Chapitre 2

Introduction au système Unix/Linux

Le système **Unics** (*Uniplexed Information and Computing Service*) a été créé aux laboratoires AT&T de Bell, en 1969 par Ken Thompson, et modifié et baptisé par Brian Kernighan [Tan01]. Il a été une version réduite de **Multics** (*Multiplexed Information and Computing Service*). Peu après on a changé le nom Unics par Unix, et à partir de ce moment le système Unix a commencé un long chemin de développement technique.

2.1 Bref historique

Le système Unix a connu un véritable succès, lorsqu'il fut récrit en langage *C* en 1973 par Denis Ritchie et Thompson[1]. L'Université de Californie à Berkeley a obtenu une copie de la version 6 du système Unix. AT&T et Berkeley ont séparément apporté de nombreuses modifications et améliorations au système Unix (**System V** d'AT&T et **4.4BSD** de Bekerley). Une vision simplifiée de l'évolution subie par Unix est montrée sur la figure 2.1. Le projet Posix (*Portable Operating System UnIX*) de normalisation du système Unix a permis de développer un standard qui définit un ensemble de procédures. Tout système Unix conforme à Posix fournit ces procédures ou appels système standards. Ces procédures constituent la bibliothèque standard d'Unix (figure 2.2). Tout logiciel écrit en n'utilisant uniquement les procédures de la norme Posix devrait fonctionner sur tous les systèmes

[1]Le langage *C* de Ritchie provient du langage *B* écrit par K. Thompson, et *B* à son tour de BCPL. *C* a été aussi standardisé par ANSI.

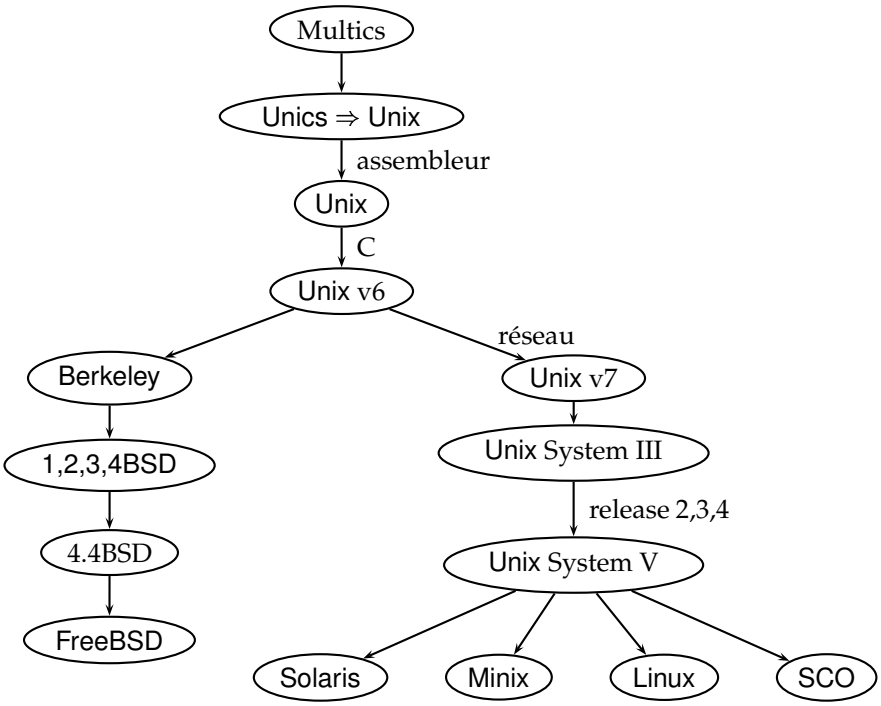

FIG. 2.1 – Evolution d'Unix.

Unix conformes à cette norme. Une version gratuite d'Unix porte le nom de

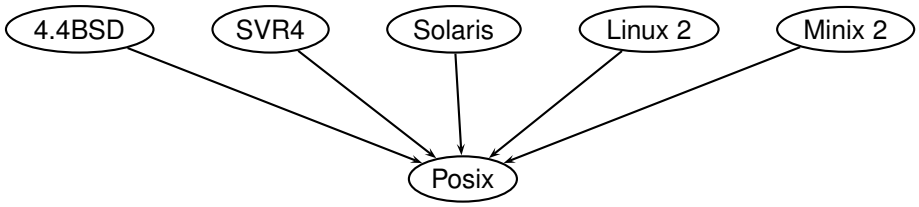

FIG. 2.2 – Standard Posix

Linux (code source disponible). Elle a été créée par Linus Torvalds en 1991. Par la suite, un grand nombre de programmeurs ont contribué à son développement accéléré. Conçu d'abord pour tourner sur des machines avec le processeur 80x86, Linux a migré à plusieurs autres plate-formes. Le système d'exploitation Linux peut être téléchargé à partir de plusieurs sites, par exemple : `http ://www.linux.org`. Dans l'annexe C le lecteur interessé trouvera de plusieurs sites internet où l'on peut télécharger d'autres versiones de Linux.

FreeBSD est une autre version gratuite d'Unix. Il est un systeme d'exploitation pour x86 compatibles, DEC Alpha, et architectures PC-98. Il est dérivé de BSD Unix et développé par une grande communauté d'individus (`http ://www.freebsd.org`).

2.2 Caractéristiques

Unix est un système interactif et multi-utilisateurs (multiprogrammé en temps partagé). Plusieurs utilisateurs peuvent disposer, en même temps, de la puissance de calcul du système. Le système Unix se charge de contrôler et de gérer l'usage des ressources en les attribuant, à tour de rôle, aux différents utilisateurs. Il permet la création, la communication et la synchronisation des processus. Unix est un système d'exploitation ouvert, portable et disponible sur différentes plate-formes. Linux est en plus gratuit et on a le droit d'étudier et de modifier le code source.

2.2.1 Structure d'un système Unix/Linux

Un système informatique sous Unix/Linux est constitué de couches de logiciels, comme illustré à la figure 2.3.

Le système d'exploitation, appelé **noyau** ou *kernel*, gère le matériel et fournit aux programmes une interface d'appels système. Le kernel de BSD est montré à la figure 2.4. Les appels système permettent aux programmes de créer et de gérer des processus et des fichiers. A chaque appel système correspond une procédure de bibliothèque que l'utilisateur peut appeler (bibliothèque standard). Chaque procédure se charge de placer les paramètres de l'appel système correspondant en un endroit prédéfini comme les registres du processeur, et de provoquer une **interruption logicielle** (instruction `TRAP`) pour passer du **mode utilisateur** au **mode noyau** et activer ainsi le système d'exploitation.

La procédure de bibliothèque a pour but de masquer les détails de l'instruction `TRAP` et de faire apparaître les appels de procédure comme des appels de procédures ordinaires. Par exemple `read(f, b, 50)` sera utilisé pour lire `50` caractères dans un stockage temporaire ou *buffer* `b`, à partir d'un fichier `f` présentement ouvert.

Lorsque le système d'exploitation prend le contrôle suite au `TRAP`, il vérifie la validité des paramètres et effectue dans ce cas le traitement demandé. A la fin du traitement, il place un code de statut, indiquant si le traitement a réussi ou échoué, dans un registre, puis redonne le contrôle à

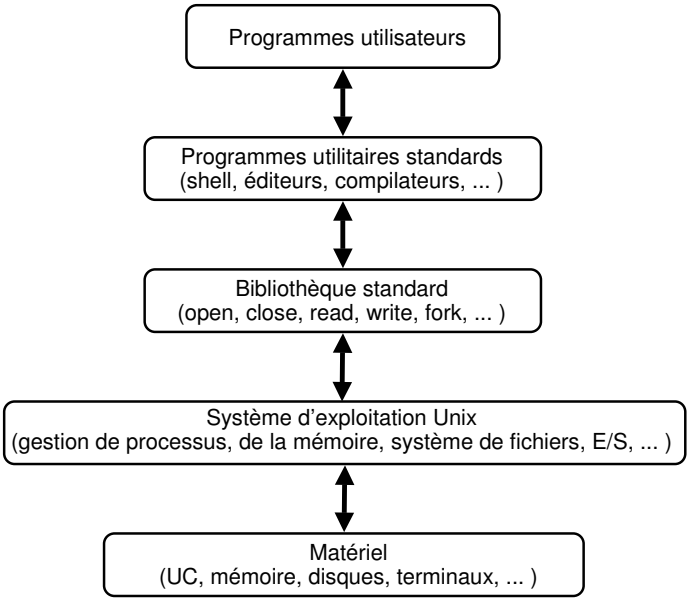

FIG. 2.3 – Architecture d'un système Unix.

la procédure de bibliothèque avec retour au mode utilisateur. La procédure de bibliothèque remet, à son tour, le contrôle à l'appelant en lui transmettant le code de statut. Les utilisateurs peuvent développer des programmes et invoquer des appels système, grâce la bibliothèque standard et à un ensemble d'utilitaires (shells, éditeurs, compilateurs, etc.) fournis avec le système Unix/Linux. Par exemple, pour exécuter l'appel système `READ`, un programme *C/C++* peut appeler la procédure de la bibliothèque standard `read()` qui va effectuer le véritable appel système `READ`.

Remarque : Les véritables appels système s'exécutent en **mode noyau**, ce qui assure la protection du système d'exploitation contre les tentatives d'intrusions et les erreurs.

2.3 Début de session

Lorsqu'un utilisateur demande à se connecter, le système l'invite à introduire son nom d'utilisateur (code) et son mot de passe. Si ces données sont correctes, le système ouvre une session de travail et lance l'**interpréteur de commandes** (processus shell) qui affiche à l'écran, aussitôt après son

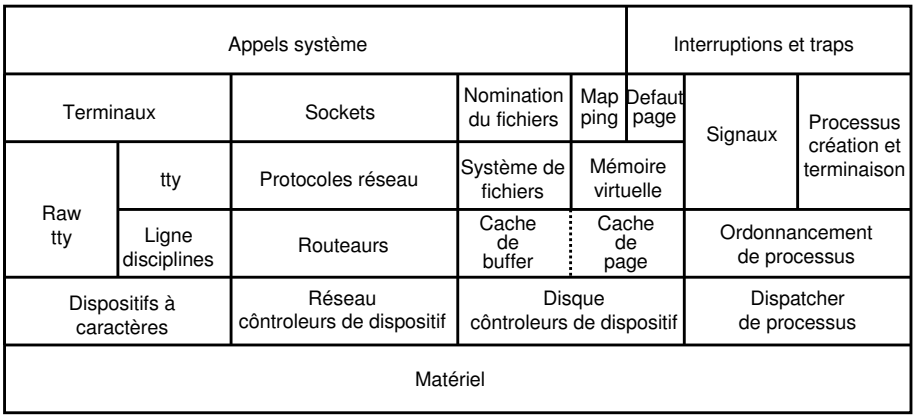

FIG. 2.4 – Architecture du kernel BSD d'Unix.

initialisation, une invitation puis se met en attente d'ordres de l'utilisateur. Dépendamment du shell utilisé, ceci peut être un simple symbole :

```
$
```

ou bien, l'invitation peut montrer le nom de la machine. Dans notre cas l'une des machines s'appelle `leibnitz` :

```
leibnitz>
```

Lorsque l'utilisateur introduit une commande, le shell vérifie si elle est correcte, puis crée, si c'est le cas, un processus qui a pour tâche d'exécuter la commande. Par exemple la commande :

```
leibnitz>cp source destination
```

copiera le fichier `source` dans le fichier `destinantion`. Le shell attend la terminaison du processus créé avant d'inviter l'utilisateur à introduire une nouvelle commande.

2.3.1 Interpréteurs de commandes

Il existe plusieurs interpréteurs de commandes ou shell populaires pour le système Unix. Parmi eux on trouve :

- Le shell de Bourne : `sh`
- Le shell de Korn : `ksh`
- Le C shell : `csh`

Le shell de Korn englobe celui de Bourne. Bon nombre de commandes sont communes aux trois shell's.

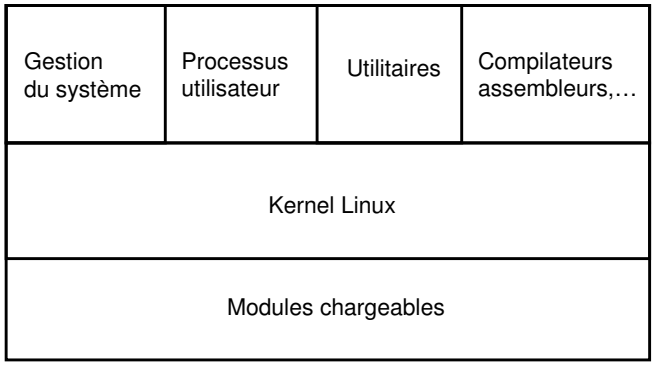

FIG. 2.5 – Architecture de Linux.

Le shell `bash`, écrit par Brian Fox, fait partie du projet **GNU**, acronyme récursif de **G**NU's **N**ot **U**nix, démarré par Richard Stallman en 1984[2]. Le projet GNU voulait créer un système complet et compatible Unix mais distribué librement. GNU avait développé plusieurs programmes comme le compilateur `gcc`, l'éditeur `emacs`, `make`, `tar`, l'interface graphique X-Windows. En 1991 le kernel Linux a rejoint le projet et on a appelé le tout **GNU/Linux**. Le shell `bash` (acronyme de *Bourne-Again SHell*) est compatible avec le standard Posix, et compatible aussi avec le shell `sh`. Il incorpore beaucoup des caractéristiques du Korn shell `ksh` et le C shell `csh`.

2.4 Commandes utiles d'Unix/Linux

Dans la table 2.1 nous montrons quelques commandes Posix très utilisés dans Unix/Linux.

`man`

```
man [-s section] mot
```

La commande `man` donne une aide en ligne. Elle affiche les informations correspondantes au titre donné, figurant dans le manuel standard de Linux. Ce manuel est composé de 8 sections.

▶ **Exemple 1.** La commande :

```
leibnitz> man chmod
```

---

[2] http ://www.gnu.org

Commande	Description
who	Afficher la liste des utilisateurs connectés
whoami	Afficher l'utilisateur de la session courante
date	Afficher la date
ps	Afficher la liste des processus de l'utilisateur
kill	Stopper un processus
passwd	Créer ou changer de mot de passe
pwd	Afficher le nom du répertoire de travail
mkdir	Créer un répertoire
cd	Changer de répertoire de travail
cat	Fusionner une liste de fichiers et afficher le résultat
head	Afficher le début d'un fichier
grep	Afficher les lignes des fichiers référencés qui contiennent une chaîne de caractères donnée
wc	Compter le nombre de mots, lignes ou caracteres.
sleep	Dormir pendant un certain temps (secondes)
find	Rechercher un fichier
ls	Afficher le contenu d'un répertoire
cp	Copier un fichier dans un autre
ln	Ajouter à un fichier existant un autre nom
mv	Renommer un fichier
rm	Supprimer un fichier
rmdir	Supprimer un répertoire
chmod	Changer les protections d'un fichier

TAB. 2.1 – Commandes Posix d'Unix.

sélectionne la section `1` des commandes, et :

```
leibnitz> man -s 2 chmod
```

sélectionne la section 2 des appels système. Évidemment si l'on veut comprendre comment `man` fonctionne, il faut taper :

```
leibnitz> man man
```

`sort`

La commande `sort` réalise la lecture des lignes provenant du terminal (jusqu'à ce que l'utilisateur tape `CRTL-D` indiquant la fin de fichier), trie les lignes par ordre alphabétique et affiche le résultat à l'écran.

▶ **Exemple 2.** Utilisation de `sort`

```
leibnitz> sort
Martin 16 HULL
Robert 17 Ottawa
Catherine 15 Montréal
^D
Catherine 15 Montréal
Martin 16 HULL
Robert 17 Ottawa
leibnitz>
```

Redirection des entrées/sorties

A chaque processus créé (commande ou programme en exécution) sont associées une entrée et deux sorties standards :

- L'entrée standard : le clavier
- La sortie standard : l'écran du terminal
- La sortie erreur standard : l'écran du terminal

Le shell permet de modifier ou faire la **redirection** des Entrées/Sorties d'un processus.

Redirection de l'entrée standard

On fait la redirection avec l'opérateur < :

```
commande < fichier
```

▶ **Exemple 3.** Cette commande trie le fichier `source`, puis affiche le résultat à l'écran :

```
leibnitz> sort < source
```

Redirection de la sortie standard

On peut rediriger la sortie standard avec l'opérateur > ou » :

```
commande > nom_fichier
commande >> nom_fichier
```

▶ **Exemple 4.** La liste des utilisateurs connectés au système est récupérée dans le fichier `users` avec la commande suivante. Si le fichier existe déjà, il y aura une erreur :

```
leibnitz> who > users
```

Si le fichier existe déjà, la liste est insérée à la fin :

```
leibnitz> who >> users
```

▶ **Exemple 5.** Cette commande lit les lignes provenant du terminal puis les recopie dans le fichier `dest` (jusqu'à ce que l'utilisateur tape `CRTL-D`, pour indiquer la fin de fichier) :

```
leibnitz> cat > dest
```

Redirection des sorties standards (erreur et normal)

```
(commande) >& nomfich
(commande) >> & nomfich
```

▶ **Exemple 6.** Redirection des sorties.

```
leibnitz> (ls ggg)>& er
leibnitz> cat er
ggg: Ce fichier ou ce répertoire n'existe pas
leibnitz> (lhiumo ) >>& er
leibnitz> cat er
ggg: Ce fichier ou ce répertoire n'existe pas
lhiumo : Commande introuvable
leibnitz>
```

Enchaînement séquentiel des commandes

Le shell permet de lancer plusieurs commandes (processus) de façon séquentielle avec l'opérateur **;** point-virgule :

```
commande_1 ; commande_2 ; .... ; commande_n
```

▶ **Exemple 7.** Cette commande permet d'afficher le répertoire de travail puis son contenu. Après elle va créer le répertoire (`cours`) dans le répertoire de travail. Elle affiche le contenu du répertoire de travail, ensuite elle change au répertoire de travail (`cours`), affiche le nouveau répertoire de travail (`cours`), puis elle revient au répertoire père, et finalement affiche le nouveau répertoire de travail :

```
leibnitz> pwd;ls;mkdir cours;ls;cd cours;pwd;cd ..;pwd
```

Enchaînement parallèle des commandes

De façon similaire, on peut lancer plusieurs commandes en parallèle avec l'opérateur `|` et de diriger la sortie de l'un vers l'entrée d'un autre. On appelle ceci **pipelines** :

```
commande_1 | commande_2 | ... | commande_n
```

▶ **Exemple 8.** La commande suivante (voir figure 2.6) crée deux processus `cat` et `sort`. `cat` fusionne les fichiers `src1` et `src2` et envoie le résultat à `sort`; `sort` trie les données reçues avant de les stocker dans le fichier `dest`. Les deux processus s'exécutent en parallèle :

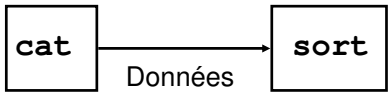

FIG. 2.6 – Enchaînement parallèle des commandes cat et sort.

```
leibnitz> cat src1 src2 | sort > dest
```

▶ **Exemple 9.** Utilisation enchaînée.

```
leibnitz> grep ter *.t|sort|head -20|tail -5 >fich
```

La commande `grep` envoie à `sort` les lignes contenant la chaîne « `ter` » de tous les fichiers se terminant par `.t`, puis `sort` trie les lignes reçues et les transmet à `head`, qui à son tour envoie les 20 premières à `tail`. Finalement, la commande `tail` écrit les cinq dernières dans le fichier `fich`.

▶ **Exemple 10.** Les sorties (erreur ou non) de la compilation/édition de liens du programme sont re-dirigées vers l'entrée du processus `more`. Ce dernier affiche à l'écran les résultats progressivement :

```
leibnitz> gcc prog.c | & more
```

Tâches de fond

Le shell permet de lancer une `commande` en tâche de fond en utilisant l'opérateur `&`. Dans ce cas, il n'attend pas la fin de l'exécution de la `commande` :

▶ **Exemple 11.** Cette commande fera la recherche récursive à partir du répertoire racine de tous les fichiers de nom `bin`. Les chemins d'accès absolus des fichiers trouvés sont stockés dans le fichier `result` :

```
leibnitz> find / -name bin -print >result &
```

2.5 Scripts shell

Chaque shell d'Unix a son propre langage de programmation conçu pour manipuler des fichiers et des processus. Ce langage comporte des structures de contrôle semblables à celles des langages de programmation classiques. Les programmes shell sont appelés **script shell**. Dans l'Annexe A vous trouverez une liste complète des commandes et des opérateurs `test [ ]` de `if`.

Syntaxe de certains commandes shell

Le shell peut effectuer des commandes répétitives sous forme de boucles `for` ou `while`, ainsi que prendre des décision avec `if` ou `case` :

- Boucle `for` :
  `for name [in word] do list; done`
- Boucle `while` :
  `while list; do list; done`
- Décision `case` :
  `case mot in [(pattern[|pattern]...) liste;;]... esac`
- Décision `if` :
  `if liste; then liste; [elif liste; then liste]... fi`

Autorisation de l'exécution du script

Il faut donner au script le mode adéquat pour que l'interpréteur de commandes puisse l'exécuter.

▶ **Exemple 12.** On peut créer un fichier `script0` :

```
leibnitz> cat > script0
for NAME in Jill Richard Sam ; do
   echo "Hola $NAME";
done
^D
leibnitz>
```

puis on peut changer le mode à `u+x` pour le rendre exécutable :

```
leibnitz> ls -l script0
-rw------- 1 jmtorres professeur 326 août 27 19:32 script0
leibnitz> chmod u+x script0
leibnitz> ls -l script0
-rwx------ 1 jmtorres professeur 326 août 27 19:32 script0
leibnitz>
```

Commentaires

Les scripts shell peuvent avoir des commentaires s'ils sont précédés du signe #

```
#*****************************************************
# Fichier : script.sh
# Auteur(s) : Juan Manuel Torres
# Cours : Systèmes d'exploitation
# École Polytechnique de Montréal
#*****************************************************
```

En particulier le *commentaire* `#!/bin/bash` indique au shell qu'on veut utiliser `bash` pour l'exécution de nôtre script.

Variables

Les variables peuvent être utilisées comme dans d'autres langages de programmation. L'assignation de la valeur à une variable est très simple. Par exemple, l'instruction :

```
$ REP=repertoire
```

stocke le mot `repertoire` dans a variable `REP`. Conventionnelement tous les noms de variables doivent être en majuscules, mais ceci n'est pas impossé par le shell. Une fois la variable définie, on peut l'utiliser librement :

```
$ REP=repertoire
$ echo $REP
$ repertoire
$ date > $REP
$ cat $REP
$ Wed Feb 19 16:01:13 EST 2003
```

Pour supprimer une variable, il suffit d'utiliser la commande `unset` :

```
$ REP=repertoire
$ echo $REP
$ repertoire
$ unset REP
$ echo $REP
$
```

Les paramètres de commandes

Les arguments d'une procédure sont récupérés dans les variables `$1`, `$2`, `...`, `$9`. `$0` garde le nom utilisé pour appeler la commande. Le nombre effectif d'arguments est donné par `$#`. `$*` et `$@` donnent la liste de tous les arguments.

Lecture interactive

La commande `read` permet de lire une ligne de l'entrée standard et d'affecter les valeurs des variables dont les noms sont donnés en arguments, à raison d'un mot par variable, la dernière variable se voyant affectée tout le reste de la ligne (s'il reste des mots !).

▶ **Exemple 13.**

Listing 2.1 – read.sh

```
#!/bin/bash

read a b c
echo "Le premier mot : " $a
echo "Le èdeuxime mot : " $b
echo "Le reste de la ligne est : " $c
```

`read` est une commande qui rend faux quand la fin du fichier est rencontrée. On peut ainsi s'en servir pour lire un fichier en totalité à l'aide d'une boucle `while`.

▶ **Exemple 14.**

Listing 2.2 – mon-cat.sh

```
#/bin/bash

while read ligne;
do
   echo "$ligne"
done
```

▶ **Exemple 15.** Ce script montre le message `"Hola $NOM"` pour chacun des noms de la liste {`Patricia`, `Richard`, `Jean`} :

Listing 2.3 – hola.sh

```
#!/bin/bash

for NOM in Patricia Richard Jean ; do
   echo "Hola $NAME";
done
```

▶ **Exemple 16.** Ce script montre la liste de fichiers qui commencent par `hda*` dans le répertoire `/dev/` :

Listing 2.4 – hda.sh

```
#!/bin/bash
```

```
for FILENAME in /dev/hda*; do
  echo "J'ai étrouv le fichier $FILENAME"
done
```

▶ **Exemple 17.** La boucle `while` permet d'évaluer des expressions :

Listing 2.5 – while.sh

```
#!/bin/bash

i=1
while true do
  echo $i
  i=$(expr $i + 1)
  if [ $i == 6 ]
  then
     exit 0
  fi
done
```

Ce script donnera comme résultat :

```
leibnitz>
1
2
3
4
5
leibnitz>
```

▶ **Exemple 18.** Ce script trouve la liste de rapports textes de 1997 à 1999 qui contiennent le mot `Linux`. Ensuite, pour tous ces fichiers, il affiche le nom en traitement, puis il copie les 20 dernières lignes dans `TEMP`, finalement il envoie, par `courriel` avec un sujet adéquat le fichier `TEMP` :

Listing 2.6 – rapport.sh

```
#!/bin/bash

for FILE in 'grep -l Linux Rapport-199[7-9].txt'; do
 echo "Traitement de $FILE"
 tail -20 $FILE > TEMP
 mail -s "20 lignes de $FILE" juan-manuel.torres@polymtl.ca
       < TEMP
```

```
done
```

▶ **Exemple 19.** Ce script renvoi le texte inversé tapé par l'utilisateur. Le `-n` de `echo` sert à ne pas sauter à la ligne suivante après l'affichage du message :

Listing 2.7 – inv.sh

```
#!/bin/bash

echo "Tapez du texte ; Ctl-D fin"
echo -n "entre: "
while read; do
  TEXT=`echo "$REPLY" | rev`
  echo "Inverse : $TEXT"
  echo -n "Votre entree: "
done
```

▶ **Exemple 20.** Test de type de fichier. Le script `affichage.sh` récupère les fichiers du répertoire courant (`set `ls``) dans les variables `$i`. La variable `$0` contient le nom du premier fichier, `$1` le nom du second et ainsi de suite. Il parcourt les noms des fichiers (`for FICH in $*`). Pour chaque nom, il teste si c'est un répertoire (`if [ -d $FICH ]`). Si c'est le cas, il affiche le nom du répertoire suivi du texte `'est un répertoire'` (`echo "$FICH est un répertoire"`). Dans le cas contraire, il vérifie s'il s'agit d'un fichier normal (`if [ -f $FICH ]`). En cas de succès, il affiche un message puis propose d'afficher le contenu du fichier `$FICH` :

Listing 2.8 – affichage.sh

```
#!/bin/bash

set `ls`
for FILENAME in $*; do
  if [ -d $FILENAME ]; then
      echo "J'ai trouve le repertoire $FILENAME"
  elif [ -f $FILENAME ]; then
      echo "$FILENAME est un fichier. Affichage ? (o | n)"
      read rep
      case $rep in
      o | O) cat $FILENAME ;;
      n | N) echo "Pas de visualisation de $FILENAME" ;;
      *) echo "Reponse incorrecte"
```

```
      esac
  fi
done
```

Exécution du programme `affichage.sh` :

```
leibnitz> cat fich
bonjour ici fich
a bientôt
leibnitz>
leibnitz> affichage.sh
fichier trouve. Affichage ? (o ou n)
o
bonjour ici fich
a bientôt
leibnitz>
```

Petite arithmétique shell

Dans la table 2.2 nous montrons un petit aperçu des capacités arithmétiques de `bash`.

Opérateurs	Opération
+, −, *	Addition, différence, multiplication
/, %	division, reste
==, !=, <=, >=	égal, différent, inférieur ou égal, supérieur ou égal
<, >	Comparaisons
>>, <<	Décalages
!, ~	Négation logique et de bits
&, |	AND, OR
&&, ||	AND, OR à niveau de bits

TAB. 2.2 – Capacités arithmétiques `bash`.

▶ **Exemple 21.** Opérations arithmétiques :

```
leibnitz> bash
bash-2.05a$ v1=5
bash-2.05a$ v2=7
bash-2.05a$ v3=$v1+$v2
bash-2.05a$ v4=$((v1+v2))
```

```
bash-2.05a$ echo $v1
5
bash-2.05a$ echo $v2
7
bash-2.05a$ echo $v3
5+7
bash-2.05a$ echo $v4
12
bash-2.05a$ echo "$v3=$v4"
5+7=12
```

Il existe aussi une autre commande `expr` qui permet d'évaluer une expression arithmétique composée de constantes et de variables shell.

▶ **Exemple 22.** Utilisation d'`expr`.

```
i=1
i=$(expr $i + 1)
echo "i=${i}"
i=$((i+1))
echo "i=${i}"
```

aura comme résultat :

```
a=2
a=3
```

2.6 Editeurs d'Unix

Plusieurs éditeurs fonctionnent sous Unix : `emacs`, `vi`, `xedit`, `pico` et `CDE Text Editor` parmi d'autres. `vi` est l'éditeur standard sur la majorité des systèmes Unix. Il a comme vertu de fonctionner là où d'autres éditeurs échouent. Malgré le fait qu'il n'est pas très convivial, il est très performant et permet de faire l'édition de très gros fichiers. `vi` utilise deux modes de fonctionnement : le **mode insertion** utilisé pour entrer du texte ; et le **mode commande** utilisé pour manipuler le fichier.

Pour créer un fichier `nomfich` avec l'éditeur `vi`, il faut :

- Lancer la commande `vi nomfich`
- Appuyer sur la touche `i` pour passer au mode insertion.
- Taper votre texte.
- Appuyer sur `<esc>` pour passer au mode commande.
- Pour la sauvegarde, taper `:w`

– Pour quitter, taper `:q`
– Pour quitter sans modifier, taper `:q!`

Il existe plusieurs clons de `vi`, par exemple : VIM (http ://www.vim.org), stevie (http ://www.scytale.com/waffle/wafiles.shtml), bvi (http ://bvi.sourceforge.net) et bien d'autres, adaptés à plusieurs platte-formes, non seulement Unix. Le lecteur interessé peut consulter le site http ://www.vi-editor.org où c'est bien documenté l'histoire et l'utilisation de cet éditeur.

2.7 Utilisateurs

Chaque **utilisateur du système** Unix/Linux est identifié par un numéro unique appelé `UID` (*user identifier*). Un utilisateur particulier, appelé le **superviseur**, **administrateur** ou `root`, dispose de certains privilèges que les autres n'ont pas. Il peut accéder à tous les fichiers et effectuer certains appels système réservés.

Le système Unix/Linux offre la possibilité de constituer des **groupes d'utilisateurs**. Chaque groupe est identifié par un numéro unique appelé `GID` (*group identifier*). Les `UID` et `GID` d'un utilisateur servent, par exemple, à définir les droits d'accès aux fichiers de l'utilisateur. L'utilisateur propriétaire d'un fichier peut permettre aux membres de son groupe l'accès en lecture au fichier et l'interdire aux autres utilisateurs.

2.8 Fichiers et répertoires

Fichiers

L'accès aux **fichiers** se fait en spécifiant le **chemin d'accès**, qui peut être absolu ou relatif par rapport au répertoire racine. Par exemple, supposons que le répertoire `'cours'` se trouve dans le répertoire racine. Le répertoire `'cours'` contient le répertoire `inf3600` qui, à son tour, contient le fichier introduction. Dans le système Unix/Linux le chemin d'accès absolu au fichier `introduction` est : `/cours/inf3600/introduction`
Si `inf3600` est le répertoire courant, le chemin d'accès relatif est :

```
introduction
```

La commande `cd` d'Unix permet de changer de répertoire :

```
cd /cours
```

Le répertoire courant est maintenant `cours`. Le chemin d'accès relatif au fichier introduction devient :

```
inf3600/introduction
```

Les deux commandes suivantes réalisent chacune l'affichage du contenu du fichier `introduction` :

```
cat inf3600/introduction
cat /cours/inf3600/introduction
```

Chaque fichier a un **propriétaire** (en général le créateur) qui appartient à un groupe d'utilisateurs. Pour contrôler les accès aux fichiers, le système Unix affecte, à chaque fichier, 9 **bits de protection**. Ce code indique les autorisations d'accès en lecture, en écriture et en exécution du fichier pour le propriétaire (3 premiers bits), le groupe du propriétaire (les trois bits suivants) et les autres (les trois derniers bits). Il faut ouvrir un fichier avant de le lire ou d'y écrire. Le système vérifie alors les droits d'accès et retourne, si l'accès est autorisé, un entier, appelé **descripteur de fichier**, qu'il faudra utiliser dans toutes les opérations ultérieures de manipulation du fichier. Si l'accès est refusé, un code d'erreur est retourné. Les appels système liés aux fichiers permettent de créer, d'ouvrir, de lire, de récupérer les caractéristiques (taille, type,...) d'un fichier.

▶ **Exemple 23.** Le code `rwxr-x-x` (ou encore `751`) d'un fichier indique que :

- Le propriétaire peut lire, écrire, et exécuter le fichier (`rwx` ou encore `7`)
- Les membres du groupe peuvent lire et exécuter le fichier, mais ils ne peuvent y écrire (`r-x` ou encore `5`)
- Les autres utilisateurs peuvent exécuter le fichier, mais ils ne peuvent ni le lire ni y écrire (`-x` ou encore `1`)

Lorsqu'on crée un fichier, le système lui attribue par défaut un code de protection. Ce code est calculé en se basant sur la valeur de `umask`. La table 2.3 montre les équivalents décimaux des accès des permissions.

Décimal	Permission
7	rwx
6	rw-
5	r-x
4	r-
3	-wx
2	-w-
1	-x
0	--

TAB. 2.3 – Permissions des fichiers.

▶ **Exemple 24.** Pour connaître la valeur du code de protection il faut taper :

```
leibnitz> umask
```

ou bien :

```
leibnitz> umask 77
```

pour le modifier. Dans le cas de nos machines Linux, le code de protection par défaut est `22`. C'est à dire que les nouveaux fichiers sont crées avec les permissions `0666 & ~ 022 = 0644 = rw-r-r-`

Répertoires

Les fichiers peuvent être regroupés dans des répertoires. Les répertoires peuvent contenir d'autres répertoires, sous une structure arborescente (figure 2.7). Les répertoires sont aussi protégés par un code de 9 bits qui indiquent si les accès en lecture, en écriture et en recherche sont autorisés pour le propriétaire, le groupe et les autres. Le répertoire `" / "` est nommé répertoire racine ou *root*. Il est composé de plusieurs sous-répertoires, comme illustré à la figure 2.7 :

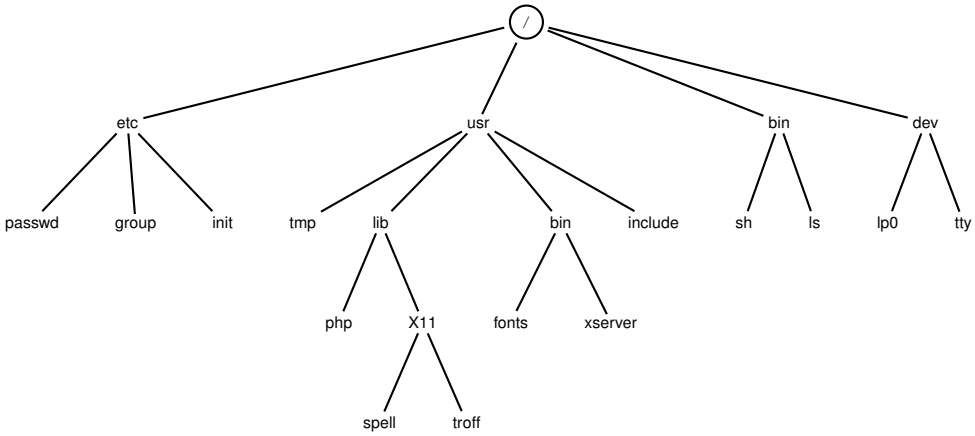

FIG. 2.7 – Aperçu des répertoires dans Unix/Linux.

- `/bin` : contient les commandes utilisateur communes, telles que `ls`, `sort` et `date`
- `/dev` : contient les fichiers représentant les points d'accès aux périphériques de votre système comme `tty`, `fd`, `hd`, `ram`, `cd`, etc.
- `/etc` : contient les commandes et les fichiers d'administration
- `/lib` : contient les librairies partagées ou des liens vers ces librairies

- `/tmp` : contient les fichiers temporaires
- `/usr` ou `/home` : contient les répertoires assignés à chaque utilisateur

2.9 Processus

Un **processus** est un programme qui s'exécute. Tout processus sous Unix/Linux a un espace d'adressage constitué de trois segments : le **texte**, les **données** et la **pile**, comme montré sur la figure 2.8.

- Le segment de **texte** contient les instructions en langage machine (le code) produit par le compilateur, l'éditeur de liens et l'assembleur. Ce segment est inaccessible en écriture. Il peut être partagé entre plusieurs processus. Il est de taille fixe.
- Le segment de **données** contient l'espace de stockage des variables du programme. Il est à son tour composé de deux parties :
  - Le segment de données système appartient à Unix est inaccessible en mode utilisateur. En mode kernel, les appels système y trouvent des informations du type descripteurs de fichiers ouverts, zone de sauvegarde des registres, informations de comptabilité. On ne trouve pas de buffers d'entrée sortie dans ce segment.
  - Le segment de données utilisateur est lui aussi composé de deux parties, contenant les données non initialisées explicitement et celles initialisées (BSS). La taille de cet espace peut augmenter ou diminuer durant l'exécution du processus.
- Le segment de **pile** sert à l'allocation des variables locales, à la sauvegarde des adresses de retour des sous-programmes, à la sauvegarde des registres, etc. Il contient aussi au départ, les variables d'environnement (shell) et la ligne de commande envoyée pour demander l'exécution du programme. Un programme peut connaître tous ses paramètres.

Chaque processus possède un numéro qui lui est propre, le `pid` (en anglais *process identifier*). C'est un numéro unique que l'on appelle l'identifiant de processus. Bien qu'il s'agit d'un chiffre entier de 32 bits, seulement sont utilisés des `pid` entre 0 et 32767, par raisons de compatibilité. Si le nombre de processus dépasse 32767, le noyau doit alors recycler des `pid`. Il existe un appel système qui retourne le `pid` du processus en cours d'exécution :

```
int getpid()
```

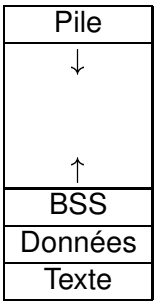

FIG. 2.8 – Espace d'adressage.

`getpid()` retourne le numéro de processus – pid – du processus appelant. La valeur de retour de la fonction est le numéro du processus courant.

Visualisation de processus

Sur Unix/Linux les processus peuvent être affichés au moyen de la commande `ps` :

```
leibnitz> ps
PID TTY          TIME CMD
6100 pts/2    00:00:00 tcsh  --- le shell sur ce terminal
6150 pts/2    00:00:00 ps    --- le programme ps lui même
leibnitz>
```

La commande `pstree` permet de visualiser l'arbre de création de processus :

```
leibnitz> pstree
init-+-acpid
     |-agetty
     |-atd
     |-2*[automount]
     |-bdflush
     |-crond
     |-gpm
     |-identd---identd---5*[identd]
     |-inetd-+-2*[in.ftpd]
     |       |-in.telnetd---tcsh---pstree
     |       `-in.telnetd---tcsh---more
     ...
     |-kswapd
     |-kupdated
```

```
|-lockd
|-lpd
...
|-sendmail
|-snmpd
|-sshd
|-syslogd
`-ypbind---ypbind---2*[ypbind]
```

En particulier on peut observer les démons[3] `init`, `getty` (accès de terminaux), `sendmail` (courrier), `lpd` (imprimante), et `pstree` lui même. Les appels systèmes d'Unix/Linux permettent la création, l'arrêt des processus, la communication et la synchronisation des processus.

Démarrage d'Unix/Linux

Au démarrage d'un système Unix/Linux, un **programme amorce** appelé **boot** est chargé du disque en mémoire grâce à un petit programme, en général câblé, qui passe ensuite le contrôle au programme amorce. Ce dernier détermine les caractéristiques du matériel et effectue un certain nombre d'initialisations. Il crée ensuite le processus `0` qui réalise d'autres initialisations (par exemple celles du système de fichiers) et crée deux processus : `init` de `PID 1` et le démon des pages de `PID 2` (voir figure 2.9).

[3]Les démons sont des processus particuliers qui tournent tout le temps. Ils seront traités dans le Chapitre **??** *Processus et threads.*

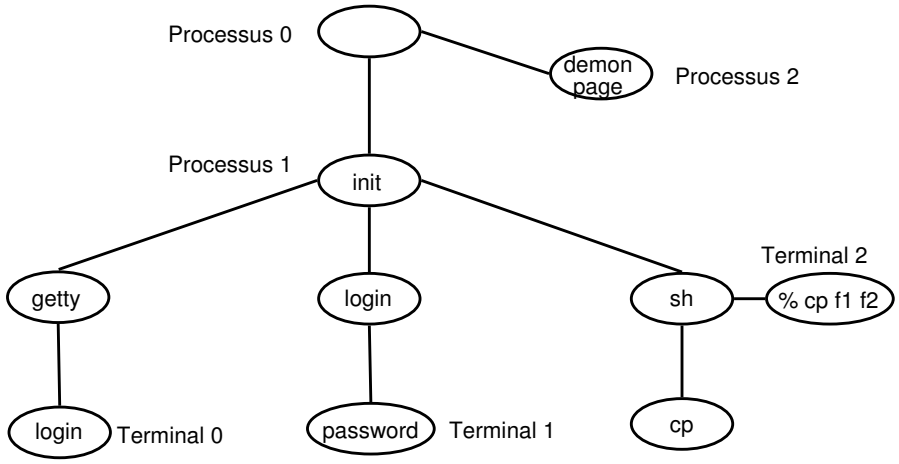

FIG. 2.9 – Processus dans Unix/Linux.

Sugestions de lecture

Référence : Silverwschatz A., Galvin P., Gagné G., *Applied Operating System Concepts*, Wiley, 2003.

Chapître 20

Section 20.1 History.

Section 20.2 Design Principles.

Section 20.3 Programmer interface.

Section 20.4 User interface.

2.10 Exercices

1. Expliquez la différence fondamentale qui existe entre un script et un `makefile`. Peut-on effectuer les opérations d'un `makefile` à l'aide d'un script ?
2. Que fait-il la commande shell :
   `cat < f1 > f2`
3. Donner la commande qui permet de réaliser ce qui suit :
   (a) Afficher les utilisateurs connectés au système.
   (b) Afficher votre code utilisateur.
   (c) Afficher votre login utilisateur.
   (d) Afficher le nom du répertoire courant.
   (e) Afficher le contenu du répertoire courant.
   (f) Créer un répertoire de nom `laboratoire`.
   (g) Afficher le contenu du répertoire courant.
   (h) Changer de répertoire pour passer à celui que vous venez de créer.
   (i) Afficher le nom du répertoire courant.
   (j) Créer un fichier de nom `fich1` en utilisant la commande `cat`. Le contenu du fichier est :

```
#include <iostream.h>

#define long 180
int main( )
{
    char chaine[long];
    cin.getline(chaine,long);
    cout << chaine << endl;
    return 0;
}
```

   (k) Afficher le code de protection du fichier crée.
   (l) Ajouter la permission en exécution pour le propriétaire et le groupe.
   (m) Changer le nom du fichier en prog1.cpp

(n) Compiler le fichier avec la commande :
```
g++ -o prog1 prog1.cpp
```
(o) Exécuter le programme : `prog1`

(p) Supprimer la permission en exécution du programme `prog1.cpp` pour le groupe.

(q) Trouver deux façons de modifier les permissions.

(r) Rediriger la sortie standard du programme vers un fichier nommé `resultat`.

(s) Afficher le contenu du fichier `resultat`.

4. Que fait chacune des commandes suivantes :

(a) `man`

(b) `ls`

(c) `man man`

(d) `ls *.cpp`

(e) `file laboratoire`

(f) `ls | sort`

(g) `ls */*.cpp`

(h) `ls | wc .w`

(i)
```
cat >> prog1.cpp
...
//fin du programme
^D
```

5. Écrire la commande qui permet d'afficher :

(a) les 2 premières lignes du fichier `prog1.cpp`

(b) les 3 dernières lignes du fichier `prog1.cpp`

(c) la 4ième ligne du fichier `prog1.cpp`

6. Éditer avec l'éditeur `vi` un programme *C/C++* nommé `prog2.cpp` qui lit une ligne tapée à l'écran et affiche le nombre de mots dans la ligne suivi de la ligne.

(a) Compiler et exécuter le programme.

(b) Exécuter la commande : `prog1 | prog2`

7. La commande `date` affiche la date en français (anglais). Écrire un script qui traduit la date en anglais (français). La date se présente sous la forme :

   ```
   lundi, 4 septembre 2000, 09 :38 :09 EDT
   ```

8. Écrire un script `mesure` qui constituera un fichier comportant la date et le nombre d'utilisateurs à ce moment. Un pointage sera fait toutes les 10 secondes. La commande sera utilisée de la façon suivante :

   ```
   mesure fichier_log &
   ```

Annexe A

Opérateurs et variables shell

`test` et `[ ]`

Opérateur	Description
! expression	True si l'expression est fausse. Elle peut être utilisée pour inverser n'importe quel autre test
-b fichier	True si le fichier est un périphérique bloc
-c fichier	True si le fichier est un périphérique caractère
-d fichier	True si le fichier est un répertoire
-e fichier	True si le fichier existe
-f fichier	True si le fichier est un fichier normal
-p fichier	True si le fichier est un FIFO
-L fichier	True si le fichier est un lien symbolique
-S fichier	True si le fichier est un socket Unix
-G fichier	True si le fichier appartient au même groupe que le GID effectif du processus courant
-n chaine	True si la longueur de la chaîne $\neq 0$
-z chaine	True si la longueur de la chaîne = 0
-O fichier	True si le fichier appartient à la même personne que l'UID effectif du processus courant
-g fichier	True si le fichier a le bit setgid armé
-r fichier	True si les permissions de l'utilisateur sont suffisantes pour lire le fichier
-k fichier	True si le fichier a le bit sticky armé
-t [fd]	True si le descripteur de fichier correspond à un terminal réel. fd=1 par défaut (sortie standard)

-u fichier	True si le fichier a le bit setuid armé
-w fichier	True si les permissions de l'utilisateur courant sont suffisantes pour écrire dans le fichier spécifié
-x fichier	True si les permissions de l'utilisateur courant sont suffisantes pour exécuter le fichier spécifié
Expression1 -a expression2	True si les deux expressions spécifiées sont vraies
Expression1 -o expression2	True si au moins une des expressions spécifiées est vraie
fichier1 -ef fichier2	True si les deux fichiers correspondent aut même numéro d'inœud sur le même périphérique
fichier1 -nt fichier2	True si la date de dernière modification du fichier1 est plus récente que celle du fichier2
fichier1 -ot fichier2	True si la date de dernière modification du premier fichier est plus ancienne que celle du second
nombre1 -eq nombre2	True si nombre1 = nombre2
nombre1 -ne nombre2	True si nombre1 $\neq$ nombre2
nombre1 -le nombre2	True si nombre1 $\leq$ nombre2
nombre1 -lt nombre2	True si nombre1 $<$ nombre2
nombre1 -ge nombre2	True si nombre1 $\geq$ nombre2
nombre1 -gt nombre2	True si nombre $>$ nombre2
chaine1 = chaine2	True si les deux chaînes sont égales
chaine1 != chaine2	True si les chaînes ne sont pas égales

Variables `bash` spéciales

W = écriture ; R = lecture

Variable	Accès	Description
!	R	ID du processus en arrière-plan le plus récent
@	R	Contient tous les paramètres du contexte courant Lorsqu'elle est utilisée entre des délimiteurs doubles, elle est évaluée comme des valeurs délimitées séparées, une pour chaque paramètre passé au contexte courant
#	R	Nombre de paramètres du contexte courant
*	R	Tous les paramètres du contexte courant Si elle est utilisée entre des délimiteurs doubles, le résultat est un unique paramètre contenant tous les paramètres passés, séparés par des espaces
$	R	ID de processus bash actuel
-	R	Liste des flags d'option courants (de la commande), une lettre pour chacun, sans séparation
_	R	Nom de chemin complet du processus courant au cours de l'initialisation. En recherche de mail, contient le nom du fichier de mail courant Dans les autres cas, contient l'argument final de la commande précédente
0	R	Nom du processus ou script en cours
1 à 9	R	9 premiers paramètres du script (ou de la fonction) courant
BASH	R	Chemin complet du shell courant
BASH_VERSION	R	Numéro de version de votre bash
BASH_VERSINFO	R	Informations sur la version actuel de bash
DISPLAY	W	Nom et le numéro de la console de la machine sur laquelle les clients X affichent leurs interfaces
EUID	R	ID utilisateur effectif numérique du processus shell courant
HISTCMD	R	Index numérique de la commande actuelle dans l'historique des commandes

HISTSIZE	W	Taille maximale de l'historique des commandes
HOME	R	Chemin du répertoire de l'utilisateur
HOSTNAME	R	Nom de la machine
HOSTTYPE	R	Architecture de la machine
IFS	W	Valeur du séparateur de champs interne. Cette valeur est utilisée pour découper les commandes en leurs composants
LANG	W	Indique la localisation courante (ou préférée) aux programmes qui supportent l'internationalisation Linux. En tant que telle, elle est souvent utilisée comme variable d'environnement
LD_LIBRARY_PATH	W	Spécifie les emplacements additionnels (séparés par :) dans lesquels doivent être cherchées les librairies partagées pour les exécutables liés dynamiquement
LD_PRELOAD	W	Spécifie une liste de librairies spécifiques (séparées par des espaces) à charger dans les autres, y compris celles spécifiées par le programme lui-même. Pour des raisons de sécurité, cette spécification peut être incompatible avec setuid et setgid
LINENO	R	Lorsqu'elle est utilisée dans un shell ou une fonction, contient l'offset dans les lignes depuis le démarrage de ce shell ou cette fonction
MACHTYPE	R	Identificateur GNU du type de machine
MAIL	W	Vous informe de l'arrivée d'un nouveau mail dans une boîte de style UNIX mbox. Si vous voulez que bash vous informe automatiquement, spécifiez l'emplacement de cette boîte
MAILCHECK	W	Intervalle (en secondes) qui indique la fréquence de contrôle d'arrivée de nouveau mail dans la boîte spécifiée
OLDPWD	R	Nom du répertoire de travail précédent
OSTYPE	R	Nom du système d'exploitation courant
PATH	W	Liste de répertoires séparés par : dans laquelle on cherche les binaires lors de l'exécution de programmes. C'est une variable d'environnement

PPID	R	ID du père du processus en cours
PS1	W	Forme de générer l'invite de bash
PWD	R	Répertoire de travail courant
RANDOM	R	Valeur aléatoire différente. Dépendant de l'implantation
REPLY	R	Valeur des données lues sur l'entrée standard lorsqu'on y accède après la commande read, à moins qu'une variable différente ne soit spécifiée à read
SECONDS	R	Nombre de secondes écoulées depuis le démarrage du processus shell courant
SHELLOPTS	R	Liste des options du shell courant
UID	R	ID utilisateur numérique réel du propriétaire du processus shell courant

Annexe B

Appels système

`fork()` : Création de nouveaux processus

Prototype :
`<sys/types.h>`
`<unistd.h>`
`pid_t fork()`
`pid_t` est un type entier défini dans `<sys/types.h>`
Valeurs retournées :
`-1` en cas d'erreur. Nombre de processus usager ou système excédé. La variable error contient le numéro d'erreur `EAGAIN`.
`>0` identifiant le processus fils.
`0` retourné au processus fils.

`execv()` : Charger un programme dans l'espace adresse d'un processus

Prototype :
`<unistd.h>`
`int execv(const char *path, char *const argv[]);`
`path` spécifie le programme à charger.
`argv` spécifie les arguments à passer au programme chargé.
`argv[0]` contient le nom du programme.
`argv[i]` contient les arguments.
Valeurs retournées :
Si `execv` retourne alors l'appel a échoué.
Appel de `perror("execv n'a pas pu exécuter le programme")` pour afficher un message d'erreur.

`execvp()` : Charger un programme dans l'espace adresse d'un

processus

Prototype :
`<unistd.h>`
`int execvp(const char *file, char *const argv[]);`
`file` spécifie le programme à charger.
Répertoire de recherche du programme sauvegardé dans la variable `PATH`
`%PATH = /bin :/usr/bin :/usr/jmtorres %export PATH`
`argv[0]` contient le nom du programme.
`argv[i]` contient les arguments.
Valeurs retournées :
Si `execvp` retourne alors l'appel a échoué.
Appel de `perror("execvp n'a pas pu exécuter le programme")` pour afficher un message d'erreur.

`execve()` : Charger un programme dans l'espace adresse d'un processus

Prototype :
`<unistd.h>`
`int execve(const char *path, char *const argv[], char *const envp[]);`
`path` spécifie le programme à charger.
`argv[0]` contient le nom du programme.
`argv[i]` contient les arguments.
`envp` spécifie un environnement à substituer à l'environnement par défaut `USER`, `PATH`, `HOME`, `LOGNAME`, `SHELL`, `TERM`, etc.
Valeurs retournées :
Si `execve` retourne alors l'appel a échoué.
Appel de `perror("execve n'a pas pu exécuter le programme")` pour afficher un message d'erreur.

`execl()` : Charger un programme dans l'espace adresse d'un processus

Prototype :
`<unistd.h>`
`int execl(const char *path, char *const arg*0, ..., const char *argn, (char *)0);`
`path` spécifie le programme à charger.
`argi (i >=0)` spécifie une liste d'arguments à passer au programme à charger. Liste d'arguments terminée par `NULL`.
`arg0` contient le nom du programme, `arg1,...,argn` contient les arguments.

Valeurs retournées :
Si `execl` retourne alors l'appel a échoué.
Appel de `perror("execl n'a pas pu exécuter le programme")` pour afficher un message d'erreur.

`execlp()` : Charger un programme dans l'espace adresse d'un processus

Prototype :
```
<unistd.h>
int execlp(const char *file, char *const arg*0, ...,
const char *argn, (char *)0);
```
`file` spécifie le programme à charger.
`argi (i >=0)` spécifie une liste d'arguments à passer au programme à charger. Liste d'arguments terminée par `NULL`.
`arg0` contient le nom du programme, `arg1,...,argn` contient les arguments.
Valeurs retournées :
Si `execlp` retourne alors l'appel a échoué.
Appel de `perror("execlp n'a pas pu exécuter le programme")` pour afficher un message d'erreur.

`execle()` : Charger un programme dans l'espace adresse d'un processus

Prototype :
```
<unistd.h>
int execle(const char *path, char *const arg*0, ...,
const char *argn, (char *)0, char *const envp[]););
```
`path` spécifie le programme à charger.
`argi (i >=0)` spécifie une liste d'arguments à passer au programme à charger. Liste d'arguments terminée par `NULL`.
`arg0` contient le nom du programme, `arg1,...,argn` contient les arguments.
Valeurs retournées :
Si `execle` retourne alors l'appel a échoué.
Appel de `perror("execle n'a pas pu exécuter le programme")` pour afficher un message d'erreur.

`exit()` : Terminaison et nettoyage d'un processus

Prototype :
```
<stdlib.h>
int atexit(void (*func) (void));
```
`atexit` enregistre la foction pointée par `func`.

`func` ne doit pas avoir d'arguments.
Fonctions exécutées en ordre inversé d'enregistrement.
Valeurs retournées :
Indication du succès de l'enregistrement de la fonction.

`atexit()` : Définition par l'utilisateur d'opérations de terminaison

Prototype :
```
<stdlib.h>
int atexit(void (*func) (void));
```
`atexit` enregistre la fonction pointée par `func`.
`func` ne doit pas avoir d'arguments.
Fonctions exécutées en ordre inversé d'enregistrement.
Valeurs retournées :
Indication du succès de l'enregistrement de la fonction.

`wait()` : Synchronisation d'un processus parent avec ses processus fils.

Prototype :
```
<sys/types.h>
<sys/wait.h>
pid_t wait(int *status);
```
`status` contient le code de statut de terminaison du processus fils.
Valeurs retournées :
Numéro de processus (`pid`) du premier processus fils terminé.
-1 si le processus n'a aucun processus fils.
La variable error contient le code d'erreur `ECHILD`.

`waitpid()` : Synchronisation d'un processus parent avec un processus fils spécifique.

Prototype :
```
<sys/types.h>
<sys/wait.h>
pid_t waitpid(pid_t pid, int *status, int options);
```
`pid` spécifie le numéro du processus fils.
`status` contient le code de statut de terminaison du processus fils.
`options` est habituellement initialisé à `WNOHANG`.
Processus parent ne bloque pas durant l'exécution du processus fils, et `waitpid` retourne `0`.
Si `pid = -1` et `options = 0` alors `waitpid` a le même comportement que `wait`.
Valeurs retournées :

Numéro de processus fils ayant terminé.
-1 si le processus n'a aucun processus fils.
La variable error contient le code d'erreur `ECHILD`.

`getpid()` : Détermination du numéro d'un processus.

Prototype :
```
<sys/types.h>
<unistd.h>
pid_t getpid(void);
```
Valeurs retournées :
Numéro du processus appelant.

`getppid()` : Détermination du numéro du parent d'un processus.

Prototype :
```
<sys/types.h>
<unistd.h>
pid_t getppid(void);
```
Valeurs retournées :
Numéro du parent du processus appelant.

`pthread_create()` : Création d'un thread dans l'espace adresse d'un processus

Prototype :
```
<pthread.h>
int pthread_create(pthread_t *thread, const pthread_
attr_t *attr,
void *(*start_routine)(void*), void *arg);
```
`thread` pointe vers une adresse contenant le `ID` du nouveau `thread`.
`attr` spécifie les attributs du `thread`. Si `attr` est `NULL` alors les attributs par défaut sont utilisés.
Valeurs retournées :
`0` en cas de succès de création du `thread`.
Un code d'erreur en cas d'échec de création du `thread`. Nombre exédentaire de `threads`, attribut invalide, permissions insuffisantes, etc.

`pthread_exit()` : Terminaison d'un texttttthread.

Prototype :
```
<pthread.h>
void pthread_exit(void *value_ptr);
```
`value_ptr` est disponible à tout `thread` se synchronisant avec le

`thread` terminé.
Valeurs retournées :
Aucune.

`pthread_join()` : Synchronisation du `thread` appelant avec un autre `thread`.

Prototype :
```
<pthread.h>
int pthread_join(pthread_t thread, void **value_ptr);
```
`thread` est un pointeur vers l'`ID` du `thread` dont la terminaison est attendue.
`value_ptr` est passé à `pthread_exit()`
Valeurs retournées :
`0` en cas de succès de synchronisation avec un `thread`.
Un code d'erreur en cas d'échec de synchronisation avec un `thread`.
Pas de synchronisation possible avec le `thread` spécifié, ID inexistant du `thread` spécifié, etc.

`pthread_detach()` :
Indication au système que l'espace adresse du thread spécifié peut être déalloué à sa terminaison.

Prototype :
```
<pthread.h>
int pthread_detach(pthread_t thread);
```
`thread` est un pointeur vers l'`ID` du `thread`.
Valeurs retournées :
`0` en cas de succès.
Un code d'erreur en cas d'échec de synchronisation avec un `thread`.
Pas de synchronisation possible avec le `thread` spécifié, ID inexistant du `thread` spécifié, etc.

`pthread_attr_init()` : Initialisation des attributs par défaut d'un `thread`.

Prototype :
```
<pthread.h>
int pthread_attr_init(pthread_attr_t *attr);
```
`attr` est initialisé avec tous les attributs par défaut spécifiés pour un système donné.
Valeurs retournées :
`0` en cas de succès d'initialisation de l'objet `attr`.
Un code d'erreur en cas d'échec d'initialisationde l'objet `attr`.

Insuffisance de mémoire.
`pthread_attr_destroy()` `destroy()` : Effacement des attributs par défaut d'un `thread`.

Prototype :
`<pthread.h>`
`int pthread_attr_destroy(pthread_attr_t *attr);`
`attr` est effacé.
Valeurs retournées :
`0` en cas de succès d'initialisation de l'objet `attr`.
Un code d'erreur en cas d'échec d'initialisationde l'objet `attr`.

`sleep()` : Dormir pendant quelque temps. Pause de `NUMBER` secondes. `SUFFIX` peut-être en `s` secondes, `m` minutes, `h` heures ou `d` journées.
Prototype : `sleep [OPTION]...NUMBER[SUFFIX]`

Annexe C

Sites Internet

- Site du cours à l'École Polytechnique de Montréal
  - http ://www.cours.polymtl.ca/inf3600/
- Site Internet de GNU
  - http ://www.gnu.org
- Sites pour les développeurs Linux
  - http ://developer.kde.org
  - http ://www.linuxprogramming.com
  - http ://www.blackdown.org
  - http ://developer.gnome.org
  - http ://www.perl.com
  - http ://java.sun.com
- Site du serveur Web Apache pour Linux/Unix
  - http ://www.apache.org
- Sites des distributions et des noyaux Linux
  - http ://www.redhat.com
  - http ://www.calderasystems.com
  - http ://www.suse.com
  - http ://www.infomagic.com
  - http ://www.linuxppc.com
  - http ://www.slackware.com
  - http ://www.linux-mandrake.com
  - http ://www.java.sun.com
  - http ://www.kernel.org
  - http ://www.debian.org
  - http ://www.linux.org
- Sites des bureaux et des gestionnaires de fenêtres Linux

  - http ://www.gnome.org
  - http ://www.kde.org
  - http ://www.x11.org
  - http ://www.fvwm.org
  - http ://www.windowmaker.org
  - http ://www.enlightment.org
  - http ://www.xfree86.org
  - http ://www.themes.org
- Site sur les appels système Unix
  - http ://www.opennc.org/onlinepubs/007908799/
- Sites sur des systèmes d'exploitation divers
  - http ://www.apple.com/macosx
  - http ://www.microsoft.com
  - http ://www.qnx.org
  - http ://www.freebsd.org
  - http ://www.novell.com
  - http ://www.palmos.com
  - http ://www.sun.com
  - http ://www.samba.com
  - http ://www.beos.com
  - http ://www.ibm.com
  - http ://www.free-vms.org
- Sites sur le simulateur Nachos
  - http ://www.cs.duke.edu/ narten/110/nachos/main/main.html
  - http ://www.cs.washington.edu/homes/tom/nachos/

Bibliographie

[Tan01] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, New Jersey, 2001.