



Shell Environments



The Shell Environment

◆ Shell environment

- Consists of a set of variables with values.
- These values are important information for the shell and the programs run from the shell.
 - ❖ Example: **PATH** determines where the shell looks for the file corresponding to your command.
 - ❖ Example: **SHELL** indicates what kind of shell you are using.
- You can define new variables and change the values of the variables.

Shell Variables (1)

- ◆ Shell variables are used by putting a \$ in front of their names
 - e.g. `echo $HOME`
- ◆ Many are defined in `.cshrc` and `.login`
- ◆ Two kinds of shell variables:
 - Environment variables
 - ❖ available in the current shell and the programs invoked from the shell
 - Regular shell variables
 - ❖ not available in programs invoked from this shell

Shell Variables (2)

◆ Setting regular variables:

- `set varname=varvalue`

◆ Example:

```
obelix[1] > set myvar="unix is easy"
```

```
obelix[2] > echo myvar
```

```
myvar
```

```
obelix[3] > echo $myvar
```

```
unix is easy
```

◆ Clearing out regular variables:

```
obelix[4] > unset myvar
```

```
obelix[5] > echo $myvar
```

```
myvar: undefined variable
```

Shell Variables (3)

◆ Setting environment variables:

```
obelix[1] > setenv MYENVVAR "env var "
```

```
obelix[2] > unsetenv MYENVVAR
```

❖ *No "=" sign here!*

◆ Example:

```
obelix[3] > setenv MYENVVAR "Unix is easy"
```

```
obelix[4] > set myregvar = "Windows is easy"
```


```
obelix[5] > tcsh
```

```
obelix[1] > echo $MYENVVAR
```

```
Unix is easy
```

```
obelix[2] > echo $myregvar
```

```
myregvar: undefined variable
```



**Here we enter
A new shell...**

Shell Variables (4)

- ◆ In sh, ksh, bash, regular variables are defined in the following way:

```
% varname=varvalue
```

- ◆ In sh, ksh, bash, environment variables are called “exported variables” and are defined in the following way:

```
% MYENVVAR="env var"
```

```
% export MYENVVAR
```

Shell Variables (5)

◆ Common shell variables:

- **SHELL**: the name of the shell being used
- **PATH**: where to find executables to execute
- **MANPATH**: where man looks for man pages
- **LD_LIBRARY_PATH**: where libraries for executables are found at run time
- **USER**: the user name of the user logged in
- **HOME**: the user's home directory
- **TERM**: the kind of terminal the user is using
- **DISPLAY**: where X program windows are shown
- **HOST**: the name of the host logged on to
- **REMOTEHOST**: the name of the host logged in from

More on Unix Quoting

◆ Single Quotes '...'

- ❖ Stop variable expansion (\$HOME, etc.)

```
obelix[16] > echo "Welcome $HOME"
```

```
Welcome /gaul/s1/student/1999/csnow
```

```
obelix[17] > echo 'Welcome $HOME'
```

```
Welcome $HOME
```

◆ Back Quotes `...`

- ❖ Replace the quotes with the results of the execution of the command.

- ❖ E.g.

```
obelix[18] > set prompt = `hostname`
```


The Search Path

- ▶ How does Unix find commands to execute?
 - If you specify a pathname, the shell looks into that path for the executable.
 - If you specify a filename, (without / in the name), the shell looks for it in the search path.
 - There is a variable `PATH` or `path`
 - `obelix[1] > echo $PATH`
 - `/gaul/s1/student/1999/csnow/bin:/bin:/usr/local/bin:.`
- ▶ The shell does not look for executables in your current directory unless:
 - You specify it explicitly, e.g. `./a.out`
 - `.` is specified in the path variable

Selecting Different Versions of a Command

- ◆ There may be multiple versions of the same command in your search path.

```
obelix[1] > whereis ps
```

```
ps: /usr/bin/ps /usr/ucb/ps
```

- ◆ The shell searches in each directory of the `$PATH` in left to right order and executes the first version.

```
obelix[2]> which ps
```

```
/usr/bin/ps
```

```
obelix[3]> /usr/ucb/ps
```

Shell Startup

- ▶ When `csh` and `tcsh` are executed, they run certain configuration files:
 - `.login` run once when you log in
 - ❖ Contains one-time things like terminal setup.
 - `.cshrc` run each time another [t]csh process runs
 - ❖ Sets lots of variables, like `PATH`.
- ▶ Other shells such as `sh` use a different file, like `.profile` to do similar things.
- ▶ Only modify the lines that you fully understand
- ▶ To reset your shell files, in case of an “accident”, execute the command script:
`/usr/local/bin/reset login env`

The alias Command

▶ alias format:

- `alias alias-name real-command`

- ❖ `alias-name` is one word

- ❖ `real-command` can have spaces in it

▶ Any reference to `alias-name` invokes `real-command`.

▶ Examples:

- `alias rm rm -i`

- `alias cp cp -i`

- `alias mv mv -i`

- `alias ls /usr/bin/ls -CF`

- ❖ This shows us the `/`, `*`, `@` after file names using `ls`.

▶ Put aliases in your `.cshrc` file to set them up whenever you log in to the system!

Command History (1)

◆ obelix[9] > history

```
1 10:57 emacs
2 10:57 ls -l .cshrc
3 10:57 cp .cshrc .cshrc2
4 10:57 emacs .cshrc
5 11:01 ps
6 13:46 pwd
7 13:46 cd ..
8 13:46 pine
9 13:46 history
```

Command History (2)

- ◆ You can rerun a command line in the history
 - `!!` reruns last shell command
 - `!str` rerun the latest command beginning with `str`
 - `!n` (where `n` is a number) rerun command number `n` in the history list
- ◆ `tcsh` allows you to use arrow keys to wander the history list easily.
- ◆ The length of the history list is determined by the variable `history`, likely set in your `.cshrc` file.

```
set history = 40
```
- ◆ The variable `savehist` determines how much history to save in the file named in `histfile` for your next session; these are also likely set in your `.cshrc` file.

Command and Filename Completion

- ◆ In tcsh and bash, you can let the shell complete a long command name by:
 - Typing a prefix of the command.
 - Hitting the **TAB** key.
 - The shell will fill in the rest for you, if possible.
- ◆ tcsh and bash also complete file names:
 - Type first part of file name.
 - Hit the **TAB** key.
 - The shell will complete the rest, if possible.
- ◆ Difference:
 - First word: command completion.
 - Other words: file name completion.