NAME

environ, execl, execle, execlp, execv, execve, execvp, fexecve — execute a file

SYNOPSIS

```
#include <unistd.h>

extern char **environ;
int execl(const char *path, const char *arg0, ... /*, (char *)0 */);
int execle(const char *path, const char *arg0, ... /*,
    (char *)0, char *const envp[]*/);
int execlp(const char *file, const char *arg0, ... /*, (char *)0 */);
int execv(const char *path, char *const argv[]);
int execve(const char *path, char *const argv[], char *const envp[]);
int execvp(const char *file, char *const argv[]);
int fexecve(int fd, char *const argv[], char *const envp[]);
```

DESCRIPTION

The *exec* family of functions shall replace the current process image with a new process image. The new image shall be constructed from a regular, executable file called the *new process image file*. There shall be no return from a successful *exec*, because the calling process image is overlaid by the new process image.

The *fexecve*() function shall be equivalent to the *execve*() function except that the file to be executed is determined by the file descriptor *fd* instead of a pathname. The file offset of *fd* is ignored.

When a C-language program is executed as a result of a call to one of the *exec* family of functions, it shall be 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*.

Conforming multi-threaded applications shall not use the *environ* variable to access or modify any environment variable while any other thread is concurrently modifying any environment variable. A call to any function dependent on any environment variable shall be considered a use of the *environ* variable to access that environment variable.

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

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

The argument *file* is used to construct a pathname that identifies the new process image file. If the *file* argument contains a <slash> character, the *file* argument shall be used as the pathname for this file. Otherwise, the path prefix for this file is obtained by a search of the directories passed as the environment variable *PATH* (see XBD Chapter 8, on page 173). If this environment variable is not present, the results of the search are implementation-defined.

There are two distinct ways in which the contents of the process image file may cause the execution to fail, distinguished by the setting of *errno* to either [ENOEXEC] or [EINVAL] (see the ERRORS section). In the cases where the other members of the *exec* family of functions would

fail and set *errno* to [ENOEXEC], the *execlp*() and *execvp*() functions shall execute a command interpreter and the environment of the executed command shall be as if the process invoked the *sh* utility using *execl*() as follows:

```
execl(<shell path>, arg0, file, arg1, ..., (char *)0);
```

where <*shell path*> is an unspecified pathname for the *sh* utility, *file* is the process image file, and for *execvp*(), where *arg*0, *arg*1, and so on correspond to the values passed to *execvp*() in *argv*[0], *argv*[1], and so on.

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

The argument *argv* is an array of character pointers to null-terminated strings. The application shall ensure that the last member of this array is a null pointer. These strings shall 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 argument *envp* is an array of character pointers to null-terminated strings. These strings shall constitute the environment for the new process image. The *envp* array is terminated by a null pointer.

For those forms not containing an *envp* pointer (*execl*(), *execv*(), *execlp*(), and *execvp*()), the environment for the new process image shall be taken from the external variable *environ* in the calling process.

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

File descriptors open in the calling process image shall remain open in the new process image, except for those whose close-on-*exec* flag FD_CLOEXEC is set. For those file descriptors that remain open, all attributes of the open file description remain unchanged. For any file descriptor that is closed for this reason, file locks are removed as a result of the close as described in *close*(). Locks that are not removed by closing of file descriptors remain unchanged.

If file descriptors 0, 1, and 2 would otherwise be closed after a successful call to one of the *exec* family of functions, and the new process image file has the set-user-ID or set-group-ID file mode bits set, and the ST_NOSUID bit is not set for the file system containing the new process image file, implementations may open an unspecified file for each of these file descriptors in the new process image.

XSI

Directory streams open in the calling process image shall be closed in the new process image.

The state of the floating-point environment in the initial thread of the new process image shall be set to the default.

The state of conversion descriptors and message catalog descriptors in the new process image is undefined.

For the new process image, the equivalent of:

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

shall be executed at start-up.

Signals set to the default action (SIG_DFL) in the calling process image shall be set to the default action in the new process image. Except for SIGCHLD, signals set to be ignored (SIG_IGN) by

the calling process image shall be set to be ignored by the new process image. Signals set to be caught by the calling process image shall be set to the default action in the new process image (see **<signal.h>**).

If the SIGCHLD signal is set to be ignored by the calling process image, it is unspecified whether the SIGCHLD signal is set to be ignored or to the default action in the new process image.

XSI After a successful call to any of the *exec* functions, alternate signal stacks are not preserved and the SA_ONSTACK flag shall be cleared for all signals.

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

XSI If the ST_NOSUID bit is set for the file system containing the new process image file, then the effective user ID, effective group ID, saved set-user-ID, and saved set-group-ID are unchanged in the new process image. Otherwise, if the set-user-ID mode bit of the new process image file is set, the effective user ID of the new process image shall be set to the user 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 shall be set to the group ID of the new process image file. The real user ID, real group ID, and supplementary group IDs of the new process image shall remain the same as those of the calling process image. The effective user ID and effective group ID of the new process image shall be saved (as the saved set-user-ID and the saved set-group-ID) for use by *setuid*( ).

XSI Any shared memory segments attached to the calling process image shall not be attached to the new process image.

Any named semaphores open in the calling process shall be closed as if by appropriate calls to *sem_close*( ).

TYM Any blocks of typed memory that were mapped in the calling process are unmapped, as if *munmap*( ) was implicitly called to unmap them.

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

Memory mappings created in the process are unmapped before the address space is rebuilt for the new process image.

SS When the calling process image does not use the SCHED_FIFO, SCHED_RR, or SCHED_SPORADIC scheduling policies, the scheduling policy and parameters of the new process image and the initial thread in that new process image are implementation-defined.

PS When the calling process image uses the SCHED_FIFO, SCHED_RR, or SCHED_SPORADIC scheduling policies, the process policy and scheduling parameter settings shall not be changed

TPS by a call to an *exec* function. The initial thread in the new process image shall inherit the process scheduling policy and parameters. It shall have the default system contention scope, but shall inherit its allocation domain from the calling process image.

Per-process timers created by the calling process shall be deleted before replacing the current process image with the new process image.

MSG All open message queue descriptors in the calling process shall be closed, as described in *mq_close*( ).

Any outstanding asynchronous I/O operations may be canceled. Those asynchronous I/O

operations that are not canceled shall complete as if the *exec* function had not yet occurred, but any associated signal notifications shall be suppressed. It is unspecified whether the *exec* function itself blocks awaiting such I/O completion. In no event, however, shall 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. Whether any I/O is canceled, and which I/O may be canceled upon *exec*, is implementation-defined.

CPT The new process image shall inherit the CPU-time clock of the calling process image. This inheritance means that the process CPU-time clock of the process being *exec*-ed shall not be reinitialized or altered as a result of the *exec* function other than to reflect the time spent by the process executing the *exec* function itself.

TCT The initial value of the CPU-time clock of the initial thread of the new process image shall be set to zero.

OB TRC If the calling process is being traced, the new process image shall continue to be traced into the same trace stream as the original process image, but the new process image shall not inherit the mapping of trace event names to trace event type identifiers that was defined by calls to the *posix_trace_eventid_open*() or the *posix_trace_trid_eventid_open*() functions in the calling process image.

If the calling process is a trace controller process, any trace streams that were created by the calling process shall be shut down as described in the *posix_trace_shutdown*() function.

The thread ID of the initial thread in the new process image is unspecified.

The size and location of the stack on which the initial thread in the new process image runs is unspecified.

The initial thread in the new process image shall have its cancellation type set to PTHREAD_CANCEL_DEFERRED and its cancellation state set to PTHREAD_CANCEL_ENABLED.

The initial thread in the new process image shall have all thread-specific data values set to NULL and all thread-specific data keys shall be removed by the call to *exec* without running destructors.

The initial thread in the new process image shall be joinable, as if created with the *detachstate* attribute set to PTHREAD_CREATE_JOINABLE.

The new process shall inherit at least the following attributes from the calling process image:

- XSI Nice value (see *nice*())
- XSI *semadj* values (see *semop*())
- Process ID
- Parent process ID
- Process group ID
- Session membership
- Real user ID
- Real group ID
- Supplementary group IDs

- Time left until an alarm clock signal (see *alarm*( ))
- Current working directory
- Root directory
- File mode creation mask (see *umask*( ))

XSI
- File size limit (see *getrlimit*( ) and *setrlimit*( ))

- Process signal mask (see *pthread_sigmask*( ))
- Pending signal (see *sigpending*( ))
- *tms_utime*, *tms_stime*, *tms_cutime*, and *tms_cstime* (see *times*( ))

XSI
- Resource limits

- Controlling terminal

XSI
- Interval timers

The initial thread of the new process shall inherit at least the following attributes from the calling thread:

- Signal mask (see *sigprocmask*( ) and *pthread_sigmask*( ))
- Pending signals (see *sigpending*( ))

All other process attributes defined in this volume of POSIX.1-2008 shall be inherited in the new process image from the old process image. All other thread attributes defined in this volume of POSIX.1-2008 shall be inherited in the initial thread in the new process image from the calling thread in the old process image. The inheritance of process or thread attributes not defined by this volume of POSIX.1-2008 is implementation-defined.

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

Upon successful completion, the *exec* functions shall mark for update the last data access timestamp of the file. If an *exec* function failed but was able to locate the process image file, whether the last data access timestamp is marked for update is unspecified. Should the *exec* function succeed, the process image file shall be considered to have been opened with *open*( ). The corresponding *close*( ) shall be 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, *posix_spawn*( ), or *posix_spawnp*( ). The *argv*[ ] and *envp*[ ] arrays of pointers and the strings to which those arrays point shall not be modified by a call to one of the *exec* functions, except as a consequence of replacing the process image.

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

RETURN VALUE

If one of the *exec* functions returns to the calling process image, an error has occurred; the return value shall be −1, and *errno* shall be set to indicate the error.

ERRORS

The *exec* functions shall fail if:

[E2BIG] The number of bytes used by the new process image's argument list and environment list is greater than the system-imposed limit of {ARG_MAX} bytes.

25899 [EACCES] Search permission is denied for a directory listed in the new process image
25900 file's path prefix, or the new process image file denies execution permission,
25901 or the new process image file is not a regular file and the implementation does
25902 not support execution of files of its type.

25903 [EINVAL] The new process image file has appropriate privileges and has a recognized
25904 executable binary format, but the system does not support execution of a file
25905 with this format.

25906 [ELOOP] A loop exists in symbolic links encountered during resolution of the *path* or *file*
25907 argument.

25908 [ENAMETOOLONG]
25909 The length of a component of a pathname is longer than {NAME_MAX}.

25910 [ENOENT] A component of *path* or *file* does not name an existing file or *path* or *file* is an
25911 empty string.

25912 [ENOTDIR] A component of the new process image file's path prefix is not a directory, or
25913 the new process image file's pathname contains at least one non-<slash>
25914 character and ends with one or more trailing <slash> characters and the last
25915 pathname component names an existing file that is neither a directory nor a
25916 symbolic link to a directory.

25917 The *exec* functions, except for *execlp*( ) and *execvp*( ), shall fail if:

25918 [ENOEXEC] The new process image file has the appropriate access permission but has an
25919 unrecognized format.

25920 The *fexecve*( ) function shall fail if:

25921 [EBADF] The *fd* argument is not a valid file descriptor open for executing.

25922 The *exec* functions may fail if:

25923 [ELOOP] More than {SYMLOOP_MAX} symbolic links were encountered during
25924 resolution of the *path* or *file* argument.

25925 [ENAMETOOLONG]
25926 The length of the *path* argument or the length of the pathname constructed
25927 from the *file* argument exceeds {PATH_MAX}, or pathname resolution of a
25928 symbolic link produced an intermediate result with a length that exceeds
25929 {PATH_MAX}.

25930 [ENOMEM] The new process image requires more memory than is allowed by the
25931 hardware or system-imposed memory management constraints.

25932 [ETXTBSY] The new process image file is a pure procedure (shared text) file that is
25933 currently open for writing by some process.

EXAMPLES

Using execl()

The following example executes the *ls* command, specifying the pathname of the executable (**/bin/ls**) and using arguments supplied directly to the command to produce single-column output.

```
#include <unistd.h>

int ret;
...
ret = execl ("/bin/ls", "ls", "-1", (char *)0);
```

Using execle()

The following example is similar to Using execl( ). In addition, it specifies the environment for the new process image using the *env* argument.

```
#include <unistd.h>

int ret;
char *env[] = { "HOME=/usr/home", "LOGNAME=home", (char *)0 };
...
ret = execle ("/bin/ls", "ls", "-l", (char *)0, env);
```

Using execlp()

The following example searches for the location of the *ls* command among the directories specified by the *PATH* environment variable.

```
#include <unistd.h>

int ret;
...
ret = execlp ("ls", "ls", "-l", (char *)0);
```

Using execv()

The following example passes arguments to the *ls* command in the *cmd* array.

```
#include <unistd.h>

int ret;
char *cmd[] = { "ls", "-l", (char *)0 };
...
ret = execv ("/bin/ls", cmd);
```

25965 **Using execve()**

25966 The following example passes arguments to the *ls* command in the *cmd* array, and specifies the
25967 environment for the new process image using the *env* argument.

```
25968 #include <unistd.h>

25969 int ret;
25970 char *cmd[] = { "ls", "-l", (char *)0 };
25971 char *env[] = { "HOME=/usr/home", "LOGNAME=home", (char *)0 };
25972 ...
25973 ret = execve ("/bin/ls", cmd, env);
```

25974 **Using execvp()**

25975 The following example searches for the location of the *ls* command among the directories
25976 specified by the *PATH* environment variable, and passes arguments to the *ls* command in the
25977 *cmd* array.

```
25978 #include <unistd.h>

25979 int ret;
25980 char *cmd[] = { "ls", "-l", (char *)0 };
25981 ...
25982 ret = execvp ("ls", cmd);
```

25983 **APPLICATION USAGE**

25984 As the state of conversion descriptors and message catalog descriptors in the new process image
25985 is undefined, conforming applications should not rely on their use and should close them prior
25986 to calling one of the *exec* functions.

25987 Applications that require other than the default POSIX locale should call *setlocale*() with the
25988 appropriate parameters to establish the locale of the new process.

25989 The *environ* array should not be accessed directly by the application.

25990 The new process might be invoked in a non-conforming environment if the *envp* array does not
25991 contain implementation-defined variables required by the implementation to provide a
25992 conforming environment. See the _CS_V7_ENV entry in **<unistd.h>** and *confstr*() for details.

25993 Applications should not depend on file descriptors 0, 1, and 2 being closed after an *exec*. A
25994 future version may allow these file descriptors to be automatically opened for any process.

25995 If an application wants to perform a checksum test of the file being executed before executing it,
25996 the file will need to be opened with read permission to perform the checksum test.

25997 Since execute permission is checked by *fexecve*(), the file description *fd* need not have been
25998 opened with the O_EXEC flag. However, if the file to be executed denies read and write
25999 permission for the process preparing to do the *exec*, the only way to provide the *fd* to *fexecve*()
26000 will be to use the O_EXEC flag when opening *fd*. In this case, the application will not be able to
26001 perform a checksum test since it will not be able to read the contents of the file.

26002 Note that when a file descriptor is opened with O_RDONLY, O_RDWR, or O_WRONLY mode,
26003 the file descriptor can be used to read, read and write, or write the file, respectively, even if the
26004 mode of the file changes after the file was opened. Using the O_EXEC open mode is different;
26005 *fexecve*() will ignore the mode that was used when the file descriptor was opened and the *exec*
26006 will fail if the mode of the file associated with *fd* does not grant execute permission to the calling
26007 process at the time *fexecve*() is called.

26008 **RATIONALE**

26009 Early proposals required that the value of *argc* passed to *main*() be "one or greater". This was
26010 driven by the same requirement in drafts of the ISO C standard. In fact, historical
26011 implementations have passed a value of zero when no arguments are supplied to the caller of
26012 the *exec* functions. This requirement was removed from the ISO C standard and subsequently
26013 removed from this volume of POSIX.1-2008 as well. The wording, in particular the use of the
26014 word *should*, requires a Strictly Conforming POSIX Application to pass at least one argument to
26015 the *exec* function, thus guaranteeing that *argc* be one or greater when invoked by such an
26016 application. In fact, this is good practice, since many existing applications reference *argv*[0]
26017 without first checking the value of *argc*.

26018 The requirement on a Strictly Conforming POSIX Application also states that the value passed as
26019 the first argument be a filename associated with the process being started. Although some
26020 existing applications pass a pathname rather than a filename in some circumstances, a filename
26021 is more generally useful, since the common usage of *argv*[0] is in printing diagnostics. In some
26022 cases the filename passed is not the actual filename of the file; for example, many
26023 implementations of the *login* utility use a convention of prefixing a <hyphen> (`'-'`) to the actual
26024 filename, which indicates to the command interpreter being invoked that it is a "login shell".

26025 Historically, there have been two ways that implementations can *exec* shell scripts.

26026 One common historical implementation is that the *execl*(), *execv*(), *execle*(), and *execve*()
26027 functions return an [ENOEXEC] error for any file not recognizable as executable, including a
26028 shell script. When the *execlp*() and *execvp*() functions encounter such a file, they assume the file
26029 to be a shell script and invoke a known command interpreter to interpret such files. This is now
26030 required by POSIX.1-2008. These implementations of *execvp*() and *execlp*() only give the
26031 [ENOEXEC] error in the rare case of a problem with the command interpreter's executable file.
26032 Because of these implementations, the [ENOEXEC] error is not mentioned for *execlp*() or
26033 *execvp*(), although implementations can still give it.

26034 Another way that some historical implementations handle shell scripts is by recognizing the first
26035 two bytes of the file as the character string `"#!"` and using the remainder of the first line of the
26036 file as the name of the command interpreter to execute.

26037 One potential source of confusion noted by the standard developers is over how the contents of
26038 a process image file affect the behavior of the *exec* family of functions. The following is a
26039 description of the actions taken:

26040 1. If the process image file is a valid executable (in a format that is executable and valid and
26041 having appropriate privileges) for this system, then the system executes the file.

26042 2. If the process image file has appropriate privileges and is in a format that is executable
26043 but not valid for this system (such as a recognized binary for another architecture), then
26044 this is an error and *errno* is set to [EINVAL] (see later RATIONALE on [EINVAL]).

26045 3. If the process image file has appropriate privileges but is not otherwise recognized:

26046 a. If this is a call to *execlp*() or *execvp*(), then they invoke a command interpreter
26047 assuming that the process image file is a shell script.

26048 b. If this is not a call to *execlp*() or *execvp*(), then an error occurs and *errno* is set to
26049 [ENOEXEC].

26050 Applications that do not require to access their arguments may use the form:

26051 `main(void)`

26052 as specified in the ISO C standard. However, the implementation will always provide the two

arguments *argc* and *argv*, even if they are not used.

Some implementations provide a third argument to *main*() called *envp*. This is defined as a pointer to the environment. The ISO C standard specifies invoking *main*() with two arguments, so implementations must support applications written this way. Since this volume of POSIX.1-2008 defines the global variable *environ*, which is also provided by historical implementations and can be used anywhere that *envp* could be used, there is no functional need for the *envp* argument. Applications should use the *getenv*() function rather than accessing the environment directly via either *envp* or *environ*. Implementations are required to support the two-argument calling sequence, but this does not prohibit an implementation from supporting *envp* as an optional third argument.

This volume of POSIX.1-2008 specifies that signals set to SIG_IGN remain set to SIG_IGN, and that the new process image inherits the signal mask of the thread that called *exec* in the old process image. This is consistent with historical implementations, and it permits some useful functionality, such as the *nohup* command. However, it should be noted that many existing applications wrongly assume that they start with certain signals set to the default action and/or unblocked. In particular, applications written with a simpler signal model that does not include blocking of signals, such as the one in the ISO C standard, may not behave properly if invoked with some signals blocked. Therefore, it is best not to block or ignore signals across *exec*s without explicit reason to do so, and especially not to block signals across *exec*s of arbitrary (not closely co-operating) programs.

The *exec* functions always save the value of the effective user ID and effective group ID of the process at the completion of the *exec*, whether or not the set-user-ID or the set-group-ID bit of the process image file is set.

The statement about *argv*[ ] and *envp*[ ] being constants is included to make explicit to future writers of language bindings that these objects are completely constant. Due to a limitation of the ISO C standard, it is not possible to state that idea in standard C. Specifying two levels of *const–qualification* for the *argv*[ ] and *envp*[ ] parameters for the *exec* functions may seem to be the natural choice, given that these functions do not modify either the array of pointers or the characters to which the function points, but this would disallow existing correct code. Instead, only the array of pointers is noted as constant. The table of assignment compatibility for *dst=src* derived from the ISO C standard summarizes the compatibility:

dst:	**char *[ ]**	**const char *[ ]**	**char *const[ ]**	**const char *const[ ]**
src:				
char *[ ]	VALID	—	VALID	—
const char *[ ]	—	VALID	—	VALID
char * const [ ]	—	—	VALID	—
const char *const[ ]	—	—	—	VALID

Since all existing code has a source type matching the first row, the column that gives the most valid combinations is the third column. The only other possibility is the fourth column, but using it would require a cast on the *argv* or *envp* arguments. It is unfortunate that the fourth column cannot be used, because the declaration a non-expert would naturally use would be that in the second row.

The ISO C standard and this volume of POSIX.1-2008 do not conflict on the use of *environ*, but some historical implementations of *environ* may cause a conflict. As long as *environ* is treated in the same way as an entry point (for example, *fork*()), it conforms to both standards. A library can contain *fork*(), but if there is a user-provided *fork*(), that *fork*() is given precedence and no problem ensues. The situation is similar for *environ*: the definition in this volume of POSIX.1-2008 is to be used if there is no user-provided *environ* to take precedence. At least three

implementations are known to exist that solve this problem.

[E2BIG] The limit {ARG_MAX} applies not just to the size of the argument list, but to the sum of that and the size of the environment list.

[EFAULT] Some historical systems return [EFAULT] rather than [ENOEXEC] when the new process image file is corrupted. They are non-conforming.

[EINVAL] This error condition was added to POSIX.1-2008 to allow an implementation to detect executable files generated for different architectures, and indicate this situation to the application. Historical implementations of shells, *execvp*( ), and *execlp*( ) that encounter an [ENOEXEC] error will execute a shell on the assumption that the file is a shell script. This will not produce the desired effect when the file is a valid executable for a different architecture. An implementation may now choose to avoid this problem by returning [EINVAL] when a valid executable for a different architecture is encountered. Some historical implementations return [EINVAL] to indicate that the *path* argument contains a character with the high order bit set. The standard developers chose to deviate from historical practice for the following reasons:

1. The new utilization of [EINVAL] will provide some measure of utility to the user community.
2. Historical use of [EINVAL] is not acceptable in an internationalized operating environment.

[ENAMETOOLONG] Since the file pathname may be constructed by taking elements in the *PATH* variable and putting them together with the filename, the [ENAMETOOLONG] error condition could also be reached this way.

[ETXTBSY] System V returns this error when the executable file is currently open for writing by some process. This volume of POSIX.1-2008 neither requires nor prohibits this behavior.

Other systems (such as System V) may return [EINTR] from *exec*. This is not addressed by this volume of POSIX.1-2008, but implementations may have a window between the call to *exec* and the time that a signal could cause one of the *exec* calls to return with [EINTR].

An explicit statement regarding the floating-point environment (as defined in the **<fenv.h>** header) was added to make it clear that the floating-point environment is set to its default when a call to one of the *exec* functions succeeds. The requirements for inheritance or setting to the default for other process and thread start-up functions is covered by more generic statements in their descriptions and can be summarized as follows:

posix_spawn( )	Set to default.
fork( )	Inherit.
pthread_create( )	Inherit.

The purpose of the *fexecve*( ) function is to enable executing a file which has been verified to be the intended file. It is possible to actively check the file by reading from the file descriptor and be sure that the file is not exchanged for another between the reading and the execution. Alternatively, an function like *openat*( ) can be used to open a file which has been found by reading the content of a directory using *readdir*( ).

FUTURE DIRECTIONS

None.

SEE ALSO

alarm( ), *atexit*( ), *chmod*( ), *close*( ), *confstr*( ), *exit*( ), *fcntl*( ), *fork*( ), *fstatvfs*( ), *getenv*( ), *getitimer*( ), *getrlimit*( ), *mknod*( ), *mmap*( ), *nice*( ), *open*( ), *posix_spawn*( ), *posix_trace_create*( ), *posix_trace_event*( ), *posix_trace_eventid_equal*( ), *pthread_atfork*( ), *pthread_sigmask*( ), *putenv*( ), *readdir*( ), *semop*( ), *setlocale*( ), *shmat*( ), *sigaction*( ), *sigaltstack*( ), *sigpending*( ), *system*( ), *times*( ), *ulimit*( ), *umask*( )

XBD Chapter 8 (on page 173), **<unistd.h>**

CHANGE HISTORY

First released in Issue 1. Derived from Issue 1 of the SVID.

Issue 5

The DESCRIPTION is updated for alignment with the POSIX Realtime Extension and the POSIX Threads Extension.

Large File Summit extensions are added.

Issue 6

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- In the DESCRIPTION, behavior is defined for when the process image file is not a valid executable.
- In this version, _POSIX_SAVED_IDS is mandated, thus the effective user ID and effective group ID of the new process image shall be saved (as the saved set-user-ID and the saved set-group-ID) for use by the *setuid*( ) function.
- The [ELOOP] mandatory error condition is added.
- A second [ENAMETOOLONG] is added as an optional error condition.
- The [ETXTBSY] optional error condition is added.

The following changes were made to align with the IEEE P1003.1a draft standard:

- The [EINVAL] mandatory error condition is added.
- The [ELOOP] optional error condition is added.

The description of CPU-time clock semantics is added for alignment with IEEE Std 1003.1d-1999.

The DESCRIPTION is updated for alignment with IEEE Std 1003.1j-2000 by adding semantics for typed memory.

The normative text is updated to avoid use of the term "must" for application requirements.

The description of tracing semantics is added for alignment with IEEE Std 1003.1q-2000.

IEEE PASC Interpretation 1003.1 #132 is applied.

The DESCRIPTION is updated to make it explicit that the floating-point environment in the new process image is set to the default.

The DESCRIPTION and RATIONALE are updated to include clarifications of how the contents of a process image file affect the behavior of the *exec* functions.

IEEE Std 1003.1-2001/Cor 1-2002, item XSH/TC1/D6/15 is applied, adding a new paragraph to the DESCRIPTION and text to the end of the APPLICATION USAGE section. This change

addresses a security concern, where implementations may want to reopen file descriptors 0, 1, and 2 for programs with the set-user-id or set-group-id file mode bits calling the *exec* family of functions.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/24 is applied, applying changes to the DESCRIPTION, addressing which attributes are inherited by threads, and behavioral requirements for threads attributes.

IEEE Std 1003.1-2001/Cor 2-2004, item XSH/TC2/D6/25 is applied, updating text in the RATIONALE from "the process signal mask be unchanged across an *exec*" to "the new process image inherits the signal mask of the thread that called *exec* in the old process image".

Issue 7

Austin Group Interpretation 1003.1-2001 #047 is applied, adding the description of _CS_V7_ENV to the APPLICATION USAGE.

Austin Group Interpretation 1003.1-2001 #143 is applied.

The *fexecve*( ) function is added from The Open Group Technical Standard, 2006, Extended API Set Part 2.

Functionality relating to the Asynchronous Input and Output, Memory Mapped Files, Threads, and Timers options is moved to the Base.

Changes are made related to support for finegrained timestamps.

