Présentation du Système Linux

Toumanari – le 16, 17 et 18 décembre 2010

Plan

- Architecture
- commandes de base
- Programmation shell
- IPC (Communication inter-processus)

Unix ou GNU/Linux ?

- 1979 : Première version d'Unix commercialisée
  - Unix Système V
  - Puis Unix BSD par l'université de Berkeley
- 1992 : Sortie de Solaris
  - Dérivée de Unix Système V
  - L'Unix de Sun
- Unix est un système payant et non libre
- L'alternative est le projet GNU/Linux
  - GNU pour (GNU is Not Unix)
  - Linux crée à l'origine par le finlandais Linus Torvald
  - C'est un système sous licence GPL (General Public Licence)
  - Il existe beaucoup de « distributions » Linux
    - Debian, Red Hat, Mandriva, …

Noyau et distribution

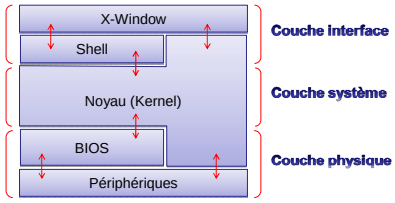
- Linux est architecturé autour d'un noyau
  - Ce noyau est appelé « Kernel »
  - Il contient toutes les fonctions de base d'un OS
    - Accès aux périphériques matériels standards
      - Disque dur, carte graphique, …
    - Accès aux périphériques spécifiques
      - A l'aide de pilotes
    - Gère les processus et la communication entre les processus
- Linux est un système multitâche préemptif
  - Le noyau gère l'exécution de chaque processus
    - Le processus peut être interrompu à tout moment
- Une distribution est un noyau auquel des logiciels ont été ajoutés
  - Possibilités de créer des distributions dédiées à un usage particulier

Linux sous licence GPL

- La licence GPL : General Public Licence
  - Concerne les modalités de distribution du noyau
  - Le code source est ouvert (Open Source)
  - Chacun peut le modifier et le revendre
    - Le code source modifié doit alors rester sous licence GPL
- Le noyau Linux est sous licence GPL
- Une distribution Linux est un ensemble noyau + logiciels sous licence GPL ou mixte
- Une distribution est constituée :
  - Du noyau Linux
  - De « packages » contenant des logiciels additionnels
- Le noyau reste entièrement sous GPL

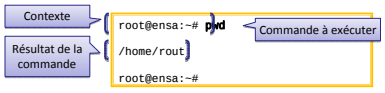
Architecture GNU/Linux

- Divisée en 3 couches distinctes
  - La couche physique : Périphériques et BIOS
  - La couche système : Gérée par le noyau
  - La couche interface : le Shell et/ou le système X-Window



Le shell ou « l'effrayante console »

- Le shell est un interpréteur de commandes
  - Permet à l'utilisateur d'interagir avec le système
  - Il lit et exécute les commandes de l'utilisateur
- C'est aussi un véritable langage de programmation
  - Il sera possible d'écrire des scripts exécutant des commandes répétitives
- Il en existe plusieurs
  - Le shell « bash » est le plus courant
  - Mais aussi les shells « csh », « ksh », « tcsh »



X-Window : Interface graphique

- C'est l'environnement graphique de tous les systèmes Unix et Linux
- Basée sur la relation client-serveur
  - Le serveur X peut accepter un client distant afin de déporter l'affichage sur une autre machine
- Pour Linux, le serveur X se nomme XFree86
- Le serveur X fonctionne avec un « gestionnaire de fenêtrage »
  - Il en existe plusieurs (Kde, Gnome, …)
  - Définit l'aspect du bureau, des fenêtres et des menus

La gestion des utilisateurs

- Contrairement à Microsoft Windows, la base de données des utilisateurs peut provenir de différents types de sources
- Grâce au PAM (Pluggable Authentication Module), les utilisateurs peuvent provenir de :
  - Un fichier (/etc/passwd et /etc/shadow)
  - Une base de données relationnelle
  - Un annuaire (type LDAP)
  - ...

Que faut-il pour réussir sous Linux ?

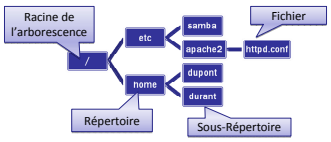
- Maîtriser le shell
  - Un serveur sous Linux n'a pas toujours d'environnement X-Window
  - Les commandes de base permettent beaucoup de choses
  - Il faut savoir utiliser le manuel des commandes (man)
- Bien connaître le système de fichiers
  - Sous Linux « tout est fichier » (même les périphériques et les processus)
  - Configurer un logiciel passe souvent par l'édition d'un simple fichier texte
- Savoir lire les fichiers de journalisation
  - Lorsqu'un logiciel ne fonctionne pas, il laisse des traces
  - Fichiers « log » stockés dans le « file system »
- LINUX est sensible à la casse (case sensitive)
  - Majuscules et minuscules sont interprétés différemment

Le système de fichier Linux

Toumanari – le 16, 17 et 18 décembre 2010

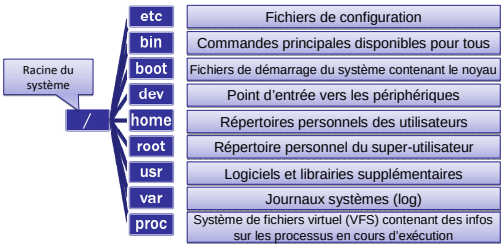
Quelques définitions

- Qu'est-ce qu'un système de fichier ?
  - Organisation physique des données sur un support
    - Sur un disque dur, une clé USB, un DVD, ...
- Qu'est-ce qu'une arborescence ?
  - Organisation logique des fichiers sur un ou plusieurs systèmes de fichiers
  - Il s'agit d'une structure de données hiérarchique de type arbre



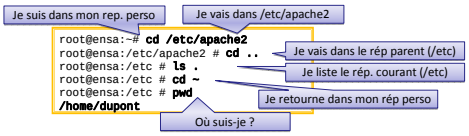
Arborescence Linux

- Voici l'arborescence typique d'un système Linux :



Les symboles associés à l'arborescence

- Différents symboles sont utilisés pour désigner des répertoires
  - Le « . » : Répertoire courant
  - Le « .. » : Répertoire parent
  - Le « ~ » : Répertoire personnel de l'utilisateur courant
- La commande « cd » permet de changer de répertoire
- La commande « ls » permet de lister un répertoire
- La commande « pwd » permet de connaître le rép. courant
- Exemples :



L'organisation du disque

- Organisation typique du poste de travail

- Le **M**aster **B**oot **R**ecord est situé dans les 1er secteurs du disque
- Il est constitué de 2 parties :
  - La table des partitions
  - Le programme d'amorçage qui charge le noyau du système
- Plusieurs types de partitions
  - Principale
  - Etendue
  - Logique

Les partitions

- Les partitions principales
  - Au maximum de 4
  - Accepte tout type de système de fichiers
- Les partitions étendues
  - Destinées à contenir des partitions logiques et non un système de fichiers
  - Nécessitent au moins une partition principale
- Les partitions logiques
  - Contenues dans une partition étendue
  - Accepte tout type de système de fichiers
- Exemple permettant d'installer plusieurs systèmes d'exploitation

Prise en charge des disques sous Linux

- Le pointeur spécial /dev permet l'accès aux disques
  - Format des pointeurs sur disque :

- Types de bus
  - hd : Périphériques IDE
  - sc : Périphériques SCSI
  - sd : Périphériques SATA
- Exemples
  - /dev/hda1 :
    - Partition 1 sur le 1er disque IDE
  - /dev/sdb2 :
    - Partition 2 sur le 2ème disque Sata

Les formats des systèmes de fichiers (1)

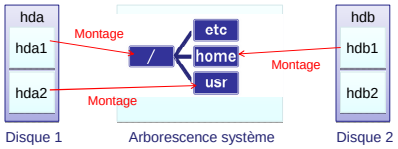
- À chaque système est associé un format
  - Définit la structure des données sur le support
- Sous Linux
  - ext2, ext3, jfs, xfs
  - ext3 est la plus courante pour Linux
- Sous Windows
  - fat, fat32, ntfs
  - Nfts est utilisé sous windows XP et Vista
- Toujours préférer un système de fichier « journalisé »
  - Chaque séquence de lecture/écriture est d'abord inscrite dans un journal avant d'être effectuée
    - Si le système se bloque pendant la séquence, elle sera achevée après le redémarrage
  - On évite les erreurs dans le système de fichiers

Les formats des systèmes de fichiers (2)

- Le format « swap » est utilisé comme « mémoire virtuelle »
  - Dans le cas où la mémoire vive est saturée
  - Par le système pour améliorer les performances
  - La taille du « swap » est fixée au double de la mémoire vive
    - Si 512Mo de mémoire vive -> 1024Mo de swap
- Linux peut lire la plupart des formats
  - Notamment Ceux de Windows : NTFS, FAT, FAT32
- Avant d'être utilisé, un disque doit être partitionné
  - A l'aide de la commande « fdisk » si Linux est déjà installé
  - Par le programme d'installation sinon (dépend de la distribution)
- Il faut ensuite créer un système de fichier
  - Avec l'utilitaire générique « mkfs »
    - mkfs.ext2, mkfs.ext3, mkfs.xfs, ...

Points de montage (1)

- Sous Linux, « Tout est fichier »
  - L'arborescence est construite à partir de « points de montage »
- Un point de montage est une association entre une partition physique et l'arborescence du système



- Avantages
  - Mettre à l'abri certaines données stratégiques comme /home
  - La défaillance du disque hdb n'entraîne pas une réinstallation totale

Points de montage (2)

- Tant qu'ils ne sont pas effectués, le système de fichiers est inaccessible
- Ils sont réalisés automatiquement au démarrage du système
  - Seulement ceux qui figurent dans le fichier « /etc/fstab »
- Il est possible de créer un point de montage manuellement
  - Pour les clés USB ou le CDROM par exemple
  - En utilisant la commande « mount »
  - Et « umount » pour supprimer le point de montage

```
root@ensa:~# mount /dev/hdd /mnt/cdrom
root@ensa:~# umount /dev/hdd
```

- Une partition est associée à un système de fichiers
  - Il faut parfois préciser le type de ce système
    - ext2, ext3, xfs, swap, jfs, iso9660, vfat, …

Le répertoire spécial « /proc » (1)

- Répertoire spécial n'existant pas physiquement sur le disque
- « /proc » est un pseudo-système de fichiers mis à jour en temps réel par le noyau
  - Chaque processus en cours d'exécution y dispose d'un sous répertoire
    - Le nom de ce sous-répertoire correspond au PID du processus
    - Des informations importantes sur le processus y sont stockées
      - Fichiers et mémoires utilisées par le processus
- Les fichiers de /proc sont en :
  - Lecture seule : Permet d'obtenir des infos sur les processus
    - Exemple : **/proc/cpuinfo** (Infos sur le processeur)
  - Ecriture : Permet de modifier des paramètres du noyau
    - Exemple : **/proc/sys/net/ipv4/ip_forward** (Activation du routage)

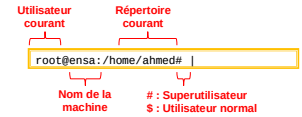
Le répertoire spécial « /proc » (2)

- Quelques fichiers intéressants à consulter
  - **cpuinfo** : informations sur le(s) processeur(s)
  - **meminfo** : utilisation de la mémoire
  - **ioports** : Adresses physiques des différents périphériques matériels
- Visualiser le fichier « /proc/cpuinfo » avec la commande « cat »

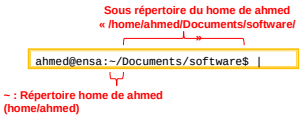
```
root@ensa:~# cat /proc/cpuinfo
processor : 0
vendor_id : GenuineIntel
cpu family          : 6
model               : 23
model name          : Intel(R) Core(TM)2 Duo CPU     T9300  @ 2.50GHz
stepping  : 8
cpu MHz             : 2500.585
cache size          : 6144 KB
fdiv_bug  : no
hlt_bug             : no
f00f_bug  : no
coma_bug  : no
…
```


Comment se repérer dans le système de fichiers ?

- La ligne de commande donne des informations :



- Autre exemple



Où suis-je, où vais-je ?

- La commande « **pwd** » permet de savoir quel est le répertoire courant
- La commande « **ls** » permet de lister les fichiers contenus dans un répertoire
- La commande « **cd** » permet de changer de répertoire
- Les symboles suivants ont une signification particulière :
  - « . » : Le point désigne le répertoire courant
    - Exemple : Exécuter un script depuis le répertoire courant

```
root@ensa:/home/ahmed# ./script.sh
```

  - « .. » : Les 2 points désignent le répertoire parent
    - Exemple : Se déplacer dans le répertoire parent

```
root@ensa:/home/ahmed/Docs# cd ..
```

  - « ~ » : Désigne le répertoire home de l'utilisateur courant

```
root@ensa:/home/ahmed/Docs# cd ~
```


Chemin relatif et absolu

- Il existe 2 méthodes pour spécifier un chemin dans le système de fichiers
  - Chemin relatif : Dépend du répertoire courant
  - Chemin absolu : Débute à la racine du système (« / »)
- Exemples

relatif		absolu
`root@ensa:/home/ahmed1# cd Documents`	=	`root@ensa:/home/ahmed# cd /home/ahmed/Documents`
`root@ensa:/home/ali# cd Documents` `root@ensa:/home/ali/Documents/# |`	≠	`root@ensa:/home/ahmed# cd /home/ahmed/Documents` `root@ensa:/home/ahmed1/Documents/# |`
`root@ensa:/etc/apache# cd ../` `root@ensa:/etc/# |`	=	`root@ensa:/etc/apache# cd /etc` `root@ensa:/etc/# |`

- Attention aux chemins relatifs à l'intérieur d'un script
  - Le script peut-être exécuté depuis n'importe où
  - Le répertoire courant est donc différent à chaque fois

Les droits des fichiers et répertoires

- Linux est un système multi-utilisateurs
  - Plusieurs utilisateurs se partagent l'espace disque
  - Les fichiers et répertoires d'un utilisateur ne doivent pas être accessibles par les autres
  - Les fichiers de configuration du système doivent être protégés
- Nécessité de spécifier des droits pour chaque fichier/répertoire
  - Plusieurs types de droits : Lecture (R), écriture (W), exécution (X)
  - Ces droits s'appliquent pour 3 groupes d'utilisateurs :
    - Le propriétaire (user) du fichier
    - Le groupe (group) propriétaire (Tous les utilisateurs membre du groupe)
    - Les autres (others). Désigne tous les utilisateurs non membres des 2 précédents
- Les droits sont responsables d'un grand nombre d'erreurs de configuration

Droits : Différence entre fichiers et répertoires

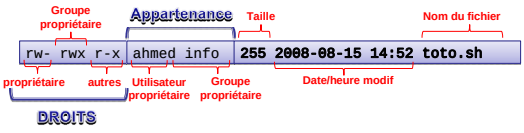
- Nous avons vu qu'il existe 3 types de droits : r, w et x
- Ces droits n'ont pas la même signification pour un fichier que pour un répertoire
- Pour un fichier :
  - **r** : Lecture (afficher)
  - **w** : Ecriture (modification)
  - **x** : Exécution (exécution d'un script)
- Pour un répertoire
  - **r** : Lire le contenu, lister les fichiers (avec ls par exemple)
  - **w** : Modifier le contenu, créer et supprimer des fichiers (avec les commandes « cp », « mv », « rm »)
  - **x** : Permet d'accéder aux fichiers du répertoire. Mais aussi de naviguer dans les sous-répertoires (avec « cd »)
  - En général, le droit w est souvent associé au droit x

Les droits sur les fichiers et répertoires

- La commande « ls -l» permet d'afficher les droits qui s'appliquent

```
root@ensa:/home/ahmed/Documents# ls -l
total 20
-rw-r--r-- 1 ahmed ahmed      0 2008-08-15 14:42 projet.txt
-rw-rw-r-x 1 ahmed compta 7406 2008-08-15 14:44 rapport2006.ods
-rw-rw-r-- 1 ahmed ahmed   7363 2008-08-15 14:44 rapport-activite.odt
-rw-rwxr-x 1 ahmed compta  255 2008-08-15 14:52 script.sh
```

- Signification des différents champs



Propriétaire : Lecture, écriture
Groupe : Lecture, écriture et exécution
Autres : Lecture et exécution

Remarques sur les droits

- Le droit « w » accordé à un répertoire permet :
  - D'y effacer des fichiers quels que soient le propriétaire et les droits qui s'appliquent à ces fichiers
  - Quand il est donné à un groupe, n'importe quel utilisateur de ce groupe peut supprimer des fichiers (dangereux)
- Les droits ne s'appliquent pas au « super-utilisateur »
  - Il a tous les droits sur tout le système de fichiers
  - C'est une très grande responsabilité puisque sous Linux tout repose sur les fichiers
    - La tendance évolue vers une utilisation très modérée voire interdite du compte « root »
- Le droit « x » accordé à un répertoire est un préalable indispensable pour exercer des droits sur les fichiers contenus
- L'utilisateur qui crée un fichier en devient le propriétaire
  - Ce fichier aura comme groupe propriétaire, le groupe primaire du propriétaire (Groupe principal auquel appartient le propriétaire)

Le super-utilisateur

- Le compte « root » possède tous les droits
  - Celui qui possède le mot de passe root peut tout faire tout
- La plupart des distributions récentes désactivent le compte « root »
  - Certains utilisateurs peuvent endosser temporairement le rôle du super-utilisateur
    - Avec la commande « sudo » (Super User Do)
      - La commande qui suit le « sudo » sera exécutée en tant que « root »

```
rout@ensa:~/Documents$ sudo commande
```

- En règle générale, on utilise « sudo » que temporairement
  - Pour des tâches administratives
    - Accéder à l'ensemble du système de fichiers
    - Gérer les utilisateurs
    - …

Usurper l'identité d'un utilisateur ?

- Parfois, le super-utilisateur doit endosser un autre rôle afin d'effectuer une tâche
- Que se passe t-il si le super-utilisateur crée un fichier ?
  - Ce fichier lui appartient
  - Selon les droits qui s'appliquent, les autres utilisateur ne peuvent pas y accéder
- Que faire alors ?
  - Se faire momentanément passer pour un autre utilisateur
  - Avec la commande « su »

```
root@ensa:~# su ali
gen@ensa:~$ touch toto.txt
gen@ensa:~$ ls -l
total 32
-rw-r--r-- 1 ali ali    0 2009-08-31 05:47 toto.txt
gen@ensa:~$ exit
exit
root@ensa:~#
```

Je me fais passer pour « ali »

Le fichier créé appartient à « ali »

Je reprends mon rôle initial

Le masque de protection

- Utilisé pour définir les droits par défaut
  - Droits appliqués pour un nouveau fichier lors de sa création
- Les bits du masque à 1 empêchent le fichier d'obtenir le droit correspondant
- Exemple avec un masque de protection de 027

```
        0 2 7
       ↙ ↓ ↘
000 010 111
rwx rwx rwx   Permissions maximum
rwx r-x ---   Permissions effectives après application du masque
```

- La commande « umask » permet de modifier le masque
  - Les fichiers et répertoires nouvellement crées seront alors protégés
  - La valeur par défaut du masque est 022
    - Est-ce suffisant ?

Les droits étendus : le SUID

- Permet de bénéficier de droits supplémentaires lors de l'exécution d'une commande
  - Un utilisateur quelconque peut alors avoir des droits supplémentaires seulement s'il exécute la commande ayant le SUID
- Exemple de la commande « passwd »
  - Elle permet de modifier son mot de passe
  - « passwd » doit écrire dans le fichier « /etc/shadow » et pourtant :

```
linux:~# ls -l /etc/shadow
-rw-r----- 1 root shadow 700 2007-12-04 18:39 /etc/shadow
```

Aucune permission d'écriture sur ce fichier

```
linux:~# ls -l /usr/bin/passwd
-rwsr-xr-x 1 root root 28480 2007-02-27 08:53 /usr/bin/passwd
```

La commande aura les droits du super-utilisateur même si n'importe quel autre utilisateur lance son exécution

Les droits étendus : le SGID

- Identique au SUID mais appliqué au groupe propriétaire
  - La commande obtiendra les droits du groupe propriétaire s'il elle est exécutée par un autre utilisateur
- Attention, appliquée à un répertoire, le SGID :
  - Modifie le groupe propriétaire d'un fichier créé dans le répertoire
  - Ce ne sera plus le groupe primaire du propriétaire
    - Mais plutôt le groupe propriétaire du répertoire
  - Il y a donc un mécanisme d'héritage entre le répertoire et les fichiers nouvellement créés qu'il contient
- Exemple :

```
drwxrws---  2 root info  4096 2008-08-24 13:05 docs-compta
```

SGID positionné sur « docs-compta »

```
-rw-r--r-- 1 ahmed info 0 2008-08-24 13:09 nouveau.txt
```

Le fichier nouvellement crée par ahmed appartient au groupe « compta »

Modifier les droits avec « chmod »

- La commande « chmod » permet de modifier les droits :
  - 2 syntaxes différentes

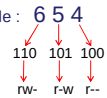
- Mode symbolique :
  - Basé sur des symboles (ugoa) et des opérateurs (+,-,=)
  - **u** (user), **g** (group), **o** (others), **a** (all users)
  - **+** (Ajouter le droit), **-** (Retirer le droit), **=** (Ajouter le droit et retirer tous les autres)
  - Exemple (Ajoute le droit d'exécution au propriétaire) :

```
chmod u+x rapport.txt
```

- Mode octal :
  - Basé sur des nombres de 0 à 7
  - A chaque bit de la traduction binaire correspond un droit
  - Exemple (rw- rw- r--) : `chmod 664 rapport.txt`

Mode octal de « chmod »

- Les droits sont représentés par un nombre octal (Base 8)
  - De 1 à 7
- La représentation binaire (base 2) donne le détail des droits
  - Exemple : 6 5 4



Propriétaire : Lecture, écriture
Groupe : Lecture et exécution
Autres : Lecture seulement

- Ce mode permet de modifier tous les droits en même temps
  - A utiliser avec précaution
  - Très efficace pour s'assurer que tous les fichiers ont les mêmes droits
  - Utilisé pour sécuriser les accès des utilisateurs aux fichiers

Modifier l'appartenance avec « chown »

- La commande « chown » (Change owner) permet de changer l'appartenance
  - Pour le propriétaire ou le groupe propriétaire
- Syntaxe :

```
chown [OPTION]... [OWNER][:[GROUP]] FILE...
```

- Exemples :
  - Modification du propriétaire (ahmed)

```
chown ahmed /usr/docs/toto.txt
```

  - Modification du groupe propriétaire (info)

```
chown :info /usr/docs/toto.txt
```

  - Modification du propriétaire (ali) et du groupe (direction)
    - Pour tout le contenu du répertoire (Option R - récursif)

```
chown -R ali:direction /usr/docs/rapports/
```


Copier des fichiers

- La commande « cp » copie des fichiers source vers une ou plusieurs destinations
- Syntaxe :

```
cp [OPTION]... SOURCE... DIRECTORY
```

- Exemples
  - Copie le fichier « toto.txt » vers « /home/ahmed »

```
cp toto.txt /home/ahmed
```

  - Copie tous les fichiers du répertoire « /home/ali » vers « /home/ahmed »

```
cp  /home/ali/* /home/ahmed
```

  - Copie le rep « /home/ali » tout entier vers « /home/ahmed »

```
cp  -r /home/ali/ /home/ahmed
```

  - Copie en conservant les droits et l'appartenance (-a)

```
cp  -a /home/ali/rapport.odt /home/ahmed
```


Déplacer/Renommer des fichiers avec « mv »

- La commande « mv » déplace ou renomme une source vers une destination.
- Syntaxe :

```
mv [OPTION]... SOURCE... DIRECTORY
```

- Exemples
  - Déplace le fichier « toto.txt » vers « /home/ahmed » sans le renommer

```
mv toto.txt /home/ahmed
```

  - Renomme le fichier « toto.txt » en « tata.txt »

```
mv toto.txt tata.txt
```

  - Déplace tous les fichiers de « rep1 » vers « rep2 »

```
mv rep1/* rep2/
```

Effacer fichiers et répertoires avec « rm »

- La commande « rm » permet d'effacer des fichiers et des répertoires
  - A utiliser avec précaution (Surtout avec l'option –r)
- Exemples :
  - Effacer le fichier « rapport.txt »

```
rout@ensa:~$ rm /home/ahmed/rapport.txt
```

  - Effacer le répertoire « /home/ahmed »

```
rout@ensa:~$ rm -r /home/ahmed/
```

  - Effacer tous les fichiers du rép. courant commençant par « rapport »

```
rout@ensa:~$ rm ./rapport*
```

  - Effacer les fichiers du rép. courant se terminant par .txt

```
rout@ensa:~$ rm ./*.txt
```

- Avant d'utiliser « rm », savoir avant quelle sera la portée
  - « rm -r /* » peut générer une catastrophe

Les inodes

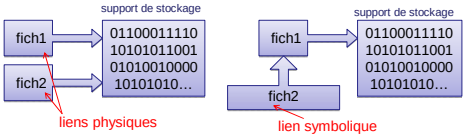
- Un « inode » est une structure de données concernant un fichier
  - Contient des informations sur :
    - Les droits, le propriétaire et le groupe
    - Le périphérique qui le contient
    - Des données relatives au système de fichiers et à l'emplacement du fichier sur le support de stockage
- A chaque fichier, correspond un « inode »
  - Il est unique pour le périphérique de stockage qui contient le fichier
- Pour connaître l'inode d'un fichier, la commande « ls » avec l'option « -i ».

```
rout@ensa:~/Documents/essais$ ls -il /home/rout/Documents
total 12
   8246 drwxr-xr-x 2 rout rout 4096 2008-08-25 15:08 essais
407185 drwxr-xr-x 3 rout rout 4096 2008-06-02 14:20 software
475069 drwxr-xr-x 3 rout rout 4096 2008-07-29 15:54 vmware-tools
```

inode

Les liens

- Un lien est un type spécial de fichier qui fait référence à un autre fichier
- Axe central du fonctionnement de Linux, le lien permet :
  - De créer des raccourcis vers des fichiers existants
    - La compatibilité des logiciels entre les distributions Linux est assurée par les liens
  - D'éviter de stocker plusieurs fois le même fichier dans des répertoires différents
- Un petit dessin :



Les liens symboliques

- Le lien symbolique est une référence vers un fichier cible
  - Lorsque le fichier cible est effacé, le lien est rompu
  - Lorsque le lien est effacé, le fichier cible n'est pas effacé
- Exemple :

```
rout@ensa:~/Documents$ ls -l
total 8
lrwxrwxrwx 1 rout rout   29 2008-08-25 14:23 ip -> /proc/sys/net/ipv4/ip_forward
dr-xr-xr-x 3 rout rout 4096 2008-06-02 14:20 software
dr-xr-xr-x 3 rout rout 4096 2008-07-29 15:54 vmware-tools
```

Nom du lien

Indique que c'est un lien

Emplacement du vrai fichier

- La commande « ln » avec l'option « -s » est utilisée pour créer un lien symbolique

```
rout@ensa:~/Documents$ ln -s /proc/sys/net/ipv4/ip_forward ip
```

Cible (Target)

Nom du lien (link name)

Les liens physiques

- Un lien physique est associé à un emplacement sur le support de stockage
  - 2 liens peuvent être associés au même « inode »
  - Similaire à la notion de « pointeurs » du langage C
  - Deux liens physiques sont considérés comme 2 fichiers indépendants
    - Même si leur contenu est au même emplacement sur le support
  - Le lien physique est vu comme un fichier régulier
- Créer un lien physique avec la commande « ln » :

```
rout@ensa:~/Documents$ ln /home/ahmed/Documents/rapport2007-2008.doc rap0708
rout@ensa:~/Documents/essais$ ls -il
total 176
470930 -rw-r--r-- 2 rout rout 84091 2008-08-25 14:48 rap0708
470930 -rw-r--r-- 2 rout rout 84091 2008-08-25 14:48 rapport-annee2007_2008.doc
```

nom fichier

cible

L' « inode » est identique. Il s'agit bien de liens physiques

Nombre de liens vers cet inode. C'est un indice permettant de supposer qu'il s'agit d'un lien

Le Shell

Toumanari – le 16, 17 et 18 décembre 2010

Shell, c'est quoi exactement ?

- Il s'agit d'une interface texte entre l'utilisateur et le système
  - Tout se fait au clavier
  - Pas de clic de souris
- L'utilisateur tape des commandes qui sont exécutées par le système
  - Le shell est donc un « interpréteur de commandes »
  - Chaque commande a une syntaxe particulière
  - Il existe des milliers de commandes différentes
- Les commandes peuvent aussi provenir d'un fichier
  - Le fichier contient les commandes à exécuter
  - L'utilisateur appel le fichier plutôt que de taper toutes les commandes
    - Utile pour les tâches répétitives
- Le shell reste le moyen le plus efficace pour contrôler le système
  - C'est aussi le plus utilisé sous Linux/Unix

Shell ou langage de programmation ?

- Le shell est un véritable environnement de programmation
  - Variables, boucles, structures de contrôle ….
- Les programmes écrits pour le shell sont interprétés au moment de l'exécution
  - Aucune compilation préalable n'est utile
  - Les performances n'égalent pas un programme en C
- Les programmes écrits pour le shell sont des « scripts » :

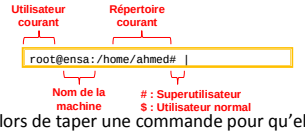
```
# Test de l'existence du fichier
if [ -f $logfile ]
then
  rm $logfile
  echo effacement de ipscan.log effectue
fi

echo scan des ip...
```

Les différents types de shell

- Il existe plusieurs types de shell
  - Bourne shell
  - Korn Shell
  - Bash (Bourne again shell)
  - Tcsh (Terminal C shell)
  - L'interpréteur de commande MS-DOS (Sous Windows)
  - PowerShell (Windows 2008 server)
- Sous Linux, on peut choisir son shell
  - Le shell bash domine le marché actuellement

Présentation de la ligne de commande (CLI)

- La ligne de commande se présente sous forme de texte ayant la signification suivante :

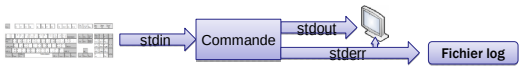


- Il suffit alors de taper une commande pour qu'elle soit exécutée
- Une commande peut être appelée :
  - En tapant son nom puis des arguments ou paramètres
  - Exemple permettant de rechercher dans le répertoire courant les fichiers dont la taille est supérieure à 2Mo

```
root@ensa:/home/ahmed# find ./ -size +2M -print
```


Les entrées/sorties

- Chaque commande dispose d'entrées/sorties :
  - Comme l'écran (sortie) ou le clavier (entrée)
- 3 types différents:
  - L'entrée standard définie par le symbole « **stdin** » et le descripteur **0**
    - Provenant du clavier par défaut
  - La sortie standard « **stdout** » et le descripteur **1**
    - Dirigée vers l'écran par défaut
  - La sortie d'erreurs « **stderr** » et le descripteur **2**
    - Egalement redirigée vers l'écran par défaut
- Possibilité de modifier le comportement par défaut
  - En utilisant les redirections d'entrées/sorties

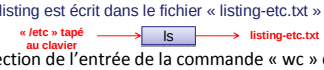


Redirections des entrées/sorties

- Capacité de rediriger les entrées/sorties d'une commande
  - « stdout » ou « stderr » vers un fichier plutôt qu'à l'écran
  - « stdin » depuis un fichier plutôt que le clavier
- Utilisation des opérateurs suivants :
  - > : Redirection de la sortie vers un fichier
  - >> : Redirection de la sortie à la fin du fichier (concaténation)
  - < : Redirection de l'entrée depuis un fichier
- Exemple de redirection de la sortie vers un fichier :

```
ls -l /etc  > listing-etc.txt
```

  - Le listing est écrit dans le fichier « listing-etc.txt »



- Redirection de l'entrée de la commande « wc » depuis un fichier
  - Compte le nb de lignes du fichier « listing-etc.txt »

```
wc -l  < listing-etc.txt
```

Redirections (2)

- 2 syntaxes supplémentaires possibles pour les redirections

```
n>&m ou
n>fichier
```

  - n : Numéro du descripteur à rediriger
  - m : Numéro du descripteur vers lequel on va rediriger
  - fichier : Fichier vers lequel s'effectuera la redirection
- Rappel : 0 (entrée standard), 1 (sortie standard), 2 (sortie erreur)
- Exemple : Rediriger la sortie d'erreur vers un fichier
  - Les messages d'erreurs seront écrits dans « erreurs.log »

```
cp /home/ahmed /home/ali 2>erreurs.log
```

- Exemple : Rediriger la sortie standard vers la sortie d'erreur

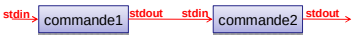
```
cp /home/ahmed /home/ali 1>&2
```

- Exemple : Rediriger stdout ET stderr vers un fichier

```
commande > fichier.txt 2>&1
```


Les tubes (pipes)

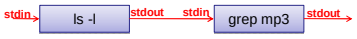
- Il s'agit de rediriger la sortie d'une commande vers l'entrée d'une autre avec l'opérateur « | »



- Élaborer des commandes complexes en une seule ligne
  - Exemple : Filtrer le résultat de la commande « ls » avec « grep »

```
ls -l /etc  | grep 'mp3'
```

    - On obtient la liste des fichiers contenant « mp3 »




```
rout@ensa:~$ ls -l /etc  | grep 'mp3'
-rw-r--r-- 1 rout rout    0 2008-08-27 15:16 morceau1.mp3
-rw-r--r-- 1 rout rout    0 2008-08-27 15:16 morceau2.mp3
-rw-r--r-- 1 rout rout    0 2008-08-27 15:16 morceau3.mp3
-rw-r--r-- 1 rout rout    0 2008-08-27 15:16 morceau4.mp3
```


Exécution conditionnelle de commandes

- Pour exécuter une seule commande, rien de plus simple
  - Taper son nom au clavier
- Pour exécuter plusieurs commandes à la suite
  - L'exemple suivant crée un répertoire, s'y déplace et crée un fichier

```
mkdir toto ; cd toto ; touch alire.txt
```

- L'exécution conditionnelle de commandes est possible
  - Les commandes s'exécutent les unes après les autres sous condition
  - Utilisation des opérateurs && et ||
    - L'exemple suivant exécute commande1 et commande2 seulement si le résultat renvoyé par commande1 est égal à 0

```
commande1 && commande2
```

    - Même chose mais si le résultat de commande1 est différent de 0

```
commande1 || commande2
```


Ca marche ou ca marche pas ?

- Une commande renvoie toujours une valeur
  - 0 si la commande s'est exécutée correctement
  - 1 ou différent de zéro dans le cas contraire
- Exemple
  - La variable « $? » correspond à la valeur renvoyée par la dernière commande exécutée (Donc la commande « cd »)

```
# cd toto
Bash : cd toto : Le répertoire n'existe pas
# echo $?
1
```

- Cette valeur peut-être exploitée dans un script pour connaître le résultat d'une commande avant d'exécuter la suite

Caractères spéciaux

- Certains caractères ont une signification particulière
  - Interprétés par le shell
- Astérisque ou étoile : *
  - Interprété comme toute suite de caractères alphanumérique
  - Exemple : Effacer tous les fichiers commençant par « rapport »

```
rm rapport*
```

- Point d'interrogation : ?
  - Interprété comme un seul caractère alphanumérique
  - Exemple : Effacer certains fichiers commencant par « rapport?.doc »

```
rm rapport?.doc
```

    - « rapport1.doc » sera effacé mais pas « rapport12.doc »
- Point virgule : ;
  - Séparateur de commandes

```
cp bilan.txt bilan2007.txt ; rm bilan.txt
```


Caractères spéciaux (2)

- Les crochets : []
  - Remplace un caractère choisi parmi ceux énumérés entre les crochets
  - Exemple : Effacer les fichiers dont la 1ère lettre est « a » ou « b » et se terminant par « .txt »

```
rm [ab]*.txt
```

```
rm rapport[12][0-9].txt
```

- L'espace
  - Utilisé comme séparateur de paramètres pour une commande
  - Exemple : Effacement de 2 fichiers passés en paramètres

```
rm rapport.doc rapport2008.txt
```


Caractères spéciaux (3)

- L'antislash : \
  - Inhibe le caractère spécial suivant
  - Exemple : Effacer un fichier contenant le caractère spécial espace

```
rm rapport\ .txt
```

- Autres caractères spéciaux : **! , ^, $, <, >, |**
  - **^** : Exprime la négation

```
rm [^r]*.txt
```

  - **$** : Utilisé pour les variables dans les scripts
  - **!** : Utilisé pour accéder à l'historique des commandes (! suivi du numéro de la commande dans l'historique. Voir la commande « history »)
  - **<,>** : Redirections
  - **|** : pipe (tube)

Chaînes de caractères

- Il existe plusieurs délimiteurs de chaînes de caractères
  - Apostrophe (simple quote) : *'texte'*
    - Le texte n'est pas du tout interprété
  - Guillemets (double quotes) : *"texte"*
    - Seuls les caractères \ , $ et ` sont interprétés
    - Utilisé pour du texte qui contient des variables ou des caractères spéciaux
  - Anti-quotes : *`texte`*
    - Le texte est interprété comme une commande à exécuter. Le résultat de cette commande sera substitué
    - Utilisé dans le but d'exploiter le résultat d'une commande
- Exemples

Chaînes de caractères : Exemples

- Exemples
  - Rechercher la chaîne « toto » dans tous les fichiers du répertoire « /home/ahmed »

```
grep 'toto' /home/ahmed/*
```

  - Rechercher les fichiers contenant la date d'aujourd'hui
    - 1) Obtenir la date d'aujourd'hui au format JJMMAA

```
root@ensa:~# date +%d%m%Y
28082008
```

    - 2) Exploiter ce résultat pour rechercher cette date dans les fichiers

```
root@ensa:~# grep `date +%d%m%Y` /home/ahmed/*
```

  - Créer un fichier « alire.txt » dans le répertoire home de l'utilisateur
    - La variable $HOME sera remplacée par sa valeur

```
root@ensa:~# touch "$HOME"/alire.txt
```


Rechercher des fichiers avec « find »

- Recherche multicritères
  - Par la date, la taille, le nom, …
- Syntaxe :

```
find [-H] [-L] [-P] [path...] [expression]  opion
```

  - « path » : Chemin où chercher
  - « expression » : Expression permettant de définir des critères de recherche
- L'expression est construite autour d'options et de tests
  - Exemple d'option : « maxdepth » limite la profondeur de recherche
  - Exemple de test : « name » recherche par le nom du fichier
- Rechercher les vidéos mpeg dont la taille est supérieure à 10Mo

Test sur la taille

```
find /home/ahmed/Documents/  -size +10M  -name "*.mpeg"   -print
```

+ grand que

Test sur le nom

Filtrer avec « grep » ou « egrep »

- Commande puissante permettant de filtrer les lignes d'une sortie de texte
  - Affiche uniquement les lignes correspondant aux critères de filtrage
  - Très utilisée pour rechercher à l'intérieur des fichiers
- Syntaxes :

```
grep [OPTIONS] PATTERN [FILE...]
grep [OPTIONS] [-e PATTERN | -f FILE] [FILE...]
```

  - « Pattern » : Expression régulière agissant comme filtre
  - « File » : Fichier ou répertoire où est débutée la recherche
- Options intéressantes
  - -r : Permet de recherche dans les sous-répertoires (Peut-être long)
  - -n : Connaître le n° de la ligne et donc la position de l'occurrence trouvée dans le fichier
  - -c compte le nombre d'occurrence
  - -v inverse la selection

Les scripts

- Un script est un fichier contenant un ensemble de commandes exécutées séquentiellement
  - Sous forme de fichier texte contenant les commandes
- Le script ne peut être exécuté que par un interpréteur
  - « /bin/sh » pour le shell bash
- Le langage de script shell est un langage évolué offrant de nombreuses possibilités
  - Boucles, variables, tests avec if, création de fonctions, …
- Dans quels cas utilise-t-on les scripts ?
  - Pour effectuer un travail répétitif
  - Pour des tâches d'administration système
  - Pour installer des programmes
  - Au démarrage du système pour démarrer les services et applis
  - Bref : Tout le temps !!!

Exécuter un script

- Pour exécuter le script, il faut appeler l'interpréteur

```
root@ensa:~# sh monscript
```

- Possibilité de simplifier l'appel en ajoutant la ligne suivante en tête du script

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

  - L'appel est alors plus simple

```
root@ensa:~# ./monscript
```

- L'utilisateur courant doit posséder le droit « x » pour le fichier
  - Exemple : Seul, l'utilisateur « ahmed » pourra exécuter le script

```
-rwxr--r-- 1 ahmed compta 7406 2008-08-15 14:44 script.sh
```

  - Pour autoriser les membres du groupe « info »

```
chmod g+x script.sh
```


Format de script

- Il s'agit d'un fichier texte au format suivant :

```
#! /bin/sh

# Commentaires : Fonction du script
# Auteur : toto
# Version : 1.0 - Oct 2008

# Début du script
…
```

- Un script peut accepter des arguments
  - Il faut donc vérifier ces arguments avant de commencer le traitement
  - Rappeler le fonctionnement du script par un message d'erreur

```
Usage de la commande : script.sh arg1 arg2 arg3

arg1 : 1er argument , arg2 : 2ème  argument, arg3 : 3ème  argument
```


Variables de substitution

- Elles sont définies implicitement et peuvent être utilisées à tout moment dans le script
- Quelques variables utiles
  - $0 : Nom du script (Utile lorsqu'on renomme le script)
  - $1 à $9 : Argument 1 à 9 passés au script
  - $# : Nombre d'arguments passés au script
  - $? : Résultat de la commande précédente
  - Exemple

```
#! /bin/sh
cp $1 tata.txt
echo $?
```

  - Exécution

```
root@ensa:~/Documents/cours-shell# ./script.sh file.txt
cp: ne peut évaluer `file.txt': Aucun fichier ou dossier de ce type
1
```


Structure de contrôle « if » (1)

- Permet d'effectuer une exécution conditionnelle

```
if [ expression ]
then
    # commandes à exécuter si la cond. est vraie
else
    # commandes à exécuter si la cond. est vraie
fi
```

- L'expression est constituée d'opérateurs
  - Liste des opérateurs numériques
    - -eq : Egalité (Equals)
    - -ne : Non égalité (Non Equals)
    - -lt : Infériorité (Less than)
    - -le : Infériorité ou égalité (Less than)
    - -gt : Supériorité (Greater then)
    - -ge : Supériorité ou égalité (Greater equals)

Structure de contrôle « if » (2)

- Liste des opérateurs sur chaîne de caractère
  - -z : Chaîne vide
  - -n : Chaîne non vide
  - = : Egalité de chaîne
  - != : Non égalité de chaîne
- Liste des opérateurs sur fichiers
  - -L : Lien symbolique
  - -d , -f : Répertoire, Fichier
  - -s : Fichier vide
  - -r, -w, -x : Droits qui s'appliquent (Lecteur, Ecriture, Exécution)
  - -nt : Plus récent (Newer than)
  - -ot : Plus vieux (Older than)

Structure de contrôle « if » (3)

- Liste des opérateurs logiques
  - ! : Négation
  - -a : Et (And)
  - -o : Ou (Or)
- Exemples

```
# Teste si le paramètre $1 est egal à 2
if [ $1 -eq 2 ]
then
   # Commande
fi
```

```
# Teste si le paramètre $1 est un fichier OU un répertoire
if [ -d $1 -o -f $2 ]
then
   # Commande
fi
```

```
# Teste si le fichier existe
if [ ! -f "/etc/toto.conf«  ]
then
   # Commande exécutée si le fichier n'existe pas
fi
```


Communication interprocessus

Toumanari – le 16, 17 et 18 décembre 2010

Généralités

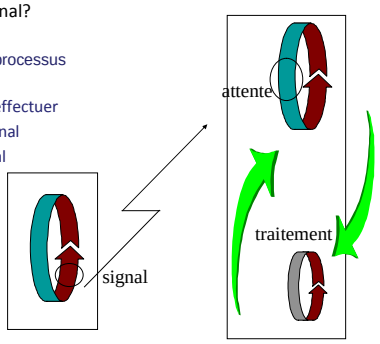
- Communications internes:

  entre processus sur une même machine:
  - exec()
  - fichiers
  - moyens de communications Unix/Linux
    - signaux
    - tubes
    - IPCs:
      - » file de messages
      - » mémoire partagée
      - » sémaphores

Les signaux

- Plusieurs processus se partagent:
  - la mémoire
  - le processeur
- Interruption logicielle envoyée à un processus
  - signal 'pendant' si non pris en compte
  - signal 'délivré' si pris en compte
- Identification par un entier
- Traitement à effectuer
  - traitement par défaut
  - handler: fonction sans retour de valeur
- Réactiver le signal après utilisation

- Qu'est-ce qu'un signal?
  - interruption d'un processus
  - fonctions utiles
    - traitement à effectuer
    - attente du signal
    - envoi du signal



- Quelques utilisations d'un signal
  - cas 1: demande d'**E/S occupée**
    - processus endormi jusqu'à E/S libérée
    - Unix envoie un **signal** à tous les processus prêts
  - cas 2: **déconnexion** d'un terminal
    - tous les processus en cours recoivent un **signal SIGHUP** et s'arrêtent
  - cas 3: fin d'un processus fils par **exit()**
    - un **signal SIGCHLD** est envoyé au père qui est en attente wait()
  - cas 4: entre **processus utilisateurs**
    - un **signal SIGUSR** est envoyé par un processus à un autre processus

- fork() et exec()
  - Après fork()
    - fils hérite des traitements ou handler
  - Après exec()
    - traitements perdus
    - signaux ignorés restent ignorés
    - les autres reprennent leur traitement par défaut

- **signal()** **Traitement d'un signal**

  - **Interface**

  #include <signal.h>
  int (*signal(sig, func))()
  int sig;
  void (*func)();

  - **Retour**:

  si ok adresse de la fonction qui était associée avant au signal
  sinon erreur -1

- **description**
  - sur la réception du signal sig
    si func = SIG_DFL
    . action par défaut (arrêt du processus)
    si func = SIG_IGN
    . signal ignoré
    si func = adresse sp interruption
    . sig déclenche l'exécution de sp
    sp a un argument : n° du signal déclencheur

- refaire signal() pour réenclencher l'action

Attente d'un signal

- **pause()**
  - **Interface**

  #include <signal.h>
  int pause()

  - **Retour**:

  si ok 0
  sinon erreur -1

  - **Description**

  Attente d'un signal particulier par un processus
  Si ce signal est ignoré, le processus est tué

Envoi d'un signal

- **kill()**
  - Interface

  #include <signal.h>
  int kill(pid, sig)
  int pid, sig;

  - Retour:

  si envoyé 0
  sinon erreur -1

- **description**

si sig = 0

. vérification processus pid existe

si sig<>0

. signal sig envoyé à un processus

si pid > 0

. pid du processus a qui est envoyé le signal

si pid = 0

. signal envoyé à tous les processus de même groupe que le processus émetteur

si pid = -1

signal envoyé à tous les processus ayant même n° utilisateur

Déclenchement alarme

- **alarm()**
  - **Interface**

```
#include <signal.h>
int alarm(seconds);
unsigned seconds;
```

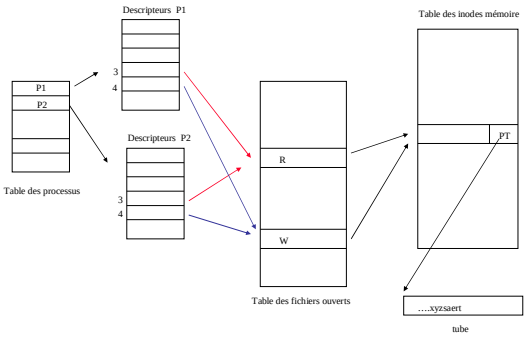
  - **Retour**:

si ok valeur restante lors de l'arrivée du signal SIGALRM

sinon erreur -1

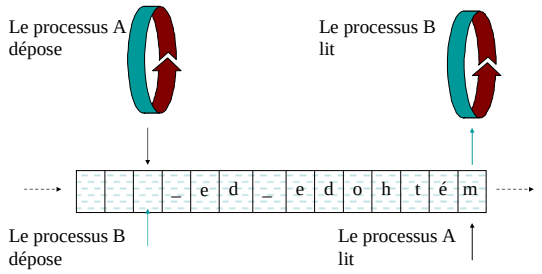
Les Tubes

- **LES TUBES OU PIPE**
  - Types:
    - tube anonyme
    - tube nommé
  - Moyen de communication entre deux processus s'exécutant sur une même machine
  - Fichiers particuliers (SGF)
  - Gérés par le noyau
  - File de données en mémoire (FIFO)
  - Lectures destructrices

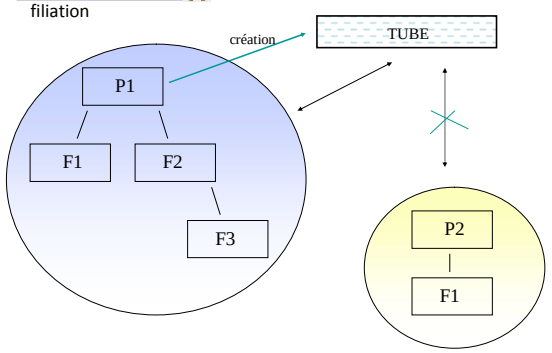


- **TUBE ANONYME**
  - Structure sans nom
  - Communication entre deux processus
  - Deux descripteurs: lecture et écriture
  - Deux pointeurs automatiques: lecture et écriture
    - pointeur de lecture sur le 1er caratère non lu
    - pointeur d'écriture sur le 1er emplacement vide
  - Processus de même filiation

- Principe:
  - pipe(): création du tube par le père
  - fork(): création du processus fils
  - héritage de l'ouverture du tube (fichier)
  - exec(): passage des descripteurs en paramètres

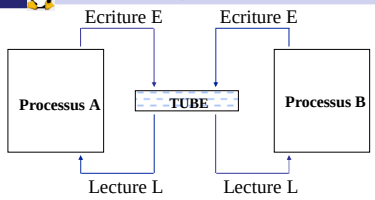


filiation



- tube vide et
  - lecture
    - » code erreur = 0
    - » processus bloqué jusqu'à dépôt de données
- tube non vide et
  - nombre de données à lire > données existantes
    - » code erreur = 0
    - » processus bloqué jusqu'à dépôt de données
- tube plein et
  - écriture
    - » code erreur = 0
    - » processus bloqué jusqu'à lecture de données

Synchronisation

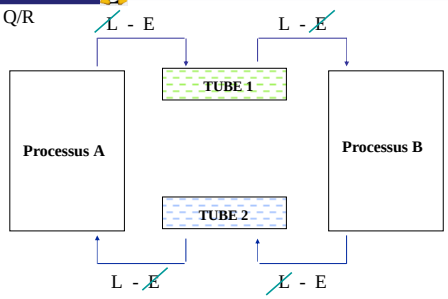


- Soit: PA transmet à PB ou PB transmet à PA

SI

- PA dépose et PA lit => PB bloqué
- PA et PB déposent et PB lit => risque que PB lise sa propre donnée

Modèle Q/R



Fermeture des descripteurs inutiles

- FONCTION TUBE

pipe (tab)

- crée un tube
- retourne les n° des deux descripteurs dans un tableau 'tab'
  - tab[0]: n° du descripteur de lecture: dl
  - tab[1]: n° du descripteur d'écriture: de
- remplit la fonction d'ouverture d'un fichier classique

- FONCTIONS SGF

read (dl, buf, nb)
- dl: n° descripteur lecture
- buf : zone de réception des octets
- nb : nombre d'octets à lire

write (de, buf, nb)
- de: n° du descripteur écriture
- buf: zone d'émission des octets
- nb: nombre d'octets à écrire

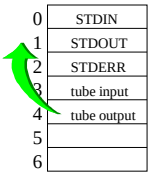
close (dl) et close (de)
- fermeture des descripteurs
- fermeture des descripteurs automatique si processus terminé
- suppression du tube si fermeture de tous les descripteurs

- EXEMPLE TUBE ANONYME

```
#include <stdio.h>
int pip[2];                              /* descripteur de pipe */
char buf [6];

{ main()
pipe(pip);                                        /*  creation pipe  */
switch (fork())
      {case -1: perror("pipe");  exit(1);
      case 0:   fils();
      default:  pere();}
pere(){write (pip[1],"hello",5); exit(0);}     /* écriture pipe */
fils() {read (pip[0],buf,5); exit(0);}                /* lecture pipe */
}
```


Uilisation de dup() pour rediriger la sortie standard

descripteurs du processus A



DUP()

1) on crée un tube:

- deux descripteurs: 3 et 4 qui pointent sur la table des fichiers: ici tube

2) on ferme le descripteur 1

- l'entrée 1 est libre

3) on duplique le descripteur 4 avec retour = dup (4)

- le descripteur 4 est recopié dans le descripteur 1 (dup prend la pemière entrée libre)

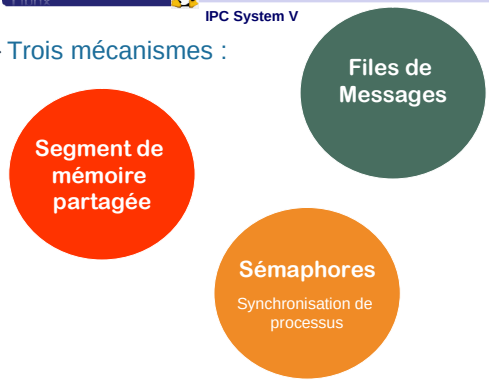
- valeur de retour: le nouveau descripteur ici le 1

4) on ferme les descripteurs 3 et 4 qui ne servent plus

5) tout envoi vers le descripteur 1 concernera le tube

IPC System V

- Trois mécanismes :



Caractéristiques communes

- Chaque objet IPC possède une identification interne
  - Permet d'accéder en programmation à l'objet considéré
  - Equivalent d'un descripteur pour un fichier
  - Des primitives C permettent d'obtenir l'id. interne
    - Ex: semget() donne le semid du sémaphore
- Chaque objet IPC possède une clé pour l'identification externe
  - Clé de type key_t
  - Equivalent au nom pour un fichier

Analogie objet IPC / fichier

- FICHIER
  - Identifiant externe : son **nom**
    - Le fichier « ensa.txt »
  - Identifiant interne : son **descripteur** (renvoyé par open)
    - Le descripteur de « ensa.txt » est égal à 3
- Objet IPC (p. ex. sémaphore)
  - Identifiant externe : sa **clé**
    - Le sémaphore 52
  - Identifiant interne : son **semid** (renvoyé par semget)
    - Le semid du sémaphore 52 est 12345

La clé IPC_PRIVATE

- Un objet IPC peut ne pas avoir de nom (de clé)
  - On le crée avec une clé égale à IPC_PRIVATE
  - Il ne peut donc être utilisé que par son créateur
    - Ou éventuellement ses fils (fork())
- En effet :
  - Pour manipuler à un IPC, il faut avoir son identifiant interne
  - Or, on l'obtient à partir de sa clé
  - Donc, si l'objet n'a pas de clé…

Le problème des clés

- L'identifiant externe doit être unique !
  - P. ex. on ne peut avoir deux sémaphores de clé 52
- Une solution « acceptable » : la primitive ftok
  - Construit une clé à partir d'un nom de fichier et d'un entier

A un fichier donné et un nombre donné correspond une unique clé.
Si on fournit un nom de fichier personnel, il y a de fortes chances que la clé obtenue soit unique

Le problème des clés

Nom d'un fichier perso

```
#include  <sys/ipc.h>
key_t ftok(const char *référence, int nombre);
```

Clé Entier

Les commandes Unix

- ipcs : affiche les objets IPC actifs sur la machine Unix

```
$ ipcs

------ Segment de mémoire partagé --------
clé     shmid    propriétaire perms   octets   nattch  états

------ Tableaux de sémaphores --------
clé     semid    propriétaire perms   nsems

------ Queues de messages --------
clé     msqid    propriétaire perms   octets-utilisés messages

$ ipcrm shm 1627649
```

shmid

- ipcrm : supprime un objet IPC
  - Normalement, cela est fait en programmation…
    - P. ex. primitive **shmctl** avec option **IPC_RMID**

Struct « ipc_perm »

Pour chaque IPC, le système utilise une structure commune, de type struct ipc_perm définie dans le fichier <sys/ipc.h>, pour enregistrer les informations permettant de déterminer les autorisations concernant les opérations ipc

```
struct ipc_perm
{
 uid_t  uid;          /*Propriétaire actuel de l'IPC. */
 gid_t  gid;          /* Groupe du propriétaire de l'IPC. */
 uid_t  cuid;         /* Créateurde l'IPC. */
 gid_t  cgid;         /* Groupe du créateur de l'IPC. */
 mode_t  mode;        /* Droits d'accès à l'IPC (r et w) . */
 key_t  key;          /* Clé de l'IPC*/
 unsigned short seq;  /* Compteur d'utilisation de l'entrée IPC*/
};
```

Principe initial (Dijkstra)

- Sémaphore : entier valant 0 ou 1 et muni de deux opérations atomiques :
  - Opération P : abaisser le sémaphore
    - Si le sémaphore vaut 1, il est décrémenté.
    - S'il vaut 0, le processus est mis en sommeil.
  - Opération V : lever le sémaphore
    - Le sémaphore est mis à 1.
    - S'il valait 0, les processus endormis en attente de son incrémentation sont réveillés.

Principe initial des sémaphores

Algorithme d'accès exclusif à une ressource partagé (p. ex. fichier ou s.m.p.)

- On associe un sémaphore à la ressource
  - Initialisé à 1
- Algorithme à suivre par chaque processus :
  - Opération P
  - Accès exclusif
  - Opération V

L'implémentation Unix

- Sémaphore : entier naturel S muni de trois opérations
  - Opération Pn :
    - Si S □ n, S = S – n
    - Si S < n, mise en sommeil du processus
  - Opération Vm
    - S = S + m et les processus endormis sont réveillés
  - Opération Z
    - Mise en sommeil jusqu'à ce que S soit égal à 0

L'implémentation Unix

- Implémentation System V plus complète que les sémaphores Dijkstra
  - Un sémaphore peut valoir un nombre infini de valeur (pas seulement 0 ou 1)
  - On peut manipuler des ensembles de sémaphores
- Donc cadre d'utilisation plus étendu :

→ **synchronisation de processus**

  - Exclusion mutuelle
    - accès à un ou plusieurs exemplaires de plusieurs ressources critiques
  - Ordonnancement de processus

etc.

Les primitives : semget

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget(key_t clef, int nb, int drap) ;
```

Nb de sémaphores dans l'ensemble

-1 ou semid (id. interne)

- Clé (identifiant externe)
- IPC_PRIVATE

- IPC_CREAT|0xyz
- 0

❖ Création d'un ensemble de sémaphore…
- Si drap vaut IPC_CREAT (IPC_CREAT | IPC_EXCL)
- Son identifiant externe sera clef ou il n'en aura pas (IPC_PRIVATE)

❖ … ou récupération du semid d'un ensemble déjà existant (créé par un autre processus)
- Si drap vaut 0

Les primitives : exemple

```
#include ...
int main()
{ key_t clef; int semid;

  clef=ftok("toto",56);
  semid=semget(clef,1,IPC_CREAT|0660);
  ...
}
```

Création d'un sémaphore

```
#include ...
int main()
{ key_t LaClef; int id;

  LaClef=ftok("toto",56);
  id=semget(LaClef,1,0);
  ...
}
```

Obtention de l'identifiant interne du sémaphore créé ci-dessus

Les primitives : semctl

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semctl(int semid, int semnum, int option, ??);
```

Numéro du sémaphore dans l'ens.

-1 si échec

Semid de l'ens. sur lequel on veut agir

Opération à réaliser

❖ Réalisation d'opérations de contrôle sur l'ensemble de sémaphores
- Initialisation
- Suppression
- Prise d'information
  - Valeur du sémaphore

2 ème arg = 0

cmd →				
GETVAL	Sémaphore (semid, semnum)	GETALL	Ensemble de sémaphores	IPC_STAT
SETVAL		SETALL		IPC_SET
GETPID				IPC_RMID
GETNCNT				
GETZCNT				

Les primitives : semctl

IPC_RMID : suppression du semaphore identifié par semid. Cette destruction est différée si le semaphoreest encore utilisé avec d'autres processus.

IPC_STAT : demande d'information sur le segment par le remplissage de la structure pointée par struct semid_ds buf (4 ème arg de semctl() à partir de la struct sem_perm de l'ensemble du sémaphore .

IPC_SET : demande de modification des caractéristiques de l'entrée identifiée par shmid avec les informations contenues dans la structure pointée par buf. Les seuls champs modifiables sont sem_perm.uid, sem_perm.gid et sem_perm.mode.

Les primitives : exemple

```
#include ...
int main()
{ key_t clef; int semid;

   clef=fotk("Valentine",56);
   semid=semget(clef,1,IPC_CREAT|0660);

   semctl(semid,0,SETVAL,0);
   ...
}
```

Initialisation du sémaphore 0 à 0

Les primitives : semop

```
struct sembuf {
  unsigned short int   sem_num ;
  short                sem_op ;
  short                sem_flg ;
}
```

Num. du sémaphore dans l'ens.

Opération à réaliser

Option

- La structure sembuf permet de définir l'opération à réaliser
  - sem_op < 0 : opération Pn
  - sem_op > 0 : opération Vn
  - sem_op = 0 : opération Z
  - Sem_flg = SEM_UNDO, IPC_NOWAIT

Linux annule automatiquement l'opération sur le sémaphore lorsque le processus se termine

Les primitives : semop

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semop(int semid, struct sembuf *tab_op, int nb_op);
```

Nb d'opérations à réaliser

-1 si échec

Tableau de structure **sembuf**

- Réalisation d'une ou plusieurs opérations P, V, Z définies dans le tableau de structure sembuf tab_op

Les primitives : exemple

```
#include ...
int main()
{ key_t clef; int semid; struct sembuf st;

   clef=fotk("toto",56);
   semid=semget(clef,1,IPC_CREAT|0660);
   semctl(semid,0,SETVAL,0);

   st.sem_op=-1;              // opération P1
   st.sem_num=0;              // sur sémaphore numéro 0
   st.sem_flg=0;        // pas d'option
   semop(semid,&st,1);
   ...
}
```

Opération P1 sur le sémaphore

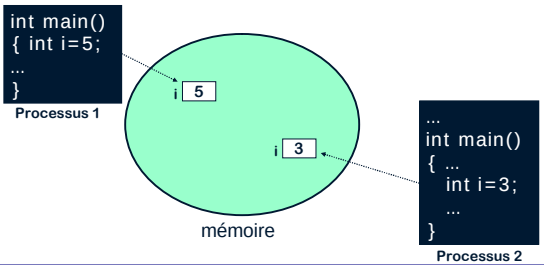
Les primitives : exemple

```
#include ...
int main()
{ key_t clef; int semid; struct sembuf st;

   clef=fotk("toto",56);
   semid=semget(clef,1,IPC_CREAT|0660);
   semctl(semid,0,SETVAL,0);

   st.sem_op=-1;              // opération P1
   st.sem_num=0;              // sur sémaphore numéro 0
   st.sem_flg=0;        // pas d'option
   semop(semid,&st,1);
   ...
   semctl(semid,0,IPC_RMID);
   ...
}
```

Suppression de l'ens. de sémaphore

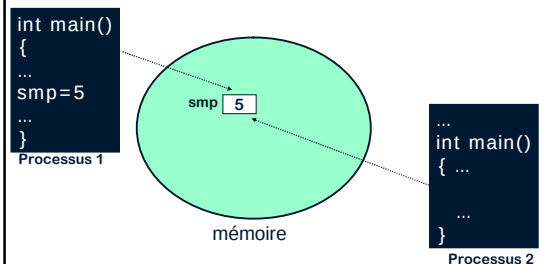
Ségment mémoire partagée

- Chaque processus possède son propre espace d'adressage
  - Il a sa zone mémoire à lui !



Ségment mémoire partagée

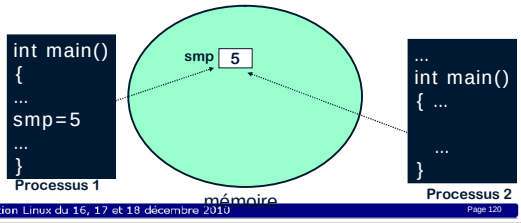
- S.m.p. : zone mémoire commune à plusieurs processus



Ségment mémoire partagée

- ATTENTION : s.m.p. = section critique !
  - Plusieurs processus sont susceptibles d'accéder en même temps à cette ressource partagée !
  - Donc : un problème d'exclusion mutuelle à résoudre.
  - Solution ? ? ?

Sémaphore



Les primitives : shmget

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t clef, int size, int drap) ;
```

Taille du s.m.p. en octets

-1 ou shmid (id. interne)

- Clé (identifiant externe)
- IPC_PRIVATE

- IPC_CREAT|0xyz
- 0

❖ Création d'un s.m.p…

- Si drap vaut IPC_CREAT
- Son identifiant externe sera clef ou il n'en aura pas (IPC_PRIVATE)

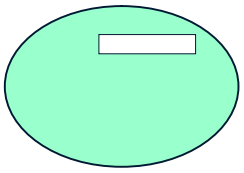
❖ … ou récupération du shmid d'un s.m.p. déjà existant (créé par un autre processus)

- Si drap vaut 0

Les primitives : exemple

```
#include ...

int main()
{ key_t clef; int shmid;

   clef=ftok("ensa",28);
   shmid=shmget(clef,1024,IPC_CREAT|0640);
   ...
}
```

Création d'un s.m.p.



Les primitives : shmat

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
void* shmat(int shmid, void *buf, int shmflag) ;
```

0

-1 ou adresse d'attachement

Id interne du s.m.p.

- 0 (lecture/écriture)
- SHM_RDONLY

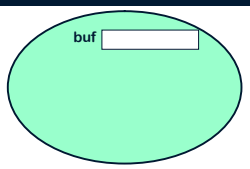
❖ Association d'un s.m.p. à une variable

- Et typage du s.m.p.

Les primitives : exemple

```
#include ...

int main()
{ key_t clef; int shmid; char *buf;

   clef=ftok("ensa",28);
   shmid=shmget(clef,1024,IPC_CREAT|0640);
   buf=shmat(shmid,0,0);
...
}
```

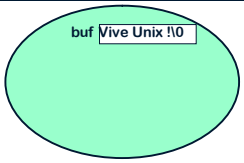
Attachement



Les primitives : exemple

```
#include ...
int main()
{ key_t clef; int shmid; char *buf;
   clef=ftok("ensa",28);
   shmid=shmget(clef,1024,IPC_CREAT|0640);
   buf=shmat(shmid,0,0);
   strcpy(buf,"Vive Unix !");
...}
```

Attachement

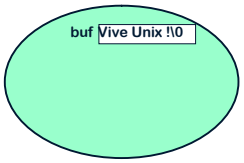


Les primitives : exemple

```
#include ...
int main()
{ key_t clef; int shmid; char *buf;
   clef=ftok("ensa",28);
   shmid=shmget(clef,1024,IPC_CREAT|0
   ;
   buf=shmat(shmid,0,0);
   strcpy(buf,"Vive Unix !");
   ... }
```

```
#include ...
int main()
{ key_t cle; int id; char *toto;
   cle=ftok("ensa",28);
   id=shmget(clef,1024,0);
   toto=shmat(id,0,0);
   printf("%s\n",toto);
   ...
}
```

A l'écran:
Vive Unix!



Les primitives : shmdt

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmdt(void *buf) ;
```

-1 si erreur

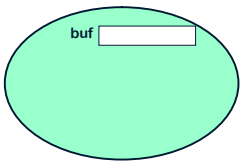
Adresse retournée par shmat().

- Rompt l'association d'un s.m.p. à une variable

Les primitives : exemple

```
#include ...
int main()
{ key_t clef; int shmid; char *buf;
   clef=ftok("ensa",28);
   shmid=shmget(clef,1024,IPC_CREAT|0640);
   buf=shmat(shmid,0,0);
   shmdt(buf);
...  }
```

Détachement



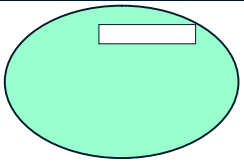
Détachement et suppression

- Un détachement n'implique pas la suppression du s.m.p.
- Il faut la demander explicitement
  - Primitive de contrôle shmctl avec l'option IPC_RMID
- Dans ce cas, la suppression aura lieu automatiquement dès que plus aucun processus ne sera attaché au s.m.p.

Les primitives : exemple

Suppression

```
#include …
int main()
{ key_t clef; int shmid; char *buf;

   clef=ftok("Bécassine",28);
   shmid=shmget(clef,1024,IPC_CREAT|0640);
   buf=shmat(shmid,0,0);
   shmdt(buf);
   shmctl(shmid,0,IPC_RMID,0);
…  }
```



struct shmid_ds dans <sys/shm.h>

```
struct ipc_perm shm_perm;
int shm_segsz; /* taille segment */
ushort shm_cpid; /* PID créateur segment */
ushort shm_lpid; /* PID dernière operation */
short shm_nattch; /* Nombre d'attachements */
time_t shm_atime; /* Heure dernier attachement */
time_t shm_dtime; /* Heure dernier détachement */
time_t shm_ctime; /* Heure derniere modification */
```


Suite ….

- Files de messages (un autre objet IPC ….)
- Pthread (Attributs d'un thread, ordonnancement, synchronisation…)
- Embarquer (Compilateurs croisés, RFS, …)
- Temps réel (RTAI RTLINUX, Xénomai …..)