- Page immuable
- Informations
- Pièces jointes
- Autres actions :

- Ubuntu Wiki
- Connexion
- Help

DashAsBinSh

Dash as /bin/sh

Sommaire

In Ubuntu 6.10, the default system shell, `/bin/sh`, was changed to `dash` (the Debian Almquist Shell); previously it had been `bash` (the GNU Bourne-Again Shell). The same change will affect users of Ubuntu 6.06 LTS upgrading directly to Ubuntu 8.04 LTS. This document explains this change and what you should do if you encounter problems.

The default login shell remains `bash`.

Why was this change made?

The major reason to switch the default shell was efficiency. `bash` is an excellent full-featured shell appropriate for interactive use; indeed, it is still the default login shell. However, it is rather large and slow to start up and operate by comparison with `dash`. A large number of shell instances are started as part of the Ubuntu boot process. Rather than change each of them individually to run explicitly under `/bin/dash`, a change which would require significant ongoing maintenance and which would be liable to regress if not paid close attention, the Ubuntu core development team felt that it was best simply to change the default shell. The boot speed improvements in Ubuntu 6.10 were often incorrectly attributed to Upstart, which is a fine platform for future development of the init system but in Ubuntu 6.10 was primarily running in System V compatibility mode with only small behavioural changes. These improvements were in fact largely due to the changed `/bin/sh`.

The Debian policy manual has long mandated that "shell scripts specifying '/bin/sh' as interpreter must only use POSIX features"; in fact, this requirement has been in place since well before the inception of the Ubuntu project. Furthermore, any shell scripts that expected to be portable to other Unix systems, such as the BSDs or Solaris, already honoured this requirement. Thus, we felt that the compatibility impact of this change would be minimal.

Of course, there have been a certain number of shell scripts written specifically for Linux systems, some of which incorrectly stated that they could run with `/bin/sh` when in fact they required `/bin/bash`, and these scripts will have broken due to this change. We regret this breakage, but feel that the proper way to address it is to make the small changes required to those scripts, discussed later in this document. In the longer term, this will promote a cleaner and more efficient system.

(This applies the same philosophy as in C and C++. Programs should be written to the standard, and if they use extensions they should declare them; that way it is clear what extensions are in use and they will at least fail with a much better error message if those extensions are not available.)

My production system has broken and I just want to get it back up!

If you are unlucky enough to have been negatively affected by this change, and only one or two shell scripts are affected, then the quickest way to fix this is to edit these scripts and change the first line to use the correct interpreter. The first line should look something like this (perhaps with some additional options):

```
#! /bin/sh
```

Change that to the following (you can preserve any options you see):

```
#! /bin/bash
```

In Makefiles, you can set the following variable at the top:

```
SHELL = /bin/bash
```

If the problems are more widespread and you want to change the default system shell back, then you can instruct the package management system to stop installing `dash` as `/bin/sh`:

```
sudo dpkg-reconfigure dash
```

Beware that this is a more invasive change, will undo boot speed benefits, and there is even an outside chance that there are a few scripts that now depend on some feature of `dash` that `bash` does not provide! (We expect the last problem to be rare, as the feature set of `dash` is largely a subset of that offered by `bash`, but we mention it for completeness.)

I am a developer. How can I avoid this problem in future?

We recommend that developers of shell scripts adhere to the POSIX standard, omitting those items flagged as XSI extensions. (This subset will be referred to as "POSIX shell" throughout the remainder of this document.) Doing so will improve portability to a variety of Unix systems, and will provide assurance that problems you encounter will be treated as bugs rather than as undocumented features! A special exception is that `echo -n` is guaranteed to be supported, although not other `echo` options (see below).

If you cannot use POSIX shell for some reason (perhaps you inherited maintenance of a large set of scripts and do not have time to rewrite them), then at least ensure that the first line of your script specifies `/bin/bash` as the interpreter.

Developers targetting the `/bin/sh` shipped by Solaris will have more stringent requirements, since that shell is a traditional Bourne shell predating the Korn shell and the POSIX standard: in particular, the `test -e` option is absent with no replacement (though most uses can be replaced with `test -r`), and `test -h` must be used rather than the `test -L` alias to test for symlinks. This document does not attempt to address Solaris in depth; consult your system's documentation.

There follows a list of some of the more common `bash` extensions (often referred to as "bashisms"). This list is not complete, but we believe that it covers most of the common extensions found in the wild. You can use `dash -n` to check that a script will run under `dash` without actually running it; this is not a perfect test (particularly not if `eval` is used), but is good enough for most purposes. The `checkbashisms` command in the `devscripts` package may also be helpful (it will output a warning `possible bashism in` for every occurrence).

As further reading, install the `autoconf-doc` package and read the "Portable Shell" pages in `info autoconf`. This documentation is aimed at people targetting a wider variety of systems which may not support the POSIX standard, but it is nevertheless useful even to those assuming POSIX.

[

The `[` command (a.k.a. `test`) must be used carefully in portable scripts. A very common pitfall is to use the `==` operator, which is a bashism (related `checkbashisms` warning text: `'==' in a test, it should use '=' instead`); use `=` instead.

While `dash` supports most uses of the `-a` and `-o` options, they have very confusing semantics even in `bash` and are best avoided. Commands like the following:

```
[ \( "$foo" = "$bar" -a -f /bin/baz \) -o ! -x /bin/quux ]
```

should be replaced with:

```
(([ "$foo" = "$bar" ] && [ -f /bin/baz ]) || [ ! -x /bin/quux ])
```

In other words, use a separate `[` invocation for each single test, combine them with && and ||, and use ordinary parentheses for grouping.

The `-nt` and `-ot` test operators, for comparing the timestamps of two files, are not defined in POSIX, and their behavior differs between bash and dash: dash will not return true if the second file is non-existent, where bash will.

NOTE: The `checkbashisms` script may display warnings of type 'test -a/-o'. You could use the above example to fix this issue.

[[

The `[[` builtin is a bashism (related `checkbashisms` warning text: `'[[ test ]]' instead of '[ test ]' (requires a Korn shell)`), and has somewhat better-defined semantics than `[` (a.k.a. `test`). However, it is still quite reasonable to use `[` instead, and portable scripts must do so. Note that argument handling is not quite the same; as above, use `=` rather than `==`.

Do not be confused by use of `[[` in Autoconf macros, used to double-quote literal strings. This is for a different purpose and is not the same as `bash`'s `[[` builtin. (The convention in Autoconf macros is to spell `[` as `test` to avoid confusion.)

((

`((...))` performs arithmetic expansion, but is a bashism. You should normally use something involving `$((...))` instead, although note that this substitutes the value of the expression.

$((n++)), $((--n)) and similar

These features are not required by POSIX.

foo=$((n++)) can be replaced with foo=$n; $((n=n+1)) foo=$((++n)) can be replaced with foo=$((n=n+1))

{

`bash` supports brace expansions over strings, such as `/usr/lib/libfoo.{a,so}`; this syntax can be useful for abbreviation, but it is not portable. Use other pathname expansions instead, or if that is not possible simply write out all the possibilities.

$'...'

`$'...'` is a bashism to expand escape sequences; for example, `$'\t'` expands to a horizontal tab. Use `"$(printf '\t')"` etc. instead.

$"..."

`$"..."` is a bashism used for translation of strings. Even if you can assume `bash`, do not use this feature, as it has intrinsic security problems! See the GNU gettext info documentation for details.

Instead, you should use the functions provided by `gettext.sh`. Again, see the gettext info documentation.

${...}

The `${...}` syntax performs variable expansion; many forms of this are portable (including the useful `${foo#bar}`, `${foo##bar}`, `${foo%bar}`, and `${foo%%bar}`, but a few are not.

`${!...}` performs indirect variable expansion, which is a bashism; use `eval` instead.

`${parameter/pattern/string}` performs pattern replacement, which is a bashism. See the next section for suggestions on fixing this.

`${parameter:offset:length}` performs substring expansion, which is a bashism. See below for suggestions on fixing this.

Array variables are not portable. With some care about suitable separators they can typically be replaced with a series of ${foo#bar} expansions and the like without noticeably affecting performance.

${parm/?/pat[/str]}

The '${parm/?/pat[/str]}' warning from the `checkbashisms` script.

This is a parameter expansion `${parameter/pattern/string}` supported in bash but not in dash. To replace these, you could make use of `echo`, `sed`, `grep`, and/or `awk` among other tools. For example.

```
# A way to capture the version of OpenGL drivers being used from glxinfo.
OPENGL_VERSION=$(glxinfo | grep "OpenGL version string:")
OPENGL_VERSION=${OPENGL_VERSION/*: /}

# The same but compatible with dash
OPENGL_VERSION=$(glxinfo | grep "OpenGL version string:" | awk 'BEGIN { FS = ":[[:space:]]+" }; { print $2 }')

echo "$OPENGL_VERSION"
```

Here's another example that may be needed when dealing with these bashisms on arrays.

```
# A Perl split like function that works in bash
LIST=$(echo $PATH)
LIST=${LIST//:/ }

# The equivalent as above, but works in dash
LIST=$(awk 'BEGIN { split( ENVIRON["PATH"], array, ":"); for (i=1;i<=length(array);i++) { print array[i]; } }')

for WORD in $LIST; do
        printf "$WORD\n"
done
```

A simple and general way, but requires tr:

```
echo $(echo "originalstring" | tr -s "substringsearched" "newsubstring")
```

${foo:3[:1]}

The '${foo:3[:1]}' warning from the `checkbashisms` script.

This is a substring expansion `${parameter:offset:length}` supported in bash but not in dash. Here's a way to replace

such expansions.

```
# Substring expansion in bash
string_after="some string"
string=${string_after:0:3}

# Equivalent substring expansion in dash.
# NOTE: The offset to use in awk is different than in bash.
# It is equivalent to '<bash_offset> + 1'.
string=$(echo $string_after | awk  '{ string=substr($0, 1, 3); print string; }' )

echo $string
```

Here's another example in which we use two variables in awk.

```
string="some string"
stringlen=${#string}
string=$(echo $string | awk -v var=$stringlen '{ string=substr($0, 1, var - 1); print string; }' )

echo string
```

[^]

Not to be confused by sed's and other program's regular expression syntax. Uses of [^] in case (parameter/word expansion in general) need to be replaced with [!].

E.g. replace:

```
case "foo" in
    [^f]*)
        echo 'not f*'
    ;;
esac
```

with

```
case "foo" in
    [!f]*)
        echo 'not f*'
    ;;
esac
```

$LINENO

POSIX requires that conforming shells expand the special parameter `$LINENO` to the current line number in a script or function; `dash` does not yet support this feature.

$PIPESTATUS

The `$PIPESTATUS` array variable in `bash` contains a list of exit status values from the processes in the most-recently-executed foreground pipeline, which can be useful to detect failing processes that are not the last in their pipeline. `dash` does not support this. Replacing this is messy; the least bad replacement is probably to echo the exit status to an unused file descriptor.

$RANDOM

Many shells, including `bash`, have a magic variable `$RANDOM` which expands to a random integer. You should not rely on this in portable scripts. Workarounds include reading random bytes from `/dev/urandom` or `/dev/random`. For example:

```
random=`hexdump -n 2 -e '/2 "%u"' /dev/urandom`
```

builtin

See:

- 'type' builtin in posix-only shell
- http://stackoverflow.com/questions/3192373/what-are-shell-built-in-commands-in-linux
- http://wiki.linuxquestions.org/wiki/Builtin
- https://en.wikipedia.org/wiki/Shell_builtin

And also the related builtin "type" explained in its section below.

(related `checkbashisms` warning text:
`possible bashism in sample_file line 116 (builtin): if builtin pids 2>/dev/null; then`)

TODO: add hints on how to actually implement POSIX-compliant Best Practice checks for shell builtins.

echo

Options to `echo` are not portable. In particular, the `echo -e` option is implemented by some shells, including `bash`, to expand escape sequences. However, `dash` is one of the other family of shells that instead expands escape sequences by default. Do not rely on either behaviour. If you need to print a string including any backslash characters (related `checkbashisms` warning text: `unsafe echo with backslash`), use the `printf` command instead, which is portable and much more reliable.

As a special exception, `echo -n` is supported on Ubuntu systems, although you may replace it with `printf` if you are concerned about wider portability.

function

The `function` builtin is a bashism (related `checkbashisms` warning text: `'function' to define a function`), and can almost always simply be removed. If you remove it, make sure that there are parentheses after the function name. A function definition in POSIX shell looks like this:

```
function_name () {
    function body
}
```

let

The `let` builtin is a bashism. It can usually be replaced by arithmetic expansion, for example:

```
-    let times=10
-    let times--
+    times=10
+    times=$((times-1))
```

Note: the standard says that you can't count on prefix/postfix support, so we have to use '-1' rather than '--'.

local

Dash (old versions maybe?) and (at least) bash do not handle local the same:

```
-    local a=5 b=6;
+    local a=5;
+    local b=6;
```

The first line in /bin/bash makes a and b local variables. And in /bin/dash this line makes b a global variable!

When executing a command in a subshell and you need its return value be aware that this must be resolved differently:

```
-    local a=$(someCommand) b=$?;
+    local a; local b
+    a=$(someCommand)
+    b=$?
```

This is not specific to dash.

printf %q|%b

bash's help command describes them as:

```
      %b        expand backslash escape sequences in the corresponding argument
      %q        quote the argument in a way that can be reused as shell input
```

Neither are supported by dash or printf(1).

select

The `select` builtin is a bashism. If you need it, sometimes you can write the same logic out by hand, perhaps in a shell function; otherwise it may be best to use `/bin/bash`.

source

The `source` builtin is a bashism (related `checkbashisms` warning text: `'source' instead of '.'`). Write this simply as `.` instead.

`dash` doesn't expand `~` on path search

This means that if you have `~/bin` in the `PATH` environment variable, `bash` finds commands located in `~/bin`, while `dash` doesn't.

This also causes python code like:

```
p4 = subprocess.Popen('p4 files', shell=True)
exitCode = p4.wait()
```

... which would attempt to execute `p4` from `~/bin` through the default shell would fail to find `p4` when `/bin/sh` points to

`dash`, but not when it points to `bash`.

To avoid this, make sure that you change the line that appends `~/bin` to search path in your old .bash_profile or .profile. Instead of:

```
PATH="~/bin:$PATH"
```

use

```
PATH="$HOME/bin:$PATH"
```

declare or typeset

The 'declare' or 'typeset' warning from the `checkbashisms` script.

The declare or typeset builtins in bash (they are exact synonyms) permit restricting the properties of variables. For example, you can declare a variable to be an integer type variable by using `-i`.

Both 'declare' and 'typeset' are not supported in dash. When changing from using declare/typeset, you normally could switch declare/typeset with local and continue as before.

Some other examples to consider (here we'll use `declare` in our examples).

```
# A variable declared with no attributes
-declare foo
+local foo

# Variables declared as arrays. Changed the same way as variables
# declared as integers
-declare -a foo
+local foo

# Variables declared as integers. Usually, nothing else needs to be done
# besides declaring these variables as local variables.
-declare -i foo
+local foo

# Declaring variables readonly from bash to dash
-declare -r foo
+readonly foo

# Using declare -f [functions]
-declare -f
+printf "<contents of function1>"
+printf "<contents of function2>"
+# repeat this until the last function
-declare -f function1 function2 ... functionN
+printf "<contents of function1>"
+printf "<contents of function2>"
+# repeat this until functionN

# Using declare -F [functions]
-declare -F
+printf "declare -f <name of function1>"
+printf "declare -f <name of function2>"
+# repeat this until the last function
-declare -F function1 function2 ... functionN
+printf "<name of function1>"
+printf "<name of function2>"
+# repeat this until functionN

# Using declare -x var[=value]
-declare -x var=value
+export var=value
```

ulimit

The 'ulimit' warning from the `checkbashisms` script.

`ulimit` is a builtin in dash as well as bash. Some changes may be needed however as they support different options between bash and dash.

type

Related `checkbashisms` warning text: `'type' instead of 'which' or 'command -v'`.

`type` is a builtin in dash as well as bash. Some changes may be needed however as they support different options between bash and dash.

time

The 'time' warning from the `checkbashisms` script.

`time` is a special builtin for bash, but not for dash. The program `time` will be used instead under dash.

The simplest way to resolve this is to ensure the package with the shell script depends on the `time` package.

Also, the following can be used to check for the presence of the `time` program.

```
# simple one liner
which time >/dev/null 2>&1 || { echo "time is not installed." && exit 1; }

# if statement
if [ ! "$(command -v time)" ]; then
    echo "time is not installed"
    exit 1
fi
```

kill -[0-9] or -[A-Z]

The 'kill -[0-9] or -[A-Z]' warning from the `checkbashisms` script.

kill is a special builtin in dash, however it is currently undocumented (see bug #396821).

read without variable

read with option other than -r

Related `checkbashisms` warning text: `'read' without a variable in the argument`

The name of a variable should be passed to the 'read' built-in; in bash, when none is passed, the input is stored in the REPLY variable. I.e. a quick way to fix it is by replacing calls to 'read' without variable with 'read REPLY'.

If 'read' is called with the -p option, it can be replaced with a consecutive call to 'echo -n' or 'printf' and 'read'.

```
# Bashism:
read -p "Enter $user's real name: "
# Replacement #1 (recommended):
printf "Enter %s's real name: " "$user"
read REPLY
# Replacement #2:
echo -n "Enter $user's real name: "
read REPLY
```

Note: some versions of checkbashisms used to combine both checks in one which was explained as "should be read [-r] variable".

<<<

"Here strings" are similar to "here docs," and in most cases can easily be replaced with one.

E.g.

```
$ cat <<<"$HOME is where the heart is."
/home/ralph is where the heart is.
$
$ cat <<E
> $HOME is where the heart is.
> E
/home/ralph is where the heart is.
$
```

Other warnings

Here's a list of other warnings that appear using `checkbashisms`. If anyone willing to document their meaning and possible alternatives, please do so. A good package to find these errors is heartbeat (<= 2.1.4-2).

- trap with signal numbers

Specification

The old developer specification is preserved as DashAsBinSh/Spec.

DashAsBinSh (dernière édition le 2015-02-09 21:36:05 par andi @ HSI-KBW-46-223-46-243.hsi.kabel-badenwuerttemberg.de[46.223.46.243]:andi)