execl(), execle(), execlp(), execlpe(), execv(), execve(), execvp(), execvpe()

Function

SYNOPSIS

execute a file

SYNOPSIS

```
#include <unistd.h>
extern char **environ;
int execl(const char *path, const char *arg0, ..., const char *argn, (char *)0);
int execle(const char *path, const char *arg0, ..., const char *argn, (char *)0, char *const envp[]);
int execlp(const char *file, const char *arg0, ..., const char *argn, (char *)0);
int execlpe(const char *file, const char *arg0, ..., const char *argn, (char *)0, char *const envp[]);
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 execvpe(const char *file, char *const argv[], char *const envp[]);
```

DESCRIPTION

The `exec()` family of functions creates a new process image from a regular, executable file. This file is either an executable object file, or an interpreter script. There is no return from a successful call to an `exec()` function, because the calling process is functionally replaced by the new process.

An interpreter script 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 script is executed, the system executes the specified interpreter. The path name specified in the interpreter file is passed as *argv*[0] to the interpreter. If *arg* was specified in the interpreter file, it is passed as *argv*[1] to the interpreter. The remaining arguments to the interpreter are the arguments passed to the `exec()` function. The contents of the file are passed to the interpreter as its standard input.

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

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

where *argc* is the argument count and *argv* is an array of pointers to the arguments themselves. In addition, the global variable `environ` is initialized to point to an array of 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 do 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 is considered a use of `environ`.

The arguments specified by a program with an `exec()` function are passed on to the new process image in the corresponding `main()` arguments. For forms of `exec()` that do not take an *envp* argument, the environment for the new process image is taken from the external variable `environ` in the calling process. Otherwise, the *envp* argument specifies the environment for the new process image.

The number of bytes available for combined argument and environment lists of the new process is specified by `ARG_MAX`. String terminators and 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. For those file descriptors that remain open, all attributes of the open file description, including file locks, remain unchanged.

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

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. 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 in the calling process image are set to the default action in the new process image.

No functions registered by atexit() in the calling process image are registered in the new process image.

No shared memory or memory mapped segments attached to the calling process image are attached to the new process image.

At a minimum, the new process also inherits the following attributes from the calling process image:

- Process ID (see *PORTING ISSUES*)
- Parent process ID
- Process group ID
- Session membership
- Real user ID
- Real group ID
- Time left until an alarm clock signal
- Current working directory
- File mode creation mask
- Process signal mask
- Pending signals
- Process execution times, as returned by times()
- Semaphore adjustment values
- Controlling terminal.
- Interval timers
- nice value

A call to an 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 are called.

PARAMETERS

path
: Specifies the path name of the new process image file.

file
: Is used to construct a path name that identifies the new process image file. If it contains a slash character, the argument is used as the path name for this file. Otherwise, the path prefix for this file is obtained by a search of the directories in the environment variable ***PATH***. If ***PATH*** is not set, the current directory is searched.

arg0, ..., *argn*
: Point to null-terminated character strings. These strings constitute the argument list for the new process image. The list is terminated by a NULL pointer. The argument *arg0* should point to a file name that is associated with the process being started by the exec() function.

argv
: Is the argument list for the new process image. This should contain an array of pointers to character strings, and the array should be terminated by a NULL pointer. The value in *argv*[0] should point to a file name that is associated with the process being started by the exec() function.

envp
: Specifies the environment for the new process image. This should contain an array of pointers to character strings, and the array should be terminated by a NULL pointer.

RETURN VALUES

If successful, the exec() functions do not return. On failure, they returns -1 and set errno to one of the following values:

E2BIG
: The number of bytes used by the new process image's argument list and environment list is greater than ARG_MAX bytes.

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, or the new process image file is not a regular file.

ENAMETOOLONG
: The length of *path* or *file* (or an element of $PATH prefixed to *file*) exceeds PATH_MAX, or a path name component 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.

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

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

ENOTDIR
: A component of the new process image file's path prefix is not a directory.

ETXTBSY
: The exec() function was called from a non-N*u*TCRACKER Platform process, other than from the child of a vfork() operation.

CONFORMANCE

execl(), execle(), execlp(), execv(), execve(), execvp(): UNIX 98, with exceptions.

execlpe(), execvpe(): PTC MKS Toolkit UNIX APIs extensions.

MULTITHREAD SAFETY LEVEL

execl(), execlp(), execlpe(), execv(), execvp(), execvpe(): MT-Safe.

execle(), execve(): Async-signal-safe.

Except for execle() and execve(), these functions are MT-Safe as long as the environment is not being modified. If the environment is being modified when these functions are called, behavior is unspecified. execle() and execve() are Async-signal-safe.

PORTING ISSUES

The N*u*TCRACKER Platform uses the Win32 CreateProcess() function to create the new process image, and does not overlay the existing image, as is done on most UNIX systems. This is visible only in that the process ID returned by getpid() in the new process image does not match that returned in the calling process image. Refer to *Process Management* in the *Windows Concepts* chapter of the *PTC MKS Toolkit UNIX to Windows Porting Guide*.

You may not call an exec() function from a non-N*u*TCRACKER Platform application (for example, from a standalone N*u*TCRACKER Platform DLL used in a native Win32 application), except from the child of a vfork() operation. Refer to *Building Standalone DLLs* in the *Porting Shared Libraries* chapter of the *PTC MKS Toolkit UNIX to Windows Porting Guide* for more information.

The Windows file systems do not support set-user-ID and set-group-ID bits for files. Hence there is no support for automatically setting effective user and/or group IDs at process execution time.

If the process being executed is not a N*u*TCRACKER Platform process, only the standard file descriptors (0, 1, 2 - stdin, stdout, stderr) are available to the new process. However, if that process then invokes a N*u*TCRACKER Platform process, all inherited file descriptors are available to the grandchild N*u*TCRACKER Platform process.

You must ensure that any path name arguments you pass to non-N*u*TCRACKER Platform applications are in Win32 format, as only N*u*TCRACKER Platform applications recognize the N*u*TCRACKER Platform format. Refer to *Path Names* in the

Windows Concepts chapter of the *PTC MKS Toolkit UNIX to Windows Porting Guide* for more information.

Priorities are inherited by new threads in the same way as on UNIX systems. Even the first thread created by a native Win32 process inherits priority in this manner. However, further creation of threads not under control of the N*u*TCRACKER Platform might revert to `THREAD_PRIORITY_NORMAL`.

AVAILABILITY

PTC MKS Toolkit for Professional Developers
PTC MKS Toolkit for Enterprise Developers
PTC MKS Toolkit for Enterprise Developers 64-Bit Edition

SEE ALSO

Functions:

_NutForkExecl(), **_NutForkExecle()**, **_NutForkExeclp()**, **_NutForkExeclpe()**, **_NutForkExecv()**, **_NutForkExecve()**, **_NutForkExecvp()**, **_NutForkExecvpe()**, **_NutQueryPid()**, **alarm()**, **atexit()**, **chmod()**, **exit()**, **fcntl()**, **fork()**, **getenv()**, **getitimer()**, **getpid()**, **mmap()**, **pthread_create()**, **putenv()**, **semop()**, **setlocale()**, **setpriority()**, **shmat()**, **sigaction()**, **sigpending()**, **sigprocmask()**, **times()**, **umask()**, **vfork()**