NAME
execve – execute program

LIBRARY
Standard C library (*libc*, *−lc*)

SYNOPSIS
#include <unistd.h>

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

DESCRIPTION
execve() executes the program referred to by *pathname*. This causes the program that is currently being run by the calling process to be replaced with a new program, with newly initialized stack, heap, and (initialized and uninitialized) data segments.

pathname must be either a binary executable, or a script starting with a line of the form:

      #!*interpreter* [optional-arg]

For details of the latter case, see "Interpreter scripts" below.

argv is an array of pointers to strings passed to the new program as its command-line arguments. By convention, the first of these strings (i.e., *argv[0]*) should contain the filename associated with the file being executed. The *argv* array must be terminated by a NULL pointer. (Thus, in the new program, *argv[argc]* will be NULL.)

envp is an array of pointers to strings, conventionally of the form **key=value**, which are passed as the environment of the new program. The *envp* array must be terminated by a NULL pointer.

This manual page describes the Linux system call in detail; for an overview of the nomenclature and the many, often preferable, standardised variants of this function provided by libc, including ones that search the **PATH** environment variable, see **exec**(3).

The argument vector and environment can be accessed by the new program's main function, when it is defined as:

      int main(int argc, char *argv[], char *envp[])

Note, however, that the use of a third argument to the main function is not specified in POSIX.1; according to POSIX.1, the environment should be accessed via the external variable **environ**(7).

execve() does not return on success, and the text, initialized data, uninitialized data (bss), and stack of the calling process are overwritten according to the contents of the newly loaded program.

If the current program is being ptraced, a **SIGTRAP** signal is sent to it after a successful **execve**().

If the set-user-ID bit is set on the program file referred to by *pathname*, then the effective user ID of the calling process is changed to that of the owner of the program file. Similarly, if the set-group-ID bit is set on the program file, then the effective group ID of the calling process is set to the group of the program file.

The aforementioned transformations of the effective IDs are *not* performed (i.e., the set-user-ID and set-group-ID bits are ignored) if any of the following is true:

• the *no_new_privs* attribute is set for the calling thread (see **prctl**(2));

• the underlying filesystem is mounted *nosuid* (the **MS_NOSUID** flag for **mount**(2)); or

• the calling process is being ptraced.

The capabilities of the program file (see **capabilities**(7)) are also ignored if any of the above are true.

The effective user ID of the process is copied to the saved set-user-ID; similarly, the effective group ID is copied to the saved set-group-ID. This copying takes place after any effective ID changes that occur because of the set-user-ID and set-group-ID mode bits.

The process's real UID and real GID, as well as its supplementary group IDs, are unchanged by a call to

execve().

If the executable is an a.out dynamically linked binary executable containing shared-library stubs, the Linux dynamic linker **ld.so**(8) is called at the start of execution to bring needed shared objects into memory and link the executable with them.

If the executable is a dynamically linked ELF executable, the interpreter named in the PT_INTERP segment is used to load the needed shared objects. This interpreter is typically */lib/ld−linux.so.2* for binaries linked with glibc (see **ld−linux.so**(8)).

Effect on process attributes

All process attributes are preserved during an **execve**(), except the following:

- The dispositions of any signals that are being caught are reset to the default (**signal**(7)).

- Any alternate signal stack is not preserved (**sigaltstack**(2)).

- Memory mappings are not preserved (**mmap**(2)).

- Attached System V shared memory segments are detached (**shmat**(2)).

- POSIX shared memory regions are unmapped (**shm_open**(3)).

- Open POSIX message queue descriptors are closed (**mq_overview**(7)).

- Any open POSIX named semaphores are closed (**sem_overview**(7)).

- POSIX timers are not preserved (**timer_create**(2)).

- Any open directory streams are closed (**opendir**(3)).

- Memory locks are not preserved (**mlock**(2), **mlockall**(2)).

- Exit handlers are not preserved (**atexit**(3), **on_exit**(3)).

- The floating-point environment is reset to the default (see **fenv**(3)).

The process attributes in the preceding list are all specified in POSIX.1. The following Linux-specific process attributes are also not preserved during an **execve**():

- The process's "dumpable" attribute is set to the value 1, unless a set-user-ID program, a set-group-ID program, or a program with capabilities is being executed, in which case the dumpable flag may instead be reset to the value in */proc/sys/fs/suid_dumpable*, in the circumstances described under **PR_SET_DUMPABLE** in **prctl**(2). Note that changes to the "dumpable" attribute may cause ownership of files in the process's */proc/*pid directory to change to *root:root*, as described in **proc**(5).

- The **prctl**(2) **PR_SET_KEEPCAPS** flag is cleared.

- (Since Linux 2.4.36 / 2.6.23) If a set-user-ID or set-group-ID program is being executed, then the parent death signal set by **prctl**(2) **PR_SET_PDEATHSIG** flag is cleared.

- The process name, as set by **prctl**(2) **PR_SET_NAME** (and displayed by *ps −o comm*), is reset to the name of the new executable file.

- The **SECBIT_KEEP_CAPS** *securebits* flag is cleared. See **capabilities**(7).

- The termination signal is reset to **SIGCHLD** (see **clone**(2)).

- The file descriptor table is unshared, undoing the effect of the **CLONE_FILES** flag of **clone**(2).

Note the following further points:

- All threads other than the calling thread are destroyed during an **execve**(). Mutexes, condition variables, and other pthreads objects are not preserved.

- The equivalent of *setlocale(LC_ALL, "C")* is executed at program start-up.

- POSIX.1 specifies that the dispositions of any signals that are ignored or set to the default are left unchanged. POSIX.1 specifies one exception: if **SIGCHLD** is being ignored, then an implementation may leave the disposition unchanged or reset it to the default; Linux does the former.

- Any outstanding asynchronous I/O operations are canceled (**aio_read**(3), **aio_write**(3)).

- For the handling of capabilities during **execve**(), see **capabilities**(7).

- By default, file descriptors remain open across an **execve**(). File descriptors that are marked close-on-exec are closed; see the description of **FD_CLOEXEC** in **fcntl**(2). (If a file descriptor is closed, this will cause the release of all record locks obtained on the underlying file by this process. See **fcntl**(2) for details.) POSIX.1 says that if file descriptors 0, 1, and 2 would otherwise be closed after a successful **execve**(), and the process would gain privilege because the set-user-ID or set-group-ID mode bit was set on the executed file, then the system may open an unspecified file for each of these file descriptors. As a general principle, no portable program, whether privileged or not, can assume that these three file descriptors will remain closed across an **execve**().

Interpreter scripts

An interpreter script is a text file that has execute permission enabled and whose first line is of the form:

```
#!interpreter [optional-arg]
```

The *interpreter* must be a valid pathname for an executable file.

If the *pathname* argument of **execve**() specifies an interpreter script, then *interpreter* will be invoked with the following arguments:

```
interpreter [optional-arg] pathname arg...
```

where *pathname* is the pathname of the file specified as the first argument of **execve**(), and *arg...* is the series of words pointed to by the *argv* argument of **execve**(), starting at *argv[1]*. Note that there is no way to get the *argv[0]* that was passed to the **execve**() call.

For portable use, *optional-arg* should either be absent, or be specified as a single word (i.e., it should not contain white space); see NOTES below.

Since Linux 2.6.28, the kernel permits the interpreter of a script to itself be a script. This permission is recursive, up to a limit of four recursions, so that the interpreter may be a script which is interpreted by a script, and so on.

Limits on size of arguments and environment

Most UNIX implementations impose some limit on the total size of the command-line argument (*argv*) and environment (*envp*) strings that may be passed to a new program. POSIX.1 allows an implementation to advertise this limit using the **ARG_MAX** constant (either defined in *<limits.h>* or available at run time using the call *sysconf(_SC_ARG_MAX)*).

Before Linux 2.6.23, the memory used to store the environment and argument strings was limited to 32 pages (defined by the kernel constant **MAX_ARG_PAGES**). On architectures with a 4-kB page size, this yields a maximum size of 128 kB.

On Linux 2.6.23 and later, most architectures support a size limit derived from the soft **RLIMIT_STACK** resource limit (see **getrlimit**(2)) that is in force at the time of the **execve**() call. (Architectures with no memory management unit are excepted: they maintain the limit that was in effect before Linux 2.6.23.) This change allows programs to have a much larger argument and/or environment list. For these architectures, the total size is limited to 1/4 of the allowed stack size. (Imposing the 1/4-limit ensures that the new program always has some stack space.) Additionally, the total size is limited to 3/4 of the value of the kernel constant **_STK_LIM** (8 MiB). Since Linux 2.6.25, the kernel also places a floor of 32 pages on this size limit, so that, even when **RLIMIT_STACK** is set very low, applications are guaranteed to have at least as much argument and environment space as was provided by Linux 2.6.22 and earlier. (This guarantee was not provided in Linux 2.6.23 and 2.6.24.) Additionally, the limit per string is 32 pages (the kernel constant **MAX_ARG_STRLEN**), and the maximum number of strings is 0x7FFFFFFF.

RETURN VALUE

On success, **execve**() does not return, on error −1 is returned, and *errno* is set to indicate the error.

ERRORS

E2BIG   The total number of bytes in the environment (*envp*) and argument list (*argv*) is too large.

EACCES

> Search permission is denied on a component of the path prefix of *pathname* or the name of a script interpreter. (See also **path_resolution**(7).)

EACCES

> The file or a script interpreter is not a regular file.

EACCES

> Execute permission is denied for the file or a script or ELF interpreter.

EACCES

> The filesystem is mounted *noexec*.

EAGAIN (since Linux 3.1)

> Having changed its real UID using one of the **set*uid**() calls, the caller was—and is now still— above its **RLIMIT_NPROC** resource limit (see **setrlimit**(2)). For a more detailed explanation of this error, see NOTES.

EFAULT

> *pathname* or one of the pointers in the vectors *argv* or *envp* points outside your accessible address space.

EINVAL

> An ELF executable had more than one PT_INTERP segment (i.e., tried to name more than one in- terpreter).

EIO   An I/O error occurred.

EISDIR

> An ELF interpreter was a directory.

ELIBBAD

> An ELF interpreter was not in a recognized format.

ELOOP

> Too many symbolic links were encountered in resolving *pathname* or the name of a script or ELF interpreter.

ELOOP

> The maximum recursion limit was reached during recursive script interpretation (see "Interpreter scripts", above). Before Linux 3.8, the error produced for this case was **ENOEXEC**.

EMFILE

> The per-process limit on the number of open file descriptors has been reached.

ENAMETOOLONG

> *pathname* is too long.

ENFILE

> The system-wide limit on the total number of open files has been reached.

ENOENT

> The file *pathname* or a script or ELF interpreter does not exist.

ENOEXEC

> An executable is not in a recognized format, is for the wrong architecture, or has some other for- mat error that means it cannot be executed.

ENOMEM

> Insufficient kernel memory was available.

ENOTDIR

A component of the path prefix of *pathname* or a script or ELF interpreter is not a directory.

EPERM

The filesystem is mounted *nosuid*, the user is not the superuser, and the file has the set-user-ID or set-group-ID bit set.

EPERM

The process is being traced, the user is not the superuser and the file has the set-user-ID or set-group-ID bit set.

EPERM

A "capability-dumb" applications would not obtain the full set of permitted capabilities granted by the executable file. See **capabilities**(7).

ETXTBSY

The specified executable was open for writing by one or more processes.

STANDARDS

POSIX.1-2001, POSIX.1-2008, SVr4, 4.3BSD. POSIX does not document the #! behavior, but it exists (with some variations) on other UNIX systems.

NOTES

One sometimes sees **execve**() (and the related functions described in **exec**(3)) described as "executing a *new* process" (or similar). This is a highly misleading description: there is no new process; many attributes of the calling process remain unchanged (in particular, its PID). All that **execve**() does is arrange for an existing process (the calling process) to execute a new program.

Set-user-ID and set-group-ID processes can not be **ptrace**(2)d.

The result of mounting a filesystem *nosuid* varies across Linux kernel versions: some will refuse execution of set-user-ID and set-group-ID executables when this would give the user powers they did not have already (and return **EPERM**), some will just ignore the set-user-ID and set-group-ID bits and **exec**() successfully.

On Linux, *argv* and *envp* can be specified as NULL. In both cases, this has the same effect as specifying the argument as a pointer to a list containing a single null pointer. **Do not take advantage of this nonstandard and nonportable misfeature!** On many other UNIX systems, specifying *argv* as NULL will result in an error (**EFAULT**). *Some* other UNIX systems treat the *envp==NULL* case the same as Linux.

POSIX.1 says that values returned by **sysconf**(3) should be invariant over the lifetime of a process. However, since Linux 2.6.23, if the **RLIMIT_STACK** resource limit changes, then the value reported by **_SC_ARG_MAX** will also change, to reflect the fact that the limit on space for holding command-line arguments and environment variables has changed.

In most cases where **execve**() fails, control returns to the original executable image, and the caller of **execve**() can then handle the error. However, in (rare) cases (typically caused by resource exhaustion), failure may occur past the point of no return: the original executable image has been torn down, but the new image could not be completely built. In such cases, the kernel kills the process with a **SIGSEGV** (**SIGKILL** until Linux 3.17) signal.

Interpreter scripts

The kernel imposes a maximum length on the text that follows the "#!" characters at the start of a script; characters beyond the limit are ignored. Before Linux 5.1, the limit is 127 characters. Since Linux 5.1, the limit is 255 characters.

The semantics of the *optional-arg* argument of an interpreter script vary across implementations. On Linux, the entire string following the *interpreter* name is passed as a single argument to the interpreter, and this string can include white space. However, behavior differs on some other systems. Some systems use the first white space to terminate *optional-arg*. On some systems, an interpreter script can have multiple arguments, and white spaces in *optional-arg* are used to delimit the arguments.

Linux (like most other modern UNIX systems) ignores the set-user-ID and set-group-ID bits on scripts.

execve() and EAGAIN
   A more detailed explanation of the **EAGAIN** error that can occur (since Linux 3.1) when calling **execve**() is as follows.

   The **EAGAIN** error can occur when a *preceding* call to **setuid**(2), **setreuid**(2), or **setresuid**(2) caused the real user ID of the process to change, and that change caused the process to exceed its **RLIMIT_NPROC** resource limit (i.e., the number of processes belonging to the new real UID exceeds the resource limit). From Linux 2.6.0 to Linux 3.0, this caused the **set*uid**() call to fail. (Before Linux 2.6, the resource limit was not imposed on processes that changed their user IDs.)

   Since Linux 3.1, the scenario just described no longer causes the **set*uid**() call to fail, because it too often led to security holes where buggy applications didn't check the return status and assumed that—if the caller had root privileges—the call would always succeed. Instead, the **set*uid**() calls now successfully change the real UID, but the kernel sets an internal flag, named **PF_NPROC_EXCEEDED**, to note that the **RLIMIT_NPROC** resource limit has been exceeded. If the **PF_NPROC_EXCEEDED** flag is set and the resource limit is still exceeded at the time of a subsequent **execve**() call, that call fails with the error **EAGAIN**. This kernel logic ensures that the **RLIMIT_NPROC** resource limit is still enforced for the common privileged daemon workflow—namely, **fork**(2) + **set*uid**() + **execve**().

   If the resource limit was not still exceeded at the time of the **execve**() call (because other processes belonging to this real UID terminated between the **set*uid**() call and the **execve**() call), then the **execve**() call succeeds and the kernel clears the **PF_NPROC_EXCEEDED** process flag. The flag is also cleared if a subsequent call to **fork**(2) by this process succeeds.

Historical
   With UNIX V6, the argument list of an **exec**() call was ended by 0, while the argument list of *main* was ended by −1. Thus, this argument list was not directly usable in a further **exec**() call. Since UNIX V7, both are NULL.

EXAMPLES
   The following program is designed to be execed by the second program below. It just echoes its command-line arguments, one per line.

```
/* myecho.c */

#include <stdio.h>
#include <stdlib.h>

int
main(int argc, char *argv[])
{
    for (size_t j = 0; j < argc; j++)
        printf("argv[%zu]: %s\n", j, argv[j]);

    exit(EXIT_SUCCESS);
}
```

   This program can be used to exec the program named in its command-line argument:

```
/* execve.c */

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

int
main(int argc, char *argv[])
{
    static char *newargv[] = { NULL, "hello", "world", NULL };
```

```
           static char *newenviron[] = { NULL };

           if (argc != 2) {
               fprintf(stderr, "Usage: %s <file-to-exec>\n", argv[0]);
               exit(EXIT_FAILURE);
           }

           newargv[0] = argv[1];

           execve(argv[1], newargv, newenviron);
           perror("execve");   /* execve() returns only on error */
           exit(EXIT_FAILURE);
       }
```

We can use the second program to exec the first as follows:

```
$ cc myecho.c -o myecho
$ cc execve.c -o execve
$ ./execve ./myecho
argv[0]: ./myecho
argv[1]: hello
argv[2]: world
```

We can also use these programs to demonstrate the use of a script interpreter. To do this we create a script whose "interpreter" is our *myecho* program:

```
$ cat > script
#!./myecho script-arg
^D
$ chmod +x script
```

We can then use our program to exec the script:

```
$ ./execve ./script
argv[0]: ./myecho
argv[1]: script-arg
argv[2]: ./script
argv[3]: hello
argv[4]: world
```

SEE ALSO

chmod(2), **execveat**(2), **fork**(2), **get_robust_list**(2), **ptrace**(2), **exec**(3), **fexecve**(3), **getauxval**(3), **getopt**(3), **system**(3), **capabilities**(7), **credentials**(7), **environ**(7), **path_resolution**(7), **ld.so**(8)