Documentation Updates for APAR PH45182

Table of Contents

XL C/C++ Runtime Library Reference

Chapter 2. Header files

Feature test macros

… …

_LARGE_FILES

… …

The following functions are enabled to operate on z/OS UNIX files of all sizes by expanding

appropriate offset and file size values to a 64-bit value:

creat(), fcntl(), fgetpos(), fopen(), freopen(), fseek(), fseeko(),

fsetpos(), fstat(), fstatat(), ftell(), ftello(), ftruncate(), getrlimit(),

lockf(), lseek(), lstat(), mmap(), open(), read(), setrlimit(),

stat(), truncate(), write()

… …

fcntl.h — POSIX functions for file operations

The fcntl.h header file declares the following POSIX functions for creating, opening, rewriting, and manipulating files.

_POSIX_C_SOURCE 200809L

faccessat() openat() utimensat()

_POSIX_SOURCE

creat() fcntl() open()

_XPLATFORM_SOURCE

futimesat() openat2()

The header also contains these constants:

_XPLATFORM_SOURCE

O_CLOEXEC O_DIRECT

getopt.h — z/OS UNIX function for parsing command-line option

The getopt.h header file contains definitions for command-line option parsing functions.

sched.h — Manipulate and examine process execution scheduling

The sched.h header file declares functions to manipulate and examine process execution scheduling.

_UNIX03_SOURCE

sched_yield()

_XPLATFORM_SOURCE

clone() setns() unshare()

When compiled with SUSV3 thread support (_UNIX03_THREADS or _XOPEN_SOURCE 600), sched.h defines the following symbols:

SCHED_FIFO SCHED_OTHER SCHED_RR

and the sched_param structure.

signal.h — Exception handling

… …

• Flags for the sa_flags field, available in z/OS UNIX only: SA_NOCLDSTOP and _SA_OLD_STYLE.

_POSIX_C_SOURCE 200809L:

• Functions:

psignal()

_XOPEN_SOURCE_EXTENDED 1:

… …

stdio.h — Standard input and output

The stdio.h header file declares functions that deal with standard input and output. One of these functions, fdopen(), is supported only in a POSIX program.

The stdio.h header file also declares these functions:

asprintf() clearerr() clrmemf() dprintf() fclose()

… …

remove() rename() renameat() renameat2() rewind()

scanf() setbuf() setvbuf() sprintf() sscanf()

svc99() tmpfile() tmpnam() ungetc() vasprintf()

vfprintf() vprintf() vsprintf()

… …

sys/epoll.h — I/O event notification facility functions

The sys/epoll.h header file contains definitions for I/O event notification facility functions.

sys/eventfd.h — Standard library functions

The sys/eventfd.h header file contains definiation for the eventfd() functions.

sys/file.h — File manipulation constants

The sys/file.h header file defines file manipulation constants and the following functions:

_XPLATFORM_SOURCE

flock()

sys/inotify.h — Monitor and notify filesystem change functions

The sys/inotify.h header file contains definitions for monitoring file system event functions.

sys/mount.h — File system mount and unmount

The sys/mount.h header contains definitions for mount and unmount file system operations functions.

sys/prctl.h — operations on a process or thread

The sys/prctl.h header file contains definitions for operations on a process or thread.

sys/random.h — random data operations

The sys/random.h header file contains definitions for random data operations, including declarations, constants, and structures used by the following functions:

_XPLATFORM_SOURCE

getrandom()

sys/resource.h — XSI resource operations

The sys/resource.h header file contains definitions for XSI resource operations, including declarations, constants, and structures used by the following functions:

- getpriority()
- getrlimit()
- getrusage()
- prlimit()
- setpriority()
- setrlimit()

sys/socket.h — Sockets definitions

The sys/socket.h header file contains sockets definitions.

The structure sockaddr_storage is exposed by defining the feature test macro _OPEN_SYS_SOCK_IPV6 or _OPEN_SYS_SOCK_EXT3.

Flag O_CLOEXEC and O_DIRECT are exposed by defining the feature test macro _XPLATFORM_SOURCE.

sys/stat.h — z/OS UNIX files and access

The sys/stat.h header file declares the following functions related to z/OS UNIX files and their access:

__chattrat() __chattrat64() chaudit() chmod() creat()

fchaudit() fchmod() fstat() fstat64() fstatat()

fstatat64() lstat() lstat64() mkdir() mkdirat()

mkfifo() mkfifoat() mknod() mknodat() __mount()

mount() __open_stat() __open_stat64() stat()

stat64() umask() umount()

sys/wait.h — Hold processes

The sys/wait.h header file declares the following functions, used for holding processes.

_POSIX_SOURCE:

wait() waitpid()

_XOPEN_SOURCE_EXTENDED 1:

waitid() wait3()

Note: wait3() has been withdrawn in Single UNIX Specification, Version 3.

_XPLATFORM_SOURCE

wait4()

sys/xattr.h — Extended attributes handling

The sys/xattr.h header file contains definitions for extended attributes handling functions.

time.h — Time and date

The time.h header file declares the time and date functions:

asctime() clock() clock_gettime() ctime() difftime()

gmtime() localtime() mktime() nanosleep() strftime()

strptime() time() tzset()

… …

unistd.h — Implementation-specific functions

The unistd.h header file declares a number of implementation-specific functions:

… …

POSIX_C_SOURCE = 2

optarg opterr optind optopt

External Variables

_POSIX_C_SOURCE 200809L

faccessat() fchownat() linkat() readlinkat() symlinkat() unlinkat()

_XOPEN_SOURCE

… …

_XOPEN_SOURCE = 500

… …

vfork()

_XPLATFORM_SOURCE

dup3() getentropy() pipe2() pivot_root() syncfs() sethostname()

The unistd.h header file also defines many symbols to represent configuration variables and implementation features provided. Some of these are used at compile time, while others are used to interrogate the system at run time, using sysconf(), confstr(), pathconf(), or fpathconf().

Chapter 3. Library functions

External variables

… …

optarg

Character pointer used by getopt() and getopt_long() for options parsing variables.

opterr

Error value used by getopt() and getopt_long().

optind

Integer pointer used by getopt() and getopt_long() for options parsing variables.

optopt

Integer pointer used by getopt() and getopt_long() for options parsing variables

… …

accept4() — Accept a new connection on a socket

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _XPLATFORM_SOURCE
#include <sys/socket.h>
int accept4(int sockfd, struct sockaddr *addr, socklen_t *addrlen, int flags);
```

General Description

The accept4() is used by a server to accept a connection request from a client. When a connection is available, the socket created is ready for use to read data from the process that requested the connection. The call accepts the first connection on its queue of pending connections for the given socket *socket*. The accept4() call creates a new socket descriptor with the same properties as *socket* and returns it to the caller. If the queue has no pending connection requests, accep4t() blocks the caller unless *socket* is in nonblocking mode. If no connection requests are queued and *socket* is in nonblocking mode, accept4() returns -1 and sets the error code to EWOULDBLOCK. The new socket descriptor cannot be used to accept new connections. The original socket, *socket*, remains available to accept more connection requests. The accept4() will extend the functionality of accept() by adding capability to set the following flags on accept syscall (SOCK_CLOEXEC and SOCK_NONBLOCK).

Parameter

Description

socket

The socket descriptor.

address

The socket address of the connecting client that is filled in by accept4() before it returns. The format of *address* is determined by the domain that the client resides in. This parameter can be NULL if the caller is not interested in the client address.

address_len

Must initially point to an integer that contains the size in bytes of the storage pointed to by *address*. On return, that integer contains the size required to represent the address of the connecting socket. If this value is larger than the size supplied on input, then the information contained in sockaddr is truncated to the length supplied on input. If *address* is NULL, *address_len* is ignored.

flags

If flags is 0, then accept4() is the same as accept(). The following values can be bitwise ORed in flags to obtain different behavior:

SOCK_NONBLOCK

Set the O_NONBLOCK file status flag on the new open file description. Using this flag saves extra calls to fcntl to achieve the same result.

SOCK_CLOEXEC

Set the close-on-exec (FD_CLOEXEC) flag on the new file descriptor. See the description of the O_CLOEXEC flag in open for reasons why this may be useful.

The *socket* parameter is a stream socket descriptor created with the socket() call. It is usually bound to an address with the bind() call. The listen() call marks the socket as one that accepts connections and allocates a queue to hold pending connection requests. The listen() call places an upper boundary on the size of the queue.

The *address* parameter is a pointer to a buffer into which the connection requester's address is placed. The *address* parameter is optional and can be set to be the NULL pointer. If set to NULL, the requester's address is not copied into the buffer. The exact format of *address*depends on the addressing domain from which the communication request originated. For example, if the connection request originated in the AF_INET domain, *address* points to a **sockaddr_in** structure, or if the connection request originated in the AF_INET6

domain, *address*points to a **sockaddr_in6** structure. The **sockaddr_in** and **sockaddr_in6** structures are defined in **netinet/in.h**. , The *address_len* parameter is used only if the address is not NULL. Before calling accept4(), you must set the integer pointed to by *address_len* to the size of the buffer, in bytes, pointed to by *address*. On successful return, the integer pointed to by *address_len* contains the actual number of bytes copied into the buffer. If the buffer is not large enough to hold the address, up to *address_len* bytes of the requester's address are copied. If the actual length of the address is greater than the length of the supplied **sockaddr**, the stored address is truncated. The **sa_len** member of the store structure contains the length of the untruncated address.

Notes:

1. This call is used only with SOCK_STREAM sockets. There is no way to screen requesters without calling accept4(). The application cannot tell the system the requesters from which it will accept connections. However, the caller can choose to close a connection immediately after discovering the identity of the requester.

A socket can be checked for incoming connection requests using the select() call.

Return Value

If successful, accept4() returns a nonnegative socket descriptor.

If unsuccessful, accept4() returns -1 and sets errno to one of the following values:

Error Code

Description

EAGAIN

If during an accept call that changes identity, the UID of the new identity is already at MAXPROCUID, the accept call fails.

EBADF

The *socket* parameter is not within the acceptable range for a socket descriptor.

EFAULT

Using *address* and *address_len* would result in an attempt to copy the address into a portion of the caller's address space into which information cannot be written.

EINTR

A signal interrupted the accept4() call before any connections were available.

EINVAL

listen() was not called for socket descriptor *socket*, or the flags parameter does not specify a valid value.

EIO

There has been a network or transport failure.

EMFILE

An attempt was made to open more than the maximum number of file descriptors allowed for this process.

EMVSERR

Two consecutive accept calls that cause an identity change are not allowed. The original identity must be restored (close() the socket that caused the identity change) before any further accepts are allowed to change the identity

ENFILE

The maximum number of file descriptors in the system are already open.

ENOBUFS

Insufficient buffer space is available to create the new socket.

ENOTSOCK

The *socket* parameter does not refer to a valid socket descriptor.

EOPNOTSUPP

The socket type of the specified socket does not support accepting connections.

EWOULDBLOCK

The socket descriptor *socket* is in nonblocking mode, and no connections are in the queue.

Related Information

- sys/socket.h — Sockets definitions
- sys/types.h — typedef symbols and structures
- bind() — Bind a name to a socket
- connect() — Connect a socket
- getpeername() — Get the name of the peer connected to a socket
- listen() — Prepare the server for incoming client requests
- socket() — Create a socket
- accept() — Accept a new connection on a socket

asprintf, vasprintf() — Print to allocated string

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _XPLATFORM_SOURCE

#include <stdio.h>

int asprintf(char **strp, const char *fmt, ...);

int vasprintf(char **strp, const char *fmt, va_list ap);
```

General Description

The functions asprintf() and vasprintf() are analogs of sprintf() and vsprintf(), except that they allocate a string large enough to hold the output including the terminating null byte, and return a pointer to it via the first argument. This pointer should be passed to free() to release the allocated storage when it is no longer needed.

Returned value
When successful, these functions return the number of bytes printed, just like sprintf(). If memory allocation wasn't possible, or some other error occurs, these functions will return -1, and the contents of strp is undefined.

Related Information

- “fprintf(), printf(), sprintf() — Format and write data” on page XX
- “vfprintf() — Format and print data to stream ” on page XX

__chattrat(), __chattrat64() — Change the attributes of a file or directory

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

__chattrat:

```
#define _OPEN_SYS_FILE_EXT 1
#include <sys/stat.h>
#include <fcntl.h>
int __chattrat(int dirfd, char *pathname, attrib_t *attributes,
               int attributes_len, int flags);
```

__chattrat64:

```
#define _LARGE_TIME_API
#define _OPEN_SYS_FILE_EXT 1
#include <fcntl.h>
#include <sys/stat.h>
int __chattrat64(int dirfd, char *pathname, attrib64_t *attributes,
                 int attributes_len, int flags);
```

Compile requirement:

Use of the __chattrat64 function requires the long long data type. For more

information on how to make the long long data type available, see z/OS XL C/C++ Language Reference.

General Description

The __chattrat() function modifies the attributes that are associated with a file. It can be used to

change the mode, owner, access time, modification time, change time, reference time, audit flags, general attribute flags, file tag, and file format and size. The file to be impacted is defined by the pathname argument.

The __chattrat() function operates in exactly the same way as __chattr(). If the pathname given in pathname is relative, then it is interpreted relative to the directory referred to by the file descriptor dirfd (rather than relative to the current working directory of the calling process, as is done by __chattr() for a relative pathname).

If pathname is relative and dirfd is the special value AT_FDCWD, then pathname is interpreted relative to the current working directory of the calling process (like __chattr).

If pathname is absolute, then dirfd is ignored.

The attributes argument is the address of an attrib_t structure that is used to identify the attributes to be modified and the new values desired. The attrib_t type is an f_attributes structure as defined in <sys/stat.h> for use with the __chattrat() function. For proper behavior the user should ensure that this structure has been initialized to zeros before it is populated. Available elements of the f_attributes structure are defined in Table 20(Struct f_attributes Element Descriptions) on page xxx.

flags can either be 0, or include the following flag:

AT_SYMLINK_NOFOLLOW

If pathname is a symbolic link, do not dereference it: instead operate on the link itself (By default, __chattrat() dereferences symbolic links.)

The __chattrat64() function behaves exactly like __chattrat() except __chattrat64() uses struct

attrib64_t instead of struct attrib_t to support time beyond 03:14:07 UTC on January 19, 2038.

The type of attrib64_t is an f_attributes64 structure as defined in <sys/stat.h> for use with the

__chattrat64() function.

The use of the f_attributes64 structure is exactly like f_attributes structure except that the types

of att_atime, att_mtime, att_ctime, and att_reftime are time64_t.

Returned Value

If successful, __chattrat() returns zero.

If unsuccessful, __chattrat() returns -1 and set errno to one of the following values:

Error Code

Description

EACCES

The calling process did not have appropriate permissions. Possible reasons include:

• The calling process was attempting to set access time or modification time to current time, and the effective UID of the calling process does not match the owner of the file; the process does not have write permission for the file; or the process does not have appropriate privileges.

• The calling process was attempting to truncate the file, and it does not have write permission for the file.

EBADF

• pathname is relative but dirfd is neither AT_FDCWD nor a valid file descriptor.

• fd is not a valid file descriptor.

EBUSY

The file is opened by a remote NFS client with a share reservation that conflicts with the requested operation.

EFBIG

The calling process was attempting to change the size of a file, but the specified length is greater than the maximum file size limit for the process.

EINVAL

• The attributes structure containing the requested changes is not valid.

• Invalid value in flags.

• pathname is NULL, dirfd is not AT_FDCWD, and flags contains AT_SYMLINK_NOFOLLOW.

ELOOP

A loop exists in symbolic links that were encountered during resolution of the pathname argument.

This error is issued if more than 24 symbolic links are detected in the resolution of pathname.

EMVSERR

An MVS environmental error or internal error occurred.

ENAMETOOLONG

pathname is longer than 1023 characters, or a component of the pathname is longer than 255

characters. (File name truncation is not supported.)

ENOENT

No file named pathname was found.

ENOSYS

The function is not supported for the specified file.

ENOTDIR

Some component of pathname is not a directory.

EPERM

The operation is not permitted for one of the following reasons:

• The calling process was attempting to change the mode or the file format but the effective UID of the calling process does not match the owner of the file, and the calling process does not have appropriate privileges.

• The calling process was attempting to change the owner but it does not have appropriate privileges.

• The calling process was attempting to change the general attribute bits but it does not have write permission for the file.

• The calling process was attempting to set a time value (not current time) but the effective UID does not match the owner of the file, and it does not have appropriate privileges.

• The calling process was attempting to set the change time or reference time to current time but it does not have write permission for the file.

• The calling process was attempting to change auditing flags but the effective UID of the calling process does not match the owner of the file and the calling process does not have appropriate privileges.

• The calling process was attempting to change the Security Auditor's auditing flags but the user does not have auditor authority.

EROFS

pathname specifies a file that is on a read-only file system.

Related Information

- "sys/stat.h — z/OS UNIX files and access" on page XX
- "fcntl.h — POSIX functions for file operations" on page XX
- "__chattr(), __chattr64() — Change the attributes of a file or directory" on page XX
- "__fchattr(), __fchattr64() — Change the attributes of a file or directory by file descriptor" on page XX
- "__lchattr(), __lchattr64() — Change the attributes of a file or directory when they point to a symbolic or external link" on page XX

clone() — Create a child process

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	AMODE64

Format

```
#define _XPLATFORM_SOURCE

#include <sched.h>

int clone(int (*fn)(void *), void *stack, int flags, void *arg, ...);
```

General Description

clone() create a new ("child") process, in a manner similar to fork().

By contrast with fork(), clone() provides more precise control over what pieces of execution context are shared between the calling process and the child process. For example, using clone(), the caller can control whether or not the two processes share the same namespace and the same parent process ID.

When the child process is created with clone(), it commences execution by calling the function pointed by the argument fn. (This differs from fork(), where execution continues in the child from the point of the fork() call.) The arg argument is passed as the argument of the function fn. The stack argument is ignored.

When the fn(arg) function returns, the child process terminates. The integer returned by fn is the exit status for the child process. The child process may also terminate explicitly by calling exit() or after receiving a fatal signal.

clone() allow a flags bit mask that modifies its behavior and allows the caller to specify what is shared between the calling process and the child process. The flags mask is specified as a bitwise-OR of zero or more of the constants listed below.

CLONE_NEWIPC
If CLONE_NEWIPC is set, then create the process in a new IPC namespace. If this flag is not set, then

(as with fork()), the process is created in the same IPC namespace as the calling process.

CLONE_NEWNS
If CLONE_NEWNS is set, the cloned child is started in a new mount namespace, initialized with a copy of the namespace of the parent. If CLONE_NEWNS is not set, the child lives in the same mount namespace as the parent.

CLONE_PARENT
If CLONE_PARENT is set, then the parent of the new child (as returned by getppid()) will be the same as that of the calling process. If this flag is not set, then (as with fork()) the child's parent is the calling process.

Note that it is the parent process, as returned by getppid(), which is signaled when the child terminates, so that if CLONE_PARENT is set, then the parent of the calling process, rather than the calling process itself, is signaled.

The CLONE_PARENT flag can't be used in clone calls by the global init process (PID 1 in the initial PID namespace) and init processes in other PID namespaces.

CLONE_NEWPID
If CLONE_NEWPID is set, then create the process in a new PID namespace. If this flag is not set, then (as with fork()) the process is created in the same PID namespace as the calling process. This flag can't be specified in conjunction with CLONE_PARENT.

CLONE_NEWUTS
If CLONE_NEWUTS is set, then create the process in a new UTS namespace, whose identifiers are initialized by duplicating the identifiers from the UTS namespace of the calling process. If this flag is not set, then (as with fork()) the process is created in the same UTS namespace as the calling process

Usage notes

A child termination signal can be specified in the flag parameter. SIGCHLD is the only supported signal and it must be specified.

If a UNIX set-user-ID or set-group-ID privileged program that switched the caller's effective UID or GID invokes the clone service, the child process that is created inherits the privilege of the set-user-ID or setgroup-ID program.

The child process is a duplicate of the process that calls the clone service (called the calling process), except for the following:

- The child process has a unique process ID (PID) in its namespace and each of any ancestor namespaces, that does not match any active process group ID.
- The child has a different parent process ID (the process ID of the process that called clone()) unless CLONE_PARENT was specified. If the new process is created in a PID namespace other than the PID namespace of the caller, the child appears to have no parent process (PPID=0) from its view within the namespace.
- The child will be created in the same namespaces as the calling process, unless one or more of the CLONE_NEWxxx flags are set or a prior call to unshare() or setns() with flag CLONE_NEWPID was issued by the calling process.
- The child has its own copy of the parent's file descriptors. Each file descriptor in the child refers to the same open file description as the corresponding file descriptor in the parent.
- The child has its own copy of the parent's open directory streams. Each child's open directory stream can share directory stream positioning with the corresponding parent's directory stream.

- If the file has its FD_CLOFORK flag set on, it is not inherited by the child process. This flag is set with a call to fcntl().
- The process and system utilization times for the child are set to zero.
- The following elements in the tms structure are set to 0 in the child:
    - tms_utime
    - tms_stime
    - tms_cutime
    - tms_cstime

  For more information about these elements, see times() — Get process and child process times.

- The child does not inherit any file locks previously set by the parent.
- The child process has no alarms set (similar to the results of a call to alarm() with an argument value of Wait_time specified as 0).
- The child has no pending signals.
- The child process has only a single thread. That thread is a copy of the thread in the parent that called clone(). The child process has a different thread ID. If the parent process was multithreaded (invoked pthread_create() at least once), the child process can only safely invoke async-signal-safe functions before it invokes an exec() family function. (This restriction also applies to any process created as the result of the child invoking clone() before it invokes an exec() family function because the child process is still considered multithreaded.) The child process does not inherit pthread attributes or pthread security environment. See Table 1 for a list of async-signal-safe functions.

When the clone requests the process to be added in one or more new namespaces (one of the CLONE_NEWxxx flags was specified) the caller must be authorized by being a superuser or having, at least, READ access to the CONTAINERS resource in the UNIXPRIV class.

The child process inherits all key 8 shared memory segments that are attached to the calling process. The internal values of the number of processes that are attached to each shared memory segment (shm_nattch) are incremented. The function clone() only supports the propagation of key 8 storage; therefore, clone() does not propagate to the child any shared memory segments that reside in a storage key other than key 8.

The child address space inherits the following address space attributes of the parent address space:

- Region size
- Time limit

The child process inherits the MEMLIMIT of the calling process.

If the parent process is multithreaded, it is the responsibility of the application to ensure that the application data is in a consistent state when the clone() occurs. For example, mutexes that are used to serialize updates to application data may need to be locked before the clone() and unlocked afterwards.

For more information on clone(), refer to z/OS UNIX System Services Programming: Assembler Callable Services Reference.

You can use MVS™ memory files from a z/OS® UNIX program. However, use of the clone() function from the program removes access from a hiperspace memory file for the child process. Use of an exec function from the program clears a memory file when the process address space is cleared.

The child process that results from a clone() in a multithreaded environment can only invoke async-signal-safe functions.

An async-signal-safe function is defined as a function that may be invoked, without restriction, from signal-catching functions. All supported async-signal-safe functions are listed in Table xx.

abort()	fpathconf()	raise()	sigpending()
accept()	fstat()	read()	sigprocmask()
access()	fsync()	readlink()	sigqueue()
aio_error()	ftruncate()	recv()	sigset()
aio_return()	getegid()	recvfrom()	sigsuspend()
aio_suspend()	geteuid()	recvmsg()	socket()
alarm()	getgid()	rename()	socketpair()
bind()	getgroups()	rmdir()	stat()
cfgetispeed()	getpeername()	select()	symlink()
cfgetospeed()	getpgrp()	send()	sysconf()
cfsetispeed()	getpid()	sendmsg()	tcdrain()
cfsetospeed()	getppid()	sendto()	tcflow()
chdir()	getsockname()	setgid()	tcflush()
chmod()	getsockopt()	setpgid()	tcgetattr()
chown()	getuid()	setsid()	tcgetpgrp()
close()	kill()	setsockopt()	tcsendbreak()
connect()	link()	setuid()	tcsetattr()
creat()	listen()	shutdown()	tcsetpgrp()
dup()	lseek()	sigaction()	time()
dup2()	lstat()	sigaddset()	times()
execle()	mkdir()	sigdelset()	umask()
execve()	mkfifo()	sigemptyset()	uname()
_Exit()	open()	sigfillset()	unlink()
_exit()	pathconf()	sigismember()	utime()
fchmod()	pause()	sleep()	wait()
fchown()	pipe()	signal()	waitpid()
fcntl()	poll()	sigpause()	write()
fork()			

Table 1. Async-signal-safe library functions

Interoperability restriction: For POSIX resources, clone() behaves as just described. But in general, MVS resources that existed in the parent do not exist in the child. This is true for open streams in MVS data sets and assembler-accessed MVS facilities, such as STIMERS. In addition, MVS allocations (through JCL, SVC99, or ALLOCATE) are not passed to the child process.

Special behavior for z/OS UNIX Services:

1. A prior loaded copy of an HFS program in the same address space is reused under the same circumstances that apply to the reuse of a prior loaded MVS unauthorized program from an unauthorized library by the MVS XCTL service with the following exceptions:
   - If the calling process is in Ptrace debug mode, a prior loaded copy is not reused.
   - If the calling process is not in Ptrace debug mode, but the only prior loaded usable copy found of the HFS program is in storage modifiable by the caller, the prior copy is not reused.
2. If the specified file name represents an external link or a sticky bit file, the program is loaded from the caller's MVS load library search order. For an external link, the external name is only used if the name is eight characters or less, otherwise the caller receives an error from the loadhfs service. For a sticky bit program, the file name is used if it is eight characters or less. Otherwise, the program is loaded from the HFS.
3. If the calling task is in a WLM enclave, the resulting task in the new process image is joined to the same WLM enclave. This allows WLM to manage the old and new process images as one 'business unit of work' entity for system accounting and management purposes.

Returned value
On success, the process ID of the child process is returned to the calling(parent) process. On failure, -1 is returned and no child process is created, errno is set to indicate the error.

Error Code Description
EAGAIN

The resources required to let another process be created are not available now, or you have already reached the maximum number of processes you are allowed to run.

EINVAL

Argument flags is not valid (A bit other than the supported flags is on, or the child termination signal is not SIGCHLD, or both CLONE_NEWPID and CLONE_PARENT are specified), or CLONE_PARENT was specified from an INIT process, or CLONE_NEWPID specified, however a setns CLONE_NEWPID was done for the current process.

ENOMEM

The process requires more space than is available.

ENOSPC

A system limit is reached.

EPERM

The calling process does not have appropriate privileges.

Related Information

- "sched.h — Manipulate and examine process execution scheduling" on page XX

- “fork() — Create a new process” on page XX
- “setns() — Allow for a thread to join a descendent namespace” on page XX
- “unshare() — Isolate a process from one or more of its namespaces” on page XX

dprintf() — Print to a file descriptor

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1-2008 Single UNIX Specification, Version 4	both	

Format

```
#define _POSIX_C_SOURCE  200809L

#include <stdio.h>

int dprintf(int fildes, const char *restrict format, ...);
```

General Description

The function dprintf() is exact analogs of fprintf() and vfprintf(), except that dprintf() output to a file descriptor fd instead of to a stdio stream.

Return value

If successful, dprintf() returns the number of characters output. The ending NULL character is not counted.

If unsuccessful, it returns a negative and errno is set to indicate the error.

Related Information

- “fprintf(), printf(), sprintf() — Format and write data” on page XX
- “vfprintf() — Format and print data to stream ” on page XX

dup3() — Duplicate an open file descriptor with flag setting

Standards

Standards / Extensions	C or C++	Dependencies
Z/OS UNIX	both	

Format

```
#define _XPLATFORM_SOURCE

#include <unistd.h>

int dup3(int oldfd, int newfd, int flags);
```

General Description

The interface dup3() is same as dup2(), except that:

- The caller can force the close-on-exec flag to be set for the new file descriptor by specifying O_CLOEXEC in flags.
- If oldfd equals newfd, then dup3() fails with the error EINVAL.

Return value

On success, the new file descriptor is returned. On error, -1 is returned, and errno is set to indicate the error.

Error code

EBADF

oldfd isn't an open file descriptor, or newfd is out of the allowed range for file descriptors

EINTR

dup3() was interrupted by a signal, or flags contain an invalid value, or oldfd was equal to newfd.

EMFILE

The per-process limit on the number of open file descriptors has been reached.

Related Information

- “unistd.h — Implementation-specific functions” on page XX
- “close() — Close a file” on page XX
- “creat() — Create a new file or rewrite an existing one” on page XX
- “dup() — Duplicate an open file descriptor” on page XX
- “fcntl() — Control open file descriptors” on page XX
- “open() — Open a file” on page XX

epoll_create(), epoll_create1() — Open an epoll file descriptor

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _XPLATFORM_SOURCE

#include <sys/epoll.h>

int epoll_create(int size);

int epoll_create1(int flags);
```

General Description

The epoll_create() function creates a new epoll instance and returns a file descriptor referring to that instance. The returned epoll instance can then be used to register interest in particular file descriptors using the epoll_ctl() function.

epoll_create() takes a size argument which should be positive to stand for the number of file descriptors that the caller expected to add to the epoll instance.

epoll_create1() is similar to epoll_create() but with a flags argument. If flags is 0, then epoll_create1() is the same as epoll_create(). The following value can be included in flags to obtain different behavior:

EPOLL_CLOEXEC

Set the close-on-exec (FD_CLOEXEC) flag on the new file descriptor.

Returned Value

If successful, epoll_create() and epoll_create1() return a file descriptor (a nonnegative integer).

If unsuccessful, epoll_create() and epoll_create1() return -1 and set errno to one of the following values:

Error Code Description

EINVAL

Invalid value specified in flags if using epoll_create1().

EMFILE

The per-user limit on the number of epoll instances has been reached, or the maximum number of file descriptors that can be open has been reached.

ENOMEM

Not enough space is available.

Related Information

- “sys/epoll.h — I/O event notification facility functions” on page XX
- “epoll_ctl() — Control interface for an epoll file descriptor” on page XX

epoll_ctl() — Control interface for an epoll file descriptor

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _XPLATFORM_SOURCE

#include <sys/epoll.h>

int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
```

Compile option: LANGLVL(EXTENDED)

General Description

The epoll_ctl() function is used to add, modify, or remove file descriptor entries in the interest list of the epoll instance. The desired I/O event may be registered for the specified file descriptor. The file descriptor can be for character special files, pipes, or sockets. The ready list can be retrieved using the epoll_wait() function.

Interest list is the set of file descriptors that the process has registered an interest in monitoring. Ready list is the set of file descriptors that are "ready" for I/O. The ready list is a subset of the file descriptors in the interest list. The ready list is dynamically populated by the kernel as a result of I/O activity on those file descriptors.

Parameter Description

epfd

File descriptor of an epoll instance.

op

Desired action to be performed on the provided epoll file descriptor. Valid values are:

EPOLL_CTL_ADD

Add an entry to the interest list of the epoll file descriptor, epfd. The entry includes the file descriptor fd, a reference to the corresponding open file description, and the settings specified in event.

EPOLL_CTL_MOD

Change the settings associated with fd in the interest list to the new settings specified in event.

EPOLL_CTL_DEL

Remove (deregister) the target file descriptor fd from the interest list.

fd

File descriptor corresponding to an open file description.

event

The object linked to the file descriptor fd. The struct epoll_event is defined as:

```
typedef union epoll_data {
    void *ptr;
    int   fd;
    uint32_t     u32;
    uint64_t     u64;
} epoll_data_t;
struct epoll_event {
    uint32_t     events;      /* Epoll events */
    epoll_data_t data; /* User data variable */
};
```

The data member of the epoll_event structure specifies data that the kernel should save and then return when this file descriptor becomes ready.

The events member of the epoll_event structure is a bit mask composed by ORing together zero or more of the following available event types:

EPOLLIN

The associated file is ready for read.

EPOLLOUT

The associated file is ready for write.

EPOLLPRI

Out-of-band data may be received without blocking.

EPOLLERR

An error has occurred.

EPOLLHUP

A hangup has occurred.

EPOLLONESHOT

Disable monitoring after event notification.

EPOLLEXCLUSIVE

Sets an exclusive wakeup mode for the epoll file descriptor. When a wakeup event occurs and multiple epoll file descriptors are attached to the same target file using EPOLLEXCLUSIVE, one or more of the epoll file descriptors will receive an event with epoll_wait(). The default in this scenario is for all epoll file descriptors to receive an event.

Returned Value

If successful, epoll_ctl() returns zero.

If unsuccessful, epoll_ctl() returns -1 and sets errno to one of the following values:

Error Code Description

EBADF

epfd or fd is not a valid file descriptor.

EEXIST

op was EPOLL_CTL_ADD, and the supplied file descriptor fd is already registered with this epoll instance.

EINVAL

epfd is not an epoll file descriptor, or fd is the same as epfd, or the requested operation op is not supported by this interface.

An invalid event type was specified along with EPOLLEXCLUSIVE in events.

op was EPOLL_CTL_MOD and events included EPOLLEXCLUSIVE.

op was EPOLL_CTL_MOD and the EPOLLEXCLUSIVE flag has previously been applied to this epfd, fd pair.

EPOLLEXCLUSIVE was specified in event and fd refers to an epoll instance.

ELOOP

fd refers to an epoll instance and this EPOLL_CTL_ADD operation would result in a circular loop of epoll instances monitoring one another or a nesting depth of epoll instances greater than 5.

ENOENT

op was EPOLL_CTL_MOD or EPOLL_CTL_DEL, and fd is not registered with this epoll instance.

ENOMEM

Not enough memory is available.

ENOSPC

The per-user limit on how many file descriptors can be registered in the interest list has been reached.

EPERM

The target file fd does not support epoll.

Related Information

- “sys/epoll.h — I/O event notification facility functions” on page XX
- “epoll_wait(), epoll_pwait() — Wait for an I/O event on an epoll file descriptor” on page XX

epoll_wait(), epoll_pwait() — Wait for an I/O event on an epoll file descriptor

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _XPLATFORM_SOURCE

#include <sys/epoll.h>

int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);

int epoll_pwait(int epfd, struct epoll_event *events, int maxevents, int timeout, const sigset_t *sigmask);
```

Compile option: LANGLVL(EXTENDED)

General Description

The epoll_wait() function waits for the desired I/O events on an interest list that were previously registered with a provided epoll instance. epoll_pwait() is similar to epoll_wait(), but takes an additional sigmask argument that specifies the desired signal mask when epoll_pwait() is blocked.

Parameter Description

epfd

File descriptor of an epoll instance.

events

Buffer used to return information from the ready list about file descriptors in the interest list that have some events available. Refer to the description of the event argument in the epoll_ctl() function for useful details.

maxevents

The maximum number of events that may be returned. This value must be greater than zero.

timeout

Timeout value in milliseconds that epoll_wait() or epoll_pwait() will block. If the timeout value is 0, epoll_wait() or epoll_pwait() returns immediately with any file descriptors that are in the ready list, if any. If the timeout value is -1, epoll_wait() or epoll_pwait() blocks until a file descriptor is in the ready list. If the timeout value is positive, it specifies that the number of milliseconds to wait for a ready list to be populated before returning to the caller.

sigmask

Desired signal mask when epoll_pwait() is blocked.

Returned Value

If successful, epoll_wait() and epoll_pwait() return the number of file descriptors ready for the requested I/O, or zero if no file descriptor became ready during the requested timeout milliseconds.

If unsuccessful, epoll_wait() and epoll_pwait() return -1 and set errno to one of the following values:

Error Code Description

EBADF

epfd is not a valid file descriptor.

EFAULT

The memory area pointed to by events is not accessible with write permissions.

EINTR

The call was interrupted by a signal handler before either any of the requested events occurred or the timeout expired.

EINVAL

epfd is not an epoll file descriptor, or maxevents is less than or equal to zero.

Related Information

- “sys/epoll.h — I/O event notification facility functions” on page XX
- “epoll_ctl() — Control interface for an epoll file descriptor” on page XX

eventfd() — Create a file descriptor for event notification

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

#define _XPLATFORM_SOURCE

#include <sys/eventfd.h>

int eventfd(unsigned int initval, int flags);

General Description

The eventfd() function creates an “eventfd object” that can be used as an event wait/notify mechanism by user applications, and by the kernel to notify user application of events. The object contains an unsigned 64-bit integer counter that is maintained by the kernel. This counter is initialized with the value specified in the argument initval.

As its return value, eventfd() returns a new file descriptor that can be used to refer to the eventfd object. The following value used in flags to change the behavior of eventfd():
EFD_CLOEXEC

Set the close-on-exec (FD_CLOEXEC) flag on the new file descriptor.

EFD_NONBLOCK

Set the O_NONBLOCK file status flag on the open file description referred to by the new file descriptor. Using this flag saves extra calls to fcntl() to achieve the same result.

EFD_SEMAPHORE

Provide semaphore-like semantics for reads from the new file descriptor.

The following operations can be performed on the file descriptor returned by eventfd():

read()

Each successful read() returns an 8-byte integer from the eventfd count. A read() fails with the error EINVAL if the size of the supplied buffer is less than 8 bytes.

The value returned by read() is in host byte order.

If the eventfd counter is zero at the time of the call to read(), then the call either blocks until the counter becomes nonzero (at which time, the read() proceeds as described above) or fails with the error EAGAIN if the file descriptor has been made nonblocking.

write()

A write() call adds the 8-byte integer value supplied in its buffer to the counter. The maximum value that may be stored in the counter is the largest unsigned 64-bit value minus 1. If the addition would cause the counter's value to exceed the maximum, then the write() either blocks until a read() is performed on the file descriptor, or fails with the error EAGAIN if the file descriptor has been made nonblocking.

A write() fails with the error EINVAL if the size of the supplied buffer is less than 8 bytes, or if an attempt is made to write the value 0xffffffffffffffff.

poll(), select()

Waits for one or more of the file descriptors to become ready to perform I/O. The returned file descriptor supports poll(), as follows:

The file descriptor is readable (the select() readlist argument; the poll() POLLIN flag) if the counter has a value greater than 0.

The file descriptor is writable (the select() writelist argument; the poll() POLLOUT flag) if it is possible to write a value of at least "1" without blocking.

If an overflow of the counter value was detected, then select() indicates the file descriptor as being both readable and writable, and poll() returns a POLLERR event.

The eventfd file descriptor also supports the other file-descriptor multiplexing APIs: pselect()

close()

When the file descriptor is no longer required it should be closed. When all file descriptors associated with the same eventfd object have been closed, the resources for object are freed by the kernel.

A copy of the file descriptor created by eventfd() is inherited by the child produced by fork(). The duplicate file descriptor is associated with the same eventfd object. File descriptors created by eventfd() are preserved across exec functions, unless the close-on-exec flag has been set.

Parameter Description

initval - The initialized value for the counter which is maintained by the kernel.

flags - Flag to change the behavior of the eventfd object.

Returned Value

On success, returns a new eventfd file descriptor. On error, -1 is returned and errno is set to indicate the error.

Error Code Description

EINVAL

An unsupported value was specified in flags.

EMFILE

The per-process limit on the number of open file descriptors has been reached.

ENFILE

The system-wide limit on the total number of open files has been reached.

ENODEV

Could not mount (internal) anonymous inode device.

ENOMEM

There was insufficient memory to create a new eventfd file descriptor.

Example

```
#include <sys/eventfd.h>
#include <unistd.h>
#include <inttypes.h>           /* Definition of PRIu64 & PRIx64 */
```

```
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>             /* Definition of uint64_t */

#define handle_error(msg) \
    do { perror(msg); exit(EXIT_FAILURE); } while (0)

int
main(int argc, char *argv[])
{
    int efd;
    uint64_t u;
    ssize_t s;

    if (argc < 2) {
        fprintf(stderr, "Usage: %s <num>...\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    efd = eventfd(0, 0);
    if (efd == -1)
        handle_error("eventfd");

    switch (fork()) {
    case 0:
        for (int j = 1; j < argc; j++) {
            printf("Child writing %s to efd\n", argv[j]);
            u = strtoull(argv[j], NULL, 0);
                    /* strtoull() allows various bases */
            s = write(efd, &u, sizeof(uint64_t));
            if (s != sizeof(uint64_t))
                handle_error("write");
        }
        printf("Child completed write loop\n");

        exit(EXIT_SUCCESS);

    default:
        sleep(2);

        printf("Parent about to read\n");
        s = read(efd, &u, sizeof(uint64_t));
        if (s != sizeof(uint64_t))
            handle_error("read");
        printf("Parent read %"PRIu64" (%#"PRIx64") from efd\n", u, u);
        exit(EXIT_SUCCESS);

    case -1:
        handle_error("fork");
    }
}
```

Related Information

- read() – Read from a file or socket on page XX
- write() — Write data on a file or socket on page XX

- poll() — Monitor activity on file descriptors and message queues on page XX
- select(), pselect() — Monitor activity on files or sockets and message queues on page XX
- close() — Close a file on page XX

faccessat() — Determine accessibility of a file relative to directory file descriptor

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1-2008 Single UNIX Specification, Version 4	both	

Format

```
#define _POSIX_C_SOURCE 200809L

#include <fcntl.h>            /* Definition of AT_* constants */

#include <unistd.h>

int faccessat(int dirfd, const char *pathname, int mode, int flags);
```

General Description

faccessat() operates in exactly the same way as access(), except for the differences described here.

If the pathname given in pathname is relative, then it is interpreted relative to the directory referred to by the file descriptor dirfd (rather than relative to the current working directory of the calling process, as is done by access() for a relative pathname).

If pathname is relative and dirfd is the special value AT_FDCWD, then pathname is interpreted relative to the current working directory of the calling process (like access()).

If pathname is absolute, then dirfd is ignored.

flags is constructed by ORing together zero or more of the following values:

AT_EACCESS

> Perform access checks using the effective user and group IDs. By default, faccessat() uses the real IDs (like access()).

AT_SYMLINK_NOFOLLOW

> If pathname is a symbolic link, do not dereference it: instead return information about the link itself.

Returned Value

If successful, faccessat() returns zero.

If unsuccessful, faccessat() returns -1 and sets errno to one of the following values:

Error Code Description

EACCES

The process does not have appropriate permissions to access the file in the specified way, or does not have search permission on some component of the pathname prefix.

EBADF

The pathname argument does not specify an absolute path and the fd argument is neither AT_FDCWD nor a valid file descriptor open for reading or searching.

EINVAL

mode was incorrectly specified or invalid flag specified in flags

ELOOP

A loop exists in symbolic links encountered during resolution of the path argument.

ENAMETOOLONG

pathname is longer than PATH_MAX characters. The PATH_MAX value is determined using pathconf().

ENOENT

There is no file named pathname, or the pathname argument is an empty string.

ENOTDIR

Some component of the pathname prefix is not a directory.

EROFS

The argument how has specified write access for a file on a read-only file system.

ETXTBSY

Write access was requested to an executable which is being executed.

Related Information

- “fcntl.h — POSIX functions for file operations” on page XX
- “unistd.h — Implementation-specific functions” on page XX
- “access() — Determine whether a file can be accessed” on page XX
- “chmod() — Change the mode of a file or directory” on page XX
- “chown() — Change the owner or group of a file or directory” on page XX

fchmodat() — Change the mode of a file relative to a directory file descriptor

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1-2008 Single UNIX Specification, Version 4	both	

Format

#define _POSIX_C_SOURCE 200809L

```
#include <fcntl.h>           /* Definition of AT_* constants */
#include <sys/stat.h>
int fchmodat(int fd, const char *path, mode_t mode, int flag);
```

General Description

The fchmodat() function shall be equivalent to the chmod() function except for the differences described here.

If the pathname given in pathname is relative, then it is interpreted relative to the directory referred to by the file descriptor dirfd (rather than relative to the current working directory of the calling process, as is done by chmod() for a relative pathname).

If pathname is relative and dirfd is the special value AT_FDCWD, then pathname is interpreted relative to the current working directory of the calling process (like chmod()).

If pathname is absolute, then dirfd is ignored.

Flags can either be 0 or include the following list, defined in <fcntl.h>:

AT_SYMLINK_NOFOLLOW

> If path name is a symbolic link, then the mode of the symbolic link is changed.

If fchmodat() is passed the special value AT_FDCWD in the fd parameter, the current working directory shall be used. If also flag is zero, the behavior shall be identical to a call to chmod().

Returned Value

If successful, fchmodat() returns zero.

If unsuccessful, fchmodat() returns -1 and sets errno to one of the following values:

Error Code Description

EACCES

> The calling process does not have permission to search some component of Pathname

EBADF

> The path argument does not specify an absolute path and the fd argument is neither AT_FDCWD nor a valid file descriptor open for reading or searching.

EINVAL

> One of the input parameters was not valid.

ELOOP

> A loop exists in symbolic links. This error is issued if more than POSIX_SYMLOOP symbolic links are encountered during resolution of the slink argument.

ENAMETOOLONG

> Pathname is longer than 1023 characters, or a component of the path name is longer than 255 characters. (Filename truncation is not supported.)

ENOENT

A component of path does not name an existing file or path is an empty string.

ENOTDIR

A component of the path prefix is not a directory.

EPERM

The effective UID of the calling process does not match the owner of the file, and the calling process does not have appropriate privilege.

EROFS

The file resides on a read-only file system.

Related Information

- “fcntl.h — POSIX functions for file operations” on page XX
- “fchmod() — Change the mode of a file or directory by descriptor” on page XX
- “chmod() — Change the mode of a file or directory” on page XX
- “chown() — Change the owner or group of a file or directory” on page XX
- “fchown() — Change the owner or group by file descriptor” on page XX

fchownat() — Change owner and group of a file relative to directory file descriptor

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 4 POSIX.1-2008	both	

Format

```
#define _POSIX_C_SOURCE  200809L

#include < unistd.h>

#include <fcntl.h>

int fchownat(int dirfd, const char *path, uid_t owner, gid_t group, int flag);
```

General Description

The fchownat() function operates in exactly the same way as chown(), , except for the differences described here. If the pathname given in path(parameter) is relative, then it is interpreted relative to the directory referred to by the file descriptor dirfd (rather than relative to the current working directory of the calling process, as is done by chown for a relative pathname).

If path is relative and dirfd is the special value AT_FDCWD, then path is interpreted relative to the current working directory of the calling process (like chown).

If pathname is absolute, then dirfd is ignored.

flags can either be 0, or include the following flag:

AT_SYMLINK_NOFOLLOW

If pathname is a symbolic link, do not dereference it: instead operate on the link itself, like lchown. (By default, fchownat() dereferences symbolic links, like chown.)

Returned Value

If successful, fchownat() returns 0.

If unsuccessful, -1 is returned with *errno* set to one of the following values:

Error Code Description

EBADF

dirfd is not a valid open file descriptor.

EPERM

Either the effective user ID does not match the owner of the file, or the calling process does not have appropriate privileges, and POSIX_CHOWN_RESTRICTED indicates that such privilege is required.

EROFS

The file resides on a read-only system.

EINVAL

Invalid flag specified in flags.

ENOTDIR

path is relative and dirfd is a file descriptor referring to a file other than a directory.

Related Information

- “unistd.h — Implementation-specific functions” on page XX
- “chown() — Change the owner or group of a file or directory” on page XX
- “chmod() — Change the mode of a file or directory” on page XX
- “fchmod() — Change the mode of a file or directory by descriptor” on page XX
- “fchown() — Change the owner or group by file descriptor” on page XX

fcntl() — Control open file descriptors

… …

File flags

… …

O_ASYNCSIG

If this flag is 1, then asynchronous I/O will be used for the file.

O_CLOEXEC

If this flag is 1, then set the close-on-exec flag for the new file descriptor.

This flag is defined by the _XPLATFORM_SOURCE feature test macro.

O_DIRECT

If this flag is 1, then try to minimize cache effects of the I/O to and from this file.

This flag is defined by the _XPLATFORM_SOURCE feature test macro.

O_NONBLOCK

… …

flock() — Apply or remove an advisory lock on an open file

Standards

Standards / Extensions	C or C++	Dependencies
BSD 4.4	both	

Format

```
#define _XPLATFORM_SOURCE

#include <sys/file.h>

int flock(int fd, int operation);
```

General Description

Apply or remove an advisory lock on the open file specified by fd. The argument operation is one of the following:

- LOCK_SH Place a shared lock. More than one process may hold a shared lock for a given file at a given time.
- LOCK_EX Place an exclusive lock. Only one process may hold an exclusive lock for a given file at a given time.
- LOCK_UN Remove an existing lock held by this process.

A call to flock() may block if an incompatible lock is held by another process. To make a nonblocking request, include LOCK_NB (by ORing) with any of the above operations.

A single file may not simultaneously have both shared and exclusive locks.

Locks created by flock() are associated with an open file description (see open()). This means that duplicate file descriptors (created by, for example, fork() or dup()) refer to the same lock, and this lock may be modified or released using any of these file descriptors. Furthermore, the lock is released either by an explicit LOCK_UN operation on any of these duplicate file descriptors, or when all such file descriptors have been closed.

If a process uses open() (or similar) to obtain more than one file descriptor for the same file, these file descriptors are treated independently by flock(). An attempt to lock the file using one of these file descriptors may be denied by a lock that the calling process has already placed via another file descriptor.

A process may hold only one type of lock (shared or exclusive) on a file. Subsequent flock() calls on an already locked file will convert an existing lock to the new lock mode.

Locks created by flock() are preserved across an execve().

A shared or exclusive lock can be placed on a file regardless of the mode in which the file was opened.

Returned value

If successful, flock() returns 0.

If unsuccessful, flock() retrurns -1 and sets errno to one of the following values:

Error Code Description

EBADF

fd is not an open file descriptor.

EINTR

While waiting to acquire a lock, the call was interrupted by delivery of a signal caughtby a handler.

EINVAL

operation is invalid.

ENOLCK

The kernel ran out of memory for allocating lock records.

EWOULDBLOCK

The file is locked and the LOCK_NB flag was selected.

Related Information

- sys/file.h – File manipulation constants
- close() – Close a file
- dup() – Duplicate an open file descriptor
- execve() – Run a new program by replacing the current process image
- fcntl() – Control open file desctiptors
- fork() – Create a new process
- open() – Open a file

fopen() — Open a file

General Description

File mode

Restriction: When running with POSIX(OFF) and specifying a mode parameter that includes t, for example, rt, rt+, r+t, wt, wt+, w+t, at, at+ or a+t, the fopen() request will fail with a message indicating a non-valid mode was specified.

Positional parameter points to a string beginning with one of the following sequences (possibly in-clude additional characters for extended file modes, as described below):

Table 25. Values for the Positional Parameter

File Mode	General Description

… …

To support extended file modes, additional characters can be included either as a last character or as a character between the characters in any of the two-character strings described above.

Table xx. Values for the Positional Parameter

Extended File Mode	General Description
e	Open the z/OS Unix file system file with the O_CLOEXEC flag. See open() for more information. This flag is ignored for fdopen(). This flag is ignored if the file not a z/OS Unix file system file.

---

fstatat(), fstatat64 — Get file status information relative to a directory file descriptor

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1-2008 Single UNIX Specification, Version 4	both	

Format

fstatat:

```
#define _POSIX_C_SOURCE 200809L

#include <fcntl.h>

#include <sys/stat.h>

int fstatat(int dirfd, const char *pathname, struct stat *statbuf, int flags);
```

fstatat64:

```
#define _LARGE_TIME_API

#define _POSIX_SOURCE

#include <fcntl.h>

#include <sys/stat.h>

int fstatat64(int dirfd, const char *pathname, struct stat64 *statbuf, int flags);
```

Compile requirement: Use of the fstatat64() function requires the long long data type. For more information on how to make the long long data type available, see z/OS XL C/C++ Language Reference.

General Description

The *fstatat()* function is equivalent to *stat()*, except for the differences described here.

If the pathname given in *pathname* is relative, then it is interpreted relative to the directory referred to by the file descriptor *dirfd* (rather than relative to the current working directory of the calling process, as is done by stat() for a relative pathname).

If *pathname* is relative and *dirfd* is the special value AT_FDCWD, then *pathname* is interpreted relative to the current working directory of the calling process.

If *pathname* is absolute, then *dirfd* is ignored.

The *flags* argument is a bit mask consisting of zero or the following flag:

AT_SYMLINK_NOFOLLOW

If *pathname* is a symbolic link, the status of the symbolic link is returned.

The *fstatat64()* function behaves exactly like fstatat() except fstatat64() uses structure *stat64* instead of struct *stat* to support time beyond 03:14:07 UTC on January 19, 2038.

Note: Environment variable _EDC_EOVERFLOW can be used to control behavior of fstatat() with respect to detecting an EOVERFLOW condition for z/OS® UNIX files. By default, fstatat() will not set EOVERFLOW when the file size can not be represented correctly in structure pointed to by *statbuf*. When _EDC_EOVERFLOW is set to YES, fstatat() will check for an overflow condition.

Large file support for z/OS UNIX files: fstatat64() automatically supports large z/OS UNIX files for both AMODE 31 and AMODE 64 C/C++ applications, which means there is no need for _LARGE_FILES feature test macro to be defined. As for fstatat(), the automatic support is only for AMODE 64 C/C++ applications. AMODE 31 C/C++ applications must be compiled with the option LANGLVL(LONGLONG) and define the _LARGE_FILES feature test macro before any headers are included to enable fstatat() to operate on z/OS UNIX files that are larger than 2 GB in size. File size and offset fields are enlarged to 63 bits in width. Therefore, any other function operating on the file is required to define the _LARGE_FILES feature test macro as well.

Returned Value

If successful, *fstatat()* returns zero.

If unsuccessful, *fstatat()* returns -1 and sets *errno* to one of the following values:

Error Code Description

EACCES

The process does not have search permission on some component of the pathname prefix.

EBADF

The file descriptor specified for *dirfd* is incorrect.

EINVAL

pathname is a NULL pointer*, or statbuf* is a NULL pointer, or an invalid flag was specified in the *flags*, or dor(.) or dot-dot(..) was specified in the *pathname*.

ELOOP

A loop exists in symbolic links encountered during resolution of the *pathname* argument. This error is returned if more than POSIX_SYMLOOP (defined in the limits.h header file) symbolic links are encountered during resolution of the pathname argument.

ENAMETOOLONG

pathname is longer than PATH_MAX characters, or some component of pathname is longer than

NAME_MAX characters while _POSIX_NO_TRUNC is in effect. For symbolic links, the length of the path name string substituted for a symbolic link exceeds PATH_MAX. The PATH_MAX and NAME_MAX values can be determined with pathconf().

ENOENT

There is no file named pathname, or pathname is an empty string.

ENOTDIR

A component of the path prefix names an existing file that is neither a directory nor a symbolic link to a directory, or the path argument contains at least one non-<slash> character and ends with one or more trailing <slash> characters and the last pathname component names an existing file that is neither a directory nor a symbolic link to a directory.

EOVERFLOW

The file size in bytes or the number of blocks allocated to the file or the file serial number cannot be represented correctly in the structure pointed to by *statbuf*.

Note: The fstatat() function might fail with error code EOVERFLOW if large file support is not enabled. The environment variable _EDC_EOVERFLOW controls this behavior. If _EDC_EOVERFLOW is set to YES the new behavior will take place. The default for _EDC_EOVERFLOW is NO. s

Related Information

- “fcntl.h — POSIX functions for file operations” on page XX
- “sys/stat.h — z/OS UNIX files and access” on page XX
- “mknod() — Make a directory or file” on page XX
- “fstat(), fstat64() — Get status information about a file” on page XX
- “lstat(), lstat64() — Get status of file or symbolic link” on page XX

futimesat() — Change timestamps of a file relative to a directory file descriptor

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _XPLATFORM_SOURCE

#include <fcntl.h>                    /* Definition of AT_* constants */

#include <sys/time.h>

int futimesat(int dirfd, const char *pathname, const struct timeval times[2]);
```

General Description

The futimesat() operates in exactly the same way as utimes(), except for the differences described below.

If the pathname given in pathname is relative, then it is interpreted relative to the directory referred to by the file descriptor dirfd (rather than relative to the current working directory of the calling process, as is done by utimes() for a relative pathname).

If pathname is relative and dirfd is the special value AT_FDCWD, then pathname is interpreted relative to the

current working directory of the calling process (like utimes()).

If pathname is absolute, then dirfd is ignored.

Returned value

On success, futimesat() returns a 0. On error, -1 is returned and errno is set to indicate the error.

Error Code Description

The same errors that occur for utimensat() can also occur for futimesat().

Related Information

- "futimes — change file timestamps" on page XX
- "openat() — open a file relative to a directory file descriptor" on page XX
- "stat() — get file information" on page XX
- "utimes — change file last access and modification times" on page XX
- "utimensat() — change file timestamps with nanosecond precision" on page XX

getrandom() — Obtain a series of random bytes

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _XPLATFORM_SOURCE
#include <sys/random.h>

ssize_t getrandom(void *buf, size_t buflen, unsigned int flags);
```

General Description

The getrandom() fills the buffer pointed to by buf with up to buflen random bytes. These bytes can be used to seed user-space random number generators or for cryptographic purposes.

The flags argument is a bit mask that can contain zero or more of the following values ORed together:

GRND_RANDOM

> This bit is always ignored, random bytes are drawn from the random source. There is no limitation to the number of bytes returned. It is only limited by the buffer size.

GRND_NONBLOCK

> This bit is always ignored. getrandom() does not block and will always return the data specified in the buf_len. Blocking for getrandom() is not supported.

Returned Value

If successful, getrandom() returns the number of bytes that were copied to the buffer buf.

If unsuccessful, getrandom() returns -1 and set *errno* to one of the following values:

Error Code **Description**

EFAULT	The address referred to by buf is outside the accessible address space.
EINTR	The call was interrupted by a signal handler.
EINVAL	An invalid flag was specified in flags.

Related Information

- “getentropy() — fill a buffer with random bytes” on page xxx

getentropy() — Fill a buffer with random bytes

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _XPLATFORM_SOURCE
#include <unistd.h>

int getentropy(void *buffer, size_t length);
```

General Description

The getentropy() function writes length bytes of high-quality random data to the buffer starting at the location pointed to by buffer. The maximum permitted value for the length argument is 256.

Returned Value

If successful, *getentropy()* returns zero.

If unsuccessful, getentropy() returns -1 and set *errno* to one of the following values:

Error Code **Description**

EFAULT	Part or all of the buffer specified by buffer and length is not in valid addressable memory.
EIO	length is greater than 256.

Related Information

- “getrandom() — obtain a series of random bytes” on page xxx

getopt_long() — Command long option parsing

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _XPLATFORM_SOURCE

#include <stdio.h>

#include <getopt.h>

int getopt_long(int argc, char *const argv[], const char *optstring,

                const struct option *longopts, int *longindex);
```

General Description

The getopt_long() function works like getopt() except that it also accepts long options, started with two dashes. (If the program accepts only long options, then optstring should be specified as an empty string (""), not NULL.) Long option names may be abbreviated if the abbreviation is unique or is an exact match for some defined option. A long option may take a parameter, of the form --option=arg or --option arg.

longopts is a pointer to the first element of an array of struct option declared in <getopt.h> as

```
struct option {
        const char *name;
        int         has_arg;
        int        *flag;
        int         val;
};
```

The meanings of the different fields are:

name is the name of the long option.

has_arg is: no_argument (or 0) if the option does not take an argument; required_argument (or 1) if the option requires an argument; or optional_argument (or 2) if the option takes an optional argument.

For optional_argument, getopt_long() can only parse the argument setting in form: --option=arg.

flag specifies how results are returned for a long option. If *flag* is NULL, then getopt_long() returns *val*. Otherwise, getopt_long() returns 0, and *flag* points to a variable which is set to *val* if the option is found, but left unchanged if the option is not found.

val is the value to return, or to load into the variable pointed to by *flag*.

The last element of the array has to be filled with zeros.

If *longindex* is not NULL, it points to a variable which is set to the index of the long option relative to *longopts*.

Returned Value

On success, it returns the option character if a short option is recognized. For a long option, it returns *val* if *flag* is NULL, otherwise it returns 0 and stores *val* in the location pointed to by *flag*.

If getopt_long() encounters an unknown option, then '?' is returned. If getopt_long() encounters an option with a missing argument, then the return value depends on the first character in optstring: if it is ':', then ':' is returned; otherwise '?' is returned.

If an option is specified with an argument but the option is defined to be no_argument in *longopts*, then getopt_long() returns '?'.

Otherwise getopt_long() returns -1 when all command line arguments have been parsed, or an unexpected error is encountered in the command line.

getopt_long() does not return any errno values.

If getopt_long() detects a missing argument or an option string not in *longopts*, it will write an error message to stderr describing the option character or string in error and the invoking program.

Related Information

- “stdio.h — Standard input and output” on page XX
- “getopt.h — z/OS UNIX function for parse command-line option” on page XX
- “getopt() — Command option parsing” on page XX

getsockopt() — Get the options associated with a socket

IP_RECVPKINFO

(RAW and UDP) Indicates whether returning the destination IP address of an incoming packet and the interface over which the packet was received is enabled or disabled. If the setsockopt() function is used to set this option, the set value is returned. The option value is passed as an int. The value 0 indicates the option is disabled and the value 1 indicates the option is enabled. When the option is enabled, the information is returned as IP_PKTINFO ancillary data on recvmsg() function calls. This option is protected by the _OPEN_SYS_SOCK_EXT4 feature test macro.

IP_TTL

(TCP and UDP) Used to retrieve the time-to-live field in the IP header for SOCK_STREAM(TCP) and SOCK_DGRAM (UDP) sockets.

Note: IP_TTL can only be used when _XPLATFORM_SOURCE is defined.

getxattr(), lgetxattr(), fgetxattr() — Retrieve an extended attribute value

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

#define _XPLATFORM_SOURCE

#include <sys/xattr.h>

ssize_t getxattr(const char *path, const char *name, void *value, size_t size);

ssize_t lgetxattr(const char *path, const char *name, void *value, size_t size);

ssize_t fgetxattr(int fd, const char *name, void *value, size_t size);

General Description

Extended attributes are extensions to the normal attributes which are associated with all inodes in the system. They are used as *name:value* pairs associated permanently with files and directories to provide additional functionality to a filesystem.

getxattr() retrieves the value of the extended attribute identified by name and associated with the given path in the filesystem. The attribute value is placed in the buffer pointed to by *value*, and *size* specifies the size of that buffer. If *size* is specified as zero, only the current size of the named extended attribute is returned, and the buffer pointed by *value* remains unchanged.

lgetxattr() is identical to *getxattr()*, except in the case of a symbolic link, where the link itself is interrogated, not the file that it refers to.

fgetxattr() is identical to *getxattr()*, only the open file referred to by *fd* is interrogated in place of *path*.

Returned Value

If successful, the size of the extended attribute value in bytes is returned.

If unsuccessful, -1 is returned with *errno* set to one of the following values:

Error Code Description

ENOATTR

The specified attribute is not set.

Attribute name is null/blank.

ENAMETOOLONG

pathname is longer than PATH_MAX characters.

ENODATA

The specified attribute is not set.

Attribute name is null/blank.

ENOTSUP

Extended attributes are not supported by the filesystem.

ERANGE

The size of the value buffer is too small to hold the result.

Attribute name is longer than PATH_MAX characters.

Related Information

- “sys/xattr.h — Extended attributes handling” on page XX
- “listxattr(), llistxattr(), flistxattr() — List extended attribute names” on page XX
- “removexattr(), lremovexattr(), fremovexattr() — Remove an extended attribute” on page XX
- “setxattr(), lsetxattr(), fsetxattr() — Set an extended attribute value” on page XX

inotify_init(), inotify_init1() — Initialize an inotify instance

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _XPLATFORM_SOURCE

#include <sys/inotify.h>

int inotify_init(void);

int inotify_init1(int flags);
```

General Description

The inotify_init() function creates a new inotify instance and returns a file descriptor associated with a new inotify event queue. The returned inotify instance can then be used to add interest in watching in particular file descriptors using the inotify_add_watch() function.

inotify_init1() is similar to inotify_init() but with a flags argument. If flags is 0, then inotify_init1() is the same as inotify_init(). The following values can be bitwise ORed in flags to obtain different behavior:

IN_NONBLOCK

Set the Non-Block file status flag on the file description referred to by the new file descriptor. This is the same O_NONBLOCK described by the open() function.

IN_CLOEXEC

Set the close-on-exec flag on the new file descriptor. Refer to the description of the O_CLOEXEC flag in the open() function for useful details.

Returned Value

If successful, inotify_init() and inotify_init1() return a new file descriptor.

If unsuccessful, inotify_init() and inotify_init1() return -1 and set errno to one of the following values:

Error Code Description

EINVAL

An invalid value was specified in flags if using inotify_init1().

EMFILE

The user limit on the total number of inotify instances has been reached, or the per-process limit on the number of open file descriptors has been reached

ENFILE

The system-wide limit on the total number of open files has been reached.

ENOMEM

Insufficient kernel memory is available.

Related Information

- “sys/inotify.h — Monitor and notify filesystem change functions” on page XX
- “open() — Open a file” on page XX

inotify_add_watch() — Add a watch to an initialized inotify instance

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _XPLATFORM_SOURCE

#include <sys/inotify.h>

int inotify_add_watch(int fd, const char *pathname, uint32_t mask);
```

General Description

The inotify_add_watch() function manipulates the "watch list" associated with an inotify instance returned by the inotify_init() function. Each watched item in the watch list specifies the pathname of a file or directory, along with some set of events that the kernel should monitor for the file referred to by that pathname. The inotify_add_watch() either creates a new watch item, or modifies an existing watch. Each watch has a unique watch descriptor, which is an integer returned by the call to inotify_add_watch() when the watch item is created.

Parameter Description

fd

File descriptor referring to the inotify instance whose watch list is to be modified.

pathname

Path name of a file or directory.

mask

Bit-mask form of the events to be monitored. Valid values are:

IN_ACCESS

File was accessed.

IN_ATTRIB

Metadata changed.

IN_CLOSE_WRITE

File opened for writing was closed.

IN_CLOSE_NOWRITE

File or directory not opened for writing was closed.

IN_CREATE

File/directory created in watched directory.

IN_DELETE

File/directory deleted from watched directory.

IN_DELETE_SELF

Watched file/directory was itself deleted.

IN_MODIFY

File was modified.

IN_MOVE_SELF

Watched file/directory was itself moved.

IN_MOVED_FROM

Generated for the directory containing the old filename when a file is renamed.

IN_MOVED_TO

Generated for the directory containing the new filename when a file is renamed.

IN_OPEN

File or directory was opened.

IN_ALL_EVENTS

Defined as a bit mask of all of the above events.

IN_MOVE

Equates to IN_MOVED_FROM | IN_MOVED_TO.

IN_CLOSE

Equates to IN_CLOSE_WRITE | IN_CLOSE_NOWRITE.

IN_DONT_FOLLOW

Don't dereference pathname if it is a symbolic link.

IN_EXCL_UNLINK

By default, when watching events on the children of a directory, events are generated for children even after they have been unlinked from the directory. This can result in large numbers of uninteresting events for some applications (e.g., if watching /tmp, in which many applications create temporary files whose names are immediately unlinked). Specifying IN_EXCL_UNLINK changes the default behavior, so that events are not generated for children after they have been unlinked from the watched directory.

IN_MASK_ADD

If a watch instance already exists for the filesystem object corresponding to pathname, add (OR) the events in mask to the watch mask (instead of replacing the mask); the error EINVAL results if IN_MASK_CREATE is also specified.

IN_ONESHOT

Monitor the filesystem object corresponding to pathname for one event, then remove from watch list.

IN_ONLYDIR

Watch pathname only if it is a directory; the error ENOTDIR results if pathname is not a directory. Using this flag provides an application with a race-free way of ensuring that the monitored object is a directory.

IN_MASK_CREATE

Watch pathname only if it does not already have a watch associated with it; the error EEXIST results if pathname is already being watched.

When events occur for monitored files and directories, those events are made available to the application as structured data that can be read from the inotify file descriptor using read() function. Each successful read() returns a buffer containing one or more of the following structure (**Compile requirement:** use of this structure requires a compiler that is designed to support C99):

```
struct inotify_event {
        int      wd;       /* Watch descriptor */
        uint32_t mask;     /* Mask describing event */
        uint32_t cookie;   /* Unique cookie associating related events */
        uint32_t len;      /* Size of name field */
        char     name[];   /* Optional null-terminated name */
}
```

wd identifies the watch for which this event occurs. It is one of the watch descriptors returned by a previous call to inotify_add_watch(). mask contains bits that describe the event that occurred. The following bits may be set in the mask field returned by read():

IN_IGNORED

Watch was removed explicitly by calling inotify_rm_watch() or automatically when file was deleted, or filesystem was unmounted.

IN_ISDIR

Subject of this event is a directory.

IN_Q_OVERFLOW

Event queue overflowed.

IN_UNMOUNT

Filesystem containing watched object was unmounted.

Returned Value

If successful, inotify_add_watch() returns a watch descriptor (a nonnegative integer).

If unsuccessful, inotify_add_watch() returns -1 and sets errno to one of the following values:

Error Code Description

EACCES

Read access to the given file is not permitted.

EBADF

The given file descriptor is not valid.

EEXIST

mask contains IN_MASK_CREATE and pathname refers to a file already being watched by the same fd.

EFAULT

pathname points outside of the process's accessible address space.

EINVAL

The given event mask contains no valid events, or mask contains both IN_MASK_ADD and IN_MASK_CREATE, or fd is not an inotify file descriptor.

ENAMETOOLONG

pathname is too long.

ENOENT

A directory component in pathname does not exist or is a dangling symbolic link.

ENOMEM

Insufficient kernel memory was available.

ENOSPC

The user limit on the total number of inotify watches was reached, or the kernel failed to allocate a needed resource.

ENOTDIR

mask contains IN_ONLYDIR and pathname is not a directory.

Related Information

- “sys/inotify.h — Monitor and notify filesystem change functions” on page XX
- “inotify_init(), inotify_init1() — Initialize an inotify instance” on page XX
- “inotify_rm_watch() — Remove an existing watch from an inotify instance” on page XX
- “read() — Read from a file or socket” on page XX

inotify_rm_watch() — Remove an existing watch from an inotify instance

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _XPLATFORM_SOURCE

#include <sys/inotify.h>

int inotify_rm_watch(int fd, int wd);
```

General Description

The *inotify_rm_watch()* function removes the watch associated with the watch descriptor returned by the *inotify_add_watch()* function from the inotify instance associated with the file descriptor. Removing a watch causes an IN_IGNORED event to be generated for this watch descriptor.

Parameter Description

fd

File descriptor of an inotify instance.

wd

File descriptor of a watched item.

Returned Value

If successful, *inotify_rm_watch()* returns zero.

If unsuccessful, *inotify_rm_watch()* returns -1 and sets *errno* to one of the following values:

Error Code Description

EBADF

fd is not a valid file descriptor.

EINVAL

The watch descriptor *wd* is not valid; or *fd* is not an inotify file descriptor.

Related Information

- “sys/inotify.h — Monitor and notify filesystem change functions” on page XX
- “inotify_add_watch() — Add a watch to an initialized inotify instance” on page XX

linkat() — Create a link to a file relative to directory file descriptor

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1-2008 Single UNIX Specification, Version 4	both	

Format

```
#define _POSIX_C_SOURCE  200809L
```

```
#include <fcntl.h>              /* Definition of AT_* constants */

#include <unistd.h>

int linkat(int olddirfd, const char *oldpath, int newdirfd, const char *newpath, int flags);
```

General Description

The function **linkat**()operates in exactly the same way as **link**(), except for the differences described here.

If the pathname given in oldpath is relative, then it is interpreted relative to the directory referred to by the file descriptor olddirfd (rather than relative to the current working directory of the calling process, as is done by **link**() for a relative pathname).

If oldpath is relative and olddirfd is the special value AT_FDCWD, then oldpath is interpreted relative to the current working directory of the calling process (like link()).

If oldpath is absolute, then olddirfd is ignored.

The interpretation of newpath is as for oldpath, except that a relative pathname is interpreted relative to the directory referred to by the file descriptor newdirfd.

The following values can be bitwise ORed in flags:

AT_EMPTY_PATH

> If oldpath is an empty string, create a link to the file referenced by olddirfd. In this case, olddirfd can refer to any type of file except a directory. This will generally not work if the file has a link count of zero (files created with O_TMPFILE and without O_EXCL are an exception).

AT_SYMLINK_FOLLOW

> By default, linkat() does not dereference oldpath if it is a symbolic link (like link()). The flag **AT_SYMLINK_FOLLOW** can be specified in flags to cause oldpath to be dereferenced if it is a symbolic link.

Return value

On success, zero is returned. On error, -1 is returned, and errno is set to indicate the error.

Error code Description

EACCES

> The process did not have appropriate permissions to create the link. Possible reasons include no search permission on a pathname component of oldpath or newpath, no write permission on the directory intended to contain the link, or no permission to access oldpath.

EBADF

> oldpath (newpath) is relative but olddirfd (newdirfd) is neither AT_FDCWD nor a valid file descriptor.

EEXIST

> Either newpath refers to a symbolic link, or a file or directory with the name newpath already exists.

EINVAL

> Either oldpath or newpath is incorrect, because it contains a NULL, or an invalid flag value was specified in flags.

ELOOP

A loop exists in symbolic links. This error is issued if the number of symbolic links encountered during resolution of oldpath or newpath is greater than POSIX_SYMLOOP.

EMLINK

oldpath already has its maximum number of links. The maximum number of links to a file is given by LINK_MAX, which you can determine by using pathconf() or fpathconf().

ENAMETOOLONG

oldpath or newpath is longer than PATH_MAX, or a component of one of the pathnames is longer than NAME_MAX while _POSIX_NO_TRUNC is in effect. For symbolic links, the length of the pathname string substituted for a symbolic link in oldfile or newname exceeds PATH_MAX. The PATH_MAX and NAME_MAX values can be determined using pathconf().

ENOENT

A pathname component of oldpath or newpath does not exist, or oldpath itself does not exist, or one of the two arguments is an empty string, or oldpath is a relative pathname and olddirfd refers to a directory that has been deleted, or newpath is a relative pathname and newdirfd refers to a directory that has been deleted.

ENOSPC

The directory intended to contain the link cannot be extended to contain another entry.

ENOTDIR

A pathname component of one of the arguments is not a directory, or oldpath is relative and olddirfd is a file descriptor referring to a file other than a directory; or similar for newpath and newdirfd.

EPERM

oldpath is the name of a directory, and links to directories are not supported.

EROFS

Creating the link would require writing on a read-only file system.

EXDEV

oldpath and newpath are on different file systems.

Related Information

- “unistd.h — Implementation-specific functions” on page XX
- “link() — Create a link to a file” on page XX
- “symlink() — Create a symbolic link to a path name” on page XX
- “unlink() — Remove a directory entry” on page XX

listxattr(), llistxattr(), flistxattr() — List extended attribute names

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

#define _XPLATFORM_SOURCE

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

ssize_t listxattr(const char *path, char *list, size_t size);

ssize_t llistxattr(const char *path, char *list, size_t size);

ssize_t flistxattr(int fd, char *list, size_t size);
```

General Description

Extended attributes are extensions to the normal attributes which are associated with all inodes in the system. They are used as *name:value* pairs associated permanently with files and directories to provide additional functionality to a filesystem.

listxattr() retrieves the list of extended attribute names associated with the given path in the filesystem. The retrieved list is placed in a caller-allocated buffer pointed by *list* with size (in bytes) specified by *size*. If *size* is specified as zero, only the current size of the list of extended attributes is returned, and the buffer pointed by *list* remains unchanged.

llistxattr() is identical to *listxattr()*, except in the case of a symbolic link, where the list of names of extended attributes associated with the link itself is retrieved, not the file that it refers to.

flistxattr() is identical to *listxattr()*, only the open file referred to by *fd* is interrogated in place of *path*.

Returned Value

If successful, the size of the extended attribute name list is returned.

If unsuccessful, -1 is returned with *errno* set to one of the following values:

Error Code Description

ENAMETOOLONG

pathname is longer than PATH_MAX characters.

ERANGE

The size of the value buffer is too small to hold the result.

ENOENT

There is no file named pathname, or pathname is an empty string.

Related Information

- “sys/xattr.h — Extended attributes handling” on page XX
- “getxattr(), lgetxattr(), fgetxattr() — Retrieve an extended attribute value” on page XX
- “removexattr(), lremovexattr(), fremovexattr() — Remove an extended attribute” on page XX
- “setxattr(), lsetxattr(), fsetxattr() — Set an extended attribute value” on page XX

mount() — Make a file system available

… …

Format

```
#define _OPEN_SYS
#include <sys/stat.h>
int mount(const char *path, char *filesystem,
          char *filesystype, mtm_t mtm,
          int parmlen, char *parm);
#define _OPEN_SYS
#define _XPLATFORM_SOURCE
#include <sys/mount.h>
int mount(const char *source, const char *target, const char *filesystemtype,
          unsigned long mountflags, const void *data);
```

General description

The mount() function defined under different feature test macros provides different input parameters and functionalities. For details, see the following respective descriptions.

When only _OPEN_SYS is defined:

Adds a file system to the hierarchical file system (HFS). The same file system cannot be mounted at more than one place in the hierarchical file system.

… …

parm

A parameter passed to the physical file system that performs the mount. This parameter may not be required. The form and content of the parm are determined by the physical file system. A hierarchical file system (HFS) data set does not require a parm.

When both _OPEN_SYS and _XPLATFORM_SOURCE are defined:

mount() attaches the filesystem specified by *source* (which is often a name referring to a filesystem, but can also be the pathname of a directory or file, or a dummy string) to the location (a directory or file) specified by the pathname in *target*.

In order to mount a file system, the caller must be an authorized program, or must be running for a user with appropriate privileges.

The *filesystemtype* argument specifies the name for the file system that will perform the mount. This 8-character name must match the TYPE operand on a FILESYSTYPE statement in the BPXPRMxx parmlib member for the file system.

The *data* argument specifies a parameter passed to the file system that performs the mount. This parameter may not be required. The form and content of the data are determined by the file system.

A call to mount() performs one of a number of general types of operation, depending on the bits specified in *mountflags*. The choice of which operation to perform is determined by testing the bits set in *mountflags*, with the tests being conducted in the order listed here:

- Remount an existing mount: *mountflags* includes MS_REMOUNT.
- Create a bind mount: *mountflags* includes MS_BIND.
- Change the propagation type of an existing mount: *mountflags* includes one of MS_PRIVATE or MS_UNBINDABLE.
- Move an existing mount to a new location: *mountflags* includes MS_MOVE.

- Create a new mount: *mountflags* includes none of the above flags.

Each of these operations is detailed later. Further flags may be specified in mountflags to modify the behavior of mount(), as described below.

The list below describes the additional flags that can be specified in *mountflags*:

MS_REC

Used in conjunction with MS_BIND to create a recursive bind mount, and in conjunction with the propagation type flags to recursively change the propagation type of all of the mounts in a subtree.

MS_RDONLY

Mount the file system as a read-only file system.

MS_NOSUID

The SETUID and SETGID mode flags will be ignored for programs that reside in this file system.

Remounting an existing mount

An existing mount may be remounted by specifying MS_REMOUNT in *mountflags*. This allows you to change the *mountflags* of an existing mount without having to unmount and remount the filesystem. *target* should be the same value specified in the initial mount() call.

The source, filesystemtype and data arguments are ignored.

The *mountflags* argument should match the values used in the original mount() call, except for those parameters that are being deliberately changed. The following mountflags can be changed: MS_NOSUID and MS_RDONLY.

The MS_REMOUNT flag can be used with MS_BIND to modify only the per-mount-point flags. This is particularly useful for setting or clearing the "read-only" flag on a mount without changing the underlying filesystem. Specifying *mountflags* as:

MS_REMOUNT | MS_BIND | MS_RDONLY

will make access through this mountpoint read-only, without affecting other mounts.

Creating a bind mount

If *mountflags* includes MS_BIND, then perform a bind mount. A bind mount makes a file or a directory subtree visible at another point within the single directory hierarchy.

The *filesystemtype* and *data* arguments are ignored.

The remaining bits (other than MS_REC, described below) in the *mountflags* argument are also ignored. (The bind mount has the same mount options as the underlying mount.) However, see the discussion of remounting above, for a method of making an existing bind mount read-only.

By default, when a directory is bind mounted, only that directory is mounted; if there are any submounts under the directory tree, they are not bind mounted. If the MS_REC flag is also specified, then a recursive bind mount operation is performed: all submounts under the source subtree (other than unbindable mounts) are also bind mounted at the corresponding location in the target subtree.

Changing the propagation type of an existing mount

If *mountflags* includes one of MS_PRIVATE or MS_UNBINDABLE, then the propagation type of an existing mount is changed. If more than one of these flags is specified, an error results.

The only other flags that can be specified while changing the propagation type is MS_REC.

The *source*, *filesystemtype*, and *data* arguments are ignored.

The meanings of the propagation type flags are as follows:

MS_PRIVATE

Make this mount private. Mount and unmount events do not propagate into or out of this mount.

MS_UNBINDABLE

Make this mount unbindable. This is like a private mount, and in addition this mount can't be bind mounted. When a recursive bind mount (mount() with the MS_BIND and MS_REC flags) is performed on a directory subtree, any unbindable mounts within the subtree are automatically pruned (i.e., not replicated) when replicating that subtree to produce the target subtree.

By default, changing the propagation type affects only the target mount. If the MS_REC flag is also specified in mountflags, then the propagation type of all mounts under target is also changed.

Moving a mount

If *mountflags* contains the flag MS_MOVE, then move a subtree: source specifies an existing mount and target specifies the new location to which that mount is to be relocated. The move is atomic: at no point is the subtree unmounted.

The remaining bits in the *mountflags* argument are ignored, as are the *filesystemtype* and *data* arguments.

Creating a new mount

If none of MS_REMOUNT, MS_BIND, MS_MOVE, MS_PRIVATE or MS_UNBINDABLE is specified in *mountflags*, then mount() performs its default action: creating a new mount. *source* specifies the source for the new mount, and *target* specifies the directory at which to create the mount point.

The *filesystemtype* and *data* arguments are employed, and further bits may be specified in *mountflags* to modify the behavior of the call.

Returned value

… …

Error Code Description

When only _OPEN_SYS is defined:

EBUSY

… …

EPERM

Superuser authority is required to issue a mount.

When both _OPEN_SYS and _XPLATFORM_SOURCE are defined:

EBUSY

The specified source file system is unavailable.

EINVAL

A parameter was incorrectly specified. Verify filesystype and mountflags. Another possible reason for this error is that the target mount point is the root of a file system or that the source file system is already mounted.

EIO

An I/O error occurred.

ELOOP

A loop exists in symbolic links. This error is issued if more than POSIX_SYMLOOP (defined in the limits.h header file) symbolic links are detected in the resolution of pathname.

ENAMETOOLONG

A pathname was longer than PATH_MAX, or some component of path name is longer that

NAME_MAX characters.

ENOENT

A pathname specified could not be found.

ENOMEM

There is not enough storage available to save the information required for this file system.

ENOTDIR

The target, or a prefix of source, is not a directory.

EPERM

Superuser authority is required to issue a mount.

Example

… …

Related Information

• “limits.h — Standard values for limits on resources” on page 38

• “sys/mount.h - File system mount and unmount” on page XX

• “sys/stat.h — z/OS UNIX files and access” on page 74

• “umount() — Remove a virtual file system” on page 1837

• “umount2() – Unmount file system” on page XX

• “w_getmntent() — Get information on mounted file systems” on page 1940

• “w_statfs() — Get the file system status” on page 1975

mkdirat() — Make a directory

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1-2008 Single UNIX Specification, Version 4	both	

Format

```
#define _POSIX_C_SOURCE 200809L

#include <fcntl.h>

#include <sys/stat.h>

int mkdirat(int dirfd, const char *pathname, mode_t mode);
```

General Description

The *mkdirat()* system call is equivalent to *mkdir()*, except for the following cases:

If the path name given in *pathname* is relative, then it is interpreted relative to the directory referred to by the file descriptor *dirfd* (rather than relative to the current working directory of the calling process, as is done by *mkdir()* for a relative pathname). If *pathname* is relative and *dirfd* is the special value *AT_FDCWD*, then *pathname* is interpreted relative to the current working directory of the calling process (like *mkdir()*). If *pathname* is absolute, then *dirfd* is ignored.

Returned Value

If successful, *mkdirat()* returns zero.

If unsuccessful, *mkdirat()* returns -1 and sets *errno* to one of the following values:

Error Code Description

EACCES

The process does not have search permission on some component of *pathname* or does not have write permission on the parent directory of the file or directory to be created.

EBADF

pathname is relative but *dirfd* is neither *AT_FDCWD* nor a valid file descriptor.

EEXIST

Either the named file refers to a symbolic link, or there is already a file or directory with the given *pathname*.

EFBIG

A request to create a directory is prohibited because the file size limit for the process is set to 0.

ELOOP

A loop exists in symbolic links. This error is issued if more than POSIX_SYMLOOP (defined in the limits.h header file) symbolic links are detected in the resolution of *pathname*.

EMLINK

The link count of the parent directory has already reached the maximum defined for the system.

ENAMETOOLONG

pathname is longer than PATH_MAX characters or some component of *pathname* is longer than NAME_MAX characters while _POSIX_NO_TRUNC is in effect. For symbolic links, the length of the *pathname* string substituted for a symbolic link exceeds PATH_MAX. The PATH_MAX and NAME_MAX values can be determined using pathconf().

ENOENT

The named file does not exist, or *pathname* is a relative path name and *dirfd* refers to a directory that has been deleted.

ENOSPC

The file system does not have enough space to contain a new file or directory, or the parent directory cannot be extended.

ENOTDIR

A component of the path prefix is not a directory, or *pathname* is relative and *dirfd* is a file descriptor referring to a file other than a directory.

EROFS

The parent directory of the file or directory to be created is on a read-only file system.

Related Information

- “fcntl.h — POSIX functions for file operations” on page XX
- “sys/stat.h — z/OS UNIX files and access” on page XX
- “mkdir() — Make a directory” on page XX

mkfifoat() — Make a FIFO special file

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1-2008 Single UNIX Specification, Version 4	both	

Format

```
#define _POSIX_C_SOURCE 200809L

#include <fcntl.h>

#include <sys/stat.h>

int mkfifoat(int dirfd, const char *path, mode_t mode);
```

General Description

The *mkfifoat()* system call is equivalent to *mkfifo()*, except in the case where *path* specifies a relative path. In this case the newly created FIFO is created relative to the directory associated with the file descriptor *dirfd* instead of the current working directory. If *mkfifoat()* is passed the special value *AT_FDCWD* in the *dirfd* parameter, the current working directory shall be used and the behavior shall be identical to a call to *mkfifo()*.

Returned Value

If successful, *mkfifoat()* returns zero.

If unsuccessful, *mkfifoat()* returns -1 and sets *errno* to one of the following values:

Error Code Description

EACCES

The process does not have search permission on some component of *pathname* or does not have write

permission on the parent directory of the file or directory to be created.

EBADF

pathname is relative but *dirfd* is neither *AT_FDCWD* nor a valid file descriptor.

EEXIST

Either the named file refers to a symbolic link, or there is already a file or directory with the given *pathname*.

EFBIG

A request to create a directory is prohibited because the file size limit for the process is set to 0.

EINVAL

The file type specified in the Mode parameter is not valid.

ELOOP

A loop exists in symbolic links. This error is issued if more than POSIX_SYMLOOP (defined in the limits.h header file) symbolic links are detected in the resolution of *pathname*.

EMLINK

The link count of the parent directory has already reached the maximum defined for the system.

ENAMETOOLONG

pathname is longer than PATH_MAX characters or some component of *pathname* is longer than NAME_MAX characters while _POSIX_NO_TRUNC is in effect. For symbolic links, the length of the *pathname* string substituted for a symbolic link exceeds PATH_MAX. The PATH_MAX and NAME_MAX values can be determined using pathconf().

ENOENT

The named file does not exist, or *pathname* is a relative path name and *dirfd* refers to a directory that has been deleted.

ENOSPC

The file system does not have enough space to contain a new file or directory, or the parent directory cannot be extended.

ENOTDIR

A component of the path prefix is not a directory, or *pathname* is relative and *dirfd* is a file descriptor referring to a file other than a directory.

EROFS

The parent directory of the file or directory to be created is on a read-only file system.

Related Information

- "fcntl.h — POSIX functions for file operations" on page XX
- "sys/stat.h — z/OS UNIX files and access" on page XX
- "mkfifo() — Make a FIFO special file" on page XX

mknodat() — Make a directory or file

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1-2008	both	

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 4		

Format

```
#define _POSIX_C_SOURCE 200809L

#include <fcntl.h>

#include <sys/stat.h>

int mknodat(int dirfd, const char *pathname, mode_t mode, dev_t dev);
```

General Description

The mknodat() system call is equivalent to mknod(), except for the following cases:

If the path name given in pathname is relative, then it is interpreted relative to the directory referred to by the file descriptor dirfd (rather than relative to the current working directory of the calling process, as is done by mknod() for a relative pathname). If pathname is relative and dirfd is the special value AT_FDCWD, then pathname is interpreted relative to the current working directory of the calling process (like mknod()). If pathname is absolute, then dirfd is ignored.

Returned Value

If successful, mknodat() returns zero.

If unsuccessful, mknodat() returns -1 and sets errno to one of the following values:

Error Code Description

EACCES

The process does not have search permission on some component of pathname or does not have write permission on the parent directory of the file or directory to be created.

EBADF

pathname is relative but dirfd is neither AT_FDCWD nor a valid file descriptor.

EEXIST

Either the named file refers to a symbolic link, or there is already a file or directory with the given pathname.

EFBIG

A request to create a directory is prohibited because the file size limit for the process is set to 0.

EINVAL

The file type specified in the Mode parameter is not valid.

ELOOP

A loop exists in symbolic links. This error is issued if more than POSIX_SYMLOOP (defined in the limits.h header file) symbolic links are detected in the resolution of pathname.

EMLINK

The link count of the parent directory has already reached the maximum defined for the system.

ENAMETOOLONG

pathname is longer than PATH_MAX characters or some component of pathname is longer than NAME_MAX characters while _POSIX_NO_TRUNC is in effect. For symbolic links, the length of the pathname string substituted for a symbolic link exceeds PATH_MAX. The PATH_MAX and NAME_MAX values can be determined using pathconf().

ENOENT

The named file does not exist, or pathname is a relative path name and dirfd refers to a directory that has been deleted.

ENOSPC

The file system does not have enough space to contain a new file or directory, or the parent directory cannot be extended.

ENOTDIR

A component of the path prefix is not a directory, or pathname is relative and dirfd is a file descriptor referring to a file other than a directory.

EPERM

The invoking process does not have appropriate privileges and the file type is not FIFO-special. The operation is not permitted. The operation requested requires a superuser authority.

EROFS

The parent directory of the file or directory to be created is on a read-only file system.

Related Information

- “fcntl.h — POSIX functions for file operations” on page XX
- “sys/stat.h — z/OS UNIX files and access” on page XX
- “mknod() — Make a directory or file” on page XX

nanosleep() — high-resolution sleep

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1-2001 POSIX.1-2008	both	

Format

```
#define _XOPEN_SOURCE 500

#include <time.h>

int nanosleep(const struct timespec *req, struct timespec *rem);
```

General Description

The function nanosleep() suspends the execution of the calling thread until either at least the time specified in *req has elapsed, or the delivery of a signal that triggers the invocation of a handler in the calling thread or that terminates the process.

If the call is interrupted by a signal handler, nanosleep() returns -1, sets errno to EINTR, and writes the remaining time into the structure pointed to by rem unless rem is NULL. The value of *rem can then be used to call nanosleep() again and complete the specified pause.

The structure timespec is used to specify intervals of time with nanosecond precision. The value of the nanoseconds field in it must be in the range 0 to 999999999.

Compared to sleep() and usleep(), nanosleep() has the following advantages: it provides a higher resolution for specifying the sleep interval; POSIX.1 explicitly specifies that it does not interact with signals; and it makes the task of resuming a sleep that has been interrupted by a signal handler easier.

Returned value

On successfully sleeping for the requested interval, nanosleep() returns 0. If the call is interrupted by a signal handler or encounters an error, then it returns -1, with errno set to indicate the error.

Error Code Description

EINTR

The pause has been interrupted by a signal that was delivered to the thread. The remaining sleep time has been written into *rem so that the thread can easily call nanosleep() again and continue with the pause.

EINVAL

The value in the tv_nsec field was not in the range 0 to 999999999 or the tv_sec field was not in the range 0 to 4294967295 for a 64-bit caller.

Related Information

- time.h – Time and date on page XX
- sleep() – Suspend execution of a thread on page XX
- usleep() – Suspend execution for an interval on page XX

open() — Open a file

… …

General description

… …

O_APPEND

Positions the file offset at the end of the file before each write operation.

O_CLOEXEC

Enable the close-on-exec flag for the new file descriptor. Specifying this flag permits a program to avoid additional fcntl() F_SETFD operations to set the FD_CLOEXEC flag. This flag is defined by the _XPLATFORM_SOURCE feature test macro.

O_CREAT

… …

O_DIRECT

Try to minimize cache effects of the I/O to and from this file. This flag is defined by the _XPLATFORM_SOURCE feature test macro.

O_DIRECTORY

If pathname is not a directory, cause the open to fail. This flag is defined by the _XPLATFORM_SOURCE feature test macro.

O_EXCL

… …

O_NOCTTY

… …

O_NOFOLLOW

If the trailing component (i.e., basename) of pathname is a symbolic link, then the open fails, with the error ELOOP. This flag is defined by the _XPLATFORM_SOURCE feature test macro.

O_NONBLOCK

… …

O_PATH

Obtain a file descriptor either to indicate a location in the filesystem tree or to perform operations that act purely at the file descriptor level. This flag is defined by the _XPLATFORM_SOURCE fea-ture test macro.

O_TRUNC

openat() — Open a file relative to a directory file descriptor

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1-2008 Single UNIX Specification, Version 4	both	

Format

```
#define _POSIX_C_SOURCE  200809L

#include <fcntl.h>

int openat(int dirfd, const char *pathname, int flags, ...);
```

General Description

The openat() system call opens a file using the specified directory file descriptor as the starting location for the path search. It operates in exactly the same way as open(), except for the differences described here.

The dirfd argument is used in conjunction with the pathname argument as follows:

- If the pathname given in pathname is absolute, then dirfd is ignored.

- If the pathname given in pathname is relative and dirfd is the special value AT_FDCWD, then pathname is interpreted relative to the current working directory of the calling process.
- If the pathname given in pathname is relative, then it is interpreted relative to the directory referred to by the file descriptor dirfd (rather than relative to the current working directory of the calling process, as is done by open() for a relative pathname).

If the pathname given in pathname is relative, and dirfd is not a valid file descriptor, an error (EBADF) is returned.

Return value

On success, a new file descriptor is returned. On error, -1 is returned, and errno is set to indicate the error.

Error code

The same error that occur for open() can also occur for openat(). The following additional errors can occur for openat():

EBADF

dirfd is not a valid file descriptor.

ENOTDIR

pathname is relative and dirfd is a file descriptor referring to a file other than a directory.

Related Information

- “sched.h — Manipulate and examine process execution scheduling” on page XX
- “open() — open a file” on page XX
- “opendir() —” on page XX
- “dirfd() —” on page XX

openat2() — Open a file with more extension

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _XPLATFORM_SOURCE

#define POSIX_SOURCE                    /* Definition of O_* and S_* constants */
#include <fcntl.h>

long openat2(int dirfd, const char *pathname, struct open_how *how, size_t size);
```

Compile option: LANGLVL(EXTENDED)

General Description

The openat2() system call is an extension of openat() and provides a superset of its functionality. For openat(), the flags and mode parameters are passed as separate integer fields, where in openat2() they are passed as a open_how structure.

The openat2() function opens the file specified by pathname. If the specified file does not exist, it may optionally be (if O_CREAT is specified in how.flags) created.

As with openat(), if pathname is a relative pathname, then it is interpreted relative to the directory referred to by the file descriptor dirfd (or the current working directory of the calling process, if dirfd is the special value AT_FDCWD). If pathname is an absolute pathname, then dirfd is ignored (unless how.resolve contains RESOLVE_IN_ROOT, in which case pathname is resolved relative to dirfd).

Rather than taking a single flags argument, an extensible structure (how) is passed to allow for future extensions. The size argument must be specified as sizeof(struct open_how).

The how argument specifies how pathname should be opened, and acts as a superset of the flags and mode arguments to openat(). This argument is a pointer to a structure of the following form:

```
struct open_how {
        uint64_t flags;       /* O_* flags */
        uint64_t mode;     /* Mode for O_CREAT */
        uint64_t resolve;  /* RESOLVE_* flags */
        /* ... */
};
```

Any future extensions to openat2() will be implemented as new fields appended to the above structure. The fields of the open_how structure are as follows:

flags This field specifies the file creation and file status flags to use when opening the file. All of the O_* flags defined for openat() are valid openat2() flag values.

mode This field specifies the mode for the new file, with identical semantics to the mode argument of openat().

Resolve This is a bit-mask of flags that modify the way in which all components of pathname will be resolved (a component is a substring delimited by '/'). The primary use case for these flags is to allow trusted programs to restrict how untrusted paths (or paths inside untrusted directories) are resolved. The full list of resolve flags is as follows:

RESOLVE_BENEATH

Do not permit the path resolution to succeed if any component of the resolution is not a descendant of the directory indicated by dirfd. This causes absolute symbolic links (and absolute values of pathname) to be rejected.

Currently, this flag also disables magic-link resolution (see below). However, this may change in the future. Therefore, to ensure that magic links are not resolved, the caller should explicitly specify RESOLVE_NO_MAGICLINKS.

RESOLVE_IN_ROOT

Treat the directory referred to by dirfd as the root directory while resolving pathname. Absolute symbolic links are interpreted relative to dirfd. If a prefix component of pathname equates to dirfd, then an immediately following component likewise equates to dirfd (just as /.. is traditionally equivalent to /). If pathname is an absolute path, it is also interpreted relative to dirfd.

The effect of this flag is as though the calling process had used chroot() to (temporarily) modify its root

directory (to the directory referred to by dirfd). However, unlike chroot() (which changes the filesystem root permanently for a process), RESOLVE_IN_ROOT allows a program to efficiently restrict path resolution on a per-open basis.

Currently, this flag also disables magic-link resolution. However, this may change in the future. Therefore, to ensure that magic links are not resolved, the caller should explicitly specify RESOLVE_NO_MAGICLINKS.

RESOLVE_NO_MAGICLINKS

Disallow all magic-link resolution during path resolution.

Magic links are symbolic link-like objects that are most notably found in proc(); examples include /proc/[pid]/exe and /proc/[pid]/fd/*.

Unknowingly opening magic links can be risky for some applications. Examples of such risks include the following:

• If the process opening a pathname is a controlling process that currently has no controlling terminal (see credentials(7)), then opening a magic link inside /proc/[pid]/fd that happens to refer to a terminal would cause the process to acquire a controlling terminal.

• In a containerized environment, a magic link inside /proc may refer to an object outside the container, and thus may provide a means to escape from the container.

Because of such risks, an application may prefer to disable magic link resolution using the RESOLVE_NO_MAGICLINKS flag.

If the trailing component (i.e., basename) of pathname is a magic link, how.resolve contains RESOLVE_NO_MAGICLINKS, and how.flags contains both O_PATH and O_NOFOLLOW, then an O_PATH file descriptor referencing the magic link will be returned.

RESOLVE_NO_SYMLINKS

Disallow resolution of symbolic links during path resolution. This option implies RESOLVE_NO_MAGICLINKS.

If the trailing component (i.e., basename) of pathname is a symbolic link, how.resolve contains RESOLVE_NO_SYMLINKS, and how.flags contains both O_PATH and O_NOFOLLOW, then an O_PATH file descriptor referencing the symbolic link will be returned.

Note that the effect of the RESOLVE_NO_SYMLINKS flag, which affects the treatment of symbolic links in all of the components of pathname, differs from the effect of the O_NOFOLLOW file creation flag (in how.flags), which affects the handling of symbolic links only in the final component of pathname.

Applications that employ the RESOLVE_NO_SYMLINKS flag are encouraged to make its use configurable (unless it is used for a specific security purpose), as symbolic links are very widely used by end-users. Setting this flag indiscriminately—i.e., for purposes not specifically related to security—for all uses of openat2() may result in spurious errors on previously functional systems. This may occur if, for example, a system pathname that is used by an application is modified (e.g., in a new distribution release) so that a pathname component (now) contains a symbolic link.

RESOLVE_NO_XDEV

Disallow traversal of mount points during path resolution (including all bind mounts). Consequently, pathname must either be on the same mount as the directory referred to by dirfd, or on the same mount as the current working directory if dirfd is specified as AT_FDCWD.

Applications that employ the RESOLVE_NO_XDEV flag are encouraged to make its use configurable (unless it is used for a specific security purpose), as bind mounts are widely used by end-users. Setting this flag indiscriminately—i.e., for purposes not specifically related to security—for all uses of openat2() may result in spurious errors on previously functional systems. This may occur if, for example, a system pathname that is used by an application is modified (e.g., in a new distribution release) so that a pathname component (now) contains a bind mount.

RESOLVE_CACHED

Make the open operation fail unless all path components are already present in the kernel's lookup cache. If

any kind of revalidation or I/O is needed to satisfy the lookup, openat2() fails with the error EAGAIN. This is useful in providing a fast-path open that can be performed without resorting to thread offload, or other mechanisms that an application might use to offload slower operations.

If any bits other than those listed above are set in how.resolve, an error is returned.

Returned value
On success, a new file descriptor is returned. On error, -1 is returned, and errno is set to indicate the error.

Error Code Description
The set of errors returned by openat2() includes all of the errors returned by openat(), as well as the following additional errors:

E2BIG
Unsupported flag specified in resolve field.

EAGAIN
RESOLVE_CACHED was set, and the open operation cannot be performed using only cached information. The caller should retry without **RESOLVE_CACHED** set in how.resolve.

EINVAL
An unknown flag or invalid value was specified in how, or mode is nonzero, but how.flags does not contain O_CREAT or O_TMPFILE.

ELOOP
how.resolve contains **RESOLVE_NO_SYMLINKS**, and one of the path components was a symbolic link (or magic link), or how.resolve contains **RESOLVE_NO_MAGICLINKS**, and one of the path components was a magic link.

EXDEV
how.resolve contains either **RESOLVE_IN_ROOT** or **RESOLVE_BENEATH**, and an escape from the root during path resolution was detected, or how.resolve contains **RESOLVE_NO_XDEV**, and a path component crosses a mount point.

Related Information

- “sched.h — Manipulate and examine process execution scheduling” on page XX
- “open() — open a file” on page XX
- “openat() — open a file relative to a directory file descriptor” on page XX
- “opendir() —” on page XX
- “dirfd() —” on page XX

pipe2() — Create pipe

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

#define _XPLATFORM_SOURCE

```
#define _POSIX_SOURCE             /* Definition of O_* constants */

#include <fcntl.h>              /* Definition of O_* constants */

#include <unistd.h>

int pipe2(int pipefd[2], int flags);
```

General Description

pipe2() creates a pipe, a unidirectional data channel that can be used for interprocess communication. The array pipefd is used to return two file descriptors referring to the ends of the pipe. pipefd[0] refers to the read end of the pipe. pipefd[1] refers to the write end of the pipe. Data written to the write end of the pipe is buffered by the kernel until it is read from the read end of the pipe.

If flags is 0, then pipe2() is the same as pipe(). The following values can be bitwise ORed in flags to obtain different behavior:

O_CLOEXEC

Set the close-on-exec (FD_CLOEXEC) flag on the two new file descriptors.

O_DIRECT

Create a pipe that performs I/O in "packet" mode. Each write to the pipe is dealt with as a separate packet and read from the pipe will read one packet at a time. Note the following points:

- Writes of greater than PIPE_BUF bytes will be split into multiple packets. The constant PIPE_BUF is defined in <limits.h>.
- If a read specifies a buffer size that is smaller than the next packet, then the requested number of bytes are read, and the excess bytes in the packet are discarded. Specifying a buffer size of PIPE_BUF will be sufficient to read the largest possible packets (seethe previous point).
- Zero-length packets are not supported. (A read that specifies a buffer size of zero is a no-op, and returns0.)

O_NONBLOCK

- Set the O_NONBLOCK file status flag on the open file descriptions referred to by the new file descriptors. Using this flag saves extra calls to fcntl() to achieve the same result.

Returned value

If successful, pipe2() returns 0.

If unsuccessful, pipe2() returns -1 and sets errno to one of the following values:

Error Code Description

EFAULT

pipefdis not valid.

EINVAL

Invalid value in flags.

EMFILE

The process has either reached the maximum number of file descriptors it can have open.

ENFILE

The maximum number of file descriptors that can be open has been reached.

Related Information

- unistd.h - Implementation-specific functions
- close() - Close a file
- fcntl() - Control open file descriptors
- open() - Open a file
- read() - Read from a file or socket
- write() - Write data on a file or socket
- pipe() – Create an unnamed pipe

pivot_root() — change the root mount

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _XPLATFORM_SOURCE

#include <unistd.h>

int pivot_root(const char *new_root, const char *put_old);
```

General Description

The function pivot_root() changes the root mount in the mount namespace of the calling process. More precisely, it moves the root mount to the directory put_old and makes new_root the new root mount. The caller must be an authorized program, or must be running for a user with appropriate privileges.

The function pivot_root() changes the root directory and the current working directory of each process or thread in the same mount namespace to new_root if they point to the old root directory. On the other hand, pivot_root() does not change the caller's current working directory (unless it is on the old root directory), and thus it should be followed by a chdir("/") call.

The following restrictions apply:

- new_root and put_old must be directories.
- new_root and put_old must not be on the same mount as the current root.
- put_old must be at or underneath new_root; that is, adding some nonnegative number of "/.." prefixes to the pathname pointed to by put_old must yield the same directory as new_root.
- new_root must be a path to a mount point but can't be "/". A path that is not already a mount point can be converted into one by bind mounting the path onto itself.
- The current root directory must be a mount point.

Returned value

If successful, pivot_root() returns 0.

If unsuccessful, pivot_root() retrurns -1 and sets errno to one of the following values:

Error Code Description

EBUSY

new_root or put_old is on the current root mount.

EINVAL

new_root is not a mount point.

EINVAL

put_old is not at or underneath new_root.

EINVAL

The current root directory is not a mount point.

ENOTDIR

new_root or put_old is not a directory.

EPERM

The calling process doesn't have authority.

Related Information

- unistd.h – Time and date on page XX
- chdir() – Changing the working directory on page XX
- chroot() – Change root directory on page XX
- mount() – Make a file system available on page XX

prctl() — Operations on a process or thread

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _XPLATFORM_SOURCE

#include <sys/prctl.h>

int prctl(int option, unsigned long arg2, unsigned long arg3, unsigned long arg4, unsigned long arg5);
```

General Description

prctl() manipulates various aspects of the behavior of the calling thread or process. These operations should be used with care.

prctl() is called with a first argument describing what to do (with values defined in <sys/prctl.h>), and further arguments with a significance depending on the first one. The first argument can be:

PR_SET_CHILD_SUBREAPER

If arg2 is nonzero, set the "child subreaper" attribute of the calling process; if arg2 is zero, unset the attribute.

When a process becomes orphaned (i.e., its immediate parent terminates), then that process will be reparented to the nearest still living ancestor subreaper. Subsequently, calls to getppid() in the orphaned process will now return the PID of the subreaper process, and when the orphan terminates, it is the subreaper process that will receive a SIGCHLD signal. The setting of the "child subreaper" attribute is not inherited by children created by fork() and clone().The setting is preserved across execve().

Establishing a subreaper process is useful in session management frameworks where a hierarchical group of processes is managed by a subreaper process that needs to be informed when one of the processes—for example, a double-forked daemon—terminates (perhaps so that it can restart that process).

PR_GET_CHILD_SUBREAPER

Return the "child subreaper" setting of the caller, in the location pointed to by (int *) arg2.

PR_SET_DUMPABLE

Set the state of the "dumpable" attribute, which determines whether core dumps are produced for the calling process upon delivery of a signal whose default behavior is to produce a core dump. arg2 must be either 0 (process is not dumpable) or 1 (process is dumpable). Normally, the "dumpable" attribute is set to 1. However, it is reset to value 0), in the following circumstances:

- The process's effective user or group ID is changed.
- The process's filesystem user or group ID is changed.
- The process executes (execve()) a set-user-ID or set-group-ID program, resulting in a change of either the effective user ID or the effective group ID.

As a security measure, the ownership is made root:root if the process's "dumpable" attribute is set to a value other than 1.

PR_GET_DUMPABLE

Return (as the function result) the current state of the calling process's dumpable attribute.

PR_SET_NAME

Set the name of the calling thread, using the value in the location pointed to by (char *) arg2. The name can be ~~up to 16 bytes long, including the terminating null byte~~ a null-terminated string with the max length of 16 characters. If the length of the string, including the terminating null byte, exceeds 16 bytes, the string is silently truncated.

PR_GET_NAME

Return the name of the calling thread, in the buffer pointed to by (char *) arg2. ~~The buffer~~ The name ~~should allow space for up to 16 bytes~~ can be a string with the max length of 16 characters. The returned string will be null-terminated if it is shorter than that.

PR_SET_NO_NEW_PRIVS

Set the calling thread's no_new_privs attribute to the value in arg2. With no_new_privs set to 1, execve() promises not to grant privileges to do anything that could not have been done without the execve call. Once set, the no_new_privs attribute cannot be unset. The setting of this attribute is inherited by children created by fork() and clone(), and preserved across execve().

PR_GET_NO_NEW_PRIVS

Return (as the function result) the value of the no_new_privs attribute for the calling thread. A value of 0 indicates the regular execve(2) behavior. A value of 1 indicates execve() will operate in the privilege-restricting mode described above.

PR_SET_PDEATHSIG

Set the parent-death signal of the calling process to arg2 (0 to clear). This is the signal that the calling process will get when its parent dies.

Warning: the "parent" in this case is considered to be the thread that created this process. In other words, the signal will be sent when that thread terminates (via, for example, pthread_exit()), rather than after all of the

threads in the parent process terminate. The parent-death signal is sent upon subsequent termination of the parent thread and also upon termination of each subreaper process (see the description of PR_SET_CHILD_SUBREAPER below) to which the caller is subsequently reparented. If the parent thread and all ancestor subreapers have already terminated by the time of the PR_SET_PDEATHSIG operation, then no parent-death signal is sent to the caller.

The parent-death signal setting is cleared for the child of a fork(). It is also cleared when executing a set-user-ID or set-group-ID binary. otherwise, this value is preserved across execve(). The parent-death signal setting is also cleared upon changes to any of the following thread credentials: effective user ID, effective group ID, filesystem user ID, or filesystem group ID.

PR_GET_PDEATHSIG

Return the current value of the parent process death signal, in the location pointed to by (int *) arg2.

Returned value

On success, PR_GET_DUMPABLE, PR_GET_NO_NEW_PRIVS return the nonnegative values described above. All other option values return 0 on success. On error, -1 is returned, it sets errno to one of the following values:

Error Code Description

EFAULT

arg2 is an invalid address.

EINVAL

option is PR_SET_PDEATHSIG and arg2 is not a valid signal number.

option is PR_GET_Name and arg2 is not a valid signal number.

Related Information

- “sys/prctl.h — Operations on a process or thread” on page XX

prlimit() — get or set the hard and soft resource limits of a specified process.

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _XOPEN_SOURCE_EXTENDED 1
#define _XPLATFORM_SOURCE
#include <sys/resource.h>

int prlimit(pid_t pid, int resource, const struct rlimit *new_limit,
            struct rlimit *old_limit);
```

General Description

prlimit() gets or sets resource limits for the specified process. A resource limit is a pair of values; one specifies the current (soft) limit and the other the maximum (hard) limit.

The pid argument specifies the process whose resource limits are to be set or retrieved. If pid is equal to 0, the resource limits are set or retrieved for the calling process.

The caller should have the following privilege for changing other process resource limit, either the caller has superuser level authority or the real, effective, and saved set user IDs of the target process match the real user ID of the caller and the real, effective, and saved set group IDs of the target process must match the real group ID of the caller.

The soft limit may be modified to any value that is less than or equal to the hard limit. For certain resourcevalues (RLIMIT_CPU, RLIMIT_NOFILE, RLIMIT_AS), the soft limit cannot be set lower than the existing usage.

The hard limit may be lowered to any value that is greater than or equal to the soft limit. The hard limit can be raised only by a process which has superuser authority. Both the soft limit and hard limit can be changed by prlimit() call.

The value RLIM_INFINITY defined in <sys/resource.h>, is considered to be larger than any other limit value. If a call to prlimit() returns RLIM_INFINITY for a resource, it means the implementation does not enforce limits on that resource. Specifying RLIM_INFINITY as any resource limit values on a successful call to prlimit() inhibits enforcement of that resource limit.

The resource argument specifies which resource to set the hard and/or soft limits for, and may be one of the following values:

RLIMIT_AS

The maximum address space size for the process, in bytes. If the limit is exceeded, malloc() and

mmap() functions will fail with an errno of ENOMEM. Also, automatic stack growth will fail.

RLIMIT_CORE

The maximum size of a dump of memory (in bytes) allowed for the process. A value of 0 (zero)

prevents file creation. Dump file creation will stop at this limit.

RLIMIT_CPU

The maximum amount of CPU time (in seconds) allowed for the process. If the limit is exceeded, a

SIGXCPU signal is sent to the process and the process is granted a small CPU time extension to allow for signal generation and delivery. If the extension is used up, the process is terminated with a

SIGKILL signal. An attempt to set the CPU limit lower than that already used will result in an EINVAL errno.

RLIMIT_DATA

The maximum size of the break value for the process, in bytes. In this implementation, this resource

always has a hard and soft limit value of RLIM_INFINITY. A call to prlimit() to set this resource to

any value other than RLIM_INFINITY will fail with an errno of EINVAL.

RLIMIT_FSIZE

The maximum file size (in bytes) allowed for the process. A value of 0 (zero) prevents file creation. If

the size is exceeded, a SIGXFSZ signal is sent to the process. If the process is blocking, catching, or ignoring SIGXFSZ, continued attempts to increase the size of a file beyond the limit will fail with an errno of EFBIG.

RLIMIT_MEMLIMIT

The maximum amount of usable storage above the 2 gigabyte bar (in 1 megabyte segments) that can be allocated. Any attempt to extend the usable amount of virtual storage above the 2 gigabyte bar fails.

RLIMIT_NOFILE

The maximum number of open file descriptors allowed for the process. This number is one greater

than the maximum value that may be assigned to a newly created descriptor. (That is, it is one-based.)

Any function that attempts to create a new file descriptor beyond the limit will fail with an EMFILE

errno. An attempt to set the open file descriptors limit lower than that already used will result in an

EINVAL errno.

Restrictions: This value may not exceed 524288.

RLIMIT_STACK

The maximum size of the stack for a process, in bytes. Note that in z/OS UNIX services, the stack is a per-thread resource. In this implementation, this resource always has a hard and soft limit value of RLIM_INFINITY. A call to prlimit() to set this resource to any value other than RLIM_INFINITY

will fail with an errno of EINVAL.

The rlp argument points to a rlimit structure. This structure contains the following members:

rlim_cur

The current (soft) limit

rlim_max

The maximum (hard) limit

If the new_limit argument is not NULL, then the rlimit structure to which it points is used to set new values for the soft and hard limits for resource. If the old_limit argument is not NULL, then a successful call to prlimit() places the previous soft and hard limits for resource in the rlimit structure pointed to by old_limit. If the new_limit and old_limit arguments are all NULL, no operation will be taken. If the new_limit and old_limit arguments are all not NULL, get operation is performed first.

Refer to the <sys/resource.h> header for more detail.

The resource limit values are propagated across exec and fork.

Special behavior for z/OS UNIX Services:

An exception exists for exec processing in conjunction with daemon support. If a daemon process invokes exec and it had previously invoked setuid() before exec, the RLIMIT_CPU, RLIMIT_AS, RLIMIT_CORE, RLIMIT_FSIZE, and RLIMIT_NOFILE limit values are set based on the limit values specified in the kernel parmlib member BPXPRMxx.

For processes which are not the only process within an address space, the RLIMIT_CPU and RLIMIT_AS limits are shared with all the processes within the address space. For RLIMIT_CPU, when the soft limit is exceeded, action will be taken on the first process within the address space. If the action is termination, all processes within the address space will be terminated.

In addition to the RLIMIT_CORE limit values, the dump file defaults are set by SYSMDUMP defaults. Refer to z/OS MVS Initialization and Tuning Reference for more information on setting up SYSMDUMP defaults using the IEADMR00 parmlib member.

Dumps of memory are taken in 4160 byte increments. Therefore, RLIMIT_CORE values affect the size of memory dumps in 4160 byte increments. For example, if the RLIMIT_CORE soft limit value is 4000, the dump will contain no data. If the RLIMIT_CORE soft limit value is 8000, the maximum size of a memory dump is 4160 bytes.

When setting RLIMIT_NOFILE, the hard limit cannot exceed the system defined limit of 524288.

Returned Value

If successful, *prlimit()* returns zero.

If unsuccessful, prlimit*()* returns -1 and set *errno* to one of the following values:

Error Code

Description

EINVAL

An invalid resource was specified, or the soft limit to set exceeds the hard limit to set, the soft limit to set is below the current usage, or the resource does not allow any value other than RLIM_INFINITY.

EMVSSAF2ERR

A Security product internal error has occurred. Consult the Reason_code parameter for the exact reason for the error.

EPERM

The limit specified to prlimit() would have raised the maximum limit value, and the calling process

does not have appropriate privileges.

Related Information

- “getrlimit() — Get current or maximum resource consumption” on page xxx
- “setrlimit() — Control maximum resource consumption” on page xxx

psignal() — print signal description

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1-2008 Single UNIX Specification, Version 4	both	

Format

```
#define _POSIX_C_SOURCE 200809L

#include <signal.h>

void psignal(int sig, const char *s);
```

General Description

The **psignal**() function displays a message on *stderr* consisting of the string *s*, a colon, a space, a string describing the signal number *sig*, and a trailing newline. If the string s is NULL or empty, the colon and space are omitted. If *sig* is invalid, the message displayed will indicate an unknown signal.

The psignal() function will not change the orientation of the stderr stream.

psignal_unlocked() is functionally equivalent to psignal() with the exception that it is not thread-safe. This function can safely be used in a multithreaded application if and only if it is called while the invoking thread owns the (FILE*) object, as is the case after a successful call to either the flockfile() or ftrylockfile() function.

Returned Value

The psignal() function returns no values.

Error Code Description

No errors are defined.

Related Information

- “signal.h — Exception handling” on page XX

readlinkat() — Read value of a symbolic link relative to a directory file descriptor

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1-2008 Single UNIX Specification, Version 4	both	

Format

```
#define _POSIX_C_SOURCE  200809L

#include <fcntl.h>                /* Definition of AT_* constants */

#include <unistd.h>

ssize_t readlinkat(int dirfd, const char *pathname, char *buf, size_t bufsize);
```

General Description

The function readlinkat() operates in exactly the same way as readlink(), except for the differences described here.

If the pathname given in pathname is relative, then it is interpreted relative to the directory referred to by the file descriptor dirfd, rather than relative to the current working directory of the calling process, as is done by readlink() for a relative pathname.

If pathname is relative and dirfd is the special value AT_FDCWD, then pathname is interpreted relative to the current working directory of the calling process.

If pathname is absolute, then dirfd is ignored.

Returned value
On success, it returns the number of bytes placed in buf. (If the returned value equals bufsize, then truncation may have occurred.) On error, -1 is returned and errno is set to indicate the error.

Error Code Description

EACCES

Search permission is denied for a component of the path prefix.

EBADF

pathname is relative but dirfd is neither AT_FDCWD nor a valid file descriptor.

EINVAL

The named file is not a symbolic link, or there was a problem with the supplied buffer.

ELOOP

A loop exists in symbolic links. This error is issued if more than POSIX_SYMLOOP symbolic links are encountered during resolution of the path argument.

ENAMETOOLONG

pathname is longer than PATH_MAX characters, or some component of pathname is longer than NAME_MAX characters while _POSIX_NO_TRUNC is in effect. For symbolic links, the length of the path name string substituted for a symbolic link exceeds PATH_MAX. The PATH_MAX and NAME_MAX values can be determined using pathconf().

ENOENT

The named file does not exist, or pathname is a relative pathname and dirfd refers to a directory that has been deleted.

ENOTDIR

A component of the path prefix is not a directory, or pathname is relative and dirfd is a file descriptor referring to a file other than a directory.

Related Information

- “unistd.h — Implementation-specific functions” on page XX
- “readlink() — Read the value of a symbolic link” on page XX
- “symlink() — Create a symbolic link to a path name” on page XX
- “unlink() — Remove a directory entry” on page XX

removexattr(), lremovexattr(), fremovexattr() — Remove an extended attribute

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _XPLATFORM_SOURCE

#include <sys/xattr.h>

int removexattr(const char *path, const char *name);

int lremovexattr(const char *path, const char *name);

int fremovexattr(int fd, const char *name);
```

General Description

Extended attributes are extensions to the normal attributes which are associated with all inodes in the system. They are used as *name:value* pairs associated permanently with files and directories to provide additional functionality to a filesystem.

removexattr() removes the extended attribute identified by name and associated with the given path in the filesystem. *lremovexattr()* is identical to *removexattr()*, except in the case of a symbolic link, where the extended attribute is re-moved from the link itself, not the file that it refers to.

fremovexattr() is identical to *removexattr()*, only the extended attribute is removed from the open file referred to by *fd* in place of *path*.

Returned Value

If successful, 0 is returned.

If unsuccessful, -1 is returned with *errno* set to one of the following values:

Error Code Description

ENAMETOOLONG

pathname is longer than PATH_MAX characters.

ENOATTR

The specified attribute is not set.

Attribute name is null/blank.

ENODATA

The specified attribute is not set.

Attribute name is null/blank.

ENOTSUP

Extended attributes are not supported by the filesystem.

Attempting to remove a read-only attribute.

Attempting to remove a general attribute of symbolic link.

ERANGE

Attribute name is longer than 256 characters.

EINVAL

Attempting to remove a general attribute with fremovexattr()

Related Information

- “sys/xattr.h — Extended attributes handling” on page XX
- “getxattr(), lgetxattr(), fgetxattr() — Retrieve an extended attribute value” on page XX
- “listxattr(), llistxattr(), flistxattr() — List extended attribute names” on page XX
- “setxattr(), lsetxattr(), fsetxattr() — Set an extended attribute value” on page XX

renameat() — change the name or location of a file

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1-2008 Single UNIX Specification, Version 4	both	

Format

```
#define _POSIX_C_SOURCE 200809L

#include <fcntl.h>

#include <stdio.h>

int renameat(int olddirfd, const char *oldpath, int newdirfd, const char *newpath);
```

General Description

The renameat() function changes the name of the file, from the name pointed to by *oldpath* to the name pointed to by *newpath*.

If the pathname given in *oldpath* is relative, then it is interpreted relative to the directory referred to by the file descriptor *olddirfd* (rather than relative to the current working directory of the calling process, as is done by rename() for a relative pathname).

If *oldpath* is relative and *olddirfd* is the special value AT_FDCWD, then *oldpath* is interpreted relative to the current working directory of the calling process (like rename()).

If *oldpath* is absolute, then *olddirfd* is ignored.

The interpretation of *newpath* is as for *oldpath*, except that a relative pathname is interpreted relative to the directory referred to by the file descriptor newdirfd.

Returned Value

If successful, *renameat()* returns zero.

If unsuccessful, *renameat()* returns -1 and sets *errno* to one of the following values:

Error Code Description

EACCES

The process does not have search permission on some component of the old or new path name or does not have the write permission on *oldpath* or *newpath* or on the parent directory of the file or directory to be renamed.

EAGAIN

One of the files or directories was temporarily unavailable.

EBUSY

oldpath and *newpath* specify directories, but one of them cannot be renamed because it is in use as a root or a mount point.

EINVAL

oldpath is part of the path name prefix of *newpath,* or *oldpath* or *newpath* refers to either . (dot) or .. (dot-dot).

EISDIR

newpath is a directory, but *oldpath* is not a directory.

ELOOP

A loop exists in symbolic links. This error is issued if the number of symbolic links encountered during resolution of *oldpath* or *newpath* is greater than POSIX_SYMLOOP.

ENAMETOOLONG

pathname is longer than PATH_MAX characters, or some component of pathname is longer than NAME_MAX characters while _POSIX_NO_TRUNC is in effect. For symbolic links, the length of the path name string substituted for a symbolic link exceeds PATH_MAX. The PATH_MAX and NAME_MAX values can be determined using pathconf().

ENOENT

No file or directory named *oldpath* was found, or either *oldpath* or *newpath* was not specified.

ENOSPC

The directory intended to contain *newpath* cannot be extended.

ENOTDIR

A component of the path name prefix for *oldpath* or *newpath* is not a directory, or *oldpath* is a directory and *newpath* is a file that is not a directory.

ENOTEMPTY

newpath specifies a directory, but the directory is not empty.

EROFS

Renaming would require writing on a read-only file system.

EXDEV

oldpath and *newpath* identify files or directories on different file systems. z/OS UNIX services do not support links between different files systems.

EBADF

oldpath (*newpath*) is relative but *olddirfd* (*newdirfd*) is not a valid file descriptor.

ENOTDIR

oldpath is relative and *olddirfd* is a file descriptor referring to a file other than a directory; or similar for *newpath* and *newdirfd*

Related Information

- “fcntl.h — POSIX functions for file operations” on page XX
- “stdio.h — Standard input and output” on page XX
- “rename() — Rename file” on page XX

renameat2() — change the name or location of a file

Standards

Standards / Extensions	C or C++	Dependencies
ZOS Unix	both	

Format

```
#define _XPLATFORM_SOURCE

#include <fcntl.h>

#include <stdio.h>

int renameat2(int olddirfd, const char *oldpath, int newdirfd,

              const char *newpath, unsigned int flags);
```

General Description

Function *renameat2*() has an additional flags argument. A *renameat2*() call with a zero flags argument is equivalent to *renameat*().

The *flags* argument is a bit mask consisting of zero or the following flag:

RENAME_NOREPLACE

Don't overwrite *newpath* of the rename. Return an error if *newpath* already exists.

Returned Value

If successful, *renameat()* returns zero.

If unsuccessful, *renameat2()* returns -1 and sets *errno* to one of the following values:

Error Code Description

EACCES

The process does not have search permission on some component of the old or new path name or does not have the write permission on *oldpath* or *newpath* or on the parent directory of the file or directory to be renamed.

EAGAIN

One of the files or directories was temporarily unavailable.

EBUSY

oldpath and *newpath* specify directories, but one of them cannot be renamed because it is in use as a root or a mount point.

EINVAL

oldpath is part of the path name prefix of *newpath,* or *oldpath* or *newpath* refers to either . (dot) or ..

(dot-dot).

EISDIR

newpath is a directory, but *oldpath* is not a directory.

ELOOP

A loop exists in symbolic links. This error is issued if the number of symbolic links encountered during resolution of *oldpath* or *newpath* is greater than POSIX_SYMLOOP.

ENAMETOOLONG

pathname is longer than PATH_MAX characters, or some component of pathname is longer than NAME_MAX characters while _POSIX_NO_TRUNC is in effect. For symbolic links, the length of the path name string substituted for a symbolic link exceeds PATH_MAX. The PATH_MAX and NAME_MAX values can be determined using pathconf().

ENOENT

No file or directory named *oldpath* was found, or either *oldpath* or *newpath* was not specified.

ENOSPC

The directory intended to contain *newpath* cannot be extended.

ENOTDIR

A component of the path name prefix for *oldpath* or *newpath* is not a directory, or *oldpath* is a directory and *newpath* is a file that is not a directory.

ENOTEMPTY

newpath specifies a directory, but the directory is not empty.

EROFS

Renaming would require writing on a read-only file system.

EXDEV

oldpath and *newpath* identify files or directories on different file systems. z/OS UNIX services do not support links between different files systems.

EBADF

oldpath (*newpath*) is relative but *olddirfd* (*newdirfd*) is not a valid file descriptor.

ENOTDIR

oldpath is relative and *olddirfd* is a file descriptor referring to a file other than a directory; or similar for *newpath* and *newdirfd*

EEXIST

flags contains RENAME_NOREPLACE and *newpath* already exists.

EINVAL

An invalid flag was specified in *flags*.

Related Information

- “fcntl.h — POSIX functions for file operations” on page XX
- “stdio.h — Standard input and output” on page XX
- “rename() — Rename file” on page XX

sethostname() — Set the name of the host processor

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

Berkeley sockets:

#define _XPLATFORM_SOURCE

#include <unistd.h>

*int sethostname(const char *name, size_t len);*

General description

sethostname() sets the hostname to the value given in the

character array name. The len argument specifies the number of

bytes in name. The name argument does not require a terminating null

byte.

Parameter Description

name

The character array contains the host name to be set.

len

The length of host name.

Returned value

If successful, sethostname() returns 0.

If unsuccessful, sethostname() returns -1 and sets errno to one of the following

values:

Error Code

Description

ENOTSUP

sethostname() is not supported from a process in the default UTS namespace.

EINVAL

name is not valid or, len is less than or equal to zero or larger than the maximum allowed size, or no associated UTS namespace found.

EACCES

Permission denied. The caller is not running with UID of zero and does not have access to the BPX.SUPERUSER FACILITY resource.

Usage notes

1. The caller needs to be running with a UID of zero or have access to the BPX.SUPERUSER FACILITY resource.
2. The values for the host name must conform to the following rules:
   - Maximum of 63 characters.
   - Must contain one or more tokens separated by a period (‘.’).
   - Each token must be at least one character and less than 64 characters.
   - Each token must start with a letter or number.

setns() — Allow for a thread to join a descendent namespace

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _XPLATFORM_SOURCE

#include <sched.h>

int setns(int fd, int nstype);
```

General Description

setns() will move the current process to an existing mount, network, IPC, or UTS namespace or specify the existing namespace to which subsequent children will belong.

When the fd refers to a PID file directory, one or more flags must be specified.

When the fd refers to a /proc/[pid]/ns link, only a single flag matching the namespace type referred to by the file descriptor or a 0 flag may be specified.

The following nstypes are supported by the setns service:

nstypes	Description
CLONE_NEWIPC	Reassociate the process into the IPC namespace associated with the input file descriptor If specified, the fd must refer to either an IPC link in the /proc/[pid]/ns directory or a PID file descriptor
CLONE_NEWNS	Reassociate the process into the mount namespace associated with the input file descriptor If specified, the fd must refer to either a mount link in the /proc/[pid]/ns directory or a PID file descriptor
CLONE_NEWPID	Reassociate the process into the IPC namespace associated with the input file descriptor If specified, the fd must refer to either a PID link in the /proc/[pid]/ns directory or a PID file descriptor

CLONE_NEWUTS	Reassociate the process into the UTS namespace associated with the input file descriptor If specified, the fd must refer to either a UTS link in the /proc/[pid]/ns directory or a PID file descriptor

Returned Value

If successful, *setns()* returns zero.

If unsuccessful, set*ns()* returns -1 and set *errno* to one of the following values:

Error Code Description

EBADF

The input parameter fd is invalid.

EINVAL

An invalid bit was specified in nstype. nstype supported by setns are CLONE_NEWNS, CLONE_NEWPID, CLONE_NEWIPC, CLONE_NEWNET, and CLONE_NEWUTS. Any other type will result in the operation failing with an EINVAL error.

ENOMEM

The process requires more space than is available.

EPERM

The calling process did not have the required privileges for this operation.

ESRCH

The calling process does not have appropriate privileges.

Characteristics and restrictions

Following is a list of characteristics or restrictions of setns():

The CLONE_NEWUTS flag may not be specified in a multiprocess address space. This will result in an errno EINVAL.

Related Information

- “sched.h — Manipulate and examine process execution scheduling” on page XX
- “clone() — Execute a caller specified program in a child process” on page XX
- “unshare() — Isolate a process from one or more of its namespaces” on page XX

setsockopt() — Set options associated with a socket

IP_UNBLOCK_SOURCE

(RAW and UDP) This option is used to undo the operation performed with the IP_BLOCK_SOURCE option (e.g., if the user "mutes" that source). The source group is specified by the ip_mreq_source structure which is defined in netinet/in.h.

IP_TTL

(TCP and UDP) Used to set the time-to-live field in the IP header for SOCK_STREAM(TCP) and SOCK_DGRAM (UDP) sockets. This socket option allows the application to limit the lifetime of the packet in the Internet and prevent it from circulating indefinitely. This is an IPv4-only socket option. Valid values are in the range 1–255.

Note: IP_TTL can only be used when *_XPLATFORM_SOURCE* is defined.

setxattr(), lsetxattr(), fsetxattr() — Set an extended attribute value

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _XPLATFORM_SOURCE

#include <sys/xattr.h>

int setxattr(const char *path, const char *name, const void *value, size_t size, int flags);

int lsetxattr(const char *path, const char *name, const void *value, size_t size, int flags);

int fsetxattr(int fd, const char *name, const void *value, size_t size, int flags);
```

General Description

Extended attributes are extensions to the normal attributes which are associated with all inodes in the system. They are used as *name:value* pairs associated permanently with files and directories to provide additional functionality to a filesystem.

setxattr() sets the value of the extended attribute identified by name and associated with the given path in the filesystem. The *size* argument specifies the size of value in bytes.

lsetxattr() is identical to *setxattr()*, except in the case of a symbolic link, where the extended attribute is set on the link itself, not the file that it refers to.

fsetxattr() is identical to *setxattr()*, only the extended attribute is set on the open file referred to by *fd* in place of *path*.

If the value of *flags* is set to zero, the extended attribute will be created if it does not exist, or the value will be replaced if the attribute already exists. To modify these semantics, one of the following values can be specified:

XATTR_CREATE

Perform a pure create, which fails if the named attribute exists already.

XATTR_REPLACE

Perform a pure replace operation, which fails if the named attribute does not already exist.

Extended attribute name	**Length**	**Value**
List of modifiable xattrs		
trusted.apfauth	0	N/A
trusted.sharelib	0	N/A
trusted.progctl	0	N/A
system.noshareas	0	N/A
system.filefmt	1	Hex Value
system.filetag	4	Hex Value
system.seclabel	8	Character
system.useraudit	4	Hex Value
system.auditoraudit	4	Hex Value
List of read-only xattrs		
system.auditid	16	Hex Value
system.dmodelacl	0	N/A
system.fmodelacl	0	N/A
system.accessacl	0	N/A
system.createtime	8	Hex value

Restrictions:

1. Trusted.apfauth, trusted.sharelib, trusted.progctl, system.noshareas are general attributes. Lsetxattr() / fsetxattr() / lremovexattr() / fremovexattr() don't support changing the general attributers.

2. The attributes(system.filefmt / system.filetag / system.useraudit / system.auditoraudit) cannot be set for symbolic links.

Returned Value

If successful, 0 is returned.

If unsuccessful, -1 is returned with *errno* set to one of the following values:

Error Code Description

EEXIST

XATTR_CREATE was specified, and the attribute exists already.

EINVAL

The flag option is an invalid value.

Attempting to modify general attributes with fsetxattr().

ENAMETOOLONG

pathname is longer than PATH_MAX characters.

ENOATTR

XATTR_REPLACE was specified, and the attribute does not exist.

Attribute name is NULL or blank.

ENODATA

The specified attribute is not set.

Attribute name is null/blank.

ENOTSUP

Extended attributes are not supported by the filesystem.

Attempting to modify general attributes with lsetxattr().

Attempting to modify a read only attribute.

ERANGE

The size of the value buffer is too small to hold the result.

The size of the value buffer is larger than 50000.

Attribute name is longer than PATH_MAX characters.

Related Information

- “sys/xattr.h — Extended attributes handling” on page XX
- “getxattr(), lgetxattr(), fgetxattr() — Retrieve an extended attribute value” on page XX
- “listxattr(), llistxattr(), flistxattr() — List extended attribute names” on page XX
- “removexattr(), lremovexattr(), fremovexattr() — Remove an extended attribute” on page XX

socket() — Create a socket

… …

Parameter description

... …

type

The type of socket created, either SOCK_STREAM, SOCK_DGRAM, or SOCK_RAW. In addition, with the _XPLATFORM_SOURCE feature test macro, the bitwise OR of the SOCK_NONBLOCK flag and SOCK_CLOEXEC flag can be specified with the socket type.

… …

The *type* parameter specifies the type and flags of the socket created. The type is analogous with the semantics of the communication requested. These socket type constants are defined in the sys/socket.h include file. The types supported are:

… …

Socket Type Description

SOCK_STREAM

Provides sequenced, two-way byte streams that are reliable and connection-oriented. They support a mechanism for out-of-band data. This type is supported in the AF_INET, AF_INET6, and AF_UNIX

domains.

Socket Flag Description

SOCK_NONBLOCK

Set the O_NONBLOCK file status flag on the open file description

SOCK_CLOEXEC

Set the FD_CLOEXEC flag on the new file descriptor

Understanding the socket() Parameters:

… …

socketpair() — Create a pair of sockets

… …

Parameter Description

… …

type

The type of socket created, either SOCK_STREAM, or SOCK_DGRAM. With the _XPLATFORM_SOURCE feature test macro, the bitwise OR of the SOCK_NONBLOCK flag and SOCK_CLOEXEC flag can be specified with the socket type.

… …

symlinkat() — Create a symbolic link to a path name

Standards

Standards / Extensions	C or C++	Dependencies
Single UNIX Specification, Version 4 POSIX.1-2008	both	

Format

```
#define _POSIX_C_SOURCE 200809L

#include <unistd.h>

int symlinkat(const char *pathname, int newdirfd, const char *slink);
```

General Description

The symlinkat() function operates in exactly the same way as symlink(), except for the differences described here. If the pathname given in slink is relative, then it is interpreted relative to the directory referred to by the file descriptor newdirfd (rather than relative to the current working directory of the calling process, as is done by symlink for a relative pathname).

If slink is relative and newdirfd is the special value AT_FDCWD, then slink is interpreted relative to the current working directory of the calling process (like symlink()).

If slink is absolute, then newdirfd is ignored.

Returned Value

If successful, symlinkat() returns 0.

If unsuccessful, -1 is returned with errno set to one of the following values:

Error Code Description

EACCES

A component of the slink path prefix denies search permission, or write permission is denied in the parent directory of the symbolic link to be created.

EEXIST

The file named by slink already exists.

EINVAL

There is a NULL character in pathname.

EIO

Added for XPG4.2: An I/O error occurred while reading from the file system.

ELOOP

A loop exists in symbolic links. This error is issued if more than POSIX_SYMLOOP symbolic links are encountered during resolution of the slink argument.

ENAMETOOLONG

pathname is longer than PATH_MAX characters, or some component of pathname is longer than NAME_MAX characters while _POSIX_NO_TRUNC is in effect. For symbolic links, the length of the path name string substituted for a symbolic link exceeds PATH_MAX. The PATH_MAX and NAME_MAX values can be determined with pathconf().

ENOENT

Added for XPG4.2: A component of slink does not name an existing file or slink is an empty string. This might be also returned for the following reason:

slink has a slash as its last component, which indicates that the preceding component is a directory. A symbolic link cannot be a directory.

ENOSPC

The new symbolic link cannot be created because there is no space left on the file system that will contain the symbolic link.

ENOTDIR

slink is relative and newdirfd is a file descriptor referring to a file other than a directory.

EROFS

The file slink cannot reside on a read-only system.

EBADF

newdirfd is not a valid file descriptor.

EFBIG

A request to create a symbolic link is prohibited because the file size limit for the process is set to 0.

Related Information

- “unistd.h — Implementation-specific functions” on page XX
- “link() — Create a link to a file” on page XX
- “readlink() — Read the value of a symbolic link” on page XX
- “symlink() — Create a symbolic link to a path name” on page XX

syncfs() — Schedule specific file system updates

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _XPLATFORM_SOURCE

#include <unistd.h>

int syncfs(int fd);
```

General Description

Syncfs() is like sync(), but synchronizes just the filesystem containing file referred to by the open file descriptor fd.

Returned value

On success, syncfs() returns 0. On error, it returns -1 and sets errno to indicate the error.

Error Code Description

EBADF

fd is not a valid file descriptor.

EIO

An error occurred during synchronization. This error may relate to data written to any file on the filesystem, or on metadata related to the filesystem itself.

ENOSPC

Disk space was exhausted while synchronizing.

Related Information

- “unistd.h — Implementation-specific functions” on page XX
- “fsync() — Write changes to direct-access storage” on page XX
- “sync() — Schedule file system updates” on page XX

umount() — Remove a virtual file system

… …

Format

#include <sys/stat.h>

int umount(const char *filesystem, mtm_t mtm);

#define _OPEN_SYS

#define _XPLATFORM_SOURCE

#include <sys/mount.h>

int umount(const char *target);

General Description

The umount() function defined under different feature test macros provides different input parameters and functionalities. For details, see the following respective descriptions.

When Only _OPEN_SYS is defined:

Removes a file system from the file hierarchy or changes the mount mode of a mounted file system between read-only and read/write.

… …

MTM_SAMEMODE

A remount request to unmount and mount back without changing the mount mode. If either

MTM_RDONLY or MTM_RDWR is also specified, it must be the current state. This can be used to attempt to regain use of a file system that has had I/O errors.

When both _OPEN_SYS and _XPLATFORM_SOURCE are defined:

umount() remove the attachment of the (topmost) filesystem mounted on *target*.

In order to unmount a file system, the caller must be an authorized program, or must be running for a user with appropriate privileges.

Returned value

… …

Error Code Description

When only _OPEN_SYS is defined:

EBUSY

… …

EPERM

Superuser authority is required to issue an unmount.

When both _OPEN_SYS and _XPLATFORM_SOURCE are defined:

EBUSY

target is busy, for one of these reasons:

- A umount() was requested, and the attached file system still has open files or other file systems mounted under it.
- A file system is currently mounted on the requested file system.
- There is a unmount request already in progress for the specified file system.

EINTR

umount() was interrupted by a signal.

EINVAL

target is not a mount point.

EIO

An I/O error occurred.

ENAMETOOLONG

A pathname was longer than PATH_MAX, or some component of path name is longer that NAME_MAX characters.

ENOENT

A target was empty or had a nonexistent component.

ENOTDIR

A component of the path prefix is not a directory.

EPERM

Superuser authority is required to issue an unmount.

Example

… …

Related Information

• “sys/mount.h - File system mount and unmount” on page XX

• “sys/stat.h — z/OS UNIX files and access” on page 74

• “mount() — Make a file system available” on page 1043

• “umount2() – Unmount file system” on page XX

umount2() — Unmount file system

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

#define _XPLATFORM_SOURCE

#include <sys/mount.h>

int umount2(const char *target, int flags);

General Description

umount2() remove the attachment of the (topmost) filesystem mounted on *target*.

In order to unmount a file system, the caller must be an authorized program, or must be running for a user with appropriate privileges.

The *flags* specifies additional flags which controls the behavior of the operation, it can be:

MNT_DETACH

Perform a lazy unmount: make the mount unavailable for new accesses, immediately disconnect the filesystem and all filesystems mounted below it from each other and from the mount table, and actually perform the unmount when the mount ceases to be busy.

An unmount drain request. The requester is willing to wait for all uses of this file system to be ended normally before the file system is unmounted.

Returned value

If successful, umount2() returns 0.

If unsuccessful, umount2() returns -1 and sets errno to one of the following values:

Error Code Description

EBUSY

target could not be unmounted because it is busy, for one of these reasons:

- A umount2() was requested, and the attached file system still has open files or other file systems mounted under it.
- A file system is currently mounted on the requested file system.
- There is a unmount request already in progress for the specified file system.

EINTR

umount2() was interrupted by a signal.

EINVAL

target is not a mount point.

EINVAL

umount2() was called with an invalid flag value in flags.

EIO

An I/O error occurred.

ENAMETOOLONG

A target was longer than PATH_MAX, or some component of path name is longer that

NAME_MAX characters.

ENOTDIR

A component of the path prefix is not a directory.

ENOENT

A target was empty or had a nonexistent component.

EPERM

Superuser authority is required to issue an unmount.

Related Information

- sys/mount.h – File system mount and unmount on page XX
- mount() – Make a file system available on page XX
- umount() – Remove a virtual file system on page XX

unlinkat() — Remove a directory or directory entry

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1-2008 Single UNIX Specification, Version 4	both	

Format

```
#define _POSIX_C_SOURCE 200809L

#include <fcntl.h>           /* Definition of AT_* constants */

#include <unistd.h>

int unlinkat(int dirfd, const char *pathname, int flags);
```

General Description

The unlinkat() function operates in exactly the same way as either unlink() or rmdir() (depending on whether or not flags includes the AT_REMOVEDIR flag) except for the differences described here.

If the pathname given in pathname is relative, then it is interpreted relative to the directory referred to by the file descriptor dirfd (rather than relative to the current working directory of the calling process, as is done by unlink() and rmdir() for a relative pathname).

If the pathname given in pathname is relative and dirfd is the special value AT_FDCWD, then pathname is interpreted relative to the current working directory of the calling process (like unlink() and rmdir()).

If the pathname given in pathname is absolute, then dirfd is ignored.

flags is a bit mask that can either be specified as 0, or by ORing together flag values that control the operation of unlinkat(). Currently, only one such flag is defined:

AT_REMOVEDIR

By default, unlinkat() performs the equivalent of unlink() on pathname. If the AT_REMOVEDIR flag is specified, then performs the equivalent of rmdir() on pathname.

Returned Value

If successful, unlinkat() returns zero.

If unsuccessful, unlinkat() returns -1 and sets errno to one of the following values:

Error Code Description

EACCES

The process did not have search permission for some component of pathname, or did not have write permission for the directory containing the link to be removed.

EBADF

The file descriptor specified for Dirfd is incorrect.

EBUSY

pathname cannot be unlinked because it is currently being used by the system or some other process.

EINVAL

One of the input parameters was not valid.

EISDIR

pathname refers to a directory.

ELOOP

A loop exists in symbolic links. This error is issued if more than POSIX_SYMLOOP symbolic links are encountered during resolution of the slink argument.

ENAMETOOLONG

Pathname is longer than 1023 characters, or a component of the path name is longer than 255 characters.

ENOENT

A component of path does not name an existing file or path is an empty string.

ENOTDIR

A component of the path prefix is not a directory.

EROFS

The file resides on a read-only file system.

Related Information

- “fcntl.h — POSIX functions for file operations” on page XX
- “unistd.h — Implementation-specific functions” on page XX
- “close() — Close a file” on page XX
- “link() — Create a link to a file” on page XX
- “open() — Open a file” on page 1099
- “remove() — Delete file” on page XX
- “rmdir() — Remove a directory” on page XX
- “unlink() — Remove a directory entry” on page XX

unshare() — isolate a process from one or more of its namespaces.

Standards

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

```
#define _XPLATFORM_SOURCE
#include <sched.h>

int unshare(int flags);
```

General Description

The unshare() will disassociate the current process from a mount, network, IPC, and/or UTS namespace into a newly created namespace or disassociate subsequent children from the PID namespace.

The *flags* argument is a bit mask that specifies which namespaces are to be disassociated by the process and a new namespace created. This argument is specified by zero or more of the following supported constants:

Flag	Description
CLONE_NEWIPC	Unshare the process from the IPC namespace and move into a new IPC namespace
CLONE_NEWNS	Unshare the process from the mount namespace and move into a new mount namespace
CLONE_NEWPID	Unshare the PID namespace so the children of the process will have a new PID namespace
CLONE_NEWUTS	Unshare the process from the UTS namespace and move into a new UTS namespace

If flags is specified as zero, then unshare() is a no-op; no changes are made to the calling process's execution context and unshare() will indicate success on return.

Returned Value

If successful, unshare() returns zero.

If unsuccessful, unshare() returns -1 and set errno to one of the following values:

Error Code

Description

EINVAL

An invalid bit was specified in *flags*. Flags supported by unshare() are CLONE_NEWNS, CLONE_NEWPID, CLONE_NEWIPC, CLONE_NEWNET, and CLONE_NEWUTS. Any other flag will result in the operation failing with an EINVAL error.

ENOMEM

The process requires more space than is available.

ENOSPC

A system limit is reached.

EPERM

The calling process did not have the required privileges for this operation.

Characteristics and restrictions

Following is a list of characteristics or restrictions of unshare():

The CLONE_NEWNET or CLONE_NEWUTS flags may not be specified in a multi-process address space. This will result in an errno EINVAL.

Related Information

- “sched.h — Manipulate and examine process execution scheduling” on page XX
- “clone() — Execute a caller specified program in a child process” on page XX
- “setns() — Allow for a thread to join a descendent namespace” on page XX

utimensat() — Change file timestamps with nanosecond precision

Standards

Standards / Extensions	C or C++	Dependencies
POSIX.1-2008 Single UNIX Specification, Version 4	both	

Format

```
#define _POSIX_C_SOURCE  200809L

#include <fcntl.h>                      /* Definition of AT_* constants */

#include <sys/stat.h>

int utimensat(int dirfd, const char *pathname, const struct timespec times[2], int flags);
```

General Description

The utimensat() updates the timestamps of a file with nanosecond precision. This contrasts with the historical utime() and utimes(), which permit only second and microsecond precision, respectively, when setting file timestamps.

The new file timestamps are specified in the array times: times[0] specifies the new “last access time” (atime); times[1] specifies the new “last modification time” (mtime). Each of the elements of times specifies a time as the number of seconds and nanoseconds since the Epoch, 1970-01-01 00:00:00 + 0000 (UTC). This information is conveyed in a structure of the following form:

```
struct timespec {

  time_t tv_sec;        /* seconds */
```

```
    long   tv_nsec;       /* nanoseconds */
};
```

If the tv_nsec field of one of the timespec structures has the special value **UTIME_NOW**, then the corresponding file timestamp is set to the current time. If the tv_nsec field of one of the timespec structures has the special value **UTIME_OMIT**, then the corresponding file timestamp is left unchanged. In both of these cases, the value of the corresponding tv_sec field is ignored.

If times is NULL, then both timestamps are set to the current time.

If pathname is relative, then by default it is interpreted relative to the directory referred to by the open file descriptor, dirfd (rather than relative to the current working directory of the calling process, as is done by utimes() for a relative pathname).

If pathname is absolute, then dirfd is ignored.

The flags field is a bit mask that may be 0, or include the following constant, defined in <fcntl.h>:

AT_SYMLINK_NOFOLLOW

If pathname specifies a symbolic link, then update the timestamps of the link, rather than the file to which it refers.

Return value

On success, utimensat() returns 0. On error, -1 is returned and errno is set to indicate the error.

Error code Description

EACCES

The process does not have appropriate permissions. Possible reasons include:

- The process is attempting to set access time or modification time to current time (times is NULL), but the effective UID of the process does not match the file's owner.
- The process does not have write permission on the file.
- The process does not have appropriate privileges.

EBADF

pathname is relative but dirfd is neither AT_FDCWD nor a valid file descriptor, or dirfd is not a valid file descriptor.

EBUSY

The file is open by a remote NFS client with a share reservation that conflicts with the requested operation.

EINVAL

- Invalid value in flags.
- pathname is NULL, dirfd is not **AT_FDCWD**, and flags contains **AT_SYMLINK_NOFOLLOW**.
- Invalid value in one of the tv_nsec fields (value outside range 0 to 999,999,999, and not UTIME_NOW or UTIME_OMIT); or an invalid value in one of the tv_sec fields.

ELOOP

A loop exists in symbolic links. This error is issued if more than POSIX_SYMLOOP symbolic links are detected in the resolution of pathname.

EMVSERR

An MVS™ environmental error has been detected.

ENAMETOOLONG

pathname is longer than PATH_MAX characters, or some component of pathname is longer than NAME_MAX characters while _POSIX_NO_TRUNC is in effect. For symbolic links, the length of the pathname string substituted for a symbolic link exceeds PATH_MAX.

ENOENT

There is no file named pathname, or the pathname argument is an empty string.

ENOTDIR

Some component of the pathname prefix is not a directory.

EPERM

times is not NULL, the effective user ID of the calling process does not match the owner of the file, and the calling process does not have appropriate privileges.

EROFS

pathname is on a read-only file system.

Related Information

- “futimes — change file timestamps” on page XX
- “futimesat() — change timestamps of a file relative to a directory file descriptor” on page XX
- “openat() — open a file relative to a directory file descriptor” on page XX
- “stat() — get file information” on page XX
- “utimes — change file last access and modification times” on page XX

wait3(), wait4() — Wait for child process to change state

Standards

wait3():

Standards / Extensions	C or C++	Dependencies
XPG4.2	both	

wait4():

Standards / Extensions	C or C++	Dependencies
z/OS UNIX	both	

Format

wait3():

```
#define _XOPEN_SOURCE_EXTENDED 1

#include <sys/wait.h>

pid_t wait3(int *stat_loc, int options, struct rusage *resource_usage);
```

wait4():

```
#define _XOPEN_SOURCE_EXTENDED 1
#define _XPLATFORM_SOURCE
#include <sys/wait.h>

pid_t wait4(pid_t pid, int *wstatus, int options, struct rusage *resource_usage);
```

General description

The wait3() and wait4() function allows the calling process to obtain status information for specified child processes.

The following wait3() call:

```
wait3(stat_loc, options, resource_usage)
```

is equivalent to the call:

```
waitpid((pid_t)-1, stat_loc, options);
```

and the following wait4() call:

```
wait4(pid, wstatus, options, resource_usage);
```

is equivalent to the call:

```
waitpid(pid, wstatus, options);
```

except that on successful completion, if the resource_usage argument to wait3() or wait4() is not a NULL pointer, the rusage structure that the third argument points to is filled in for the child process identified by the return value.

In other words, wait3() waits for any child, while wait4() can be used to select a specific child, or any child, on which to wait. With a pid argument of -1, wait4() is equivalent to wait3(). See wait() for further details.

Note:

… …

Returned value

… …

In addition to the error conditions specified on waitpid(), under the following conditions, wait3() and wait4() may fail and set errno to one of the following values:

Error Code

Description

… …

ESRCH

pid is equal to INT_MIN (wait4() only).

Related information

… …

Language Environment Runtime Messages

Chapter 4. XL C/C++ runtime messages

EDC7029E

The getopt_long() function detected an invalid option option_string when it was invoked from program program_name.

Explanation
The getopt_long() function detected that an option that was parsed was not one of the recognized set of specified options.

System action
The getopt_long() function returns the option in error. The application continues to run.

Programmer response
Respecify a recognized option.

Symbolic Feedback Code
EDC6RL

EDC7030E

The getopt_long() function detected an option option_string that is missing an argument when it was invoked from program program_name.

Explanation
The getopt_long() function encountered an option that required an option-argument, but the option-argument was not found.

System action
The getopt_long() function returns the option in error. The application continues to run.

Programmer response
Respecify the option with an option-argument.

Symbolic Feedback Code
EDC6RM

Chapter 9. Language Environment errno2 values

CB270048

Explanation: name is not valid or, len is zero or larger than the maximum allowed size.
Programmer response: Correct the argument.
Symbolic Feedback Code: JrEdcXfr5Einval41