NAME

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

SYNOPSIS

#include <unistd.h>

int execl(const char ***_path_, const char ***_arg0_**, …, const char ***_argn_**, char *** /*NULL*/**);**

int execv(const char ***_path_, char *const** _argv[ ]_**);**

int execle(const char ***_path_,char *const** _arg0[ ]_**, … , const char ***_argn_**,**
    char * /*NULL*/**, char *const** _envp[ ]_**);**

int execve (const char ***_path_, char *const** _argv[ ]_ **char *const** _envp[ ]_**);**

int execlp (const char ***_file_, const char ***_arg0_**, …, const char ***_argn_**, char *** /*NULL*/**);**

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

calloc, malloc, free, realloc – Allocate and free dynamic memory

SYNOPSIS

#include <stdlib.h>

void *calloc(size_t _nmemb_**, size_t** _size_**);**
void *malloc(size_t _size_**);**
void free(void _*ptr_**);**
void *realloc(void _*ptr_**, size_t** _size_**);**

DESCRIPTION

calloc() allocates memory for an array of _nmemb_ elements of _size_ bytes each and returns a pointer to the allocated memory. The memory is set to zero.

malloc() allocates _size_ bytes and returns a pointer to the allocated memory. The memory is not cleared.

free() frees the memory space pointed to by _ptr_, which must have been returned by a previous call to **malloc()**, **calloc()** or **realloc()**. Otherwise, or if **free(**_ptr_**)** has already been called before, undefined behaviour occurs. If _ptr_ is **NULL**, no operation is performed.

realloc() changes the size of the memory block pointed to by _ptr_ to _size_ bytes. The contents will be unchanged to the minimum of the old and new sizes; newly allocated memory will be uninitialized. If _ptr_ is **NULL**, the call is equivalent to **malloc(size)**; if size is equal to zero, the call is equivalent to **free(**_ptr_**)**. Unless _ptr_ is **NULL**, it must have been returned by an earlier call to **malloc()**, **calloc()** or **realloc()**.

RETURN VALUE

For **calloc()** and **malloc()**, the value returned is a pointer to the allocated memory, which is suitably aligned for any kind of variable, or **NULL** if the request fails.

free() returns no value.

realloc() returns a pointer to the newly allocated memory, which is suitably aligned for any kind of variable and may be different from _ptr_, or **NULL** if the request fails. If _size_ was equal to 0, either NULL or a pointer suitable to be passed to _free_() is returned. If **realloc()** fails the original block is left untouched - it is not freed or moved.

CONFORMING TO

ANSI-C

SEE ALSO

brk(2), **posix_memalign**(3)

NAME

opendir – open a directory / readdir – read a directory

SYNOPSIS

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

#include <dirent.h>

DIR *opendir(const char *name);

struct dirent *readdir(DIR *dir);
```

DESCRIPTION opendir

The **opendir()** function opens a directory stream corresponding to the directory *name*, and returns a pointer to the directory stream. The stream is positioned at the first entry in the directory.

RETURN VALUE

The **opendir()** function returns a pointer to the directory stream or NULL if an error occurred.

DESCRIPTION readdir

The **readdir()** function returns a pointer to a dirent structure representing the next directory entry in the directory stream pointed to by *dir*. It returns NULL on reaching the end-of-file or if an error occurred. It is safe to use **readdir()** inside threads if the pointers passed as *dir* are created by distinct calls to **opendir()**.

The data returned by **readdir()** is overwritten by subsequent calls to **readdir()** for the **same** directory stream.

The *dirent* structure is defined as follows:

```
struct dirent {
    long            d_ino;         /* inode number */
    off_t           d_off;         /* offset to the next dirent */
    unsigned short  d_reclen;      /* length of this record */
    unsigned char   d_type;        /* type of file; not supported by all filesystem types */
    char            d_name[256];   /* filename */
};
```

RETURN VALUE

The **readdir()** function returns a pointer to a dirent structure, or NULL if an error occurs or end-of-file is reached.

ERRORS

EACCES
Permission denied.

ENOENT
Directory does not exist, or *name* is an empty string.

ENOTDIR
name is not a directory.

NAME

qsort – sorts an array

SYNOPSIS

```
#include <stdlib.h>

void qsort(void *base, size_t nmemb, size_t size,
      int(*compar)(const void *, const void *));
```

DESCRIPTION

The **qsort**() function sorts an array with *nmemb* elements of size *size*. The *base* argument points to the start of the array.

The contents of the array are sorted in ascending order according to a comparison function pointed to by *compar*, which is called with two arguments that point to the objects being compared.

The comparison function must return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second. If two members compare as equal, their order in the sorted array is undefined.

RETURN VALUE

The **qsort**() function returns no value.

SEE ALSO

sort(1), **alphasort**(3), **strcmp**(3), **versionsort**(3)

ATTRIBUTES

Multithreading (see pthreads(7))

The **qsort**() function is thread-safe if the comparison function *compar* does not access any global variables.

NAME

printf, fprintf, sprintf, snprintf, vprintf, vfprintf, vsprintf, vsnprintf – formatted output conversion

SYNOPSIS

```
#include <stdio.h>

int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int sprintf(char *str, const char *format, ...);
int snprintf(char *str, size_t size, const char *format, ...);
 ...
```

DESCRIPTION

The functions in the **printf**() family produce output according to a *format* as described below. The function **printf**() writes output to *stdout*, the standard output stream; **fprintf**() writes output to the given output *stream*; **sprintf**() and **snprintf**(), write to the character string *str*.

The function **snprintf**() writes at most *size* bytes (including the trailing null byte ('\0')) to *str*.

These functions write the output under the control of a *format* string that specifies how subsequent arguments (or arguments accessed via the variable-length argument facilities of **stdarg**(3)) are converted for output.

Return value

Upon successful return, these functions return the number of characters printed (not including the trailing '\0' used to end output to strings).

The functions **snprintf**() and **vsnprintf**() do not write more than *size* bytes (including the trailing '\0'). If the output was truncated due to this limit then the return value is the number of characters (not including the trailing '\0') which would have been written to the final string if enough space had been available. Thus, a return value of *size* or more means that the output was truncated.

If an output error is encountered, a negative value is returned.

Format of the format string

The format string is a character string, beginning and ending in its initial shift state, if any. The format string is composed of zero or more directives: ordinary characters (not **%**), which are copied unchanged to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments. Each conversion specification is introduced by the character **%**, and ends with a *conversion specifier*. In between there may be (in this order) zero or more *flags*, an optional minimum *field width*, an optional *precision* and an optional *length modifier*.

The conversion specifier

A character that specifies the type of conversion to be applied. An example for a conversion specifier is:

o, **u**, **x**, **X**
The *unsigned int* argument is converted to unsigned octal (**o**), unsigned decimal (**u**), or unsigned hexadecimal (**x** and **X**) notation.

s
The *const char ** argument is expected to be a pointer to an array of character type (pointer to a string). Characters from the array are written up to (but not including) a terminating null byte ('\0'); if a precision is specified, no more than the number specified are written. If a precision is given, no null byte need be present; if the precision is not specified, or is greater than the size of the array, the array must contain a terminating null byte.

SEE ALSO

printf(1), **asprintf**(3), **dprintf**(3), **scanf**(3), **setlocale**(3), **wcrtomb**(3), **wprintf**(3), **locale**(5)

NAME

stat, fstat, lstat – get file status

SYNOPSIS

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int stat(const char *path, struct stat *buf);
int fstat(int fd, struct stat *buf);
int lstat(const char *path, struct stat *buf);
```

DESCRIPTION

These functions return information about a file. No permissions are required on the file itself, but — in the case of **stat**() and **lstat**() — execute (search) permission is required on all of the directories in *path* that lead to the file.

stat() stats the file pointed to by *path* and fills in *buf*.

lstat() is identical to **stat**(), except that if *path* is a symbolic link, then the link itself is stat-ed, not the file that it refers to.

fstat() is identical to **stat**(), except that the file to be stat-ed is specified by the file descriptor *fd*.

All of these system calls return a *stat* structure, which contains the following fields:

```
struct stat {
    dev_t       st_dev;      /* ID of device containing file  */
    ino_t       st_ino;      /* inode number                  */
    mode_t      st_mode;     /* protection                    */
    nlink_t     st_nlink;    /* number of hard links          */
    uid_t       st_uid;      /* user ID of owner              */
    gid_t       st_gid;      /* group ID of owner             */
    dev_t       st_rdev;     /* device ID (if special file)   */
    off_t       st_size;     /* total size, in bytes          */
    blksize_t   st_blksize;  /* blocksize for file system I/O */
    blkcnt_t    st_blocks;   /* number of blocks allocated    */
    time_t      st_atime;    /* time of last access           */
    time_t      st_mtime;    /* time of last modification     */
    time_t      st_ctime;    /* time of last status change    */
};
```

The *st_dev* field describes the device on which this file resides.

The *st_rdev* field describes the device that this file (inode) represents.

The *st_size* field gives the size of the file (if it is a regular file or a symbolic link) in bytes. The size of a symlink is the length of the pathname it contains, without a trailing null byte.

The *st_blocks* field indicates the number of blocks allocated to the file, 512-byte units. (This may be smaller than *st_size*/512 when the file has holes.)

The *st_blksize* field gives the "preferred" blocksize for efficient file system I/O. (Writing to a file in smaller chunks may cause an inefficient read-modify-rewrite.)

Not all of the Linux file systems implement all of the time fields. Some file system types allow mounting in such a way that file accesses do not cause an update of the *st_atime* field. (See "noatime" in **mount**(8).)

The field *st_atime* is changed by file accesses, for example, by **execve**(2), **mknod**(2), **pipe**(2), **utime**(2) and

read(2) (of more than zero bytes). Other routines, like **mmap**(2), may or may not update *st_atime*.

The field *st_mtime* is changed by file modifications, for example, by **mknod**(2), **truncate**(2), **utime**(2) and **write**(2) (of more than zero bytes). Moreover, *st_mtime* of a directory is changed by the creation or deletion of files in that directory. The *st_mtime* field is *not* changed for changes in owner, group, hard link count, or mode.

The field *st_ctime* is changed by writing or by setting inode information (i.e., owner, group, link count, mode, etc.).

The following POSIX macros are defined to check the file type using the *st_mode* field:

S_ISREG(m)	is it a regular file?
S_ISDIR(m)	directory?
S_ISCHR(m)	character device?
S_ISBLK(m)	block device?
S_ISFIFO(m)	FIFO (named pipe)?
S_ISLNK(m)	symbolic link? (Not in POSIX.1-1996.)
S_ISSOCK(m)	socket? (Not in POSIX.1-1996.)

The following mask values are defined for the file mode component of the *st_mode* field:

S_IRWXU	00700	owner has read, write, and execute permission
S_IRUSR	00400	owner has read permission
S_IWUSR	00200	owner has write permission
S_IXUSR	00100	owner has execute permission

RETURN VALUE

On success, zero is returned. On error, −1 is returned, and *errno* is set appropriately.

ERRORS

EACCES
Search permission is denied for one of the directories in the path prefix of *path*. (See also **path_resolution**(7).)

EBADF
fd is bad.

EFAULT
Bad address.

ELOOP
Too many symbolic links encountered while traversing the path.

ENAMETOOLONG
File name too long.

ENOENT
A component of the path *path* does not exist, or the path is an empty string.

ENOMEM
Out of memory (i.e., kernel memory).

ENOTDIR
A component of the path is not a directory.

SEE ALSO

access(2), **chmod**(2), **chown**(2), **fstatat**(2), **readlink**(2), **utime**(2), **capabilities**(7), **symlink**(7)

NAME

strcat, strchr, strcmp, strcpy, strdup, strlen, strncat, strncmp, strncpy, strstr, strtok − string operations

SYNOPSIS

#include <string.h>

char *strcat(char ****dest*, const char ****src***);**
Append the string *src* to the string *dest*, returning a pointer *dest*.

char *strchr(const char ****s*, int** *c***);**
Return a pointer to the first occurrence of the character *c* in the string *s*.

int strcmp(const char ****s1*, const char ****s2***);**
Compare the strings *s1* with *s2*. It returns an integer less than, equal to, or greater than zero if *s1* is found, respectively, to be less than, to match, or be greater than *s2*.

char *strcpy(char ****dest*, const char ****src***);**
Copy the string *src* to *dest*, returning a pointer to the start of *dest*.

char *strdup(const char ****s*);**
Return a duplicate of the string *s* in memory allocated using **malloc**(3).

size_t strlen(const char ****s*);**
Return the length of the string *s*.

char *strncat(char ****dest*, const char ****src***, size_t** *n***);**
Append at most *n* characters from the string *src* to the string *dest*, returning a pointer to *dest*.

int strncmp(const char ****s1*, const char ****s2***, size_t** *n***);**
Compare at most *n* bytes of the strings *s1* and *s2*. It returns an integer less than, equal to, or greater than zero if *s1* is found, respectively, to be less than, to match, or be greater than *s2*.

char *strncpy(char ****dest*, const char ****src***, size_t** *n***);**
Copy at most *n* bytes from string *src* to *dest*, returning a pointer to the start of *dest*.

char *strstr(const char ****haystack*, const char ****needle***);**
Find the first occurrence of the substring *needle* in the string *haystack*, returning a pointer to the found substring.

char *strtok(char ****s*, const char ****delim***);**
Extract tokens from the string *s* that are delimited by one of the bytes in *delim*.

DESCRIPTION

The string functions perform operations on null-terminated strings.

NAME

time – get time in seconds

SYNOPSIS

#include <time.h>

time_t time(time_t ***tloc*);**

DESCRIPTION

time() returns the time as the number of seconds since the Epoch, 1970-01-01 00:00:00 +0000 (UTC).

If *tloc* is non-NULL, the return value is also stored in the memory pointed to by *tloc*.

When *tloc* is NULL, the call cannot fail.

RETURN VALUE

On success, the value of time in seconds since the Epoch is returned. On error, *((time_t) −1)* is returned, and *errno* is set appropriately.

NAME

waitpid – wait for child process to change state

SYNOPSIS

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

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

DESCRIPTION

waitpid( ) suspends the calling process until one of its children changes state; if a child process changed state prior to the call to **waitpid( )**, return is immediate. *pid* specifies a set of child processes for which status is requested.

If *pid* is equal to **(pid_t)−1**, status is requested for any child process.

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

If *pid* is equal to **(pid_t)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*.

If **waitpid( )** returns because the status of a child process is available, then that status may be evaluated with the macros defined by **wstat**(5)**.** If the calling process had specified a non-zero value of *stat_loc*, the status of the child process will be stored in the location pointed to by *stat_loc*.

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

WCONTINUED	The status of any continued child process specified by *pid*, whose status has not been reported since it continued, is also reported to the calling process.
WNOHANG	**waitpid( )** will not suspend execution of the calling process if status is not immediately available for one of the child processes specified by *pid*.
WNOWAIT	Keep the process whose status is returned in *stat_loc* in a waitable state. The process may be waited for again with identical results.

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.

RETURN VALUES

If **waitpid( )** returns because the status of a child process is available, this function returns a value equal to the process ID of the child process for which status is reported. If **waitpid( )** returns due to the delivery of a signal to the calling process, **−1** is returned and **errno** is set to **EINTR**. If this function 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** is returned, and **errno** is set to indicate the error.

ERRORS

waitpid( ) will fail if one or more of the following is true:

ECHILD	The process or process group specified by *pid* does not exist or is not a child of the calling process or can never be in the states specified by *options*.
EINTR	**waitpid( )** was interrupted due to the receipt of a signal sent by the calling process.
EINVAL	An invalid value was specified for *options*.

SEE ALSO

exec(2), **exit**(2), **fork**(2), **sigaction**(2), **wstat**(5)