DIU Enseigner l'Informatique au Lycée

Systèmes d'exploitation

kn@lri.fr
http://www.lri.fr/~kn

Plan

Système d'exploitation

Quelques systèmes:

- Windows XP/NT/2003/7/8, ...
- Linux, FreeBSD, NetBSD, OpenBSD, ...
- MacOS X (basé sur une variante de FreeBSD), ...
- Unix, AIX, Solaris, HP-UX, ...
- Symbian OS (Nokia), iOS, Android, ...

Système d'exploitation

Qu'est-ce qu'un système d'exploitation ?

- c'est un *programme*
- qui *organise* l'accès aux *ressources* de la machine

Quelles sont les ressources d'une machine?

- Processeur (temps d'exécution)
- Mémoire
- Accès aux périphériques de stockage
- Accès aux périphériques d'entrées/sorties
- ...

Système d'exploitation

Haut niveau — *Applications*: navigateur Web, éditeur de texte, anti-virus, jeu, compilateur, ...

↓

Système d'exploitation:

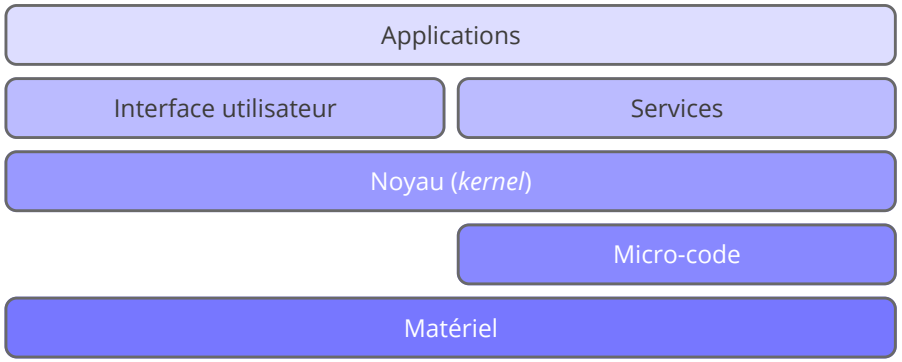
- Gestion des ressources
- Interface avec le matériel (pilotes)

Bas niveau — *Matériel*: CPU, mémoire, périphériques, ...

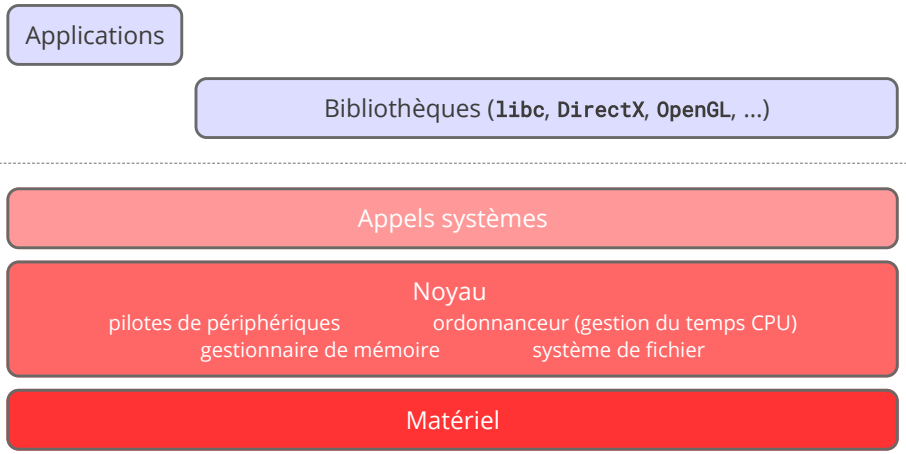
Le système Unix

1965 :	MultICS: *Multiplexed Information and Computing Service* (Bell & MIT)
1969 :	Unix: 1ère version en assembleur (AT&T)
1972-73 :	Unix réécrit en C
1976 :	Invention de TCP/IP
1977 :	*Berkeley Software Distribution* (BSD)
1985 :	Unix System V
1988 :	Minix
1992 :	Linux

Unix : architecture



Zoom sur le noyau



Le *Shell* Unix

- Interface utilisateur *en mode texte*

L'utilisateur écrit des commandes dont le résultat est affiché à l'écran

- Interface « historique » sous Unix
- Expose à l'utilisateur certains concepts Unix (permissions, propriétaires, processus, ...)
- Ces concepts sont importants pour pouvoir concevoir des sites Web

Le *Shell* Unix

Exemple de session *shell*:

```
$ ls
Documents   Downloads   Public    Person
$ cd Documents
$ ls
compte_rendu.txt
$ mv compte_rendu.txt cr.txt
$ ls
cr.txt
```


Plan

Système de gestion de fichiers (*filesystem*)

- *Organise* les données sur le support physique
- Protège contre les *corruptions de données*
- Gestion optimale de l'espace disponible
- *Accès efficace* aux données
- *Abstraction* du support physique (DVD, mémoire flash, disque réseau, ...)
- Enregistrement des *méta-données* (date de création, propriétaire, taille, ...)

Le concept de *fichier*

Un fichier est une *collection d'informations numériques* réunies sous un même *nom* et enregistrée sur un support de stockage

- Manipulable comme une unité
- Selon les systèmes, le *nom* a plus ou moins d'importance
- possède un type

Le concept de *fichier*

Ne pas confondre:

- type du fichier: il influe sur le comportement du système (fichier « normal », répertoire, lien (raccourcis), fichier système, ...). C'est une méta-donnée conservée par le système de fichier
- type du contenu: le type des *données* contenues dans le fichier:
  - DOS puis Windows: l'extension (les 3 derniers caractères après le « . ») détermine le type de contenu
  - MacOS puis OS X/iOS: les premiers octets du fichier déterminent son type
  - Premiers octets ou extension, selon les interfaces utilisées

Les attributs d'un fichier

Nom **:**
Propriétaire **:** utilisateur qui possède ce fichier
Groupe **:** groupe d'utilisateurs qui possède ce fichier
Emplacement **:** localisation du fichier sur le support physique
Taille **:** en octet (peut être la taille réelle ou la taille occupée sur le support)
Permissions **:** « qui a quel droit » sur le fichier (lecture, écriture, exécution, ...)
Type **:**
Dates **:** dernier accès, dernière modification, création, ...

Organisation logique des fichiers

Usuellement, les fichiers sont regroupés en *répertoires*. Les répertoires sont imbriqués les uns dans les autres de manière a former une *arborescence*.

Sous Unix il y a un répertoire racine, « */* » (*slash*) qui contient toute l'arborescence du système.
Chaque utilisateur possède aussi un répertoire personnel

Noms de fichiers et chemins

Un chemin est une *liste de répertoire* à traverser pour atteindre un fichier ou répertoire donné. Sous Unix, le séparateur de chemin est le « / »
Les chemins absolus commencent par un `/` et dénotent des fichiers à partir de la racine. Exemple:

```
/home/kim/Documents/ProgInternet/cours01.pdf
```

Les chemins relatifs dénotent des fichiers à partir du répertoire courant. Exemple:

```
Documents/ProgInternet/cours01.pdf
```

si on se trouve dans le répertoire **`/home/kim`**

Les noms spéciaux: « . » dénote le répertoire courant, « .. » le répertoire parent, « ~ » le répertoire de l'utilisateur et « ~**`toto`** » le répertoire de l'utilisateur **`toto`**

Utilisation du *Shell*

Le *shell* affiche un *invite de commande* (*prompt*). Exemple:

```
kim@machine $
```

On peut alors saisir une commande:

```
kim@machine $ ls *.txt
```

Le shell affiche la *sortie* de la commande:

```
fichier1.txt   fichier2.txt
```

Certains caractères doivent être précédés d'un « \ » (échappés):

```
kim@machine $ ls mon\ fichier\#1.txt
```


La ligne de commande

Une ligne de commande a la forme:

```
prog item1 item2 item3 item4 …
```

1. Si **`prog`** est un chemin il doit dénoter *un fichier exécutable*
2. Si **`prog`** est un simple nom, il doit dénoter un fichier exécutable se trouvant dans un des *répertoires prédéfinis* (**`/bin`**, **`/usr/bin`**, ...)
3. Pour chaque $\texttt{item}_i$ (séparés par un ou plusieurs espaces non échappés) le *shell* fait une *expansion de nom*
4. La liste de toutes les chaînes de caractères expansées est passée comme argument au programme **`prog`**

Expansion des noms / Expressions régulières glob

Certains caractères sont *interprétés* de manière spéciale par le *shell*. Ces caractères sont « expansés » selon des règles. Si la forme *expansée* correspond a un ou plusieurs fichiers existants, alors leurs noms sont placés sur la ligne de commande. Sinon la chaîne de caractère de départ garde sa valeur textuelle.

Expressions régulières glob

Règles d'expansion: `*` n'importe quelle chaîne
`?` n'importe quel caractère
`[ab12…]` un caractère dans la liste
`[^ab12…]` un caractère absent de liste
`[a-z]` un caractère dans l'intervalle
`[^a-z]` un caractère absent de l'intervalle
`?(`m_1`|…|`m_n`)` `@(`m_1`|…|`m_n`)` `*(`m_1`|…|`m_n`)` `+(`m_1`|…|`m_n`)`
k motifs parmi m_i
`?`: $0 \leq k \leq 1$ `@`: $k = 1$ `*`: $k \geq 0$ `+`: $k \geq 1$
`!(`m_1`|…|`m_n`)`: ni m_1, ..., ni m_n

Expressions régulières glob : exemples

`ls !(*[aeiouy]?)` La chaîne « `!(*[aeiouy]?)` » est remplacée par la liste de tous les fichiers dont l'avant dernière lettre du nom n'est pas une voyelle. S'il n'y a pas de tel fichier, la chaîne « `!(*[aeiouy]?)` » est passée à la commande `ls`.

`ls [0-9]*` affiche la liste des fichiers commençant par un chiffre

`ls +(abc)` affiche la liste des fichiers dont le nom est une répétition de « abc ».

Commandes shell de base

◆ *cd chemin* : *chemin* devient le répertoire courant. Si absent, utilise le répertoire personnel
◆ *ls* $chemin_1$ … $chemin_n$: affiche le nom des *n* fichiers. Si *n=0* affiche le contenu du répertoire courant. Avec l'option `-l` affiche la liste détaillée.
◆ *cp* $chemin_1$ $chemin_2$: copie de fichier
◆ *mv* $chemin_1$ $chemin_2$: déplacement de fichier (et renommage)
◆ *rm* $chemin_1$ … $chemin_n$: supprime les fichiers (définitif)

Droits et propriétés des fichiers

Sous Unix un utilisateur est identifié par son *login* (ou nom d'utilisateur). Chaque utilisateur est dans un *groupe principal*.
Chaque fichier appartient à un utilisateur et à un groupe.
Chaque fichier possède 3 permissions pour son propriétaire, son groupe et tous les autres. Les permissions sont lecture, écriture, exécution (plus d'autres non abordées dans ce cours).

Permission	**fichier**	**répertoire**
lecture (r)	lire le contenu du fichier	lister le contenu du répertoire
écriture (w)	écrire dans le fichier	supprimer/renommer/créer des fichiers dans le répertoire
exécution (x)	exécuter le fichier (si c'est un programme)	rentrer dans le répertoire

```
$ ls -l
drwxr-x---  9 kim prof 4096 Sep  7 21:31 Documents
```

La commande *chmod*

`chmod` *permissions* *chemin*$_1$ … *chemin*$_n$

modifie les permissions des fichiers *1* à *n*. La chaîne ***permissions*** est soit une suite de modifications de permissions *symbolique* soit l'ensemble des permissions données de manière *numérique*:

```
chmod 755 fichier.txt
chmod u-w,a+x,g=w fichier.txt
```


Permissions numériques

On groupe les *bits* de permissions par trois puis on convertit en décimal:

Utilisateur			Groupe			Autres		
r	w	x	r	w	x	r	w	x
1	1	0	1	0	0	0	0	0
6			4			0		

Le fichier est lisible et modifiable mais pas exécutable par son propriétaire, lisible pour le groupe. Les autres ne peuvent ni le lire ni le modifier.

Permissions symboliques

```
cible modifieur permission
```

- *cible* : u (utilisateur), g (groupe), o (others), a (all)
- *modifieur* : + (autorise), - (interdit), = (laisse inchangé)
- *permission* : r (lecture), w (écriture), x (exécution)

Exemple:

```
chmod u+rw,u-x,g+r,g-wx,o-rwx fichier.txt
```


Liens symboliques (1)

Pour des raisons d'organisation, on veut pouvoir « voir » le même fichier ou répertoire sous deux noms différents (ou à deux endroits différents). Par exemple:

```
$  ls -l Documents/Cours
total 8
drwxr-xr-x 3 kim prof 4096 Sep  9 11:30 Licence
drwxr-xr-x 3 kim prof 4096 Sep  9 11:30 Master

$  cd Documents/Cours/Master; ls
 Compilation    XMLProgInternet

$  cd XML_Prog_Internet; ls
cours01 cours02  cours03 cours04 cours05  cours06   Prereq

$ ls -l Prereq
lrwxrwxrwx 1 kim prof 28 Sep  9 11:30 Prereq -> ../../Licence/UnixProgWeb/
```

Liens symboliques (2)

La commande ***ln*** permet de créer des *liens symboliques*. Un lien est un petit fichier qui contient un *chemin* vers un fichier de destination.

Exemple d'utilisation

```
$ ln -s ../foo/bar/baz/toto.txt rep/titi.txt
```

crée un lien vers le fichier **toto.txt** sous le nom **titi.txt** (chacun placé dans des sous/sur répertoires)

- Ouvrir/modifier le lien > ouvre/modifie la cible
- Supprimer le lien > supprime le lien mais pas la cible
- Si la cible est un répertoire, faire **cd** nous place « dans » la cible, mais le répertoire parent est celui d'où l'on vient

Cela permet de créer l'illusion que la cible a été copiée à l'identique, sans les inconvénients

À propos de la suppression

La commande **rm fichier** efface un fichier définitivement
La commande **rm -d rep** efface un répertoire s'il est vide
La commande **rm -r rep** efface un répertoire récursivement mais demande confirmation avant d'effacer des éléments
La commande **rm -rf rep** efface un répertoire récursivement et sans confirmation

Toute suppression est définitive

Gag classique :

```
$  mkdir \~
...
$ ls
  Documents  Photos  Musique   ~
$ rm -rf ~
```

☠ ☠ ☠ ☠ ☠ ☠

Obtenir de l'aide sur une commande

La commande ***man*** permet d'obtenir de l'aide sur une commande. Lors qu'une page d'aide est affichée, on peut la faire défiler avec les touches du clavier, la quitter avec « **q** » et rechercher un mot avec la touche « **/** »

```
LS(1L)            Manuel de l'utilisateur Linux            LS(1L)
NOM
       ls, dir, vdir - Afficher le contenu d'un répertoire.
SYNOPSIS
       ls [options] [fichier...]
       Options POSIX : [-1acdilqrtuCFR]
       Options GNU (forme courte) : [-1abcdfgiklmnopqrstuxABCD
       FGLNQRSUX] [-w cols] [-T cols] [-I motif] [--full-time]
       [--format={long,verbose,commas,across,vertical,single-col
       umn}]                  [--sort={none,time,size,extension}]
       [--time={atime,access,use,ctime,status}]
       [--color[={none,auto,always}]] [--help] [--version] [--]
DESCRIPTION
       La commande ls affiche tout d'abord l'ensemble de ses
       arguments fichiers autres que des répertoires. Puis ls
       affiche l'ensemble des fichiers contenus dans chaque
       répertoire indiqué.  dir et vdir sont des versions de ls
       affichant par défaut leurs résultats avec d'autres for
       mats.
```


Recherche de fichiers

La commande **find rep** ***critères*** permet de trouver tous les fichiers se trouvant dans le répertoire **rep** (ou un sous répertoire) et répondant à certains critères. Exemples de critères :

- **-name '*toto*'** dont le nom contient **toto**
- **-iname '*toto*'** pareil, mais insensible à la casse
- **-size +200M** dont la taille sur le disque est supérieure à 200 Mo
- c_1 **-a** c_2 pour lesquels les critères c_1 et c_2 sont vrais
- c_1 **-o** c_2 pour les-quel l'un au moins des critères c_1 et c_2 est vrais
- **-user toto** qui appartiennent à l'utilisateur **toto**
- **-exec** *cmd* **{} \;** pour exécuter *cmd* sur chaque fichié trouvé. La chaîne **{}** est remplacée par le nom de fichier et **\;** sert à marquer la fin de commande.

Comment trouver toutes les options de la commande **find** ? ***man find***

Recherche de fichiers (exemples)

Trouver tous les fichiers (dans un sous-répertoire) du répertoire courant dont le *nom se finit par* ***.jpg*** et dont la taille est supérieure à 1 Mo

```
find . -name '*.jpg' -a -size +1M
```

Trouver tous les fichiers (dans un sous-répertoire) du répertoire courant dont le *nom se finit par* ***.mpg*** *(sans tenir compte de la casse)* et dont la taille est supérieure à 10 Mo, et **rajouter l'extension .bak à ces fichiers**

```
find . -iname '*.mpg' -a -size +10M -exec mv {} {}.bak \;
```


Quelques commandes utiles

- **`cat` *`fichier`*** : permet d'afficher le contenu d'un fichier dans le terminal
- **`less` *`fichier`*** : permet de lire le contenu d'un fichier (avec défilement en utilisant les flèches du clavier si le fichier est trop grand)
- **`sort` *`fichier`*** : permet d'afficher les lignes d'un fichier triées (on peut spécifier des options de tri)
- **`file` *`fichier`*** : permet de connaître le type d'un fichier
- **`wc` *`fichier`*** : permet de compter le nombre de caractères/mots/lignes d'un fichier
- **`head` *`fichier`*** : permet de garder les *n* premières lignes d'un fichier

On verra comment composer ces commandes pour exécuter des opérations complexes

Shell et entrées/sorties

Dans le *shell*, l'opérateur | permet d'enchaîner la sortie d'un programme avec l'entrée d'un autre:

```
$ ls -l *.txt | sort -n -r -k 5 | head -n 1
```

1. affiche la liste détaillée des fichiers textes
2. trie (et affiche) l'entrée standard par ordre numérique décroissant selon le 5ème champ
3. affiche la première ligne de l'entrée standard

```
-rw-rw-r    1 kim kim 1048576 Sep 24 09:20 large.txt
```


Fonctionnement des redirections

`cmd < fichier` :
: *fichier* est ouvert en lecture avant le lancement de *cmd*, le contenu est redirigé vers l'entrée standard de *cmd*.

`cmd > fichier` :
: *fichier* est ouvert en écriture avant le lancement de *cmd*. Si *fichier* n'existe pas il est créé. S'il existe il est tronqué à la taille 0. La sortie standard de *cmd* est redirigée vers *fichier*.

`cmd >> fichier` :
: *fichier* est ouvert en écriture avant le lancement de *cmd*. Si *fichier* n'existe pas il est créé. S'il existe, le curseur d'écriture est placé en fin de fichier. La sortie standard de *cmd* est redirigée vers *fichier*.

`cmd 2> fichier` :
: Comme **>** mais avec la sortie d'erreur

`cmd 2>> fichier` :
: Comme **>>** mais avec la sortie d'erreur

Attention à l'ordre d'exécution !

Quelques exemples de commandes problématiques :

```
$ sort fichier.txt > fichier.txt
```

fichier.txt devient vide ! Il est ouvert en écriture et tronqué avant l'exécution de la commande.

```
$ sort < fichier.txt > fichier.txt
```

fichier.txt devient vide ! Il est ouvert en écriture et tronqué avant l'exécution de la commande.

```
$ sort < fichier.txt >> fichier.txt
```

fichier.txt contient son contenu original, suivi de son contenu trié !

```
$ cat < fichier.txt >> fichier.txt
```

fichier.txt est rempli jusqu'à saturation de l'espace disque !

Quelques explications (1/2)

La commande **sort** doit trier son entrée standard. Elle doit donc la lire intégralement avant de produire la moindre sortie. Pour

```
$ sort < fichier.txt >> fichier.txt
```

on a donc :

1. Ouverture de *fichier.txt* en lecture
2. Ouverture de *fichier.txt* en écriture, avec le curseur positionné en fin
3. Lecture de toute l'entrée
4. Écriture de toute la sortie en fin de *fichier.txt*

Quelques explications (2/2)

La commande **cat** ré-affiche son entrée standard sur sa sortie standard. Elle peut donc lire le fichier morceaux par morceaux et les afficher au fur et à mesure. Supposons que *fichier.txt* contient **AB** :

```
$ cat < fichier.txt >> fichier.txt
```

1. Ouverture de *fichier.txt* en lecture
2. Ouverture de *fichier.txt* en écriture, avec le curseur positionné en fin
3. Lecture de A (et positionnement du curseur de lecture sur B)
4. Écriture de A en fin de fichier *fichier.txt*
5. Lecture de B (et positionnement du curseur de lecture sur A)
6. Écriture de B en fin de fichier *fichier.txt*
7. Lecture de A (et positionnement du curseur de lecture sur B)
8. Écriture de A en fin de fichier *fichier.txt*
9. ...

Conseils...

On évitera toujours de manipuler le même fichier en entrée et en sortie. Il vaut mieux rediriger vers un fichier temporaire, puis renommer ce dernier (avec la commande **mv**).

Enchainement de commandes et code de sortie

Sous Unix, chaque commande renvoie un code de sortie (un entier entre 0 et 255).
Note : lors de l'écriture d'un programme C (ou C++) c'est le fameux `int` renvoyé par la fonction :

```
int main(int argc, char **argv) { … }
```

Par convention, un code de 0 signifie terminaison normale, un code différent de 0 une erreur. On peut enchaîner des commandes de plusieurs façons :

cmd$_1$; *cmd*$_2$
: cmd$_2$ est exécutée après cmd$_1$

cmd$_1$ && *cmd*$_2$
: cmd$_2$ est exécutée après cmd$_1$ si cette dernière réussit (code de sortie 0)

cmd$_1$ || *cmd*$_2$
: cmd$_2$ est exécutée après cmd$_1$ si cette dernière échoue (code de sortie différent de 0)

Plan

Définitions

Programme : séquences d'instructions effectuant une tâche sur un ordinateur.
Exécutable : fichier binaire contenant des instructions machines interprétables par le microprocesseur.
Thread : plus petite unité de traitement (≡ séquence d'instructions) pouvant être ordonnancée par le système d'exploitation.
Processus : instance d'un programme (≡ « un programme en cours d'exécution »). Un processus est constitué de un ou plusieurs *threads*.

Exemple: programme

Dans un fichier « `counter.c` » (attention c'est du pseudo C)

```
int count = 0;
int exit = 0;

void display() {
  while (exit == 0) {
    sleep (3);
    printf("%i\n", count);
  }
}

void listen() {
  while (exit == 0) {
    wait_connect(80);
    count++;
  }
}
```

```
void main () {
  run_function(display);
  run_function(listen);
  while (getc () != '\n') { };
  exit = 1;
  return;
}
```

Exemple: programme

Compilation

```
gcc -o counter.exe counter.c
```

Le *fichier* « **counter.exe** » est un exécutable (fichier binaire contenant du code machine)

```
./counter.exe     ← il faut la permission +x sur le fichier
```

Le contenu de l'exécutable est copié en mémoire et le processeur commence à exécuter la première instruction du programme.

Exemple: *threads*

1. main
2. attente d'un évènement clavier → ←changement de *thread*
3. listen
4. attente de connexion → ←changement de *thread*
5. display (affiche 0 à l'écran)
6. attente pendant 3s → (les 3 *threads* attendent un évènement externe)
   nouvelle connexion sur le port 80 ← réveil du *thread* `listen`
7. listen (incrémente **count**)
   attente de connexion →
   ... fin des 3s
   ← réveil du *thread* **display**
8. display (affiche 1 à l'écran)

Les *threads* partagent leur mémoire (variables communes)

Exemple: processus

(différence: les processus *ne partagent pas leur espace mémoire*)

1. Exécution de counter.exe pendant 50µs
2. Exécution de firefox.exe pendant 50µs
3. Exécution du processus qui dessine le bureau pendant 50 µs
   ...

C'est le *gestionnaire de processus* qui décide quel programme a la main et pour combien de temps (priorité aux tâches critiques par exemple)
Le système d'exploitation stocke pour chaque processus un ensemble d'informations, le PCB (*Process Control Block*).

Proccess Control Block

Le PCB contient:

- l'*identificateur du processus* (pid)
- l'*état* du processus (en attente, en exécution, bloqué, ...)
- le compteur d'instructions (*i.e.* où on en est dans le programme)
- le *contexte courant* (état des registres, ...)
- position dans *la file d'attente de priorité globale*
- informations mémoire (zones allouées, zones accessibles, zones partagées)
- listes des fichiers ouverts (en lecture, en écriture), liste des connexions ouvertes, ...

...

Opérations sur les processus

- *création* et *destruction* de processus
- *suspension* et *reprise*
- *duplication* (*fork*)
- modification de la *priorité*
- modification des *permissions*

États d'un processus

Un processus change d'état au cours de son exécution

Nouveau : le processus est en cours de création

Exécution : le processus s'exécute

En attente : le processus attend un évènement particulier (saisie au clavier, écriture sur le disque, ...)

Prêt : le processus est prêt à reprendre son exécution et attend que l'OS lui rende la main

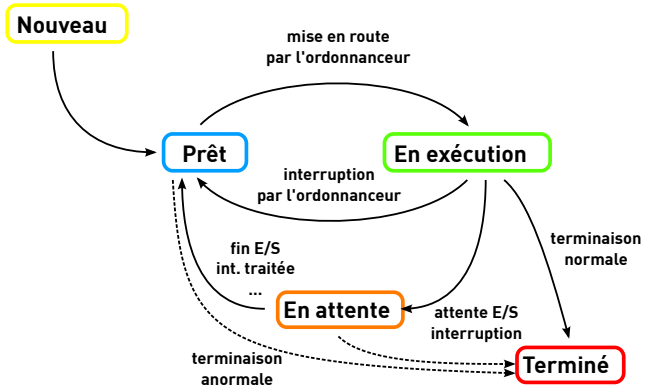
terminé : le processus a fini son exécution

États d'un processus

L'OS détermine et modifie l'état d'un processus:

- En fonction d'évènements internes au processus:
  - lecture d'un fichier (si le contenu n'est pas disponible, le processus passe de « prêt » à « en attente »)
  - le processus attends volontairement pendant x secondes
  - ...
- En fonction d'évènements externes au processus:
  - un fichier devient disponible
  - un *timer* arrive à 0
  - le matériel déclenche une *interruption*

États d'un processus



La commande *ps*

Permet d'avoir des informations sur les processus en cours d'exécution (voir « `man ps` » pour les options):

```
$ ps -o user,pid,state,cmd x
          USER     PID   S CMD
          …
          kim      27030 Z [chrome] <defunct>
          kim      27072 S /opt/google/chrome/chrome --type=renderer
          kim      29146 S bash
          kim      29834 S evince
          kim      29858 S emacs cours.xhtml
          kim      29869 R ps -o user,pid,state,cmd x
```


États des processus (sous Linux)

R : *Running* (en cours d'exécution)
S : *Interruptible sleep* (en attente, interruptible)
D : *Uninterruptible sleep* (en attente, non-interruptible)
T : *Stopped* (interrompu)
Z : *Zombie* (terminé mais toujours listé par le système)

Signaux

L'OS peut envoyer des *signaux* à un processus. Sur réception d'un signal, un processus peut interrompre son comportement normal et exécuter son *gestionnaire de signal*. Quelques signaux:

Nom	Code	Description
INT/TERM	2,15	demande au processus de se terminer
QUIT	3	interrompt le processus et produit un *dump*
KILL	9	interrompt le processus immédiatement
SEGV	11	signale au processus une erreur mémoire
STOP	24	suspend l'exécution du processus
CONT	28	reprend l'exécution d'un processus suspendu

Processus et terminal

Un processus est lié au *terminal* dans lequel il est lancé. Si on exécute un programme dans un terminal et que le processus ne rend pas la main, le terminal est bloqué

```
$ gedit
```

On peut envoyer au processus le signal *STOP* en tapant

```
ctrl-Z
```

dans le terminal:

```
$ gedit
^Z
[1]+  Stopped                 gedit
```

Le processus est suspendu, la fenêtre est gelée (ne répond plus).

Processus et terminal

On peut reprendre l'exécution du programme de deux manières:

```
$ fg
```

Reprend l'exécution du processus et le remet en avant plan (terminal bloqué)

```
$ bg
```

Reprend l'exécution du processus et le remet en arrière plan (terminal libre)
On peut lancer un programme directement en arrière plan en faisant:

```
$ gedit &
```

On peut envoyer un signal à un processus avec la commande « **`kill [-signal] pid`** »

```
$ kill -9 2345
```


Processus et entrées/sorties

Le terminal et le processus sont liés par trois fichiers spéciaux:

1. L'entrée standard (***`stdin`***), reliée au clavier
2. La sortie standard (***`stdout`***), reliée à l'affichage
3. La sortie d'erreur (***`stderr`***), reliée à l'affichage

Dans le *shell*, on peut utiliser les opérateurs **`<`**, **`>`** et ***`2>`*** pour récupérer le contenu de ***`stdin`***, ***`stdout`*** et ***`stderr`***:

```
$ sort < toto.txt
$ ls -l  > liste_fichiers.txt
$ ls -l * 2> erreurs.txt
```


Shell et entrées/sorties

Dans le *shell*, l'opérateur | permet d'enchaîner la sortie d'un programme avec l'entrée d'un autre:

```
$ ls -l *.txt | sort -n -r -k 5 | head -n 1
```

1. affiche la liste détaillée des fichiers textes
2. trie (et affiche) l'entrée standard par ordre numérique décroissant selon le 5ème champ
3. affiche la première ligne de l'entrée standard

```
-rw-rw-r    1 kim kim 1048576 Sep 24 09:20 large.txt
```


Fonctionnement des redirections

`cmd < fichier` :
: *fichier* est ouvert en lecture avant le lancement de *cmd*, le contenu est redirigé vers l'entrée standard de *cmd*.

`cmd > fichier` :
: *fichier* est ouvert en écriture avant le lancement de *cmd*. Si *fichier* n'existe pas il est créé. S'il existe il est tronqué à la taille 0. La sortie standard de *cmd* est redirigée vers *fichier*.

`cmd >> fichier` :
: *fichier* est ouvert en écriture avant le lancement de *cmd*. Si *fichier* n'existe pas il est créé. S'il existe, le curseur d'écriture est placé en fin de fichier. La sortie standard de *cmd* est redirigée vers *fichier*.

`cmd 2> fichier` :
: Comme **`>`** mais avec la sortie d'erreur

`cmd 2>> fichier` :
: Comme **`>>`** mais avec la sortie d'erreur

Attention à l'ordre d'exécution !

Quelques exemples de commandes problématiques :

```
$ sort fichier.txt > fichier.txt
```

fichier.txt devient vide ! Il est ouvert en écriture et tronqué avant l'exécution de la commande.

```
$ sort < fichier.txt > fichier.txt
```

fichier.txt devient vide ! Il est ouvert en écriture et tronqué avant l'exécution de la commande.

```
$ sort < fichier.txt >> fichier.txt
```

fichier.txt contient son contenu original, suivi de son contenu trié !

```
$ cat < fichier.txt >> fichier.txt
```

fichier.txt est rempli jusqu'à saturation de l'espace disque !

Quelques explications (1/2)

La commande `sort` doit trier son entrée standard. Elle doit donc la lire intégralement avant de produire la moindre sortie. Pour

```
$ sort < fichier.txt >> fichier.txt
```

on a donc :

1. Ouverture de *fichier.txt* en lecture
2. Ouverture de *fichier.txt* en écriture, avec le curseur positionné en fin
3. Lecture de toute l'entrée
4. Écriture de toute la sortie en fin de *fichier.txt*

Quelques explications (2/2)

La commande `cat` ré-affiche son entrée standard sur sa sortie standard. Elle peut donc lire le fichier morceaux par morceaux et les afficher au fur et à mesure. Supposons que *fichier.txt* contient **AB** :

```
$ cat < fichier.txt >> fichier.txt
```

1. Ouverture de *fichier.txt* en lecture
2. Ouverture de *fichier.txt* en écriture, avec le curseur positionné en fin
3. Lecture de A (et positionnement du curseur de lecture sur B)
4. Écriture de A en fin de fichier *fichier.txt*
5. Lecture de B (et positionnement du curseur de lecture sur A)
6. Écriture de B en fin de fichier *fichier.txt*
7. Lecture de A (et positionnement du curseur de lecture sur B)
8. Écriture de A en fin de fichier *fichier.txt*
9. ...

Conseils...

On évitera toujours de manipuler le même fichier en entrée et en sortie. Il vaut mieux rediriger vers un fichier temporaire, puis renommer ce dernier (avec la commande `mv`).

Enchainement de commandes et code de sortie

Sous Unix, chaque commande renvoie un code de sortie (un entier entre 0 et 255).
Note : lors de l'écriture d'un programme C (ou C++) c'est le fameux `int` renvoyé par la fonction :

```
int main(int argc, char **argv) { … }
```

Par convention, un code de 0 signifie terminaison normale, un code différent de 0 une erreur. On peut enchaîner des commandes de plusieurs façons :

cmd$_1$; *cmd*$_2$
: cmd$_2$ est exécutée après cmd$_1$

cmd$_1$ && *cmd*$_2$
: cmd$_2$ est exécutée après cmd$_1$ si cette dernière réussit (code de sortie 0)

cmd$_1$ || *cmd*$_2$
: cmd$_2$ est exécutée après cmd$_1$ si cette dernière échoue (code de sortie différent de 0)

Plan

Script shell

Mentalité Unix beaucoup de petits programmes spécifiques, que l'on combine au moyen de scripts pour réaliser des actions complexes. Exemple de fichier script:

```
#!/bin/bash
for i in  img_*.jpg
do
  base=$(echo "$i" | cut -f 2- -d '_')
  nouveau=photo_"$base"
  if test -f "$nouveau"
  then
     echo "Attention, le fichier $nouveau existe déjà"
     continue
  else
     echo "Renommage de $i en $nouveau"
     mv "$i" "$nouveau"
  fi
done
```


Rendre un script exécutable

Si un fichier texte (quel que soit son extension), commence par les caractères **`#!/chemin/vers/un/programme`**, on peut rendre ce fichier exécutable (**`chmod +x`**). Si on l'exécute, le contenu du fichier est passé comme argument à **`programme`** (qui est généralement un interpréteur)

`#!/bin/bash` signifie que le corps du fichier est passé au programme **`bash`** qui est l'interprète de commande (le *shell*).

Que mettre dans un script

- des commandes (comme si on les entrait dans le terminal)
- des structures de contrôle (boucles **`for`**, **`if then else`**)
- des définitions de variables

Définitions de variables

On peut définir des variables au moyen de la notation
`VARIABLE=`*`contenu`*
et on peut utiliser la variable avec la notation **`$VARIABLE`**

- Attention, pas d'espace autour du **`=`**
- nom de variable en majuscule ou minuscule
- contenu est une chaine de caractères. Si elle contient des espaces, utiliser `" ... "`

exemple de définition :

```
i=123
j="Ma super chaine"
TOTO=titi
echo $TOTO
```

exemple d'utilisation: **`echo $j $i $TOTO`**
affiche « **`Ma super chaine 123 titi`** »

Boucles `for`

Les boucles **`for`** ont la syntaxe:

```
for VARIABLE in elem_1 ... elem_n
do
    ....
done
```

chaque $\texttt{elem}_i$ est expansé (comme une ligne de commande) avant l'évaluation de la boucle:

```
for i in *.txt
do
   echo $i est un fichier texte
done
```

On peut quitter une boucle prématurément en utilisant **`break`** et passer directement au tour suivant avec **`continue`**

Conditionnelle

La syntaxe est :

```
if commande
then
    ...
else
    ...
fi
```

`commande` est évaluée. Si elle se termine avec succès, la branche **`then`** est prise. Si elle se termine avec un code d'erreur, la branche **`else`** est prise. On peut utiliser la commande **`test`** qui permet de tester plusieurs conditions (existance d'un fichier, égalités de deux nombres, ...) et se termine par un succès si le teste est vrai et par un code d'erreur dans le cas contraire

Conditionnelle (exemple)

On regarde tour à tour si fichier1.txt, fichier2.txt, ... existent :

```
for i in 1 2 3 4 5 6
do
  if test -f "fichier$i".txt
 then
    echo le fichier "fichier$i".txt existe
 fi
done
```


Sous-commandes et chaines

Il est pratique de pouvoir mettre l'affichage d'une commande dans une variable. On utilise **`$( commande ... )`**:

```
MESFICHIER=$(ls *.txt)
for i in $MESFICHIER
do
     echo Fichier: $i
done
```

Attention à la présence de guillemets autour des variables. S'il y a f1.txt et f2.txt dans le répertoire courant:

```
MESFICHIER=$(ls *.txt)
for i in $MESFICHIER
do
     echo Fichier: $i
done
affiche:
  Fichier: f1.txt
  Fichier: f2.txt
```

```
MESFICHIER=$(ls *.txt)
for i in "$MESFICHIER"
do
     echo Fichier: $i
done
affiche:
  Fichier: f1.txt f2.txt
```


Expressions arithmétiques

On peut effectuer des *calculs arithmétiques* au moyen de :

```
$(( … expression … ))
```

Par exemple :

```
echo $(( 1 + 2 + 9 + 10 + 20))
```

affiche **42** sur la sortie standard. Il est possible d'utiliser les opérations **+**, **-**, *****, **/** et **%** sur les entiers. Les entiers sont signés et leur taille dépend de l'architecture de la machine (32 ou 64 bits). On peut bien-sûr réutiliser des variables dans les expressions.

```
X=3
Y=4
Z=$(( $X + $Y ))
echo $(( $Z * 2 ))
```

affiche 14 sur la sortie standard.

Variables spéciales

Certaines variables sont prédéfinies par le *shell* :

- **`$1`**, **`$2`**, ..., **`$n`** : valeur du $n^{\text{ième}}$ argument du script courant
- **`$?`** : code de retour de la dernière commande (**`0`** si succès, **`n > 0`** en cas d'échec)
- **`$HOME`** : répertoire utilisateur de l'utilisateur courant
- **`$PWD`** : répertoire courant dans lequel s'exécute le script
- **`$USER`** : *login* de l'utilisateur courant
- **`$SHELL`** : chemin vers le *shell* courant
- **`$PATH`** : liste des répertoires dans lesquels sont cherchés les exécutables

Portée des variables

Une variable *non-définie* est automatiquent remplacée par la chaîne de caractères *vide* sans provoquer d'erreur.
Un script *hérite d'une copie* de toutes les variables du shell dans lequel il est lancé.
Une variable définie dans un script *ne survit pas à ce dernier*
La directive **`export`** permet de définir des variables qui survivent à un script ou de modifier des variables du shell depuis un script.

Commandes utiles

- **`seq`** *`m n`* : affiche la liste de tous les nombres entre *m n*
- **`echo ...`** affiche ses arguments sur la sortie standard
- **`printf "chaine" ...`** affiche ses arguments au moyen d'une chaine de format (comme le printf de C)
- **`date`** : affiche la date courante
- **`cut`** : découpe une chaine selon des caractères de séparations ou des positions

Plan

Processus de type *daemon*

Un *daemon* (prononcé démon) est un processus qui *non-interactif* qui tourne en tâche de fond (pas d'entrée/sortie sur le terminal, pas d'interface graphique, ...). On communique avec ce processus via des *signaux* ou en lisant ou écrivant dans des fichiers ou connexions réseau. Le plus souvent, leur but est de fournir un *service*

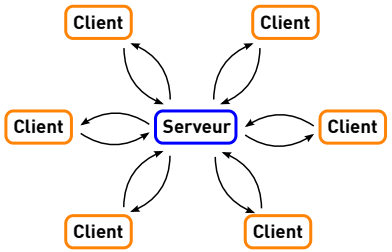
Exemple de scénario: « *Les utilisateurs doivent interagir avec le matériel. L'accès au matériel demande des droits administrateur.* »

- Solution 1 : tout le monde est administrateur (DOS, Win XP, ...)
- Solution 2 : on crée un programme particulier qui a les privilèges suffisants pour la tâche en question. Les utilisateurs communiquent avec ce programme

Quelques *daemons* sous Linux

Nom	Description
sshd	*shell* distant sécurisé
crond	exécution périodique de programmes
cupsd	serveur d'impressions
pulseaudio	serveur de son (mixe les sons des différentes applications)
udevd	détection de matériel *hotplug*
nfsd	serveur de fichier réseau
smtpd	livraison des e-mail
httpd	serveur de pages Web

Architecture client-serveur



Des processus clients communiquent avec le serveur à travers le réseau. Les clients sont indépendant et ne communiquent pas entre eux. *Attention* plusieurs clients peuvent se trouver su la même machine physique!

Architecture client-serveur

- Le serveur attend des connexions entrantes
- Les clients peuvent se connecter à tout moment
- L'application client est généralement légère, envoie une requête au serveur et attend un résultat
- Le serveur est une application plus lourde qui:
  - effectue des calculs trop coûteux pour le client
  - gère l'accès à une ressource distante partagée
  - ...

Exemples: serveur de bases de données, serveur mail, serveur Web, terminal de carte bancaire, ...