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 'posix_spawn.3p' command

$ man posix_spawn.3p

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

posix_spawn, posix_spawnp ? spawn a process (ADVANCED REALTIME)

SYNOPSIS

```
#include <spawn.h>
int posix_spawn(pid_t *restrict pid, const char *restrict path,
    const posix_spawn_file_actions_t *file_actions,
    const posix_spawnattr_t *restrict attrp,
    char *const argv[restrict], char *const envp[restrict]);
int posix_spawnp(pid_t *restrict pid, const char *restrict file,
    const posix_spawn_file_actions_t *file_actions,
    const posix_spawnattr_t *restrict attrp,
    char *const argv[restrict], char *const envp[restrict]);
```

DESCRIPTION

The posix_spawn() and posix_spawnp() functions shall create a new process (child process) from the specified process image. The new process image shall be constructed from a regular executable file called the new process image file.

When a C program is executed as the result of this call, it shall be 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 vari?able:

```
extern char **environ;
```

shall be initialized as a pointer to an array of character pointers to the environment strings.

The argument argv is an array of character pointers to null-terminated strings. The last member of this array shall be a null pointer and is not counted in argc. These strings constitute the argument list avail?able to the new process image. The value in argv[0] should point to a filename string that is associated with the process image being started by the posix_spawn() or posix_spawnp() function.

The argument envp is an array of character pointers to null-terminated strings. These strings constitute the environment for the new process image. The environment array is terminated by a null pointer.

The number of bytes available for the combined argument and environment lists of the child process is {ARG_MAX}. The implementation shall specify in the system documentation (see the Base Definitions volume of POSIX.1?2017, Chapter 2, Conformance) whether any list overhead, such as length words, null terminators, pointers, or alignment bytes, is in?cluded in this total.

The path argument to posix_spawn() is a pathname that identifies the new process image file to execute.

The file parameter to posix_spawnp() shall be used to construct a path?name that identifies the new process image file. If the file parameter contains a <slash> character, the file parameter shall be used as the pathname for the new process image file. Otherwise, the path prefix for this file shall be obtained by a search of the directories passed as the environment variable PATH (see the Base Definitions volume of POSIX.1?2017, Chapter 8, Environment Variables). If this environment

variable is not defined, the results of the search are implementation-defined.

If file_actions is a null pointer, then file descriptors open in the calling process shall remain open in the child process, except for those whose close-on-exec flag FD_CLOEXEC is set (see fcntl()). For those file descriptors that remain open, the child process shall not inherit any file locks, but all remaining attributes of the correspond? ing open file descriptions (see fcntl()), shall remain unchanged.

If file_actions is not NULL, then the file descriptors open in the child process shall be those open in the calling process as modified by the spawn file actions object pointed to by file_actions and the FD_CLOEXEC flag of each remaining open file descriptor after the spawn file actions have been processed. The effective order of processing the spawn file actions shall be:

1. The set of open file descriptors for the child process shall ini? tially be the same set as is open for the calling process. The child process shall not inherit any file locks, but all remaining attributes of the corresponding open file descriptions (see fc? ntl()), shall remain unchanged.
2. The signal mask, signal default actions, and the effective user and group IDs for the child process shall be changed as specified in the attributes object referenced by attrp.
3. The file actions specified by the spawn file actions object shall be performed in the order in which they were added to the spawn file actions object.
4. Any file descriptor that has its FD_CLOEXEC flag set (see fcntl()) shall be closed.

If file descriptor 0, 1, or 2 would otherwise be closed in the new process image created by posix_spawn() or posix_spawnp(), implementa? tions may open an unspecified file for the file descriptor in the new process image. If a standard utility or a conforming application is ex? ecuted with file descriptor 0 not open for reading or with file de? scriptor 1 or 2 not open for writing, the environment in which the

utility or application is executed shall be deemed non-conforming, and consequently the utility or application might not behave as described in this standard.

The posix_spawnattr_t spawn attributes object type is defined in <spawn.h>. It shall contain at least the attributes defined below.

If the POSIX_SPAWN_SETPGROUP flag is set in the spawn-flags attribute of the object referenced by attrp, and the spawn-pgroup attribute of the same object is non-zero, then the child's process group shall be as specified in the spawn-pgroup attribute of the object referenced by at? trp.

As a special case, if the POSIX_SPAWN_SETPGROUP flag is set in the spawn-flags attribute of the object referenced by attrp, and the spawn-pgroup attribute of the same object is set to zero, then the child shall be in a new process group with a process group ID equal to its process ID.

If the POSIX_SPAWN_SETPGROUP flag is not set in the spawn-flags attri? bute of the object referenced by attrp, the new child process shall in? herit the parent's process group.

If the POSIX_SPAWN_SETSCHEDPARAM flag is set in the spawn-flags attri? bute of the object referenced by attrp, but POSIX_SPAWN_SETSCHEDULER is not set, the new process image shall initially have the scheduling pol? icy of the calling process with the scheduling parameters specified in the spawn-schedparam attribute of the object referenced by attrp.

If the POSIX_SPAWN_SETSCHEDULER flag is set in the spawn-flags attri? bute of the object referenced by attrp (regardless of the setting of the POSIX_SPAWN_SETSCHEDPARAM flag), the new process image shall ini? tially have the scheduling policy specified in the spawn-schedpolicy attribute of the object referenced by attrp and the scheduling parame? ters specified in the spawn-schedparam attribute of the same object.

The POSIX_SPAWN_RESETIDS flag in the spawn-flags attribute of the ob? ject referenced by attrp governs the effective user ID of the child process. If this flag is not set, the child process shall inherit the effective user ID of the parent process. If this flag is set, the ef?

fective user ID of the child process shall be reset to the parent's real user ID. In either case, if the set-user-ID mode bit of the new process image file is set, the effective user ID of the child process shall become that file's owner ID before the new process image begins execution.

The POSIX_SPAWN_RESETIDS flag in the spawn-flags attribute of the ob? ject referenced by attrp also governs the effective group ID of the child process. If this flag is not set, the child process shall inherit the effective group ID of the parent process. If this flag is set, the effective group ID of the child process shall be reset to the parent's real group ID. In either case, if the set-group-ID mode bit of the new process image file is set, the effective group ID of the child process shall become that file's group ID before the new process image begins execution.

If the POSIX_SPAWN_SETSIGMASK flag is set in the spawn-flags attribute of the object referenced by attrp, the child process shall initially have the signal mask specified in the spawn-sigmask attribute of the object referenced by attrp.

If the POSIX_SPAWN_SETSIGDEF flag is set in the spawn-flags attribute of the object referenced by attrp, the signals specified in the spawn-sigdefault attribute of the same object shall be set to their default actions in the child process. Signals set to the default action in the parent process shall be set to the default action in the child process. Signals set to be caught by the calling process shall be set to the de? fault action in the child process.

Except for SIGCHLD, signals set to be ignored by the calling process image shall be set to be ignored by the child process, unless otherwise specified by the POSIX_SPAWN_SETSIGDEF flag being set in the spawn-flags attribute of the object referenced by attrp and the signals being indicated in the spawn-sigdefault attribute of the object referenced by attrp.

If the SIGCHLD signal is set to be ignored by the calling process, it is unspecified whether the SIGCHLD signal is set to be ignored or to

the default action in the child process, unless otherwise specified by the POSIX_SPAWN_SETSIGDEF flag being set in the spawn_flags attribute of the object referenced by attrp and the SIGCHLD signal being indi? cated in the spawn_sigdefault attribute of the object referenced by at? trp.

If the value of the attrp pointer is NULL, then the default values are used.

All process attributes, other than those influenced by the attributes set in the object referenced by attrp as specified above or by the file descriptor manipulations specified in file_actions, shall appear in the new process image as though fork() had been called to create a child process and then a member of the exec family of functions had been called by the child process to execute the new process image.

It is implementation-defined whether the fork handlers are run when posix_spawn() or posix_spawnp() is called.

RETURN VALUE

Upon successful completion, posix_spawn() and posix_spawnp() shall re? turn the process ID of the child process to the parent process, in the variable pointed to by a non-NULL pid argument, and shall return zero as the function return value. Otherwise, no child process shall be created, the value stored into the variable pointed to by a non-NULL pid is unspecified, and an error number shall be returned as the func? tion return value to indicate the error. If the pid argument is a null pointer, the process ID of the child is not returned to the caller.

ERRORS

These functions may fail if:

EINVAL The value specified by file_actions or attrp is invalid.

If this error occurs after the calling process successfully returns from the posix_spawn() or posix_spawnp() function, the child process may exit with exit status 127.

If posix_spawn() or posix_spawnp() fail for any of the reasons that would cause fork() or one of the exec family of functions to fail, an error value shall be returned as described by fork() and exec, respec?

tively (or, if the error occurs after the calling process successfully returns, the child process shall exit with exit status 127).

If POSIX_SPAWN_SETPGROUP is set in the spawn-flags attribute of the object referenced by attrp, and posix_spawn() or posix_spawnp() fails while changing the child's process group, an error value shall be returned as described by setpgid() (or, if the error occurs after the calling process successfully returns, the child process shall exit with exit status 127).

If POSIX_SPAWN_SETSCHEDPARAM is set and POSIX_SPAWN_SETSCHEDULER is not set in the spawn-flags attribute of the object referenced by attrp, then if posix_spawn() or posix_spawnp() fails for any of the reasons that would cause sched_setparam() to fail, an error value shall be returned as described by sched_setparam() (or, if the error occurs after the calling process successfully returns, the child process shall exit with exit status 127).

If POSIX_SPAWN_SETSCHEDULER is set in the spawn-flags attribute of the object referenced by attrp, and if posix_spawn() or posix_spawnp() fails for any of the reasons that would cause sched_setscheduler() to fail, an error value shall be returned as described by sched_setscheduler() (or, if the error occurs after the calling process successfully returns, the child process shall exit with exit status 127).

If the file_actions argument is not NULL, and specifies any close, dup2, or open actions to be performed, and if posix_spawn() or posix_spawnp() fails for any of the reasons that would cause close(), dup2(), or open() to fail, an error value shall be returned as described by close(), dup2(), and open(), respectively (or, if the error occurs after the calling process successfully returns, the child process shall exit with exit status 127). An open file action may, by itself, result in any of the errors described by close() or dup2(), in addition to those described by open().

The following sections are informative.

EXAMPLES

None.

APPLICATION USAGE

These functions are part of the Spawn option and need not be provided on all implementations.

See also the APPLICATION USAGE section for exec.

RATIONALE

The posix_spawn() function and its close relation posix_spawnp() have been introduced to overcome the following perceived difficulties with fork(): the fork() function is difficult or impossible to implement without swapping or dynamic address translation.

* Swapping is generally too slow for a realtime environment.
* Dynamic address translation is not available everywhere that POSIX might be useful.
* Processes are too useful to simply option out of POSIX whenever it must run without address translation or other MMU services.

Thus, POSIX needs process creation and file execution primitives that can be efficiently implemented without address translation or other MMU services.

The posix_spawn() function is implementable as a library routine, but both posix_spawn() and posix_spawnp() are designed as kernel opera? tions. Also, although they may be an efficient replacement for many fork()/exec pairs, their goal is to provide useful process creation primitives for systems that have difficulty with fork(), not to provide drop-in replacements for fork()/exec.

This view of the role of posix_spawn() and posix_spawnp() influenced the design of their API. It does not attempt to provide the full func? tionality of fork()/exec in which arbitrary user-specified operations of any sort are permitted between the creation of the child process and the execution of the new process image; any attempt to reach that level would need to provide a programming language as parameters. Instead, posix_spawn() and posix_spawnp() are process creation primitives like the Start_Process and Start_Process_Search Ada language bindings pack? age POSIX_Process_Primitives and also like those in many operating sys? tems that are not UNIX systems, but with some POSIX-specific additions.

To achieve its coverage goals, posix_spawn() and posix_spawnp() have control of six types of inheritance: file descriptors, process group ID, user and group ID, signal mask, scheduling, and whether each signal ignored in the parent will remain ignored in the child, or be reset to its default action in the child.

Control of file descriptors is required to allow an independently writ? ten child process image to access data streams opened by and even gen? erated or read by the parent process without being specifically coded to know which parent files and file descriptors are to be used. Con? trol of the process group ID is required to control how the job control of the child process relates to that of the parent.

Control of the signal mask and signal defaulting is sufficient to sup? port the implementation of system(). Although support for system() is not explicitly one of the goals for posix_spawn() and posix_spawnp(), it is covered under the ``at least 50%'' coverage goal.

The intention is that the normal file descriptor inheritance across fork(), the subsequent effect of the specified spawn file actions, and the normal file descriptor inheritance across one of the exec family of functions should fully specify open file inheritance. The implementa? tion need make no decisions regarding the set of open file descriptors when the child process image begins execution, those decisions having already been made by the caller and expressed as the set of open file descriptors and their FD_CLOEXEC flags at the time of the call and the spawn file actions object specified in the call. We have been assured that in cases where the POSIX Start_Process Ada primitives have been implemented in a library, this method of controlling file descriptor inheritance may be implemented very easily.

We can identify several problems with posix_spawn() and posix_spawnp(), but there does not appear to be a solution that introduces fewer prob? lems. Environment modification for child process attributes not speci? fiable via the attrp or file_actions arguments must be done in the par? ent process, and since the parent generally wants to save its context, it is more costly than similar functionality with fork()/exec. It is

also complicated to modify the environment of a multi-threaded process temporarily, since all threads must agree when it is safe for the environment to be changed. However, this cost is only borne by those invocations of posix_spawn() and posix_spawnp() that use the additional functionality. Since extensive modifications are not the usual case, and are particularly unlikely in time-critical code, keeping much of the environment control out of posix_spawn() and posix_spawnp() is appropriate design.

The posix_spawn() and posix_spawnp() functions do not have all the power of fork()/exec. This is to be expected. The fork() function is a wonderfully powerful operation. We do not expect to duplicate its functionality in a simple, fast function with no special hardware requirements. It is worth noting that posix_spawn() and posix_spawnp() are very similar to the process creation operations on many operating systems that are not UNIX systems.

Requirements

The requirements for posix_spawn() and posix_spawnp() are:

* They must be implementable without an MMU or unusual hardware.
* They must be compatible with existing POSIX standards.

Additional goals are:

* They should be efficiently implementable.
* They should be able to replace at least 50% of typical executions of fork().
* A system with posix_spawn() and posix_spawnp() and without fork() should be useful, at least for realtime applications.
* A system with fork() and the exec family should be able to implement posix_spawn() and posix_spawnp() as library routines.

Two-Syntax

POSIX exec has several calling sequences with approximately the same functionality. These appear to be required for compatibility with existing practice. Since the existing practice for the posix_spawn*() functions is otherwise substantially unlike POSIX, we feel that simplicity outweighs compatibility. There are, therefore, only two names

for the posix_spawn*() functions.

The parameter list does not differ between posix_spawn() and posix_spawnp(); posix_spawnp() interprets the second parameter more elaborately than posix_spawn().

Compatibility with POSIX.5 (Ada)

The Start_Process and Start_Process_Search procedures from the POSIX_Process_Primitives package from the Ada language binding to POSIX.1 encapsulate fork() and exec functionality in a manner similar to that of posix_spawn() and posix_spawnp(). Originally, in keeping with our simplicity goal, the standard developers had limited the capa? bilities of posix_spawn() and posix_spawnp() to a subset of the capa? bilities of Start_Process and Start_Process_Search; certain non-default capabilities were not supported. However, based on suggestions by the ballot group to improve file descriptor mapping or drop it, and on the advice of an Ada Language Bindings working group member, the standard developers decided that posix_spawn() and posix_spawnp() should be suf? ficiently powerful to implement Start_Process and Start_Process_Search. The rationale is that if the Ada language binding to such a primitive had already been approved as an IEEE standard, there can be little jus? tification for not approving the functionally-equivalent parts of a C binding. The only three capabilities provided by posix_spawn() and posix_spawnp() that are not provided by Start_Process and Start_Process_Search are optionally specifying the child's process group ID, the set of signals to be reset to default signal handling in the child process, and the child's scheduling policy and parameters. For the Ada language binding for Start_Process to be implemented with posix_spawn(), that binding would need to explicitly pass an empty sig? nal mask and the parent's environment to posix_spawn() whenever the caller of Start_Process allowed these arguments to default, since posix_spawn() does not provide such defaults. The ability of Start_Process to mask user-specified signals during its execution is functionally unique to the Ada language binding and must be dealt with in the binding separately from the call to posix_spawn().

Process Group

The process group inheritance field can be used to join the child process with an existing process group. By assigning a value of zero to the spawn-pgroup attribute of the object referenced by attrp, the setpgid() mechanism will place the child process in a new process group.

Threads

Without the posix_spawn() and posix_spawnp() functions, systems without address translation can still use threads to give an abstraction of concurrency. In many cases, thread creation suffices, but it is not al? ways a good substitute. The posix_spawn() and posix_spawnp() functions are considerably ``heavier'' than thread creation. Processes have sev? eral important attributes that threads do not. Even without address translation, a process may have base-and-bound memory protection. Each process has a process environment including security attributes and file capabilities, and powerful scheduling attributes. Processes ab? stract the behavior of non-uniform-memory-architecture multi-processors better than threads, and they are more convenient to use for activities that are not closely linked.

The posix_spawn() and posix_spawnp() functions may not bring support for multiple processes to every configuration. Process creation is not the only piece of operating system support required to support multiple processes. The total cost of support for multiple processes may be quite high in some circumstances. Existing practice shows that support for multiple processes is uncommon and threads are common among ``tiny kernels''. There should, therefore, probably continue to be AEPs for operating systems with only one process.

Asynchronous Error Notification

A library implementation of posix_spawn() or posix_spawnp() may not be able to detect all possible errors before it forks the child process. POSIX.1?2008 provides for an error indication returned from a child process which could not successfully complete the spawn operation via a special exit status which may be detected using the status value re?

turned by wait(), waitid(), and waitpid().

The stat_val interface and the macros used to interpret it are not well suited to the purpose of returning API errors, but they are the only path available to a library implementation. Thus, an implementation may cause the child process to exit with exit status 127 for any error de? tected during the spawn process after the posix_spawn() or posix_spawnp() function has successfully returned.

The standard developers had proposed using two additional macros to in? terpret stat_val. The first, WIFSPAWNFAIL, would have detected a sta? tus that indicated that the child exited because of an error detected during the posix_spawn() or posix_spawnp() operations rather than dur? ing actual execution of the child process image; the second, WSPAWN? ERRNO, would have extracted the error value if WIFSPAWNFAIL indicated a failure. Unfortunately, the ballot group strongly opposed this because it would make a library implementation of posix_spawn() or posix_spawnp() dependent on kernel modifications to waitpid() to be able to embed special information in stat_val to indicate a spawn fail? ure.

The 8 bits of child process exit status that are guaranteed by POSIX.1?2008 to be accessible to the waiting parent process are insuf? ficient to disambiguate a spawn error from any other kind of error that may be returned by an arbitrary process image. No other bits of the exit status are required to be visible in stat_val, so these macros could not be strictly implemented at the library level. Reserving an exit status of 127 for such spawn errors is consistent with the use of this value by system() and popen() to signal failures in these opera? tions that occur after the function has returned but before a shell is able to execute. The exit status of 127 does not uniquely identify this class of error, nor does it provide any detailed information on the na? ture of the failure. Note that a kernel implementation of posix_spawn() or posix_spawnp() is permitted (and encouraged) to return any possible error as the function value, thus providing more detailed failure in? formation to the parent process.

Thus, no special macros are available to isolate asynchronous posix_spawn() or posix_spawnp() errors. Instead, errors detected by the posix_spawn() or posix_spawnp() operations in the context of the child process before the new process image executes are reported by setting the child's exit status to 127. The calling process may use the WIFEX? ITED and WEXITSTATUS macros on the stat_val stored by the wait() or waitpid() functions to detect spawn failures to the extent that other status values with which the child process image may exit (before the parent can conclusively determine that the child process image has be? gun execution) are distinct from exit status 127.

FUTURE DIRECTIONS

None.

SEE ALSO

alarm(), chmod(), close(), dup(), exec, exit(), fcntl(), fork(), fs? tatat(), kill(), open(), posix_spawn_file_actions_addclose(), posix_spawn_file_actions_adddup2(), posix_spawn_file_actions_destroy(), posix_spawnattr_destroy(), posix_spawnattr_getsigdefault(), posix_spaw? nattr_getflags(), posix_spawnattr_getpgroup(), posix_spaw? nattr_getschedparam(), posix_spawnattr_getschedpolicy(), posix_spaw? nattr_getsigmask(), sched_setparam(), sched_setscheduler(), setpgid(), setuid(), times(), wait(), waitid()

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

COPYRIGHT

Any typographical or formatting errors that appear in this page are most likely to have been introduced during the conversion of the source files to man page format. To report such errors, see https://www.ker?nel.org/doc/man-pages/reporting_bugs.html .