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 'system.3p' command

$ man system.3p

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

system ? issue a command

SYNOPSIS

#include <stdlib.h>

int system(const char *command);

DESCRIPTION

The functionality described on this reference page is aligned with the ISO C standard. Any conflict between the requirements described here and the ISO C standard is unintentional. This volume of POSIX.1?2017 defers to the ISO C standard.

If command is a null pointer, the system() function shall determine whether the host environment has a command processor. If command is not a null pointer, the system() function shall pass the string pointed to by command to that command processor to be executed in an implementa? tion-defined manner; this might then cause the program calling system() to behave in a non-conforming manner or to terminate.

The system() function shall behave as if a child process were created

using fork(), and the child process invoked the sh utility using ex?
ecl() as follows:

execl(<shell path>, "sh", "-c", command, (char *)0);

where <shell path> is an unspecified pathname for the sh utility. It is unspecified whether the handlers registered with pthread_atfork() are called as part of the creation of the child process.

The system() function shall ignore the SIGINT and SIGQUIT signals, and shall block the SIGCHLD signal, while waiting for the command to termi?
nate. If this might cause the application to miss a signal that would have killed it, then the application should examine the return value from system() and take whatever action is appropriate to the applica?
tion if the command terminated due to receipt of a signal.

The system() function shall not affect the termination status of any child of the calling processes other than the process or processes it itself creates.

The system() function shall not return until the child process has ter?
minated.

The system() function need not be thread-safe.

RETURN VALUE

If command is a null pointer, system() shall return non-zero to indi?
cate that a command processor is available, or zero if none is avail?
able. The system() function shall always return non-zero when command is NULL.

If command is not a null pointer, system() shall return the termination status of the command language interpreter in the format specified by waitpid(). The termination status shall be as defined for the sh util?
ity; otherwise, the termination status is unspecified. If some error prevents the command language interpreter from executing after the child process is created, the return value from system() shall be as if the command language interpreter had terminated using exit(127) or _exit(127). If a child process cannot be created, or if the termina?
tion status for the command language interpreter cannot be obtained, system() shall return -1 and set errno to indicate the error.

ERRORS

The system() function may set errno values as described by fork().

In addition, system() may fail if:

ECHILD The status of the child process created by system() is no longer available.

The following sections are informative.

EXAMPLES

None.

APPLICATION USAGE

If the return value of system() is not -1, its value can be decoded through the use of the macros described in <sys/wait.h>. For conve? nience, these macros are also provided in <stdlib.h>.

Note that, while system() must ignore SIGINT and SIGQUIT and block SIGCHLD while waiting for the child to terminate, the handling of sig? nals in the executed command is as specified by fork() and exec. For example, if SIGINT is being caught or is set to SIG_DFL when system() is called, then the child is started with SIGINT handling set to SIG_DFL.

Ignoring SIGINT and SIGQUIT in the parent process prevents coordination problems (two processes reading from the same terminal, for example) when the executed command ignores or catches one of the signals. It is also usually the correct action when the user has given a command to the application to be executed synchronously (as in the '!' command in many interactive applications). In either case, the signal should be delivered only to the child process, not to the application itself.

There is one situation where ignoring the signals might have less than the desired effect. This is when the application uses system() to per? form some task invisible to the user. If the user typed the interrupt character ("^C", for example) while system() is being used in this way, one would expect the application to be killed, but only the executed command is killed. Applications that use system() in this way should carefully check the return status from system() to see if the executed command was successful, and should take appropriate action when the

command fails.

Blocking SIGCHLD while waiting for the child to terminate prevents the application from catching the signal and obtaining status from sys? tem()'s child process before system() can get the status itself.

The context in which the utility is ultimately executed may differ from that in which system() was called. For example, file descriptors that have the FD_CLOEXEC flag set are closed, and the process ID and parent process ID are different. Also, if the executed utility changes its en? vironment variables or its current working directory, that change is not reflected in the caller's context.

There is no defined way for an application to find the specific path for the shell. However, confstr() can provide a value for PATH that is guaranteed to find the sh utility.

Using the system() function in more than one thread in a process or when the SIGCHLD signal is being manipulated by more than one thread in a process may produce unexpected results.

RATIONALE

The system() function should not be used by programs that have set user (or group) ID privileges. The fork() and exec family of functions (ex? cept execlp() and execvp()), should be used instead. This prevents any unforeseen manipulation of the environment of the user that could cause execution of commands not anticipated by the calling program.

There are three levels of specification for the system() function. The ISO C standard gives the most basic. It requires that the function ex? ists, and defines a way for an application to query whether a command language interpreter exists. It says nothing about the command language or the environment in which the command is interpreted.

POSIX.1?2008 places additional restrictions on system(). It requires that if there is a command language interpreter, the environment must be as specified by fork() and exec. This ensures, for example, that close-on-exec works, that file locks are not inherited, and that the process ID is different. It also specifies the return value from sys? tem() when the command line can be run, thus giving the application

some information about the command's completion status.

Finally, POSIX.1?2008 requires the command to be interpreted as in the shell command language defined in the Shell and Utilities volume of POSIX.1?2017.

Note that, system(NULL) is required to return non-zero, indicating that there is a command language interpreter. At first glance, this would seem to conflict with the ISO C standard which allows system(NULL) to return zero. There is no conflict, however. A system must have a com? mand language interpreter, and is non-conforming if none is present. It is therefore permissible for the system() function on such a system to implement the behavior specified by the ISO C standard as long as it is understood that the implementation does not conform to POSIX.1?2008 if system(NULL) returns zero.

It was explicitly decided that when command is NULL, system() should not be required to check to make sure that the command language inter? preter actually exists with the correct mode, that there are enough processes to execute it, and so on. The call system(NULL) could, theo? retically, check for such problems as too many existing child pro? cesses, and return zero. However, it would be inappropriate to return zero due to such a (presumably) transient condition. If some condition exists that is not under the control of this application and that would cause any system() call to fail, that system has been rendered non-con? forming.

Early drafts required, or allowed, system() to return with errno set to [EINTR] if it was interrupted with a signal. This error return was re? moved, and a requirement that system() not return until the child has terminated was added. This means that if a waitpid() call in system() exits with errno set to [EINTR], system() must reissue the waitpid().

This change was made for two reasons:

1. There is no way for an application to clean up if system() returns [EINTR], short of calling wait(), and that could have the undesir? able effect of returning the status of children other than the one started by system().

2. While it might require a change in some historical implementations, those implementations already have to be changed because they use wait() instead of waitpid().

Note that if the application is catching SIGCHLD signals, it will re?ceive such a signal before a successful system() call returns.

To conform to POSIX.1?2008, system() must use waitpid(), or some simi?lar function, instead of wait().

The following code sample illustrates how system() might be implemented on an implementation conforming to POSIX.1?2008.

```
#include <signal.h>
int system(const char *cmd)
{
    int stat;
    pid_t pid;
    struct sigaction sa, savintr, savequit;
    sigset_t saveblock;
    if (cmd == NULL)
        return(1);
    sa.sa_handler = SIG_IGN;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sigemptyset(&savintr.sa_mask);
    sigemptyset(&savequit.sa_mask);
    sigaction(SIGINT, &sa, &savintr);
    sigaction(SIGQUIT, &sa, &savequit);
    sigaddset(&sa.sa_mask, SIGCHLD);
    sigprocmask(SIG_BLOCK, &sa.sa_mask, &saveblock);
    if ((pid = fork()) == 0) {
        sigaction(SIGINT, &savintr, (struct sigaction *)0);
        sigaction(SIGQUIT, &savequit, (struct sigaction *)0);
        sigprocmask(SIG_SETMASK, &saveblock, (sigset_t *)0);
        execl("/bin/sh", "sh", "-c", cmd, (char *)0);
        _exit(127);
```

```
        }
        if (pid == -1) {
            stat = -1; /* errno comes from fork() */
        } else {
            while (waitpid(pid, &stat, 0) == -1) {
                if (errno != EINTR){
                    stat = -1;
                    break;
                }
            }
        }
        sigaction(SIGINT, &savintr, (struct sigaction *)0);
        sigaction(SIGQUIT, &savequit, (struct sigaction *)0);
        sigprocmask(SIG_SETMASK, &saveblock, (sigset_t *)0);
        return(stat);
    }
```

Note that, while a particular implementation of system() (such as the one above) can assume a particular path for the shell, such a path is not necessarily valid on another system. The above example is not por? table, and is not intended to be.

Note also that the above example implementation is not thread-safe. Implementations can provide a thread-safe system() function, but doing so involves complications such as how to restore the signal disposi? tions for SIGINT and SIGQUIT correctly if there are overlapping calls, and how to deal with cancellation. The example above would not restore the signal dispositions and would leak a process ID if cancelled. This does not matter for a non-thread-safe implementation since canceling a non-thread-safe function results in undefined behavior (see Section 2.9.5.2, Cancellation Points). To avoid leaking a process ID, a thread-safe implementation would need to terminate the child process when acting on a cancellation.

One reviewer suggested that an implementation of system() might want to use an environment variable such as SHELL to determine which command

interpreter to use. The supposed implementation would use the default command interpreter if the one specified by the environment variable was not available. This would allow a user, when using an application that prompts for command lines to be processed using system(), to spec? ify a different command interpreter. Such an implementation is discour? aged. If the alternate command interpreter did not follow the command line syntax specified in the Shell and Utilities volume of POSIX.1?2017, then changing SHELL would render system() non-conforming. This would affect applications that expected the specified behavior from system(), and since the Shell and Utilities volume of POSIX.1?2017 does not mention that SHELL affects system(), the application would not know that it needed to unset SHELL.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.9.5.2, Cancellation Points, exec, pipe(), pthread_atfork(), wait()

The Base Definitions volume of POSIX.1?2017, <limits.h>, <signal.h>, <stdlib.h>, <sys_wait.h>

The Shell and Utilities volume of POSIX.1?2017, sh

COPYRIGHT

nel.org/doc/man-pages/reporting_bugs.html .