-- --

# IEEE P1003.2 Draft 11.2 – September 1991

```
uniq -u -f 1 uniq_0I.t
#01 foo0 bar0 foo1 bar1
#02 bar0 foo1 bar1 foo1
#03 foo0 bar0 foo1 bar1
#04
#07 bar0 foo1 bar1 foo0
```

This suppresses unique lines, comparing each line of the input file data starting from the third character:

```
uniq -d -s 2 uniq_0I.t
```

In the last example, the `uniq` utility found no input matching the above criteria.

**History of Decisions Made**

The `-f` and `-s` options were added to replace the obsolescent `-`*n* and `+`*m* options so that `uniq` could meet the syntax guidelines in an upward-compatible way.

The output specifications in Output Files do not show a terminating `<newline>` because they both specify <*line*>, which includes its own `<newline>` (because of the definition of *line* ).

## 4.70 `wait` — Await process completion

### 4.70.1 Synopsis

`wait` [*pid* . . . ]

### 4.70.2 Description

When an asynchronous list (see 3.9.3.1) is started by the shell, the process ID of
the last command in each element of the asynchronous list shall become known in 1
the current shell execution environment; see 3.12.

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 exited 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* operand.

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

### 4.70.3 Options

None.

### 4.70.4 Operands

The following operand shall be supported by the implementation:

*pid* The unsigned decimal integer process ID of a command, for which the utility is to wait for the termination.

### 4.70.5 External Influences

#### 4.70.5.1 Standard Input

None.

#### 4.70.5.2 Input Files

None.

#### 4.70.5.3 Environment Variables

The following environment variables shall affect the execution of `wait`:

**LANG** This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

**LC_ALL** This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

**LC_CTYPE** This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments).

**LC_MESSAGES** This variable shall determine the language in which messages should be written.

#### 4.70.5.4 Asynchronous Events

Default.

### 4.70.6 External Effects

#### 4.70.6.1 Standard Output

None.

#### 4.70.6.2 Standard Error

Used only for diagnostic messages.

#### 4.70.6.3 Output Files

None.

### 4.70.7 Extended Description

None.

### 4.70.8 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` shall 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 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 in 4.32.) 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 unknown.

### 4.70.9 Consequences of Errors

Default.

### 4.70.10 Rationale. *(This subclause is not a part of P1003.2)*

**Examples, 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.

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.
```

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 initialization error because the exit status returned was usually zero if the requested process ID was not found. POSIX.2 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 {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 `$!` was not referenced before another asynchronous list was started. (This means that the shell only has to keep the status of the last asynchronous list started if the application did not reference `$!`. 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.)

This new functionality was added because it is needed to accurately determine the exit status of any asynchronous list. 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 will cause the shell to keep track of 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.

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.

### History of Decisions Made

The description of `wait` does not refer to the *waitpid* () function from POSIX.1 {8}, because that would needlessly overspecify this interface. However, the wording requires that `wait` is required to wait for an explicit process when it is given an argument, so that the status information of other processes is not consumed. Historical implementations use POSIX.1 {8} *wait* () until *wait* () returns the requested process ID or finds that the requested process does not exist. Because this means that a shell script could not reliably get the status 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 *waitpid* () function in POSIX.1 {8}.

The ability to wait for multiple *pid* operands was adopted from the KornShell at the request of ballot comments and objections.

Some implementations of `wait` support waiting for asynchronous lists identified by the use of job identifiers. For example, `wait %1` would wait for the first background job. This standard does not address job control issues, but allows these features to be added as extensions. Job control facilities will be provided by the UPE.

## 4.71 `wc` — Word, line, and byte count

### 4.71.1 Synopsis

`wc [−clw]` [*file* . . . ]

### 4.71.2 Description

The `wc` utility shall read one or more input files and, by default, write the number of `<newline>`s, words, and bytes contained in each input file to the standard output.

The utility also shall write a total count for all named files, if more than one input file is specified.

The `wc` utility shall consider a *word* to be a nonzero-length string of characters delimited by white space.

### 4.71.3 Options

The `wc` utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The following options shall be supported by the implementation:

`−c` Write to the standard output the number of bytes in each input file.

`−l` Write to the standard output the number of `<newline>`s in each input file.

`−w` Write to the standard output the number of words in each input file.

When any option is specified, `wc` shall report only the information requested by the specified option(s).

### 4.71.4 Operands

The following operand shall be supported by the implementation:

*file* A pathname of an input file. If no *file* operands are specified, the standard input shall be used.