-- --

IEEE P1003.2 Draft 11.2 – September 1991

B.3 C Binding for Shell Command Interface

B.3.0.1 C Binding for Shell Command Interface Rationale. *(This subclause is not a part of P1003.2)*

The *system* () and *popen* () functions should not be used by programs that have set user (or group) ID privileges, as defined in POSIX.1 {8}. The *fork* () and *exec* family of functions [except *execlp* () and *execvp* ()], also defined in POSIX.1 {8}, should be used instead. This prevents any unforeseen manipulation of the user's environment that could cause execution of commands not anticipated by the calling program.

If the original and '*popen* ()ed'' processes both intend to read or write or read and write a common file, and either will be using FILE-type C functions [*fread* (), *fwrite* (), etc.], the rules in POSIX.1 {8}8.2.3 must be observed.

B.3.1 C Binding for Execute Command

Function: *system* ()

B.3.1.1 Synopsis

```
#include <stdlib.h>
int system(const char *command );
```

B.3.1.2 Description

This standard requires the *system* () function as described in the C Standard {7}.

The *system* () function shall execute the command specified by the string pointed to by *command*. The environment of the executed command shall be as if a child process were created using the POSIX.1 {8}*fork* () function, and the child process invoked the `sh` utility (see 4.56) using the POSIX.1 {8}*execl* () function as follows:

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

where *<shell path>* is an unspecified pathname for the `sh` utility.

The *system* () function shall ignore the SIGINT and SIGQUIT signals, and block the SIGCHLD signal, while waiting for the command to terminate. 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 application 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(es) it itself creates.

The *system* () function shall not return until the child process has terminated.

B.3.1.3 Returns

If *command* is **NULL**, the *system*() function shall return nonzero.

If *command* is not **NULL**, the *system*() function shall return the termination status of the command language interpreter in the format specified by the *waitpid*() function in POSIX.1 {8}. The termination status of the command language interpreter is as specified for the `sh` utility, except that 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 termination status for the command language interpreter cannot be obtained, *system*() shall return −1 and set *errno* to indicate the error.

B.3.1.4 Errors

The *system*() function may set *errno* values as described by *fork*() in POSIX.1 {8}.

B.3.1.5 Rationale. *(This subclause is not a part of P1003.2)*

The C Standard {7} specifies that when *command* is **NULL**, *system*() returns nonzero if there is a command interpreter available and zero if one is not available. At first reading, it might appear that POSIX.2 conflicts with this, since it requires *system*(**NULL**) to always return nonzero. There is no conflict, however. A POSIX.2 implementation must always have a command interpreter available, and is nonconforming if none is present. It is therefore permissible for the *system*() function on a POSIX.2 system to implement the behavior specified by the C Standard {7} as long as it is understood that the implementation is not POSIX.2 conforming if *system*(**NULL**) returns zero. 1

Note that, while *system*() must ignore SIGINT and SIGQUIT and block SIGCHLD while waiting for the child to terminate, the handling of signals 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 will be 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 perform 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 will be 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 *system* ()'s child process before *system* () can get the status itself.

Examples, Usage

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

Earlier drafts of this standard required, or allowed, *system* () to return with *errno* [EINTR] if it was interrupted with a signal. This error return was removed, 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* [EINTR], *system* () must re-issue 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 undesirable effect of returning 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 receive such a sig- 1
nal before a successful *system* () call returns. 1

History of Decisions Made

The C Standard {7} requires that a call to *system* () with a **NULL** will return a nonzero value, indicating the presence of a command language interpreter available to the system. It was explicitly decided that when *command* is **NULL**, *system* () should not be required to check to make sure that the command language interpreter actually exists with the correct mode, that there are enough processes to execute it, etc. The call *system* (**NULL**) could, theoretically, check for such problems as too many existing child processes, 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 nonconformant.

Modified in Draft 6 to reflect the availability of the *waitpid* () function in POSIX.1 {8}. To conform to this standard, *system* () must use *waitpid* (), or some similar function, instead of *wait* ().

Figure B-1 illustrates how *system* () might be implemented on a POSIX.1 {8} implementation.

```
#include <signal.h>

int system(const char *cmd)                                        1
{
    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);                                 1
    sigemptyset(&savequit.sa_mask);                                1
    sigaction(SIGINT, &sa, &savintr);
    sigaction(SIGQUIT, &sa, &savequit);
    sigaddset(&sa.sa_mask, SIGCHLD);                               1
    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);
}
```

Figure B-1 – Sample *system* () Implementation

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 portable, and is not intended to be. 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.

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 specify a different command interpreter. Such an implementation is discouraged. If the alternate command interpreter did not follow the command line syntax specified in POSIX.2, then changing **SHELL** would render *system* () nonconformant. This would affect applications that expected the specified behavior from *system* (), and since this standard does not mention that **SHELL** affects *system* (), the application would not know that it needed to unset **SHELL** .

B.3.2 C Binding for Pipe Communications with Programs

Functions: *popen* (), *pclose* ()

B.3.2.1 Synopsis

```
#include <stdio.h>

FILE *popen(const char *command , const char *mode );

int pclose(FILE *stream );
```

B.3.2.2 Description

The *popen* () function shall execute the command specified by the string *command* . It shall create a pipe between the calling program and the executed command, and return a pointer to a C Standard {7} stream that can be used to either read from or write to the pipe. The *pclose* () function shall close the stream, wait for the command to terminate, and return the termination status from the command language interpreter.

The environment of the executed command shall be as if a child process were created within the *popen* () call using the *fork* () function, and the child invoked the `sh` utility using the call:

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

where <*shell path*> is an unspecified pathname for the `sh` utility. However, 1
popen () shall ensure that any streams from previous *popen* () calls that remain 1
open in the parent process are closed in the new child process. 1

The *mode* argument to *popen* () is a string that specifies I/O mode:

(1) If *mode* is `"r"`, when the child process is started its file descriptor STDOUT_FILENO shall be the writable end of the pipe, and the file descriptor *fileno* (*stream*) in the calling process, where *stream* is the stream pointer returned by *popen* (), shall be the readable end of the pipe.