Le système d'exploitation

UNIX

I/ Présentation

1. Introduction

Unix est un système d'exploitation moderne, complet et efficace, disponible sur la plupart des ordinateurs vendus, du PC au super calculateur Cray. C'est pourtant un système ancien, puisque son histoire remonte à la fin des années 60. Son architecture ouverte et sa grande diffusion dans les centres de recherches et les universités lui ont permis d'évoluer en intégrant de nombreuses améliorations.

Aujourd'hui, les principaux domaines d'application d'Unix sont :

- serveurs (web, applications, stockage) ;
- grand public (variante d'Apple Mac OSX, et Linux) ;
- systèmes embarqués (bornes wifi, routeurs...)
- recherche scientifique.

Le système GNU/Linux est une version "libre" d'Unix, très répandue en particulier sur les PC (voir plus loin). Les commandes décrites ci-dessous sont assez standard, et devraient fonctionner sur la plupart des versions d'Unix.

2. Historique et versions d'UNIX

Le système Unix a été écrit en 1969 par Ken Thompson dans les laboratoires de Bell, en s'inspirant du système MULTICS. Il a très vite été réécrit en langage C, langage développé dans ce but par Dennis Ritchie.

L'écriture du logiciel en langage C, et non en assembleur comme c'était alors l'usage, était une grande innovation qui permit de porter le système sur une grande variété de machines. Bien entendu, une portion (environ 5%) du code dépend de l'architecture matérielle de l'ordinateur : gestion des entrées/sorties, interruptions, gestion mémoire... Mais le gros du système reste commun d'une machine à l'autre, ce qui simplifie le travail et garantit un comportement identique (portabilité). Il est ainsi infiniment plus simple de porter un logiciel d'une variante d'Unix à une autre (par exemple de Sun/Solaris à Linux) que d'un PC sous Windows à un autre système.

Depuis la fin des années 70, il existe deux grandes familles d'Unix. D'une part une version développée essentiellement par l'université de Berkeley (Californie), et nommée UNIX BSD, d'autre part l'Unix Système V commercialisé par ATT. De nombreuses autres versions ont vu le jour, qui sont le plus souvent une adaptation de BSD ou Système V par un fabriquant particulier :

- AIX IBM, Bull (stations de travail, mainframes) ;
- HP/UX Hewlett-Packard (stations) ;
- SCO Unix SCO (PC) ;
- OSF/1 DEC;
- Solaris Sun MicroSystems (stations Sun et PC) ;
- GNU/Linux Logiciel libre (et gratuit).

Notons que la plupart de ces versions tendent à disparaitre, au profit du système libre GNU/Linux.

Ces différentes versions possèdent quelques incompatibilités. Pour y remédier, une norme a été proposée par l'IEEE, le système POSIX. La plupart des versions modernes d'Unix sont des sur-ensembles de POSIX ; un programme écrit en respectant POSIX sera donc portable sur toutes ces versions.

Les premières versions d'Unix ne permettaient que le travail sur des terminaux alphanumériques (il n'en existait pas d'autres à l'époque). Un grand pas en avant a été fait avec le développement au MIT du système X Windows (X11). Ce système permet le multifenêtrage sur écran graphique et le développement d'interfaces utilisateurs sophistiquées et "conviviales" (inspirées du Macintosh). De nombreux environnements graphiques sont maintenant disponibles : Motif, KDE, Gnome, etc.

Linux est une version libre d'Unix (le code source du système est disponible gratuitement et redistribuable) qui connait actuellement un grand succès, tant chez les utilisateurs particuliers (en tant qu'alternative à Windows) que sur pour les serveurs Internet/Intranet. Linux est diffusé par différentes

sociétés ou organisations, sous formes de distributions qui utilisent le même noyau (ou presque) et organisent de diverses façons le système (packages, mises à jour, etc). Les distributions les plus répandues sont Red Hat (et sa variante Fedora), Suse (achetée par l'entreprise Novell), Debian (totalement libre, ainsi que sa variante Ubuntu), Slackware et Mandriva (ex Mandrake) et s'adressent chacune à différents types d'utilisateurs.

3. Architecture générale du système

Unix est un système d'exploitation multitâche et multi-utilisateurs. Le fonctionnement multitâche est assuré par un mécanisme préemptif : le système interrompt autoritairement la tâche en cours d'exécution pour passer la main à la suivante ; ceci évite tout risque de blocage du système à la suite d'une erreur survenue dans un programme utilisateur. La cohabitation simultanée de plusieurs utilisateurs est rendue possible par un mécanisme de droits d'accès s'appliquant à toutes les ressources gérées par le système (processus, fichiers, périphériques, etc.).

a) Le noyau Unix

Le noyau est le programme qui assure la gestion de la mémoire, le partage du processeur entre les différentes tâches à exécuter et les entrées/sorties de bas niveau. Il est lancé au démarrage du système (le boot) et s'exécute jusqu'à son arrêt. C'est un programme relativement petit, qui est chargé en mémoire principale.

Le rôle principal du noyau est d'assurer une bonne répartition des ressources de l'ordinateur (mémoire, processeur(s), espace disque, imprimante(s), accès réseaux) sans intervention des utilisateurs. Il s'exécute en mode superviseur, c'est à dire qu'il a accès à toutes les fonctionnalités de la machine : accès à toute la mémoire, et à tous les disques connectés, manipulations des interruptions, etc.

Tous les autres programmes qui s'exécutent sur la machine fonctionnent en mode utilisateur : il leur est interdit d'accéder directement au matériel et d'utiliser certaines instructions. Chaque programme utilisateur n'a ainsi accès qu'à une certaine partie de la mémoire principale, et il lui est impossible de lire ou écrire les zones mémoires attribuées aux autres programmes.

Lorsque l'un de ces programmes désire accéder à une ressource gérée par le noyau, par exemple pour effectuer une opération d'entrée/sortie, il exécute un appel système. Le noyau exécute alors la fonction correspondante, après avoir vérifié que le programme appelant est autorisé à la réaliser.

b) Les processus

Unix est un système multi-tâches, ce qui signifie que plusieurs programmes peuvent s'exécuter en même temps sur la même machine. Comme on ne dispose en général que d'un processeur, à un instant donné un seul programme peut s'exécuter. Le noyau va donc découper le temps en tranches (quelques millièmes de secondes) et attribuer chaque tranche à un programme.

On parle de système en temps partagé. Du point de vue des programmes, tout se passe comme si l'on avait une exécution réellement en parallèle. L'utilisateur voit s'exécuter ses programmes en même temps, mais d'autant plus lentement qu'ils sont nombreux.

On appelle processus un programme en cours d'exécution. A un instant donné, un processus peut être dans l'un des états suivants :

actif : le processus s'exécute sur un processeur (il n'y a donc qu'un seul processus actif en même temps sur une machine monoprocesseur) ;

prêt : le processus peut devenir actif dès que le processeur lui sera attribué par le système ;

bloqué : le processus a besoin d'une ressource pour continuer (attente d'entrée/sortie par exemple). Le blocage ne peut avoir lieu qu'à la suite d'un appel système. Un processus bloqué ne consomme pas de temps processeur; il peut y en avoir beaucoup sans pénaliser les performances du système.

Remarque : le passage de l'état actif à l'état prêt (interruption) est déclenché par le noyau lorsque la tranche de temps allouée au processus s'est écoulée. Concrètement, le noyau programme dans ce but une interruption matérielle. Ceci implique que toute section d'un programme utilisateur peut se voir interrompue n'importe où ; les processus n'ont bien sûr pas accès aux instructions de masquage d'interruptions. Toute opération critique (que l'on ne peut interrompre) devra donc être réalisée dans le noyau, un processus y accédant via un appel système.

c) Gestion de la mémoire

Le système Unix fonctionne en mémoire virtuelle paginée. Ceci permet de faire fonctionner des processus demandant une quantité d'espace mémoire supérieure à la mémoire physique installée.

Lorsqu'un processus demande l'allocation d'une page de mémoire et qu'il n'y en a pas de disponible en mémoire centrale, le noyau traite un défaut de page. Il choisit une page (qui n'a pas été utilisé depuis longtemps) et l'écrit sur une partition spéciale du disque dur. La place libérée est alors attribuée au processus demandeur.

Ce mécanisme demande la réservation d'une (ou plusieurs) partition spéciale sur l'un des disques durs, nommée partition de swap. La mémoire disponible pour les processus est donnée par la somme de la taille de mémoire physique (RAM) et des partitions de swap. Bien entendu, les performances du système se dégradent lorsque la fréquence des défauts de page augmente; dans ce cas, il faut augmenter la mémoire physique.

Sur un système typique, la partition de swap est deux à trois fois plus grande que la mémoire centrale (exemple : PC avec 32Mo de RAM, partition de swap de 64Mo).

d) Les outils UNIX

Unix est livré avec un grand nombre de programmes utilitaires. Ces programmes sont très divers, mais surtout orientés vers le traitement de fichiers de textes et le développement de logiciels. Tout système Unix inclut normalement un compilateur C (certains vendeurs tendent à abandonner cet usage).
Les utilitaires les plus importants sont les suivants :

- Interpréteurs de commandes (nommés Shells), permettant l'accès d'un utilisateur au système. Les Shells sont assez sophistiqués et s'apparentent à de véritables langages de programmation interprétés.
- Commandes de manipulation de fichiers ;
- Commandes de gestion des processus ;
- éditeurs de texte ;
- Outils de développement : compilateurs, débuggeurs, analyseurs lexicaux et syntaxiques, etc.

Nous détaillerons certains de ces outils par la suite.

4. Les utilisateurs, l'accès au système

Chaque utilisateur humain du système doit disposer d'un compte protégé par un mot de passe. Lorsque l'on se connecte à la machine, on doit fournir son nom et son mot de passe. Le nom est un pseudonyme attribué une fois pour toute à un utilisateur par l'administrateur du site. Le mot de passe peut être modifié par l'utilisateur aussi souvent qu'il le désire.

Avec la généralisation des interfaces graphiques, l'accès à un système sous Unix s'est diversifié et (théoriquement) simplifié. La procédure d'entrée d'un utilisateur dans le système se nomme le login. La sortie est donc le logout.

Lors du login, l'utilisateur devra toujours fournir son nom d'utilisateur et un mot de passe. Après vérification du mot de passe, le système lance un interpréteur de commande (Shell).

Chaque utilisateur dispose de ses propres fichiers, dont il peut autoriser ou non l'accès aux autres utilisateurs. Il dispose d'un certain nombre de droits (accès à certains périphériques, etc).

Il peut lancer l'exécution de processus (le nombre maximal de processus par utilisateur peut être limité sur certains sites). Les processus lancés par un utilisateur ont les mêmes droits d'accès que lui.

II/ Les fichiers Unix

Le système de fichier est géré par le noyau. Les fichiers Unix correspondent soit à des fichiers de données sur disque dur, soit à des répertoires, soit encore à des fichiers spéciaux permettant la gestion de certaines ressources du système (par exemple, lorsqu'un périphérique d'entrées/sorties permet des opérations comme ouverture, écriture, lecture (terminal, imprimante), on y accède généralement par un fichier spécial (device)).

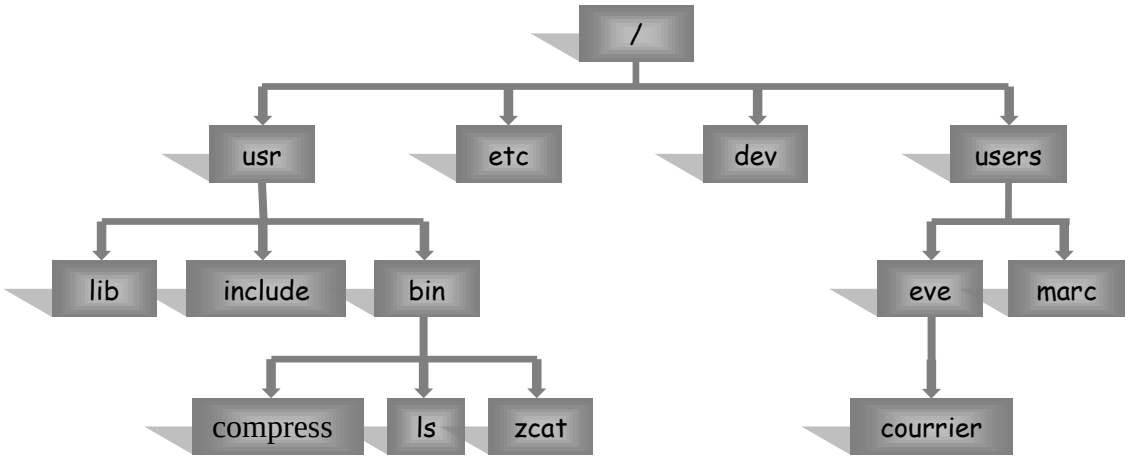
1. Répertoires

Les fichiers sont organisés en répertoires et sous-répertoires, formant une arborescence.

Dans chaque répertoire, on trouve au moins deux fichiers, nommés . (point) et .. (point point). Le premier (.) permet de référencer le répertoire lui même, et le second (..) d'accéder au répertoire parent (du dessus).

A chaque instant, toute tâche possède un répertoire courant, ou répertoire de travail. La commande pwd affiche ce répertoire. La commande cd permet de changer le répertoire courant.

Exemple d'arborescence de fichiers sous Unix.



La racine de l'arbre est le répertoire / (en haut). Ce répertoire contient ici 4 sous-répertoires.

a) Chemins absolus et relatifs

Pour désigner un fichier quelconque, on peut utiliser soit un chemin absolu, soit un chemin relatif.

Un chemin absolu spécifie la suite des répertoires à traverser en partant de la racine, séparés par des caractères / (et non \ comme sous DOS). Par exemple, le chemin */usr/bin/compress* désigne le fichier compress, qui se trouve dans le répertoire bin, lui-même dans le répertoire usr de la racine. Le premier caractère / indique qu'il s'agit d'un chemin absolu.

Il est souvent pratique d'utiliser un chemin relatif, à partir du répertoire courant. Par exemple, si l'on travaille dans le répertoire eve, on peut accéder au répertoire marc en spécifiant le chemin *../marc*.

Du même endroit, on peut accéder au fichier compress via le chemin *../../usr/bin/compress* (dans ce cas précis, il est plus simple d'utiliser le chemin absolu).

Tout chemin qui ne commence pas par un caractère / (prononcé slash) est interprété comme un chemin relatif au répertoire courant. On peut ainsi accéder aux fichiers du répertoire courant en donnant simplement leur nom.

b) Répertoire de connexion

A chaque utilisateur connu du système est associé un répertoire de connexion (home directory). L'utilisateur y place ses fichiers personnels, et peut y créer autant de sous-répertoires qu'il le désire. Dans l'exemple ci-dessus, le répertoire de connexion de eve est */users/eve*.

Après le login, l'interpréteur de commande a pour répertoire courant le répertoire de connexion de l'utilisateur.

Le répertoire de connexion contient aussi certains fichiers de configuration permettant à l'utilisateur de personnaliser son environnement de travail. Ces fichiers sont normalement invisibles (car leur nom commence par un point, voir la commande ls).

A tout moment, on peut revenir au répertoire de connexion grâce à la commande cd. On peut aussi spécifier un chemin à partir du répertoire de connexion d'un utilisateur en utilisant le caractère ~. Par exemple, *~eve/courrier* désigne le répertoire */users/eve/courrier*.

2. Types de fichiers

Nous appelons fichier tout point dans l'arborescence des fichiers. Tous ne correspondent donc pas à des fichiers de données ordinaires. On distingue 5 types de fichiers :

- les fichiers ordinaires, qui contiennent des données. Unix ne fait aucune différence entre les fichiers de texte et les fichiers binaires. Dans un fichier texte, les lignes consécutives sont séparées par un seul caractère '\n'.
- les répertoires, qui contiennent une liste de références à d'autres fichiers Unix ;
- les fichiers spéciaux, associés par exemple à des pilotes de périphériques ;
- les tubes et sockets, utilisés pour la communication entre processus ;
- les liens symboliques (fichiers "pointant" sur un autre fichier).

3. Droits d'accès

A chaque fichier est associé un utilisateur propriétaire et un ensemble de droits d'accès. Les droits d'accès définissent les possibilités de lecture, écriture et exécution du fichier pour les utilisateurs.

Les utilisateurs d'un fichier donné sont divisés en trois ensembles :

- le propriétaire du fichier ;
- les utilisateurs du même groupe de travail que le propriétaire ;
- les autres utilisateurs ayant accès au système.

Un utilisateur appartenant à l'un de ces ensembles à accès ou non au fichier en lecture (r), en écriture (w) ou en exécution (x). Ces droits (ou permissions) d'accès ne peuvent être changés que par le propriétaire du fichier, grâce à la commande chmod.
La commande ls -l permet d'afficher les droits d'accès à un fichier ; par exemple :

```
$ ls -l polyunix.tex
-rwxr----- 1 emmanuel users 67504 Mar 25 23:29 polyunix.tex
```

indique que fichier polyunix.tex contient 67504 caractères et appartient à l'utilisateur emmanuel. La date et l'heure indiquées sont celles de la dernière modification du contenu du fichier.

Les caractères en début de ligne (-rwxr-----) indiquent le type et les droits d'accès sur ce fichier.

- Le premier caractère donne le type, ici – dénote un fichier ordinaire.
- Les neufs caractères restants sont divisés en trois groupes de trois, indiquant respectivement les droits du propriétaire du fichier, les droits des utilisateurs du même groupe que le propriétaire, et enfin les droits des autres utilisateurs. Le caractère r correspond au droit le lecture (read), w au droit d'écriture (write) et x au droit d'exécution.

Le fichier polyunix.tex montré ci-dessus est donc accessible en lecture, écriture et exécution par son propriétaire, en lecture par les utilisateurs du même groupe et pas du tout aux autres.

4. Le super-utilisateur

Afin de permettre l'administration du système, un utilisateur spécial, nommé super utilisateur (ou root), est toujours considéré par le système comme propriétaire de tous les fichiers (et des processus).

La personne qui gère le système est normalement la seule à connaître son mot de passe. Elle seule peut ajouter de nouveaux utilisateurs au système.

III/ Commandes de base (Shell)

Un Shell est un interpréteur de commande en mode texte. Il peut s'utiliser en mode interactif ou pour exécuter des programmes écrits dans le langage de programmation du Shell (appelés Shell scripts).

En mode interactif, le Shell affiche une invite en début de ligne (prompt), par exemple un caractère $, pour indiquer à l'utilisateur qu'il attend l'entrée d'une commande. La commande est interprétée et exécutée après la frappe de la touche "Entrée". Voici un exemple d'utilisation d'un Shell ; les lignes débutants par $, sont entrées par l'utilisateur (les lignes entrées par l'administrateur sont repérées par #), les autres sont affichées en réponse :

$ pwd
/users/emmanuel/COURS/SYSTEME/POLYUNIX
$ ls
Makefile polyunix.dvi polyunix.tex
fig polyunix.idx polyunix.toc
hello.c polyunix.ind ps
$ ls fig
arbounix.fig tabdesc.fig tube.fig
*$ ls -l *.c*
-rw-r--r-- 1 emmanuel users 84 Mar 25 1996 hello.c

Chaque ligne entrée par l'utilisateur est interprétée par le Shell comme une commande, dont il lance l'exécution. Le premier mot de la ligne est le nom de la commande (par exemple pwd ou ls) ; il est éventuellement suivi d'un certain nombre d'arguments (par exemple fig ou -l).

1. Les différents Shells

Il existe plusieurs Shells Unix : C-Shell (csh ou tcsh), Bourne Shell (sh ou bash), Korn Shell (ksh), ... L'interprétation des commandes simples est semblable pour tous ; par contre l'utilisation pour écrire des scripts diffère beaucoup (définition des variables, structures de contrôle, etc).

Les variantes tcsh et bash apportent un plus grand confort d'utilisation en mode interactif (historique, terminaison automatique des commandes, etc) ; tcsh est compatible avec csh, et bash avec sh. De nos jours, le Shell bash s'est imposé comme le standard sur le système Linux.

Le point commun à tous les Shells est l'emploi d'une syntaxe concise mais obscure et difficilement mémorisable, rendant leur apprentissage difficile (mais leur usage assez divertissant à la longue !). Il est difficile d'administrer finement un système Unix sans posséder quelques bases sur sh et csh, car de nombreux scripts de configuration sont écrits dans ces langages. La tendance actuelle est de généraliser l'emploi d'interfaces graphiques, qui restent toutefois moins souples et puissantes que les scripts. D'autre part, les scripts complexes sont de plus en plus souvent écrits dans des langages interprétés plus puissants comme Python ou Perl.

Pour plus de détails sur ces commandes ou sur leurs options, se reporter au manuel en ligne (commande man).

2. Méta caractères du Shell

Certains caractères, appelés méta caractères, sont interprétés spécialement par le Shell avant de lancer la commande entrée par l'utilisateur.

Par exemple, si l'on entre ls *.c, le Shell remplace l'argument *.c par la liste des fichiers du répertoire courant dont le nom termine par .c.

Les méta caractères permettent donc de spécifier facilement des ensembles de fichiers, sans avoir à rentrer tous leurs noms. Voici les plus utilisés :

- * remplacé par n'importe quelle suite de caractères ;
- ? remplacé par un seul caractère quelconque ;
- [ ] remplacé par l'un des caractères mentionnés entre les crochets.

On peut spécifier un intervalle avec - : [a-z] spécifie donc l'ensemble des lettres minuscules.

Exemples :

```
$ ls
ABCDEF      a      grrr   prog   prog.o
Q.R.S       aa     hel.l.o prog.c x.y.z
$ ls A*
ABCDEF
$ ls *.c
prog.c
$ ls *g*
grrr        prog   prog.c prog.o
$ ls *.?
Q.R.S       hel.l.o prog.c prog.o x.y.z
$ ls [hg]*
grrr        hel.l.o
$ ls *.[a-z].*
hel.l.o     x.y.z
```

On peut empêcher l'interprétation des méta caractères par le Shell en plaçant l'argument entre apostrophes '.

3. Initialisation d'un Shell

Lors de leur démarrage, les Shell exécutent des fichiers de configuration, qui peuvent contenir des commandes quelconques et sont généralement utilisés pour définir des variables d'environnement et des alias.

- csh exécute le fichier ~/.cshrc (le "rc" signifie run command) ;
- tcsh exécute ~/.csh, ou à défaut (si ce dernier n'existe pas) ~/.cshrc ;
- sh exécute ~/.profile ;
- bash exécute ~/.bash_profile ou à défaut ~/.profile.

Rappelons que les fichiers dont le nom commence par un point ne sont pas affichés par la commande ls (sauf si l'on emploie l'option -a) ; les fichiers d'initialisation sont donc "invisibles".

4. Variables d'environnement

Le système Unix défini pour chaque processus une liste de variables d'environnement, qui permettent de définir certains paramètres : répertoires d'installation des utilitaires, type de terminal, etc. Chaque programme peut accéder à ces variables pour obtenir des informations sur la configuration du système.

Depuis le Shell sh (ou bash), les variables d'environnement sont manipulées par les commandes env (affiche la liste), export VARIABLE=VALEUR (donne une valeur à une variable), et echo $VARIABLE (affiche la valeur de la variable).

5. Commandes de manipulation des fichiers

cat [fichier1 ...]

Recopie les fichiers spécifiés l'un après l'autre sur la sortie standard (concaténation). Si aucun argument n'est spécifié, lit sur l'entrée standard (jusqu'à rencontrer un caractère fin de fichier CTRL-d).

La sortie standard est normalement l'écran, et l'entrée standard le clavier, donc cat fichier affiche simplement à l'écran le contenu du fichier spécifié.

cd [chemin]

Change le répertoire courant. Sans argument, ramène dans le répertoire de connexion (HOME).

chmod mode fichier

Modifie les droits d'accès au fichier. Les droits sont spécifiés sous la forme : ensemble d'utilisateurs +/- type de droit.

Exemples :

chmod a+r boite # donne le droit a tous (a) de lire (r) boite;

chmod og-w boite # interdit (-) aux autres (o) et au groupe (g) d'écrire (w);

chmod u+x boite # donne le droit d'exécution (x) au propriétaire (u) du fichier boite.

On peut aussi exprimer les droits sous forme d'un nombre exprimé en base 8 (octal) : chaque groupe de droits (rwx) est codé sur 3 bits.

valeur	droit
0	--
1	-x
2	-w-
3	-wx
4	r-
5	r-x
6	rw-
7	rwx

Ainsi, chmod 777 boite # donne les droits rwx a tous (a) et chmod 750 boite # rwx pour le propriétaire, rx pour le groupe et rien pour les autres.

cp [-ipra] source... dest

Si dest est un répertoire, copie le ou les fichier(s) source vers dest. Si dest est un nom de fichier, renomme source.

Principales options :

- -i demander confirmation en cas d'écrasement de la destination ;
- -p préserve les dates d'accès et de modification ;
- -r copie récursive (descend les sous-répertoires);
- -a copie récursive avec préservation des dates et modes. Utile pour sauvegardes.

echo [-n] message

Affiche les arguments, tels qu'ils sont évalués par le Shell. L'option –n supprime le saut de ligne.

ls [-aldF] chemin1 chemin2 ... chemini

chemini est un nom de fichier ou de répertoire. Si c'est un fichier, affiche sa description ; si c'est un répertoire, affiche la description de son contenu.

Options :

- -a liste tous les fichiers (y compris les .* normalement cachés).
- -l format long (taille, date, droits, etc).
- -d décrit le répertoire et non son contenu.
- -F format court avec indication du type de fichier (ajoute * si exécutable, / si répertoire).
- -i affiche les numéros d'inode des fichiers.

mkdir [chemin]

Crée un répertoire. Le chemin peut être relatif (par exemple mkdir ../exam) ou absolu (par ex. mkdir /users/emmanuel/cours).

mv [-i] source dest

Si dest est un répertoire, déplace le fichier source vers dest. Si dest est un nom de fichier, renomme source. L'option -i permet de demander confirmation en cas d'écrasement de la destination.

pwd

Affiche le répertoire courant.

rm [-ri] fichier ...

Supprime le ou les fichiers spécifiés. L'option -i permet de demander confirmation pour chacun. L'option -r agit de façon récursive, c'est à dire détruit aussi les répertoires (plein ou vide) et leurs sous-répertoires.

6. Redirections des entrées/sorties

Chaque programme sous Unix dispose au moins de deux flux de données : l'entrée standard (stdin), utilisée en lecture, qui est normalement associée au clavier du terminal, et la sortie standard (stdout), utilisée en écriture, normalement associée à l'écran du terminal (ou à la fenêtre de lancement le cas échéant).

Tous les flux de données sont manipulés comme de simples fichiers : on utilisera par exemple la même commande pour lire un caractère au clavier, dans un fichier sur disque ou via une liaison réseau. Ceci simplifie grandement l'écriture des programmes et améliore leur réutilisabilité.

a) Redirections vers/depuis des fichiers

Il est très simple de rediriger l'entrée ou la sortie standard d'un programme lors de son lancement depuis un Shell Unix.

Pour la sortie, on utilisera la construction suivante : *ls > resultat*

Dans ce cas, au lieu d'afficher sur l'écran, la commande ls va créer un fichier nommé ici resultat et y écrire. Rien n'apparait à l'écran (sauf s'il se produit une erreur).

Si l'on désire ne pas effacer l'ancien contenu du fichier, mais écrire à sa fin, on peut utiliser >> : ls >> resultat.

Enfin, pour rediriger l'entrée standard, on utilise < : cat < UnFichier

Il est possible de rediriger l'entrée et la sortie en même temps : cat < UnFichier > Resultat

Notons enfin qu'il existe un troisième flux standard, utilisé pour l'écriture des messages d'erreur (nommé stderr en C, ou cerr en C++). Il se redirige avec >&.

b) Redirections vers des tubes

De façon similaire, il est possible de rediriger la sortie standard d'une commande vers l'entrée standard d'une autre commande grâce au tube (pipe) noté |. Il permet d'affecter la sortie standard d'une commande à l'entrée standard d'une autre, comme un tuyau permettant de faire communiquer l'entrée standard d'une commande avec la sortie standard d'une autre. Les différentes commandes s'exécutent alors en parallèle.

affiche le contenu du répertoire trié à l'envers.

ls | sort -r

7. Contrôle de tâches

Normalement, le Shell attend la fin de l'exécution d'une commande avant d'afficher le prompt suivant. Il arrive que l'on désire lancer une commande de longue durée tout en continuant à travailler dans le Shell. Pour cela, il suffit de terminer la ligne de commande par le caractère & (et commercial).
Par exemple :
$ calcul &
$ ls -Ral / > ls-Ral.txt &
$

Ici, les deux commandes s'exécutent en parallèle, tandis que le Shell attend notre prochaine instruction.
On dit que les processus s'exécutent en tâches de fond. Pour connaitre la liste des tâches de fond lancées de ce Shell, utiliser la commande jobs :
$ jobs
[1] Running calcul
[2] Running ls -Ral / > ls-Ral.txt
$
Le nombre entre crochets est le numéro de la tâche de fond (job). On peut envoyer un signal à une tâche en utilisant kill et % à la place du PID :
$ kill [-signal] %numero_de_tache
Pour remettre une tâche de fond au "premier plan", utiliser la commande fg (utile si la tâche réalise une entrée clavier par exemple).
Enfin, la commande sleep n suspend l'exécution durant n secondes (le processus Shell passe à l'état bloqué durant ce temps).

8. Comment le système exécute une commande ?

Lorsque l'on entre une ligne de commande sous un Shell, ce dernier demande au système d'exécuter la commande, après avoir éventuellement substitué les méta-caractères (*, ? etc.) et remplacés les alias.

A chaque commande doit alors correspondre un fichier exécutable (avec le droit x). Ce fichier exécutable porte le nom de la commande et est recherché par le système dans une liste de répertoires (spécifiée par la variable d'environnement PATH).

La commande which permet de savoir quel est le fichier correspondant à une commande.
which commande affiche le chemin du fichier exécutable correspondant à la commande.
Exemple :
$ which csh
/bin/csh

Le fichier exécutable est soit un fichier binaire (du code en langage machine), qui est alors exécuté directement par le processeur, soit un fichier texte contenant un script.

IV/ Scripts Shell en bash

1. Pourquoi utiliser bash ?

Bash est une version évoluée du Shell sh (le "Bourne Shell"). Le Shell peut être utilisé comme un simple interpréteur de commande, mais il est aussi possible de l'utiliser comme langage de programmation interprété (scripts).
La connaissance du Shell est indispensable au travail de l'administrateur Unix :

- le travail en "ligne de commande" est souvent beaucoup plus efficace qu'à travers une interface graphique ;

- dans de nombreux contextes (serveurs, systèmes embarqués, liaisons distantes lentes) on ne dispose pas d'interface graphique ;
- le Shell permet l'automatisation aisée des tâches répétitives (en scripts) ;
- de très nombreuses parties du système Unix sont écrites en Shell, il faut être capable de les lire pour comprendre et éventuellement modifier leur fonctionnement.

a) Autres versions de Shell

Il existe plusieurs versions de Shell : sh (ancêtre de bash), csh (C Shell), ksh (Korn Shell), zsh, etc. Bash est un logiciel libre, utilisé sur toutes les distributions récentes de Linux et de nombreuses autres variantes d'Unix. Connaissant bash, l'apprentissage d'un autre Shell sur le terrain ne devrait pas poser de difficultés.

b) Shell ou Python

Il est possible d'écrire des programmes en Shell. Pour de nombreuses tâches simples, c'est effectivement très commode. Néanmoins, le langage Shell est forcément assez limité ; pour des programmes plus ambitieux il est recommandé d'utiliser des langages plus évolués comme Python ou Perl, voire des langages compilés (C, C++) si l'on désire optimiser au maximum les performances (au prix de coûts de développement plus importants).

c) Les mauvais côtés des Shell

Le Shell possède quelques inconvénients :

- documentation difficile d'accès pour le débutant (la page de manuel "man bash" est très longue et technique) ;
- messages d'erreurs parfois difficiles à exploiter, ce qui rend la mise au point des scripts fastidieuse ;
- syntaxe cohérente, mais ardue (on privilégie la concision sur la clarté) ;
- relative lenteur (langage interprété sans pré-compilation).

Ces mauvais côtés sont compensés par la facilité de mise en œuvre (pas besoin d'installer un autre langage sur votre système).

2. Variables et évaluation

Les variables sont stockées comme des chaînes de caractères. Les variables d'environnement sont des variables exportées aux processus (programmes) lancés par le Shell. Les variables d'environnement sont gérées par Unix, elles sont donc accessibles dans tous les langages de programmation.

Pour définir une variable :

```
$ var='ceci est une variable'
```

Attention : pas d'espace autour du signe égal. Les quotes (apostrophes) sont nécessaires si la valeur contient des espaces. C'est une bonne habitude de toujours entourer les chaînes de caractères de quotes.

Pour utiliser une variable, on fait précéder son nom du caractère "$" : $ echo $var

ou encore : $ echo 'bonjour, ' $var

Dans certains cas, on doit entourer le nom de la variable par des accolades :

```
$ X=22
$ echo Le prix est ${X}0 euros
```

affiche "Le prix est de 220 euros" (sans accolades autour de X, le Shell ne pourrait pas savoir que l'on désigne la variable X et non la variable X0).

Lorsqu'on utilise une variable qui n'existe pas, le Bash renvoie une chaîne de caractère vide, et n'affiche pas de message d'erreur (contrairement à la plupart des langages de programmation, y compris csh) :

```
$ echo Voici${UNE_DROLE_DE_VARIABLE}!
```

affiche "Voici !".

Pour indiquer qu'une variable doit être exportée dans l'environnement (pour la passer aux commandes lancée depuis ce Shell), on utilise la commande export :

```
$ export X
```

ou directement : $ export X=22

a) Accès aux variables d'environnement dans des programmes

- En langage Python : on accède aux variables via un dictionnaire défini dans le module os : os.environ. Ainsi, la valeur de la variable $PATH est notée os.environ['PATH'].
- En langage C : les fonctions getenv et setenv permettent de manipuler les variables d'environnement. Le programme suivant affiche la valeur de la variable PATH :

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {
char *var = getenv("PATH");
printf("la valeur de PATH est %s\n", var );
}
```

b) évaluation, guillemets et quotes

Avant évaluation (interprétation) d'un texte, le Shell substitue les valeurs des variables. On peut utiliser les guillemets (") et les quotes (') pour modifier l'évaluation.

- les guillemets permettent de grouper des mots, sans supprimer le remplacement des variables. Par exemple, la commande suivante ne fonctionne pas :

```
$ x=Jean
bash: no: command not found
```

Avec des guillemets, c'est bon :

```
$ x="Jean"
```

On peut utiliser une variable entre guillemets :

```
$ y="Titre: $x a fait ses devoirs"
$ echo $y
Titre: Jean a fait ses devoirs
```

- les quotes (apostrophes) groupent les mots et suppriment toute évaluation:

```
$ z='Titre: $x a fait ses devoirs'
$ echo $z
Titre: $x Jean
```

c) Expressions arithmétiques

Normalement, Bash traite les valeurs des variables comme des chaînes de caractères. On peut effectuer des calculs sur des nombres entiers, en utilisant la syntaxe $((...)) pour délimiter les expressions arithmétiques :

```
$ n=1
$ echo $((n + 1))
2
$ p=$((n*5/2))
$ echo $p
2
```

d) Découpage des chemins

Les scripts Shell manipulent souvent chemins (pathnames) et noms de fichiers. Les commandes basename et dirname sont très commodes pour découper un chemin en deux parties (répertoires, nom de fichier) :

```
$ dirname /un/long/chemin/vers/toto.txt
/un/long/chemin/vers
```

$ basename /un/long/chemin/vers/toto.txt
toto.txt

e) évaluation de commandes

Il est courant de stocker le résultat d'une commande dans une variable. Le "résultat" est la chaîne affichée par la commande, et non son code de retour.
Bash utilise plusieurs notations pour cela : les back quotes (') ou les parenthèses :

```
$ REP=‘dirname /un/long/chemin/vers/toto.txt‘
$ echo $REP
/un/long/chemin/vers
```

ou, de manière équivalente :

```
$ REP=$(dirname /un/long/chemin/vers/toto.txt)
$ echo $REP
/un/long/chemin/vers
```

(attention: pas d'espace autour du signe égal).
La commande peut être compliquée, par exemple avec un tube :

```
$ Fichiers=$(ls /usr/include/*.h | grep std)
$ echo $Fichiers
/usr/include/stdint.h /usr/include/stdio_ext.h
/usr/include/stdio.h /usr/include/stdlib.h
/usr/include/unistd.h
```

f) Découpage de chaînes

Bash possède de nombreuses fonctionnalités pour découper des chaînes de caractères. L'une des plus pratiques est basée sur des motifs.
La notation ## permet d'éliminer la plus longue chaine en correspondance avec le motif :

```
$ Var=’tonari no totoro’
$ echo ${Var##*to}
ro
```

ici le motif est *to, et la plus longue correspondance "tonari no toto". Cette forme est utile pour récupérer l'extension (suffixe) d'un nom de fichier :

```
$ F=’rep/bidule.tgz’
$ echo ${F##*.}
tgz
```

La notation # (un seul #) est similaire mais élimine la plus courte chaîne en correspondance :

```
$ Var=’tonari no totoro’
$ echo ${Var#*to}
nari no totoro
```

De façon similaire, on peut éliminer la fin d'une chaine :

```
$ Var=’tonari no totoro’
$ echo ${Var%no*}
tonari
```

Ce qui permet de supprimer l'extension d'un nom de fichier :

```
$ F=’rep/bidule.tgz’
$ echo ${F%.*}
rep/bidule
```

% prend la plus courte correspondance, et %% prend la plus longue :

```
$ Y=’archive.tar.gz’
$ echo ${Y%.*}
archive.tar
$ echo ${Y%%.*}
Archive
```

3. Exécution conditionnelle

L'instruction if permet d'exécuter des instructions si une condition est vraie. Sa syntaxe est la suivante :

```
if [ condition ]
then
   action
fi
```

action est une suite de commandes quelconques. L'indentation (tabultation) n'est pas obligatoire mais très fortement recommandée pour la lisibilité du code. On peut aussi utiliser la forme complète :

```
if [ condition ]
then
   action1
else
   action2
fi
```

ou encore enchaîner plusieurs conditions :

```
if [ condition1 ]
then
   action1
elif [ condition2 ]
then
   action2
elif [ condition3 ]
then
   action3
else
   action4
fi
```

Opérateurs de comparaison

Le Shell étant souvent utilisé pour manipuler des fichiers, il offre plusieurs opérateurs permettant de vérifier diverses conditions sur ceux-ci : existence, dates, droits. D'autres opérateurs permettent de tester des valeurs, chaînes ou numériques. Le tableau ci-dessous donne un aperçu des principaux opérateurs:

Opérateur	Description	Exemple
Opérateurs sur des fichiers		
-e filename	vrai si filename existe	[-e /etc/shadow]
-d filename	vrai si filename est un répertoire	[-d /tmp/trash]
-f filename	vrai si filename est un fichier ordinaire	[-f /tmp/glop]
-L filename	vrai si filename est un lien symbolique	[-L /home]
-r filename	vrai si filename est lisible (r)	[-r /boot/vmlinuz]
-w filename	vrai si filename est modifiable (w)	[-w /var/log]
-x filename	vrai si filename est exécutable (x)	[-x /sbin/halt]
file1 -nt file2	vrai si file1 plus récent que file2	[/tmp/foo -nt /tmp/bar
file1 -ot file2	vrai si file1 plus ancien que file2	[/tmp/foo -ot /tmp/bar
Opérateurs sur les chaînes		
-z chaine	vrai si la chaine est vide	[-z "$VAR"]
-n chaine	vrai si la chaine est non vide	[-n "$VAR"]
chaine1 = chaine2	vrai si les deux chaînes sont égales	["$VAR" = "totoro"]
chaine1 != chaine2	vrai si les deux chaînes sont différentes	["$VAR" != "tonari"]
Opérateurs de comparaison numérique		
num1 -eq num2	égalité	[$nombre -eq 27]
num1 -ne num2	inégalité	[$nombre -ne 27]
num1 -lt num2	inférieur (<)	[$nombre -lt 27]
num1 -le num2	inférieur ou égal (<=)	[$nombre -le 27]
num1 -gt num2	supérieur (>)	[$nombre -gt 27]
num1 -ge num2	supérieur ou égal (>=)	[$nombre -ge 27]

Quelques points délicats doivent être soulignés :

- Toutes les variables sont de type chaîne de caractères. La valeur est juste convertie en nombre pour les opérateurs de conversion numérique.
- Il est nécessaire d'entourer les variables de guillemets (") dans les comparaisons.

Le code suivant affiche "OK" si $var est égale à "tonari no totoro" :

```
if [ "$myvar" = "tonari no totoro" ]
then
echo "OK"
fi
```

Par contre, si on écrit la comparaison comme

```
if [ $myvar = "tonari no totoro" ]
```

le Shell déclenche une erreur si $myvar contient plusieurs mots. En effet, la substitution des variables a lieu avant l'interprétation de la condition.

Scripts shell

Un script Bash est un simple fichier texte exécutable (droit x) commençant par les caractères #!/bin/bash (doivent être les premiers caractères du fichier).

Voici un exemple de script :

```
#!/bin/bash
if [ "${1##*.}" = "tar" ]
then
echo $1 est une archive tar
else
```

```
echo $1 n\'est pas une archive tar
fi
```

Ce script utilise la variable $1, qui est le premier argument passé sur la ligne de commande.

Arguments de la ligne de commande

Lorsqu'on entre une commande dans un Shell, ce dernier sépare le nom de la commande (fichier exécutable ou commande interne au Shell) des arguments (tout ce qui suit le nom de la commande, séparés par un ou plusieurs espaces). Les programmes peuvent utiliser les arguments (options, noms de fichiers à traiter, etc).
En bash, les arguments de la ligne de commande sont stockés dans des variables spéciales :
$0, $1, ... les arguments
$# le nombre d'arguments
$* tous les arguments

Le programme suivant illustre l'utilisation de ces variables :

```
#!/bin/bash
echo 'programme :' $0
echo 'argument 1 :' $1
echo 'argument 2 :' $2
echo 'argument 3 :' $3
echo 'argument 4 :' $4
echo "nombre d'arguments :" $#
echo "tous:" $*
```

Exemple d'utilisation, si le script s'appelle "myargs.sh" :

```
$ ./myargs.sh un deux trois
programme : ./myargs.sh
argument 1 : un
argument 2 : deux
argument 3 : trois
argument 4 :
nombre d'arguments : 3
tous: un deux trois
```

4. Autres structures de contrôle

Nous avons déjà évoqué l'instruction if et les conditions. On utilise souvent des répétitions (for) et des choix multiples (case).

a) Boucle for

Comme dans d'autre langages (par exemple python), la boucle for permet d'exécuter une suite d'instructions avec une variable parcourant une suite de valeurs. Exemple :

```
for x in un deux trois quatre
do
        echo x= $x
done
```

affichera :

```
x= un
x= deux
x= trois
```

```
x= quatre
```
On utilise fréquemment for pour énumérer des noms de fichiers, comme dans cet exemple :
```
for fichier in /etc/rc*
do
   if [ -d "$fichier" ]
   then
      echo "$fichier (repertoire)"
   else
   echo "$fichier"
   fi
done
```
Ou encore, pour traiter les arguments passés sur la ligne de commande :
```
#!/bin/bash
for arg in $*
do
   echo $arg
done
```

b) Instruction case

L'instruction case permet de choisir une suite d'instruction suivant la valeur d'une expression :
```
case "$x" in
   go)
      echo "demarrage"
      ;;
   stop)
      echo "arret"
      ;;
   *)
      echo "valeur invalide de x ($x)''
esac
```
Noter les deux ; pour signaler la fin de chaque séquence d'instructions.

5. Définition de fonctions

Il est souvent utile de définir des fonctions. La syntaxe est simple :

```
mafonction() {
        echo "appel de mafonction..."
}

mafonction
mafonction
```

qui donne :

```
appel de mafonction...
appel de mafonction...
```

Voici pour terminer un exemple de fonction plus intéressant :

```
tarview() {
   echo -n "Affichage du contenu de l'archive $1 "
   case "${1##*.}" in
      tar)
```

```
        echo "(tar compresse)"
        tar tvf $1
    ;;
    tgz)
        echo "(tar compresse gzip)"
        tar tzvf $1
    ;;
    bz2)
        echo "(tar compresse bzip2)"
        cat $1 | bzip2 -d | tar tvf -
    ;;
    *)
        echo "Erreur, ce n'est pas une archive"
    ;;
  esac
}
```

Plusieurs points sont à noter :

- echo -n permet d'éviter le passage à la ligne ;
- La fonction s'appelle avec un argument ($1) tarview toto.tar

V/ Utilitaires UNIX

Cette partie décrit quelques commandes utilitaires Unix très pratiques. Ces programmes sont livrés en standard avec toutes les versions d'Unix.

1. L'éditeur vi

vi est un éditeur de texte "plein écran" (par opposition aux éditeurs "ligne" qui ne permettaient que l'édition d'une ligne à la fois).

Comparé aux éditeurs modernes, vi est d'abord malcommode, mais il est assez puissant et toujours présent sur les systèmes Unix (notons qu'il existe sous Unix des éditeurs pour tous les goûts, depuis Emacs pour les développeurs jusqu'à gedit et autres jedit pour les utilisateurs allergiques au clavier (mais pas à la souris)).

Il est donc très utile d'en connaître le maniement de base. Seulement les commandes et modes les plus utilisés sont décrits, il en existe beaucoup d'autres.

A un instant donné, vi est soit en mode commande, soit en mode insertion :

- mode commande : les caractères tapés sont interprétés comme des commandes d'édition. vi démarre dans ce mode, il faut donc lui indiquer (commande 'i') que l'on veut insérer du texte ;
- mode insertion : les caractères sont insérés dans le texte édité. On peut quitter ce mode en pressant la touche "ESC" (ou "Echap" sur certains claviers).

Appel de vi depuis le shell :

vi	fichier édite fichier.
vi +n	fichier commence à la ligne n.
vi -r	fichier récupération du fichier après un crash.

Mouvements du curseur (en mode commande seulement) :

Touche	Action
flèches	Déplace curseur (pas toujours bien configuré).
ESPACE	Avance à droite.
h	Recule à gauche.
CTRL-n	Descend d'une ligne.
CTRL-p	Monte d'une ligne.
CTRL-b	Monte d'une page.
CTRL-f	Descend d'une page.
nG	Va à la ligne n (n est un nombre).

Commandes passant en mode insertion :

Touche	Commence l'insertion
i	Avant le curseur.
I	Au début de la ligne.
A	A la fin de la ligne.

Autres commandes :

r	Remplace le caractère sous le curseur.
x	Supprime un caractère.
d$	Efface jusqu'à la fin de la ligne.
dd	Efface la ligne courante.
/chaîne	Cherche la prochaine occurrence de la chaîne.
?chaîne	Cherche la précédente occurrence de la chaîne.

Quitter, sauvegarder : (terminer la commande par la touche "Entrée")

:w	écrit le fichier.
:x	écrit le fichier puis quitte vi.
:q !	Quitte vi sans sauvegarder les changements.
!	!commande Exécute commande Shell sans quitter l'éditeur.

2. <u>Commandes diverses</u>

compress [fichier]

Compresse le fichier (pour gagner de l'espace disque ou accélérer une transmission réseau). Le fichier est remplacé par sa version compressée, avec l'extension '.Z'. Décompression avec uncompress ou zcat.

date

Affiche la date et l'heure. Comporte de nombreuses options pour indiquer le format d'affichage.

diff fichier1 fichier2

Compare ligne à ligne des deux fichiers texte fichier1 et fichier2, et décrit les transformations à appliquer pour passer du premier au second. Diverses options modifient le traitement des blancs, majuscules etc.

diff peut aussi générer un script pour l'éditeur ed permettant de passer de fichier1 à fichier2 (utile pour fabriquer des programmes de mise à jour ("patchs")).

file fichier

Essaie de déterminer le type du fichier (exécutable, texte, image, son,...) et l'affiche.

find [options]

Cette commande permet de retrouver dans un répertoire ou une hiérarchie de répertoires les fichiers possédant certaines caractéristiques (nom, droits, date etc..) ou satisfaisant une expression booléenne donnée.

find parcourt récursivement une hiérarchie de fichiers. Pour chaque fichier rencontré, find teste successivement les prédicats spécifiés par la liste d'options, jusqu'au premier qui échoue ou jusqu'à la fin de la liste.

Principales options :

-name nom le fichier à ce nom;

-print écrit le nom du fichier (réussit toujours) ;

-exec exécute une commande. {} est le fichier courant. Terminer par; .

-type (d : catalogue, f : fichier ordinaire, p : pipe, l : lien symbolique).

-newer fichier compare les dates de modification ;
-o ou ;
-prune si le fichier courant est un catalogue, élague l'arbre à ce point.

Exemples :

```
# Recherche tous les fichier nommes "essai"
# a partir de la racine
find / -name essai -print
# Recherche tous les fichier commencant par "ess"
# a partir du repertoire courant
find . -name 'ess*' -print
# Affiche a l'ecran le contenu de tous les fichiers .c
find . -name '*.c' -print -exec cat '{}' \;
```

(l'écriture de ce dernier exemple est compliquée par le fait que le Shell traite spécialement les caractères *, {, et ;, que l'on doit donc entourer de quotes '.)

head [-n] [fichier]
Affiche les n premières lignes du fichier. Si aucun fichier n'est spécifié, lit sur l'entrée standard.

lpr [-Pnom-imprimante] [fichier]
Demande l'impression du fichier (le place dans une file d'attente). Si aucun fichier n'est spécifié, lit sur l'entrée standard.
Voir aussi la commande lp.
L'impression d'un fichier sous Unix passe par un spooler d'impression.
Ce spooler est réalisé par un démon (c'est à dire un processus système qui s'exécute en tâche de fond).

lpq [-Pnom-imprimante]
Permet de connaître l'état de la file d'attente associée à l'imprimante.

lprm [-Pnom-imprimante] numjob
Retire un fichier en attente d'impression. On doit spécifier le numéro du job, obtenu grâce à la commande lpq.

lp [-dnom-imprimante] fichier
Comme lpr, sur certains systèmes.

md5sum [fichier]
Calcule et/ou vérifie la somme "MD5" d'un fichier ou flux. Voir RFC 1321.

man [n] commande
Affiche la page de manuel (aide en ligne) pour la commande. L'argument n permet de spécifier le numéro de la section de manuel (utile lorsque la commande existe dans plusieurs sections). Les numéros des sections sont : 1 (commandes utilisateur), 2 (appels systèmes), 3 (fonctions librairies C), 4 (devices), 5 (formats de fichiers), 6 (jeux), 7 (divers), 8 (administration) et n (new, programmes locaux).

more [fichier]
Affiche un fichier page par page (aide en ligne, taper 'h'). Si aucun fichier n'est spécifié, lit sur l'entrée standard.

less [fichier]
less is more. Comme more, avec plus de fonctionnalités (taper 'h' pour en savoir plus).

tail [+n | -n] [fichier]
La forme tail +n permet d'afficher un fichier à partir de la ligne n. La forme tail -n affiche les n dernières lignes du fichier. Si aucun fichier n'est spécifié, lit sur l'entrée standard.

tar options [fichier ou répertoire]
La commande tar permet d'archiver des fichiers ou une arborescence de fichiers, c'est à dire de les regrouper dans un seul fichier, ce qui est très pratique pour faire des copies de sauvegardes d'un disque, envoyer plusieurs fichiers en une seul fois par courrier électronique, etc.

Pour créer une nouvelle archive, utiliser la forme $ tar cvf nom-archive répertoire qui place dans le nouveau fichier "nom-archive" tous les fichiers situés sous le répertoire indiqué. On donne généralement l'extension .tar aux fichiers d'archives.

Pour afficher le contenu d'une archive, utiliser $ tar tvf nom-archive

Pour extraire les fichiers archivés, utiliser $ tar xvf nom-archive

les fichiers sont crées à partir du répertoires courant.

Nombreuses autres options. Notons que les noms de fichiers peuvent être remplacés par - pour utiliser l'entrée ou la sortie standard (filtres), comme dans les exemples ci-dessous :

- Archivage d'un répertoire et de ses sous-répertoires dans un fichier archive.tar :

  $ tar cvf archive.tar repertoire
- Archivage et compression au vol :

  $ tar cvf - repertoire | gzip > archive.tar.gz
- Pour afficher l'index de l'archive ci-dessus :

  $ zcat archive.tar.gz | tar tvf -
- l'option z permet de (dé)comprimer directement. On utilise dans ce cas l'extension .tgz :

  $ tar cvfz archive.tgz repertoire
- Copie complète récursive d'un répertoire repert dans le répertoire destination :

  $ tar cvf - repert | (cd destination; tar xvfp -)

Les parenthèses sont importantes : la deuxième commande tar s'exécute ainsi dans le répertoire de destination.

Cette façon de procéder est supérieure à cp -r car on préserve les propriétaires, droits, et dates de modifications des fichiers (très utile pour effectuer des sauvegardes).

uncompress [fichier]

Décompresse un fichier (dont le nom doit terminer par .Z) compressé par compress.

wc [-cwl] [fichier ...]

Affiche le nombre de caractères, mots et lignes dans le(s) fichier(s). Avec l'option -c, on obtient le nombre de caractères, avec -w, le nombre de mots (words) et avec -l le nombre de lignes.

which commande

Indique le chemin d'accès du fichier lancé par "commande".

who [am i]

Liste les utilisateurs connectés au système. La forme who am i donne l'identité de l'utilisateur.

zcat [fichiers]

Similaire à cat, mais décompresse au passage les fichiers (ou l'entrée standard) compressés par compress.

uuencode [fichier] nom

Utilisé pour coder un fichier en n'utilisant que des caractères ascii 7 bits (codes entre 0 et 127), par exemple pour le transmettre sur un réseau ne transmettant que les 7 bits de poids faible.

uudencode lit le fichier (ou l'entrée standard si l'on ne précise pas de nom) et écrit la version codée sur la sortie standard. Le paramètre nom précise le nom du fichier pour le décodage par uudecode.

uudecode [fichier]

Décode un fichier codé par uuencode. Si l'argument fichier n'est pas spécifié, lit l'entrée standard. Le nom du fichier résultat est précisé dans le codage (paramètre nom de uuencode).

3. Filtres de texte

Les filtres sont des utilitaires qui prennent leurs entrées sur l'entrée standard, effectuent un certain traitement, et fournissent le résultat sur leur sortie standard. Notons que plusieurs utilitaires déjà présentés peuvent être vus comme des filtres (head, tail).

grep [option] motif fichier1 ... fichiern

Affiche chaque ligne des fichiers fichieri contenant le motif motif. Le motif est une expression régulière.

Options : -v affiche les lignes qui ne contiennent pas le motif.
-c seulement le nombre de lignes.
-n indique les numéros des lignes trouvées.
-i ne distingue pas majuscules et minuscules.

sort [-rn] [fichier]

Trie les lignes du fichier, ou l'entrée standard, et écrit le résultat sur la sortie. L'option -r renverse l'ordre du tri, l'option -n fait un tri numérique.

tr [options] chaine1 chaine2

Recopie l'entrée standard sur la sortie standard en remplaçant tout caractère de chaine1 par le caractère de position correspondante dans chaine2.

uniq [-cud] fichier

Examine les données d'entrée ligne par ligne et détermine les lignes dupliquées qui sont consécutives : -d permet de retenir que les lignes dupliquées. -u permet de retenir que les lignes non dupliquées. -c permet de compter l'indice de répétition des lignes.

4. Manipulation des processus

kill -sig PID

Envoie le signal sig au processus de numéro PID. sig peut être soit le numéro du signal, soit son nom (par exemple kill -STOP 1023 est l'équivalent de kill -19 1023).

ps [-e][-l]

Affiche la liste des processus.

L'option -l permet d'obtenir plus d'informations. L'option -e permet d'afficher les processus de tous les utilisateurs.

ps affiche beaucoup d'informations. Les plus importante au début sont :

- UID : identité du propriétaire du processus ;
- PID : numéro du processus ;
- PPID : PID du père du processus ;
- NI : priorité (nice) ;
- S : état du processus (R si actif, S si bloqué, Z si terminé).

nice [-priorite] commande

Lance l'exécution de la commande en modifiant sa . Permet par exemple de lancer un processus de calcul en tâche de fond sans perturber les processus interactifs (éditeurs, shells etc.), en lui affectant une priorité plus basse.

Le noyau accorde moins souvent le processeur aux processus de basse priorité.

renice priorite -p pid

Modifie la priorité d'un processus.

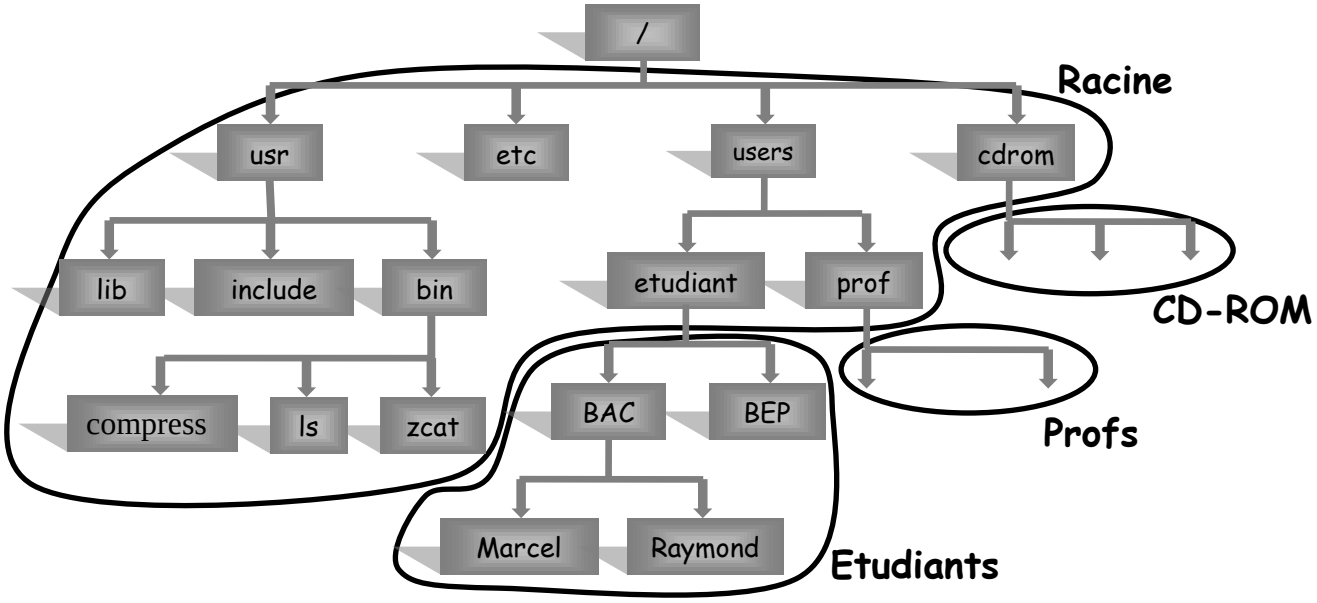
5. Gestion des systèmes de fichiers

a) Principes

Tous les disques, disquettes et CD-ROMs connectés à un ordinateur sous Unix sont accédés via une arborescence unique, partant du répertoire racine /. La complexité du système est ainsi masquée à l'utilisateur, qui peut utiliser les mêmes commandes pour accéder à un fichier sur disque dur ou sur CDROM.

Remarquez que ce principe est différent de celui employé par les systèmes MS-DOS et Windows, pour lesquels chaque volume (disque) possède une racine spécifique repérée par une lettre (A:\, C:\, etc).

Bien entendu, les fichiers sont organisés de façons différentes sur les disques durs, les disquettes, ou les CD-ROMS : chaque périphérique possède son propre système de fichiers.



Dans cet exemple, l'arborescence des fichiers est composée de quatre systèmes de fichiers : le système racine (point de montage /, deux systèmes pour les utilisateurs (/users/etudiants et /users/profs), et un système pour le CD-ROM (/cdrom).

Les différents systèmes de fichier, que l'on peut considérer comme des sous-arborescences associées à un périphérique spécial, sont montées sur l'arborescence racine. Chaque système de fichier doit posséder un point de montage, qui est au départ (avant montage) un simple répertoire vide.

La commande mount permet d'associer au point de montage le système de fichier correspondant. Il faut lui préciser le pilote de périphérique (device driver) utilisé pour accéder à ce système. Nous n'étudions pas dans ce cours la gestion des pilotes de périphériques sous UNIX ; notons simplement que chaque pilote correspond à un pseudo-fichier dans le répertoire système /dev.

b) <u>Commandes de base</u>

Nous décrivons simplement trois commandes permettant d'observer l'état des systèmes de fichiers. Nous de décrivons pas leur utilisation pour administrer le système (ajouter un système de fichier, etc).

mount [point d'entree]

Permet d'afficher la liste des systèmes de fichiers en service (montés). Le format d'affichage dépend légèrement de la version d'Unix utilisée.

Un exemple simple sous Linux :

```
$ mount
/dev/sda3 on / type ext2 (rw)
/dev/sda4 on /users type ext2 (rw)
/dev/sdb1 on /jaz type ext2 (rw)
```

On a ici trois systèmes de fichiers, gérés par les pilotes /dev/sda3, /dev/sda4 et /dev/sdb1 (trois disques durs SCSI), et montés respectivement sur la racine, /users et /jaz.

La commande mount est aussi utilisée pour monter un nouveau système de fichiers dans l'arborescence (lors du démarrage du système ou de l'ajout d'un disque).

df [chemin]

df (describe filesystem) affiche la capacité totale d'un système de fichiers (généralement en KiloOctets), le volume utilisé et le volume restant disponible.

Le format d'affichage dépend aussi de la version utilisée. Exemple sous Linux :

```
$ df /users
Filesystem 1024-blocks Used Available Capacity Mounted on /dev/sda4 417323 347789 47979 88%
/users
```

On a ici 417323 Ko au total, dont 347789 sont utilisés et 47979 libres.

du [-s] [chemin]

du (disk usage) affiche la taille occupée par le fichier ou répertoire spécifié par l'argument chemin. Avec l'option -s, n'affiche pas le détail des sous-répertoires.

Exemple :

```
$ du POLYUNIX
10 POLYUNIX/fig
19 POLYUNIX/ps
440 POLYUNIX
$ du -s POLYUNIX
440 POLYUNIX
```

Le répertoire POLYUNIX occupe ici 440 Ko.

VI/ Programmation en C sous Unix

1. Le compilateur C

Par convention, les fichiers contenant du source C possèdent l'extension .c, les fichiers headers l'extension .h et les fichiers objets .o. Il n'y a pas de convention spéciale pour les exécutables qui doivent par contre posséder le droit d'exécution x.

Soit le fichier hello.c contenant le texte source suivant :

```
/* Exemple simple */
#include <stdio.h>
main() {
printf("Hello, world !\n");
}
```

Le compilateur C peut s'appeler depuis une ligne de commande shell sous la forme :

```
$ cc hello.c
```

($ désigne le prompt du shell). Le programme hello.c est alors compilé et un fichier exécutable nommé a.out est créé. On peut lancer son exécution en tapant :

```
$ a.out
Hello, world !
```

Il est possible de spécifier le nom de l'exécutable produit grâce à l'option -o :

```
$ cc hello.c -o hello
```

La commande cc admet de nombreuses options sur la ligne de commande.

Une option très utile est -I qui spécifie un répertoire où rechercher les fichiers inclus par la directive #include. Par exemple, si l'on a placé nos fichiers .h dans sous-répertoire inc, on pourra utiliser :

```
$ cc -Iinc exo.c -o exo
```

Sur certains systèmes, d'autres options sont nécessaire pour spécifier la variante de langage C utilisée (K&R ou ANSI) et l'utilisation ou non de la norme POSIX.

2. La commande make

Lorsque l'on développe un programme un peu plus important que l'exemple donné plus haut, il devient vite fastidieux d'utiliser directement la commande cc, avec les bonnes options et sans oublier de recompiler tous les fichiers que l'on a modifiés depuis la dernière mise à jour.

Supposons que l'on travaille sur un projet comportant deux fichiers sources, main.c et func.c, qui utilisent tous deux un fichier inc/incl.h placé dans un sous-répertoire. L'exécutable que l'on fabrique est nommé essai.

En utilisant cc, il faut faire :

$ cc -c -Iinc -o main.o main.c
$ cc -c -Iinc -o func.o func.c
$ cc -o essai main.o func.o

L'option -c spécifie que l'on veut fabriquer un fichier objet (.o) qui sera lié ultérieurement pour obtenir un exécutable.

Si l'on modifie main.c, il faut refaire les étapes 1 et 3. Si l'on modifie le fichier inclut, il faut tout refaire. On dit que essai dépend de main.o et func.o, qui dépendent eux même de main.c, func.c et de incl.h.

Ceci signifie que notre exécutable final (essai) est à jour si sa date de modification est plus grande que celle de main.o et func.o, et ainsi de suite.

La commande make permet d'automatiser ce genre de chaînes de production. Son utilisation permet au développeur de s'assurer qu'un module donné est toujours à jour par rapport aux divers éléments dont il dépend. make se charge de déclencher les actions nécessaires pour reconstruire le module cible.

make [-f fichier-dep]

make utilise un fichier spécial (appelé "makefile") indiquant les dépendances entre les divers fichiers à maintenir. Ce fichier est nommé par défaut Makefile.

Syntaxe d'un fichier makefile :

but : dependance1 [...[dependance_n]]
commande1
commande2
...

La ou les cibles doivent apparaître sur la première ligne et être suivies de ' :'. A la suite figure la liste des dépendances conduisant à la cible. Les règles de production sont décrites sur les lignes suivantes, qui doivent impérativement commencer par un caractère de tabulation.

Voici un exemple de fichier Makefile traitant l'exemple précédent :

all: essai
essai: main.o func.o
cc -o essai func.o main.o
CFLAGS= -Iinc
func.o: inc/incl.h
main.o: inc/incl.h

La première ligne (all) indique que la cible par défaut est essai : lorsque l'on actionne make sans arguments, on veut fabriquer essai.

Ensuite, on indique que essai dépend des deux fichiers objets, et qu'il se fabrique à partir d'eux en actionnant cc -o essai ....

CFLAGS est une variable spéciale indiquant les options à passer au compilateur C, ici -Iinc.

Enfin, on spécifie que les fichiers .o dépendent du fichier .h. Notons que make utilise un certain nombre de règles par défaut, par exemple celle disant que les fichiers .o dépendent des fichiers .c et se fabriquent en utilisant cc. Il n'est donc pas nécessaire de faire figurer la règle complète, qui s'écrirait :

main.o: inc/incl.h main.c
cc -Iinc -c -o main.o main.c

3. Arguments sur la ligne de commande

L'exécution d'un programme nommé exemple est lancé depuis un shell par :

$ exemple

Si l'on entre

$ exemple un deux trois

le programme est lancé de la même façon, et il reçoit les chaînes de caractères "un", "deux" et "trois" comme arguments. En langage C, ces arguments peuvent facilement être récupérés par la fonction main() du programme (qui, rappelons le, est la première fonction appelée lors du lancement de ce programme).

Il suffit de déclarer la fonction main() comme ceci :

void main(int argc, char *argv[])

la variable argc reçoit le nombre de paramètres sur la ligne de commande. argv est un tableau de chaînes de caractères contenant les paramètres.

Exemple : soit un exécutable nommé essai, si l'on entre $ essai il fait beau, on aura :

argc = 4

argv[0] = "essai"
argv[1] = "il"
argv[2] = "fait"
argv[3] = "beau"
Notons que argv[0] contient toujours le nom du programme lancé.

4. Variables d'environnement

En langage C, on peut accéder à la liste de ces variables par l'intermédiaire du troisième argument de la fonction main(), qui est alors déclarée comme :
void main(int argc, char *argv[], char *arge[])
arge est un tableau de chaînes de caractères définissant les variables d'environnement, sous la forme "NOM=VALEUR".
On peut aussi utiliser les fonctions getenv() et setenv().

Attention, chaque processus hérite d'une copie des variables d'environnement de son père. Les modifications qu'il peut apporter à ces variables n'affecteront donc que ses descendants (les processus qu'il lancera), jamais le processus père.

5. Allocation mémoire

Pour allouer des blocs de mémoire dynamique sous UNIX, il est conseillé d'utiliser les fonctions de la librairie C standard : malloc() et free() en C, new et delete en C++.
#include <stdlib.h>
void *malloc(int nb_octets);
void *calloc(int nb_elem, int taille_elem);
void free(void *ptr);

La fonction malloc() alloue un bloc de mémoire contigüe de nb_octets octets. La fonction calloc() alloue un bloc de nb_elem x taille_elem octets et l'initialise à zéro. Les blocs alloués sont libérés après usage par la fonction free().

6. Manipulation de fichiers

a) Fonctions de la librairie C

La librairie C standard permet de manipuler des fichiers (ouverture, lecture, écriture) via des pointeurs sur des flux de type FILE. Ces fonctions sont définies dans le fichier include <stdio.h>. Voici les principales fonctions utilisées:
#include <stdio.h>
FILE *fopen(char *chemin, char *mode);
int fclose(FILE *);
int fprintf(FILE *, char *format, ...);
int fwrite(void *ptr, int size, int nbelem, FILE *);
int fread(void *ptr, int size, int nbelem, FILE *);
int fflush(FILE *);
int feof(FILE *);

Notez que ces fonctions existent aussi bien en C sous DOS que sous Macintosh ou Unix. Elles doivent être utilisées si l'on désire écrire un programme portable. Dans ce cas, il est évident que l'on s'interdit l'usage de toutes les fonctionnalités propres à chaque système d'exploitation.

Note : les opérations d'entrées/sorties effectuées par les fonctions ci-dessus sont bufférisées ; ainsi, l'appel à fwrite() ou fprintf() copie simplement les données dans un tampon (ou buffer), qui est vidé une fois plein vers le fichier concerné. Ceci permet d'éviter de multiplier inutilement les opérations d'entrées/sorties, souvent coûteuses. Une conséquence troublante pour le débutant est que lors de l'envoi de caractères vers l'écran, ceux-ci n'apparaissent pas immédiatement, sauf si l'on force la vidange du tampon avec un retour chariot ou l'appel explicite à fflush().

b) Fichiers et descripteurs sous UNIX

Pour manipuler plus finement les fichiers Unix, il est souvent nécessaire d'utiliser directement des appels système de plus bas niveau que ceux de la librairie C.

Le noyau UNIX maintient une table des fichiers ouverts par un processus. Le terme fichier désigne ici un flux de données unidirectionnel qui n'est pas nécessairement associé à un fichier disque.

Chaque processus dispose toujours à son lancement de trois flux de données :

1. l'entrée standard, associée normalement au clavier du terminal de lancement;
2. la sortie standard, associée normalement à l'écran du terminal (ou à la fenêtre de lancement) ;
3. la sortie d'erreur, qui coïncide par défaut avec la sortie standard.

Un programme accède aux flux de données ouverts en spécifiant des descripteurs, qui sont simplement des nombre entiers correspondant à l'indice du flux dans la table. La table est remplie dans l'ordre d'ouverture des fichiers, en commençant à 3 puisque les trois premiers descripteurs sont toujours occupés par les flux standards

c) Appels systèmes manipulant les fichiers

#include <sys/types.h>

Table des descripteurs d'un processus après ouverture d'un fichier TOTO. Le prochain fichier ouvert se verrait attribuer le descripteur numéro 4.

0	Entrée standard
1	Sortie standard
2	Sortie d'erreur
3	Fichier TOTO

```
#include <sys/stat.h>
#include <fcntl.h>
int open( char *chemin, int mode, ... );
int close( int desc );
int read( int desc, void *ptr, int nb_octets );
int write( int desc, void *ptr, int nb_octets );
```

La fonction open() ouvre un fichier en lecture ou en écriture, crée un nouveau descripteur et le retourne.
close() ferme un descripteur.
read() et write() permettent de lire ou d'écrire une zone mémoire sur un descripteur.

7. Lancement d'une commande

Pour lancer l'exécution d'une commande externe depuis un programme C, on peut utiliser l'appel

```
int system ( char *comm );
```

Le processus qui appelle system() est bloqué jusqu'à la fin de la commande lancée.

VII/ Les processus

Nous étudions dans cette section comment sont créés des processus UNIX, et quels sont les appels système à utiliser en langage C.

1. Définition

Un programme qui s'exécute est appelé un processus. Un processus comporte du code machine exécutable, une zone mémoire (données allouées par le processus) et une pile (pour les variables locales des fonctions et la gestion des sous-programmes).

Lors de la création d'un processus, le noyau lui attribue un numéro unique, nommé PID (processus identifier). Ce numéro permet de repérer le processus.

La commande ps affiche la liste des processus. Chaque processus a un utilisateur propriétaire, qui est utilisé par le système pour déterminer ses permissions d'accès aux fichiers.

2. Création de processus

Lorsqu'on entre une commande dans un Shell, le Shell lance un processus pour l'exécuter (sauf pour les commandes les plus fréquentes, comme ls, qui sont normalement traitées directement par le Shell pour gagner en efficacité). Le Shell attend ensuite la fin de ce processus, puis demande la commande suivante.

Chaque processus a ainsi un processus père, qui est celui qui l'a lancé. Le numéro du processus père est noté PPID (parent PID). Un processus n'a bien sûr qu'un seul père, mais peut lancer l'exécution de plusieurs autres processus, nommés processus fils. On a donc affaire à une arborescence de processus. Lors de l'initialisation du système Unix, un premier processus, nommé init, est créé avec un PID=1. init est l'ancêtre de tous les processus.

A bas niveau, il n'y a qu'une seule façon de donner naissance à un nouveau processus, la duplication. Un processus peut demander au système sa duplication en utilisant la primitive fork() (que nous étudions plus loin). Le système crée alors une copie complète du processus, avec un PID différent.

L'un des deux processus est fils de l'autre.

Il n'est évidement guère utile d'avoir deux copies du même programme s'exécutant dans le système. Après duplication, le fils va "changer de programme" en utilisant la primitive exec(). Cette primitive conserve l'identité du processus mais remplace son code exécutable (et ses données) par celui d'une nouvelle commande.

Voyons ce qu'il se passe lorsque qu'un Shell exécute la commande
compress toto
qui demande la compression du fichier nommé toto :

1. le Shell se duplique (fork) ; on a alors deux processus Shell identiques.
2. le Shell père se met en attente de la fin du fils (wait).
3. le Shell fils remplace son exécutable par celui de la commande compress ;
4. la commande compress s'exécute et compacte le fichier toto ; lorsqu'elle se termine, le processus fils disparaît.
5. le père est alors réactivé, et affiche le prompt suivant.

3. Manipulation de processus en langage C

Examinons maintenant les appels systèmes à utiliser pour créer un nouveau processus.

a) La primitive fork()

#include <unistd.h>
int fork();

L'appel à fork() duplique le processus. L'exécution continue dans les deux processus après l'appel à fork(). Tout se passe comme si les deux processus avaient appelé fork(). La seule différence (outre le PID et le PPID) est la valeur retournée par fork() :

- dans le processus père (celui qui l'avait appelé), fork() retourne le PID du processus fils créé ;
- dans le processus fils, fork() retourne 0.

Notons que le fork peut échouer par manque de mémoire ou si l'utilisateur a déjà créé trop de processus ; dans ce cas, aucun fils n'est créé et fork() retourne -1.

b) Les primitives getpid() et getppid()

L'appel système getpid() retourne le PID du processus appelant. getppid() retourne le PID du père du processus.

c) La primitive exec()

La primitive execlp() permet le recouvrement d'un processus par un autre exécutable.

int execlp(char *comm, char *arg, ..., NULL);

comm est une chaîne de caractères qui indique la commande à exécuter. arg spécifie les arguments de cette commande (argv).

Exemple : exécution de ls -l /usr

execlp("ls", "ls", "-l", "/usr", NULL);

Notons que la fonction execlp() retourne -1 en cas d'erreur. Si l'opération se passe normalement, execlp() ne retourne jamais puisqu'elle détruit (remplace) le code du programme appelant.

d) La primitive exit()

exit() est une autre fonction qui ne retourne jamais, puisqu'elle termine le processus qui l'appelle.

#include <stdlib.h>
void exit(int status);

L'argument status est un entier qui permet d'indiquer au Shell (ou au père de façon générale) qu'une erreur s'est produite. On le laisse à zéro pour indiquer une fin normale.

```
/* Exemple utilisation primitive fork() sous UNIX
*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
void main(void) {
   int pid; /* PID du processus fils */
   int i;
   pid = fork();
   switch (pid) {
   case -1:
      printf("Erreur: echec du fork()\n");
      exit(1);
      break;
   case 0:
      /* PROCESSUS FILS */
      printf("Processus fils : pid = %d\n", getpid() );
      exit(0); /* fin du processus fils */
      break;
   default:
      /* PROCESSUS PERE */
      printf("Ici le pere: le fils a un pid=%d\n", pid
);
      wait(0); /* attente de la fin du fils */
      printf("Fin du pere.\n");
   }
}
```

e) La primitive wait()

```
#include <sys/types.h>
#include <sys/wait.h>
int wait( int *st );
```

L'appel wait() permet à un processus d'attendre la fin de l'un de ses fils.
Si le processus n'a pas de fils ou si une erreur se produit, wait() retourne -1.
Sinon, wait() bloque jusqu'à la fin de l'un des fils, et elle retourne son PID.
L'argument st doit être nul.

f) La primitive sleep()

```
#include <unistd.h>
int sleep( int seconds );
```

L'appel sleep() est similaire à la commande Shell sleep. Le processus qui appelle sleep est bloqué durant le nombre de secondes spécifié, sauf s'il reçoit entre temps un signal.

Notons que l'effet de sleep() est très différent de celui de wait() : wait bloque jusqu'à ce qu'une condition précise soit vérifiée (la mort d'un fils), alors que sleep attend pendant une durée fixée. sleep ne doit jamais être utilisé pour tenter de synchroniser deux processus.

VIII/ Communications inter-processus

Unix fournit plusieurs mécanismes permettant à des processus s'exécutant sur la même machine d'échanger de l'information. Nous étudions dans cette partie les signaux, les tubes et les sémaphores.
Nous n'aborderons pas ici les procédures de communication entre systèmes distants, surtout basés sur le protocole TCP/IP.

1. Signaux asynchrones

Les signaux permettent d'avertir simplement un processus qu'un événement est arrivé.

Il existe environ 32 signaux prédéfinis, ayant chacun une signification. Il n'est pas possible de définir de nouveaux signaux. Les signaux sont envoyés soit par le système lui même (en particulier lorsque le processus tente une opération illégale comme une division par zéro ou un accès à une zone mémoire ne lui appartenant pas), soit par un autre processus appartenant au même utilisateur.

Certains signaux, comme le numéro 9, entraînent toujours la mort du processus qui les reçoit. Pour d'autres au contraire, le processus peut spécifier une fonction traitante (handler) qui sera automatiquement appelée par le système lors de la réception d'un signal particulier.

Depuis le Shell, on peut envoyer un signal à un processus en utilisant la commande kill. Son usage le plus courant est sans doute de provoquer la fin d'un processus (le tuer), grâce au signal 9.

Voici quelques signaux importants définis par tous les systèmes Unix (la liste complète est définie sur chaque système dans le fichier include <signal.h>.

Numéro	Nom	Signification
1	HUP	fin de session
8	FPE	exception calcul virgule flottante
9	KILL	fin du processus (non modifiable)
10	USR1	définissable par l'utilisateur
11	SEGV	référence mémoire invalide
12	USR2	définissable par l'utilisateur
17	CHLD	terminaison d'un fils
18	CONT	reprise d'un processus stoppé
19	STOP	stoppe le processus
29	WINCH	redimensionnement de fenêtre

Manipulation des signaux en langage C

On peut envoyer un signal à un autre processus en appelant la primitive

```
int kill( int pid, int signum )
```

qui envoie le signal de numéro signum au processus de PID pid.

Pour spécifier qu'une fonction C doit être appelée lors de la réception d'un signal non mortel, on utilise

```
void (*signal(int signum, (void *handler)(int))))(int);
```

L'argument handler est une fonction C qui prend un entier en argument et sera automatiquement appelée lors de la réception du signal par le processus.

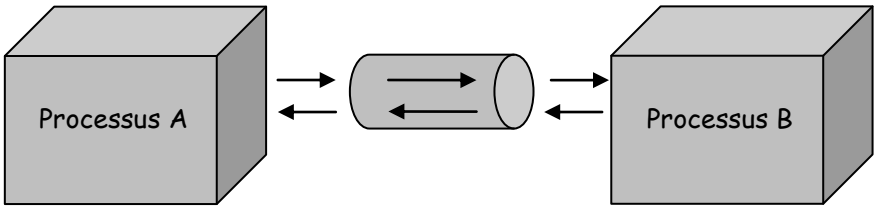
Voici un exemple de fonction traitante :

```
void traite_signal_usr1( int signum ) {
printf("signal %d recu.\n", signum );
}
```

Cette fonction est installée par l'appel :

```
signal( SIGUSR1, traite_signal_usr1 );
```

Tube de communication bidirectionnelle entre deux processus A et B.



Notons que dans certaines versions d'Unix, le traitant doit être réinstallé à chaque fois que le signal est reçu (pour cela, appeler signal() dans la fonction traitante).

On peut aussi spécifier que l'on désire ignorer un type de signal en utilisant la constante SIG_IGN :

```
signal( numero_signal_a_ignorer, SIG_IGN );
```

2. Les tubes

Les tubes permettent l'échange bidirectionnel de données entre deux processus s'exécutant sur la même machine. Le Shell permet de manipuler très facilement des chaînes de processus reliés par des tubes.

a) Lancement d'une commande avec popen()

Une première utilisation très pratique des tubes est le lancement d'une commande par un programme C. Nous avons vu plus haut que l'appel system() lançait une commande ; cependant il n'est alors pas possible de récupérer directement la sortie de la commande.

La fonction popen() lance un processus exécutant la commande voulue et relie ce processus à l'appelant par un tube unidirectionnel. Les deux processus s'exécutent alors en parallèle.

```
FILE *popen( char *command, char *mode );
int pclose( FILE *stream );
```

Le paramètre mode indique le sens du tube ("w" en écriture, "r" en lecture).

La commande est habituellement un filtre. Par exemple, supposons que nous voulions compter le nombre de ligne dans un fichier texte nommé texte.txt". Une première solution est d'ouvrir le fichier avec fopen(), de le lire en comptant le caractères fin de ligne jusqu'à arriver au bout, puis de le refermer. Une autre solution, plus simple, est d'utiliser la commande existante wc, comme suit :

```
char res[256]; /* buffer pour resultat */
FILE *df = popen( "wc -l texte.txt", "r" );
fgets( res, 255, df ); /* lecture du resultat */
pclose( df ); /* fermeture du tube */
int nblignes = atoi( res ); /* conversion en entier */
```

b) Création d'un tube avec pipe()

Il est parfois nécessaire d'échanger de l'information dans les deux sens entre deux processus (modèle client/serveur, etc.).

En langage C, l'appel système pipe() permet la création d'un nouveau tube. Le tube est alors représenté par un couple de descripteurs, l'un pour écrire, l'autre pour lire.

```
#include <unistd.h>
int pipe( int filedes[2]);
```

En cas d'erreur, la valeur retournée est différente de zéro. pipe() crée une paire de descripteurs et les place dans le tableau filedes[]. filedes[0] est utilisé pour lire dans le tube, et filedes[1] pour y écrire.

Pour l'instant, il n'y a personne à l'autre extrémité du tube. C'est pourquoi l'appel à pipe() est généralement suivi par la création d'un processus fils avec lequel on va communiquer.

Dans ce contexte, l'appel système dup2() est très utile.

```
int dup2( int newfd, int oldfd );
```

dup2() ferme le descripteur oldfd (s'il est ouvert) puis le remplace par le descripteur newfd.

L'appel à dup2() permet par exemple de remplacer l'entrée standard d'un processus par la sortie d'un tube existant.

3. <u>Synchronisation : sémaphores</u>

Les sémaphores sont des outils de synchronisation très pratiques pour gérer l'accès concurrent (c'est à dire simultané) de plusieurs processus à une ressource critique (non partageable, par exemple un périphérique). Unix système V à introduit une implémentation très complète des sémaphores, qui est maintenant disponible dans presque (?) toutes les versions POSIX.

Nous allons détailler les appels systèmes permettant de manipuler des sémaphores en langage C. Nous n'expliquons ici que le minimum nécessaire à l'utilisation de base.

```
#include <sys/sem.h>
int semget( key_t key, int nbsems, int flags );
```

semget() est utilisé soit pour créer un nouveau sémaphore, soit pour récupérer l'identificateur d'un sémaphore existant. Retourne un identificateur (nombre entier utilisé dans les autres primitives), ou -1 en cas d'erreur.

Le paramètre key est un entier qui est une sorte de nom du sémaphore, permettant à tous les processus de le repérer. On l'obtient à l'aide des appels getcwd() et ftok() que nous ne détaillons pas ici.

Le paramètre flags donne les droits sur le sémaphore. S'il comporte la valeur IPC_CREAT, un nouveau sémaphore est créé ; sinon, un sémaphore récupère un sémaphore déjà existant.

int semctl(int semid, int num, int cmd, arg)
permet de changer la valeur du sémaphore identifié par semid. Utilisé pour donner la valeur initiale du sémaphore après sa création.

int semop(int semid, sops, nops)
applique une opération sur le sémaphore (augmente ou baisse la valeur).
Si la valeur devient négative, met le processus en attente.

4. Autres techniques de communication

Le système Unix fournit aussi d'autres moyens de communication interprocessus que nous n'étudierons pas ici. Nous mentionnerons simplement les segments de mémoire partagée (deux processus sur la même machine peuvent partager une partie de leurs données), et les sockets (utilisés surtout pour les communications distantes avec TCP/IP).

IX/ Lexique pour comprendre le manuel Unix

Le manuel en ligne d'UNIX est la principale source d'information sur les commandes du système. Il est rédigé dans un anglais technique très simple qui devrait vous être accessible sans efforts après acclimatation (de nombreuses pages sont maintenant traduites). Le lexique ci-dessous défini les termes informatiques les plus couramment utilisés dans ce manuel.

blank vide (ligne vide, écran vide).
by default par défaut, ce qui se passe si l'on ne spécifie rien d'autre.
byte octet (8 bits).
comma virgule.
concatenate "concaténer", c'est à dire mettre bout à bout.
directory répertoire.
file fichier.
handler traitant (d'interruption, de signal, ...), c'est à dire fonction gérant une situation spécifique.
head tête (début).
inhibit inhiber (empêcher, interdire).
job travail, tâche (processus contrôlé par un Shell).
key touche (littéralement "clé").
parse analyser la syntaxe.
path chemin d'accès à un fichier ou répertoire.
pipe tube (communication interprocessus).
prompt demander (à l'utilisateur) ; ce qui s'affiche au début des lignes d'un interpréteur de commandes.
quote guillemet ' ; double quote = "; backquote = `.
recursive récursif.
return retourner (d'une fonction) ; touche "entrée".
shell littéralement "coquille"; désigne sous UNIX un interpréteur de commandes interactif.
skip sauter.
sort trier.
space espace.
status état.
syntax syntaxe.
tail queue (fin).