination status information is available for two or more child processes, the order in which their status is reported is unspecified.

As described in *Section 2.13*, *Status Information*, the *wait*() and *waitpid*() functions consume the status information they obtain.

The behavior when multiple threads are blocked in *wait*(),

waitid(), or *waitpid*() is described in *Section 2.13*, *Status Information*.

The *waitpid*() function shall be equivalent to *wait*() if the *pid* argument is (**pid_t**)-1 and the *options* argument is 0. Otherwise, its behavior shall be modified by the values of the *pid* and *options* arguments.

The *pid* argument specifies a set of child processes for which *status* is requested. The *waitpid*() function shall only return the status of a child process from this set:

* If *pid* is equal to (**pid_t**)-1, *status* is requested for any child process. In this respect, *waitpid*() is then equivalent to *wait*().

* If *pid* is greater than 0, it specifies the process ID of a single child process for which *status* is requested.

* If *pid* is 0, *status* is requested for any child process whose process group ID is equal to that of the calling process.

* If *pid* is less than (**pid_t**)-1, *status* is requested for any child process whose process group ID is equal to the absolute value of *pid*.

The *options* argument is constructed from the bitwise-inclusive OR of zero or more of the following flags, defined in the *<sys/wait.h>* header:

WCONTINUED The *waitpid*() function shall report the status of any continued child process specified by *pid* whose status has not been reported since it continued from a job control stop.

WNOHANG The *waitpid*() function shall not suspend execution of the calling thread if *status* is not immediately available for one of the child processes specified by *pid*.

WUNTRACED The status of any child processes specified by *pid* that are stopped, and whose status has not yet been reported since they stopped, shall also be reported to the requesting process.

If *wait*() or *waitpid*() return because the status of a child process is available, these functions shall return a value equal to the process ID of the child process. In this case, if the value of the argument *stat_loc* is not a null pointer, information shall be stored in the location pointed to by *stat_loc*. The value stored at the location pointed to by *stat_loc* shall be 0 if and only if the status returned is from a terminated child process that terminated by one of the following means:

1. The process returned 0 from *main*().

2. The process called *_exit*() or *exit*() with a *status* argument of 0.

3. The process was terminated because the last thread in the process terminated.

Regardless of its value, this information may be interpreted using the following macros, which are defined in *<sys/wait.h>* and evaluate to integral expressions; the *stat_val* argument is the integer value pointed to by *stat_loc*.

WIFEXITED(*stat_val*)
: Evaluates to a non-zero value if *status* was returned for a child process that terminated normally.

WEXITSTATUS(*stat_val*)
: If the value of WIFEXITED(*stat_val*) is non-zero, this macro evaluates to the low-order 8 bits of the *status* argument that the child process passed to *_exit*() or *exit*(), or the value the child process returned from *main*().

WIFSIGNALED(*stat_val*)
: Evaluates to a non-zero value if *status* was returned for a child process that terminated due to the receipt of a signal that was not caught (see *<signal.h>*).

WTERMSIG(*stat_val*)
: If the value of WIFSIGNALED(*stat_val*) is non-zero, this macro evaluates to the number of the signal that caused the termination of the child process.

WIFSTOPPED(*stat_val*)
: Evaluates to a non-zero value if *status* was returned for a child process that is currently stopped.

WSTOPSIG(*stat_val*)
: If the value of WIFSTOPPED(*stat_val*) is non-zero, this macro evaluates to the number of the signal that caused the child process to stop.

WIFCONTINUED(*stat_val*)
: Evaluates to a non-zero value if *status* was returned for a child process that has continued from a job control stop.

It is unspecified whether the *status* value returned by calls to *wait*() or *waitpid*() for processes created by *posix_spawn*() or *posix_spawnp*() can indicate a WIFSTOPPED(*stat_val*) before subsequent calls to *wait*() or *waitpid*() indicate WIFEXITED(*stat_val*) as the result of an error detected before the new process image starts executing.

It is unspecified whether the *status* value returned by calls to *wait*() or *waitpid*() for processes created by *posix_spawn*() or *posix_spawnp*() can indicate a WIFSIGNALED(*stat_val*) if a signal is sent to the parent's process group after *posix_spawn*() or *posix_spawnp*() is called.

If the information pointed to by *stat_loc* was stored by a call to *waitpid*() that specified the WUNTRACED flag and did not specify the WCONTINUED flag, exactly one of the macros WIFEXITED(**stat_loc*), WIFSIGNALED(**stat_loc*), and WIFSTOPPED(**stat_loc*) shall evaluate to a non-zero value.

If the information pointed to by *stat_loc* was stored by a call to *waitpid*() that specified the WUNTRACED and WCONTINUED flags, exactly one of the macros WIFEXITED(**stat_loc*), WIFSIGNALED(**stat_loc*), WIFSTOPPED(**stat_loc*), and WIFCONTINUED(**stat_loc*) shall evaluate to a non-zero value.

If the information pointed to by *stat_loc* was stored by a call to *waitpid*() that did not specify the WUNTRACED or WCONTINUED flags, or by a call to the *wait*() function, exactly one of the macros WIFEXITED(**stat_loc*) and WIFSIGNALED(**stat_loc*) shall evaluate to a non-zero value.

If the information pointed to by *stat_loc* was stored by a call to *waitpid*() that did not specify the WUNTRACED flag and specified the WCONTINUED flag, exactly one of the macros WIFEXITED(**stat_loc*), WIFSIGNALED(**stat_loc*), and WIFCONTINUED(**stat_loc*) shall evaluate to a non-zero value.

If _POSIX_REALTIME_SIGNALS is defined, and the implementation queues the SIGCHLD signal, then if *wait*() or *waitpid*() returns because the status of a child process is available, any pending SIGCHLD signal associated with the process ID of the child process shall be discarded. Any other pending SIGCHLD signals shall remain pending.

Otherwise, if SIGCHLD is blocked, if *wait*() or *waitpid*() return because the status of a child process is available, any pending SIGCHLD signal shall be cleared unless the status of another child process is available.

For all other conditions, it is unspecified whether child *status* will be available when a SIGCHLD signal is delivered.

There may be additional implementation-defined circumstances under which *wait*() or *waitpid*() report *status*. This shall not occur unless the calling process or one of its child processes explicitly makes use of a non-standard extension. In these cases the interpretation of the reported *status* is implementation-defined.

If a parent process terminates without waiting for all of its child processes to terminate, the remaining child processes shall be assigned a new parent process ID corresponding to an implementation-defined system process.

RETURN VALUE

top

If *wait*() or *waitpid*() returns because the status of a child process is available, these functions shall return a value equal to the process ID of the child process for which *status* is reported. If *wait*() or *waitpid*() returns due to the delivery of a signal to the calling process, -1 shall be returned and *errno* set to **[EINTR]**. If *waitpid*() was invoked with WNOHANG set in *options*, it has at least one child process specified by *pid* for which *status* is not available, and *status* is not available for any process specified by *pid*, 0 is returned. Otherwise, -1 shall be returned, and *errno* set to indicate the error.

ERRORS top

The *wait*() function shall fail if:

ECHILD The calling process has no existing unwaited-for child processes.

EINTR The function was interrupted by a signal. The value of the location pointed to by *stat_loc* is undefined.

The *waitpid*() function shall fail if:

ECHILD The process specified by *pid* does not exist or is not a child of the calling process, or the process group specified by *pid* does not exist or does not have any member process that is a child of the calling process.

EINTR The function was interrupted by a signal. The value of the location pointed to by *stat_loc* is undefined.

EINVAL The *options* argument is not valid.

The following sections are informative.

EXAMPLES top

Waiting for a Child Process and then Checking its Status

The following example demonstrates the use of *waitpid*(), *fork*(), and the macros used to interpret the status value returned by *waitpid*() (and *wait*()). The code segment creates a child process which does some unspecified work. Meanwhile the parent loops performing calls to *waitpid*() to monitor the status of the child. The loop terminates when child termination is detected.

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

pid_t child_pid, wpid;
int status;

child_pid = fork();
if (child_pid == -1) {      /* fork() failed */
    perror("fork");
    exit(EXIT_FAILURE);
}

if (child_pid == 0) {       /* This is the child */
    /* Child does some work and then terminates */
    ...

} else {                    /* This is the parent */
    do {
        wpid = waitpid(child_pid, &status, WUNTRACED
#ifdef WCONTINUED       /* Not all implementations support this */
        | WCONTINUED
```

```
        #endif
                );
                if (wpid == -1) {
                    perror("waitpid");
                    exit(EXIT_FAILURE);
                }

                if (WIFEXITED(status)) {
                    printf("child exited, status=%d\n", WEXITSTATUS(status));

                } else if (WIFSIGNALED(status)) {
                    printf("child killed (signal %d)\n", WTERMSIG(status));

                } else if (WIFSTOPPED(status)) {
                    printf("child stopped (signal %d)\n", WSTOPSIG(status));

        #ifdef WIFCONTINUED     /* Not all implementations support this */
                } else if (WIFCONTINUED(status)) {
                    printf("child continued\n");
        #endif
                } else {    /* Non-standard case -- may never happen */
                    printf("Unexpected status (0x%x)\n", status);
                }
            } while (!WIFEXITED(status) && !WIFSIGNALED(status));
        }
```

Waiting for a Child Process in a Signal Handler for SIGCHLD

The following example demonstrates how to use *waitpid*() in a signal handler for SIGCHLD without passing -1 as the *pid* argument. (See the APPLICATION USAGE section below for the reasons why passing a *pid* of -1 is not recommended.) The method used here relies on the standard behavior of *waitpid*() when SIGCHLD is blocked. On historical non-conforming systems, the status of some child processes might not be reported.

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

        #define CHILDREN 10

        static void
        handle_sigchld(int signum, siginfo_t *sinfo, void *unused)
        {
            int sav_errno = errno;
            int status;

            /*
             * Obtain status information for the child which
             * caused the SIGCHLD signal and write its exit code
             * to stdout.
            */
            if (sinfo->si_code != CLD_EXITED)
            {
                static char msg[] = "wrong si_code\n";
                write(2, msg, sizeof msg - 1);
```

```
        }
        else if (waitpid(sinfo->si_pid, &status, 0) == -1)
        {
            static char msg[] = "waitpid() failed\n";
            write(2, msg, sizeof msg - 1);
        }
        else if (!WIFEXITED(status))
        {
            static char msg[] = "WIFEXITED was false\n";
            write(2, msg, sizeof msg - 1);
        }
        else
        {
            int code = WEXITSTATUS(status);
            char buf[2];
            buf[0] = '0' + code;
            buf[1] = '\n';
            write(1, buf, 2);
        }
        errno = sav_errno;
    }

    int
    main(void)
    {
        int i;
        pid_t pid;
        struct sigaction sa;

        sa.sa_flags = SA_SIGINFO;
        sa.sa_sigaction = handle_sigchld;
        sigemptyset(&sa.sa_mask);
        if (sigaction(SIGCHLD, &sa, NULL) == -1)
        {
            perror("sigaction");
            exit(EXIT_FAILURE);
        }

        for (i = 0; i < CHILDREN; i++)
        {
            switch (pid = fork())
            {
            case -1:
                perror("fork");
                exit(EXIT_FAILURE);
            case 0:
                sleep(2);
                _exit(i);
            }
        }

        /* Wait for all the SIGCHLD signals, then terminate on SIGALRM */
        alarm(3);
        for (;;)
            pause();

        return 0; /* NOTREACHED */
    }
```

APPLICATION USAGE top

Calls to *wait*() will collect information about any child process. This may result in interactions with other interfaces that may be waiting for their own children (such as by use of *system*()). For this and other reasons it is recommended that portable applications not use *wait*(), but instead use *waitpid*(). For these same reasons, the use of *waitpid*() with a *pid* argument of -1, and the use of *waitid*() with the *idtype* argument set to P_ALL, are also not recommended for portable applications.

As specified in *Consequences of Process Termination*, if the calling process has SA_NOCLDWAIT set or has SIGCHLD set to SIG_IGN, then the termination of a child process will not cause status information to become available to a thread blocked in *wait*(), *waitid*(), or *waitpid*(). Thus, a thread blocked in one of the wait functions will remain blocked unless some other condition causes the thread to resume execution (such as an **[ECHILD]** failure due to no remaining children in the set of waited-for children).

RATIONALE top

A call to the *wait*() or *waitpid*() function only returns *status* on an immediate child process of the calling process; that is, a child that was produced by a single *fork*() call (perhaps followed by an *exec* or other function calls) from the parent. If a child produces grandchildren by further use of *fork*(), none of those grandchildren nor any of their descendants affect the behavior of a *wait*() from the original parent process. Nothing in this volume of POSIX.1-2017 prevents an implementation from providing extensions that permit a process to get *status* from a grandchild or any other process, but a process that does not use such extensions must be guaranteed to see *status* from only its direct children.

The *waitpid*() function is provided for three reasons:

1. To support job control
2. To permit a non-blocking version of the *wait*() function
3. To permit a library routine, such as *system*() or *pclose*(), to wait for its children without interfering with other terminated children for which the process has not waited

The first two of these facilities are based on the *wait3*() function provided by 4.3 BSD. The function uses the *options* argument, which is equivalent to an argument to *wait3*(). The WUNTRACED flag is used only in conjunction with job control on systems supporting job control. Its name comes from 4.3 BSD and refers to the fact that there are two types of stopped processes in that implementation: processes being traced via the *ptrace*() debugging facility and (untraced) processes stopped by job control signals. Since *ptrace*() is not part of this volume of POSIX.1-2017, only the second type is relevant. The name WUNTRACED was retained because its usage is the same, even though the name is not intuitively meaningful in this context.

The third reason for the *waitpid*() function is to permit independent sections of a process to spawn and wait for children without interfering with each other. For example, the following problem occurs in developing a portable shell, or command interpreter:

```
stream = popen("/bin/true");
(void) system("sleep 100");
(void) pclose(stream);
```

On all historical implementations, the final *pclose*() fails to reap the *wait*() *status* of the *popen*().

The status values are retrieved by macros, rather than given as specific bit encodings as they are in most historical implementations (and thus expected by existing programs). This was necessary to eliminate a limitation on the number of signals an implementation can support that was inherent in the traditional encodings. This volume of POSIX.1-2017 does require that a *status* value of zero corresponds to a process calling *_exit*(0), as this is the most common encoding expected by existing programs. Some of the macro names were adopted from 4.3 BSD.

These macros syntactically operate on an arbitrary integer value. The behavior is undefined unless that value is one stored by a successful call to *wait*() or *waitpid*() in the location pointed to by the *stat_loc* argument. An early proposal attempted to make this clearer by specifying each argument as **stat_loc* rather than *stat_val*. However, that did not follow the conventions of other specifications in this volume of POSIX.1-2017 or traditional usage. It also could have implied that the argument to the macro must literally be **stat_loc*; in fact, that value can be stored or passed as an argument to other functions before being interpreted by these macros.

The extension that affects *wait*() and *waitpid*() and is common in historical implementations is the *ptrace*() function. It is called by a child process and causes that child to stop and return a *status* that appears identical to the *status* indicated by WIFSTOPPED. The *status* of *ptrace*() children is traditionally returned regardless of the WUNTRACED flag (or by the *wait*() function). Most applications do not need to concern themselves with such extensions because they have control over what extensions they or their children use. However, applications, such as command interpreters, that invoke arbitrary processes may see this behavior when those arbitrary processes misuse such extensions.

Implementations that support **core** file creation or other implementation-defined actions on termination of some processes traditionally provide a bit in the *status* returned by *wait*() to indicate that such actions have occurred.

Allowing the *wait*() family of functions to discard a pending SIGCHLD signal that is associated with a successfully waited-for child process puts them into the *sigwait*() and *sigwaitinfo*() category with respect to SIGCHLD.

This definition allows implementations to treat a pending SIGCHLD signal as accepted by the process in *wait*(), with the same meaning of ``accepted'' as when that word is applied to the *sigwait*() family of functions.

Allowing the *wait*() family of functions to behave this way permits an implementation to be able to deal precisely with SIGCHLD signals.

In particular, an implementation that does accept (discard) the SIGCHLD signal can make the following guarantees regardless of the queuing depth of signals in general (the list of waitable children can hold the SIGCHLD queue):

1. If a SIGCHLD signal handler is established via *sigaction*() without the SA_RESETHAND flag, SIGCHLD signals can be accurately counted; that is, exactly one SIGCHLD signal will be delivered to or accepted by the process for every child process that terminates.

2. A single *wait*() issued from a SIGCHLD signal handler can be guaranteed to return immediately with status information for a child process.

3. When SA_SIGINFO is requested, the SIGCHLD signal handler can be guaranteed to receive a non-null pointer to a **siginfo_t** structure that describes a child process for which a wait via *waitpid*() or *waitid*() will not block or fail.

4. The *system*() function will not cause the SIGCHLD handler of a process to be called as a result of the *fork*()/*exec* executed within *system*() because *system*() will accept the SIGCHLD signal when it performs a *waitpid*() for its child process. This is a desirable behavior of *system*() so that it can be used in a library without causing side-effects to the application linked with the library.

An implementation that does not permit the *wait*() family of functions to accept (discard) a pending SIGCHLD signal associated with a successfully waited-for child, cannot make the guarantees described above for the following reasons:

Guarantee #1
: Although it might be assumed that reliable queuing of all SIGCHLD signals generated by the system can make this guarantee, the counter-example is the case of a process that blocks SIGCHLD and performs an indefinite loop of *fork*()/*wait*() operations. If the implementation supports queued signals, then eventually the system will run out of memory for the queue. The guarantee cannot be made because there must be some limit to the depth of queuing.

Guarantees #2 and #3
: These cannot be guaranteed unless the *wait*() family of functions accepts the SIGCHLD signal. Otherwise, a *fork*()/*wait*() executed while SIGCHLD is blocked (as in the *system*() function) will result in an invocation of the handler when SIGCHLD is unblocked, after the process has

disappeared.

Guarantee #4
Although possible to make this guarantee, *system*() would have to set the SIGCHLD handler to SIG_DFL so that the SIGCHLD signal generated by its *fork*() would be discarded (the SIGCHLD default action is to be ignored), then restore it to its previous setting. This would have the undesirable side-effect of discarding all SIGCHLD signals pending to the process.

FUTURE DIRECTIONS top

None.

SEE ALSO top

Section 2.13, *Status Information*, exec(1p), exit(3p), fork(3p), system(3p), waitid(3p)

The Base Definitions volume of POSIX.1-2017, *Section 4.12*, *Memory Synchronization*, signal.h(0p), sys_wait.h(0p)

COPYRIGHT top

Pages that refer to this page: signal.h(0p), sys_wait.h(0p), sh(1p), wait(1p), abort(3p), exit(3p), _Exit(3p), getrusage(3p), kill(3p), pclose(3p), popen(3p), posix_spawn(3p), pthread_join(3p), sigaction(3p), sighold(3p), system(3p), times(3p), waitid(3p), waitpid(3p)