NAME

wait – await process completion

SYNOPSIS

sh

wait [*n*]
wait [%*jobid* . . .]

csh

wait [*n*]

ksh

wait [*pid* . . . **]**

DESCRIPTION

sh

Wait for your background process whose process id is *n* and report its termination status. If *n* is omitted, all your shell's currently active background processes are waited for and the return code will be zero. **wait** accepts a job identifier, when Job Control is enabled, and the argument, *jobid*, is preceded by a percent-sign.

The shell itself executes **wait**, without creating a new process. If you get the error message **cannot fork, too many processes**, try using the **wait** command to clean up your background processes. If this doesn't help, the system process table is probably full or you have too many active foreground processes. (There is a limit to the number of process ids associated with your login, and to the number the system can keep track of.)

Not all the processes of a 3- or more-stage pipeline are children of the shell, and thus cannot be waited for.

If *n* is not an active process id, all your shell's currently active background processes are waited for and the return code will be zero.

csh

Wait for your background process whose process id is *n* and report its termination status. If *n* is omitted, all your shell's currently active background processes are waited for and the return code will be zero.

The shell itself executes **wait**, without creating a new process. If you get the error message **cannot fork, too many processes**, try using the **wait** command to clean up your background processes. If this doesn't help, the system process table is probably full or you have too many active foreground processes. (There is a limit to the number of process ids associated with your login, and to the number the system can keep track of.)

Not all the processes of a 3- or more-stage pipeline are children of the shell, and thus cannot be waited for.

If *n* is not an active process id, all your shell's currently active background processes are waited for and the return code will be zero.

ksh

When an asynchronous list is started by the shell, the process ID of the last command in each element of the asynchronous list becomes known in the current shell execution environment.

If the **wait** utility is invoked with no operands, it will 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 will wait until all of them have terminated. If one or more *pid* operands are specified that represent unknown process IDs, **wait** will treat them as if they were known process IDs that exited with exit status **127**. The exit status returned by the **wait** utility will be the exit status of the process requested by the last *pid* operand.

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

OPERANDS

The following operand is 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. A job control job ID that identifies a background process group to be waited for.
2. The job control job ID notation is applicable only for invocations of **wait** in the current shell execution environment. The exit status of **wait** is determined by the last command in the pipeline.

Note that the job control job ID type of *pid* is available only on systems supporting the job control option.

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 will return immediately because there will be 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 serious problem when a job that was expected to run for a long time actually terminated quickly with a syntax or initialisation error because the exit status returned was usually zero if the requested process ID was not found. This document requires the implementation to keep the status of terminated jobs available until the status is requested, so that scripts like:

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

wait $!

echo Job 2 exited with status $?

will work without losing status on any of the jobs. The shell is allowed to discard the status of any process that it determines the application cannot get the process ID from the shell. It is also required to remember only 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 **$!** was not referenced before another asynchronous list was started. (This means that the shell only has to keep the status of the last asynchronous list started if the application did not reference **$!**. If the implementation of the shell is smart enough to determine that a reference to **$!** was not saved anywhere that the application 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 applications 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 figure out which signal 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 will return the exit status of the second **sleep** in the pipeline:

wait <*pid of sleep 31i***>**

wait %%

ENVIRONMENT

See **environ**(5) for descriptions of the following environment variables that affect the execution of **wait**: **LC_CTYPE**, **LC_MESSAGES**, and **NLSPATH**.

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 operand specified is known, then the exit status of **wait** will be the exit status information of the command indicated by the last operand specified. If the process terminated abnormally due to the receipt of a signal, the exit status will be greater than **128** and will 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 will 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 unknown.

SEE ALSO

csh(1), **jobs**(1), **ksh**(1), **sh**(1)