execve()

Execute a file

Synopsis:

```
#include <process.h>

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

Arguments:

path
: A path name that identifies the new process image file.

argv
: An array of character pointers to NULL-terminated strings. Your application must ensure that the last member of this array is a NULL pointer. These strings constitute the argument list available to the new process image. The value in *argv[0]* must point to a filename that's associated with the process being started.

envp
: An array of character pointers to NULL-terminated strings. These strings constitute the environment for the new process image. Terminate the *envp* array with a NULL pointer.

Library:

`libc`

Use the `-l c` option to qcc to link against this library. This library is usually included automatically.

Description:

The *execve()* function replaces the current process image with a new process image specified by *path*. The new image is constructed from a regular, executable file called the new process image file. No return is made because the calling process image is replaced by the new process image.

In order to start the new process, your process must have the PROCMGR_AID_SPAWN ability enabled. For more information, see *procmgr_ability()*.

When a C-language program is executed as a result of this call, it's 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 isn't counted in `argc`.

Multithreaded applications shouldn't 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 the `environ` variable to access that environment variable.

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

The number of bytes available for the new process's combined argument and environment lists is ARG_MAX.

File descriptors open in the calling process image remain open in the new process image, except for when *fcntl()*'s FD_CLOEXEC flag is set. For those file descriptors that remain open, all attributes of the open file description, including file locks remain unchanged. If a file descriptor is closed for this reason, file locks are removed as described by *close()* while locks not affected by *close()* aren't changed.

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

Signals set to SIG_DFL in the calling process are set to the default action in the new process image. Signals set to SIG_IGN by the calling process images are 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. After a successful call, alternate signal stacks aren't preserved and the SA_ONSTACK flag is cleared for all signals.

After a successful call, any functions previously registered by *atexit()* are no longer registered.

If the `path` is on a filesystem mounted with the ST_NOSUID flag set, the effective user ID, effective group ID, saved set-user ID and saved set-group ID are unchanged for the new process. Otherwise, if the set-user ID mode bit is set, the effective user ID of the new process image is set to the user ID of `path`. Similarly, if the set-group ID mode bit is set, the effective group ID of the new process is set to the group ID of `path`. The real user ID, real group ID, and supplementary group IDs of the new process remain the same as those of the calling process. 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 used by *setuid()*).

Any shared memory segments attached to the calling process image aren't attached to the new process image. If the calling process had locked any memory, the locks are released.

The new process doesn't inherit the calling process's default timer tolerance (set by a call to *procmgr_timer_tolerance()*).

The new process also inherits at least the following attributes from the calling process image:

- 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()*)

process signal mask (see *sigprocmask()*)

pending signal (see *sigpending()*)

`tms_utime`, `tms_stime`, `tms_cutime`, and `tms_cstime` (see *times()*)

resource limits

controlling terminal

interval timers.

A call to this 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.

Upon successful completion, the `st_atime` field of the file is marked for update. If the *exec** failed but was able to locate the process image file, whether the `st_atime` field is marked for update is unspecified. On success, the process image file is considered to be opened with *open()*. The corresponding *close()* 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.

***exec*()* summary**

Function	Description	POSIX?
execl()	NULL-terminated argument list	Yes
execle()	NULL-terminated argument list, specify the new process's environment	Yes
execlp()	NULL-terminated argument list, search for the new process in ***PATH***	Yes
execlpe()	NULL-terminated argument list, search for the new process in ***PATH***, specify the new process's environment	No
execv()	NULL-terminated array of arguments	Yes
execve()	NULL-terminated array of arguments, specify the new process's environment	Yes
execvp()	NULL-terminated array of arguments, search for the new process in ***PATH***	Yes
execvpe()	NULL-terminated array of arguments, search for the new process in ***PATH***, specify the new process's environment	No

Returns:

When *execve()* is successful, it doesn't return; otherwise, it returns -1 and sets `errno`.

Errors:

E2BIG
: The argument list and the environment is larger than the system limit of ARG_MAX bytes.

EACCES
: The calling process doesn't have permission to search a directory listed in `path`, or it doesn't have permission to execute `path`, or `path`'s filesystem was mounted with the ST_NOEXEC flag.

ELOOP
: Too many levels of symbolic links or prefixes.

ENAMETOOLONG
: The length of `path` or an element of the ***PATH*** environment variable exceeds PATH_MAX.

ENOENT
: One or more components of the pathname don't exist, or the `path` argument points to an empty string.

ENOEXEC

The new process's image file has the correct access permissions, but isn't in the proper format.

ENOMEM

There's insufficient memory available to create the new process.

ENOTDIR

A component of `path` isn't a directory.

EPERM

The calling process doesn't have the required permission (see *procmgr_ability()*), or an underlying call to *mmap()* failed because it attempted to set PROT_EXEC for a region of memory covered by an untrusted memory-mapped file.

ETXTBSY

The text file that you're trying to execute is busy (e.g. it might be open for writing).

Classification:

POSIX 1003.1

Safety:	
Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

Parent topic: E

Related concepts

Processes and Threads (Getting Started with QNX Neutrino)

Related reference

execl()
execle()
execlp()
execlpe()
execv()
execvp()
execvpe()
abort()
atexit()
errno
_exit()
exit()
getenv()
main()
posix_spawn(), posix_spawnp()
procmgr_ability()
putenv()
spawn()
spawnl()
spawnle()
spawnlp()
spawnlpe()
spawnp()
spawnv()
spawnve()
spawnvp()
spawnvpe()
system()