Full credit is given to the above companies including the OS that this PDF file was generated!

Red Hat Enterprise Linux Release 9.2 Manual Pages on 'exec.3p' command

$ man exec.3p

EXEC(3P) POSIX Programmer's Manual EXEC(3P)

PROLOG

This manual page is part of the POSIX Programmer's Manual. The Linux implementation of this interface may differ (consult the corresponding Linux manual page for details of Linux behavior), or the interface may not be implemented on Linux.

NAME

environ, execl, execle, execlp, execv, execve, execvp, fexecve ? exe? cute 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 ex? cept 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 func? tion 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 vari? able, which must be declared by the user if it is to be used directly:

```
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.

Applications can change the entire environment in a single operation by assigning the environ variable to point to an array of character point? ers to the new environment strings. After assigning a new value to env? iron, applications should not rely on the new environment strings re? maining part of the environment, as a call to getenv(), putenv(), setenv(), unsetenv(), or any function that is dependent on an environ? ment variable may, on noticing that environ has changed, copy the envi? ronment strings to a new array and assign environ to point to it.

Any application that directly modifies the pointers to which the envi? ron variable points has undefined behavior.

Conforming multi-threaded applications shall not use the environ vari? able 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> charac? ter, 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 the Base Defi? nitions volume of POSIX.1?2017, Chapter 8, Environment Variables). 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 arg0, arg1, 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 string 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 ar?

ray 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 string 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(), exe? clp(), 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 in? cluded 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 re? moved by closing of file descriptors remain unchanged.

If file descriptor 0, 1, or 2 would otherwise be closed after a suc? cessful call to one of the exec family of functions, implementations may open an unspecified file for the file descriptor in the new process image. If a standard utility or a conforming application is executed with file descriptor 0 not open for reading or with file descriptor 1 or 2 not open for writing, the environment in which the utility or ap? plication is executed shall be deemed non-conforming, and consequently the utility or application might not behave as described in this stan? dard.

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 im? age shall be set to the default action in the new process image. Ex? cept 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. Sig? nals 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 im? age, it is unspecified whether the SIGCHLD signal is set to be ignored or to the default action in the new process image.

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.

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 im? age shall be set to the user ID of the new process image file. Simi? larly, 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 se?
tuid().

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().

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

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.

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

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 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 con? tention scope, but shall inherit its allocation domain from the calling process image.

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

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 noti? fications shall be suppressed. It is unspecified whether the exec func? tion 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.

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.

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

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 de? scribed in the posix_trace_shutdown() function.

The thread ID of the initial thread in the new process image is unspec? ified.

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-spe? cific 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:

* Nice value (see nice())
* 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())
* 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())
* Resource limits
* Controlling terminal
* Interval timers

The initial thread of the new process shall inherit at least the fol? lowing 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?2017 shall be inherited in the new process image from the old process image.

All other thread attributes defined in this volume of POSIX.1?2017 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?2017 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 im? age being loaded and executed. No destructor functions or cleanup han? dlers 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 func? tions, 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 conse? quence of replacing the process image.

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.

EACCES The new process image file is not a regular file and the imple? mentation does not support execution of files of its type.

EINVAL The new process image file has appropriate privileges and has a

recognized executable binary format, but the system does not support execution of a file with this format.

The exec functions, except for fexecve(), shall fail if:

EACCES Search permission is denied for a directory listed in the new process image file's path prefix, or the new process image file denies execution permission.

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

ENAMETOOLONG

The length of a component of a pathname is longer than {NAME_MAX}.

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

ENOTDIR

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

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

ENOEXEC

The new process image file has the appropriate access permission but has an unrecognized format.

The fexecve() function shall fail if:

EBADF The fd argument is not a valid file descriptor open for execut? ing.

The exec functions may fail if:

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

The exec functions, except for fexecve(), may fail if:

ELOOP More than {SYMLOOP_MAX} symbolic links were encountered during

resolution of the path or file argument.

ENAMETOOLONG

The length of the path argument or the length of the pathname constructed from the file argument exceeds {PATH_MAX}, or path? name resolution of a symbolic link produced an intermediate re? sult with a length that exceeds {PATH_MAX}.

ETXTBSY

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

The following sections are informative.

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 argu? ment.

```
#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 ar? ray.

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

Using execve()

The following example passes arguments to the ls command in the cmd ar? ray, and specifies the environment for the new process image using the env argument.

```
#include <unistd.h>
int ret;
char *cmd[] = { "ls", "-l", (char *)0 };
char *env[] = { "HOME=/usr/home", "LOGNAME=home", (char *)0 };
...
ret = execve ("/bin/ls", cmd, env);
```

Using execvp()

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

```
#include <unistd.h>
int ret;
char *cmd[] = { "ls", "-l", (char *)0 };
...
ret = execvp ("ls", cmd);
```

APPLICATION USAGE

As the state of conversion descriptors and message catalog descriptors in the new process image is undefined, conforming 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 as the global locale in the new process image should call setlocale() with the appropriate parameters.

When assigning a new value to the environ variable, applications should ensure that the environment to which it will point contains at least the following:

1. Any implementation-defined variables required by the implementation to provide a conforming environment. See the _CS_V7_ENV entry in <unistd.h> and confstr() for details.
2. A value for PATH which finds conforming versions of all standard utilities before any other versions.

The same constraint applies to the envp array passed to execle() or ex? ecve(), in order to ensure that the new process image is invoked in a conforming environment.

Applications should not execute programs with file descriptor 0 not open for reading or with file descriptor 1 or 2 not open for writing, as this might cause the executed program to misbehave. In order not to pass on these file descriptors to an executed program, applications should not just close them but should reopen them on, for example, /dev/null. Some implementations may reopen them automatically, but ap? plications should not rely on this being done.

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

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

Note that when a file descriptor is opened with O_RDONLY, O_RDWR, or

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

RATIONALE

Early proposals required that the value of argc passed to main() be ``one or greater''. This was driven by the same requirement in drafts of the ISO C standard. In fact, historical implementations have passed a value of zero when no arguments are supplied to the caller of the exec functions. This requirement was removed from the ISO C standard and subsequently removed from this volume of POSIX.1?2017 as well. The wording, in particular the use of the word should, requires a Strictly Conforming POSIX Application to pass at least one argument to the exec function, thus guaranteeing that argc be one or greater when invoked by such an application. In fact, this is good practice, since many exist? ing applications reference argv[0] without first checking the value of argc.

The requirement on a Strictly Conforming POSIX Application also states that the value passed as the first argument be a filename string asso? ciated with the process being started. Although some existing applica? tions pass a pathname rather than a filename string in some circum? stances, a filename string is more generally useful, since the common usage of argv[0] is in printing diagnostics. In some cases the filename passed is not the actual filename of the file; for example, many imple? mentations of the login utility use a convention of prefixing a <hy? phen-minus> ('?') to the actual filename, which indicates to the com? mand interpreter being invoked that it is a ``login shell''.

Also, note that the test and [utilities require specific strings for the argv[0] argument to have deterministic behavior across all imple? mentations.

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

One common historical implementation is that the execl(), execv(), exe? cle(), and execve() functions return an [ENOEXEC] error for any file not recognizable as executable, including a shell script. When the exe? clp() and execvp() functions encounter such a file, they assume the file to be a shell script and invoke a known command interpreter to in? terpret such files. This is now required by POSIX.1?2008. These imple? mentations of execvp() and execlp() only give the [ENOEXEC] error in the rare case of a problem with the command interpreter's executable file. Because of these implementations, the [ENOEXEC] error is not men? tioned for execlp() or execvp(), although implementations can still give it.

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

One potential source of confusion noted by the standard developers is over how the contents of a process image file affect the behavior of the exec family of functions. The following is a description of the ac? tions taken:

1. If the process image file is a valid executable (in a format that is executable and valid and having appropriate privileges) for this system, then the system executes the file.
2. If the process image file has appropriate privileges and is in a format that is executable but not valid for this system (such as a recognized binary for another architecture), then this is an error and errno is set to [EINVAL] (see later RATIONALE on [EINVAL]).
3. If the process image file has appropriate privileges but is not otherwise recognized:
   a. If this is a call to execlp() or execvp(), then they invoke a command interpreter assuming that the process image file is a shell script.

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

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

main(void)

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?2017 defines the global variable environ, which is also pro? vided by historical implementations and can be used anywhere that envp could be used, there is no functional need for the envp argument. Ap? plications 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?2017 specifies that signals set to SIG_IGN re? main 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 particu? lar, applications written with a simpler signal model that does not in? clude 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 execs without explicit reason to do so, and especially not to block signals across execs of arbitrary (not closely cooperating) 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			
const char *[]		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 natu? rally use would be that in the second row.

The ISO C standard and this volume of POSIX.1?2017 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 li?
brary can contain fork(), but if there is a user-provided fork(), that
fork() is given precedence and no problem ensues. The situation is sim?
ilar for environ: the definition in this volume of POSIX.1?2017 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-con? forming.

EINVAL This error condition was added to POSIX.1?2008 to allow an im? plementation 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 pro? duce 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 argu? ment contains a character with the high order bit set. The stan? dard developers chose to deviate from historical practice for the following reasons:

1. The new utilization of [EINVAL] will provide some mea? sure of utility to the user community.
2. Historical use of [EINVAL] is not acceptable in an in? ternationalized 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?2017 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?2017, 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, a 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()

The Base Definitions volume of POSIX.1?2017, Chapter 8, Environment Variables, <unistd.h>

The Shell and Utilities volume of POSIX.1?2017, test

COPYRIGHT