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 'wait.1p' command

$ man wait.1p

WAIT(1P) POSIX Programmer's Manual WAIT(1P)

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

wait ? await process completion

SYNOPSIS

wait [pid...]

DESCRIPTION

When an asynchronous list (see Section 2.9.3.1, Examples) is started by the shell, the process ID of the last command in each element of the asynchronous list shall become known in the current shell execution en? vironment; see Section 2.12, Shell Execution Environment.

If the wait utility is invoked with no operands, it shall wait until all process IDs known to the invoking shell have terminated and exit with a zero exit status.

If one or more pid operands are specified that represent known process IDs, the wait utility shall wait until all of them have terminated. If one or more pid operands are specified that represent unknown process IDs, wait shall treat them as if they were known process IDs that ex? ited with exit status 127. The exit status returned by the wait utility

shall be the exit status of the process requested by the last pid oper?

and.

The known process IDs are applicable only for invocations of wait in the current shell execution environment.

OPTIONS

None.

OPERANDS

The following operand shall be supported:

pid One of the following:

1. The unsigned decimal integer process ID of a command, for which the utility is to wait for the termination.
2. A job control job ID (see the Base Definitions volume of POSIX.1?2017, Section 3.204, Job Control Job ID) that identifies a background process group to be waited for. The job control job ID notation is applicable only for invocations of wait in the current shell execution envi? ronment; see Section 2.12, Shell Execution Environment. The exit status of wait shall be determined by the last command in the pipeline.

   Note: The job control job ID type of pid is only available on systems supporting the User Porta? bility Utilities option.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables shall affect the execution of wait:

LANG Provide a default value for the internationalization vari? ables that are unset or null. (See the Base Definitions vol? ume of POSIX.1?2017, Section 8.2, Internationalization Vari? ables for the precedence of internationalization variables used to determine the values of locale categories.)

LC_ALL If set to a non-empty string value, override the values of all the other internationalization variables.

LC_CTYPE Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters in arguments).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard er? ror.

NLSPATH Determine the location of message catalogs for the processing of LC_MESSAGES.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

The standard error shall be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

If one or more operands were specified, all of them have terminated or were not known by the invoking shell, and the status of the last oper? and specified is known, then the exit status of wait shall be the exit status information of the command indicated by the last operand speci? fied. If the process terminated abnormally due to the receipt of a sig? nal, the exit status shall be greater than 128 and shall be distinct from the exit status generated by other signals, but the exact value is unspecified. (See the kill -l option.) Otherwise, the wait utility shall exit with one of the following values:

0 The wait utility was invoked with no operands and all process IDs known by the invoking shell have terminated.

1?126 The wait utility detected an error.

127 The command identified by the last pid operand specified is un?

known.

CONSEQUENCES OF ERRORS

Default.

The following sections are informative.

APPLICATION USAGE

On most implementations, wait is a shell built-in. If it is called in a subshell or separate utility execution environment, such as one of the following:

```
(wait)
nohup wait ...
find . -exec wait ... \;
```

it returns immediately because there are no known process IDs to wait for in those environments.

Historical implementations of interactive shells have discarded the exit status of terminated background processes before each shell prompt. Therefore, the status of background processes was usually lost unless it terminated while wait was waiting for it. This could be a se? rious problem when a job that was expected to run for a long time actu? ally terminated quickly with a syntax or initialization error because the exit status returned was usually zero if the requested process ID was not found. This volume of POSIX.1?2017 requires the implementation to keep the status of terminated jobs available until the status is re? quested, so that scripts like:

```
j1&
p1=$!
j2&
wait $p1
echo Job 1 exited with status $?
wait $!
echo Job 2 exited with status $?
```

work without losing status on any of the jobs. The shell is allowed to

discard the status of any process if it determines that the application cannot get the process ID for that process from the shell. It is also required to remember only {CHILD_MAX} number of processes in this way. Since the only way to get the process ID from the shell is by using the '!' shell parameter, the shell is allowed to discard the status of an asynchronous list if "$!" was not referenced before another asynchro? nous list was started. (This means that the shell only has to keep the status of the last asynchronous list started if the application did not reference "$!". If the implementation of the shell is smart enough to determine that a reference to "$!" was not saved anywhere that the ap? plication can retrieve it later, it can use this information to trim the list of saved information. Note also that a successful call to wait with no operands discards the exit status of all asynchronous lists.)

If the exit status of wait is greater than 128, there is no way for the application to know if the waited-for process exited with that value or was killed by a signal. Since most utilities exit with small values, there is seldom any ambiguity. Even in the ambiguous cases, most appli? cations just need to know that the asynchronous job failed; it does not matter whether it detected an error and failed or was killed and did not complete its job normally.

EXAMPLES

Although the exact value used when a process is terminated by a signal is unspecified, if it is known that a signal terminated a process, a script can still reliably determine which signal by using kill as shown by the following script:

```
sleep 1000&
pid=$!
kill -kill $pid
wait $pid
echo $pid was terminated by a SIG$(kill -l $?) signal.
```

If the following sequence of commands is run in less than 31 seconds:

```
sleep 257 | sleep 31 &
jobs -l %%
```

either of the following commands returns the exit status of the second sleep in the pipeline:

```
wait <pid of sleep 31>
wait %%
```

RATIONALE

The description of wait does not refer to the waitpid() function from the System Interfaces volume of POSIX.1?2017 because that would need? lessly overspecify this interface. However, the wording means that wait is required to wait for an explicit process when it is given an argu? ment so that the status information of other processes is not consumed. Historical implementations use the wait() function defined in the Sys? tem Interfaces volume of POSIX.1?2017 until wait() returns the re? quested process ID or finds that the requested process does not exist. Because this means that a shell script could not reliably get the sta? tus of all background children if a second background job was ever started before the first job finished, it is recommended that the wait utility use a method such as the functionality provided by the wait? pid() function.

The ability to wait for multiple pid operands was adopted from the Ko? rnShell.

This new functionality was added because it is needed to determine the exit status of any asynchronous list accurately. The only compatibility problem that this change creates is for a script like

```
while sleep 60 do
    job& echo Job started $(date) as $! done
```

which causes the shell to monitor all of the jobs started until the script terminates or runs out of memory. This would not be a problem if the loop did not reference "$!" or if the script would occasionally wait for jobs it started.

FUTURE DIRECTIONS

None.

SEE ALSO

Chapter 2, Shell Command Language, kill, sh

The Base Definitions volume of POSIX.1?2017, Section 3.204, Job Control Job ID, Chapter 8, Environment Variables

The System Interfaces volume of POSIX.1?2017, wait()

COPYRIGHT