Systèmes d'exploitation

Module UE141 : mise à niveau

Licence professionnelle

Pierre Nerzic

IUT de Lannion

I - Avant de commencer

1) Présentation du cours

Ce module est destiné aux étudiants qui n'ont pas eu une formation complète sur les systèmes d'exploitation, en particulier Unix.

- Rappeler ou préciser les principes d'utilisation des systèmes d'exploitation,
- Rappeler ou présenter la programmation de scripts dans quelques langages.

2) Plan du cours

a) Module UE141

1)Utilisation d'Unix

a)Fichiers et répertoires

b)Commandes et processus

2)Programmation de scripts sur Unix

a) Bourne Shell

b) Awk

3)Programmation de scripts sur Windows

a) JScripts

b) La suite est dans l'UE142

3) Calendrier de l'UE141

4 semaines en septembre : 8h CM et 12h TP

1 DS en 5e semaine et 1 TP noté à rendre + participation aux séances

Partie 1: Rappels, concepts de base

Un PC =

- Unité centrale : mono ou multiprocesseur
- Mémoire centrale (RAM)
- Mémoire secondaire : disques durs, disquettes, cd-rw, bandes...
- Périphériques : écran-clavier-souris, imprimante, scanner, modem...

Ces PC peuvent être isolés, connectés par le réseau (partage de fichiers selon le modèle client-serveur), ou fortement reliés (partage également du temps de calcul).

Le système d'exploitation initialise ces éléments et permet de les utiliser facilement (utilisateur humain et logiciels).

Un système est composé de :

- { Noyau + bios + modules } pour le fonctionnement
- Librairies de fonctions système pour les programmes

- Interfaces : shells, graphiques pour les utilisateurs

Il existe de nombreux systèmes ou variantes, en général 1 ou 2 par type de machine, les choses évoluent très vite (marketing + évolution normale).

Famille Unix : Solaris, HP-ux, linux,... micro et mini-ordinateurs

Famille Windows, pour PC plus ou moins sophistiqués

Autres systèmes :*<inachevé : très rares dans les PME>*

Partie 2: Gestion des données

I - Comptes et connexion

Chaque utilisateur est référencé sur la machine (voir UE142 pour les détails). Avant de travailler, il doit se connecter (login + password).

Il dispose ensuite :

- d'un répertoire : home dir, noté ~ dans la plupart des shells Unix
- d'un shell pour lancer des commandes
- de caractéristiques (numéro UID, groupe...).

Commandes à connaître :

- **`logout`**
- **`passwd`**
- **`who, date, rup`**
- **`man` *`commande`***

II - Fichiers et répertoires

a) Fichiers

Nommage

Les fichiers sont identifiés pour l'utilisateur par un nom. En général, ce nom est suffixé par 1 à 3 caractères (.c, .pas, .txt ...) afin de qualifier le contenu.

- sur l'unix de base, ce suffixe est totalement optionnel,
- sur Windows et les interfaces Unix (Gnome, KDE), le suffixe est associé à un type de données MIME et une application d'affichage, d'édition... Il est donc indispensable de bien nommer les fichiers.

Sur MS-Dos, le nom des fichiers était composé de 8+3 caractères (« noms courts ») ; sur les systèmes modernes, il n'y a pas de telles limites (« noms longs »). Une astuce a permis à Microsoft de faire cohabiter les noms courts et les noms longs pour les mêmes fichiers (voir UE142).

Sur Unix, les fichiers dont le nom commence par un point ne sont pas affichés – ils sont « cachés ». Il faut utiliser l'option –a de ls.

Contenu

Le contenu des fichiers est codé par ce qui s'appelle un format de fichier. Il s'agit d'une norme plus ou moins secrète définissant la signification des octets.

Sur la plupart des systèmes, le format des fichiers est défini par les applications. Le système ne gère que la suite d'octets que les logiciels créent. C'est un inconvénient, car deux applis peuvent ne pas avoir le même niveau de décodage du format.

Certains systèmes sont capables de gérer une structuration des fichiers permettant d'optimiser le stockage sur disque au plus près des données : VMS.

Commandes utiles pour l'affichage de fichiers :

- **file *fichier***
- **cat, more, less, pg**

- **strings**
- **od**
- **touch**

Commandes pour la gestion des fichiers :

- **cp *fichier nouveau***
- **mv ancien nouveau**
- **rm *fichier***

Le contenu des fichiers peut être compressé afin de réduire la place qu'ils occupent sur le disque :

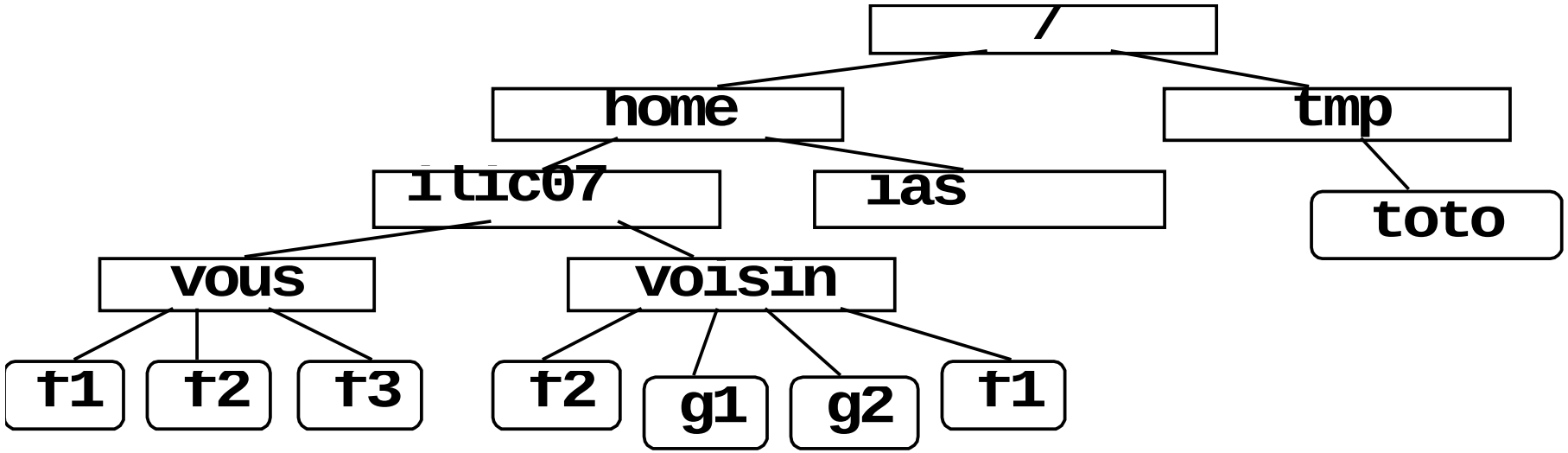
- **compress, uncompress (.Z)**
- **gzip, gunzip (.gz)**
- **bzip2, bunzip2 (.bz2)**

Pour transmettre un fichier par courriel et coder les caractères 8bits sur 6bits.

- `openssl`

b) Répertoires

Arborescence de fichiers et de répertoires



Caractéristiques Unix :

- unicité de la racine (root) de cet arbre, notée « / »
- répertoire courant, dit de travail (working dir) noté « . »

- répertoire d'origine (home dir) noté « ~ »
- unicité du répertoire parent (parent dir) noté « .. »

Caractéristiques Windows :

- il existe autant d'arbres que de volumes connectés (disque durs, cdrom, réseau). Le système attribue une lettre à chacun. En général, le 1er disque s'appelle C. La racine est notée C:\

Chemins (path) : itinéraire à suivre pour aller à un autre répertoire.

Syntaxe générale : [/]rep1/rep2/rep3...

- Absolus, ils désignent un fichier de manière certaine mais souvent longue à taper. Le chemin commence par le signe / sur Unix et \ sur Windows.
    - Une variante sur Unix, quand le chemin commence par ~/ il démarre au répertoire d'origine. Attention, ~rep1/ et ~/rep1/ ne désignent pas le même emplacement, ~rep1 = /home/rep1 (le home dir d'un hypothétique utilisateur appelé rep1), tandis que ~/rep1 = /home/moi/rep1, selon la configuration des comptes.

- Relatifs, ils désignent un fichier à partir de l'emplacement courant. Le chemin ne commence pas par /

Noms complets (full pathname) : chemin/nom désigne un fichier éloigné.

Toute commande acceptant un nom de fichier, accepte aussi un nom complet.

- `mv chemin1/nom1  chemin2/nom2`

c) Autres éléments de l'arbre des fichiers

La commande ls –l affiche la nature des objets listés.

Fichiers normaux = -

Répertoires = d

Outre les fichiers et les répertoires, on rencontre aussi :

Liens : l

Aussi nommés raccourcis (windows) ou alias (macintosh), il s'agit de références à des fichiers existants. Un fichier est situé dans un répertoire, un lien est un autre nom pour ce même fichier, et il est situé en général dans un autre répertoire.

Unix propose deux sortes de liens : matériels (physiques) ou symboliques (logiques). Windows ne propose que la deuxième sorte.

(man ln) Un lien matériel est la façon unix d'associer le nom d'un fichier à son contenu. Les répertoires sont des listes de couples (nom du fichier, n° interne du fichier appelé i-node). Un contenu de fichier (i-node) peut avoir plusieurs noms : autant de couples (nom1, n°), (nom2, même n°)... qui peuvent être placés dans des répertoires différents. Aucun de ces noms n'est « original » ou privilégié et **`ls -l`** ne montre pas que ce sont des liens.

Un fichier n'est effacé réellement que lorsque son dernier nom est supprimé. Le nombre de noms (liens) d'un fichier est indiqué par la commande ls –l. Sur certains systèmes, les liens physiques ne peuvent

être établis qu'à l'intérieur d'un même volume (car le n° d'i-node seul n'est pas un identifiant suffisant).

Un lien symbolique est d'un tout autre genre. Il s'agit d'un petit fichier spécial qui contient un nom complet : le chemin d'accès au fichier original. Le système remplace le lien par le chemin effectif. Il se peut que le fichier original n'existe pas. Les liens symboliques permettent de désigner des fichiers situés sur d'autres volumes, en particulier ceux montés par NFS. **`ls -l`** affiche l.

`ln` *`source lien`*

Périphériques

Unix uniformise la représentation des périphériques contenant des données. Chacun est vu sous la forme d'un fichier. Écrire dans un tel fichier = envoyer des données au périphérique, lire = recevoir des données.

Il y a deux types de périphériques :

Mode caractère (ls –l affiche c) : on peut échanger un nombre quelconque de données. Ex : imprimante, souris, clavier

Mode bloc (ls –l affiche b) : on ne peut qu'échanger des blocs (ex : 1 Ko) de données via des buffers. Ex : disques durs, cdrom

Tubes nommés : p

Il s'agit d'un ancien dispositif de communication entre processus. Il fonctionne comme un tube fifo, tout en étant un fichier ayant une durée de vie indépendante des programmes. ls –l les indique par la lettre p.

Sockets : s

C'est le dispositif actuel de communication entre processus et machines. C'est également un FIFO ayant une certaine durée de vie. ls –l montre s.

d) Stockage de l'arbre des fichiers

Volumes

Un volume est le nom qu'on donne au support de stockage : disquette, cdrom.

Attention, certains supports sont divisés en zones séparées, ex : partitions d'un disque dur. Dans ce cas, chaque partition est considérée comme un volume.

Les volumes sont exploités par les périphériques : lecteur de disquette, de cdrom.

Certains volumes sont amovibles, d'autres non. Pour les premiers, on peut parfois choisir le lecteur à employer.

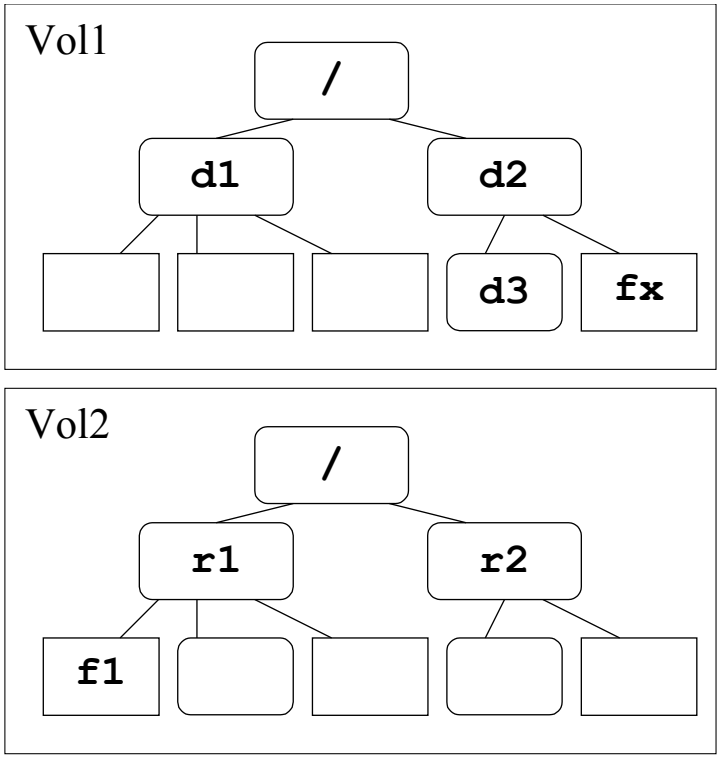
Montage

Le montage est la prise en compte par le système d'un volume mis dans un lecteur. Cela consiste à raccorder les fichiers qu'il contient à ceux déjà en place.

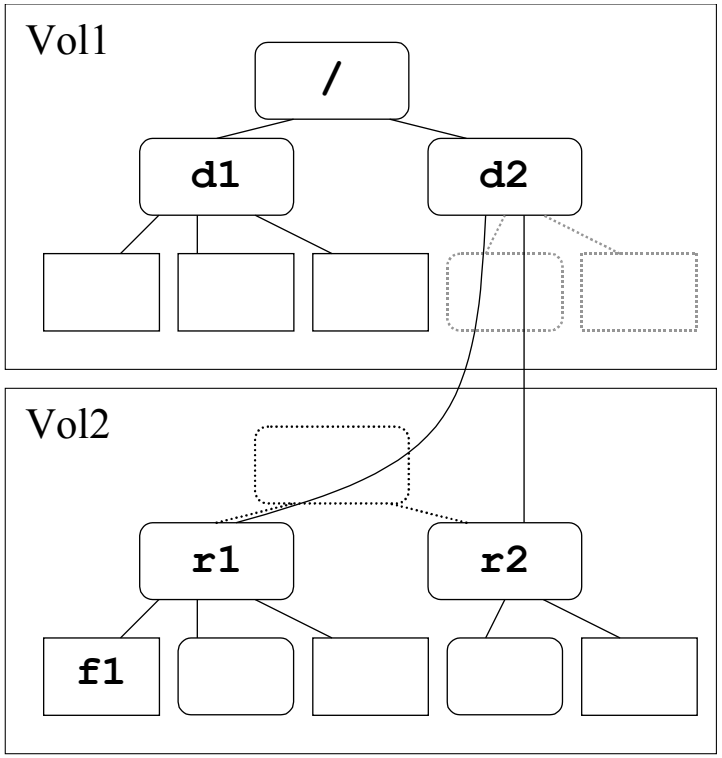
En général, un volume contient une seule arborescence de fichiers.

Sur Windows : une lettre est associée à chaque lecteur ; elle sert de racine à l'arborescence du volume. Ex : D:\

Sur Unix, les arborescences sont fusionnées :



Avant montage, vol1 est l'arbre initial



Après montage de vol2 sur /d2

Le montage, tant qu'il est en fait, masque les fichiers du *point de montage*. Ici, le contenu de d2 est inaccessible tant que vol2 est monté sur /d2.

L'accès aux fichiers du volume monté est transparent pour les utilisateurs.

Technique utilisable pour les volumes distants. Voir UE142 pour les détails.

- **`df`**
- **`mount`** et fichier **`/etc/fstab`** (voir UE142)

III - Protection des fichiers et répertoires

Protection = gestion des autorisations d'accès aux fichiers et répertoires par les utilisateurs

Sécurité = sûreté, fiabilité du système pour éviter les conséquences des pannes.

Accès aux fichiers et répertoires

Les actions des commandes sur les fichiers et répertoires sont décomposées en primitives :

- lecture, consultation, parcours du contenu Read
- écriture, modification------, changement du contenu Write
- exécution, utilisation du contenu eXecute

Ex : cp /tmp/toto ~/sys => X sur /, X sur tmp, R sur toto, X sur ~, X et W sur sys. C'est la pratique ou l'étude du programme source qui permet de le savoir.

Ce sont les instructions d'ouverture de fichier qui entraînent ces contrôles.

Ces demandes d'accès sont confrontés à un tableau qui indique quels utilisateurs peuvent les effectuer.

Catégories d'utilisateurs

Les utilisateurs sont rangés dans 3 catégories par rapport à un fichier :

- « U » quand on est le propriétaire (owner) du fichier
- « G » quand on est dans le même groupe que le fichier
- « O » dans tout autre (other) cas.

Les utilisateurs comme les fichiers sont rangés dans des groupes.

- **groups**
- **id**

En général, le groupe d'un fichier est celui de son propriétaire, mais ce n'est pas obligatoire : un fichier créé par untel du groupe projet peut appartenir à un autre groupe. C'est le cas à l'IUT : les fichiers créés par les étudiants appartiennent au groupe enseign.

Le groupe d'un fichier n'est pas nécessairement celui de son créateur. L'oublier fait commettre des erreurs de raisonnement. Voir § suivant.

Examen des permissions et appartenances

Algo suivi par le système :

1) Soit une commande Unix qui tente un accès sur un fichier ou répertoire : il faut vérifier qu'il y a X sur tous les répertoires du chemin d'accès et le ou les droits demandés sur le fichier en question :

2) Soit un objet fichier ou répertoire sur lequel un utilisateur veut faire un accès rwx : déterminer la catégorie ugo de cet utilisateur sur le fichier et consulter la liste des droits de cette catégorie pour savoir si c'est ok.

La liste des droits est associée à l'i-node du fichier (les liens physiques ont les mêmes droits). **`ls -l[d]`** affiche le tableau ugo*rwx des permissions :

```
-rwxr-x--x    1   nerzic   projet    0    09-08 15:45   toto
TUUUGGGOOO nbliens proprio groupe   taille  date         nom
```

Changement des permissions

- **`chmod -R quichgtquoi fichier`**

qui=[u][g][o][a] chgt = +, - ou = quoi=[r][w][x] [*autres...*][u][g][o]

L'option –R permet d'opérer les mêmes changements dans tout un arbre de fichiers.

- **`chmod code fichier`**

code = nombre en base 8 représentant les droits à mettre. Ex : 751 = rwxr-x--x

- **`umask anticode`**

Ce code permet de définir les droits qui sont attribués par défaut à un fichier nouvellement créé. L'anticode donne les bits qui doivent être enlevés. Ex : umask 022 fait que les nouveaux fichiers n'auront pas le droit w pour g et o.

Changements des appartenances

Sur certains systèmes ces commandes ne sont autorisées que pour l'administrateur : (option –r également disponible)

- **`chgrp nouvgroupe fichier[s]`**

- **`chown nouvproprio fichier[s]`**

Autres caractéristiques des fichiers liées aux droits

ls –l montre parfois d'autres indicateurs :

- s pour U sur un programme exécutable : appelé SetUID, il signifie qu'on va faire tourner le programme sous le nom (UID) du propriétaire du programme et non pas sous le nom de l'utilisateur du programme.

```
-r-s--x--x    1 root     root       15104 2002-03-14  passwd
```

- s pour G : appelé SetGID, même concept mais avec le groupe

- s pour G sur un répertoire : signifie que les objets créés dans ce répertoire hériteront du groupe du répertoire et non de celui du créateur = ce qu'untel crée dans ce répertoire n'est pas de son groupe mais de celui du répertoire.

```
drwxrws---    2 moi enseign   4096 09-05 13:56 moi
```

- t ou T pour un programme: (sticky bit) signifie que le programme en question doit impérativement rester en mémoire centrale, qu'il ne peut pas être mis sur disque en cas de swap par la MMU.

- t pour O sur un répertoire signifie que seul le propriétaire de ce répertoire ou le propriétaire des fichiers peuvent effacer les fichiers qui sont dedans, même si les droits w sont donnés à tout le monde.

Partie 3: Gestion des travaux

On s'intéresse aux commandes en cours d'exécution : les processus.

I - Lancement d'un programme

1) Shell

Le Shell est le logiciel qui :

- maintient la connexion d'un utilisateur avec son compte
- permet de saisir les commandes et les interpréter.

La syntaxe d'une commande est :

`nom options paramètres`

- les options s'écrivent en général –lettre(s) ou –mot. En général, on peut cumuler les options dans n'importe quel ordre ex : **`ls –ali = ls –lia = ls -l -i -a = ls --format=long --inode --all`**. Ne compter sur aucune uniformité entre les commandes, voir la doc systématiquement.

- Les paramètres = les noms des fichiers concernés.

L'interprétation d'une commande consiste à remplacer certains éléments par d'autres, sauf s'ils sont encadrés par des '...' ou marqués par \ :

- Les signes de mise en arrière-plan, les tubes et redirections sont retirés et modifient ensuite la manière de lancer la commande.
- Les rappels de commandes déjà tapées, ex : !p -> ping ...
- Les alias sur les commandes, ex : delete -> rm -fr
- Les variables sont remplacées par leur valeur, ex : cp $home/toto /tmp
- Certains signes, ex : rm ~/truc, le ~ est remplacé par /home/moi
- Les jokers par les noms des fichiers correspondant. Attention, un paramètre contenant un joker est remplacé par la liste de tous les fichiers correspondants. Ex : **`cc t*.c`** devient **`cc titi.c toto.c tutu.c`** sur une seule ligne et non pas trois appels à cc.

Il existe de nombreux shells tous équivalents dans leurs dernières versions. On peut distinguer deux grandes familles :

- la famille « archaïque » : Bourne Shell (sh, bsh, bash), Korn shell (ksh) très utilisés malgré une syntaxe peu lisible
- la famille « C » : C-Shell (csh, tcsh, zsh) qui s'inspire du langage C

Certains langages généralistes Perl, Python, CAML ... permettent de gérer les commandes Unix.

Les Shells offrent tous :

- des éléments de confort : complétion des noms de fichiers, historique des commandes, substitution variées, contrôle d'exécution
- des possibilités de programmation : structures de données (variables) et de contrôle (conditionnelles, boucles...). Les programmes sont appelés scripts dans le monde Unix et batch dans le monde Windows.
- des commandes internes, ex : alias, pushd, popd, echo... cf which

Le shell est souvent couplé à un autre logiciel lors de connexions distantes : xterm, telnet, ssh.

2) Exécution synchrone, asynchrone, différée

Le démarrage d'une commande crée un processus. C'est une entité qui matérialise l'exécution. Il existe trois modes de lancement :

- Synchrone, au premier plan : le shell attend la fin du processus avant de redonner la main à l'utilisateur. La notion de plan n'a rien à voir avec d'éventuelles fenêtres.

  ✍ **`commandecomplète`**

- Asynchrone, en arrière-plan : le shell reprend la main tout de suite, l'exécution se déroule en parallèle. On appelle « job » un tel processus.

  ✍ **`commandecomplète &`**

- Différée : l'exécution de la commande a lieu plus tard, à une heure choisie par l'utilisateur. Voir UE142 pour les détails.

  ✍ **`at`**

Certaines commandes se détachent automatiquement, ex : gvim.

3) Vie et mort d'un processus

a) Processus

Pour afficher la liste des processus :

- `ps -edfu`

PB : deux versions de ps coexistent parfois sur les systèmes. Les options ne sont pas les mêmes mais donnent des résultats similaires.

Elles affichent :

- PID : n° identifiant le processus
- PPID : PID du processus créateur de ce processus
- STAT : état du processus : R, S ou W, Z
- TIME : temps machine consacré au processus
- USER ou UID : nom ou uid du propriétaire
- CMD : commande et paramètres à l'origine du processus

Pour supprimer un processus :

- **kill -9 PID**

PB : deux versions de kill coexistent : celle du système /bin/kill et celle du shell !

b) Jobs

Les jobs sont des processus lancés en arrière-plan par un shell.

- **jobs**
- **fg %nj**
- **^Z, ^C**
- **bg %nj**
- **kill -9 %nj**

c) Code de retour

Les processus qui se terminent normalement effectuent l'instruction exit(nb); Le nombre nb est appelé code de retour et indique en général la raison de la fin du processus : 0 tout va bien, ≠ 0 pour une erreur. Ce

code de retour peut ensuite être examiné par le shell. Voir la doc en ligne pour les codes employés

4) Priorité des processus

« Plus un processus consomme de temps machine, moins on lui en donne ».

PRI : fréquence en Hz des activations possibles du processus

C : charge que représente le processus pour l'UC.

NI : nombre permettant de réduire la priorité du processus.

- ✍ **`nice -gentillesse commandecomplète`**

II - Redirection des entrées/sorties

Toute commande Unix est reliée au terminal texte (tty) par :

- une entrée dite standard **`stdin`** connectée au clavier (voie n°0)
- une sortie dite standard **`stdout`** connectée à l'écran (voie n°1)
- une sortie des erreurs **`stderr`** connectée à l'écran (voie n°2)

Ces voies de communication peuvent être redirigées vers des fichiers ou d'autres commandes. Cela permet :

- de récupérer dans un fichier les résultats d'une commande,
- de fournir à une commande des données situées dans un fichier,
- de ne pas reprogrammer les commandes selon qu'elles lisent/écrivent sur un terminal ou dans un fichier.

C'est possible car les terminaux sont considérés comme des fichiers : par défaut une commande est reliée avec /dev/pts/nn (selon les systèmes).

`tty`

1) Redirection vers un fichier

Pour rediriger l'entrée d'une commande (pour les commandes qui utilisent leur entrée standard) :

- **commandecomplète < fichier**

Ex : **mail dupond@rg.be < lettre**

Pour rediriger la sortie d'une commande :

- **commandecomplète > fichier**
- **commandecomplète >> fichier**

Ex : **ps -fu nerzic > liste ; ps -fu delhay >> liste**

Pour rediriger les erreurs d'une commande :

- **commandecomplète &> fichier**

Ex : **gcc plante.c &> erreurs**

2) Tubes

Ce dispositif se place entre deux processus et relie la sortie de la première à l'entrée de la deuxième. On peut enchaîner plusieurs tubes pour faire des traitements complexes.

- **commandecomplète1 | commandecomplète2**

Ex : **ps -ed | wc -l**

On utilise les tubes avec des filtres. Il s'agit de commandes réalisant un traitement utile sur des données de type texte.

III - Etude de quelques filtres usuels

1) Affichage

- **`more`** affiche page par page
- **`less`** affiche page par page en plus confortable

2) Comptage

- **`wc`** affiche le nombre de lignes, mots et caractères

  `… | wc -[l][w][c] | …`

- **`cat -n`** numérote les lignes
- **`uniq -c`** compte les lignes consécutives identiques

  `… | sort | uniq -c | …`

3) Tri - classement

- **`sort`** affiche les lignes triées dans l'ordre alphabétique

  `… | sort -n | …` ordre numérique

  `… | sort -r | …` ordre inverse

  `… | sort -t: -k 4 | …# Make sure we have our PIDDIR, even if it's on a tmpfs` sur les champs 4 et svts séparés par un :| sort | uniq –c |

4) Sélection de colonnes

- **`cut -d'£' -f`*`liste`*** garde les champs de la liste
- **`colrm col1 [col2]`** enlève les caractères indiqués

5) Sélection de lignes

- **`head`** affiche le début des données

… **| head -n nombre |** …

- **tail** affiche la fin des données
- **grep mot** affiche les lignes contenant un certain mot

… **| grep 'motif' |** … cherche le m mot otif dans les lignes

… **| grep -v 'motif' |** … refuse les lignes contenant motif

Le motif (pattern) est comparé (matching) à chaque ligne. Il s'agit d'une chaîne de caractères, contenant éventuellement des jokers (pour paramétrer la correspondance voulue).

jokers de correspondance (caractères variables)

J : correspond au simple caractère J quand *J* est un joker

. : correspond à un caractère quelconque (1 seul)

[*xyz*] : correpond à l'un des caractères énumérés

[*x-z*] : correspond à l'un des caractères de l'intervalle

[^*xyz*] : correpond à aucun des caractères énumérés

jokers de répétition (exposants appliqué aux jokers)

J+ : recherche les répétitions d'au moins une fois le joker

J* : recherche les répétitions d'un nombre quelconque de fois (y compris aucune)

J{n1,n2} : recherche les répétitions de n1 à n2 fois du joker

jokers de positionnement (où se trouve le motif % ligne)

^motif : le motif doit être en début de ligne ou de chaîne
motif$: le motif doit être en fin de ligne ou de chaîne

6) Divers mot

- **tr 'liste1' 'liste2'** traduit les caractères
- **tr -d 'liste'** élimine les caractères de la liste
- **tr -s 'liste'** réduit les suites des caractères de la liste

Partie 4: Programmation de scripts

Un script est un fichier texte contenant des commandes et qui, rendu exécutable, est considéré lui-même comme une nouvelle commande.

Les scripts sont interprétés : les commandes et instructions sont décodées et exécutées dans l'instant, sans compilation. Les erreurs de syntaxe et d'exécution sont détectées au dernier moment.

Le langage de programmation sous-jacent propose :

- des variables de type chaîne, tableaux de chaînes et nombres, en général, il n'est pas nécessaire de les déclarer, seule l'affectation suffit.
- des mécanismes de substitution : variables, chaînes actives...
- des structures de contrôle : tests, boucles...
- des mécanismes de liaison avec les commandes (processus).

I - Langages de scripts sur Unix

1) Bash

Le Shell **sh** est l'ancêtre de tous les langages de scripts sur Unix. Sa syntaxe date de l'époque où les analyseurs syntaxiques étaient inexistants. Les possibilités du langage ont été améliorées depuis au point de devenir exagérées par rapport à ce qu'on a vraiment besoin : on veut lancer et gérer des commandes unix qui manipulent des fichiers.

Un script bash débute par :

```
#!/bin/bash
# commentaires indiquant le rôle du script
# son mode d'emploi, auteur et date
```

Ensuite, on y met des commandes Unix identiques à celles qu'on entre au clavier : une par ligne ou séparées par ;

NB : un ; est équivalent à un retour à la ligne.

Ne pas appeler un script « test » ou « script » car ce sont des commandes Unix. Utiliser which en cas de doute.

Pour pouvoir l'exécuter : chmod u+x lescript, il devient comme une nouvelle commande.

Pour la mise au point, en général on peut mettre des options de traçage, telles que –veux, selon les versions qui affichent les lignes exécutées avec les substitutions effectuées.

a) Variables

Le Shell ne sait gérer que des chaînes de caractères. Les nombres sont représentés par la liste de leurs chiffres. Les variables sont stockées dans une table qui est spécifique à chaque processus shell.

- **`set`** affiche la liste des variables

Les chaînes

- **`nom=valeur`** affecte la valeur à la variable
- **`let nom=valeur`** idem, il existe aussi typeset qui précise le typage

- **`read nom`** lecture d'une variable au clavier

Prendre garde à bien coller le = au nom de la variable, sinon il y aurait confusion avec une improbable commande nom à laquelle on passe un paramètre nommé =.

Valeurs affectables à une variable

- **`mot`** la valeur est le mot tel quel
- **`chose1chose2`** la valeur est la concaténation des deux
- **`$nom`** la valeur de la variable nom
- **`${nom}autre`** la valeur de la variable nom suivi de autre
- **`'mot1 mot2'`** la valeur est la chaîne telle quelle
- **`"mot1 mot2"`** les variables de la chaîne sont remplacées
- **`` `commande` ``** la valeur = stdout de la commande

Les nombres

- ***`resu`*`=``expr` *`expression`*`` avec la commande expr

- **`resu=$[`*`expression`*`]`** la valeur de l'expression
- **`resu=$((`*`expression`*`))`** idem ?pourquoi 2 syntaxes?

Les tableaux

Un tableau en shell est une chaine contenant des mots séparés par des espaces. Le shell sait extraire l'un des mots à l'aide d'un indice.

Les tableaux sont utiles pour représenter des listes (de fichiers, de personnes...). Il existe des structures de contrôle pour appliquer des commandes à tous les éléments d'un tableau.

- **`tab=(liste…)`**
- **`${tab[*]}`** toute la liste
- **`${#tab[*]}`** le nombre d'éléments de la liste
- **`${tab[i]}`** le ième élément de la liste (i : 0 à $# - 1)
- **`${#tab[i]}`** le nombre de caractères du ième élément de la liste (piège!)

Il existe une méthode consistant à placer les mots d'un tableau dans les paramètres du script (voir §b) :

- **`set liste...`** met le 1er mot dans $1, le 2e dans $2…
- **`shift`** décale les mots de $i+1 dans $i en perdant $1

b) Variables spéciales ou prédéfinies

Les variables d'environnement

Leur valeur est transmise aux processus fils, mais « rien ne revient ». Ce sont comme des variables locales qui arrivent préinitialisées chez les processus fils.

- **`export nom=valeur`**

Les variables prédéfinies

En Bash, elles sont généralement en majuscules.

- **`$HOME`** le répertoire d'origine
- **`$PWD`** le répertoire courant
- **`$PATH`** tableau des chemins contenant les commandes

- `$?` le code de retour de la précédente commande
- `$#` le nombre de paramètres passés au script
- `$*` le tableau des paramètres passés au script
- `$1,$2…` les différents paramètres passés au script
- `$$` le PID de « ce » shell

c) Structures de contrôle

Sous-shell

Il s'agit d'un groupement de commandes à l'intérieur d'une commande.

- `(commande1 ; commande2 ; …)`

On peut le mettre en arrière-plan, effectuer une redirection globale. Attention, les transmissions de variables ne se font pas vers l'extérieur :

entre=oui ; (echo $entre ; sort=non) ; echo $sort

Conditionnelles

En Bash ou Korn, les tests sont effectués par des commandes et non par des expressions ; c'est le code de retour (0=vrai, autre=faux) qui sert de résultat de test.

```
if commande_test
then commande(s)_vrai
else commande(s)_faux
fi
```

Avec cela, on utilise :

- une commande classique qui renvoie un code de retour booléen :

```
if grep ^main $1 ; then echo $1 source C; fi
```

- la commande test :

```
if test -f $2 ; then echo $2 present ; fi
```

- la commande [:

```
if [ ! -f $1 ] ; then echo $1 absent ; fi
```

Ces deux dernières utilisent les opérateurs suivants :

- comparaison de chaînes : = != -z

- comparaison de nombres : -eq -ne -lt -le -ge -gt

- connecteurs logiques : && || ! ()

Voici les conditionnelles généralisées, elles permettent d'employer des motifs (*, ?, [] du shell et non de grep)

```
case valeur in
        motif1) commande1 ;;
        motif2) commande2 ;;
      .........
        *) commandedef ;;
   esac
```

Boucles

Voici la boucle itérative non bornée classique :

```
while commande
```

```
        do commande(s)
    done
```

Pour parcourir toute une énumération :

- `for elem in liste`

```
        do commande(s)
    done
```

Les boucles peuvent être interrompues par **`break`** ou forcées à passer à l'itération suivante par **`continue`**.

Savoir-faire

On emploie les boucles bornées sur, principalement, quatre genres de données :

Une liste de mots :

```
for ext in txt c pas
do
    cp *.$ext ~/sauv
```

```
done
```

Une liste de fichiers :

```
for fich in *.c
do
    cat entete $fich > $fich.tmp
    mv $fich.tmp $fich
done
```

La liste des paramètres fournis au script :

```
for param in $*
do
    echo "voici $param, rien d'autre." > $param
done
```

Les mots produits par une commande :

```
for uti in `who | colrm 9`
do
    mail $uti < lettre
done
```

On emploie les boucles non bornées sur des données dont on ne connait pas le nombre à priori :

```
echo veuillez saisir des noms de fichier
while read nom
do
    touch nom
done
```

On peut employer les sous-shells :

```
who | colrm 9 | ( while read uti ; do ...$uti...; done)
```

d) Fonctions

Définir une fonction est aisé, on a le choix entre deux syntaxes équivalentes :

- `function nom { commande(s) ; return valeur ; }`
- `nom() { commande(s) ; }`

Pour l'appeler : `nom arguments…` Les arguments sont mis dans $1, $2... à l'intérieur de la fonction.

Le résultat de la fonction est mis dans le code de retour : $?

```
# définition de la fonction
function essai { echo je reçois : $* ; return $# ; }

# appel de la fonction
essai salut ca va
echo $?
```

e) Exemples commentés

Voici quelques exemples de scripts bash.

Entête d'un script Bash

```
#!/bin/bash -f
# ce script crée un répertoire et sauve les fichiers dedans
# usage : sauve fichier…
```

Tester la présence de paramètres

```
if [ $# -eq 0 ] ; then
    echo usage : $0 fichier…
    exit 1       # erreur fatale
fi
```

Récupérer l'affichage d'une commande dans une variable

```
jour=~/sav/`date +%d-%m-%Y`      # voir man date
```

Tester si un nom complet est celui d'un répertoire

```
if [ ! -d $jour ] ; then      #  s'il  n'existe  pas,  on  le
crée
```

Tester le bon déroulement d'une commande

```
    if mkdir $jour ; then : ; else     # : instruction vide
       echo erreur, le répertoire $jour ne peut être créé
       exit 2      # erreur fatale
    fi
fi
```

Récupérer l'affichage d'une commande dans une variable (bis)

```
nombre=`ls $jour | wc -l`     # nb de fichiers déjà mis
```

Faire une boucle sur les paramètres

```
for param    # in $*    optionnel
```

Tester si une chose est un fichier

```
    do if test ! -f $param ; then
       echo fichier $param non trouvé
```

```
        continue   # on retourne au foreach pour le suivant
    fi
    cp $param $jour
    echo fichier $param copié dans $jour
    nombre=$((nombre + 1))
done
echo $nombre fichiers copiés
```

Voici un autre exemple : un script qui affiche le nom et la taille des fichiers ayant une certaine extension (ex : .pas, .c)

Lecture d'informations au clavier

```
echo -n "entrez l'extension recherchée : "
read extension
```

Parcours de la liste des fichiers

```
for fich in *.$extension ; do
    echo -n $fich ' : '
    wc -c < $fich
done
```

Autre exemple, sur les tableaux.

Mots d'une commande

```
infos=(`ls -l toto`)
echo proprio = ${infos[2]}, taille = ${infos[4]}
```

Autre exemple

```
cd /home
for uti in * ; do
    taille=`du -ks $uti`
    if [ $taille -gt 20000] ; then
       mail $uti <<lettre
Votre compte dépasse 20Mo, veuillez enlever des fichiers.
Merci
lettre
    fi
done
```

f) Fichiers de configuration

~/.profile contient les instructions exécutées par chaque nouvel interpréteur shell, sauf si l'option –f est spécifiée dans le script.

g) Divers utile

Les commandes suivantes peuvent être utiles pour manipuler des chemins :

- **basename, dirname**

La commande suivante, par exemple, extrait les chiffres d'une chaîne :

- **expr $chaine : '[a-z]*\([0-9]*\).*'**

La redirection suivante permet de prendre des informations dans le script :

- **commande <<DATA**

  ...lignes…

  DATA

NB: le mot DATA doit se trouver au tout début de sa ligne.

2) AWK

Ce langage de programmation puissant et lisible permet de traiter facilement des fichiers de données : awk automatise le parcours du fichier, il ne reste qu'à spécifier ce qu'on fait des données lues.

Consulter la documentation pour le détail des possibilités (énormes).

a) Principes de fonctionnement

Le traitement d'un fichier suit généralement ce schéma :

initialisation
ouvrir le fichier
TantQue non(fin de fichier) Faire
    lire une ligne de données
    traiter la ligne lue
FinTantQue
fermer le fichier
terminaison

Par exemple, si on veut compter le nombre de lignes d'un fichier :

```
initialisation :    nombre = 0
traitement :        nombre = nombre + 1
terminaison :       afficher nombre
```

Le langage awk se charge de tout ce qui est en noir et on ne doit programmer que ce qui est en rouge. On place ces éléments dans un fichier appelé « script awk ». Attention, le script awk n'est pas exécutable en tant que tel :

- `awk -f script  données`
- `cat  données | awk -f script`

Algorithme général de Awk :

Le script awk contient des lignes d'*instructions* structurées ainsi :

```
    condition { actions }
```

Awk parcourt chaque ligne du fichier, cherche les instructions dont la condition est vraie et exécute les actions correspondantes.

```
BEGIN { nombre = 0 }
{ nombre = nombre + 1 }
END { print "il y a " nombre " lignes dans ce fichier" }
```

Lorsque la condition est absente, elle est considérée comme toujours vraie. Les conditions BEGIN et END sont optionnelles.

b) Quelques types de conditions

- **BEGIN** condition spéciale vraie seulement au début du traitement, on l'utilise pour initialiser le script. Attention à ne pas confondre la condition BEGIN avec la notion de bloc du langage pascal.
- **END** condition spéciale, vraie à la fin des données, on l'utilise pour conclure le script (bilan...). END est comparable à EOF en C et Pascal.
- **(condition)** une condition booléenne construite comme en C-Shell ou en C
- ***/motif/*** c'est une sorte de booléen vrai si la ligne courante contient le motif

c) Motifs

Comme avec grep, les motifs sont cherchés parmi les lignes du fichier à traiter. Les jokers sont les suivants :

jokers de correspondance (caractères variables)

`\J` : correspond au simple caractère J quand J est un joker

`.` : correspond à un caractère quelconque (1 seul)

`[xyz]` : correpond à l'un des caractères énumérés

`[x-z]` : correspond à l'un des caractères de l'intervalle

`[^xyz]` : correpond à un caractère autre que ceux énumérés

jokers de répétition (exposants appliqué aux jokers)

`J+` : répète au moins une fois le joker

`J*` : répète le joker un nombre quelconque de fois (y compris aucune)

`J{n1,n2}` : répète n1 à n2 fois le joker

jokers de positionnement (où se trouve le motif % ligne)

`^motif` : le motif doit être en début de ligne ou de chaîne
`motif$` : le motif doit être en fin de ligne ou de chaîne

d) Actions

Les actions (affectations, procédures...) à faire quand la condition est vraie sont placées entre { } (signification identique à begin-end du pascal). En voici qques :

- **`print expression`**

affiche l'expression (variables, chaînes...)

- **`system(`*`commande`*`)`**

exécute la commande unix

- **`exit`**

arrête l'exécution en cours et saute directement aux actions END

e) Variables et types

Les types *nombre*, *chaîne* et *tableau* sont confondus. Le type est choisi dynamiquement en fonction de la dernière affectation.

✍ **`variable = valeur`**

La valeur peut être une chaîne encadrée par deux guillemets ou une expression numérique ou alphanumérique.

NB: un mot écrit sans guillemets est une variable (pas de $)

Les tableaux sont entièrement dynamiques, les indices peuvent être des nombres ou des chaînes.

Ex :
```
tab[i] = 4
compte[100] = "guest"
uid["root"] = 1
prenom["Tournesol"] = "Tryphon"
```

Il existe des variables prédéfinies pour permettre un traitement plus facile de la ligne courante :

- **NR** numéro de la ligne courante
- **$0** le contenu de la ligne courante
- **FS** séparateur de champ, c'est l'espace par défaut
- **NF** nombre de champs (mots) dans la ligne courante
- **$1** le premier champ de la ligne en cours, $2 le deuxième... leur séparateur est **FS**

```
# script qui numérote les lignes d'un fichier
{ print NR " " $0 }
```

Placer cette ligne dans un script **numérote.awk** et lancer l'exécution par **awk -f numérote.awk fichier**

```
# script awk qui compte les lignes d'un fichier
BEGIN { nb = 0 }
{ nb = nb + 1 }
END { print nb }
```

f) Opérateurs et fonctions

Les opérateurs et fonctions mathématiques sont ceux du langage C-Shell et du langage C : **+ - * / %** ...

Voici quelques fonctions sur les chaînes parmi l'immense collection disponible :

- ***chaîne1 chaîne2*** deux chaînes juxtaposées sont concaténées
- **match(*chaîne, motif*)** renvoie la position du motif dans la chaîne (1 à ...) ou 0 s'il est absent
- **sub(*rech,remp,chaîne*)** remplace rech par remp
- **length(*chaîne*)** longueur de la chaîne
- **substr(*chaîne,déb,fin*)** extrait un morceau de la chaîne délimité par les indices

g) Structures de contrôle

- Les instructions et structures de contrôle awk sont celles du langage C — elles sont légèrement différentes du C-Shell :

```
if (condition) {
   instructions…
} else {
   instructions…
}
```

Les conditions sont exprimées comme en C-Shell, ou comme ci-dessous, pour tester si un mot se trouve dans un tableau :

```
if (variable in tableau) { instructions }
```

- Pour boucler sur les indices d'un tableau :

```
for (variable in tableau) { instructions }
```

Les instructions sont répétées sur chaque élément du tableau.

h) Exemple de synthèse

On désire compter le nombre de processus de chaque utilisateur et leur charge induite à partir de la liste produite par la commande ps –edf :

```
USER        PID  PPID   C DATE              TIME         CMD
root         11     1   0 Sep19 ?          00:00:00 [khubd]
root        559     1   0 Sep19 ?          00:00:09 syslogd -m 0
daemon      578     1   0 Sep19 ?          00:00:00 portmap
rpcuser     593     1   0 Sep19 ?          00:00:00 rpc.statd
root        763     1   0 Sep19 ?          00:00:04 /usr/sbin/automount
daemon      778     1   0 Sep19 ?          00:00:00 /usr/sbin/atd
```

Voici le script awk :

```
# compte les processus de chaque utilisateur
(NR>1) {      # sauter la 1e ligne (titre)
    # $1 = colonne USER, $4 = colonne C
    if ($1 in NbProcs) {
       NbProcs[$1] = NbProcs[$1] + 1
       Charge[$1] = Charge[$1] + $4
    } else {
       NbProcs[$1] = 1
       Charge[$1] = $4
    }
}
END {
    print "Bilan :"
    for (uti in NbProcs) {
       print uti,":",NbProcs[uti]," Ctot=",Charge[uti]
    }
}
```

Lancer l'exécution par **`ps -edf | awk -f nbprocs.awk`**

II - Langages de scripts sur Windows

1) Batch

C'est le système MS-DOS sous-jacent dans Windows qui a introduit ces scripts. On peut regretter un grand manque de souplesse dans la programmation à cause des imperfections et de l'ancienneté du système : tubes et redirections limités, structures de contrôle minables.

Le nom d'un script doit se terminer par **`.bat`**. Il suffit de taper le nom de base pour lancer l'exécution. Les instructions du script sont affichées sauf si le mode sans echo est demandé : instruction **`@echo off`**

a) Commandes MS-DOS

Voici d'abord quelques commandes ms-dos utiles pour écrire des scripts :

- **`help`**
- **`cd, dir/w`**

- **copy, xcopy, ren, del**
- **md, rd**
- **find**

b) Paramètres et variables

Les paramètres des scripts sont nommés **%1**, **%2**...

Les chaînes sont délimitées par "..."

c) Structures de contrôle

- ***:label***
- **goto *label***
- **if *test* goto *label***

Les goto peuvent être calculés (goto cas%n avec des labels cas1 cas2...)

Les *tests* sont réduits à ceux-ci :

- errorlevel nb
- exist fichier
- v1 == v2
- not *test*

✍ **`for %%variable in (liste) do`**

La liste peut être constituée de noms de fichiers à l'aide d'un joker.

2) JScript

C'est un excellent langage similaire à Javascript (lui-même inspiré par Python) qui permet de développer des outils de manipulation des fichiers, des comptes et du réseau. Microsoft propose aussi VBScript qui a le même champ d'application mais une syntaxe différente.

Pour ces développements, on dispose :

- d'un langage de programmation,
- d'un accès au système d'exploitation sous forme d'objets (au sens POO),
- d'un interprète qui fait le lien : wscript.exe ou cscript.exe

Le cours ne pouvant tout présenter, se référer à la biblio pour les détails :

http://www.microsoft.com/technet/scriptcenter/default.mspx

Ce lien contient de nombreux exemples et tutoriels.

a) Quelques éléments de syntaxe du langage

Programmes

Nommer les fichiers avec l'extension .js ; elle est reconnue par le système et associée à l'interpréteur wscript.exe. Un double-clic lance l'exécution du script.

```
/* mon premier jscript */
WScript.Echo("mon premier script jscript !");
```

Il n'y a pas de notion de procédure principale, l'exécution du programme commence à sa première ligne, exécute les instructions rencontrées en sautant les définitions de fonctions.

Instructions

Comme en C ou en java, la deuxième syntaxe est nommée bloc :

- `instruction ;`
- `{ instruction(s)… }`

Commentaire : `//…` ou `/*…*/`

Structures de contrôle

Les mêmes qu'en C, on peut mettre un bloc à la place des instructions :

- `if (test) instruction else instruction ;`
- `for (init ; test ; prog) instruction ;`
- `while (test) instruction ;`

Les conditions sont écrites exactement comme en C et Java.

Quelques unes en plus, dont :

- `try instruction catch (exception) instruction ;`
- `throw exception ;`

Fonctions

Un mélange de C et de pascal :

- `function nom([param,…]) bloc`

- `return expression ;`

On les définit où on veut, de préférence avant leur appel.

Variables et types

La définition d'une variable se fait presque comme en pascal :

- `var nom = valeur[, …] ;`

Les types de données sont nombreux et sont souvent des classes d'objets : nombres, chaînes, tableaux.

Objets, classes, propriétés et méthodes prédéfinies

- `var nom = new type([valeur]) ;`

Ceci crée une nouvelle instance du type indiqué (constructeur d'objets). Selon la classe on dispose de méthodes (procédures ou fonctions d'instance) et propriétés (variables d'instance).

La notation objet.*info* désigne une propriété de l'objet. Les propriétés peuvent être créées dynamiquement : par une simple affectation.

La notation objet.*action*() fait appeler une méthode de l'objet.

Chaînes : String()

La propriété chaine.length donne la longueur de la chaîne. C'est une propriété, ne pas écrire chaine.length()

La méthode chaine.substr(pos,len) extrait une sous-chaîne, chaine.indexof(chaine2) cherche la position de chaine2 dans la chaîne.

Tableaux : Array()

La propriété tab.length donne le nombre d'éléments du tableau

Il y a de nombreuses méthodes intéressantes : concat, join, pop, push, slice, sort… Consulter la documentation

b) Quelques objets utiles

Les interpréteurs Jscript et VBScript s'appuient sur des « objets » pour manipuler le système d'exploitation. Un objet est une représentation d'un composant du système. Par exemple l'arbre des fichiers est représenté par un objet FileSystemObject. Les objets ont des propriétés et des fonctions appelées méthodes

Objet WScript

Objet de base pour gérer l'environnement d'exécution du script. Cet objet existe déjà dans l'environnement.

La méthode Echo(chaine) affiche une petite fenêtre sur l'écran.

```
WScript.Echo("bienvenue");
```

La méthode Quit([code]) arrête le script, renvoie le code d'erreur s'il est présent.

```
WScript.Quit(1);
```

Classe FileSystemObject

Cette classe représente le système de fichiers. Ses méthodes et propriétés permettent de gérer les fichiers, répertoires et disques.

Tout commence par la création d'un objet de cette classe :

```
var FSO = WScript.Create("Scripting.FileSystemObject");
```

FSO représente alors le système de fichiers : volumes, répertoires et fichiers.

Gestion des fichiers et répertoires

FSO.GetFile(nomcomplet) et FSO.GetFolder(nomcomplet) fournissent chacun un type d'objet dont voici quelques propriétés utiles :

Attributes (entier codant les caractérisques du fichier : 0 normal, 1 readonly, 2 caché, 4 system), DateCreated, DateLastAccessed, DateLastModified, Size, Type

Pour les répertoires, la propriété SubFolders renvoie la liste des sous-répertoires.

Et quelques méthodes :

Copy(destination), Delete(force)

Exemple :

```
var fich = FSO.GetFile("toto.txt");
if (fich.Size < 10) fich.Delete(true);
```

Liste des fichiers et répertoires

C'est malheureusement et anormalement compliqué en Jscript : la propriété Files d'un répertoire donne une liste de noms de fichiers mais

il n'existe pas de structure de contrôle capable de la parcourir. Il faut l'associer à un énumérateur :

```
var dir = FSO.GetFolder("c:\\temp");
var fichiers = new Enumerator(dir.Files);
for ( ; !fichiers.atEnd() ; fichiers.moveNext()) {
    // obtenir le nom du fichier courant
    var fichier = fichiers.item() ;
    // traiter ce fichier
    ...
}
```

VBScript offre une structure de parcours des listes bien plus simple.

Lecture et écriture de fichiers

Il existe de nombreuses méthodes liées aux fichiers, voici un exemple :

```
var fich = FSO.CreateTextFile("c:\\temp\\essai.txt",true);
fich.WriteLine("Essai de création de fichier");
fich.Close();
fich = FSO.OpenTextFile("c:\\temp\\essai.txt",1);
ligne = fich.ReadLine();
```

```
fich.Close();
```

Quelques exemples de parcours de fichiers

L'un des traitements les plus fréquents qu'on ait à faire sur les fichiers est de parcourir un répertoire et ses sous-réperrtoires et traiter chacun de ses fichiers. On emploie une fonction récursive.

```
/* parcours récursif d'un répertoire */

// initialisation
var FSO =
      WScript.CreateObject("Scripting.FileSystemObject");

function TraiterFichier(nomcomplet)
// cette fonction est appelée avec un nom complet
// de fichier par Parcourir à l'occasion du parcours
// d'un arbre de fichiers
{
    // faire quelque chose avec chemin et fichier
    // exemple : afficher sa date de modification
```

```
    var fichier = FSO.GetFile(nomcomplet);
    var nom = fichier.Name;          // obtenir son nom seul
    var chemin = fichier.ParentFolder; // et son chemin
    var date = fichier.DateLastModified;  // et sa date
    Wscript.Echo("chemin="+chemin+
       " nom="+nom+" date="+date);
}

function ParcourirRepertoire(rep)
// on appelle cette fonction avec un chemin, p.ex: d:\\test
// elle parcourt ce répertoire et appelle la fonction
// TraiterFichier sur chacun des fichiers trouvés dedans
// puis s'appelle elle-même sur chacun des sous-répertoires
// rencontrés.
{
    var dir = FSO.GetFolder(rep);

    // d'abord traiter les fichiers de cet endroit
    var liste = new Enumerator(dir.files);
    for (; !liste.atEnd(); liste.moveNext()) {
```

```
        TraiterFichier(liste.item())
    }

    // ensuite, parcourir les sous-répertoires
    var liste = new Enumerator(dir.subfolders);
    for (; !liste.atEnd(); liste.moveNext()) {
        ParcourirRepertoire(liste.item())
    }
}



//// Programme principal ////

ParcourirRepertoire("d:\\test");
```