NAME

exec, execl, execv, execle, execve, execlp, execvp – execute a file

SYNOPSIS

#include <unistd.h>

int execl(const char **path*, **const char** **arg0*, **. . ., const char** **argn*, **char** * /*NULL*/**);**

int execv(const char **path*, **char** ***const** *argv*[ ]**);**

int execle (const char **path*,**char** ***const** *arg0*[ ], **. . . , const char** **argn*,
  char * /*NULL*/, **char** ***const** *envp*[ ]**);**

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

int execlp (const char **file*, **const char** **arg0*, **. . ., const char** **argn*, **char** * /*NULL*/**);**

int execvp (const char **file*, **char** ***const** *argv*[ ]**);**

MT-LEVEL

execle( ) and **execve( )** are Async-Signal-Safe

DESCRIPTION

exec( ) in all its forms overlays a new process image on an old process. The new process image is constructed from an ordinary, executable file. This file is either an executable object file, or a file of data for an interpreter. There can be no return from a successful **exec( )** because the calling process image is overlaid by the new process image.

An interpreter file 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 file is exec'd, the system execs the specified interpreter. The pathname specified in the interpreter file is passed as *arg0* to the interpreter. If *arg* was specified in the interpreter file, it is passed as *arg1* to the interpreter. The remaining arguments to the interpreter are *arg0* through *argn* of the originally exec'd file.

When a C program is executed, it is called as follows:

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

where *argc* is the argument count, *argv* is an array of character pointers to the arguments themselves, and *envp* is an array of character pointers to the environment strings. As indicated, *argc* is at least one, and the first member of the array points to a string containing the name of the file.

path points to a path name that identifies the new process file.

file points to the new process file. If *file* does not contain a slash character, the path prefix for this file is obtained by a search of the directories passed in the **PATH** environment variable (see **environ**(5)). The environment is supplied typically by the shell. If the new process file is not an executable object file, **execlp( )** and **execvp( )** use the contents of that file as standard input to the shell.

Solaris **exec( )** uses **/usr/bin/sh** (see **sh**(1)).

XPG4 **exec( )** uses the XPG4-compliant shell **/usr/bin/ksh** (see **ksh**(1)).

The arguments *arg0*, ..., *argn* point to null-terminated character strings. These strings constitute the argument list available to the new process image. Conventionally at least *arg0* should be present. It will become the name of the process, as displayed by the **ps** command. *arg0* points to a string that is the same as *path* (or the last component of *path*). The list of argument strings is terminated by a **(char *)0** argument.

argv is an array of character pointers to null-terminated strings. These strings constitute the argument list available to the new process image. By convention, *argv* must have at least one member, and it should point to a string that is the same as *path* (or its last component). *argv* is terminated by a null pointer.

envp is an array of character pointers to null-terminated strings. These strings constitute the environment for the new process image. *envp* is terminated by a null pointer. For **execl( ), execv( ), execvp( ),** and **execlp( ),** the C run-time start-off routine places a pointer to the environment of the calling process in the global object **extern char** ****environ**, and it is used to pass the environment of the calling process to the new process.

File descriptors open in the calling process remain open in the new process, except for those whose close-on-exec flag is set; (see **fcntl**(2)). For those file descriptors that remain open, the file pointer is unchanged.

Signals that are being caught by the calling process are set to the default disposition in the new process image (see **signal**(3C)). Otherwise, the new process image inherits the signal dispositions of the calling process.

If the set-user-ID mode bit of the new process file is set (see **chmod**(2)), **exec( )** sets the effective user ID of the new process to the owner ID of the new process file. Similarly, if the set-group-ID mode bit of the new process file is set, the effective group ID of the new process is set to the group ID of the new process file. The real user ID and real group ID of the new process remain the same as those of the calling process.

If the effective user-ID is **root** or super-user, the set-user-ID and set-group-ID bits will be honored when the process is being controlled by **ptrace**.

The shared memory segments attached to the calling process will not be attached to the new process (see **shmop**(2)). Memory mappings in the calling process are unmapped before the new process begins execution (see **mmap**(2)).

Profiling is disabled for the new process; see **profil**(2).

Timers created by **timer_create**(3R) are deleted before the new process begins execution.

Any outstanding asynchronous I/O operations may be cancelled.

The new process also inherits the following attributes from the calling process:

nice value (see **nice**(2))
scheduler class and priority (see **priocntl**(2))
process ID
parent process ID
process group ID
supplementary group IDs

semadj values (see **semop**(2))
session ID (see **exit**(2) and **signal**(3C))
trace flag (see **ptrace**(2) request 0)
time left until an alarm (see **alarm**(2))
current working directory
root directory
file mode creation mask (see **umask**(2))
resource limits (see **getrlimit**(2))
utime, **stime**, **cutime**, and **cstime** (see **times**(2))
file-locks (see **fcntl**(2) and **lockf**(3C))
controlling terminal
process signal mask (see **sigprocmask**(2))
pending signals (see **sigpending**(2))

Upon successful completion, **exec( )** marks for update the **st_atime** field of the file, unless the file is on a read-only file system. Should the **exec( )** succeed, the process image file is considered to have been **open**( ) -ed. 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 **exec( ).**

RETURN VALUES

If **exec( )** returns to the calling process, an error has occurred; the return value is –1 and **errno** is set to indicate the error.

ERRORS

exec( ) will fail and return to the calling process if one or more of the following are true:

E2BIG The number of bytes in the new process's argument list is greater than the system-imposed limit of **ARG_MAX** bytes. The argument list limit is sum of the size of the argument list plus the size of the environment's exported shell variables.

EACCES Search permission is denied for a directory listed in the new process file's path prefix.

EACCES The new process file is not an ordinary file.

EACCES The new process file mode denies execute permission.

EAGAIN Total amount of system memory available when reading using raw I/O is temporarily insufficient.

EFAULT An argument points to an illegal address.

EINTR A signal was caught during the **exec( )** function.

ELOOP Too many symbolic links were encountered in translating *path* or *file*.

EMULTIHOP Components of *path* require hopping to multiple remote machines and the file system type does not allow it.

ENAMETOOLONG The length of the *file* or *path* argument exceeds {**PATH_MAX**}, or the length of a *file* or *path* component exceeds {**NAME_MAX**} while {**_POSIX_NO_TRUNC**} is in effect.

ENOENT One or more components of the new process path name of the file do not exist or is a null pathname.

ENOEXEC The **exec( )** is not an **execlp( )** or **execvp( ),** and the new process file has the appropriate access permission but an invalid magic number in its header.

ENOLINK *path* points to a remote machine and the link to that machine is no longer active.

ENOMEM The new process requires more memory than is allowed by the limit imposed by **getrlimit( )**, see **brk**(2). MAXMEM.

ENOTDIR A component of the new process path of the file prefix is not a directory.

SEE ALSO

ksh(1), **ps**(1), **sh**(1), **alarm**(2), **brk**(2), **chmod**(2), **exit**(2), **fcntl**(2), **fork**(2), **getrlimit**(2), **mmap**(2), **nice**(2), **priocntl**(2), **profil**(2), **ptrace**(2), **semop**(2), **shmop**(2), **signal**(3C), **sigpending**(2), **sigprocmask**(2), **times**(2), **umask**(2), **lockf**(3C), **timer_create**(3R), **system**(3S), **a.out**(4), **environ**(5), **xpg4**(5)

WARNINGS

If a program is **setuid** to a user ID other than the super-user, and the program is executed when the real user ID is super-user, then the program has some of the powers of a super-user as well.