NAME

exec, execl, execv, execle, execve, execlp, execvp – execute a file

SYNOPSIS

#include <unistd.h>

int execl(const char *_path_, const char *_arg0_, ... /*, (char *)0 */);
int execle(const char *_path_, const char *_arg0_, ... /*,
(char *)0, char *const _envp_[]*/);
int execlp(const char *_file_, const char *_arg0_, ... /*, (char *)0 */);
int execv(const char *_path_, char *const _argv_[]);
int execve(const char *_path_, char *const _argv_[], char *const _envp_[]);
int execvp(const char *_file_, char *const _argv_[]);

DESCRIPTION

Each of the functions in the **exec** family overlays a new process image on an old process. The new process image is constructed from an ordinary, executable file. This file is either an executable object file, or a file of data for an interpreter. There can be no return from a successful call to one of these functions because the calling process image is overlaid by the new process image.

When a C program is executed, it is called as follows:

int main (int argc, char *argv[], char *envp[]);

where *argc* is the argument count, *argv* is an array of character pointers to the arguments themselves, and *envp* is an array of character pointers to the environment strings. As indicated, *argc* is at least one, and the first member of the array points to a string containing the name of the file.

The arguments *arg0*, ..., *argn* point to null-terminated character strings. These strings constitute the argument list available to the new process image. Conventionally at least *arg0* should be present. The *arg0* argument points to a string that is the same as *path* (or the last component of *path*). The list of argument strings is terminated by a **(char *)0** argument.

The *argv* argument is an array of character pointers to null-terminated strings. These strings constitute the argument list available to the new process image. By convention, *argv* must have at least one member, and it should point to a string that is the same as *path* (or its last component). The *argv* argument is terminated by a null pointer.

The *path* argument points to a path name that identifies the new process file.

The *file* argument points to the new process file. If *file* does not contain a slash character, the path prefix for this file is obtained by a search of the directories passed in the **PATH** environment variable (see **environ**(5)).

File descriptors open in the calling process remain open in the new process.

Signals that are being caught by the calling process are set to the default disposition in the new process image (see **signal**(3C)). Otherwise, the new process image inherits the signal dispositions of the calling process.

RETURN VALUES

If a function in the **exec** family returns to the calling process, an error has occurred; the return value is **–1** and **errno** is set to indicate the error.

NAME

fork — create a new process

SYNOPSIS

#include <unistd.h>

pid_t fork(void);

DESCRIPTION

The *fork*() function shall create a new process. The new process (child process) shall be an exact copy of the calling process (parent process) except as detailed below:

* *... (omitted)*

After *fork*(), both the parent and the child processes shall be capable of executing independently before either one terminates.

RETURN VALUE

Upon successful completion, *fork*() shall return 0 to the child process and shall return the process ID of the child process to the parent process. Both processes shall continue to execute from the *fork*() function. Otherwise, –1 shall be returned to the parent process, no child process shall be created, and *errno* shall be set to indicate the error.

ERRORS

The *fork*() function shall fail if:

EAGAIN

The system lacked the necessary resources to create another process, or the system-imposed limit on the total number of processes under execution system-wide or by a single user {CHILD_MAX} would be exceeded.

The *fork*() function may fail if:

ENOMEM

Insufficient storage space is available.

NAME

fprintf, printf, snprintf, sprintf — print formatted output

SYNOPSIS

#include <stdio.h>

int fprintf(FILE *restrict *stream*, const char *restrict *format*, ...);
int printf(const char *restrict *format*, ...);
int snprintf(char *restrict *s*, size_t *n*,
const char *restrict *format*, ...);
int sprintf(char *restrict *s*, const char *restrict *format*, ...);

DESCRIPTION

The *fprintf*() function shall place output on the named output *stream*. The *printf*() function shall place output on the standard output stream *stdout*. The *sprintf*() function shall place output followed by the null byte, **'\0'**, in consecutive bytes starting at **s*; it is the user's responsibility to ensure that enough space is available.

The *snprintf*() function shall be equivalent to *sprintf*(), with the addition of the *n* argument which states the size of the buffer referred to by *s*. If *n* is zero, nothing shall be written and *s* may be a null pointer. Otherwise, output bytes beyond the *n*-1st shall be discarded instead of being written to the array, and a null byte is written at the end of the bytes actually written into the array.

If copying takes place between objects that overlap as a result of a call to *sprintf*() or *snprintf*(), the results are undefined.

Each of these functions converts, formats, and prints its arguments under control of the *format*. The *format* is a character string, beginning and ending in its initial shift state, if any. The *format* is composed of zero or more directives: *ordinary characters*, which are simply copied to the output stream, and *conversion specifications*, each of which shall result in the fetching of zero or more arguments. The results are undefined if there are insufficient arguments for the *format*. If the *format* is exhausted while arguments remain, the excess arguments shall be evaluated but are otherwise ignored.

Conversions can be applied to the *n*th argument after the *format* in the argument list, rather than to the next unused argument. In this case, the conversion specifier character **%** (see below) is replaced by the sequence "%*n*$", where *n* is a decimal integer in the range [1,{NL_ARGMAX}], giving the position of the argument in the argument list. This feature provides for the definition of format strings that select arguments in an order appropriate to specific languages (see the EXAMPLES section).

The *format* can contain either numbered argument conversion specifications (that is, "%*n*$" and "**m*$"), or unnumbered argument conversion specifications (that is, **%** and *), but not both. The only exception to this is that **%%** can be mixed with the "%*n*$" form. The results of mixing numbered and unnumbered argument specifications in a *format* string are undefined. When numbered argument specifications are used, specifying the *N*th argument requires that all the leading arguments, from the first to the (*N*−*1*)th, are specified in the format string.

In format strings containing the **%** form of conversion specification, each conversion specification uses the first unused argument in the argument list.

Each conversion specification is introduced by the **'%'** character or by the character sequence "%*n*$", after which the following appear in sequence:

* Zero or more *flags* (in any order), which modify the meaning of the conversion specification.
* An optional minimum *field width*.
* An optional *precision* that gives the minimum number of digits to appear for the **d**, **i**, **o**, **u**, **x**, and **X** conversion specifiers
* An optional length modifier that specifies the size of the argument.
* A *conversion specifier* character that indicates the type of conversion to be applied.

The conversion specifiers and their meanings are:

d, i — The **int** argument shall be converted to a signed decimal in the style "[−]*dddd*". The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it shall be expanded with leading zeros. The default precision is 1. The result of converting zero with an explicit precision of zero shall be no characters.

c — The **int** argument shall be converted to an **unsigned char**, and the resulting byte shall be written.

s — The argument shall be a pointer to an array of **char**. Bytes from the array shall be written up to (but not including) any terminating null byte. If the precision is specified, no more than that many bytes shall be written. If the precision is not specified or is greater than the size of the array, the application shall ensure that the array contains a null byte.

p — The argument shall be a pointer to **void**. The value of the pointer is converted to a sequence of printable characters, in an implementation-defined manner.

If a conversion specification does not match one of the above forms, the behavior is undefined. If any argument is not the correct type for the corresponding conversion specification, the behavior is undefined.

In no case shall a nonexistent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field shall be expanded to contain the conversion result. Characters generated by *fprintf*() and *printf*() are printed as if *fputc*() had been called.

The last data modification and last file status change timestamps of the file shall be marked for update between the call to a successful execution of *fprintf*() or *printf*() and the next successful completion of a call to *fflush*() or *fclose*() on the same stream or a call to *exit*() or *abort*()

RETURN VALUE

Upon successful completion, the *dprintf*(), *fprintf*(), and *printf*() functions shall return the number of bytes transmitted.

Upon successful completion, the *sprintf*() function shall return the number of bytes written to *s*, excluding the terminating null byte.

Upon successful completion, the *snprintf*() function shall return the number of bytes that would be written to *s* had *n* been sufficiently large excluding the terminating null byte.

If an output error was encountered, these functions shall return a negative value and set *errno* to indicate the error.

If the value of *n* is zero on a call to *snprintf*(), nothing shall be written, the number of bytes that would have been written had *n* been sufficiently large excluding the terminating null shall be returned, and *s* may be a null pointer.

ERRORS

For the conditions under which *fprintf*(), and *printf*() fail and may fail, refer to *fputc*( ) or *fputwc*( ).

In addition, all forms of *fprintf*() shall fail if:

EILSEQ
A wide-character code that does not correspond to a valid character has been detected.

EOVERFLOW
The value to be returned is greater than {INT_MAX}.

The *fprintf*(), and *printf*() functions may fail if:

ENOMEM
Insufficient storage space is available.

The *snprintf*() function shall fail if:

EOVERFLOW
The value of *n* is greater than {INT_MAX}.

NAME

malloc — a memory allocator

SYNOPSIS

#include <stdlib.h>

void *malloc(size_t *size*);

DESCRIPTION

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 *malloc*() function shall allocate unused space for an object whose size in bytes is specified by *size* and whose value is unspecified.

The order and contiguity of storage allocated by successive calls to *malloc*() is unspecified. The pointer returned if the allocation succeeds shall be suitably aligned so that it may be assigned to a pointer to any type of object and then used to access such an object in the space allocated (until the space is explicitly freed or reallocated). Each such allocation shall yield a pointer to an object disjoint from any other object. The pointer returned points to the start (lowest byte address) of the allocated space. If the space cannot be allocated, a null pointer shall be returned. If the size of the space requested is 0, the behavior is implementation-defined: either a null pointer shall be returned, or the behavior shall be as if the size were some nonzero value, except that the behavior is undefined if the returned pointer is used to access an object.

RETURN VALUE

Upon successful completion with *size* not equal to 0, *malloc*() shall return a pointer to the allocated space. If *size* is 0, either:

* A null pointer shall be returned and *errno* may be set to an implementation-defined value, or
* A pointer to the allocated space shall be returned. The application shall ensure that the pointer is not used to access an object.

Otherwise, it shall return a null pointer and set *errno* to indicate the error.

ERRORS

The *malloc*() function shall fail if:

ENOMEM
Insufficient storage space is available.

The following sections are informative.

EXAMPLES

None.

APPLICATION USAGE

None.

RATIONALE

None.

FUTURE DIRECTIONS

None.

SEE ALSO

calloc( ), *free*( ), *getrlimit*( ), *posix_memalign*( ), *realloc*( )

The Base Definitions volume of POSIX.1-2017, **<stdlib.h>**

NAME

stpcpy, strcpy — copy a string and return a pointer to the end of the result

SYNOPSIS

#include <string.h>

char *stpcpy(char *restrict *s1*, const char *restrict *s2*);
char *strcpy(char *restrict *s1*, const char *restrict *s2*);

DESCRIPTION

The *stpcpy*() and *strcpy*() functions shall copy the string pointed to by *s2* (including the terminating NUL character) into the array pointed to by *s1*.

If copying takes place between objects that overlap, the behavior is undefined.

RETURN VALUE

The *stpcpy*() function shall return a pointer to the terminating NUL character copied into the *s1* buffer.

The *strcpy*() function shall return *s1*.

No return values are reserved to indicate an error.

ERRORS

No errors are defined.

NAME

strdup, strndup — duplicate a specific number of bytes from a string

SYNOPSIS

#include <string.h>

char *strdup(const char *s);
char *strndup(const char *s, size_t size);

DESCRIPTION

The *strdup*() function shall return a pointer to a new string, which is a duplicate of the string pointed to by *s*. The returned pointer can be passed to *free*(). A null pointer is returned if the new string cannot be created.

The *strndup*() function shall be equivalent to the *strdup*() function, duplicating the provided *s* in a new block of memory allocated as if by using *malloc*(), with the exception being that *strndup*() copies at most *size* plus one bytes into the newly allocated memory, terminating the new string with a NUL character. If the length of *s* is larger than *size*, only *size* bytes shall be duplicated. If *size* is larger than the length of *s*, all bytes in *s* shall be copied into the new memory buffer, including the terminating NUL character. The newly created string shall always be properly terminated.

RETURN VALUE

The *strdup*() function shall return a pointer to a new string on success. Otherwise, it shall return a null pointer and set *errno* to indicate the error.

Upon successful completion, the *strndup*() function shall return a pointer to the newly allocated memory containing the duplicated string. Otherwise, it shall return a null pointer and set *errno* to indicate the error.

ERRORS

These functions shall fail if:

ENOMEM
Storage space available is insufficient.

NAME

strlen, strnlen — get length of fixed size string

SYNOPSIS

#include <string.h>

size_t strlen(const char *s);
size_t strnlen(const char *s, size_t *maxlen*);

DESCRIPTION

For *strlen*(): 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 *strlen*() function shall compute the number of bytes in the string to which *s* points, not including the terminating NUL character.

The *strnlen*() function shall compute the smaller of the number of bytes in the array to which *s* points, not including any terminating NUL character, or the value of the *maxlen* argument. The *strnlen*() function shall never examine more than *maxlen* bytes of the array pointed to by *s*.

RETURN VALUE

The *strlen*() function shall return the length of *s*; no return value shall be reserved to indicate an error.

The *strnlen*() function shall return the number of bytes preceding the first null byte in the array to which *s* points, if *s* contains a null byte within the first *maxlen* bytes; otherwise, it shall return *maxlen*.

ERRORS

No errors are defined.

The following sections are informative.

EXAMPLES

Getting String Lengths

The following example sets the maximum length of *key* and *data* by using *strlen*() to get the lengths of those strings.

```
#include <string.h>
...
struct element {
    char *key;
    char *data;
};
...
char *key, *data;
int len;

*keylength = *datalength = 0;
...
if ((len = strlen(key)) > *keylength)
    *keylength = len;
if ((len = strlen(data)) > *datalength)
    *datalength = len;
...
```

NAME

strtok, strtok_r — split string into tokens

SYNOPSIS

#include <string.h>

char *strtok(char *restrict *s*, const char *restrict *sep*);
char *strtok_r(char *restrict *s*, const char *restrict *sep*,
char **restrict *state*);

DESCRIPTION

For *strtok*(): 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.

A sequence of calls to *strtok*() breaks the string pointed to by *s* into a sequence of tokens, each of which is delimited by a byte from the string pointed to by *sep*. The first call in the sequence has *s* as its first argument, and is followed by calls with a null pointer as their first argument. The separator string pointed to by *sep* may be different from call to call.

The first call in the sequence searches the string pointed to by *s* for the first byte that is *not* contained in the current separator string pointed to by *sep*. If no such byte is found, then there are no tokens in the string pointed to by *s* and *strtok*() shall return a null pointer. If such a byte is found, it is the start of the first token.

The *strtok*() function then searches from there for a byte that *is* contained in the current separator string. If no such byte is found, the current token extends to the end of the string pointed to by *s*, and subsequent searches for a token shall return a null pointer. If such a byte is found, it is overwritten by a NUL character, which terminates the current token. The *strtok*() function saves a pointer to the following byte, from which the next search for a token shall start.

Each subsequent call, with a null pointer as the value of the first argument, starts searching from the saved pointer and behaves as described above.

The implementation shall behave as if no function defined in this volume of POSIX.1-2017 calls *strtok*().

The *strtok*() function need not be thread-safe.

The *strtok_r*() function shall be equivalent to *strtok*(), except that *strtok_r*() shall be thread-safe and the argument *state* points to a user-provided pointer that allows *strtok_r*() to maintain state between calls which scan the same string. The application shall ensure that the pointer pointed to by *state* is unique for each string (*s*) being processed concurrently by *strtok_r*() calls. The application need not initialize the pointer pointed to by *state* to any particular value. The implementation shall not update the pointer pointed to by *state* to point (directly or indirectly) to resources, other than within the string *s*, that need to be freed or released by the caller.

RETURN VALUE

Upon successful completion, *strtok*() shall return a pointer to the first byte of a token. Otherwise, if there is no token, *strtok*() shall return a null pointer.

The *strtok_r*() function shall return a pointer to the token found, or a null pointer when no token is found.

ERRORS

No errors are defined.

NAME

strtol, strtoll — convert a string to a long integer

SYNOPSIS

#include <stdlib.h>

long strtol(const char *restrict *nptr*, char **restrict *endptr*, int *base*);
long long strtoll(const char *restrict *nptr*, char **restrict *endptr*,
int *base*)

DESCRIPTION

These functions shall convert the initial portion of the string pointed to by *nptr* to a type **long** and **long long** representation, respectively. First, they decompose the input string into three parts:

1. An initial, possibly empty, sequence of white-space characters (as specified by *isspace*())
2. A subject sequence interpreted as an integer represented in some radix determined by the value of *base*
3. A final string of one or more unrecognized characters, including the terminating NUL character of the input string.

Then they shall attempt to convert the subject sequence to an integer, and return the result.

If the value of *base* is 0, the expected form of the subject sequence is that of a decimal constant, octal constant, or hexadecimal constant, any of which may be preceded by a **'+'** or **'–'** sign. A decimal constant begins with a non-zero digit, and consists of a sequence of decimal digits. An octal constant consists of the prefix **'0'** optionally followed by a sequence of the digits **'0'** to **'7'** only. A hexadecimal constant consists of the prefix 0x or 0X followed by a sequence of the decimal digits and letters **'a'** (or **'A'**) to **'f'** (or **'F'**) with values 10 to 15 respectively.

If the value of *base* is between 2 and 36, the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by *base*, optionally preceded by a **'+'** or **'–'** sign. The letters from **'a'** (or **'A'**) to **'z'** (or **'Z'**) inclusive are ascribed the values 10 to 35; only letters whose ascribed values are less than that of *base* are permitted. If the value of *base* is 16, the characters 0x or 0X may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first non-white-space character that is of the expected form. The subject sequence shall contain no characters if the input string is empty or consists entirely of white-space characters, or if the first non-white-space character is other than a sign or a permissible letter or digit.

If the subject sequence has the expected form and the value of *base* is 0, the sequence of characters starting with the first digit shall be interpreted as an integer constant. If the subject sequence has the expected form and the value of *base* is between 2 and 36, it shall be used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a <hyphen-minus>, the value resulting from the conversion shall be negated. A pointer to the final string shall be stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

In other than the C or POSIX locale, additional locale-specific subject sequence forms may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of *nptr* shall be stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

These functions shall not change the setting of *errno* if successful.

Since 0, {LONG_MIN} or {LLONG_MIN}, and {LONG_MAX} or {LLONG_MAX} are returned on error and are also valid returns on success, an application wishing to check for error situations should set *errno* to 0, then call *strtol*() or *strtoll*(), then check *errno*.

RETURN VALUE

Upon successful completion, these functions shall return the converted value, if any. If no conversion could be performed, 0 shall be returned and *errno* may be set to **[EINVAL]**.

If the value of *base* is not supported, 0 shall be returned and *errno* shall be set to **[EINVAL]**.

If the correct value is outside the range of representable values, {LONG_MIN}, {LONG_MAX}, {LLONG_MIN}, or {LLONG_MAX} shall be returned (according to the sign of the value), and *errno* set to **[ERANGE]**.

ERRORS

These functions shall fail if:

EINVAL
The value of *base* is not supported.

ERANGE
The value to be returned is not representable.

These functions may fail if:

EINVAL
No conversion could be performed.

EXAMPLE

The program shown below demonstrates the use of **strtol**(). The first command-line argument specifies a string from which **strtol**() should parse a number.

Program source

```
#include <stdlib.h>
#include <limits.h>
#include <stdio.h>
#include <errno.h>

int main(int argc, char *argv[])
{
    if (argc < 2) {
        fprintf(stderr, "Usage: %s str\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    errno = 0;    /* To distinguish success/failure after call */
    char *endptr;
    long val = strtol(argv[1], &endptr, 10);

    /* Check for various possible errors */
    if ((errno == ERANGE && (val == LONG_MAX || val == LONG_MIN))
            || (errno != 0 && val == 0)) {
        perror("strtol");
        exit(EXIT_FAILURE);
    }

    if (*endptr != '\0')
        printf("Further characters after number: %s\n", endptr);


    printf("strtol() returned %ld\n", val);
}
```


NAME

time — get time

SYNOPSIS

#include <time.h>

time_t time(time_t **tloc*);

DESCRIPTION

The *time*() function shall return the value of time in seconds since the Epoch.

The *tloc* argument points to an area where the return value is also stored. If *tloc* is a null pointer, no value is stored.

RETURN VALUE

Upon successful completion, *time*() shall return the value of time.

NAME

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

SYNOPSIS

#include <sys/wait.h>

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

DESCRIPTION

The *wait*() and *waitpid*() functions shall obtain 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.

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*>).

...

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.

RETURN VALUE

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

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.

APPLICATION USAGE

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.