NAME

opendir – open a directory / readdir – read a directory

SYNOPSIS

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

DIR *opendir(const char *name);
int closedir(DIR *dirp);
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. On error, NULL is returned, and *errno* is set appropriately.

DESCRIPTION closedir

The **closedir()** function closes the directory stream associated with *dirp*. A successful call to **closedir()** also closes the underlying file descriptor associated with *dirp*. The directory stream descriptor *dirpP is not available after this call.*

RETURN VALUE

The **closedir()** function returns 0 on success. On error, −1 is returned, and *errno* is set appropriately.

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 */
    char        d_name[256]; /* filename */
};
```

RETURN VALUE

On success, **readdir**() returns a pointer to a *dirent* structure. (This structure may be statically allocated; do not attempt to **free**(3) it.)

If the end of the directory stream is reached, NULL is returned and *errno* is not changed. If an error occurs, NULL is returned and *errno* is set appropriately. To distinguish end of stream and from an error, set *errno* to zero before calling **readdir**() and then check the value of *errno* if NULL is returned.

ERRORS

EACCES
Permission denied.

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

ENOTDIR
name is not a directory.

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);
```

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

lstat(): _BSD_SOURCE || _XOPEN_SOURCE >= 500

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.)

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

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

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.

EXAMPLES

For one example of use, see the example under **bsearch**(3).

Another example is the following program, which sorts the strings given in its command-line arguments:

```
#include <stdio.h> #include <stdlib.h> #include <string.h>  static int cmpstringp(const void *p1, const void
*p2) {
   /* The actual arguments to this function are "pointers to
      pointers to char", but strcmp(3) arguments are "pointers
      to char", hence the following cast plus dereference. */
   return strcmp(*(const char **) p1, *(const char **) p2); }  int main(int argc, char *argv[]) {
   if (argc < 2) {
       fprintf(stderr, "Usage: %s <string>...\n", argv[0]);
       exit(EXIT_FAILURE);
   }
   qsort(&argv[1], argc − 1, sizeof(char *), cmpstringp);
   for (size_t j = 1; j < argc; j++)
       puts(argv[j]);
   exit(EXIT_SUCCESS); }
```

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

asctime, ctime, gmtime, localtime, mktime, asctime_r, ctime_r, gmtime_r, localtime_r − transform date and time to broken-down time or ASCII

LIBRARY

Standard C library (*libc*, *−lc*)

SYNOPSIS

#include <time.h>

char *asctime(const struct tm ****tm*);**
char *asctime_r(const struct tm *restrict *tm*,
char *buf*[**restrict 26**]);

char *ctime(const time_t ****timep*);**
char *ctime_r(const time_t *restrict *timep*,
char *buf*[**restrict 26**]);

struct tm *gmtime(const time_t ****timep*);**
struct tm *gmtime_r(const time_t *restrict *timep*,
struct tm *restrict *result*);

struct tm *localtime(const time_t ****timep*);**
struct tm *localtime_r(const time_t *restrict *timep*,
struct tm *restrict *result*);

time_t mktime(struct tm ****tm*);**

DESCRIPTION

The **ctime**(), **gmtime**(), and **localtime**() functions all take an argument of data type *time_t*, which represents calendar time. When interpreted as an absolute time value, it represents the number of seconds elapsed since the Epoch, 1970-01-01 00:00:00 +0000 (UTC).

The **asctime**() and **mktime**() functions both take an argument representing broken-down time, which is a representation separated into year, month, day, and so on.

Broken-down time is stored in the structure *tm*, described in **tm**(3type).

The call **ctime**(*t*) is equivalent to **asctime**(**localtime**(*t*)). It converts the calendar time *t* into a null-terminated string of the form

"Wed Jun 30 21:49:08 1993\n"

The abbreviations for the days of the week are "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", and "Sat". The abbreviations for the months are "Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", and "Dec". The return value points to a statically allocated string which might be overwritten by subsequent calls to any of the date and time functions. The function also sets the external variables *tzname*, *timezone*, and *daylight* as if it called **tzset**(3). The reentrant version **ctime_r**() does the same, but stores the string in a user-supplied buffer which should have room for at least 26 bytes. It need not set *tzname*, *timezone*, and *daylight*.

The **gmtime**() function converts the calendar time *timep* to broken-down time representation, expressed in Coordinated Universal Time (UTC). It may return NULL when the year does not fit into an integer. The return value points to a statically allocated struct which might be overwritten by subsequent calls to any of the date and time functions. The **gmtime_r**() function does the same, but stores the data in a user-supplied struct.

The **localtime**() function converts the calendar time *timep* to broken-down time representation, expressed relative to the user's specified timezone. The function also sets the external variables *tzname*, *timezone*, and *daylight* as if it called **tzset**(3). The return value points to a statically allocated struct which might be overwritten by subsequent calls to any of the date and time functions. The **localtime_r**() function does the same, but stores the data in a user-supplied struct. It need not set *tzname*, *timezone*, and *daylight*.

The **asctime**() function converts the broken-down time value *tm* into a null-terminated string with the same

format as **ctime**(). The return value points to a statically allocated string which might be overwritten by subsequent calls to any of the date and time functions. The **asctime_r**() function does the same, but stores the string in a user-supplied buffer which should have room for at least 26 bytes.

The **mktime**() function converts a broken-down time structure, expressed as local time, to calendar time representation. The function ignores the values supplied by the caller in the *tm_wday* and *tm_yday* fields. The value specified in the *tm_isdst* field informs **mktime**() whether or not daylight saving time (DST) is in effect for the time supplied in the *tm* structure: a positive value means DST is in effect; zero means that DST is not in effect; and a negative value means that **mktime**() should (use timezone information and system databases to) attempt to determine whether DST is in effect at the specified time.

The **mktime**() function modifies the fields of the *tm* structure as follows: *tm_wday* and *tm_yday* are set to values determined from the contents of the other fields; if structure members are outside their valid interval, they will be normalized (so that, for example, 40 October is changed into 9 November); *tm_isdst* is set (regardless of its initial value) to a positive value or to 0, respectively, to indicate whether DST is or is not in effect at the specified time. The function also sets the external variables *tzname*, *timezone*, and *daylight* as if it called **tzset**(3).

If the specified broken-down time cannot be represented as calendar time (seconds since the Epoch), **mktime**() returns *(time_t) −1* and does not alter the members of the broken-down time structure.

RETURN VALUE

On success, **gmtime**() and **localtime**() return a pointer to a *struct tm*.

On success, **gmtime_r**() and **localtime_r**() return the address of the structure pointed to by *result*.

On success, **asctime**() and **ctime**() return a pointer to a string.

On success, **asctime_r**() and **ctime_r**() return a pointer to the string pointed to by *buf*.

On success, **mktime**() returns the calendar time (seconds since the Epoch), expressed as a value of type *time_t*.

On error, **mktime**() returns the value *(time_t) −1*. The remaining functions return NULL on error. On error, *errno* is set to indicate the error.

ERRORS

EOVERFLOW
The result cannot be represented.

ATTRIBUTES

For an explanation of the terms used in this section, see **attributes**(7).

Interface	Attribute	Value
asctime()	Thread safety	MT-Unsafe race:asctime locale
asctime_r()	Thread safety	MT-Safe locale
ctime()	Thread safety	MT-Unsafe race:tmbuf race:asctime env locale
ctime_r(), **gmtime_r**(), **localtime_r**(), **mktime**()	Thread safety	MT-Safe env locale
gmtime(), **localtime**()	Thread safety	MT-Unsafe race:tmbuf env locale

NOTES

The four functions **asctime**(), **ctime**(), **gmtime**(), and **localtime**() return a pointer to static data and hence are not thread-safe. The thread-safe versions, **asctime_r**(), **ctime_r**(), **gmtime_r**(), and **localtime_r**(), are specified by SUSv2.

SEE ALSO

date(1), **gettimeofday**(2), **time**(2), **utime**(2), **clock**(3), **difftime**(3), **strftime**(3), **strptime**(3), **timegm**(3), **tzset**(3), **time**(7)