NAME

dup - duplicate an open file descriptor

SYNOPSIS

```
#include <unistd.h>

int dup(int fildes);
```

DESCRIPTION

The dup() function returns a new file descriptor having the following in common with the original open file descriptor fildes:

- o same open file (or pipe)
- o same file pointer (that is, both file descriptors share one file pointer)
- o same access mode (read, write or read/write).

The new file descriptor is set to remain open across exec functions (see fcntl(2)).

The file descriptor returned is the lowest one available.

The dup(fildes) function call is equivalent to:

```
fcntl(fildes, F_DUPFD, 0)
```

RETURN VALUES

Upon successful completion, a non-negative integer representing the file descriptor is returned. Otherwise, -1 is returned and errno is set to indicate the error.

ERRORS

The dup() function will fail if:

EBADF The fildes argument is not a valid open file descriptor.

EINTR A signal was caught during the execution of the dup() function.

EMFILE The process has too many open files (see getrlimit(2)).

ENOLINK The fildes argument is on a remote machine and the link to that machine is no longer

active.

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	Async-Signal-Safe

SEE ALSO

close(2), creat(2), exec(2), fcntl(2), getrlimit(2), open(2), pipe(2), dup2(3C), lockf(3C), attributes(5), standards(5)

No manual entry for 3.

NAME

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

SYNOPSIS

```
#include <unistd.h>

int execl(const char *path, const char *arg0,
     ... /* const char *argn, (char *)0 */);

int execv(const char *path, char *const argv[]);

int execle(const char *path, const char *arg0,
     ... /* const char *argn, (char *)0,char *const envp[]*/);

int execve(const char *path, char *const argv[],
     char *const envp[]);

int execlp(const char *file, const char *arg0,
     ... /* const char *argn, (char *)0 */);

int execvp(const char *file, char *const argv[]);
```

DESCRIPTION

Each of the functions in the exec family replaces the current process image with a new process image. The new image is constructed from a regular, executable file called the new process image file. This file is either an executable object file or a file of data for an interpreter. There is no return from a successful call to one of these functions because the calling process image is overlaid by the new process image.

An interpreter file begins with a line of the form

```
#! pathname [arg]
```

where pathname is the path of the interpreter, and arg is an optional argument. When an interpreter file is executed, the system invokes the specified interpreter. The pathname specified in the interpreter file is passed as arg0 to the interpreter. If arg was specified in the interpreter file, it is passed as arg1 to the interpreter. The remaining

arguments to the interpreter are arg0 through argn of the originally exec'd file. The interpreter named by pathname must not be an interpreter file.

When a C-language program is executed as a result of this call, it is entered as a C-language function call as follows:

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

where argc is the argument count and argv is an array of character pointers to the arguments themselves. In addition, the following variable:

```
extern char **environ;
```

is initialized as a pointer to an array of character pointers to the environment strings. The argv and environ arrays are each terminated by a null pointer. The null pointer terminating the argv array is not counted in argc.

The value of argc is non-negative, and if greater than 0, argv[0] points to a string containing the name of the file. If argc is 0, argv[0] is a null pointer, in which case there are no arguments. Applications should verify that argc is greater than 0 or that argv[0] is not a null pointer before dereferencing argv[0].

The arguments specified by a program with one of the exec functions are passed on to the new process image in the main() arguments.

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

The file argument is used to construct a pathname that identifies the new process image file. If the file argument contains a slash character, it is used as the pathname for this file. Otherwise, the path prefix for this file is obtained by a search of the directories passed in the PATH environment variable (see environ(5)). The environment is supplied typically by the shell. If the process image file is not a valid executable object file, execlp() and execvp() use the

contents of that file as standard input to the shell. In this case, the shell becomes the new process image. The standard to which the caller conforms determines which shell is used. See standards(5).

The arguments represented by arg0... are pointers to null-terminated character strings. These strings constitute the argument list available to the new process image. The list is terminated by a null pointer. The arg0 argument should point to a filename that is associated with the process being started by one of the exec functions.

The argv argument is an array of character pointers to null-terminated strings. The last member of this array must be a null pointer. These strings constitute the argument list available to the new process image. The value in argv[0] should point to a filename that is associated with the process being started by one of the exec functions.

The envp argument is an array of character pointers to null-terminated strings. These strings constitute the environment for the new process image. The envp array is terminated by a null pointer. For execl(), execv(), execvp(), and execlp(), the C-language run-time start-off

routine places a pointer to the environment of the calling process in the global object extern char **environ, and it is used to pass the environment of the calling process to the new process image.

The number of bytes available for the new process's combined argument and environment lists is ARG_MAX. It is implementation-dependent whether null terminators, pointers, and/or any alignment bytes are included in this total.

File descriptors open in the calling process image remain open in the new process image, except for those whose close-on-exec flag FD_CLOEXEC is set; see fcntl(2). For those file descriptors that remain open, all attributes of the open file description, including file locks, remain unchanged.

The preferred hardware address translation size (see memcntl(2)) for the stack and heap of the new process image are set to the default system page size.

Directory streams open in the calling process image are closed in the new process image.

The state of conversion descriptors and message catalogue descriptors in the new process image is undefined. For the new process, the equivalent of:

```
setlocale(LC_ALL, "C")
```

is executed at startup.

Signals set to the default action (SIG_DFL) in the calling process image are set to the default action in the new process image (see signal(3C)). Signals set to be ignored (SIG_IGN) by the calling process image are set to be ignored by the new process image. Signals set to be caught by the calling process image are set to the default action in the new process image (see signal.h(3HEAD)). After a successful call to any of the exec functions, alternate signal stacks are not preserved and the SA_ONSTACK flag is cleared for all signals.

After a successful call to any of the exec functions, any functions previously registered by atexit(3C) are no longer registered.

The saved resource limits in the new process image are set to be a copy of the process's corresponding hard and soft resource limits.

If the ST_NOSUID bit is set for the file system containing the new process image file, then the effective user ID and effective group ID are unchanged in the new process image. If the set-user-ID mode bit of the new process image file is set (see chmod(2)), the effective user ID of the new process image is set to the owner ID of the new process image file. Similarly, if the set-group-ID mode bit of the new process image file is set, the effective group ID of the new process image is set to the group ID of the new process image file.

The real user ID and real group ID of the new process image remain the same as those of the calling process image. The effective user ID and effective group ID of the new process image are saved (as the saved set-user-ID and the saved set-group-ID for use by setuid(2).

The privilege sets are changed according to the following rules:

1. The inheritable set, I, is intersected with the limit set, L. This mechanism enforces the limit set for processes.

2. The effective set, E, and the permitted set, P, are made equal to the new inheritable set.

The system attempts to set the privilege-aware state to non-PA both before performing any modifications to the process IDs and privilege sets as well as after completing the transition to new UIDs and privilege sets, following the rules outlined in privileges(5).

If the {PRIV_PROC_OWNER} privilege is asserted in the effective set, the set-user-ID and set-group-ID bits will be honored when the process is being controlled by ptrace(3C). Additional restriction can apply when the traced process has an effective UID of 0. See privileges(5).

Any shared memory segments attached to the calling process image will not be attached to the new process image (see shmop(2)). Any mappings established through mmap() are not preserved across an exec. Memory mappings created in the process are unmapped before the address space is rebuilt for the new process image. See mmap(2).

Memory locks established by the calling process via calls to mlockall(3C) or mlock(3C) are removed. If locked pages in the address space of the calling process are also mapped into the address spaces the locks established by the other processes will be unaffected by the call by this process to the exec function. If the exec function fails, the effect on memory locks is unspecified.

If _XOPEN_REALTIME is defined and has a value other than -1, any named semaphores open in the calling process are closed as if by appropriate calls to sem_close(3RT)

Profiling is disabled for the new process; see profil(2).

Timers created by the calling process with timer_create(3RT) are deleted before replacing the current process image with

the new process image.

For the SCHED_FIFO and SCHED_RR scheduling policies, the

policy and priority settings are not changed by a call to an exec function.

All open message queue descriptors in the calling process are closed, as described in mq_close(3RT).

Any outstanding asynchronous I/O operations may be cancelled. Those asynchronous I/O operations that are not canceled will complete as if the exec function had not yet occurred, but any associated signal notifications are suppressed. It is unspecified whether the exec function itself blocks awaiting such I/O completion. In no event, however, will the new process image created by the exec function be affected by the presence of outstanding asynchronous I/O operations at the time the exec function is called.

All active contract templates are cleared (see contract(4)).

The new process also inherits the following attributes from the calling process:

- o nice value (see nice(2))
- o scheduler class and priority (see priocntl(2))
- o process ID
- o parent process ID
- o process group ID
- o task ID
- o supplementary group IDs
- o semadj values (see semop(2))
- o session membership (see exit(2) and signal(3C))
- o real user ID
- o real group ID

- o project ID
- o trace flag (see ptrace(3C) request 0)
- o time left until an alarm clock signal (see alarm(2))
- o current working directory
- o root directory
- o file mode creation mask (see umask(2))
- o file size limit (see ulimit(2))
- o resource limits (see getrlimit(2))

- o tms_utime, tms_stime, tms_cutime, and tms_cstime (see times(2))
- o file-locks (see fcntl(2) and lockf(3C))
- o controlling terminal
- o process signal mask (see sigprocmask(2))
- o pending signals (see sigpending(2))
- o processor bindings (see processor_bind(2))
- o processor set bindings (see pset_bind(2))
- o limit privilege set
- o privilege debugging flag (see privileges(5) and getpflags(2))

A call to any exec function from a process with more than one thread results in all threads being terminated and the new executable image being loaded and executed. No destructor functions will be called.

Upon successful completion, each of the functions in the exec family marks for update the st_atime field of the file. If an exec function failed but was able to locate the process image file, whether the st_atime field is marked for update is unspecified. Should the function succeed, the process image file is considered to have been opened with open(2). The corresponding close(2) is considered to occur at a time after this open, but before process termination or

successful completion of a subsequent call to one of the exec functions. The argv[] and envp[] arrays of pointers and the strings to which those arrays point will not be modified by a call to one of the exec functions, except as a consequence of replacing the process image.

The saved resource limits in the new process image are set to be a copy of the process's corresponding hard and soft limits.

RETURN VALUES

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

ERRORS

The exec functions will fail if:

E2BIG The number of bytes in the new process's argument list is greater than the system-imposed limit of {ARG_MAX} bytes. The argu-

ment list limit is sum of the size of the argument list plus the size of the environment's exported shell variables.

EACCES Search permission is denied for a directory listed in the new process file's path prefix.

The new process file is not an ordinary file.

The new process file mode denies execute permission.

The {FILE_DAC_SEARCH} privilege overrides the restriction on directory searches.

The {FILE_DAC_EXECUTE} privilege overrides the lack of execute permission.

EAGAIN Total amount of system memory available when reading using raw I/O is temporarily insufficient.

EFAULT An argument points to an illegal address.

EINVAL The new process image file has the appropriate permission and has a recognized executable binary format, but the system does not support execution of a file with this format.

EINTR A signal was caught during the execution of one of the functions in the exec family.

ELOOP Too many symbolic links were encountered in translating path or file.

ENAMETOOLONG The length of the file or path argument exceeds {PATH_MAX}, or the length of a file or path component exceeds {NAME_MAX} while {_POSIX_NO_TRUNC} is in effect.

ENOENT One or more components of the new process path name of the file do not exist or is a null pathname.

ENOLINK The path argument points to a remote machine and the link to that machine is no longer active.

ENOTDIR A component of the new process path of the file prefix is not a directory.

The exec functions, except for execlp() and execvp(), will fail if:

ENOEXEC The new process image file has the appropriate access permission but is not in the proper format.

The exec functions may fail if:

ENAMETOOLONG Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.

ENOMEM The new process image requires more memory than is allowed by the hardware or system-imposed by memory management constraints. See brk(2).

ETXTBSY The new process image file is a pure procedure (shared text) file that is currently open for writing by some process.

USAGE

As the state of conversion descriptors and message catalogue descriptors in the new process image is undefined, portable applications should not rely on their use and should close them prior to calling one of the exec functions.

Applications that require other than the default POSIX locale should call setlocale(3C) with the appropriate parameters to establish the locale of the new process.

The environ array should not be accessed directly by the application.

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	See below.
Standard	See standards(5).

The execle() and execve() fucntions are Async-Signal-Safe.

SEE ALSO

ksh(1), ps(1), sh(1), alarm(2), brk(2), chmod(2), exit(2), fcntl(2), fork(2), getpflags(2), getrlimit(2), memcntl(2), mmap(2), nice(2), priocntl(2), profil(2), semop(2), shmop(2), sigpending(2), sigprocmask(2), times(2), umask(2), lockf(3C), ptrace(3C), setlocale(3C), signal(3C),

system(3C), timer_create(3RT), a.out(4), contract(4), attributes(5), environ(5), privileges(5), standards(5)

WARNINGS

If a program is setuid to a user ID other than the superuser, and the program is executed when the real user ID is super-user, then the program has some of the powers of a super-user as well.

No manual entry for 3.

NAME

exit, return, goto - shell built-in functions to enable the execution of the shell to advance beyond its sequence of steps

SYNOPSIS

sh

```
exit [n]

return [n]
```

csh

```
exit [ ( expr )]

goto label
```

ksh

```
*exit [n]

*return [n]
```

DESCRIPTION

sh

exit will cause the calling shell or shell script to exit with the exit status specified by n. If n is omitted the exit status is that of the last command executed (an EOF will also cause the shell to exit.)

return causes a function to exit with the return value specified by n. If n is omitted, the return status is that of the last command executed.

csh

exit will cause the calling shell or shell script to exit, either with the value of the status variable or with the value specified by the expression expr.

The goto built-in uses a specified label as a search string amongst commands. The shell rewinds its input as much as possible and searches for a line of the form label: possibly preceded by space or tab characters. Execution continues after the indicated line. It is an error to jump to a label that occurs between a while or for built-in command and its corresponding end.

ksh

exit will cause the calling shell or shell script to exit with the exit status specified by n. The value will be the least significant 8 bits of the specified status. If n is omitted then the exit status is that of the last command executed. When exit occurs when executing a trap, the last command refers to the command that executed before the

trap was invoked. An end-of-file will also cause the shell to exit except for a shell which has the ignoreeof option (See set below) turned on.

return causes a shell function or '.' script to return to the invoking script with the return status specified by n. The value will be the least significant 8 bits of the specified status. If n is omitted then the return status is that of the last command executed. If return is invoked while not in a function or a '.' script, then it is the same as an exit.

On this man page, ksh(1) commands that are preceded by one or two * (asterisks) are treated specially in the following ways:

1. Variable assignment lists preceding the command remain in effect when the command completes.

2. I/O redirections are processed after variable assignments.

3. Errors cause a script that contains them to abort.

4. Words, following a command preceded by ** that are in the format of a variable assignment, are expanded with the same rules as a variable assignment. This means that tilde substitution is performed after the = sign and word splitting and file name generation are not performed.

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Availability	SUNWcsu

SEE ALSO

break(1), csh(1), ksh(1), sh(1), attributes(5)

No manual entry for 3.

NAME

fork, fork1, forkall - create a new process

SYNOPSIS

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

pid_t fork(void);

pid_t fork1(void);

pid_t forkall(void);
```

DESCRIPTION

The fork(), fork1(), and forkall() functions create a new process. The address space of the new process (child process) is an exact copy of the address space of the calling process (parent process). The child process inherits the following attributes from the parent process:

- real user ID, real group ID, effective user ID, effective group ID
- environment
- open file descriptors
- close-on-exec flags (see exec(2))
- signal handling settings (that is, SIG_DFL, SIG_IGN, SIG_HOLD, function address)
- supplementary group IDs
- set-user-ID mode bit
- set-group-ID mode bit
- profiling on/off status
- nice value (see nice(2))
- scheduler class (see priocntl(2))
- all attached shared memory segments (see shmop(2))
- process group ID -- memory mappings (see mmap(2))
- session ID (see exit(2))
- current working directory

- o root directory
- o file mode creation mask (see umask(2))
- o resource limits (see getrlimit(2))
- o controlling terminal
- o saved user ID and group ID
- o task ID and project ID
- o processor bindings (see processor_bind(2))
- o processor set bindings (see pset_bind(2))
- o process privilege sets (see getppriv(2))
- o process flags (see getpflags(2))
- o active contract templates (see contract(4))

Scheduling priority and any per-process scheduling parameters that are specific to a given scheduling class might or might not be inherited according to the policy of that particular class (see priocntl(2)). The child process might or might not be in the same process contract as the parent (see process(4)). The child process differs from the parent process in the following ways:

- o The child process has a unique process ID which does not match any active process group ID.
- o The child process has a different parent process ID (that is, the process ID of the parent process).
- o The child process has its own copy of the parent's file descriptors and directory streams. Each of the child's file descriptors shares a common file pointer with the corresponding file descriptor of the parent.
- o Each shared memory segment remains attached and the value of shm_nattach is incremented by 1.
- o All semadj values are cleared (see semop(2)).
- o Process locks, text locks, data locks, and other memory locks are not inherited by the child (see plock(3C) and memcntl(2)).
- o

The child process's tms structure is cleared: tms_utime, stime, cutime, and cstime are set to 0 (see times(2)).

- o The child processes resource utilizations are set to 0; see getrlimit(2). The it_value and it_interval values for the ITIMER_REAL timer are reset to 0; see getitimer(2) .

- o The set of signals pending for the child process is initialized to the empty set.

- o Timers created by timer_create(3RT) are not inherited by the child process.

- o No asynchronous input or asynchronous output operations are inherited by the child.

- o Any preferred hardware address tranlsation sizes (see memcntl(2)) are inherited by the child.

- o The child process holds no contracts (see contract(4)).

Record locks set by the parent process are not inherited by the child process (see fcntl(2)).

Although any open door descriptors in the parent are shared by the child, only the parent will receive a door invocation from clients even if the door descriptor is open in the child. If a descriptor is closed in the parent, attempts to operate on the door descriptor will fail even if it is still open in the child.

Threads

A call to forkall() replicates in the child process all of the threads (see thr_create(3C) and pthread_create(3C)) in the parent process. A call to fork1() replicates only the calling thread in the child process.

In Solaris 10, a call to fork() is identical to a call to fork1(); only the calling thread is replicated in the child process. This is the POSIX-specified behavior for fork().

In previous releases of Solaris, the behavior of fork() depended on whether or not the application was linked with the POSIX threads library. When linked with -lthread (Solaris Threads) but not linked with -lpthread (POSIX Threads), fork() was the same as forkall(). When linked with -lpthread, whether or not also linked with -lthread, fork() was the same as fork1().

In Solaris 10, neither -lthread nor -lpthread is required for multithreaded applications. The standard C library provides all threading support for both sets of application programming interfaces. Applications that require replicate-all fork semantics must call forkall().

fork() Safety

If a multithreaded application calls fork() or fork1(), and the child does more than simply call one of the exec(2) functions, there is a possibility of deadlock occurring in the child. The application should use pthread_atfork(3C) to ensure safety with respect to this deadlock. Should there be any outstanding mutexes throughout the process, the application should call pthread_atfork() to wait for and acquire those mutexes prior to calling fork() or fork1(). See "MT-Level of Libraries" on the attributes(5) manual page.

RETURN VALUES

Upon successful completion, fork(), fork1(), and forkall() return 0 to the child process and return the process ID of the child process to the parent process. Otherwise, (pid_t)-1 is returned to the parent process, no child process is created, and errno is set to indicate the error.

ERRORS

The fork(), fork1(), and forkall()function will fail if:

EAGAIN — A resource control or limit on the total number of processes, tasks or LWPs under execution by a single user, task, project, or zone has been exceeded, or the total amount of system memory available is temporarily insufficient to duplicate this process.

ENOMEM — There is not enough swap space.

EPERM — The {PRIV_PROC_FORK} privilege is not asserted in the effective set of the calling process.

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	fork() is Standard. fork1() and forkall() are Stable.
MT-Level	Async-Signal-Safe.

SEE ALSO

alarm(2), exec(2), exit(2), fcntl(2), getitimer(2), getrlimit(2), memcntl(2), mmap(2), nice(2), priocntl(2), semop(2), shmop(2), times(2), umask(2), door_create(3DOOR), exit(3C), plock(3C), pthread_atfork(3C), pthread_create(3C), signal(3C), system(3C), thr_create(3C) timer_create(3RT), wait(3C), contract(4), process(4)attributes(5), privileges(5), standards(5)

NOTES

An applications should call _exit() rather than exit(3C) if it cannot execve(), since exit() will flush and close standard I/O channels and thereby corrupt the parent process's

standard I/O data structures. Using exit(3C) will flush buffered data twice. See exit(2).

The thread in the child that calls fork() or fork1() must not depend on any resources held by threads that no longer exist in the child. In particular, locks held by these threads will not be released.

In a multithreaded process, forkall() in one thread can cause blocking system calls to be interrupted and return with an EINTR error.

No manual entry for 3.

NAME

getcwd - get pathname of current working directory

SYNOPSIS

```
#include <unistd.h>

char *getcwd(char *buf, size_t size);
```

DESCRIPTION

The getcwd() function places an absolute pathname of the current working directory in the array pointed to by buf, and returns buf. The pathname copied to the array contains no components that are symbolic links. The size argument is the size in bytes of the character array pointed to by buf and must be at least one greater than the length of the pathname to be returned.

If buf is not a null pointer, the pathname is stored in the space pointed to by buf.

If buf is a null pointer, getcwd() obtains size bytes of space using malloc(3C). The pointer returned by getcwd() can be used as the argument in a subsequent call to free().

RETURN VALUES

Upon successful completion, getcwd() returns the buf argument. If buf is an invalid destination buffer address, NULL is returned and errno is set to EFAULT. Otherwise, a null pointer is returned and errno is set to indicate the error.

ERRORS

The getcwd() function will fail if:

EFAULT The buf argument is an invalid destination buffer address.

EINVAL The size argument is equal to 0.

ERANGE The size argument is greater than 0 and less than the length of the pathname plus 1.

The getcwd() function may fail if:

EACCES A parent directory cannot be read to get its name.

ENOMEM Insufficient storage space is available.

EXAMPLES

Example 1 Determine the absolute pathname of the current working directory.

The following example returns a pointer to an array that holds the absolute pathname of the current working directory. The pointer is returned in the ptr variable, which points to the buf array where the pathname is stored.

```
#include <stdlib.h>
#include <unistd.h>
...
long size;
char *buf;
char *ptr;
size = pathconf(".", _PC_PATH_MAX);
if ((buf = (char *)malloc((size_t)size)) != NULL)
      ptr = getcwd(buf, (size_t)size);
...
```

Example 2 Print the current working directory.

The following example prints the current working directory.

```
#include <unistd.h>
#include <stdio.h>

main()
{
   char *cwd;
   if ((cwd = getcwd(NULL, 64)) == NULL) {
       perror("pwd");
       exit(2);
   }
   (void)printf("%s\n", cwd);
   free(cwd); /* free memory allocated by getcwd() */
   return(0);
}
```

USAGE

Applications should exercise care when using chdir(2) in conjunction with getcwd(). The current working directory is global to all threads within a process. If more than one thread calls chdir() to change the working directory, a subsequent call to getcwd() could produce unexpected results.

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard

MT-Level	MT-Safe

SEE ALSO

chdir(2), malloc(3C), attributes(5), standards(5)

No manual entry for 3.

NAME

getdents - read directory entries and put in a file system independent format

SYNOPSIS

```
#include <dirent.h>

int getdents(int fildes, struct dirent *buf, size_t nbyte);
```

DESCRIPTION

The getdents() function attempts to read nbyte bytes from the directory associated with the file descriptor fildes and to format them as file system independent directory entries in the buffer pointed to by buf. Since the file system independent directory entries are of variable lengths, in most cases the actual number of bytes returned will be less than nbyte. The file system independent directory entry is specified by the dirent structure. See dirent.h(3HEAD).

On devices capable of seeking, getdents() starts at a position in the file given by the file pointer associated with fildes. Upon return from getdents(), the file pointer is incremented to point to the next directory entry.

RETURN VALUES

Upon successful completion, a non-negative integer is returned indicating the number of bytes actually read. A return value of 0 indicates the end of the directory has been reached. Otherwise, -1 is returned and errno is set to indicate the error.

ERRORS

The getdents() function will fail if:

EBADF — The fildes argument is not a valid file descriptor open for reading.

EFAULT — The buf argument points to an illegal address.

EINVAL — The nbyte argument is not large enough for one directory entry.

EIO — An I/O error occurred while accessing the file system.

ENOENT — The current file pointer for the directory is not located at a valid entry.

ENOLINK The fildes argument points to a remote machine and the link to that machine is no longer active.

ENOTDIR The fildes argument is not a directory.

EOVERFLOW The value of the dirent structure member d_ino or d_off cannot be represented in an ino_t or off_t.

USAGE

The getdents() function was developed to implement the readdir(3C) function and should not be used for other purposes.

The getdents() function has a transitional interface for 64-bit file offsets. See lf64(5).

SEE ALSO

readdir(3C), dirent.h(3HEAD), lf64(5)

No manual entry for 3.

NAME

malloc, calloc, free, memalign, realloc, valloc, alloca - memory allocator

SYNOPSIS

```
#include <stdlib.h>

void *malloc(size_t size);

void *calloc(size_t nelem, size_t elsize);

void free(void *ptr);

void *memalign(size_t alignment, size_t size);

void *realloc(void *ptr, size_t size);

void *valloc(size_t size);

#include <alloca.h>

void *alloca(size_t size);
```

DESCRIPTION

The malloc() and free() functions provide a simple, general-purpose memory allocation package. The malloc() function returns a pointer to a block of at least size bytes suitably aligned for any use. If the space assigned by malloc() is overrun, the results are undefined.

The argument to free() is a pointer to a block previously allocated by malloc(), calloc(), or realloc(). After free() is executed, this space is made available for further allocation by the application, though not returned to the system. Memory is returned to the system only upon termination of the application. If ptr is a null pointer, no action occurs. If a random number is passed to free(), the results are undefined.

The calloc() function allocates space for an array of nelem elements of size elsize. The space is initialized to zeros.

The memalign() function allocates size bytes on a specified alignment boundary and returns a pointer to the allocated block. The value of the returned address is guaranteed to be an even multiple of alignment. The value of alignment must be a power of two and must be greater than or equal to the size of a word.

The realloc() function changes the size of the block pointed to by ptr to size bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the

lesser of the new and old sizes. If the new size of the block requires movement of the block, the space for the previous instantiation of the block is freed. If the new size is larger, the contents of the newly allocated portion of the block are unspecified. If ptr is NULL, realloc() behaves like malloc() for the specified size. If size is 0 and ptr is not a null pointer, the space pointed to is freed.

The valloc() function has the same effect as malloc(), except that the allocated memory will be aligned to a multiple of the value returned by sysconf(_SC_PAGESIZE).

The alloca() function allocates size bytes of space in the stack frame of the caller, and returns a pointer to the allocated block. This temporary space is automatically freed when the caller returns. If the allocated block is beyond the current stack limit, the resulting behavior is undefined.

RETURN VALUES

Upon successful completion, each of the allocation functions returns a pointer to space suitably aligned (after possible pointer coercion) for storage of any type of object.

If there is no available memory, malloc(), realloc(), memalign(), valloc(), and calloc() return a null pointer. When realloc() is called with size > 0 and returns NULL, the block pointed to by ptr is left intact. If size, nelem, or elsize is 0, either a null pointer or a unique pointer that can be passed to free() is returned.

If malloc(), calloc(), or realloc() returns unsuccessfully, errno will be set to indicate the error. The free() function does not set errno.

ERRORS

The malloc(), calloc(), and realloc() functions will fail if:

ENOMEM — The physical limits of the system are exceeded by size bytes of memory which cannot be allocated.

EAGAIN — There is not enough memory available to allocate size bytes of memory; but the application could try again later.

USAGE

Portable applications should avoid using valloc() but should instead use malloc() or mmap(2). On systems with a large page size, the number of successful valloc() operations might be 0.

These default memory allocation routines are safe for use in multithreaded applications but are not scalable. Concurrent accesses by multiple threads are single-threaded through the use of a single lock. Multithreaded applications that make heavy use of dynamic memory allocation should be linked with allocation libraries designed for concurrent access, such as libumem(3LIB) or libmtmalloc(3LIB). Applications that want to avoid using heap allocations (with brk(2)) can do so by using either libumem or libmapmalloc(3LIB). The allocation libraries libmalloc(3LIB) and libbsdmalloc(3LIB) are available for special needs.

Comparative features of the various allocation libraries can be found in the umem_alloc(3MALLOC) manual page.

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	See below.
MT-Level	Safe

The malloc(), calloc(), free(), realloc(), valloc() functions are Standard. The memalign() and alloca() functions are Stable.

SEE ALSO

brk(2), getrlimit(2), libbsdmalloc(3LIB), libmalloc(3LIB), libmapmalloc(3LIB), libmtmalloc(3LIB), libumem(3LIB), umem_alloc(3MALLOC), watchmalloc(3MALLOC), attributes(5)

WARNINGS

Undefined results will occur if the size requested for a block of memory exceeds the maximum size of a process's heap, which can be obtained with getrlimit(2)

The alloca() function is machine-, compiler-, and most of all, system-dependent. Its use is strongly discouraged.

No manual entry for 3.

NAME

mmap - map pages of memory

SYNOPSIS

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

void *mmap(void *addr, size_t len, int prot, int flags,  int
fildes, off_t off);
```

DESCRIPTION

The mmap() function establishes a mapping between a process's address space and a file or shared memory object. The format of the call is as follows:

```
pa = mmap(addr, len, prot, flags, fildes, off);
```

The mmap() function establishes a mapping between the address space of the process at an address pa for len bytes to the memory object represented by the file descriptor fildes at offset off for len bytes. The value of pa is a function of the addr argument and values of flags, further described below. A successful mmap() call returns pa as its result. The address range starting at pa and continuing for len bytes will be legitimate for the possible (not necessarily current) address space of the process. The range of bytes starting at off and continuing for len bytes will be legitimate for the possible (not necessarily current) offsets in the file or shared memory object represented by fildes.

The mmap() function allows [pa, pa + len) to extend beyond the end of the object both at the time of the mmap() and while the mapping persists, such as when the file is created prior to the mmap() call and has no contents, or when the file is truncated. Any reference to addresses beyond the end of the object, however, will result in the delivery of a SIGBUS or SIGSEGV signal. The mmap() function cannot be used to implicitly extend the length of files.

The mapping established by mmap() replaces any previous mappings for those whole pages containing any part of the address space of the process starting at pa and continuing for len bytes.

If the size of the mapped file changes after the call to mmap() as a result of some other operation on the mapped file, the effect of references to portions of the mapped region that correspond to added or removed portions of the file is unspecified.

The mmap() function is supported for regular files and shared memory objects. Support for any other type of file is

unspecified.

The prot argument determines whether read, write, execute, or some combination of accesses are permitted to the data being mapped. The prot argument should be either PROT_NONE or the bitwise inclusive OR of one or more of the other flags in the following table, defined in the header <sys/mman.h>.

PROT_READ Data can be read.

PROT_WRITE Data can be written.

PROT_EXEC Data can be executed.

PROT_NONE Data cannot be accessed.

If an implementation of mmap() for a specific platform cannot support the combination of access types specified by prot, the call to mmap() fails. An implementation may permit accesses other than those specified by prot; however, the implementation will not permit a write to succeed where PROT_WRITE has not been set or permit any access where PROT_NONE alone has been set. Each platform-specific implementation of mmap() supports the following values of prot: PROT_NONE, PROT_READ, PROT_WRITE, and the inclusive OR of PROT_READ and PROT_WRITE. On some platforms, the PROT_WRITE protection option is implemented as PROT_READ|PROT_WRITE and PROT_EXEC as PROT_READ|PROT_EXEC. The file descriptor fildes is opened with read permission, regardless of the protection options specified. If PROT_WRITE is specified, the application must have opened the file descriptor fildes with write permission unless MAP_PRIVATE is specified in the flags argument as described below.

The flags argument provides other information about the handling of the mapped data. The value of flags is the bitwise inclusive OR of these options, defined in <sys/mman.h>:

MAP_SHARED Changes are shared.

MAP_PRIVATE Changes are private.

MAP_FIXED Interpret addr exactly.

MAP_NORESERVE Do not reserve swap space.

MAP_ANON Map anonymous memory.

MAP_ALIGN Interpret addr as required aligment.

MAP_TEXT Map text.

MAP_INITDATA Map initialized data segment.

The MAP_SHARED and MAP_PRIVATE options describe the disposition of write references to the underlying object. If MAP_SHARED is specified, write references will change the memory object. If MAP_PRIVATE is specified, the initial write reference will create a private copy of the memory

object page and redirect the mapping to the copy. The private copy is not created until the first write; until then, other users who have the object mapped MAP_SHARED can change the object. Either MAP_SHARED or MAP_PRIVATE must be specified, but not both. The mapping type is retained across fork(2).

When MAP_FIXED is set in the flags argument, the system is informed that the value of pa must be addr, exactly. If MAP_FIXED is set, mmap() may return (void *)-1 and set errno to EINVAL. If a MAP_FIXED request is successful, the mapping established by mmap() replaces any previous mappings for the process's pages in the range [pa, pa + len). The use of MAP_FIXED is discouraged, since it may prevent a system from making the most effective use of its resources.

When MAP_FIXED is set and the requested address is the same as previous mapping, the previous address is unmapped and the new mapping is created on top of the old one.

When MAP_FIXED is not set, the system uses addr to arrive at pa. The pa so chosen will be an area of the address space that the system deems suitable for a mapping of len bytes to the file. The mmap() function interprets an addr value of 0 as granting the system complete freedom in selecting pa, subject to constraints described below. A non-zero value of addr is taken to be a suggestion of a process address near which the mapping should be placed. When the system selects a value for pa, it will never place a mapping at address 0, nor will it replace any extant mapping, nor map into areas considered part of the potential data or stack "segments".

When MAP_ALIGN is set, the system is informed that the alignment of pa must be the same as addr. The alignment value in addr must be 0 or some power of two multiple of page size as returned by sysconf(3C). If addr is 0, the system will choose a suitable alignment.

The MAP_NORESERVE option specifies that no swap space be reserved for a mapping. Without this flag, the creation of a writable MAP_PRIVATE mapping reserves swap space equal to the size of the mapping; when the mapping is written into, the reserved space is employed to hold private copies of the data. A write into a MAP_NORESERVE mapping produces results which depend on the current availability of swap space in the system. If space is available, the write succeeds and a private copy of the written page is created; if space is not available, the write fails and a SIGBUS or SIGSEGV signal is delivered to the writing process. MAP_NORESERVE mappings are inherited across fork(); at the time of the fork(), swap space is reserved in the child for all private pages that currently exist in the parent; thereafter the child's mapping behaves as described above.

When MAP_ANON is set in flags, and fildes is set to -1, mmap() provides a direct path to return anonymous pages to the caller. This operation is equivalent to passing mmap() an open file descriptor on /dev/zero with MAP_ANON elided from the flags argument.

The MAP_TEXT option informs the system that the mapped region will be used primarily for executing instructions. This information can help the system better utilize MMU resources on some platforms. This flag is always passed by the dynamic linker when it maps text segments of shared objects. When the MAP_TEXT option is used for regular file mappings on some platforms, the system can choose a mapping size larger than the page size returned by sysconf(3C). The specific page sizes that are used depend on the platform and the alignment of the addr and len arguments. Several diffrent mapping sizes can be used to map the region with larger page sizes used in the parts of the region that meet

alignment and size requirements for those page sizes.

The MAP_INITDATA option informs the system that the mapped region is an initialized data segment of an executable or shared object. When the MAP_INITDATA
option is used for regular file mappings on some platforms, the system can choose a mapping size larger than the page size returned by sysconf(). The MAP_INITDATA option should be used only by the dynamic linker for mapping initialized data of shared objects.

The off argument is constrained to be aligned and sized according to the value returned by sysconf() when passed _SC_PAGESIZE or _SC_PAGE_SIZE. When MAP_FIXED is specified, the addr argument must also meet these constraints. The system performs mapping operations over whole pages. Thus, while the len argument need not meet a size or alignment constraint, the system will include, in any mapping operation, any partial page specified by the range [pa, pa + len).

The system will always zero-fill any partial page at the end of an object. Further, the system will never write out any modified portions of the last page of an object which are beyond its end. References to whole pages following the end of an object will result in the delivery of a SIGBUS or SIGSEGV signal. SIGBUS signals may also be delivered on various file system conditions, including quota exceeded errors.

The mmap() function adds an extra reference to the file associated with the file descriptor fildes which is not removed by a subsequent close(2) on that file descriptor. This reference is removed when there are no more mappings to the file by a call to the munmap(2) function.

The st_atime field of the mapped file may be marked for update at any time between the mmap() call and the corresponding munmap(2) call. The initial read or write reference to a mapped region will cause the file's st_atime field to be marked for update if it has not already been marked for update.

The st_ctime and st_mtime fields of a file that is mapped with MAP_SHARED and PROT_WRITE, will be marked for update at some point in the interval between a write reference to the mapped region and the next call to msync(3C) with MS_ASYNC

or MS_SYNC for that portion of the file by any process. If there is no such call, these fields may be marked for update at any time after a write reference if the underlying file is modified as a result.

If the process calls mlockall(3C) with the MCL_FUTURE flag, the pages mapped by all future calls to mmap() will be locked in memory. In this case, if not enough memory could be locked, mmap() fails and sets errno to EAGAIN.

The mmap() function aligns based on the length of the mapping. When determining the amount of space to add to the address space, mmap() includes two 8-Kbyte pages, one at each end of the mapping that are not mapped and are therefore used as "red-zone" pages. Attempts to reference these pages result in access violations.

The size requested is incremented by the 16 Kbytes for these pages and is then subject to rounding constraints. The constraints are:

- For 32-bit processes:

```
If length > 4 Mbytes
        round to 4-Mbyte multiple
elseif length > 512 Kbytes
        round to 512-Kbyte multiple
else
        round to 64-Kbyte multiple
```

- For 64-bit processes:

```
If length > 4 Mbytes
        round to 4-Mbyte multiple
else
        round to 1-Mbyte multiple
```

The net result is that for a 32-bit process:

- If an mmap() request is made for 4 Mbytes, it results in 4 Mbytes + 16 Kbytes and is rounded up to 8 Mbytes.

- If an mmap() request is made for 512 Kbytes, it results in 512 Kbytes + 16 Kbytes and is rounded up to 1 Mbyte.

- If an mmap() request is made for 1 Mbyte, it results in 1 Mbyte + 16 Kbytes and is rounded up to 1.5 Mbytes.

- Each 8-Kbyte mmap request "consumes" 64 Kbytes of virtual address space.

To obtain maximal address space usage for a 32-bit process:

- Combine 8-Kbyte requests up to a limit of 48 Kbytes.

- Combine amounts over 48 Kbytes into 496-Kbyte chunks.

- o Combine amounts over 496 Kbytes into 4080-Kbyte chunks.

To obtain maximal address space usage for a 64-bit process:

- o Combine amounts < 1008 Kbytes into chunks <= 1008 Kbytes.
- o Combine amounts over 1008 Kbytes into 4080-Kbyte chunks.

The following is the output from a 32-bit program demonstrating this:

```
map 8192 bytes: 0xff390000
map 8192 bytes: 0xff380000

    64-Kbyte delta between starting addresses.

map 512 Kbytes: 0xff180000
map 512 Kbytes: 0xff080000

    1-Mbyte delta between starting addresses.

map 496 Kbytes: 0xff000000
map 496 Kbytes: 0xfef80000

    512-Kbyte delta between starting addresses

map 1 Mbyte: 0xfee00000
map 1 Mbyte: 0xfec80000

    1536-Kbyte delta between starting addresses
```



```
map 1008 Kbytes: 0xfeb80000
map 1008 Kbytes: 0xfea80000

    1-Mbyte delta between starting addresses

map 4 Mbytes: 0xfe400000
map 4 Mbytes: 0xfdc00000

    8-Mbyte delta between starting addresses

map 4080 Kbytes: 0xfd800000
map 4080 Kbytes: 0xfd400000

    4-Mbyte delta between starting addresses
```

The following is the output of the same program compiled as a 64-bit application:

```
map 8192 bytes: 0xffffffff7f000000
map 8192 bytes: 0xffffffff7ef00000

    1-Mbyte delta between starting addresses

map 512 Kbytes: 0xffffffff7ee00000
map 512 Kbytes: 0xffffffff7ed00000
```

```
        1-Mbyte delta between starting addresses

     map 496 Kbytes: 0xffffffff7ec00000
     map 496 Kbytes: 0xffffffff7eb00000

        1-Mbyte delta between starting addresses
```



```
     map 1 Mbyte: 0xffffffff7e900000
     map 1 Mbyte: 0xffffffff7e700000

        2-Mbyte delta between starting addresses

     map 1008 Kbytes: 0xffffffff7e600000
     map 1008 Kbytes: 0xffffffff7e500000

        1-Mbyte delta between starting addresses

     map 4 Mbytes: 0xffffffff7e000000
     map 4 Mbytes: 0xffffffff7d800000

        8-Mbyte delta between starting addresses

     map 4080 Kbytes: 0xffffffff7d400000
     map 4080 Kbytes: 0xffffffff7d000000

        4-Mbyte delta between starting addresses
```

RETURN VALUES

Upon successful completion, the mmap() function returns the address at which the mapping was placed (pa); otherwise, it returns a value of MAP_FAILED and sets errno to indicate the error. The symbol MAP_FAILED is defined in the header <sys/mman.h>. No successful return from mmap() will return the value MAP_FAILED.

If mmap() fails for reasons other than EBADF, EINVAL or ENOTSUP, some of the mappings in the address range starting at addr and continuing for len bytes may have been unmapped.

ERRORS

The mmap() function will fail if:

EACCES The fildes file descriptor is not open for read, regardless of the protection specified; or fildes is not open for write and PROT_WRITE was specified for a MAP_SHARED type mapping.

EAGAIN The mapping could not be locked in memory.

There was insufficient room to reserve swap space for the mapping.

EBADF The fildes file descriptor is not open (and MAP_ANON was not specified).

EINVAL The arguments addr (if MAP_FIXED was specified) or off are not multiples of the page size as returned by sysconf().

The argument addr (if MAP_ALIGN was specified) is not 0 or some power of two multiple of page size as returned by sysconf(3C).

MAP_FIXED and MAP_ALIGN are both specified.

The field in flags is invalid (neither MAP_PRIVATE or MAP_SHARED is set).

The argument len has a value equal to 0.

MAP_ANON was specified, but the file descriptor was not -1.

MAP_TEXT was specified but PROT_EXEC was not.

MAP_TEXT and MAP_INITDATA were both specified.

EMFILE The number of mapped regions would exceed an implementation-dependent limit (per process or per system).

ENODEV The fildes argument refers to an object for which mmap() is meaningless, such as a terminal.

ENOMEM The MAP_FIXED option was specified and the range [addr, addr + len) exceeds that

allowed for the address space of a process.

The MAP_FIXED option was not specified and there is insufficient room in the address space to effect the mapping.

The mapping could not be locked in memory, if required by mlockall(3C), because it would require more space than the system is able to supply.

The composite size of len plus the lengths obtained from all previous calls to mmap() exceeds RLIMIT_VMEM (see getrlimit(2)).

ENOTSUP The system does not support the combination of accesses requested in the prot argument.

ENXIO Addresses in the range [off, off + len) are invalid for the object specified by fildes.

The MAP_FIXED option was specified in flags and the combination of addr, len and off is invalid for the object specified by fildes.

EOVERFLOW The file is a regular file and the value of off plus len exceeds the offset maximum establish in the open file description associated with fildes.

The mmap() function may fail if:

EAGAIN The file to be mapped is already locked using advisory or mandatory record locking. See fcntl(2).

USAGE

Use of mmap() may reduce the amount of memory available to other memory allocation functions.

MAP_ALIGN is useful to assure a properly aligned value of pa for subsequent use with memcntl(2) and the MC_HAT_ADVISE

command. This is best used for large, long-lived, and heavily referenced regions. MAP_FIXED and MAP_ALIGN are always mutually-exclusive.

Use of MAP_FIXED may result in unspecified behavior in further use of brk(2), sbrk(2), malloc(3C), and shmat(2). The use of MAP_FIXED is discouraged, as it may prevent an implementation from making the most effective use of resources.

The application must ensure correct synchronization when using mmap() in conjunction with any other file access method, such as read(2) and write(2), standard input/output, and shmat(2).

The mmap() function has a transitional interface for 64-bit file offsets. See lf64(5).

The mmap() function allows access to resources using address space manipulations instead of the read()/write() interface. Once a file is mapped, all a process has to do to access it is use the data at the address to which the object was mapped.

Consider the following pseudo-code:

```
fildes = open(...)
lseek(fildes, offset, whence)
read(fildes, buf, len)
/* use data in buf */
```

The following is a rewrite using mmap():

```
fildes = open(...)
address = mmap((caddr_t) 0, len, (PROT_READ | PROT_WRITE),
```

```
                MAP_PRIVATE, fildes, offset)
     /* use data at address */
```

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Standard
MT-Level	Async-Signal-Safe

SEE ALSO

close(2), exec(2), fcntl(2), fork(2), getrlimit(2), memcntl(2), mprotect(2), munmap(2), shmat(2), lockf(3C), mlockall(3C), msync(3C), plock(3C), sysconf(3C), attributes(5), lf64(5), standards(5), null(7D), zero(7D)

No manual entry for 3.

NAME

pipe - create an interprocess channel

SYNOPSIS

```
#include <unistd.h>

int pipe(int fildes[2]);
```

DESCRIPTION

The pipe() function creates an I/O mechanism called a pipe and returns two file descriptors, fildes[0] and fildes[1]. The files associated with fildes[0] and fildes[1] are streams and are both opened for reading and writing. The O_NDELAY, O_NONBLOCK, and FD_CLOEXEC flags are cleared on both file descriptors. The fcntl(2) function can be used to set these flags.

A read from fildes[0] accesses the data written to fildes[1] on a first-in-first-out (FIFO) basis and a read from fildes[1] accesses the data written to fildes[0] also on a FIFO basis.

Upon successful completion pipe() marks for update the st_atime, st_ctime, and st_mtime fields of the pipe.

RETURN VALUES

Upon successful completion, 0 is returned. Otherwise, -1 is returned and errno is set to indicate the error.

ERRORS

The pipe() function will fail if:

EMFILE More than {OPEN_MAX} file descriptors are already in use by this process.

ENFILE The number of simultaneously open files in the system would exceed a system-imposed limit.

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE

Interface Stability	Standard
MT-Level	Async-Signal-Safe

SEE ALSO

sh(1), fcntl(2), fstat(2), getmsg(2), poll(2), putmsg(2), read(2), write(2), attributes(5), standards(5), streamio(7I)

NOTES

Since a pipe is bi-directional, there are two separate flows of data. Therefore, the size (st_size) returned by a call to fstat(2) with argument fildes[0] or fildes[1] is the number of bytes available for reading from fildes[0] or fildes[1] respectively. Previously, the size (st_size) returned by a call to fstat() with argument fildes[1] (the write-end) was the number of bytes available for reading from fildes[0] (the read-end).

No manual entry for 3.

NAME

waitpid - wait for child process to change state

SYNOPSIS

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

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

DESCRIPTION

The waitpid() function will suspend execution of the calling thread until status information for one of its terminated child processes is available, or until delivery of a signal whose action is either to execute a signal-catching function or to terminate the process. If more than one thread is suspended in waitpid(), wait(3C), or waitid(2) awaiting termination of the same process, exactly one thread will return the process status at the time of the target process termination. If status information is available prior to the call to waitpid(), return will be immediate.

The pid argument specifies a set of child processes for which status is requested, as follows:

- If pid is less than (pid_t)-1, status is requested for any child process whose process group ID is equal to the absolute value of pid.
- If pid is equal to (pid_t)-1, status is requested for any child process.
- If pid is equal to (pid_t)0 status is requested for any child process whose process group ID is equal to that of the calling process.
- If pid is greater than (pid_t)0, it specifies the process ID of the child process for which status is requested.

One instance of a SIGCHLD signal is queued for each child process whose status has changed. If waitpid() returns because the status of a child process is available, and WNOWAIT was not specified in options, any pending SIGCHLD signal associated with the process ID of that child process is discarded. Any other pending SIGCHLD signals remain pending.

If the calling process has SA_NOCLDWAIT set or has SIGCHLD set to SIG_IGN and the process has no unwaited children that were transformed into zombie processes, it will block until all of its children terminate, and waitpid() will fail and set errno to ECHILD.

If waitpid() returns because the status of a child process is available, then that status may be evaluated with the macros defined by wait.h(3HEAD) If the calling process had specified a non-zero value of stat_loc, the status of the child process will be stored in the location pointed to by stat_loc.

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

WCONTINUED The status of any continued child process specified by pid, whose status has not been reported since it continued, is also reported to the calling process.

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

WNOWAIT Keep the process whose status is returned in stat_loc in a waitable state. The process may be waited for again with identical results.

WUNTRACED The status of any child processes specified by pid that are stopped, and whose status has not yet been reported since they stopped, is also reported to the calling process. WSTOPPED is a synonym for WUNTRACED.

RETURN VALUES

If waitpid() returns because the status of a child process is available, it returns a value equal to the process ID of the child process for which status is reported. If waitpid() returns due to the delivery of a signal to the calling process, -1 is returned and errno is set to EINTR. If waitpid() was invoked with WNOHANG set in options, it has at least one child process specified by pid for which status is not available, and status is not available for any process

specified by pid, then 0 is returned. Otherwise, -1 is returned and errno is set to indicate the error.

ERRORS

The waitpid() function will fail if:

ECHILD The process or process group specified by pid does not exist or is not a child of the calling process or can never be in the states specified by options.

EINTR The waitpid() function was interrupted due to the receipt of a signal sent by the calling process.

EINVAL An invalid value was specified for options.

USAGE

With options equal to 0 and pid equal to (pid_t)-1, waitpid() is identical to wait(3C). The waitpid() function is implemented as a call to the more general waitid(2) function.

ATTRIBUTES

See attributes(5) for descriptions of the following attributes:

ATTRIBUTE TYPE	ATTRIBUTE VALUE
Interface Stability	Committed
MT-Level	Async-Signal-Safe
Standard	See standards(5).

SEE ALSO

intro(2), exec(2), exit(2), fork(2), pause(2), sigaction(2), ptrace(3C), signal(3C), siginfo.h(3HEAD), wait(3C), wait.h(3HEAD), attributes(5), standards(5)

No manual entry for 3.