Licence d'informatique
Module de C/Unix

csh — le C shell, Aspects interactifs

Philippe Marquet

Nouvelle révision, septembre 1998

Introduction

Un shell est un programme qui s'exécute au dessus du système d'exploitation. C'est un interpréteur de commandes, une passerelle vers les autres outils, utilitaires et programmes pouvant s'exécuter sur le système d'exploitation.

Le C shell (`csh`) est un shell du système d'exploitation Unix. D'autres shells disponibles sous Unix sont le Bourne shell, `sh`, le plus classique ; le Korn shell, `ksh`, une évolution du Bourne shell ; `tcsh`, une amélioration du C-shell...

Deux modes d'utilisation d'un shell sont distingués : le mode interactif et le mode batch. Le premier mode permet d'exécuter une à une des commandes que l'utilisateur entre au clavier. Sous le second mode, le shell exécute les instructions d'un fichier de commandes : le shell est un véritable langage de programmation ; il inclut des affectations, des structures de contrôle...

Les différents shells disponibles sous Unix ont des caractéristiques et donc des qualités différentes. Il est ainsi habituel d'utiliser le C shell comme interpréteur interactif de commandes pour son interface utilisateur agréable et puissante et d'utiliser le Bourne shell pour l'écriture de programmes shells.

Nous présentons dans ce document les aspects interactifs du C shell. Des détails supplémentaires sont disponibles dans le manuel en ligne :

```
% man csh
```

On consultera aussi la page de manuel `shell_builtins (1)`.

Table des matières

Id : csh.tex,v 1.5 2001/10/30 11 :15 :54 marquet Exp

1 Exécution de commandes Unix

Un utilisateur travaille souvent avec les mêmes commandes et sur les mêmes fichiers. Il est donc agréable de pouvoir définir des abréviations pour ces commandes, de pouvoir relancer, éventuellement éditer une commande préalablement lancée. Ces possibilités sont offertes en C shell via les ¡¡ alias ¿¿ et les ¡¡ history ¿¿. Nous présentons aussi dans cette section les mécanismes permettant l'exécution d'un fichier de commandes, les raccourcis que propose le C shell pour les accès aux fichiers, et les commandes de manipulation de jobs.

1.1 History

Le C shell enregistre chacune des commandes sous un numéro. Il est possible d'afficher ce numéro dans le prompt ; il sera ensuite facile de référencer les commandes par leur numéro :

```
% set prompt = '\!% '
```

Le nouveau prompt est alors (un autre numéro peut apparaître)

```
2%
```

La commande

```
2% set history = 13
```

indique au C shell d'enregistrer les commandes. Il alloue alors de la place pour 13 commandes ; les 13 dernières. Entrons les commandes suivantes (seul le résultat de la commande `history` est affiché) :

```
3% ls -a
4% date
5% who
6% history
    2  set history = 13
    3  ls -a
    4  date
    5  who
    6  history
```

Le C shell a mémorisé les commandes et leur numéro. Examinons quelques commandes d'history. La plupart d'entre-elles sont invoquées par le caractère "`!`". Par exemple, `!!` répète la dernière commande

```
7%  !!
history
```

Remarquons que le C shell affiche la commande avant de l'exécuter. Le `!4` répète la commande numéro 4

```
8%  !4
date
```

Le `!w` répète la dernière commande commençant par "w"

```
9%  !w
who
```

Le `!-2` répète l'avant-dernière commande

```
10%  !-2
date
```

Il est aussi possible de construire une nouvelle commande à partir des précédentes. Par exemple, le `!$` remplace la dernière composante de la commande précédente :

```
11% ls /usr/include/stdio.h
12% vi  !$
vi /usr/include/stdio.h
```

Les autres composantes sont référencées par "`:<digit>`" ou "`:<digit>-<digit>`". Par exemple

```
13% spell sujet1.txt sujet2.txt > sujets.erreurs
14% vi  !!:1-2
vi sujet1.txt sujet2.txt
```

Pour corriger une erreur, il est possible d'utiliser le caractère caret "`^`". Par exemple, si nous tapons, par erreur

```
15% ls /user/include
/user/include not found
```

nous corrigeons le `user` en `usr` par

```
16% ^er^r^
ls /usr/include
```

Pour éditer une commande plus ancienne, on utilise le "`:s`", par exemple :

```
17%  !13:s/spell/ispell/
ispell sujet1.txt sujet2.txt > sujets.erreurs
```

Il est possible de visualiser le résultat d'une history sans exécuter la commande en la faisant suivre de "`:p`"

```
18%  !13:s/spell/ispell/:p
ispell sujet1.txt sujet2.txt > sujets.erreurs
```

Si la modification nous satisfait, nous lançons effectivement la commande par "`!!`"

```
18%  !!
ispell sujet1.txt sujet2.txt > sujets.erreurs
```

1.2 Alias

Un alias est une abréviation pour une commande ou une série de commandes fréquemment utilisée. Par exemple, pour abréger la commande `history` en `h`, nous utilisons

```
% alias h history
```

De même, après la commande

```
% alias ll ls -lag
```

l'alias `ll` peut être utilisé avec ou sans paramètre :

```
% ll
% ll /usr/include
```

Un alias peut abréger plusieurs commandes et/ou alias

```
% alias ah 'h | head -15'
```

liste les anciennes history. (Nous utilisons des quotes "' " car la commande comporte un constructeur pipe " | ".)

La définition d'un alias peut aussi utiliser les caractères " !*" pour référencer les paramètres de l'alias ; par exemple

```
% alias cd 'cd \!* ; pwd'
% cd /usr/include
/usr/include
% cd
/home/enseign/marquet
```

(Nous utilisons le caractère "\verb" devant le " !" pour éviter que ce dernier ne soit interprété par le mécanisme d'history.)

Appelée sans paramètre, la commande `alias` affiche tous les alias définis :

```
% alias
ah   h | head -15
cd   cd  !* ; pwd
h    history
ll   ls -lag
```

Utilisation des alias Nous donnons ici quelques utilisations qui peuvent être faites des alias.

1. Abréger les commandes souvent utilisées

   ```
   % alias m more
   ```

2. Attacher systématiquement une option à une commande. Par exemple, pour qui n'a jamais tapé

   ```
   % rm * .bak
   ```

   (arghh l'espace !) au lieu de

   ```
   % rm *.bak
   ```

   l'alias suivant évite les désagréments

   ```
   % alias rm 'rm -i'
   ```

3. Créer un environnent particulier pour une commande. Par exemple, il est possible d'interdire aux autres utilisateurs d'écrire sur notre terminal (via les commandes `write` et `talk` par exemple) par la commande `mesg`. Pour éviter d'être perturbé lors de l'édition d'un fichier par `vi`, nous définissons un alias

   ```
   % alias vi '(mesg n ; /usr/ucb/vi \!* ; mesg y)'
   ```

   (Nous utilisons le chemin absolu de `vi` pour éviter une boucle d'alias.)

4. Changer l'environnement lors de chaque appel à une commande. Par exemple, pour mettre à jour le prompt avec le répertoire courant

   ```
   % alias cd 'cd \!* ; set prompt = "`pwd` %"'
   % cd /usr/include
   /usr/include %
   ```

1.3 Accès aux fichiers

Que faire sous Unix, si ce n'est manipuler des fichiers ! Quelques caractéristiques du C shell facilitent les références aux noms de fichiers.

Le tilde Le caractère "~" référence notre home directorie ; la chaîne "`~user`" celle de l'utilisateur `user`. Par exemple,

```
% diff ~/.login ~toto/.login
```

compare mon fichier `.login` et celui de l'utilisateur `toto`.

La variable `filec` En positionnant la variable `filec`,

```
% set filec
```

le C shell étend les noms de fichiers. Par exemple

```
% ls
abraba       abracadabra  toto
```

à supposer que nous désirions afficher le contenu du fichier `abracadabra` ; nous tapons (sans retour chariot)

```
% cat a
```

en appuyant sur la touche d'échappement `ESC`, le C shell étend le nom `a` en un nom de fichier possible

```
% cat abra
```

en entrant `^D` (control D), le C shell affiche les différentes possibilités d'extension du nom `abra` formant un nom de fichier valide :

```
% cat abra^D
abraba       abracadabra
```

pour étendre `abra` en `abracadabra`, nous tapons un `c` et un `ESC` supplémentaire :

```
% cat abracadabra
```

1.4 Navigation dans les répertoires

Trois commandes du C shell permettent de naviguer dans les répertoires. Il est en effet courant de vouloir se déplacer momentanément dans une directorie pour ensuite revenir dans la directorie courante.

Pour cela le C shell gère une pile de répertoires.

On se déplace dans un répertoire par la commande `pushd`, par exemple :

```
% cd
% pushd ~/tmp
~/tmp ~
```

qui affiche la pile des répertoires, le répertoire courant en tête.

La commande `dirs` affiche également cette pile de répertoires :

```
% dirs
~/tmp ~
```

On dépile un répertoire par la commande `popd`

```
% popd
~
```

qui affiche le nouvel état de la pile de répertoires.

1.5 Manipulation des jobs

Unix autorise un utilisateur à exécuter simultanément plusieurs processus (à la base un processus est un programme en cours d'exécution). Le C shell propose différentes constructions pour aider le contrôle de ces processus.

Job Le C shell définit la notion de *job*. Un job correspond à une ligne de commande. Un job peut correspondre à un processus, par exemple

```
% who
```

Il peut correspondre à plusieurs processus lancés en séquence

```
% date ; who
```

ou correspondre à plusieurs processus dépendant les uns des autres

```
% who | wc -l
```

Foreground Les jobs peuvent être lancés en *foreground* (avant-plan). C'est le cas pour les exemples ci-dessus. Les entrées/sorties du job sont alors directement reliées au terminal. L'utilisateur attend la fin de son exécution avant d'avoir le prompt et de pouvoir lancer un nouveau job.

Background Si l'exécution d'un job est longue (par exemple recompiler une application complète, chercher les erreurs orthographiques dans un gros document...), il est possible de l'exécuter en *background* (arrière-plan). Pour ce faire, on fait suivre le job du caractère "`&`"

```
% make all &
[1] 1651
```

Le C shell rend de suite la main, sans attendre la fin de l'exécution du job. Si le job doit produire des sorties, il est préférable de rediriger celles-ci dans un fichier ; par exemple

```
% du -s /home/licence/* > du.out &
[2] 1659
```

Lorsque nous lançons un job en background, le C shell répond par un numéro identifiant le job (ici le `[2]`) et un numéro de processus (`1659`). (Le numéro de job est unique à un instant donné pour une exécution du C shell ; le numéro de processus est unique pour l'ensemble du système.)

Statut d'un job Un job peut être actif ou suspendu. Dans ce dernier cas, il sera possible de l'activer à nouveau ; c'est-à-dire qu'il continuera son exécution à partir de l'état dans lequel il a été suspendu.

Gestion des jobs Diverses commandes de gestion de jobs sont disponibles sous le C shell.

La commande `jobs` liste les jobs courants

```
% jobs
[1]  - Running      make all
[2]  + Running      du -s /home/licence/* > du.out
```

Les autres commandes utilisent en paramètre un *numéro de job* ; ce numéro est construit sur les schémas suivants :

`1659`	numéro de processus Unix
`%2`	numéro de job
`%`	le job courant (paramètre par défaut)
`%du`	job de nom "`du*`"
`%?home`	job de nom "`*home*`"

Les principales commandes sont

`bg`	re-lance le job en background
`fg`	re-lance le job en foreground
`kill`	tue le job
`stop`	suspend le job

De plus, lors de l'exécution d'un job en foreground, la commande `^Z` (control Z) suspend le job. Par exemple, nous lançons l'édition d'un fichier avec `vi`

```
% vi du.out
```

Sous l'éditeur, nous tapons `^Z`, le C shell nous rend alors la main et affiche

```
% vi du.out
Stopped
%
```

Le job `vi du.out` est suspendu :

```
% jobs
[1]  - Running       make all
[2]  + Stopped       vi du.out
```

Il est ensuite possible de le relancer pour continuer l'édition :

```
% fg %vi
```

1.6 Exécution d'un shell script

Comme nous l'avons déjà dit, un shell est un véritable langage de programmation et peut être utilisé pour écrire des scripts (un script est un programme écrit en shell). Cependant, il est habituel de choisir un autre shell que le C shell pour interpréter ces fichiers de commandes, typiquement le Bourne shell, `sh`.

Supposons que nous ayons un fichier `monscript` contenant des commandes Bourne shell. Il est possible d'exécuter ces commandes en invoquant directement le Bourne shell :

```
% sh monscript
```

Il est aussi possible de d'indiquer que les scripts shells doivent être exécutés par le Bourne shell en positionnant la variable `shell` :

```
% set shell = '/bin/sh'
```

Le mieux semble d'indiquer dans le fichier script lui-même qu'il doit être exécuté par le Bourne shell en mettant la ligne suivante en tête du fichier :

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

2 Mise en place d'un environnement de travail

Notre environnement de travail sous le C shell peut être personnalisé en positionnant un certain nombre de paramètres. Nous allons présenter les principaux dans cette section.

2.1 Choisir `csh`

Différents shells sont disponibles sous Unix. Lors de la création de notre compte, l'administrateur système a choisi le shell par défaut qui sera exécuté lors de chacune de nos sessions. Il est possible de modifier ce choix initial par la commande `chsh` (ou `ypchsh` si l'on travaille sur un réseau de machines administrées par des *yellow pages*)

```
% chsh
Changing login shell for marquet on homel.
Old shell: /bin/sh
New shell:
```

ce à quoi nous répondons :

```
New shell: csh
```

bien évidement !

2.2 Recherche des commandes

Nous présentons ici comment le C shell retrouve les commandes qu'il doit interpréter :

1. il recherche si la commande correspond à un alias et réalise l'éventuelle substitution ;
2. il recherche si la commande est une commande *built-in*, c'est-à-dire une commande directement interprétée par le shell (par exemple `alias`, `cd`, `set`...) ; si c'est le cas il l'exécute, sinon
3. il recherche la commande dans les *chemins de recherches*. Les chemins de recherches sont une liste de directories contenues dans la variable `path`. Le C shell recherche dans chacune de ces directories s'il existe un fichier portant le nom de la commande.

Il est habituel de mettre ses propres commandes dans une directories ; mettons `~/bin`. Nous modifions la variable `path` en conséquence :

```
% set path = (~/bin $path)
```

Cependant, le C shell ne recherche pas les commandes dans chacun des répertoires de la liste `$path` à chaque invocation de commande ; il construit une table de hashage interne lui permettant une recherche plus rapide des commandes. Considérons que nous ayons mis au point une commande `where` et que nous désirions l'ajouter à nos "outils" :

```
% cp where ~/bin
```

tentons d'exécuter cette commande

```
% where mail
where: Command not found.
```

Le shell ne retrouve pas la commande. Il est nécessaire de reconstruire sa table de hashage à l'aide de la commande

```
% rehash
% where mail
...
```

2.3 Variables

De nombreuses informations définissant l'environnement d'un utilisateur sont déterminées par la valeurs de variables dites *variables d'environnement*. D'autres variables, dites *variables prédéfinies*, déterminent aussi notre environnement de travail. Enfin, des variables peuvent être définies et utilisées par l'utilisateur pour ses propres applications.

Variables environnement Une variable environnement est affectée par la commande `setenv`, sa valeur est référencée par le préfixe "`$`". Par exemple

```
setenv MAIL /usr/spool/mail/$USER
```

positionne la variable `MAIL` (qui contient le nom du fichier de mail) et utilise la variable `USER` (qui contient notre nom d'utilisateur).

L'ensemble des variables d'environnement est affiché par les commandes `printenv` et/ou `env`

```
% printenv
HOME=/home/enseign/marquet
SHELL=/bin/csh
TERM=xterm
USER=marquet
PATH=/home/enseign/marquet/bin:.:/usr/ucb:/bin:/usr/bin:/etc:/usr/etc
LOGNAME=marquet
PWD=/home/enseign/marquet/tmp
OPENWINHOME=/usr/openwin
PRINTER=lp
```

Notons l'existence de la variable `PATH` qui code la même information que la variable `path` sous une autre forme. Le C shell assure la cohérence entre ces deux variables.

La variable `TERM` indique le type du terminal sur lequel nous travaillons ; sa valeur peut être utilisée pour personnaliser l'environnement de travail en fonction du terminal.

La variable `OPENWINHOME` définit la racine dans le système de fichiers à partir de laquelle `openwindow` (le système de gestion de fenêtres) va retrouver ses exécutables et fichiers de configuration. De la même manière, de nombreux logiciels utilisent une telle variable d'environnement.

Variables Les variables prédéfinies et les variables utilisateurs sont affectées par la commande `set` ; leur valeur est référencée par le caractère "`$`". Par exemple

```
% set path = (~/bin $path)
```

affecte la variable `path` comme concaténation de `~/bin` et de son ancienne valeur.

Sans paramètre, la commande `set` affiche la valeur de l'ensemble des variables définies.

```
% set
argv    ()
cwd     /home/enseign/marquet/tmp
filec
history 13
ignoreeof
noclobber
home    /home/enseign/marquet
path    (/home/enseign/marquet/bin . /usr/ucb /bin /usr/bin /etc /usr/etc)
prompt  %
shell   /bin/csh
status  0
term    xterm
user    marquet
```

Les variables suivantes sont principalement utilisées pour connaître l'état de l'environnement :

– La variable `argv` est pour un script C shell l'ensemble des arguments de la commande appelée.

- La variable cwd est le nom de la directorie courante.
- La variable home désigne notre home directorie.
- La variable status est le résultat fournit par la dernière commande.
- La variable term indique le type du terminal sur lequel nous travaillons.
- La variable user est notre nom d'utilisateur

La valeur des variables suivantes permet de modifier l'environnement :

- La variable filec contrôle l'extention des noms de fichiers à l'aide de ^D et ESC (cf. supra).
- La variable history indique le nombre de commandes qui sont mémorisées par le C shell pour les mécanismes d'history.
- En positionnant la variable ignoreeof, nous évitons que la frappe d'un ^D (control D) au prompt ne termine le shell ; on utilise alors la commande exit ou logout pour terminer le shell.
- En positionnant la variable noclobber, on évite qu'une redirection des résultats sur un fichiers n'écrase le contenu du fichier

```
% set noclobber
% spell sujet1.txt > sujets.erreurs
% ls
sujet1.txt      sujet2.txt      sujets.erreurs
% spell sujet1.txt > sujets.erreurs
sujets.erreurs: File exists.
```

  La dernière instruction n'a pas supprimé l'ancien contenu du fichier sujets.erreurs.
- La variable path indique les directories dans lesquelles rechercher les commandes.
- La variable prompt contient la chaîne de caractères qui sera affichée comme prompt.

2.4 Fichiers de configuration

Afin de faciliter la mise au point d'un environnement de travail. Trois fichiers sont utilisés par le C shell, il s'agit des fichiers ~/.login, ~/.cshrc, et ~/.logout.

Préalablement à toute modification de ces fichiers, gardons-en une copie :

```
% cp ~/.login ~/.login.orig
% cp ~/.cshrc ~/.cshrc.orig
% cp ~/.logout ~/.logout.orig
```

Il est également important de se prévenir d'erreurs stupides lors de la mise au point des ces fichiers. Il est nécessaire de rester loggé sur un autre terminal pour "essayer" les modifications faites dans ces fichiers. Sinon, à supposer qu'une erreur se soit glissée dans ces modifications, il pourrait être impossible d'accéder à notre compte sans l'intervention de l'administrateur système !

Ces fichiers contiennent des commandes C shell qui seront exécutées

- Lors du login, c'est-à-dire lors du démarrage d'une session, les commandes des deux fichiers ~/.cshrc puis ~/.login sont exécutées.
- Lors du démarrage d'un nouveau C shell, par exemple lors de l'ouverture d'une fenêtre shell, les commandes du fichier ~/.cshrc sont exécutées.
- Lors de la terminaison d'une session, les commandes du fichier ~/.logout sont exécutées.