L3 Informatique : Cours Systèmes et Réseaux
Programmation système

Olivier Togni
Université de Bourgogne, IEM/LE2I
Bureau G206
olivier.togni@u-bourgogne.fr

September 19, 2017

Langages système

Plusieurs langages de programmation système disponibles pour l'utilisateur :

- shell
- awk
- C
- perl

Rem : On peut tout faire en C, certains traitements sont plus simples à écrire en shell/awk

Environnement

Shell = liste de couples (variable,valeur) + cmd internes

Variables prédéfinies: PATH, IFS, HOME, ...

Cmds internes: `cd`, `echo`, `pwd`, `read`, `set`, `trap`, `wait`, ...

Envir. peut être initialisé grâce aux fichiers d'initialisation du shell, lus au moment de la connexion (sh: .profile, csh:.login, .cshrc; bash:.bash_profile .bashrc)

Environnement

Traitement d'une cmd:

1. Phase de saisie: édition, historique, complétion
2. Phase de substitutions: analyse de la liste de cmd
3. Exécution: la cmd résultant du 2 est recherchée puis exécutée

Exemple: `date > RC.L; ls -l .??* |grep rc >> RC.L`
devient après substitutions

```
date > RC.L; ls-l .bash_history .bash_profile
.dtprofile .profile |grep rc >> RC.L5
```

Traitement d'une commande simple

ligne `<mot0> <mot1> <mot2> ...  [<redirections>]`

Si mot0 est une cmd interne au shell $\Rightarrow$ exec directe

Sinon un fichier de nom `<mot0>` est recherché dans la liste de repertoires de fichiers executable (définie par la variable PATH)
- fichier non trouvé
- fichier non executable par l'util
- fichier exec avec droits OK (ex: cp, rm,...) $\Rightarrow$ lancement de la cmd
- script avec accès OK $\Rightarrow$ lancement d'un shell qui exécute le script. Le chemin absolu du shell peut être précisé dans la 1iere ligne du script: `#!/bin/sh` ou bien `#!/bin/csh`, ...

Valeur d'une cmd

Toute cmd qui termine son exec retourne une valeur.

- terminaison normale: le processus execute un appel système `exit(i)` => valeur=i (normalement 0).
- term. anormale: process interrompu par réception du signal $n^0 s$ (ex: 9 du kill -9) => valeur = c s, avec c=1 si fichier core produit, 0 sinon et s = numéro signal.

La valeur est interprétée par le shell comme un booléen (0=vrai, autre=faux)

ATTENTION de ne pas confondre le **résultat** (ce qui est envoyé vers la sortie std) d'une cmd avec sa valeur.
Ex: pwd valeur=0, resultat=/home1/togni

Exemple

Ex: cat sortie.c

```
main(int argc, char **argv)
{
    if(argc==2) exit(atoi(argv[1]));
}
```

if sortie 0 then echo oui else echo non
if sortie 100 then echo oui else echo non
Le signe $? représente la valeur de la dernière cmd exécutée
Ex prec: sortie 100
echo $?
va donner 100

Cmd composée

pipeline= suite de cmd simples reliées par des symboles de redirection
cmd composée= liste de pipelines reliés par des symboles d'enchainement:

```
cmd1 ; cmd2 séquentiel: cmd2 lancée quand cmd1 est terminée
cmd1 & cmd2 parallele: cmd2 lancée en meme temps que cmd1
cmd1 && cmd2    et logique: cmd2 lancée si cmd1 ret la val vrai
cmd1 || cmd2    ou logique: cmd2 lancée si cmd1 ret faux
```

Rem: `cmd &` suivi d'une commande vide lance cmd en mode détaché (rend la main au shell immédiatement)

Exs: `cd $CHEMIN && ls -l`
`a.out 2>&1>LOG || echo Erreur`

Parenthésage

Permet de forcer les priorités

Ex: `echo $F1` est-ce `$F1` ou bien `$F 1`?

- Accolades Ex: { `cat $F1 || cat $F2`} `| wc -l`
- Parenthèses: idem mais exec de la cmd dans un sous shell => on peut alors rediriger les ES d'une liste facilement. Ex: `(date; pwd; ls-cF) >LOG`

Valeur d'une cmd composée = valeur de la cmd la + à droite

Substitutions

Avant d'executer une ligne, le shell recherche les motifs a substituer dans la ligne de cmd et remplace chacun d'entre eux par le résultat de la substitution dans cet ordre:

1. substitution de variables
   $<var> est remplacé par la valeur de la variable <var>
2. substitution de commandes Le motif ‘<cmd>‘ est remplacé par le résultat de l'exec de la cmd <cmd>
3. substitution de chemins
   Trois motifs:
   $*$: suite éventuellement vide de caractères
   ? : un caractère qcq
   [] : alphabet (ensemble de caractères)

Substitutions

Ex: $[abcdef] = [a - f]$, $[A - Za - z]$=toutes les lettres

Rem: le caractère . en tête de fichier subit un traitement particulier: en général, les fichiers commençant par '.' sont les fichiers de config auxquels on accède en le spécifiant explicitement $\Rightarrow$ motif $*$ produit la liste de tous les fichiers du rép courant ne commençant pas par un '.' et $.*$ ceux commençant par un point.

Quotations

Permettent de bloquer la substitution de certains motifs:

Le caractère \ transforme le carac suivant en carac normal

Une suite entre apostrophes (quote) n'est pas interprétée et est considérée comme un seul mot
Ex: `echo`
`echo '/***\'`
`echo '/bin/[a-c]*'`

Une chaine entre guillemets est partiellement quotée; seuls les motifs de substitution de chemin sont inhibés

Les variables

Variables Shell de type chaîne de caract: ident=let[let+chiffre]*

Affectation: `ident=mot` (attention, pas d'espace autour du ==

Vecteur de variables que l'on peut affecter au moyen de la cmd set:
`set <mot1> <mot2>`
place resp. mot1 dans la 1iere case, mot2 dans la 2ieme, etc
seuls les 9 premiers éléments sont directement accessibles grâce aux motifs:
$1 : val du premier élément, $2 : val du 2ieme, ..., $9 : val du 9ieme
$# : nombre d'éléments courant
$* : liste des éléments

Les variables

Ex: `$ set ‘date‘`

La commande `shift` permet d'accéder, qd ils existent, aux autres éléments du vecteur en décalant chaque élément vers la gauche (le 1er est perdu)

Lors de l'exec d'un script, le vecteur de variables est initialisé avec la liste des paramètres eventuels

Les autres motifs associés au $: `$0`: nom de la cmd courante (shell courant ou script)
`$$`: pid de la cmd courante
`$!`: pid du dernier process détaché

La commande Test

`test <arg1> <arg2> ...`
ou bien `[ <arg1> <arg2> ... ]`

renvoie une valeur vraie (0) ou bien faux (1) suivant arguments.

Tester l'existence et la nature de fichiers:

test	-f		fichier
	-d		rep
	-r	<chemin>	en lecture
	-w		en ecriture
	-s		de taille > 0

Ex: `test -d $NOM && echo repertoire`

La commande Test

Comparer

- 2 chaines qcq `test <chn1> =,!= <chn2>`
- 2 chaines numériques

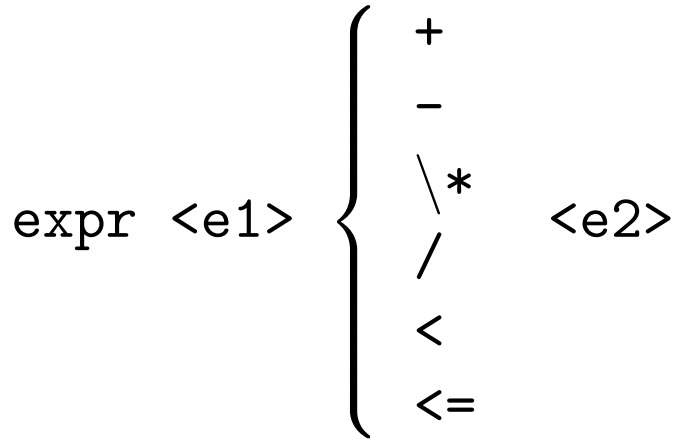
`test <chn1>` { `-eq` | `-ne` | `-gt` | `-ge` | `-lt` | `-le` } `<chn2>`

Ex: `if [ $# -lt 1 ] then echo "usage: $0 fichier fichier exit 1 fi`

Arguments `-o et -a` pour le OU et le ET logiques.
Ex: `test $jour -ge 1 -a $jour -le 31`

La commande expr

Évaluer des expressions arithmétiques ou logiques simples (sans parenthèses)



Ex: `NB=‘expr $NB + 1‘`

En bash, plus simple (et plus rapide) : `NB=$(($NB + 1))`

La commande expr

Recherche de motifs:

`expr <ch1> : <masque>`

A pour résultat la longueur sur laquelle la chaine et le masque coïncident. Le masque est une expr régulière entre quotes
Ex: `expr $1 : 'bon.*'` renvoie 7 si $1 vaut bonjour ou bien bonsoir, 0 si banane

Expressions régulières

Utilisées par egrep, grep, expr, vi, sed, awk, ...
Servent à établir une correspondance carac/carac entre un motif et plusieurs chaînes sauf si un méta-caractère est rencontré dans le motif.
Méta-caractères:

```
^   pour debut de ligne
$   pour fin de ligne
.   remplace un caractère
*   de 0 à n fois le caractère la précédant
[]  ensemble de caractères utilisables
\   ignorer l'effet du caractère suivant
{}  indiquer le nombre d'occurences (attention, préfixées par \)
()  donner des numéros d'ordre aux mots
Exs:
a\{3,4\} définit "aaa" ou "aaaa"
[^0-9] pas un chiffre
```

Structures de contrôle

Si-Alors-Sinon:

```
if <test>
then
    <liste>
[ elif <test>
  then
    <liste>
    .
    .
    . ]
[ else
    <liste> ]
fi
```

Ex de script:

```
#!/bin/sh
if [ $# = 0 ]
then
    cd ..
else
    cd $1
fi
echo "--> `pwd`"
```

Structures de contrôle

Aiguillage:

```
case <mot> in
  <motif> [ | <motif> ...])
    <liste>
    ...
    ;;
  .
  .
  .
esac
```

Exemple d'aiguillage

```
case $N in
  mbox )
    echo $N : boite aux lettres
    ;;
  .* )
    echo $N : fichier de configuration
    ;;
  *.c | *.h | *.cpp )
    echo $N : fichier source C
    ;;
  * )
    echo $N : type inconnu
    ;;
esac
```

Iterations

```
while <test>
do
    <liste>
done
```

Utilisation courante:

```
while [ $# -gt 0 ]
do
    case $1 in
      -o )
          shift
          $OUTPOUT=$1
          ;;
    esac
    shift
done
```

Iterations

La boucle `for` permet de répéter un traitement pour tous les éléments d'une liste

```
for <nom> [ in <mot1> ... ]
do
    <liste>
    .
    .
    .
done
```

Exemple de boucle for

```
for N in *
do
    cp $N $N.svg
done
```

répète la cmd `cp <nom> <nom>.svg` pour `<nom>` prenant successivement pour valeur tous les noms de fichiers du rép. courant.

La cmd `for N in ‘cat $LISTE‘; do ... done` permet d'itérer un traitement sur tous les mots d'une liste de mots contenue dans un fichier de nom contenu dans la variable `LISTE`.

Les échapements

Trois types:
`continue`: passage à l'itération suivante

`break [<n>]`: sortie de $< n >$ itérations

`exit [<n>]`: sortie du prg avec la valeur $< n >$

Fonctions et sous-programmes

Indispensable pour modularité et éviter duplication de code

Définies en utilisant la syntaxe du C

Ex:

```
display_usage ()
{
    echo Usage: $0 [-acfx] fichier
    return 1
}
add()
{
    return ‘expr $1 + $2‘
}
usage ()
{
    display_usage
    exit 1
}
if [ $# -ne 2 ]
then usage
else add $1 $2
```

Fonctions et sous-programmes

Passage des paramètres: seules les variables exportées sont accessible dans le ss-prg. Le prg peut utiliser la valeur de retour comme paramètre.

On utilise le mot clé `builtin` précédant un nom de cmd `<cmd>` pour indiquer que l'on fait référence à la commande `<cmd>` originale , même si celle-ci a été redéfinie par une fonction.

Ex: `cd () { builtin cd $1 ; pwd}`

Programme récursif

```
$ more tree
for N in $1/*
do
   if [ -d $N ]
   then
    echo Répertoire: $N
    tree $N
   else
    echo Fichier $N
   fi
done
```

Autres commandes internes

`:` retourne vrai

`.` ou `source` exécution locale (sans utiliser un autre shell)

`eval` évaluation d'une chaine

`exec` remplacement du shell courant

Exemple: sh-- un shell restreint

```
#!/bin/sh
echo shell restreint
while :
do
  echo -n $PS1
  read COMMAND ARGS
  case $COMMAND in
    PATH=* )
      echo interdit de modifier PATH
      ;;
    cd* )
      echo vous ne pouver bouger
      ;;
    sh |csh | bash )
      echo interdit de changer de shell
      ;;
    exit | bye )
      break 1
      ;;
    * )
      eval $COMMAND $ARGS
      ;;
  esac
done
echo bye
```

Shell: conclusion

- utile et pratique pour de petites tâches liées au système, notamment la gestion des fichiers
- “grosses” applications difficiles à concevoir et à maintenir
- Bash est une extension de sh (et de csh et ksh): tableaux de variables, arithmétique, pile de répertoires, ...

Le langage AWK

Crée par Aho, Weinberger et Kernighan.

Il peut être classé dans la catégorie des filtres. Mais c'est beaucoup plus qu'un utilitaire de gestion des fichiers textes, il intègre un langage interprété très voisin du C.

Cet utilitaire, comme sed, s'applique à un fichier, ou à un flot de données provenant de son entrée, si le fichier (précédé de -f) est absent. Il lit ce fichier ligne par ligne (toutes les lignes par défaut). Les lignes sont séparées par des retour-chariots.

Champs

Chaque ligne (enregistrement) est décomposée en *champs*, les champs étant des chaines de caractères séparées les unes des autres par un caractère particulier appelé le séparateur.

Ce caractère de séparation peut être fourni explicitement, sous la forme de l'option `-F ":"`, par exemple, ou invoqué dans le motif d'une expression rationnelle.

La valeur du premier champ est contenue dans la variable $1, celle du deuxieme dans $2, etc. $0 représente la ligne entière.

Utilisation

La façon habituelle de l'invoquer est
`awk {liste de commandes} fichier_de_données`

ou bien
`awk -f fichier_prg.awk fichier_de_données`

Chaque ligne du programme awk est de la forme
`motif { action }`
ou `motif` est soit une expression awk, soit une expression régulière entre / / décrivant la chaîne à rechercher dans chaque enregistrement du fichier de données et `action` est la suite d'instructions à executer quand l'enregistrement contient le motif.

Motifs

On peut avoir les formes suivantes:

- `BEGIN { action }` L'action est exécutée une fois avant de lire les données
- `END { action }` L'action est exécutée une fois après que awk ait lu toutes les données
- `{ action }` Exécute l'action pour chaque enregistrement lu sur le fichier de données
- `motif` Chaque enregistrement satisfaisant le motif est affiché (action par défaut: `print $0`
- `motif1,motif2 { action }` Exécute l'action pour chaque enregistrement d'entrée à partir du premièr enregistrement satisfaisant motif1 et jusqu'au prochain satisfaisant motif2. Si un enregistrement ultérieur satisfait motif1, le processus est répété

Exemple

```
ls -al | awk '$6 == "oct" { sum += $5 }
                    END { print sum }'
```

Pour chaque ligne produite par `ls -l`, awk teste si le sixième champ est égal à oct, auquel cas il ajoute la valeur du cinquième champ à la variable sum.
Cet enchaînement de commandes compte donc le nombre d'octets total de tous les fichiers du répertoire courant modifiés en Octobre.

Variables

Les variables sont de type numérique ou chaîne de caractères, elles n'ont pas besoin d'être déclarées. La première affectation fixe le type de la variable.
Ex de variables prédéfinies:

- `FS` Séparateur de champs
- `RS` Séparateur de d'enregistrement
- `NF` Le nombre de champs dans l'enregistrement courant
- `NR` Le nombre total d'enregistrements lus dans tous les fichiers de données

Type tableau: un indice de tableau peut être un nombre ou une chaîne. Exemple: a[3]="toto"; a["deux"]=titi.

Instructions

Les opérateurs logiques et arithmétiques sont les mêmes que ceux du langage C (++, +=, etc).

Les Instructions de contrôle ont une forme très similaire à leur homologues en langage C:

`if (`*`expr`*`)` *`instr1`* `[;else` *`instr2`*`]`

`while (expr)` *`instr`*

`do` *`instr`* `while (`*`expr`*`)`

`for (`*`expr1`*`;` *`expr2`*`;` *`expr3`*`)` *`instr`*

`for (`*`var`* `in` *`tableau`*`)` *`instr`*

Fonctions

Les fonctions se définissent de la même manière qu'en C.
L'instruction `return [`*`expr`*`]` en fin de fonction permet de spécifier la valeur de retour de la fonction.
Ex. de fonctions prédéfinies sur les chaînes:

- `index(s,t)` retourne la position dans la chaîne s où la chaîne t apparrait pour la première fois
- `match(s,er)` retourne la position dans la chaîne s où l'expression régulière er apparrait pour la première fois
- `length(s)` calcule le nombre de caractères de s
- `split(s,a,fs)` divise la chaîne s en utilisant le séparateur fs et range les éléments obtenus dans le tableau a
- `substr(s,p[,n])` retourne la sous-chaîne de s commençant par le p-ième caractère et contenant n caractères.

Exemple

```
$1 != prev {print; prev = $1}
```

Affiche chaque ligne du fichier dont le premier champ est différent de celui de la ligne précédente

Compilateur C

Un programme C comprend trois parties:

- les directives pour le pré-processeur,
- les définitions de fonctions,
- le prog principal (fonction main).

Les dir. pré-processeur contiennent les inclusions de bibliothèques `#include...` avec `/usr/include` pour les biblio. prédéfinies et `/usr/include/sys` pour les bib. spécifiques au système.

Utilisation: `cc ou gcc fic.c [-o nomexec]`
Si pas -o alors l'exécutable créé aura pour nom `a.out`

Il y a aussi g++ et `CC`: compilateurs C++

Bibliothèques C

Les librairies contiennent la définition de fonctions susceptibles d'être utilisées par plusieurs programmes

Deux types de librairies :

- statique (`lib.a ou lib.o`) → les définitions des fonctions de la librairies sont incluses dans le fichier executable pendant l'édition des liens
- dynamique (`lib.so`) → seul l'emplacement mémoire des définitions est indiqué; elle seront incluses à l'exécution du programme

Création d'une librairie :

- statique : compilation sans édition de liens puis archivage `gcc -c biblio1.c biblio2.c ; ar csr lib1.a biblio1.o biblio2.o`
- dynamique : option -shared de gcc pour obtenir du code partageable `gcc -o lib1.so -shared biblio1.o biblio2.o`

Entrées/Sorties

E/S formatées:
`printf(const char * format,...)` et
`scanf`: affichage et lecture clavier

`sprintf`: affichage dans une chaine
`fprintf`: affichage dans un fichier

Exemple:

```
#include <math.h>
#include <stdio.h>
fprintf (stdout, "pi = %.5f\n", 4 * atan (1.0));
```

Entrées/Sorties

Accès au fichier:

- Fonctions de bas niveau (appel direct au noyau): un numéro de handle (type `int`) identifie le fichier
  fonctions `int open(chemin, mode)`, `create`, `read`, `write`, `close`
  et constantes `O_RDONLY`, `O_WRONLY`, `O_RDWR`
- Fonctions de haut niveau: un id de flux (type `FILE *`) identifie le fichier
  `fopen`, `fclose`, `fread`, `fwrite`, `fgets` et constantes `r`,`w`,`a`
- Fonction `sync` ou `fsync` pour forcer l'écriture du bloc sur disque

Autres fonctions utiles

- messages d'erreur: bcp d'opérations renvoient le code -1 et positionnent une variable `errno` pour préciser l'erreur. La fonction `perror` de erno.h affiche le msg d'erreur
- lecture des droits sur un fichier: `acces(chemin, mode)` renvoie 0 si accès autorisé
- accès à l'environnement: `getenv(nom)` pour lire une variable d'envir. depuis un prg C, `setenv(nom,valeur,ecrast)` ou `unsetenv(nom)` pour ajouter/supprimer une variable d'envir.
- processus: `int getpid()`, `getppid()`: numéro de process courant/père; `sleep(int sec)` met en sommeil un processus
- manipulations de chaînes: `NULL` n'a pas la même signification sur tous les systèmes, donc utiliser les fonctions `mem...` de string.h: `memcpy`, `memmove`, `memcmp`, `memset`

Ordonnancement de commandes/ Make

Make permet de décrire (dans un fichier `makefile` des instructions de construction de fichiers en indiquant pour chaque cible les fichiers qui sont nécessaires à sa création et les instructions pour créer la cible

Si la cible n'existe pas ou est plus ancienne que l'un de ses composants alors la cible est reconstruite en exécutant les instructions

Usage : partout où il y a du C, C++, LateX, etc, exemple : le noyau Linux

Invocation : `make [-f fich_make] [nom_cible]`
Si pas de cible, construction de la première rencontrée

Make : exemple

Exemple simple :

```
 main : main.o fct.o
        gcc -o main main.o fct.o
main.o : main.c fct.h
         gcc -c main.c
fct.o : fct.c fct.h
        gcc -c fct.c
```

Variables : affectation par `maVar = Val` (ajout) ou `maVar := Val` (écrasement) et utilisation par `$(maVar)`

Prédéfinies :

- $: le nom de la cible
- $¡ : le nom de la première dépendance
- $^: le nom de toutes les dépendances (séparées par des espaces)
- $?: le nom de toutes les dépendances plus récentes que la cible (séparées par des espaces)

Make : règles implicites

On peut réécrire l'exemple précédent en utilisant les variables :

```
OBJ := main.o
OBJ = fct.o  (equivalent à OBJ := $(OBJ) fct.o )
INCL := fct.h

main : $(OBJ)
       gcc -o $@ $^
main.o : main.c $(INCL)
         gcc -c $<
fct.o : fct.c $(INCL)
        gcc -c $<
```

Make : règles implicites

Règles implicites : make connait un certain nombre de règles (par ex. comment tranformer un fichier .c en .o)
Si pas de règle, recherche de règle implicite

```
main : $(OBJ)
       gcc -o $@ $^
```

Création de règle implicite :

```
%.o : $(SRCDIR)/%.c
      gcc {c $<
```

Make : cible abstraite et parallelisation

Cible abstraite : qui ne correspond à aucun fichier ⇒ effectuée à chaque lancement
Ex :

```
OBJ = main.o fct.o
clean :
        rm $(OBJ)
```

Make peut construire deux cibles (indépendantes) en parallèle
`make -j [n]` : n commandes en parallèle

Version distribuée : dmake

Descendants : Ant (Apache group)

Pour aller plus loin

- bash reference manual
  `https://www.gnu.org/software/bash/manual/bash.html`
- shell : cours de D. Bouillet, Telecom SudParis
  `http://www-inf.it-sudparis.eu/cours/UNIX/Shell/SHELL.pdf`
- Le langage awk, H. Wertz,
  `http://www.ai.univ-paris8.fr/ hw/unx5.pdf`
- Programmation avancée sous Linux, M. Mitchel, J. Oldham (traduit par S. Le Ray)