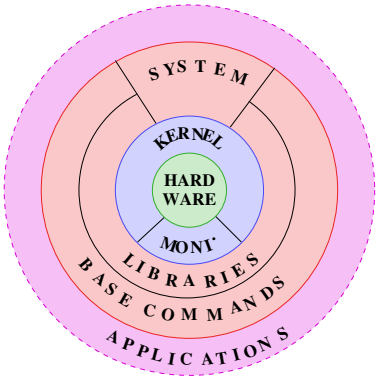
# *Système d'exploitation*

Le 12 septembre 2017 , SVN-ID 425

# Contents

# 1 Fondement

## 1.1 Organisation




**matériel** CPU, RAM, contrôleurs et périphériques.

**moniteur** Petit programme en ROM, qui tourne au démarrage de la machine.

**noyau** Gère et donne accès au matériel

**système** Couche de standardisation

**applications**

## 1.2 Système de protection

### 1.2.1 UID & GID (User & Group IDentifier)

Le système de protection est basé sur les identifiants d'utilisateurs UID et de groupes GID. Ce sont des entiers, des tables permettent de les convertir en un nom humainement compréhensible.

**utilisateur** toute personne travaillant sur une machine a ouvert une session $\Longrightarrow$ 1 UID, 1 GID principal et 0 ou plusieurs GID auxiliaires.

- 1 UID identifie un utilisateur.
- 2 utilisateurs peuvent appartenir à un même groupe.

**programme** un programme est lancé par un utilisateur $\Longrightarrow$ 1 UID, 1 GID principal et 0 ou plusieurs GID auxiliaires.

**fichier** il appartient à 1 seul utilisateur et à un seul groupe.

**création de fichier** il appartient à l'UID et au GID principal de l'utilisateur (programme) qui l'a créé.

### 1.2.2 Droits d'un fichiers

| droits d'accès | | | | propriétaire et groupe | |
|---|---|---|---|---|---|
| masque octal | format usuel | | | | |
| | prop. | group | autre | UID | GID |
| 777 | rwx | rwx | rwx | 101 | 100 |
| 600 | rw- | --- | --- | 110 | 200 |
| 060 | --- | rw- | --- | 111 | 200 |
| 005 | --- | --- | r-x | 112 | 200 |
| 644 | rw- | r-- | r-- | 113 | 200 |
| 755 | rwx | r-x | r-x | 0 | 0 |

Pour un fichier non répertoire

**r** accès en lecture

**w** accès en écriture

**x** il est possible d'essayer de le lancer

Pour un fichier répertoire

**r** les noms des fichiers du répertoire peuvent être lus

**w** un fichier du répertoire peut être créé ou détruit

**x** les fichiers du répertoire peuvent être accédés

### 1.2.3 Changement des droits

**chmod masque-octal f1 f2 ...**

- chmod 755 tutu
- chmod 640 titi

**chmod [ugoa] [+-=] [rwx] f1 f2 ...**

- chmod a-x tutu
- chmod go+rx titi toto

### 1.2.4 Exercice

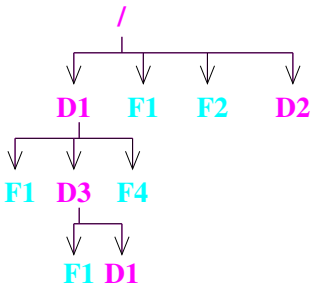
Complétez les colonnes accès de la table ci-dessous. Le répertoire D contient le fichier F.

| mon | | répertoire D | | | fichier F | | | accès | |
|---|---|---|---|---|---|---|---|---|---|
| uid | gid | uid | gid | masque | uid | gid | masque | r | w |
| * | * | * | * | 755 | * | * | 666 | | |
| 10 | 20 | 11 | 21 | **5 | 10 | 20 | 6** | | |
| 10 | 20 | 11 | 21 | **0 | 10 | 20 | 6** | | |
| 10 | 20 | 11 | 21 | **6 | 10 | 20 | 6** | | |
| 10 | 20 | 11 | 20 | *6* | 11 | 20 | *6* | | |
| 10 | 20 | 11 | 20 | *5* | 11 | 20 | *2* | | |
| 10 | 20 | 11 | 20 | *1* | 11 | 20 | *4* | | |

## 1.3 Fichiers et systèmes de fichiers

### 1.3.1 Système de fichiers

#### *1.3.1.1 Définition*

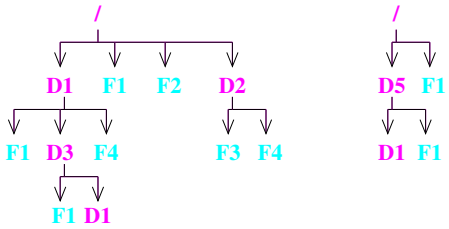



Arborescence de fichiers:

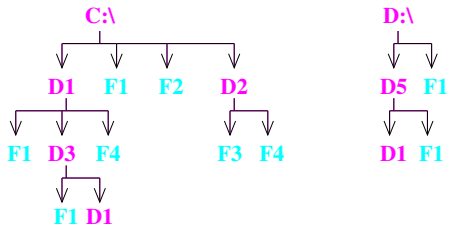
- les noeuds: fichiers répertoires
- les feuilles: fichiers ou répertoires vides
- la racine le haut de l'arbre

Support physique: disques durs, RAM

#### *1.3.1.2 Plusieurs systèmes de fichiers (Windows)*

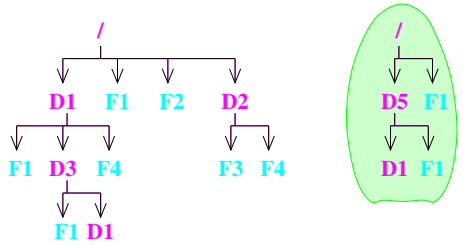



Chaque disque et/ou partition a un système de fichiers
$\Longrightarrow$ nombreux systèmes de fichiers.

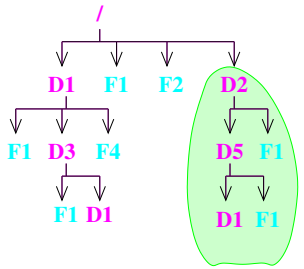



Sous Windows, les systèmes de fichiers sont visibles
$\Longrightarrow$ identifiés par une lettre suivie de ':'.

#### *1.3.1.3 Plusieurs systèmes de fichiers (Unix)*

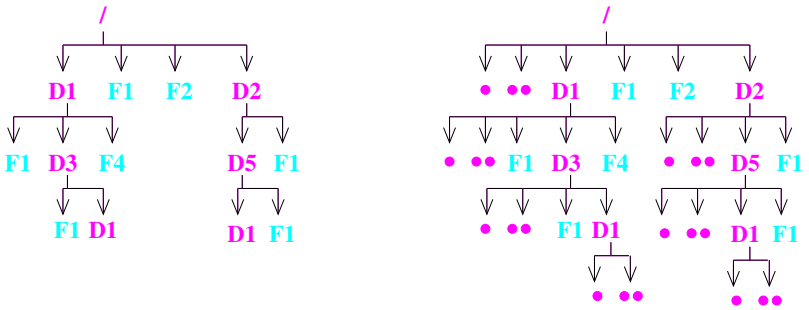



Chaque disque et/ou partition a un système de fichiers
$\Longrightarrow$ nombreux systèmes de fichiers.




Un système de fichiers principal sur le quel sont montés les systèmes de fichiers auxiliaires
$\Longrightarrow$ un seul système de fichiers.

#### 1.3.1.4 Fichiers . & ..




Tout répertoire contient au moins 2 fichiers:

- • alias du répertoire courant.
- •• alias du répertoire supérieur.
- •• alias du répertoire courant pour la racine.

### 1.3.2 Types de fichiers

**répertoire** opérations disponibles: création et suppression de fichiers.

**non répertoire** opérations disponibles: lecture, écriture, positionnement du pointeur (optionnel).

**régulier** ils sont soit binaire, soit texte, ils ont une fin et le positionnement est disponible.

**spécial** ils sont associés à des périphériques et/ou à des drivers (terminal, disque dur, flux vidéo audio, ...).

**lien** leurs contenus sont la référence d'un autre fichier.

- Lire/écrire un fichier lien revient à lire/écrire le fichier référencé.
- Un lien peut référencer un autre lien.
- Un lien mort: le référencé n'existe pas.

Quelques fichiers spéciaux:

**/dev/sda** le premier disque dur, il a une fin et le positionnement du pointeur est disponible.

**FIFO** Fichier ou tout ce qui est écrit ne peut être lu qu'une seule fois. Il est créé avec la commande mkfifo.

**/dev/ttyS0, /dev/pts/0** liaison série physique (RS232) ou émulée (terminal), pas de fin, pas de positionnement, configurable.

**/dev/null** fichier poubelle, capacité infinie.

**/dev/zero** fichier sans fin contenant que des octets nuls (0x00).

**/dev/random /dev/urandom** fichier sans fin de nombres aléatoires.

### 1.3.3 Chemins & CWD

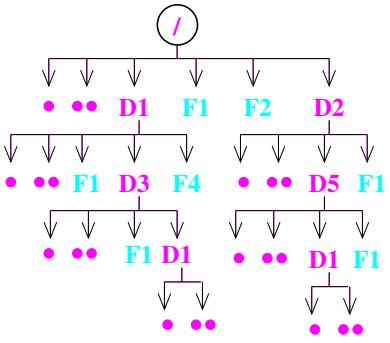
#### 1.3.3.1 Définition

Suite de noms (nom=chaine de caractères sans '/') séparés par au moins un '/' et précédés et terminés par $\emptyset$ ou plusieurs '/':
$[/]nom_0/nom_1/.../nom_n[/]$
Un chemin est valide si

- les sous chemins "$[/].../nom_i$" pour $i < n$ doivent être des répertoires de "$[/].../nom_{i-1}$.
- $nom_n$ doit être un fichier ou un répertoire du répertoire $[/]nom_0/nom_1/.../nom_{n-1}$.
- si le chemin se termine par '/', $nom_n$ doit être un répertoire.

#### 1.3.3.2 Chemins absolus

Un chemin absolu commence par un '/', on part de la racine du système de fichier.
Parmi les chemins ci-dessous indiquez ceux qui sont identiques et ceux qui n'en sont pas.



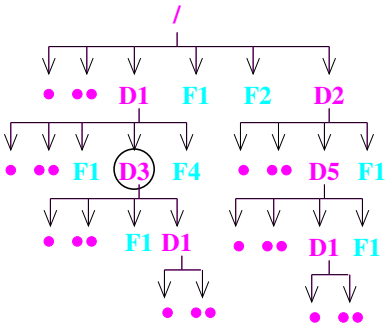

/D1/D3/D1
/D1/D3/D1/
/D1/D3/D1/.
/D1/D3/D1.
///D1//D3////D1

/D2/D5/D1/../../F1
/D2/D5/D1/../../F1/
/../D2/D5//.//D1/../../F1
/D2/D5/D1/../XX/../../F1

#### *1.3.3.3 Chemins relatifs*

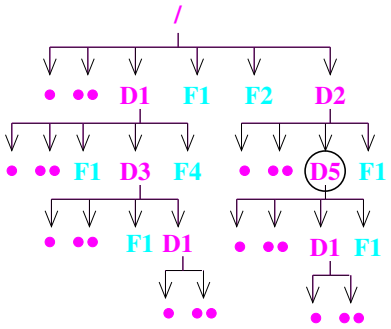
Tout programme qui tourne a un répertoire de travail associé qui s'appelle CWD ou WD (Current Working Directory).
Un chemin relatif ne commence pas par un '/', il part du répertoire CWD.

Le chemin absolu du chemin relatif CHE est: CWD/CHE




Donnez les chemins relatifs de:

- /D2/F1: ../../D2/F1
- /F1: ../../F1
- /D1/D3/F1 via D1: ../D3/F1 ou ../../D1/D3/F1
- /D1/D3/F1 le plus court ne commençant pas par 'F': ./F1




Donnez les chemins relatifs de:

- /D2/F1: ../F1
- /F1 : ../../F1
- /D1/D3/F1: ../../D1/D3/F1
- /D2/D5/F1 le plus court ne commençant pas par 'F': ./F1

### 1.3.4 Quizz

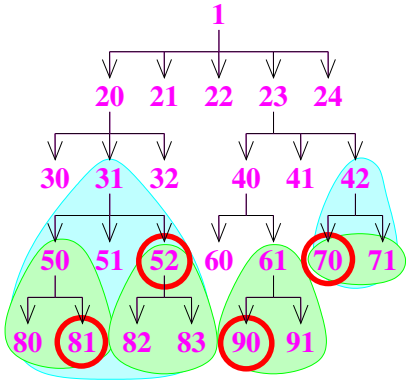
Soit un système de fichiers sans lien, indiquez si les propositions suivantes sont vraies ou fausses.

1. Il existe un chemin absolu pour chaque fichier.
2. Il existe un chemin absolu unique pour chaque fichier.
3. Soit un CWD et un fichier, il existe toujours un chemin relatif partant de ce CWD et désignant ce fichier.
4. Les chemins "F" et "./F" donnent toujours le même fichier.
5. Les chemins "../F" et "F" donnent toujours des fichiers différents.
6. Les chemins "D1/F" et "D1/D2/../F" donnent toujours le même fichier.
7. Les chemins "/F" et "F" donnent toujours des fichiers différents.

## 1.4 Processus

### 1.4.1 Définition

- Entité d'exécution $\Longrightarrow$ un programme qui tourne.
- Identifié par un numéro PID.
- Ils sont organisés en arbre.
- Ils sont groupés en session.
- Ils sont groupés en groupe terminal avec un leader.




Comme tout programme, il produit des sorties en fonction d'entrées.

### 1.4.2 Entrées/sorties

- Unité d'exécution: code, données, pile.
- Identifiant de processus.
- Identifiant d'utilisateur et identifiants de groupe.
- Identifiant de session et de groupe terminal.
- Arguments: tableau de chaines de caractères.
- variables d'environnement: tableau de chaines de caractères
  nom-var=valeur-var
- flux d'entrée et de sortie.
- statut: valeur entre 0 et 255.


### 1.4.3 Convention

#### *1.4.3.1 Arguments*

Les arguments passés à un programme sont un tableau de chaine de caractères. La seule convention est:

> Le premier argument est le nom d'invocation du programme

Le premier argument est donc souvent inutilisé mais est utile pour:

1. Écrire des messages d'erreur avec le nom du programme.
2. Retrouver le répertoire d'installation du programme.
3. Écrire un seul programme qui en fonction du nom d'invocation fait des choses différentes (ex: busybox)
   factorisation de code $\Longrightarrow$ gain de place

#### *1.4.3.2 Convention: Variables d'environnement*

Le format (convention) des variables d'environnement passées à un programme est:
nom-var=valeur-var Quelques variables d'environnement conventionnelles:

**HOME** Sa valeur est le chemin absolu du répertoire de travail de l'utilisateur UID.

**TERM** Sa valeur est le type de terminal (linux, xterm, vt100, ...).

**LANG** Sa valeur donne le langage de l'utilisateur et le charset utilisé.

**PATH** Sa valeur est une suite de chemins séparés par le caractère ':' (ex: PATH=/bin:/usr/bin:/usr/local/bin).
Si on lance un programme par un chemin sans '/' (ex: gnu) alors le système lancera le premier fichier gnu exécutable trouvé dans la suite de répertoires du PATH (ex: soit les chemins /bin/gnu, /usr/bin/gnu, /usr/local/bin/gnu).

#### 1.4.3.3 Convention: Flux d'entrée/sortie

Un programme qui démarre dispose de 3 flux d'entrée/sortie:

**Flux 0** flux standard d'entrée, ouvert en lecture, abréviation stdin (libc):
$\Longrightarrow$ dédié à l'acquisition de données d'entrée.

**Flux 1** flux standard de sortie, ouvert en écriture, abréviation stdout (libc):
$\Longrightarrow$ dédié à l'écriture de données de sortie.

**Flux 2** flux standard d'erreur, ouvert en écriture, abréviation stderr (libc):
$\Longrightarrow$ dédié à l'écriture de message d'erreur.

#### 1.4.3.4 Statut

La valeur renvoyée par un programme ($0 \leq valeur \leq 255$) indique si le programme s'est déroulé sans problème.
Elle peut être récupérée par l'entité qui a lancé le programme.

$$\text{statut} = 0 \Longrightarrow \text{ok},$$
$$\text{statut} \neq 0 \Longrightarrow \text{erreur}.$$

## 1.5 Système Unix

### 1.5.1 Shell

#### 1.5.1.1 Fonction

Le Shell est un programme dont les fonctions sont:

**Shell interactif** C'est l'interface utilisateur standard d'Unix.

- Dès qu'un utilisateur ouvre un terminal, il discute avec un Shell.
- Il permet de lancer des commandes simple ou avec des d'options.
- Il permet de chainer des commandes de façon très souple.
- Il permet de taper très rapidement grâce à ses mécanismes d'expansion, la complétion et le rappel de commandes.

**Shell script** C'est un langage de programmation complet (variables, alternatives, boucles) dédié à l'écriture de programme système.
Ces programmes lancent et/ou chainent d'autres programmes de manière automatique.

#### 1.5.1.2 Historique

**1977** sh (Bourne shell)

**1978** csh (C shell)

**1981** tcsh (C shell)

**1983** ksh (Kom shell)

**1988** bash (Bourne again shell)

**1990** ash (réécriture du Bourne shell)

**1990** zsh

A part csh et tcsh qui ont divergé, les autres sont compatibles avec le Bourne shell $\Longrightarrow$ scripts écrits il y a 40 ans fonctionnent encore.
bash et zsh sont très confortables et très semblables.

### 1.5.2 Lancer une commande

#### 1.5.2.1 Syntaxe

Une commande en avant plan (fg):

```
sh> chmod 644 gnu/bee <C-R>
sh>
```

**sh>** l'invite de commande (prompt) écrite par le Shell.

**chmod 644 gnu/bee** la commande tapée par l'utilisateur. C'est une suite de chaines de caractères séparées par des espaces (séquence d'au moins un blanc ou tabulation).

**chmod** la première chaine de caractères est le programme à exécuter (chemin relatif ou absolu ou basename avec le PATH).

**644** $1^{er}$ argument, sa signification dépend de la commande.

**gnu/bee** $2^{nd}$ argument, sa signification dépend de la commande.

<C-R> La touche entrée tapée par l'utilisateur. Elle indique au Shell que la commande est complète et qu'elle doit être exécutée.

**sh>** Quand la commande est terminée, le Shell affiche à nouveau le prompt et attend une nouvelle commande.

Une commande en arrière plan (bg):

**sh> sleep 60 &** <C-R>
**sh>**

**sh>** l'invite de commande (prompt) écrite par le Shell.

**chmod 644 gnu/bee** la commande tapée par l'utilisateur. C'est une suite de chaines de caractères séparées par des espaces (séquence d'au moins un blanc ou tabulation).

**sleep** la première chaine de caractères est le programme à exécuter (chemin relatif ou absolu ou basename avec le PATH).

**60** $1^{er}$ argument, sa signification dépend de la commande.

**&** en fin de commande indique l'arrière plan

<C-R> La touche entrée tapée par l'utilisateur. Elle indique au Shell que la commande est complète et qu'elle doit être exécutée.

**sh>** Sans attendre la fin de la commande, le Shell affiche à nouveau le prompt et attend une nouvelle commande.

#### *1.5.2.2 Ce qui se passe*

Voir la figure 1. La configuration standard du tty donne pour des données tapées:

**le leader:**

**CTL-C** $\Longrightarrow$ mort du processus

**CTL-Z** $\Longrightarrow$ suspension du processus

**read du tty** il obtient les données tapées

**CTL-D** il obtient une fin du fichier tty

**un non leader:**

**read du tty** $\Longrightarrow$ suspension du processus

#### *1.5.2.3 Raccourcis*

Sur l'invite de commande du Shell on peut:

- Avec ↑ et ↓ se déplacer dans l'historique des commandes déjà exécutées.
- Avec ← et → se déplacer dans la commande rappelée et la modifier.
- Enfin la touche tabulation déclenche la complétion:
  - Sur le premier mot de la commande la complétion est faite sur le PATH
  - Sur les autres mots de la commande la complétion est faite sur le système de fichiers.

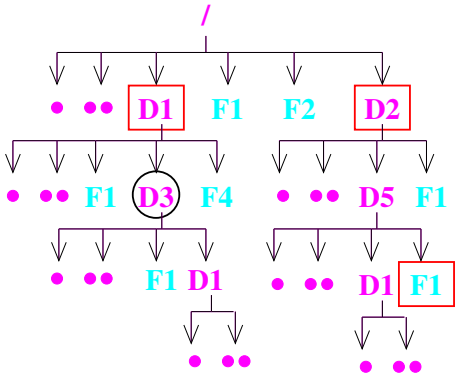
  Si il y a conflit pour compléter, une deuxième tabulation affiche les possibilités.

### 1.5.3 Les commandes de base

#### *1.5.3.1 Liste*

La figure 2 présente une liste des commandes les plus utilisées.

#### *1.5.3.2 Quelques exemples*




Soit le système de fichiers ci-dessus:

1. Faites que le CWD soit le répertoire D2 encadré.

   **sh> cd /D2**

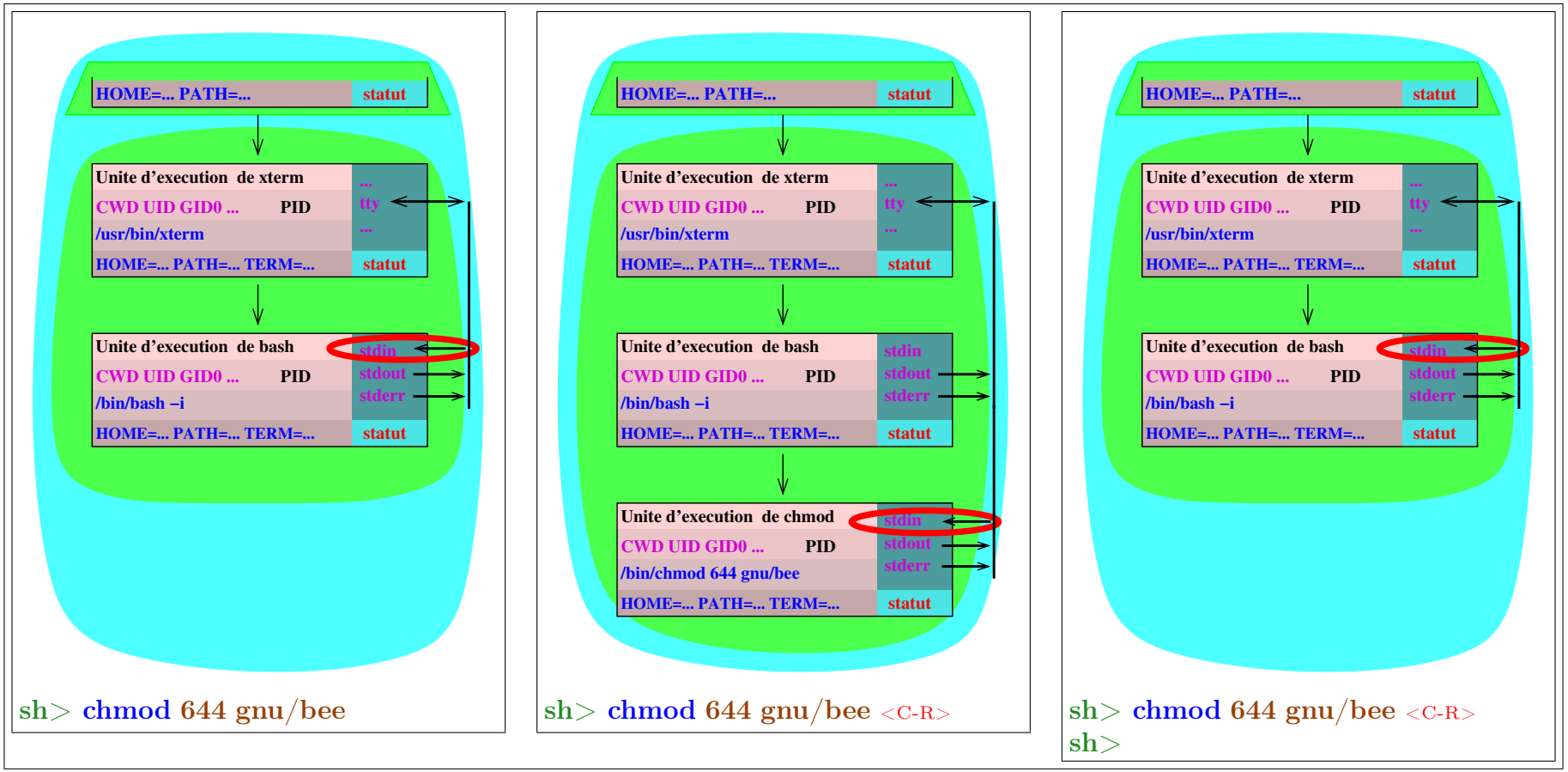



Figure 1: Lancement d'une commande premier plan.

Les commandes de bases en mode interactif:

| | | |
|---|---|---|
| cd | : | change le CWD. |
| mkdir | : | création de répertoires |
| rmdir | : | suppression de répertoires |
| ls | : | affiche les informations relatives aux fichiers et répertoires |
| cp | : | copie de fichiers et répertoires |
| mv | : | déplacement/renommage de fichiers et répertoires |
| rm | : | suppression de fichiers et répertoires |
| less | : | un pageur |
| chmod | : | modification des permission de fichiers et répertoires |
| wc | : | compte le nombre de lettres, mots et lignes d'un fichier |
| find | : | recherche de fichier dans une arborescence |
| grep | : | recherche un motif dans un fichier |

Les commandes de bases utiles pour les scripts:

| | | |
|---|---|---|
| echo | : | affiche un message sur le flux standard de sortie |
| read | : | lit un message sur le flux standard d'entrée |
| cat | : | concaténation de fichiers |
| test | : | compare des nombres, des chaines; obtention de propriétés de fichiers |
| dirname | : | obtention des répertoires des chemins (/aa/bb/cc/dd) |
| basename | : | obtention des noms de base des chemins (/aa/bb/cc/dd) |
| head/tail | : | extraction de lignes |
| sed/cut | : | extraction de lignes et parties de lignes |

Figure 2: les commandes essentielles

```
sh> pwd
/D2
sh> ls
D5 F1
sh>
ou
sh> cd ../../D2
sh>
```

2. Affichez le contenu des répertoires D1 et D2 encadrés.
   - mauvaise méthode

     ```
     sh> cd ..
     sh> ls
     F1 D3 F4
     sh> cd /D2
     sh> ls
     D5 F1
     sh> cd ../D1/D3
     sh>
     ```
   - bonne méthode

     ```
     sh> ls .. /D2
     ..: :
     F1 D3 F4
     /D2:
     D5 F1 :
     sh>
     ```

3. Copier le fichier F1 encadré dans D1/gnu.
   - mauvaise méthode

     ```
     sh> cd /D2
     ```

```
sh> ls
D5 F1
sh> cd /D5
sh> ls
D1 F1
sh> cp F1 /D1/D3/D1/gnu
sh> cd /D1/D3/
sh>
```

- bonne méthode

```
sh> cp /D2/D5/F1 D1/gnu
sh>
```

4. Détruisez l'arborescence donnée par le répertoire D2 encadré.

```
sh> rm /D2
sh>
```

5. Pourquoi la séquence ci-dessous ne détruit pas le fichier "-f"?

```
sh> ls .
gnu -f bee
sh> rm -f
sh> ls .
gnu -f bee
sh>
```

Car -f est une option de **rm**. Comment le détruire?

```
sh> rm ./-f
sh>
```

# 2 Shell interactif

## 2.1 Séquence d'instructions

### 2.1.1 Séquence simple

La fin d'une commande est indiquée par un <C-R>ou le caractère ';'

```
sh> gcc 1.c
sh> ./a.out 2 + 3
5
sh>
```

```
sh> gcc 1.c ; ./a.out 2 + 3
5
sh>
```

Si la compilation échoue, la commande **./a.out** est lancée

Le statut renvoyé par la séquence est celui de la dernière commande

### 2.1.2 Séquence conditionnelle

Les opérateurs **&&** (et) et **||** (ou) permettent de chainer 2 commandes en fonction du statut de la première.

```
sh> ./a.out -1 gnu || ./a.out -2 gnu || ./a.out -3 gnu
```

1. La seconde commande **./a.out** n'est lancée que si la première échoue.
2. La troisième commande **./a.out** n'est lancée que si les 2 premières échouent.
3. Le statut de la séquence est faux ($\neq 0$) si les 3 commandes échouent.

```
sh> gcc 1.c && ./a.out 2 + 3
```

1. La commande **./a.out** n'est lancée que si la compilation a réussi.
2. Le statut de la séquence est vrai ($= 0$) si les 2 commandes réussissent.

### 2.1.3 Groupe d'instructions

Une séquence d'instructions peut être parenthésée soit avec des accolades (lancée par le Shell courant), soit avec des parenthèses (lancée par un autre processus Shell).

```
sh> { true  && false ;  } || ( false && true  ) || { false && false ;  }
```

1. Le second facteur est lancé dans un autre Shell.
2. Les **;** avant les **}** sont obligatoires.

```
sh> ! { { true  && false ;  } || ( false && true  ) || { false && false ;  }
```

1. **!** est l'opérateur "non logique".
2. La négation de la commande précédente.
3. Fonctionne aussi avec des parenthèses.

```
sh> ( cd gnu/bee ;  cp f1 f2 )
```

1. Le CWD n'a pas changé.

## 2.2 Variables et environnement

### 2.2.1 Variables locales

Une variable est identifiée par un nom, l'expression régulière "[a-zA-Z_][a-zA-Z0-9_]*" définit les noms valides.

Affectation d'une variable

```
sh> CMD=cd
sh> DIR=bee
sh> SDIR=gnu
sh>
```

Pas d'espace autour du '='.

Valeur d'une variable

```
sh> $CMD $DIR/$SDIR
```

ou

```
sh> ${CMD}
${DIR}/${SDIR}
```

1. La valeur d'une variable non définie $\Longrightarrow$ chaine vide.

2. La forme parenthésée permet la concaténation de variables:
   **$DIR**gnu ⟹ **DIRgnu** non définie
   **${DIR}**gnu ⟹ beegnu

3. La forme parenthésée permet de nombreuses extensions:
   **${DIR:-undef}** ⟹ undef si **DIR** non définie.

#### 2.2.2 Variables d'environnement

Au démarrage le Shell crée une variable locale en la marquant pour chaque variable d'environnement définie.
On peut marquer une variable avec la commande Shell **export**:

```
sh> NVE=gnu          sh> export NVE       sh> export NVE=gnu
sh> export NVE       sh> NVE=gnu          sh>
sh>                  sh>
```

Toutes les variables marquées sont transmises aux processus créés.

La commande "**export -n NVE**" démarque la variable **NVE**.

Les variables marquées peuvent être visualisées par la commande Shell **export** sans argument ou la commande Unix **env**.

On ajoute une ou plusieurs variables d'environnement à une commande par:

```
sh> VE1=gnu VE2=bee cmd gnat
sh>
```

## 2.3 Expansions

### 2.3.1 Commande

#### *2.3.1.1 Principales expansions*

Le Shell a beaucoup de commandes implicites qu'il étend pour les exécuter. Les principales sont:

**$vname** La valeur de la variable **vname**.

**${vname...}** Une valeur déduite de la valeur de la variable **vname**.

**$$** Le PID du Shell.

**$?** Le statut de la dernière commande exécutée.

**!n** La $n^{ième}$ commande de l'historique.

**!str** La dernière commande entrée commancant par str.

**$[ expr ]**

**$(( expr ))** Le résultat de l'expression arithmétique expr.

**‘ cmd gnu bee ... ‘**

**$( cmd gnu bee ... )**
Le flux stdout de la commande **cmd gnu bee ...**.

#### *2.3.1.2 Quelques exemples*

```
sh> !gcc
gcc -I . -L /usr/local/lib bee.c
sh>
```

```
sh> i=10
sh> i=$[2 * $i + 1]
sh> echo $i $[$i + 1]
21 22
sh>
```

```
sh> cmd $(cat gnu/bee) $[2<<10]
sh>
```

```
sh> cmd $(ls)
sh>
```

```
sh> V=$(echo $[1+1] gnu bee)
sh> echo $V
2 gnu bee
sh>
```

```
sh> V=$(cat)
aaa bbb
CTL-D
sh> echo $V
aaa bbb
sh>
```

### 2.3.2 Fichiers

#### *2.3.2.1 Principe*

Soit une commande:

| str0 $[str1 $(str2 str3) ] str4 $str5 ${str6:-str7 str8} str9 |
|---|

elle est composée de chaines de caractères $str_i$. Chacune de ces chaines

sauf les noms de variables (ex: str5 et str6) et les expressions arithmétiques (ex: str1) va être "expansée".
Chacune des $str_i$ concernées est considérée comme une expression régulière sur le système de fichiers. (Les fichiers commençant par '.' sont omis par défaut).

1. Si un ou plusieurs fichiers sont compatibles avec cette expression régulière, alors $str_i$ est remplacée par ces fichiers séparés par un espace.

2. Si aucun fichier n'est compatible avec cette expression régulière, alors $str_i$ est conservée telle quelle.

Par exemple si $str_i$ est "*"

```
sh> ls
sh> echo *
*
sh>
```

```
sh> ls
bee gnu
sh> echo *
bee gnu
sh>
```

#### 2.3.2.2 *Format des expressions régulières*

**~str** en début de chaine (str sans /) donne le HOME de l'utilisateur str.

**~/** en début de chaine donne le HOME.

**~** en début de chaine et la chaine n'a qu'un caractère donne le HOME.

***** 0 ou plusieurs caractères sauf le '/'.

**?** 1 caractère sauf le '/'.

**[** $m_0 m_1 m_2 ...$**]** 1 caractère appartenant à au moins un motif $m_i$. $m_i$ est
soit un caractère (ex: x),
soit une séquence de 3 caractères (ex: E-K $\Longrightarrow$ {E F G H I J K}).

**{expr0,expr1,...}** correspond au choix à une des expressions.

**autre caractère** le caractère.

#### 2.3.2.3 *Quelques exemples*

```
sh> ls /[a-zA-Z]*/[a-zAZ]*/*[02-5]
ls: /[a-zA-Z]*/[a-zAZ]*/*[02-5]: No
such file or directory
sh>
```

```
sh> FILES=../D2/*.c
sh> gcc $FILES
sh> # ou
sh> gcc ../D2/*.c
sh>
```

```
sh> wc -l prj/*.[ch] prj/*/*{.c,.h}
2000
sh>
```

```
sh> ls
echo main.c
sh> *
main.c
sh>
```

### 2.3.3 Échappement

Vu le grand nombre d'expansions possibles, le Shell propose 3 mécanismes d'échappement (eg: qui bloquent l'expansionn).

**\c** Le caractère c devient un caractère standard.

```
sh> echo \$FILES
$FILES
sh>
```

```
sh> ls gnu\ bee
ls: gnu bee: No such file or directory
sh>
```

**"str"** Dans la chaine de caractères str:
les espaces sont échappés,
les expansions de fichiers sont échappées,
les expansions de commandes sont toujours actives.

```
sh> ls "gnu bee"
ls: gnu bee: No such file
or directory
sh>
```

```
sh> FILES="/*"
sh> echo "$FILES ${FILES} $[1+1]"
/* /* 2
sh>
```

**'str'** Dans la chaine de caractères str tout est échappé.

## 2.4 Redirections

### 2.4.1 Principe

Voir figure 3.

### 2.4.2 Syntaxe

**cmd ...** < **path** redirige le fichier path vers le flux stdin.
Le fichier path doit exister et être accessible en lecture.

**cmd ...** > **path** redirige le flux stdout vers le fichier path.
Si le fichier path existe, il est écrasé.
Si le fichier path n'existe pas, il est créé.
Il doit être accessible en écriture.

**cmd ... 2> path** similaire à "> path" mais pour le le flux stderr.

**cmd ...** ¦ **cmd ...** redirige le flux stdout de cmd1 sur le flux stdin de cmd2.

**cmd ...** >> **path** redirige le flux stdout vers le fichier path.
Si le fichier path existe, les écritures se feront à la fin du fichier.
Si le fichier path n'existe pas, il est créé.
Il doit être accessible en écriture.

**cmd ... 2>> path** similaire à ">> path" mais pour le le flux stderr.

**cmd ... 1>&2** redirige le flux stdout sur stderr.

**cmd ... 2>&1** redirige le flux stderr sur stdout.

```
cmd ... <<EOF
hello
word
EOF
```

initialise le flux stdin à hello word.

Les redirections peuvent être écrites n'importe où sur la ligne de commande.

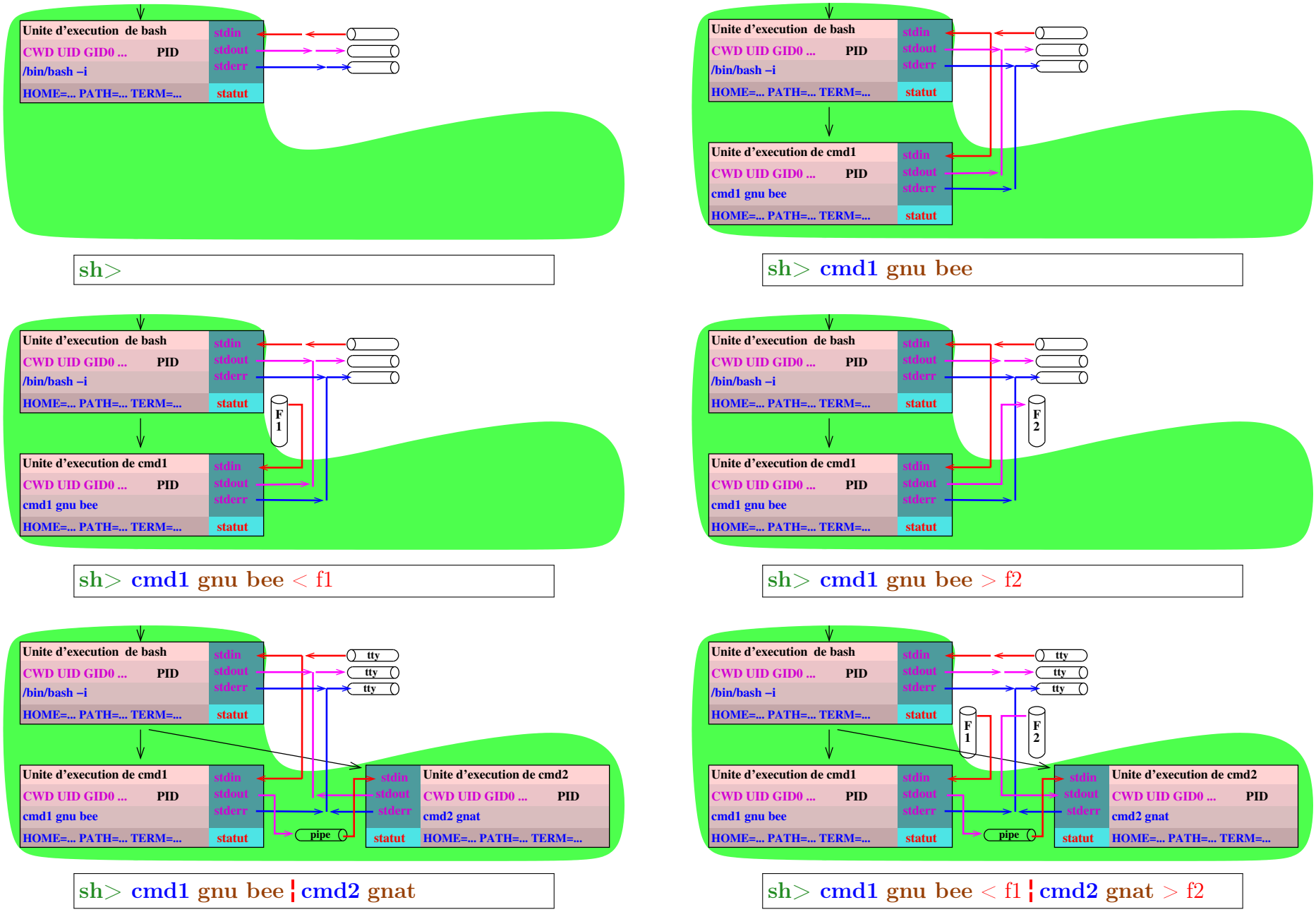



Figure 3: Principe des redirection

# 3 Shell script

## 3.1 Mon premier script

### 3.1.1 Exécution

Fichier hello.sh:

```
1 #!/bin/bash
2
3 echo -n "  " Hello
4 echo World
```

Pour lancer le script:

**Méthode 1** `sh> bash hello.sh`

**Méthode 2** rendre le script exécutable (chmod) puis `sh> ./hello.sh`

#! doivent être les 2 premiers caractères du script.

Le système lance la commande suivant "#!" en ajoutant l'argument "./hello.sh" à la fin.

**Méthode 3** `sh> bash < hello.sh`

Dans tous les cas /bin/bash lit le script et il exécute les commandes. Il s'arrête à la fin du script.

Les méthodes 1 et 2 sont équivalentes.
Dans la méthode 3 le flux stdin du Shell est le script lui même.

```
1 echo -n "  " Hello
2 cat                    ⟹
3 echo World
```

### 3.1.2 "Sourcer" un script

Dans un Shell qui tourne, on peut à tout moment "sourcer" un script par la commande "." ou "source"

`sh> source hello.sh`

ou

`sh> . hello.sh`

Dans ce cas, le Shell interrompt sa lecture du flux stdin, il lit le script à la place. A la fin du script, il se remet à lire le flux stdin.

Dans ce cas, le script peut modifier les données du Shell (ex: PATH).

Les scripts Shell de démarrage (.bashrc, .profile, ...) sont sourcés. Ceci permet de configurer son Shell.

## 3.2 Instructions de contrôle

### 3.2.1 Alternative

#### *3.2.1.1 Syntaxe*

```
1 if cmd-cond
2 then
3   ...
4 else
5   ...
6 fi
```

```
1 if cmd-cond ; then
2   ...
3 else
4   ...
5 fi
```

- La clause then est obligatoire.
- La clause else est facultative.
- Le ; est quasi obligatoire sinon then est compris comme un argument de cmd-cond.
- cmd-cond est soit une commande simple ou un groupe de commandes.
- Si la commande cmd-cond renvoie le statut 0 (ok) la clause then est exécutée.
- Si la commande cmd-cond renvoie un statut $\neq 0$ (pas ok) la clause else est exécutée.

#### *3.2.1.2 Exemple 1*

Tester si le fichier file contient main.

```
1 if grep -q main file ; then
2   echo main dans file
3 else
4   echo pas de main dans file
```

```
5 fi
```

Si file n'existe pas grep écrit un message d'erreur d'où:

```
1 if grep -q main file 2> /dev/null ; then
2   echo main dans file
3 else
4   echo pas de main dans file
5   echo ou file non accessible
6 fi
```

#### 3.2.1.3 Exemple 2

Tester si le répertoire dir possède les 2 fichiers .h et .c dont le nom de base est donné par la variable BASE.

```
1 if < dir/$BASE.h && < dir/$BASE.c ; then
2   echo trouvés
3 else
4   echo pas trouvés
5 fi
```

Fonctionne mais le Shell écrit des messages d'erreur si les fichiers n'existent pas.

```
1 if { < dir/$BASE.h && < dir/$BASE.c ; } 2> /dev/null ; then
2   echo trouvés
3 else
4   echo pas trouvés
5 fi
```

#### 3.2.1.4 Exemple 3

Écrire oui si on est en 2016?

```
1 if date | grep -q 2016 ; then
2   echo oui
3 fi
```

Enfin pour les petites demi alternatives, on peut aussi utiliser les opérateurs logiques && et ||.

```
1 date | grep -q 2016 && echo oui
```

### 3.2.2 Choix conditionnel

#### 3.2.2.1 Cas d'une constante

```
1 case mot in
2   motif )
3     ...
4     ;;
5   motif | motif )
6     ...
7     ;;
8   ...
9 esac
```

Format des motifs:

~**str** en début de chaine (str sans /) donne le HOME de l'utilisateur str.

~/ en début de chaine donne le HOME.

~ en début de chaine et la chaine n'a qu'un caractère donne le HOME.

* 0 ou plusieurs caractères.

? 1 caractère.

[ $m_0 m_1 m_2 ...$] 1 caractère défini par les $m_i$. $m_i$ est soit un caractère (ex: x), soit une séquence de caractères (ex: E-K $\Longrightarrow$ {E F G H I J K}).

- La première clause dont un motif décrit le mot est exécutée puis le contrôle reprend après le cas.
- Si aucun motif ne décrit le mot, aucune clause n'est exécutée et le contrôle reprend après le cas.

#### 3.2.2.2 Choix conditionnel d'expressions booléennes

```
1 if cmd-cond1 ; then
2   ...
3 elif cmd-cond2 ; then
4   ...
5 elif cmd-cond3 ; then
6   ...
7 else
8   ...
9 fi
```

elif permet d'emboiter des si sans multiplier les fin de si.

- Choisit la première clause dont la condition cmd-cond$_i$ est vraie.
- Si aucune cmd-cond$_i$ n'est vraie exécute la clause else.
- La clause else est facultative.

#### 3.2.2.3 Exemple

Tester si la variable n est un nombre de 1 à 3 chiffres.

```
case $n in
   [0-9])          good=1 ;;
   [0-9][0-9])     good=1 ;;
   [0-9][0-9][0-9])
                   good=1 ;;
   *)              good=0 ;;
esac
```

```
case $n in
   [0-9] | [0-9][0-9] |
   [0-9][0-9][0-9] )
           good=1 ;;
   * )     good=0 ;;
esac
```

Déterminer si le mot donné par la variable m commence, se termine ou contient la chaine de caractères ia.

```
debut=0; fin=0; mil=0;
case $m in
   ia* )          debut=1 ;;
   *ia )          fin=1 ;;
   *ia*)          mil=1 ;;
esac
```

### 3.2.3 Boucle while

#### 3.2.3.1 Syntaxe

```
while cmd-cond
do
      ...
      ...
done
```

```
while cmd-cond ; do
      ...
      ...
done
```

- Le ; est quasi obligatoire sinon “do” est compris comme un argument de cmd-cond.
- Si la commande cmd-cond renvoie le statut 0 (ok), le corps de la boucle est exécuté puis le contrôle reprend en début de boucle.
- Si la commande cmd-cond renvoie un statut $\neq 0$ (pas ok), le contrôle reprend après la boucle.
- Dans le corps de boucle les commandes suivantes sont disponibles:
  **continue** branche en début de boucle,
  **break** donne le contrôle après la boucle.

#### 3.2.3.2 Exemple

La commande ping −c 1 host permet de tester si la machine host est accessible via le réseau.

Écrire un script qui toutes les 30 secondes émet 5 beeps si la machine 192.168.1.10 n’est pas accessible.

```
while true ; do
      sleep 30
      ping -c 1 192.168.1.10 > /dev/null 2>&1 || \
        echo -e -n "\b\b\b\b\b"
done
```

### 3.2.4 Boucle For

#### 3.2.4.1 Syntaxe

```
for v in str-list
do
      ...
      ...
done
```

```
for v in str-list ; do
      ...
      ...
done
```

- Le ; est quasi obligatoire sinon “do” est compris comme un élément de str-list.
- str-list est un suite de chaines de caractères str$-i$ séparées par un ou plusieurs espaces.
- La variable v prend dans l’ordre de str-list toutes les valeurs str$_i$ et le corps de boucle est exécuté pour chaque valeur.
- Dans le corps de boucle les commandes suivantes sont disponibles:
  **continue** branche en début de boucle,

**break** donne le contrôle après la boucle.

- La chaine de caractères {n..m} est expansée en tous les nombres de n à m compris.
- La chaine de caractères {n..m..i} est expansée en tous les nombres de n compris à m avec un pas de i.

#### *3.2.4.2 Exemple*

Écrire Hello World lettre par lettre.

```
1 for l in H e l l o "␣" W o r l d ; do
2   echo -n $l
3 done
4 echo
```

Écrire 10 lignes contenant chacune 5 fois silence.

```
1 for i in {1..10} ; do
2   for j in {19..15} ; do
3     echo -n silence "␣"
4   done
5   echo
6 done
```

Ecrire sur une ligne les noms de base des fichiers du répertoire donné par la variable dir, suffixés entre parenthèses de leur nombre de lignes.

```
1 for bn in $(cd $dir ; ls) ; do
2   echo -n "$bn(" $(wc -l $dir/$bn 2> /dev/null | \
3     sed -e 's/\([0-9]*\).*/\1/' ) ")␣"
4 done
5 echo
```

## 3.3 Quelques indispensables

### 3.3.1 Commande test

Pour pouvoir écrire des programmes, il faut pouvoir faire des tests:

- le fichier str existe-t-il?
- le fichier str est il un répertoire?
- les chaines $str_1$ et $str_2$ sont-elles égales?
- $n_1$ est-il inférieur à $n_2$ ?
- ...

la commande **test** répond à ce besoin.

Les arguments de la commande définissent une expression booléenne, elle renvoie un statut de 0 si l'expression est vraie et de 1 sinon.

sh> **test** **$arg_0$** **$arg_1$** **$arg_2$** **...**

ou

sh> **[** **$arg_0$** **$arg_1$** **$arg_2$** **...** **]**

#### *3.3.1.1 Arguments*

**str**

**-n str** str est non vide

**-z str** str est vide

**$str_1$ = $str_2$** $str_1$ et $str2_2$ sont égales (!= pour différentes).

**$str_1$** \< **$str_2$** $str_1$ est inférieure (ordre lexicographique) à $str2_2$ (\> pour supérieure) (bash builtin).

**$n_1$ OP $n_2$** l'entier $n_1$ est OP à l'entier $n_2$ avec OP dans { -eq, -ne, -lt, -le, -gt, or -ge } (égal, diffèrent, inférieur, inférieur ou égal, supérieur, supérieur ou égal).

**-e file** le fichier file existe et n'est pas un lien mort.

**-f file** le fichier file** existe et est régulier.

**-d file** le fichier file** existe et est un répertoire.

**-h file** le fichier file existe et est lien symbolique.

**-r file** le fichier file** existe et peut être lu (file peut être un répertoire).

**-w file** le fichier file** existe et peut être écrit (file peut être un répertoire).

**-x file** le fichier file** existe et peut être exécuté (file peut être un répertoire).

**! expr** opérateur booléen non.

**expr -o expr** opérateur booléen ou.

**expr -a expr** opérateur booléen et.

**\( et \)** permet de parenthèser une expression booléenne.

Note **: ou dans le cas d'un lien, le fichier pointé final.

#### 3.3.1.2 Exemples

Déterminez si les chaines de caractères définies par les variables S1 et S2 sont égales:

```
1 if test $S1 = $S2 ; then
2   echo S1 = S2
3 fi
```

```
1 if [■$S1 = $S2■] ; then
2   echo S1 = S2
3 fi
```

Sauf si S1 est la chaine vide ou S1 n'est pas définie, l'expansion sera:

```
1 if test = $S2 ; then
2   echo S1 = S2
3 fi
```

```
1 if [■ = $S2■] ; then
2   echo S1 = S2
3 fi
```

Dans ce cas, la commande test écrit un message d'erreur "mauvais formatage de l'expression" et renvoie un statut différent de ∅. D'où la correction:

```
1 if test "$S1" = "$S2" ; then
2   echo S1 = S2
3 fi
```

```
1 if [■"$S1" = "$S2"■] ; then
2   echo S1 = S2
3 fi
```

Déterminez si l'entier défini par la variable n est compris entre 10 et 23 compris:

```
1 if test 10 -le $n -a $n -le 23 ; then
2   echo oui
3 fi
```

ou

```
1 if [■10 -le $n -a $n -le 23■] ; then
2   echo oui
3 fi
```

Écrire $n lignes contenant le mot silence:

```
1 i=0
2 while test $i -lt $n ; do
3   echo silence
4   i=$[i+1]  # ou $[i++]
5 done
```

Écrire ok si le fichier $f est régulier et accessible en lecture, écriture et non accessible en exécution:

```
1 if test -f $f -a -r $f -a -w $f -a ! -x $f ; then
2   echo ok
3 fi
```

ou

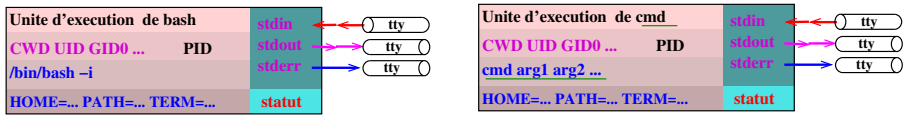
```
1   if test -f $f && [ -r $f -a -w $f -a ! -x $f ] ; then
2     echo ok
3   fi
```

### 3.3.2 Autres commandes

Ces commandes sont des builtin commandes (commandes internes, elles n'ont pas d'exécutables associés).

**exec cmd $arg_1$ $arg_2$ ...** Remplace l'unité d'exécution du processus Shell, par celle de l'exécutable **cmd** et les arguments par $arg_1$ $arg_2$

....




avant l'exec — après l'exec

$\Longrightarrow$ pas de création de processus.

$\Longrightarrow$ l'instruction qui suit l'exec ....

**exit n** Termine le Shell avec le statut n. Si n est omis, le statut est $\emptyset$.

**read $v_1$ $v_2$ $v_3$** Lit une ligne dans le flux stdin et affecte le contenu la ligne dans les variables $v_i$. La ligne est considérée comme n chaines de caractères séparées par un ou plusieurs espaces. Soit $s_1$ $s_2$ $s_3$ le contenu d'une ligne

| inst. \ var. | $v_1$ | $v_2$ | $v_3$ | $v_4$ |
|---|---|---|---|---|
| read | | | | |
| read $v_1$ | $s_1$ $s_2$ $s_3$ | | | |
| read $v_1$ $v_2$ | $s_1$ | $s_2$ $s_3$ | | |
| read $v_1$ $v_2$ $v_3$ | $s_1$ | $s_2$ | $s_3$ | |
| read $v_1$ $v_2$ $v_3$ $v_4$ | $s_1$ | $s_2$ | $s_3$ | "" |

### 3.3.3 Lire la ligne de commande

#### 3.3.3.1 Expansions dédiées

Quand on lance un script Shell, comment récupère-t-on ses arguments

```
sh> ./script petit lapin
sh>
```

```
1 #!/bin/bash
2
3 taille=... # arg1 (petit)
4 animal=... # arg2 (lapin)
```

Dans un script, on a les expansions suivantes:

**$#** le nombre d'arguments.

**$0** le nom de la commande.

**$1** le premier argument.

**$2** le second argument.

**$i** le $i^{ième}$ argument.

**$*** la liste des arguments séparés par un espace.

**"$@"** la liste des arguments quotés séparés par un espace.

#### 3.3.3.2 Exemples

Écrire le script "**punition n word**" qui écrit sur le flux stdout n lignes contenant le mot word:

```
1  #!/bin/bash
2  #
3  # usage: punition n word
4
5  i=0
6  while test $i -lt $1 ; do
7    echo $2
8    i=$[i+1]
9  done
```

Écrire le script "**somme $n_1$ $n_2$ ...**" qui écrit sur le flux stdout la ligne "$n_1 + n_2 \ldots + {}_n$".

```
1  #!/bin/bash
2  #
3  # usage: somme n1 n2 n3 ...
4
5  first=1
6  for n in $* ; do
7    test $first = 0 && echo -n +
8    echo -n $n
9    first=0;
10 done
```

#### 3.3.3.3 Commandes dédiées

**shift** Équivalent à "shift 1".

**shift n** Décale les arguments de n positions vers la gauche.

La table ci-dessous donne les valeurs des $1, $2, $3, $4, $5 en supposant que leurs valeurs initiales sont $v_i$ et que les valeurs des variables 6,7, ... sont la chaine de caractères vide.

| var. / inst. | $1 | $2 | $3 | $4 | $5 |
|---|---|---|---|---|---|
| echo $* | $v_1$ | $v_2$ | $v_3$ | $v_4$ | $v_5$ |
| shift 1 | $v_2$ | $v_3$ | $v_4$ | $v_5$ | |
| shift 2 | $v_3$ | $v_4$ | $v_5$ | | |
| shift 3 | $v_4$ | $v_5$ | | | |
| shift 4 | $v_5$ | | | | |
| shift 5 | | | | | |
| shift 6 | | | | | |
| ... | | | | | |

**set $str_1$ $str_2$ ... $str_n$**

Affecte $str_i$ à $\$_i$ pour i inférieur ou égal à n.

Affecte la chaine vide à $\$_i$ pour i supérieur n.

#### 3.3.3.4 Exemples

Écrire le script "**punition n word**" qui écrit sur le flux stdout n lignes contenant le mot word. Si n est omis, sa valeur par defaut est 5.

```
1  #!/bin/bash
2  #
3  # usage: punition [n] word
4
5  test $# = 1 && set 5 "$1"
6
7  i=0
8  while test $i -lt $1 ; do
9    echo $2
10   i=$[i+1]
11 done
```

Écrire le script "**punition n word$_1$ word$_2$ ...**" qui écrit sur le flux stdout n lignes contenant les mots word$_i$:

```
1  #!/bin/bash
2  #
3  # usage: punition n word1 word2 ...
4
5  n=$1
6  shift
7  i=0
8  while test $i -lt $n ; do
9    echo $*
10   i=$[i+1]
11 done
```

## 3.4 Création de Builtin commande

```
1  #!/bin/bash
2  ...
3  # usage: plus n1 n2 n3 ...
4  function plus( )
5  {
6    s=0
7    while test $# -gt 0 ; do
8      s=$[$s+$1]
9      shift
10   done
11   echo $s
12 }
13 ...
14 plus 10 12 100
15 ...
16 s=$(plus 1 2 3)
17 ...
```

Définit la commande plus.

Les tokens "function" et "()" sont facultatif mais il en faut au moins un

des 2.

Une fois définie la commande plus s'utilise comme n'importe quelle commande.

Si une commande plus existait accessible par le PATH, elle est masquée.

Si une builtin commande plus existait déjà, elle est écrasée.

**variable**

- non déclarée locale $\Longrightarrow$ globale
- déclarée locale $\Longrightarrow$ locale à partir de la déclaration

"local v" ou "local v=6" pour déclarer une variable v locale.

**statut de retour** le statut de la dernière commande exécutée.

**return** identique à "return 0"

**return n** quitte la commande avec un statut de n.

La commande exit peut être appelée dans une builtin commande, elle termine le script.

La commande return ne peut pas être appelée en dehors d'une builtin commande.

## 3.5 Exemple complet

### 3.5.1 Introduction

Pour illustrer ce chapitre, nous allons prendre l'exemple:

Réalisez le script "ecp f1 f2" qui copie le fichier régulier f1 dans f2. Le chemin de f2 est éventuellement créé.

### 3.5.2 Entête

```
1 #!/bin/bash
2 #
3 # usage: ecp f1 f2
4 # fonction: copie f1 dans f2
5 #    en creant le chemin f2
```

- ligne 1 automatique
- écrivez l'usage et la fonction (fixe les idées et aide la reprise)

### 3.5.3 Ligne de commande

```
6  # analyse des args
7  if test $# != 2 ; then
8    echo "$0: ..." 1>&2
9    exit 1
10 fi
11 src="$1"
12 des="$2"
```

- vérifiez le nombre d'arguments
- message d'erreur clair + sur stderr + exit avec statut d'erreur
- donnez des noms significatifs aux arguments

### 3.5.4 Test des arguments

```
13 # test de src
14 if ! test -f "$src"
15         -a -r "$src" then
16   echo "$0: ..." 1>&2
17   exit 1
18 fi
```

- vérifie si src est lisible et n'est pas un répertoire.
- message d'erreur clair + sur stderr + exit avec statut d'erreur

```
19 # test de dest
20 if test -d "$des" ; then
21   des="$des/$(basename "$src")"
22 fi
23 if test -h "$des" &&
24      ! rm "$des" 2> /dev/null then
25   echo "$0: ..." 1>&2 ; exit 1
26 fi
27 if test -d "$des" ; then
28   echo "$0: ..." 1>&2 ; exit 1
29 fi
```

- calcule le vrai nom du fichier destination
- vérifie que ce n'est pas un répertoire
- le supprime si c'est un lien

### 3.5.5 Corps

```
30    # des est regulier ou n'existe pas
31    if ! mkdir -p "$(dirname "$des")" \
32          2> /dev/null ; then
33      echo "$0: ..." 1>&2 ; exit 1
34    fi
35    if ! cp "$src" "$des" \
36         2>/dev/null then
37      echo "$0: ..." 1>&2 ; exit 1
38    fi
39    exit 0
```

- création des répertoires du fichier destination si besoin (mkdir -p)
- laisse cp faire le job.
- le exit 0 final n'est pas nécessaire.

### 3.5.6 Debug

Pour déboguer, le bash propose l'option -x qui affiche toutes les commandes expansées.

Pour l'activer, il faut soit lancer le script cmd par
bash -x **cmd** $\mathbf{arg_1}$ $\mathbf{arg_2}$ **...**

ou ajouter -x à la premiere ligne du script:
#!bash -x

ou insérer un "set -x" dans le script, le mode debug sera actif après cette commande.

Le mode debug peut être désactivé par l'exécution de la commande "set +x".

Pour déboguer, le bash propose aussi l'option -v qui affiche toutes les commandes mais sans les expanser. Elle s'active de la même manière et les 2 options sont cumulables (set -xv).

## 3.6 Astuces et pièges

### 3.6.1 Césure de lignes

Soit la ligne: **if test** −f f && rm f 2>/dev/null ; **then**

```
1 | if test -f f && rm f
2 |   2>/dev/null ; then
```

```
1 | if test -f f && rm f \
2 |   2>/dev/null ; then
```

```
1 | if test -f f && rm
2 |   f 2>/dev/null ; then
```

```
1 | if test -f f && rm \
2 |   f 2>/dev/null ; then
```

```
1 | if test -f f &&
2 |   rm f 2>/dev/null ; then
```

OK

⟹ En cas de doute, un \ ne peut pas faire de mal.

### 3.6.2 Espace dans les Expansions

```
1 | x=8
2 | if test $x ; then
```

VRAI: x est non vide

```
1 | x="a b"
2 | if test $x ; then
```

FAUX: x est vide + message d'erreur de test

```
1 | x="1 = 0"
2 | if test $x ; then
```

FAUX: x est vide

⟹ double quoter $x ("$x")

```
1 | p="bee/gnat"
2 | if test -d $(dirname $p)
```

OK: test -d bee

```
1 | p="bee/gnat gnu"
2 | if test -d $(dirname $p)
```

ERREUR: dirname bee/gnat gnu ⟹ message d'erreur

```
1 | p="bee/gnat gnu"
2 | if test -d $(dirname "$p")
```

OK: test -d bee

```
1 | p="bee old/gnat gnu"
2 | if test -d $(dirname "$p")
```

ERREUR: test -d bee old ⟹ message d'erreur

```
p="bee old/gnat gnu"
if test -d "$(dirname "$p")"
```
OK: test -d "bee old"

$\Longrightarrow$ double quoter $(cmd) ("$(cmd)")

```
set "a" "b" "c"
for s in $* ; do
```
OK: 3 fois la boucle

```
set "a" "b" "c d"
for s in $* ; do
```
ERREUR: 4 fois la boucle

```
set "a" "b" "c d"
for s in "$*" ; do
```
ERREUR: 1 fois la boucle

```
set "a" "b" "c d"
for s in "$@" ; do
```
OK: 3 fois la boucle

$\Longrightarrow$ ne pas utiler $* mais ("$@")

On double quote sauf ... (cas très rares)
On utilise pas $* mais "$@" souf ... (cas très rares)

### 3.6.3 Variable dans un sous processus

```
( cd .. ; test "$x" || \
    x=$(echo *) )
```
ARGH!: x est inchangée

```
test "$x" || \
    x=$(cd .. ; echo *)
```
OK: si x était vide, x est initialisée aux fichiers de ..

```
cd .. ; test "$x" || \
    x=$(echo *) ; cd -
```
OK: mais plus lourd

```
read x y < f
```
OK: lit la $1^{ière}$ ligne de f dans x et y

```
cat f | read x y
```
ARGH! read est fait dans un sous processus

Les principaux éléments de syntaxe qui créent des sous processus sont:
( ), $( ) et ¦

### 3.6.4 Utilisation de read

```
read a b
read x y
```
OK: lit 2 lignes du flux stdin

```
while read a b ; do
   ...
done
```
OK: lit tout le flux stdin ligne par ligne

```
read a b < f1
read x y < f1
```
ARGH!: lit 2 fois la même ligne (sauf si f1 est un flux tty, audio)

```
while read a b < f1 ; do
   ...
done
```
ARGH!: lit à l'infini la $1^{ière}$ ligne du flux stdin.

```
while read a b ; do
   ...
done < f1
```
OK: lit tout le fichier f1 ligne par ligne

```
1 x=0 ; a=0
2 sed /^#/d f1 | \
3 while read a b ; do
4   x=$a
5   ...
6 done
```

OK: lit tout le fichier f1 ligne par ligne en sautant les lignes commençant par #. Que valent ces variables après le done?

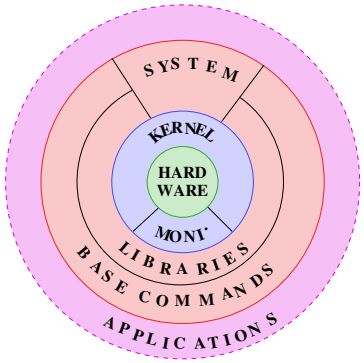
**x**

**a**

# 4 Appel système

## 4.1 Organisation

### 4.1.1 Couches Principales




**matériel** CPU, RAM, contrôleurs et périphériques.

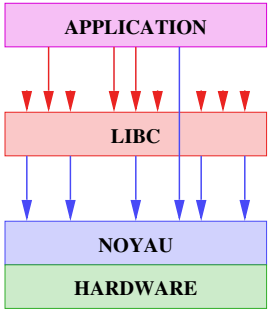
**moniteur** Petit programme en ROM, qui tourne au démarrage de la machine.

**noyau** Gère et donne accès au matériel

**système** Couche de standardisation

**applications**

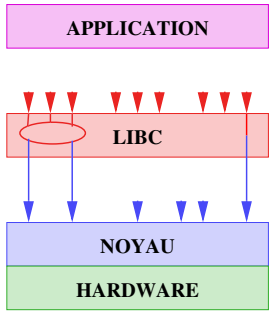
### 4.1.2 Appel système et libc




**Application**
Utilisation de la libc $\Longrightarrow$ portabilité,
Utilisation directe des appels système $\Longrightarrow$ difficile

**libc** (↓) Ensemble de services complets normalisés $\Longrightarrow$ portabilité

**Appels système** (↓) Services du noyau, peu nombreux $\Longrightarrow$ fonctions basiques



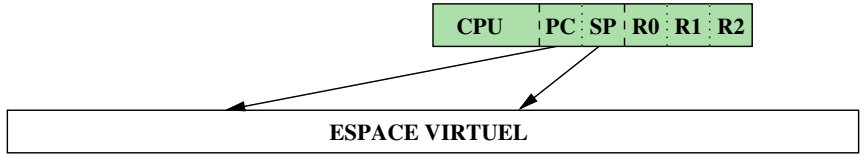

**libc: module standalone** service n'utilisant pas les appels système (ex: module strxxx)

**libc: module interface**
malloc, free, ... $\Longrightarrow$ brk utilisable
fopen, fread, ... $\Longrightarrow$ performance acceptable

**Libc: driver quasi-direct sur appels système**
$\Longrightarrow$ portabilité pour la plupart (ex: open, read)
$\Longrightarrow$ souvent pas très pratique (ex: time)
$\Longrightarrow$ risque d'utilisation non performante

### 4.1.3 Espaces mémoire

#### *4.1.3.1 Espaces virtuels*



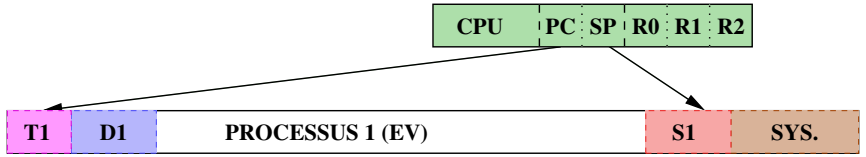

**CPU** Quelques registres

**PC** **P**rogram **C**ounter, adresse de l'instruction à exécuter.

**SP** **S**tack **P**ointeur, adresse du haut de la pile d'exécution.

**Espace virtuel** La mémoire que voit le processeur.

#### *4.1.3.2 Espace virtuel d'un processus*




**T1: segment text** il contient les instructions. Le PC se balade dans ce segment.

**D1: segment données** il contient les données globales du processus.

**S1: segment pile** il contient la pile d'exécution: données locales aux fonctions, les paramètres d'appel des fonctions et les adresses de retour dans l'appelant.

**trous** un accès à une adresse dans un trou $\Longrightarrow$ exception "segmentation fault"

**espace user/système**

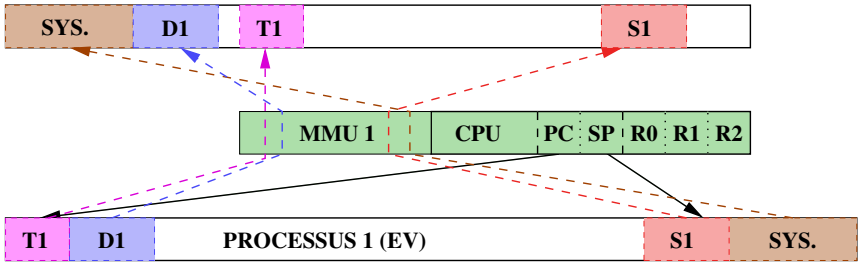
- Si le processeur est en mode système: il peut accéder à tout l'espace virtuel (sauf les trous).
- Si le processeur est en mode user: il ne peut pas accéder à l'espace système ($\Longrightarrow$ "privilege violation").

#### *4.1.3.3 Appel système*

C'est le mécanisme qui permet à un processus en mode utilisateur:

1. passer en mode système
2. exécuter une fonction du noyau (en mode système)
3. revenir en mode utilisateur après l'exécution de la fonction.

#### *4.1.3.4 MMU: Memory Management Unit*




**MMU** Unité matérielle contenant une batterie de registre

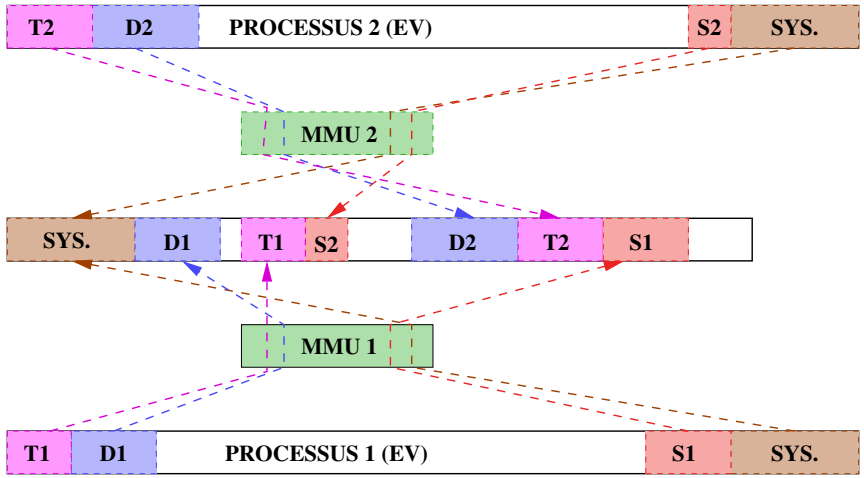
**Fonction** En fonction des valeurs contenues dans les registres:

- Convertit les adresses virtuelles en adresses physiques.
- Assure les protections mémoire (trou, ...).

**MMU i** La MMU configurée pour le processus i.

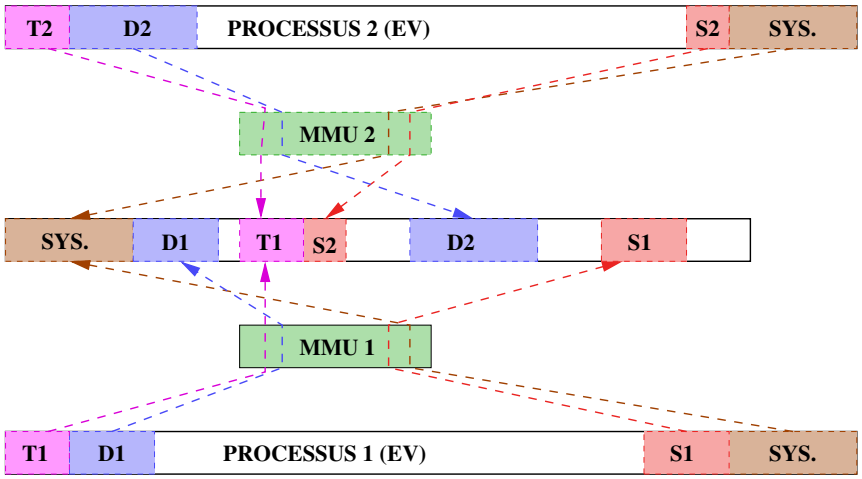
#### *4.1.3.5 Quelques configurations*

***Partage de l'espace système***



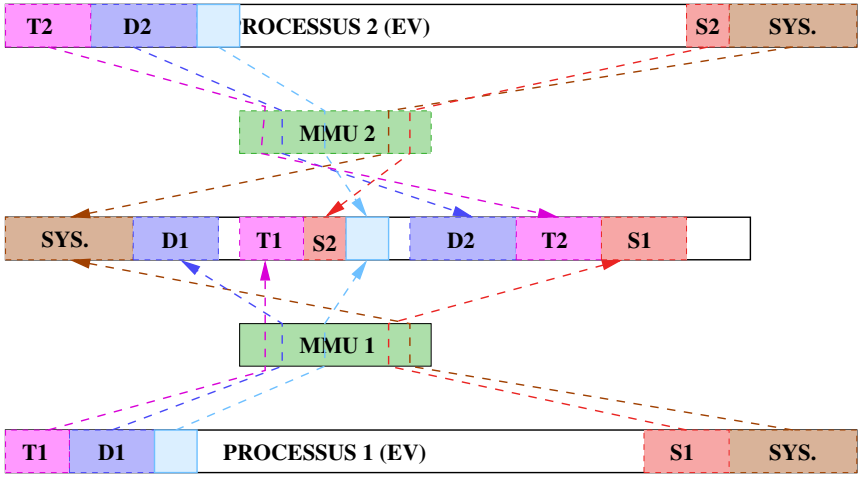

Le segment système est partagé entre les deux processus.

### *2 processus du même exécutable*



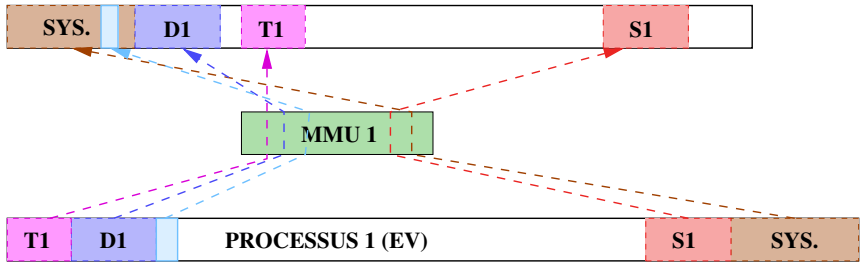

Les segments système et text sont partagés entre les deux processus.

### *Mémoire partagée*



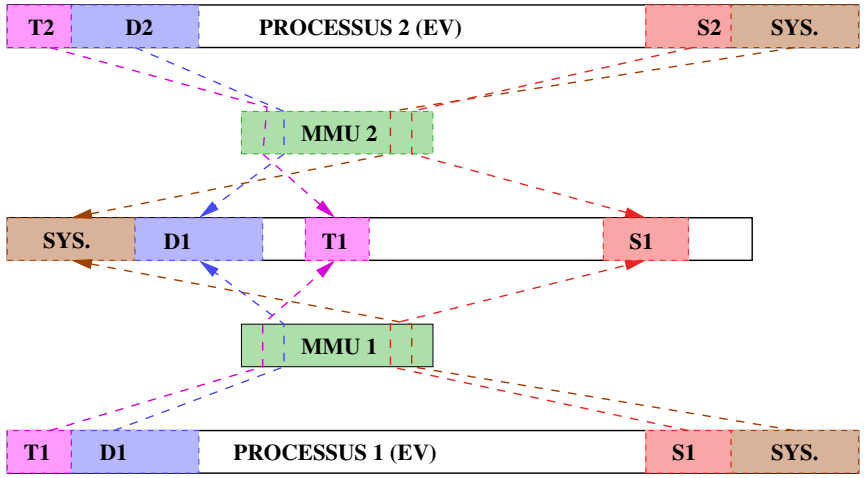

Le segment système et un bout de segment données sont partagés entre les deux processus. Les 2 processus peuvent s'échanger des données au travers de ce segment.

### *Accès direct à un tampon système*



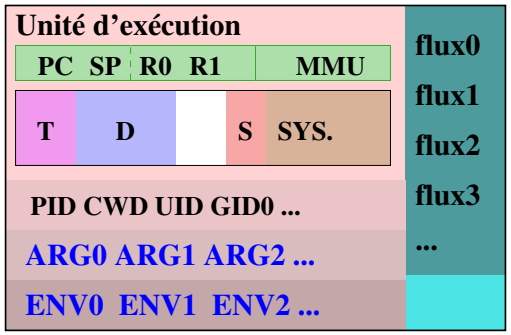

Un tampon de donnée système est mappé dans l'espace utilisateur (segment donnée). Le processus en mode utilisateur peut accéder à cette partie de l'espace système.

### *Processus légers*




MMU 1 = MMU 2
En quoi ces 2 processus diffèrent ils?

### 4.1.4 Processus




**Unité d'exécution**
Registres et MMU
Espace virtuel

**Informations diverse**

**Paramètres**

**Flux**

Où sont stockés ces éléments

## 4.2 Format général d'un appel système

### 4.2.1 Prototype

```
1  extern int errno;
2
3  int unAppelSysteme(
4     Tp1 p1,
5     Tp2 p2,
6     ...
7  );
```

**type retour** toujours int
-1 : erreur
$\geq \emptyset$ : ok
$< \emptyset$ : impossible

**arguments** de 0 à 6

**errno** $\geq \emptyset$, code d'erreur en cas d'echec seulement

### 4.2.2 Utilisation standard

```
1  #include <stdio.h>
2  #include <string.h> // pour strerror
3  #include <errno.h>  // pour errno
4
5  int main(int argc, char*argv[])
6  {
7     ...
8     if ( unAppelSysteme(...)==-1 ) {
9        fprintf(stderr,
10          "%s:Fatal: unAppelSysteme fails : %s\n",
11          argv[0], strerror(errno));
12       return 1;   // ou exit(1)
13    }
14    ...
15    return 0;
16 }
```

**ligne 8 (==-1)** Teste si erreur.

**ligne 9-12** Message standard explicite d'erreur sur stderr et fin avec status$\neq \emptyset$

**ligne 11 (errno)** Contient le code d'erreur.

**ligne 11 (strerror)** Convertit le code d'erreur errno en char* (message explicite).

# 5 Flux

## 5.1 Algorithmes

### 5.1.1 Lecture/écriture

Soit f un fichier au sens large: fichier régulier, fichier spécial, terminal réseau, ...

Lecture du fichier f

```
1  fd = ouvrir f en lecture
2  statut = lire(fd, n1, tamp)
   si statut = ERR ou EOF alors
      aller à fin
   fin si
3  statut = lire(fd, n2, tamp)
   si statut = ERR ou EOF alors
      aller à fin
   fin si
... ...
fin fermer fd
```

Écriture du fichier f

```
1  fd = ouvrir f en écriture
2  statut = écrire(fd, n1, tamp)
   si statut = ERR alors
      aller à fin
   fin si
3  statut = écrire(fd, n2, tamp)
   si statut = ERR alors
      aller à fin
   fin si
... ...
fin fermer fd
```

**fd** les opérations sur fichier nécessitent un descripteur de flux

**ouvrir** Crée un descripteur de flux, les modes sont RO, WO ou RW.

**fermer** Libère toutes les allocations associées au flux.
$\Longrightarrow$ Le fichier associé aux flux (si il existe) n'est pas détruit.
$\Longrightarrow$ Le système ferme tous les flux ouverts d'un processus lors de sa terminaison.

**lire** transfère n octets du flux vers un tampon mémoire

**écrire** transfère n octets d'un tampon mémoire vers le flux

**statut E.O.F** seul lire peut le recevoir.

**statut ERR** ouvrir, lire, écrire peuvent le recevoir, fermer* non.

**séquentiel** fd contient en autre un curseur de lecture dans le flux. lire n octets avance le curseur de n octets dans le flux (idem pour écrire).
$\Longrightarrow$ le lire suivant lit les octets suivants.

**bloquant**

**ouvrir** Non sur un fichier régulier local, possible sur d'autres flux.

**fermer** Non.

**lire** Potentiellement oui.

**écrire** Potentiellement non.

### 5.1.2 Positionnement

Positionnement dans un fichier f.

```
1  fd = ouvrir f en lecture
2  statut = lire(fd, n1, tamp)
   si statut = ERR ou EOF alors
      aller à fin
   fin si
4  statut = déplace(fd, m, POS)
   si statut = ERR alors
      aller à fin
   fin si
5  statut = lire(fd, n2, tamp)
   si statut = ERR ou EOF alors
      aller à fin
   fin si
... ...
fin fermer fd
```

**ouvrir** Crée un descripteur de flux, les modes sont RO, WO ou RW.

**déplace** Déplace le curseur du flux de m octets par rapport à une position fixe (POS).
$\Longrightarrow$ POS = debut ou fin ou curseur.

$\Longrightarrow$ n'est possible que sur des flux le supportant (ex: fichier régulier).

### 5.1.3 Quizz

1. Une lecture sur un flux avec un statut OK garantit que les données reçues sont bonnes.

2. Une écriture sur un flux avec un statut OK garantit que les données sont arrivées à destination.

3. Une écriture sur un flux associé à un fichier régulier ne peut pas renvoyer un statut ERR.

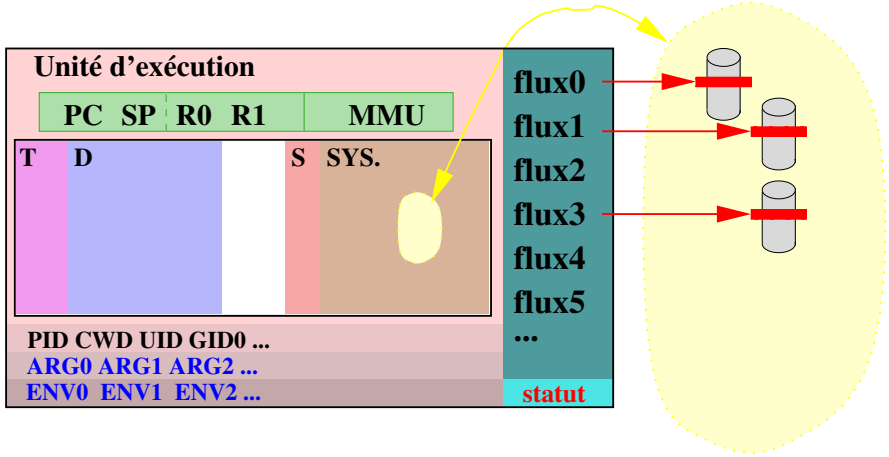
4. Après une lecture sur un flux avec un statut EOF, une seconde lecture donne toujours EOF.

5. Sur un flux sans erreur possible, on obtiendra toujours un statut EOF après un certain nombre de lectures.

6. Il n'est pas utile de fermer les flux que l'on utilise plus puisque le noyau le fera à la terminaison du processus.

## 5.2 Les flux noyau

### 5.2.1 Descripteurs de flux

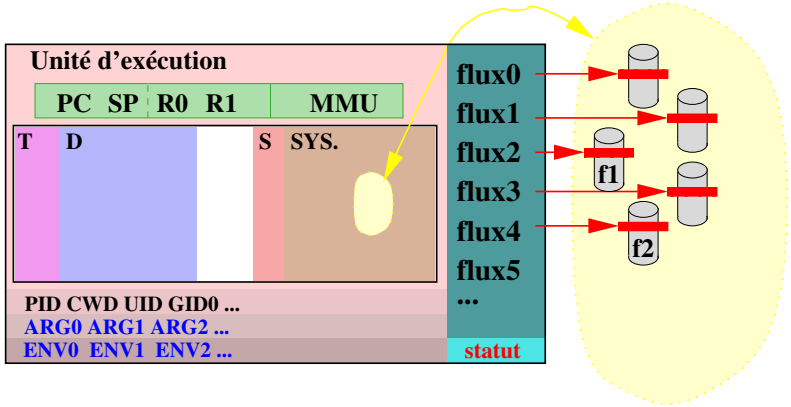



**type** Entier

**correspondance** Le $i^{ième}$ flux du processus

**exemple** flux 0, 1, 3 définis, 2, 4, ... non définis

### 5.2.2 Ouverture




**Synopsis sans création** int open(const char*f, int flags);

**Synopsis avec création** int open(const char*f, int flags, mode_t mode);

**Fonction** Associe un descripteur de flux au fichier f et le renvoie.

**Retour** Le descripteur de flux ou -1.

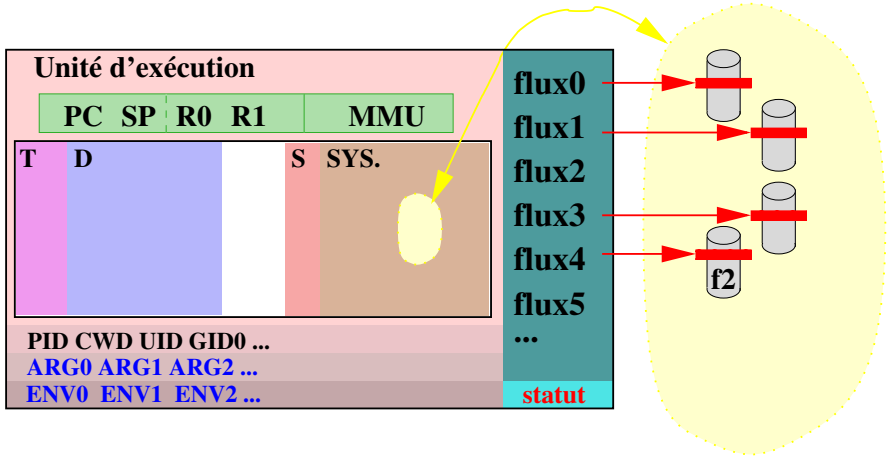
**Exemple 1** open("f1",O_RDONLY) recherche le $1^{er}$ fd libre $\Longrightarrow$ 2.

**Exemple 2** open("f2",O_WRONLY) recherche le $1^{er}$ fd libre $\Longrightarrow$ 4.

**Flags mode** O_RDONLY, O_WRONLY, O_RDWR

**Flags autres** O_TRUNC, O_APPEND, O_CREAT

### 5.2.3 Fermeture

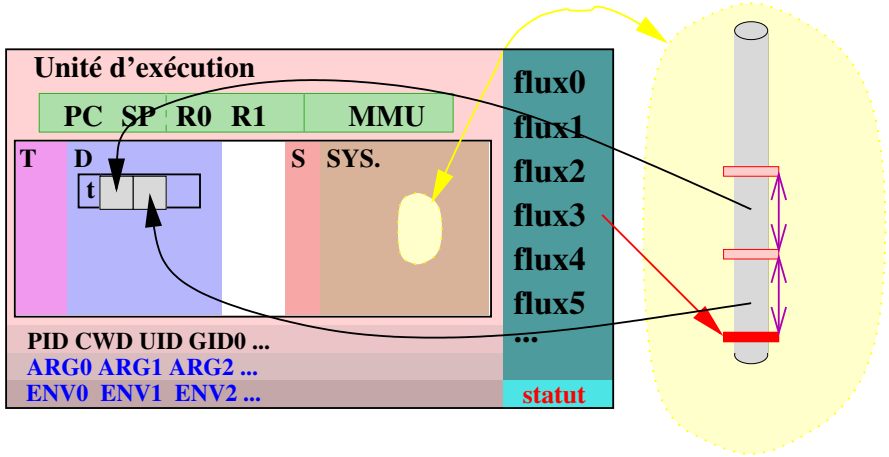



**Synopsis** int close(int fd);

**Fonction** Désalloue le descripteur de flux fd.

**Exemple** close(2) le descripteur 2 est libre

### 5.2.4 Lecture




**Synopsis** size_t read(int fd, void *buf, size_t count);

**Fonction** Essaye de lire count octets du flux fd dans le tampon mémoire buf et incrémente le curseur de count.

**Retour** Le nombre d'octets lus.

**E.O.F** Un retour de la valeur ∅.

**Exemple** nbl=read(3,t,10); nbl=read(3,t+10,10);

### 5.2.5 Écriture

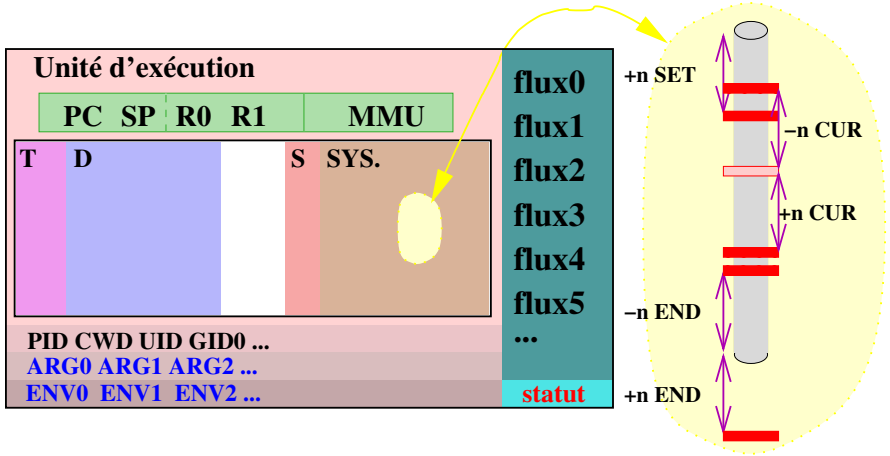
**Synopsis** size_t write(int fd, void *buf, size_t count);

**Fonction** Essaye de d'écrire count octets du flux tampon mémoire buf dans le flux fd et incrémente le curseur de count.

**Retour** En cas de succès, le nombre d'octets écrits sinon -1 et errno est mis à jour.

**Exemple** nbe=write(3,t,10);

### 5.2.6 Positionnement




**Synopsis** int lseek(int fd, off_t offset, int whence);

**Fonction** Positionne le curseur du flux fd à offset octets de la position whence.

**Retour** La nouvelle position de curseur en cas de succès, sinon -1 et errno est mis à jour.

**Exemple** "pos=lssek(3,+10,SEEK_CUR)" avance à partir de la position courante.

**Exemple** "pos=lssek(3,-10,SEEK_CUR)" recule à partir de la position courante.

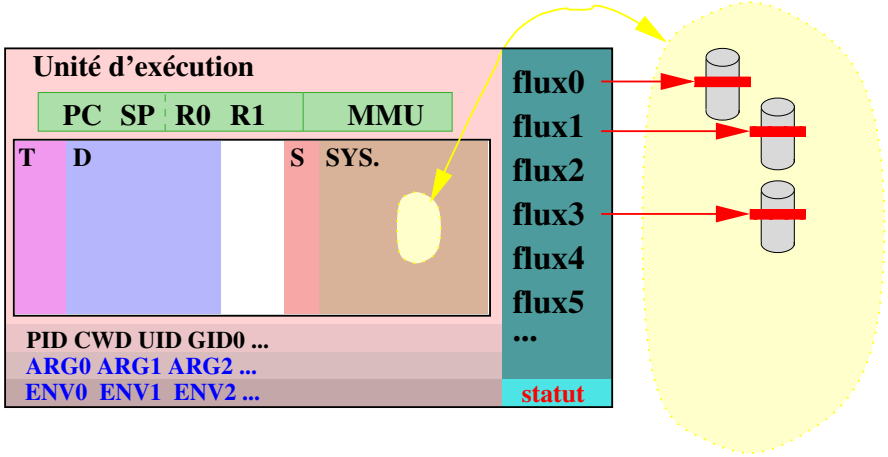
**Exemple** "pos=lssek(3,+10,SEEK_END)" avance à partir de la fin.

**Exemple** "pos=lssek(3,-10,SEEK_END)" recule à partir de la fin.

**Exemple** "pos=lssek(3,+10,SEEK_SET)" avance à partir du début.

**Exemple** "pos=lssek(3,-10,SEEK_SET)" $\Longrightarrow$-1

### 5.2.7 Duplication de descripteurs




**Synopsis** int dup(int fd);

**Fonction** Duplique le descripteur de flux fd et le renvoie.

**Exemple** "dup(3)" Recherche le $1^{er}$ fd libre $\Longrightarrow$ 2.
Les descripteurs 2 et 3 accèdent le même flux.

**Synopsis** int dup2(int oldfd, int newfd)

**Fonction** Fait que le descripteur de flux newfd accède le même flux que oldfd. Si newfd était ouvert, il est fermé.

**Exemple** "dup2(1,0)" Les descripteurs 0 et 1 accèdent le même flux. Le flux 0 est perdu.

### 5.2.8 Exemple

Écrivez un programme à deux arguments src et dest. Il considère src et dest comme 2 chemins de fichiers.

Il crée ou écrase le fichier dest avec au plus les n$*10^{ième}$ octets de src, n allant de 0 à 9 inclus.

*Header et teste du nombre d'arguments*

```
1  #include <stdio.h>
2  #include <unistd.h>
3  #include <sys/types.h>
4  #include <sys/stat.h>
5  #include <fcntl.h>
6  #include <string.h>
7  #include <errno.h>
8
9  int main(int argc, char*argv[])
10 {
11   int i;
12   if ( argc!=3 ) {
13     fprintf(stderr,
14       "%s:Fatal:␣%d␣mauvais␣nb␣d'args␣(2␣attendus)\n",
15       argv[0],argc);
16     return 1;  // ou exit(1)
17   }
```

*Ouverture des flux*

```
20   int fsrc;
21   if ( (fsrc=open(argv[1],O_RDONLY))==-1 ) {
22     fprintf(stderr,
23       "%s:Fatal:␣pb␣ouverture␣%s␣en␣lecture:␣%s\n",
24       argv[0],argv[1],strerror(errno));
25     return 1;
26   }
27   int fdes;
28   if ( (fdes=open(argv[2],
```

```
                        O_WRONLY|O_TRUNC|O_CREAT,
                        0666))==-1  ) {
    fprintf(stderr,
      "%s:Fatal:␣pb␣ouverture␣%s␣en␣ecriture:␣%s\n",
      argv[0],argv[2],strerror(errno));
    return 1;
  }
```

*Lecture écriture, méthode 1*

```
  for (i=0 ; i<10 ; i+=1) {
    char c;
    int status = read(fsrc,&c,1);
    if (status == -1) {
      fprintf(stderr,
        "%s:Fatal:␣pb␣lecture␣%s␣:␣%s\n",
        argv[0],argv[1],strerror(errno));
      return 1;
    }
    if (status == 0) break;

    status =  write(fdes,&c,1);
    if (status == -1 || status == 0) {
      fprintf(stderr,
        "%s:Fatal:␣pb␣ecriture␣%s␣:␣%s\n",
        argv[0],argv[2],strerror(errno));
      return 1;
    }

    if ( lseek(fsrc,9,SEEK_CUR)==-1  ) {
      fprintf(stderr,
        "%s:Fatal:␣pb␣avancee␣ds␣%s␣:␣%s\n",
        argv[0],argv[1],strerror(errno));
      return 1;
    }
  }
```

*Lecture écriture, méthode 2*

```
  for (i=0 ; i<10 ; i+=1) {
    char c;
    if ( lseek(fsrc,i*10,SEEK_SET)==-1  ) {
      fprintf(stderr,
        "%s:Fatal:␣pb␣lseek␣ds␣%s:␣%s",
        argv[0],argv[1],strerror(errno));
      return 1;
    }
    int status = read(fsrc,&c,1);
    if (status == -1) {
      fprintf(stderr,
        "%s:Fatal:␣pb␣lecture␣%s␣:␣%s\n",
        argv[0],argv[1],strerror(errno));
      return 1;
    }
    if (status == 0) break;

    status =  write(fdes,&c,1);
    if (status == -1 || status == 0) {
      fprintf(stderr,
        "%s:Fatal:␣pb␣ecriture␣%s␣:␣%s\n",
        argv[0],argv[2],strerror(errno));
      return 1;
    }
  }
```

*Lecture écriture, méthode 3*

```
  for (i=0 ; i<10 ; i+=1) {
    char t[10];
    int status = read(fsrc,t,10);
    if (status == -1) {
      fprintf(stderr,
        "%s:Fatal:␣pb␣lecture␣%s␣:␣%s\n",
        argv[0],argv[1],strerror(errno));
      return 1;
    }
    if (status == 0) break;
```

```
48
49      status =  write(fdes,&t[0],1);
50      if (status == -1  || status == 0) {
51        fprintf(stderr,
52          "%s:Fatal:␣pb␣ecriture␣%s␣:␣%s\n",
53          argv[0],argv[2],strerror(errno));
54        return 1;
55      }
56    }
```

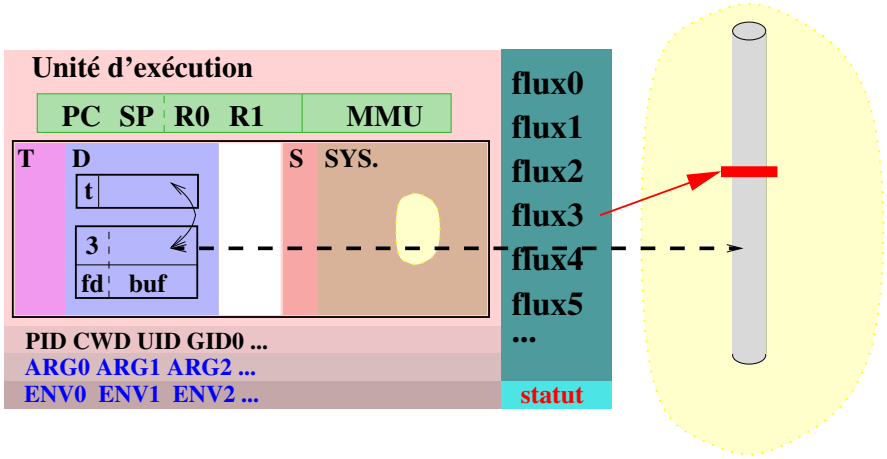
*Terminaison*

```
57    close(fsrc); close(fdes);
58    return 0;
59  }
```

## 5.3 Les flux libc

### 5.3.1 Descripteurs de flux

#### *5.3.1.1 Type FILE et principe*




**type** FILE∗: int fd, void* buf, ...

**localisation** Espace virtuel utilisateur.

**fonction 1** Formatage des entrées/sorties $\Longrightarrow$ facilité d'utilisation.

**fonction 2** Minimiser le nombre d'appels système $\Longrightarrow$ performance.

#### *5.3.1.2 Définitions et opérations sur FILE*

**extern FILE ∗stdin, ∗stdout, ∗stderr;** Variables globales pointant les descripteurs des flux standard d'entrée, de sortie et d'erreur.

**#define EOF ...** constante indiquant fin de fichier.

**int fileno(FILE ∗f)** Renvoie le descripteur de flux noyau associé au flux libc f.

**int feof(FILE∗ f)** Renvoie 0 si le flux libc f est en fin de fichier.

**int fflush(FILE ∗f)**

**Fonction** Sauve si besoin le tampon associé au flux f (en utilisant write).

**Retour** ∅ si le tampon est sauvé et/ou à jour, sinon EOF et met à jour errno.

**int fseek(FILE ∗f, long offset, int whence)**

**Fonction** Sauve si besoin le tampon associé au flux f (en utilisant write). Positionne le curseur du descripteur de flux fd à offset octets de la position whence (SEEK_SET,SEEK_CUR,SEEK_END).

**Retour** ∅ en cas de succès, EOF dans le cas contraire. Dans ce dernier cas errno contient le code d'erreur.

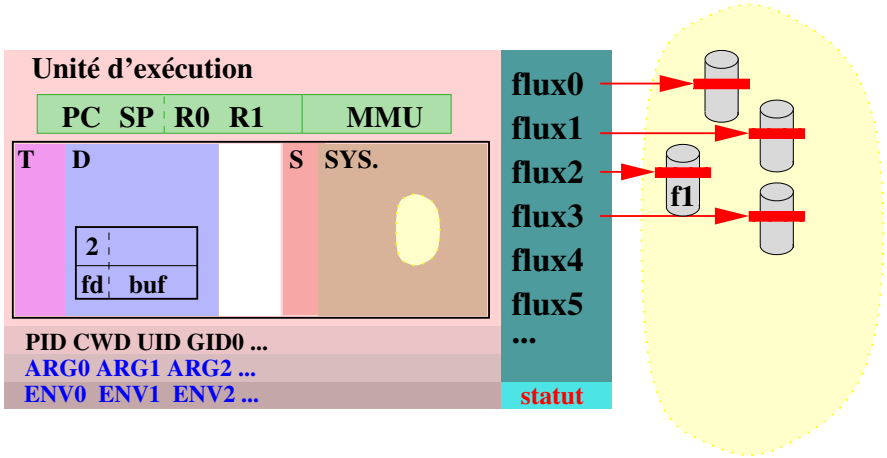
**long ftell(FILE ∗f)**

**Fonction** Donne la position du curseur du flux f.

**Retour** La position en cas de succès, EOF dans le cas contraire. Dans ce dernier cas errno contient le code d'erreur.

### 5.3.2 Ouverture et Fermeture

#### 5.3.2.1 Ouverture




**Synopsis** FILE∗fopen(const char∗ fn, const char∗flag)

**Synopsis** FILE∗fdopen(int fd, const char∗flag)

**Fonction** Associe un descripteur de flux au fichier f ou à fd et le renvoie.

**Retour** Le descripteur de flux ou (FILE∗)∅+ errno.

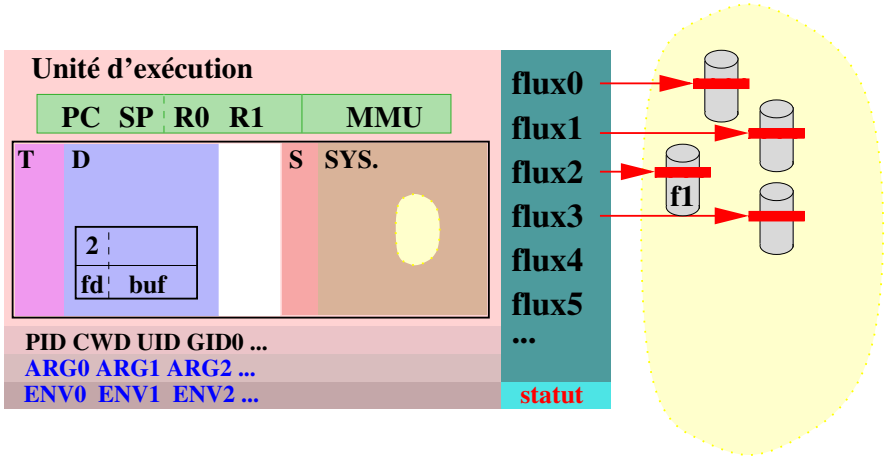
**Exemple** "fopen("f1","r")" crée un flux noyau (2) attaché au fichier f1, alloue un FILE∗ et l'associe au flux noyau.

**Exemple** "fdopen(3,"rw")" alloue un FILE∗ et l'associe au flux noyau (3).

**Modes**

| flag | accès | pos | tronqué | création |
|---|---|---|---|---|
| r | ro | début | non | jamais |
| r+ | rw | début | non | jamais |
| w | wo | début | oui | si besoin |
| w+ | rw | début | oui | si besoin |
| a | w | fin | non | si besoin |
| a+ | rw | fin | non | si besoin |

#### 5.3.2.2 Fermeture




**Synopsis** int fclose(FILE∗ f)

**Fonction** Ferme le flux f.

**Retour** ∅ si pas d'erreur.

**Exemple** "fclose(f)": Écriture du tampon si besoin, libère le flux noyau (2), désalloue la structure f.

### 5.3.3 Entrées/Sorties non formatées

#### 5.3.3.1 Fonctions

**size_t fread(void ∗buf, size_t sze, size_t nbe, FILE ∗f)**

**Fonction** Essaye de lire nbe éléments de taille sze (nbe∗sze octets) du flux f et les range dans le tampon buf.

**Retour** Le nombre d'éléments transférés. ∅ indique soit E.O.F soit une erreur.

**E.O.F** Est indiquée par la fonction feof(f).

**size_t fwrite(void ∗buf, size_t sze, size_t nbe, FILE ∗f)**

**Fonction** Essaye de d'écrire les nbe premiers éléments de taille sze (nbe∗sze octets) du tampon buf dans le flux f.

**Retour** Le nombre d'éléments transférés. ∅ indique une erreur.

#### 5.3.3.2 *Exemples de boucles de lecture*

```
FILE *f;
while ( (n=fread(buf,
            16,5,f)) ) {
  // traiter n elements
}
if ( !feof(f) ) {
  // erreur lecture
}
// E.O.F
```

### 5.3.4 Entrées/Sorties formatées

Ces fonctions sont réservées à la lecture ou l'écriture de fichiers texte.

**int fscanf(FILE *f, const char *fmt, ...)**

**int fprintf(FILE *f, const char *fmt, ...)**

**int scanf(FILE *f, const char *fmt, ...)**

**int printf(const char *fmt, ...)**

**char* fgets(char *l, int size, FILE *f)**

**int sscanf(const char *str, const char *fmt, ...)**

**int sprintf(char *str, const char *fmt, ...)**

### 5.3.5 En pratique

**Règle d'or** Lorsque qu'on travaille sur 1 flux il faut choisir d'utiliser l'interface noyau ou l'interface libc. Par contre on peut très bien lire un flux avec l'interface noyau et un autre avec l'interface libc.

**stderr** Le flux stderr (2) n'est pas tamponné.

**flux tty** En écriture, ils sont tamponné par ligne.

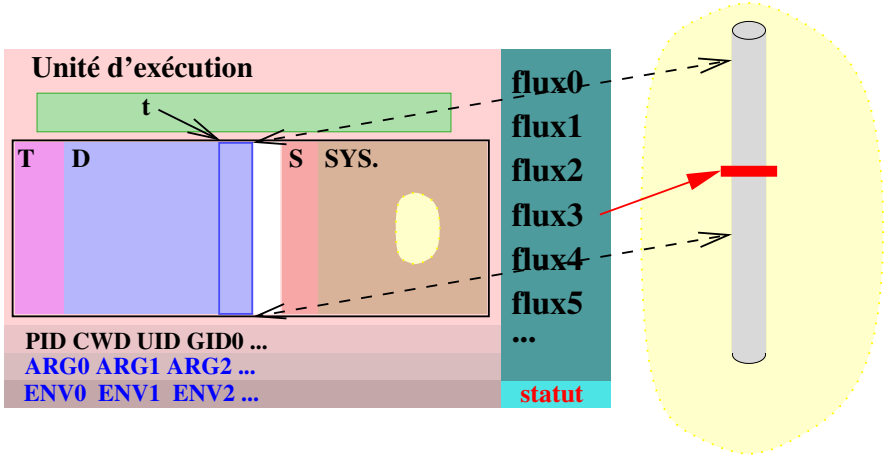
```
printf("hello world"); // tamponné
printf("hello world\n"); // non tamponné
```

**EOF et erreur de lecture** Les fonctions de lecture des flux libc retournent la même valeur pour erreur de lecture et E.O.F. Les cas d'erreurs de lecture sont rares une fois que le flux est ouvert avec succès:

- Fichier régulier local $\Longrightarrow$ défaillance matérielle.
- Fichier régulier réseau $\Longrightarrow$ le noyau bloque la lecture jusqu'à ce que le réseau revienne.
- Fichier tty ou FIFO, c'est impossible.

## 5.4 Mapping

### 5.4.1 Principe




**Synopsis** void*mmap((void*)∅, size_t len, int prot, MAP_SHARED, int fd, off_t offset)

**Fonction** Mappe les octets [offset:offset+len-1] du flux décrit par fd dans l'espace utilisateur. Renvoie l'adresse du mapping.

**Retour** L'adresse du mapping en cas de succès, sinon MAP_FAILED et errno est mis à jour.

**prot** PROT_READ pour accès en lecture, PROT_WRITE pour accès en écriture, PROT_READ|PROT_READ pour accès en lecture et écriture.

**Exemple**

```
char* t=mmap(...);
c=t[10];
t[10] ='A';
```

### 5.4.2 Munmap

**Synopsis** int munmap(void* adr, size_t len)

**Fonction** Unmap la zone mémoire [adr:adr+len-1] de l'espace virtuel utilisateur. Le flux associé n'est pas fermé.

**Retour** ∅ en cas de succès, sinon -1 et errno est mis à jour.

### 5.4.3 Exemple

Lecture du flux standard d'entrée.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <errno.h>

int main(int argc, char* argv[])
{
  int len;
  if ( (len=lseek(STDIN_FILENO,0,SEEK_END))==-1 ) {
    fprintf(stderr,"%s: seek failed: %s\n",
          argv[0],strerror(errno));
    return 1;
  }
  char *p = mmap(0,len,PROT_READ,
          MAP_SHARED,STDIN_FILENO,0);
  if ( p==MAP_FAILED ) {
    fprintf(stderr,"%s: mmap failed: %s\n",
          argv[0],strerror(errno));
    return 1;
  }
  write (STDOUT_FILENO,p,len);
  return 0;
}
```

## 5.5 Comparaison

**Efficacité théorique** Du disque à la variable utilisateur: 1/2/3 copies pour mmap/flux noyau/libc.

**Efficacité pratique** Mal utilisés, les flux noyau peuvent être catastrophiques car les appels système coûtent chers.

Mal utilisés, mmap peut coûter cher car les mapping et unmaping sont des opérations complexes et peuvent générer une fragmentation de l'espace virtuel.

$\Longrightarrow$ Les flux libc donnent une efficacité acceptable.

**Facilité d'utilisation** Les flux libc sont faciles à utiliser surtout si il y a des E/S formatées.

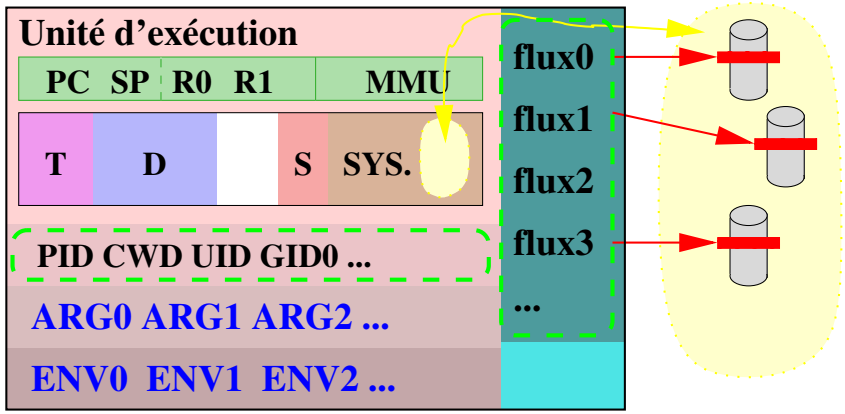
Mmap est le plus complexe, ceci est du au contraintes d'alignement, et l'impossibilité d'ajouter des octets à un fichier mappé.

si on a pas de contraintes d'efficacité sur les E/S et que le tampon ne pose pas de problème $\Longrightarrow$ flux libc.

# 6 Quelques fonctions système

## 6.1 Exec

### 6.1.1 L'appel système execve




**Synopsis** int execve(const char *path, char *const argv[], char *const envp[])

**Fonction** Exécute le programme path dans le processus courant. Seuls les identifiants (PID, UID*, ...) et les flux sont conservés.
Le programme lancé commence par la fonction main* avec argv et envv comme arguments.

**Retour** En cas de succès, **il n'y a pas de retour**, et en cas d'échec, -1 et errno est mis à jour.

**Exemple**

```
char* a[]={ "Aa", "Ab", "Ac", "Ad", ∅};
char* e[]={ "Ea", "Eb", "Ec", ∅};
execve("./a.out",a,e);
```

### 6.1.2 Interface libc

**Synopsis**

int execlp(const char *path, const char * a0, ... , (char*)∅)

int execvp(const char *path, char *const arg[])

**Retour**

En cas de succès, **il n'y a pas de retour**, et en cas d'échec, -1 et errno est mis à jour.

**Fonction**

Ces 2 fonctions appellent execve.

- path est cherché avec la variable d'environnement PATH.
- L'environnement utilisé pour execve est l'environnement courant.

## 6.2 Exit

**Synopsis** void _exit(int statut);

**Retour** Pas de retour

**Fonction**

- Termine le processus et libère tout ce qui était alloué par le processus (E.V, unmapping, fermeture des descripteurs de fichiers ouverts).
- La valeur statut est envoyé au père du processus comme "Statut de fin du processus".
- Le signal SIGCHLD est envoyé au processus père (voir chapitre suivant).
- Le processus 1 devient le père des processus fils.

**Synopsis** void exit(int statut);

**Retour** Pas de retour

**Fonction**

- Libère toutes les allocations système faites par la libc (flux libc, suppression des fichiers temporaires, ...).
- Puis appel de _exit(statut).

## 6.3 Environnement

**Synopsis**

char* getenv(const char *name)
int setenv(const char *name, const char *value, int overwrite)
int unsetenv(const char *name)

**Retour** getenv renvoie la valeur de la variable d'environnement name ou (char*)∅si elle n'existe pas.

setenv et unsetenv renvoient ∅en cas de succès et -1 en cas d'échec.

**Fonction** Ces fonctions permettent de récupérer la valeur d'une variable d'environnement, d'ajouter ou modifier une variable d'environnement et de supprimer une variable d'environnement.

**Note** On peut récupérer les variables d'environnement dans le main:

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

où envv est terminé par un (char*)∅.

## 6.4 Divers

**Synopsis**

int getpid();
int getppid();
char* getcwd(char*buf, size_t bufsz);
int chdir(const char*path);
unsigned int sleep(unsigned int sec);
int usleep(useconds_t usec);
int system(const char* cmd);

**Fonction/Retour**

getpid renvoie le PID du processus, getppid renvoie le PID du processus père.

getcwd et chdir permettent d'obtenir ou de changer le CWD.

sleep (usleep) suspend le processus pendant au moins sec secondes (usec $\mu$s).

system lance un Shell (/bin/sh) qui exécute la commande cmd. Le processus est suspendu jusqu'à la fin du Shell.

# 7 Communication inter-processus

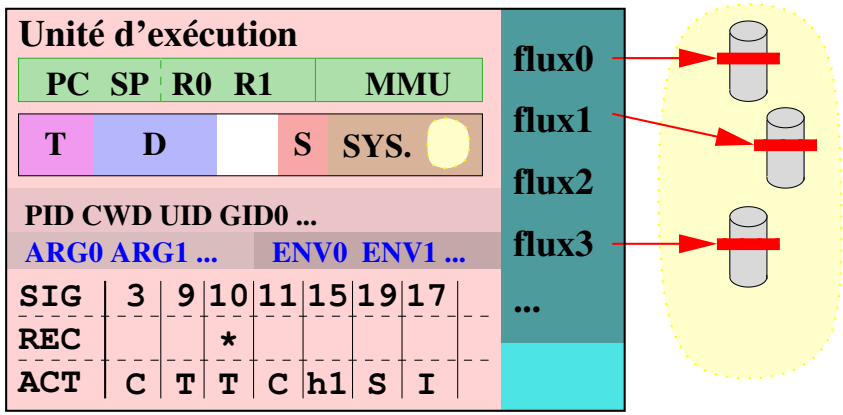
## 7.1 Signaux

### 7.1.1 Qu'est un signal?

#### *7.1.1.1 Définition*

Un signal est un événement (élément dans un ensemble prédéfini) que l'on peut envoyer à un processus. Il y a 5 traitements possibles pour un processus qui reçoit un signal:

1. Ignorer le signal.
2. Se terminer.
3. Se terminer avec core.
   (a) Interrompre l'exécution en cours.
   (b) Générer un core du processus (facultatif).
   (c) Terminer le processus.
4. Se suspendre.
   (a) Interrompre l'exécution en cours.
   (b) Mettre le processus en mode "endormi".
5. Traitement spécifique.
   (a) Interrompre l'exécution en cours
   (b) Exécuter une fonction gestionnaire (même E.V.)
   (c) Reprendre l'exécution en cours




- Une table indiquant pour chaque signal le traitement associé.
- Émettre un signal à un processus P
  $\Longrightarrow$ marquer le signal comme reçu,
  $\Longrightarrow$ le réveiller s'il est suspendu*.
- Que ce passe-t-il si on réenvoie le même signal?

#### *7.1.1.2 L'ensemble des signaux*

| NAME | NUM | Def. Act. | comment |
|---|---|---|---|
| SIGHUP | 1 | Term | Hangup detected on controlling terminal or death of controlling process |
| SIGINT | 2 | Term | Interrupt from keyboard |
| SIGQUIT | 3 | Core | Quit from keyboard |
| SIGILL | 4 | Core | Illegal Instruction |
| SIGABRT | 6 | Core | Abort signal from abort(3) |
| SIGBUS | 7 | Core | Bus error (bad memory access) |
| SIGFPE | 8 | Core | Floating point exception |
| SIGKILL | 9 | Term | Kill signal |
| SIGSEGV | 11 | Core | Invalid memory reference |

| | | | |
|---|---|---|---|
| SIGPIPE | 13 | Term | Broken pipe: write to pipe with no readers |
| SIGALRM | 14 | Term | Timer signal from alarm(2) |
| SIGTERM | 15 | Term | Termination signal |
| SIGUSR1 | 10 | Term | User-defined signal 1 |
| SIGUSR2 | 12 | Term | User-defined signal 2 |
| SIGCHLD | 17 | Ign | Child stopped or terminated |
| SIGCONT | 18 | Cont | Continue if stopped |
| SIGSTOP | 19 | Stop | Stop process |
| SIGTSTP | 20 | Stop | Stop typed at tty |
| SIGTTIN | 21 | Stop | tty input for background process |
| SIGTTOU | 22 | Stop | tty output for background process |

The signals SIGKILL and SIGSTOP cannot be caught, blocked, or ignored.

#### 7.1.1.3 Fonctionnement interne

***P est suspendu en attente d'un signal et reçoit SIGUSR1 (10 avec Term.)***

- S'il est suspendu, il n'a pas de processeur.
- Il reçoit le signal, il est réveillé (éligible)
- Un jour ou l'autre il prend un processeur
- Il termine son travail système (appel système ou interruption)
- Il est prêt à repasser en mode utilisateur
- Il regarde les signaux reçus (trouve SIGUSR1 et action Terminaison)
- Exécute la routine système de terminaison _exit.
- Donne la main.

***P est en mode utilisateur et reçoit SIGUSR1 (10 avec Term)***

- Est-ce possible?
- S'il est en mode utilisateur, il a le processeur.
- Il continue de tourner tranquillement
- Il passe en mode système (appel système ou interruption)
- Il fait son travail système (appel système ou interruption)
- Il est prêt à repasser en mode utilisateur
- Il regarde les signaux reçus (trouve SIGUSR1 et action Terminaison)
- Exécute la routine système de terminaison _exit.
- Donne la main.

***P est en mode user et reçoit SIGTERM (15 avec h1)***

- S'il est en mode utilisateur, il a le processeur.
- Il continue de tourner tranquillement
- ∗ Il passe en mode système (Appel système ou interruption)
- Il fait son travail système (appel système ou interruption)
- Il est prêt à repasser en mode utilisateur
- Il regarde les signaux reçus (trouve SIGTERM et action h1())
- Contexte1= contexte retour normal
- Change le contexte pour lancer h1 (pc=h1 et sp=zone vierge, et retour h1 déclenche l'appel système "retour de gestionnaire")
- Passe en mode utilisateur
- h1() s'exécute
- En mode système "retour de gestionnaire"
- Contexte=context1
- Retour en mode utilisateur (où il avait quitté en ∗)

#### 7.1.1.4 Conclusion

- La durée entre l'envoi d'un signal (quasi instantané) et son traitement est très variable.
- Elle dépend de plein de paramètres (de ce que fait le processus, charge de la machine, ...)
- Les signaux sont très loin du temps réels.
- Lorsqu'un processus s'envoie un signal à lui-même, cette durée peut elle être longue?

### 7.1.2 Interface

#### *7.1.2.1 Envoyer un signal*

**Synopsis** int kill(pid_t pid, int sig);

**Fonction** Envoie le signal sig au processus pid.

**Retour** $\emptyset$ en cas de succès, -1 en cas d'échec et errno est mis à jour.

**Exemple**

```
kill(getpid(),SIGKILL);
printf("Je me suis tué\n");
// verra-t-on ce printf ?
```

#### *7.1.2.2 Fixer le gestionnaire d'un signal*

**Synopsis**

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int sig, sighandler_t handler);
```

**Fonction** Met le gestionnaire du signal sig à handler. Handler est soit

**une adresse en E.V. utilisateur** la fonction gestionnaire.

**SIG_IGN** Ce signal sera ignoré.

**SIG_DFL** Remet le gestionnaire par défaut.

**Retour** Le gestionnaire précédent en cas de succès, SIG_ERR en cas d'échec et errno est mis à jour.

**Exemple**

```
// désactive le <CTL-C>
signal(SIGQINT,SIG_IGN);
```

#### *7.1.2.3 Autres*

**Synopsis**

```
int pause(void);
unsigned int alarm(unsigned int durée);
useconds_t ualarm(useconds_t durée, ∅);
```

**Fonction**

Pause suspend le processus jusqu'à l'arrivée d'un signal non ignoré.

Alarm (resp: ualarm) indique au noyau d'envoyer un signal SIGALRM au processus après au moins durée secondes (resp: $\mu$-secondes).

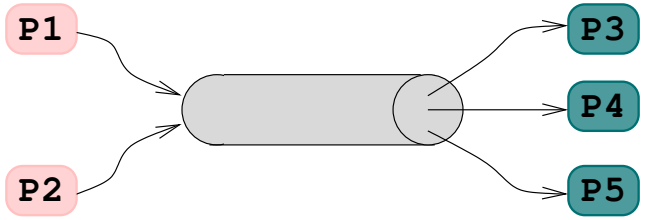
**Retour** Pause renvoie toujours -1.

Alarm et ualarm renvoie $\emptyset$ si il n'y a pas d'alarme en cours, sinon la durée restante pour atteindre l'alarme en cours.

## 7.2 FIFO

### 7.2.1 Introduction

#### *7.2.1.1 Définitions*




**FIFO** Fisrt In First Out (file d'attente à un guichet).

**Canal de communication** Il a une taille maximale et 2 états:

**vide** Il n'y a aucune donnée dans le canal.

**plein** Il y a "taille maximale" données dans le canal.

**Producteurs/Écrivains** Ceux qui écrivent des données dans la FIFO.

**Consomateurs/Lecteurs** Ceux qui lisent les données de la FIFO.

**Canal de synchronisation**

$\Longrightarrow$ Un consomateur est bloqué si la FIFO est vide.

$\Longrightarrow$ Un producteur est bloqué si la FIFO est pleine.

#### 7.2.1.2 Les différentes FIFO

**tty** N<->N, même machine, flux d'octets

**pipe** N<->N, même machine, flux d'octets, processus parenté

**pipe nommé** N<->N, même machine, flux d'octets

**message IPC** N<->N, même machine, flux de messages

**unix socket stream** 1<->1, même machine, flux d'octets

**unix socket datagram** N–>1, même machine, flux de messages

**socket TCP** 1<->1, inter machine, flux d'octets

**socket UDP** N–>1, inter machine, flux de messages

### 7.2.2 Accès aux flux des pipes

**Création d'un pipe non nommé**

int pipe(int fd[2]);

- fd[0] $\Longrightarrow$ sortie de la FIFO, lecture
- fd[1] $\Longrightarrow$ entrée de la FIFO, écriture

**Création d'un pipe nommé**

```
sh> mkfifo path
```

ou

```
sh> mknod path p
```

- Crée le fichier spécial path correspondant à une FIFO.
- Pour écrire ou lire la FIFO il suffit d'ouvrir le fichier path.

**Lecture/écriture d'un pipe** Une fois que l'on a le descripteur de flux (Unix ou libc), il suffit d'utiliser les primitives d'E/S standard.

**Spécificité ouverture**

- Ouverture RO est bloquante si il n'y a pas d'écrivains.
- Ouverture WO est bloquante si il n'y a pas de lecteurs.

**Spécificité lecture**

- Un read(pipefd,buf,n) peut retourner une valeur positive ($> \emptyset$) et inférieure à n sans que l'on soit en fin de flux.
- Un read(pipefd,buf,n) est bloquant si le pipe est vide et qu'il y a des écrivains potentiels.
- Un read(pipefd,buf,n) renvoie $\emptyset$ si le pipe est vide et qu'il n'y a pas d'écrivains.

**Spécificité écriture** Une écriture dans un pipe sans lecteur génère un signal SIGPIPE.

- Terminaison du programme si le gestionnaire de SIGPIPE est SIG_DFL (Terminaison).
- Renvoie -1 si le gestionnaire de SIGPIPE est SIG_IGN ou une fonction. Dans ce cas errno vaut EPIPE.

### 7.2.3 Exemple

Soit fifo1 et fifo2 deux pipes nommées, écrire les programmes ho et ell dont les comportements sont donnés ci-dessous:

- ho écrit h, o et '\n' sur le flux standard de sortie.
- ell écrit e, l et l sur le flux standard de sortie.
- Lancés dans n'importe quel ordre, ils écrivent "hello" sur le flux standard de sortie.

```
sh> ./ho &   # ou ./ell
sh> ./ell    # ou ./ho
hello
sh>
```

#### 7.2.3.1 Solution 1

```
10 // ho
11 int main()
12 {
13   char c;
14   int m2s=open(
15     "fifo", O_WRONLY);
16   int s2m=open(
17     "fifo2",O_RDONLY);
18
```

```
10 // ell
11 int main()
12 {
13   char c;
14   int m2s=open(
15     "fifo", O_RDONLY);
16   int s2m=open(
17     "fifo2",O_WRONLY);
18
```

```c
   write(1,"h",1);
   write(m2s,&c,1);

   read(s2m,&c,1);
   write(1,"o\n",2);

   return 0;
}
```

```c

   read(m2s,&c,1);
   write(1,"ell",3);
   write(s2m,&c,1);


   return 0;
}
```

#### 7.2.3.2 Solution 2

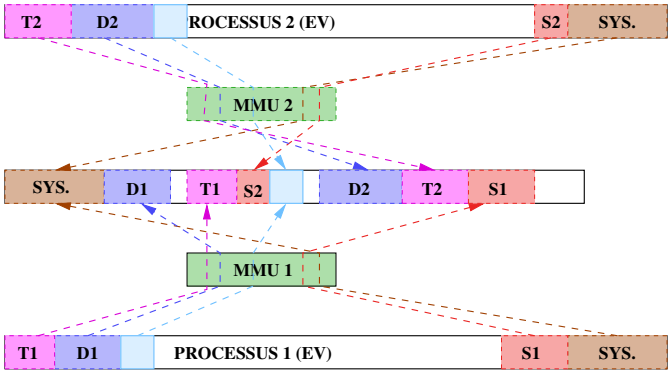
```c
// ho
int main()
{
   char c;

   write(1,"h",1);
   int s2m=open(
      "fifo",O_RDONLY);

   read(s2m,&c,1);
   write(1,"o\n",2);

   return 0;
}
```

```c
// ell
int main()
{
   char c;

   int s2m=open(
      "fifo",O_WRONLY);
   write(1,"ell",3);
   write(s2m,&c,1);


   return 0;
}
```

## 7.3 SHM et Sémaphore

### 7.3.1 Mémoire partagée

#### 7.3.1.1 Principes




En jouant avec les MMU, on peut accrocher le même espace de mémoire physique aux segments de données de 2 processus.

$\Longrightarrow$ On appelle un tel segment , un segment de mémoire partagée.

- Ils ont généralement des adresses virtuelles différentes.
- Les 2 processus peuvent s'échanger des données au travers de ce segment.

#### 7.3.1.2 Les différents outils

**Thread** Le segment données est partagé.

**mmap** Permet de créer des segments de mémoire partagée pour des processus parentés.

**IPC System V** voir "**sh> man svipc**" (sv=System V).

**POSIX Shared memory** voir "**sh> man shm_overview**"

### 7.3.2 Sémaphore

#### 7.3.2.1 Problème

Soit "int*p;" un pointeur dans un segment partagé par 3 processus qui pointe sur la même case mémoire physique.

P1: *p += 1; P2: *p += 3; P3: *p += 5;

On aimerait que quand les 3 processus ont fait leurs modifications ∗p soit incrémenté de 9.

"∗p += n;" est traduit en assembleur par plusieurs instructions par exemple: "r=∗p; r+=n; ∗p=r" où r est un registre du processeur.

Séquencement 1

| P1 | P2 | P3 |
|---|---|---|
| r=∗p | | |
| r+=1 | | |
| ∗p=r | | |
| | r=∗p | |
| | r+=3 | |
| | ∗p=r | |
| | | r=∗p |
| | | r+=5 |
| | | ∗p=r |

∗p + 9

Séquencement 2

| P1 | P2 | P3 |
|---|---|---|
| r=∗p | | |
| | r=∗p | |
| | r+=3 | |
| | ∗p=r | |
| r+=1 | | |
| ∗p=r | | |
| | | r=∗p |
| | | r+=5 |
| | | ∗p=r |

∗p + 6

Séquencement 3

| P1 | P2 | P3 |
|---|---|---|
| r=∗p | | |
| r+=1 | | |
| | r=∗p | |
| | r+=3 | |
| | ∗p=r | |
| | | r=∗p |
| | | r+=5 |
| | | ∗p=r |
| ∗p=r | | |

∗p + 1

Suivant le séquencement des instructions assembleur ∗p peut avoir toutes les valeurs suivantes:
∗p+1, ∗p+3, ∗p+5, ∗p+4, ∗p+6, ∗p+8 et ∗p+9.

#### 7.3.2.2 Sémaphore d'exclusion mutuelle

Un sémaphore simple est une entité ayant un état binaire (LIBRE, BLOQUÉ), une file de processus et 2 fonctions.

**P()**
- si l'état est BLOQUÉ, enfiler le processus et le suspendre.
- si l'état est LIBRE, mettre l'état à BLOQUÉ.

**V()**
- si l'état est LIBRE, erreur.
- si l'état est BLOQUÉ et la file est vide, mettre l'état à LIBRE.
- si l'état est BLOQUÉ et la file est non vide, défiler le $1^{er}$ processus de la file et le réveiller.

Ainsi si mutex est un sémaphore initialisé à {LIBRE, ∅}, ces codes

| P1 | P2 | P3 |
|---|---|---|
| P(mutex); | P(mutex); | P(mutex); |
| ∗p += 1; | ∗p += 3; | ∗p += 5; |
| V(mutex); | V(mutex); | V(mutex); |

garantissent que les 3 modifications de ∗p se feront de façon séquentielle sans s'enchevêtrer.

$\Longrightarrow$ exclusion mutuelle ou exécution atomique.

#### 7.3.2.3 Rendez-vous

Les sémaphores peuvent aussi être utilisés pour synchroniser des processus (eg: fixer des points de rendez-vous).

```
10  S1 <- {BLOQUE, vide}
11  S2 <- {BLOQUE, vide}
12  ...
13  V(S2);
14  P(S1);
15  RDV
16  ...
```

```
10  S1 <- {BLOQUE, vide}
11  S2 <- {BLOQUE, vide}
12  ...
13  P(S2);
14  V(S1);
15  RDV
16  ...
```

#### 7.3.2.4 Les différents outils

**futex** Sémaphore rapide (Fast Mutex).
Ils ne sont utilisables qu'entre threads.

**POSIX thread** Inclus une API de sémaphores.
Ils ne sont utilisables qu'entre threads.

**IPC System V** Voir "**sh> man svipc**" (sv=System V).
Ils sont utilisables sans restriction.

# 8 Processus

## 8.1 Processus Unix

### 8.1.1 Processus lourd

#### *8.1.1.1 Syntaxe*

**Synopsis** pid_t fork(void)

**Fonction** Crée un clone du processus courant. Ce clone est un fils du processus courant.

**Retour** En cas de succès 0 dans le fils et PID du fils dans le père. En cas de d'échec -1 et errno est mis à jour (dans le père seulement).

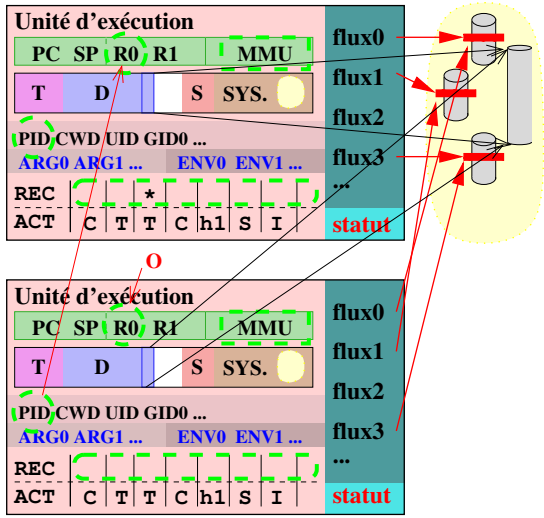
**Exemple**

```
1   ...  // le père tourne
2   int pid =fork();
3   if ( pid==0 ) { // fils
4     execlp("ls","ls","-l",NULL);
5     ...
6     exit(1);
7   }
8   // le père continue ici
9   if ( pid<0 ) { ... ; exit(1); }
10  ...
```

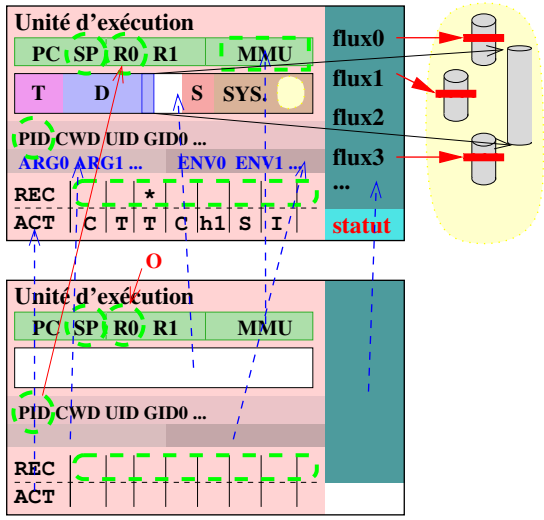
#### *8.1.1.2 Principe*

Les mécanismes de duplication des entités d'un processus lourd sont présentés sur la figure 4 (page 49) et résumés ci-dessous:

**PID:** Nouvelle valeur.
**E.V. Utilisateur:** cloné.
**MMU:** Pointe sur le clone.
**Mémoire partagée:** Conservée.
**Flux:** Conservés (mêmes curseurs)
**Signaux:** Reçus effacés, gestionnaires conservés
**Registres:** Identiques, sauf un (retour du fork)




Processus lourd




Processus léger

Figure 4: Différentes créations de processus

Fils clone parfait du père à part le PID et R0.

Père et fils reviennent en mode utilisateur et exécutent le même code mais dans des espaces physiques différents.

. Sont ils indépendants pour:

1. Le déroulement du code ?
2. La lecture et l'écriture mémoire dans leur E.V?
3. La fermeture et l'ouverture de flux?
4. La lecture et l'écriture des flux?
5. La gestion des signaux?

### 8.1.2 Processus léger

#### *8.1.2.1 Syntaxe*

**Synopsis** pid_t sys_clone(int flag, void* stack, ...)

**Fonction** Crée un clone du processus courant. Ce clone est un fils du processus courant. flags indique ce qui est partagé entre les 2 processus, Le fils aura son pointeur de pile initialisé à stack.

**Retour** En cas de succès 0 dans le fils et PID du fils dans le père. En cas de d'échec -1 et errno est mis à jour (dans le père seulement).

**Exemple**

```
1  ... // le père tourne
2  sys_clone_asm(pid,
3    CLONE_VM|CLONE_FILE|CLONE_SIGHAND|SIGCHLD,
4    ((uchar*)malloc(SZ))+SZ);
5  if ( pid==0 ) {
6    // le fils continue ici
7    execlp("ls","ls","-l",NULL);
8    ...
9    exit(1);
10 }
11 // le père continue ici
12 if ( pid<0 ) { ... ; exit(1); }
```

**Remarque** Le wrapper direct à l'appel système sys_clone n'existe pas dans la libc, car le changement de pile rend son implémentation impossible, syscall n'est d'aucun secours, il faut le faire en assembleur.

Dans la libc il existe une fonction clone qui donne la main au fils dans une fonction (voir man clone).

#### *8.1.2.2 Principe*

Les mécanismes de duplication des entités d'un processus léger sont présentés sur la figure 4 (page 49) et résumés ci-dessous:

**PID:** Nouvelle valeur.
**E.V. Utilisateur:** Partagé.
**MMU:** non modifiée.
**Mémoire partagée:** Conservée.
**Flux:** Conservés (mêmes flux)
**Signaux:** Reçus effacés, gestionnaires partagés
**Registres:** Identiques, sauf SP (pile vierge) et un autre (retour du clone)

Fils clone parfait du père à part le PID, SP, et R0.

Père et fils reviennent en mode utilisateur et exécutent le même code dans le même espace physique.

Sont ils indépendants pour:

1. Le déroulement du code ?
2. Que se passe-t-il si le fils atteint la fin de la fonction (instruction C return) qui a appelé sys_clone?

3. La lecture et l'écriture mémoire dans leur E.V?
4. La fermeture et l'ouverture de flux?
5. La lecture et l'écriture des flux?
6. La gestion des signaux?

### 8.1.3 Attente

#### *8.1.3.1 Principe*

Un processus père peut se mettre en attente d'événements sur ses processus fils:

- terminaison du fils
- suspension du fils
- réactivation du fils

#### *8.1.3.2 Syntaxe*

**Synopsis** pid_t wait(int* status)

**Synopsis** pid_t waitpid(-1,int* status, WUNTRACED|WCONTINUED)

**Fonction** Attend un événement sur un processus fils, et le code dans status. wait ne traque que la terminaison d'un fils. waitpid traque la terminaison, la suspension ou l'activation du fils.

**Retour** Le pid du processus fils qui a subit l'événement, sinon -1 et errno est mis à jour.

**Exemple**

```
1  int status,pid;
2  ...
3  pid = wait(&status);
4  if ( pid == -1 )
5    fprintf(stderr,"%d: pas de fils\n",getpid());
6  else if ( WIFEXITED(status) )
7    fprintf(stderr,
8      "%d: child %d exited with status %d\n",
9      getpid(),pid, WEXITSTATUS(status));
10 else if ( WIFSIGNALED(status) )
11   fprintf(stderr,
12     "%d: child %d exited due to signal %d\n",
13     getpid(),ret, WTERMSIG(status));
14 else
15   fprintf(stderr,"%d: cas inattendu\n",getpid());
16 ...
```

### 8.1.4 Processus zombie


Un processus qui se termine doit délivrer sa terminaison à son père.

Si son père ne *mange* pas sa terminaison, le noyau libère toutes ses allocations et ne conserve que son PID et sa terminaison.

Que se passe-t-il si le père meurt avant son fils?

Les différentes façons pour un père de *manger* la terminaison d'un fils sont:

- Positionner le gestionnaire du signal SIGCHLD.
- Faire un wait ou waidpid qui renvoie le PID du fils.

### 8.1.5 Exemple

```
8  int main(int argc, char* argv[])
9  {
10     int status,pid;
11
12     pid=fork();
13
14     if ( pid==0 ) {  // fils
15         write(1,"hel",3);
```

```
16          exit(0);
17      }
18      // père
19      if ( pid==-1 ) {
20          fprintf(stderr,"%s: échec fork :%s\n",
21              argv[0],strerror(errno));
22          exit(1);
23      }
24
25      wait(&status);
26      write(1,"lo\n",3);
27
28      return 0;
29  }
```

## 8.2 Thread POSIX

### 8.2.1 Introduction

Les threads POSIX ou Pthread sont une API (publiée en 1995) pour le développement d'applications parallèles partageant les mêmes données.

- Gestion de processus légers (création, attente fin, ...).
- Synchronisation (sémaphore).
- Gestion de signaux.
- Gestion d'une zone locale de storage (TLS)
- Gestion de l'ordonnancement.

Un thread POSIX (voir figure 4, page 49) est un processus léger.

POSIX y ajoute à (ou réserve une partie de) l'espace virtuel à la pile et à la TLS du nouveau processus. La TLS contient des variables:

- propres au thread pour sa gestion interne (ex: état du thread, valeur de retour, ...).
- utilisateur qui ne peuvent pas être partagées (ex: errno qui devient une fonction, les variables marquées __thread en C++).
- une table d'action de fin de thread. Pour l'utilisateur sa structure est (clé, pointeur, function). Au départ d'un thread cette table est vide. API propose des fonctions pour ajouter, rechercher, enlever des éléments à la table. En fin de thread tous les "function(pointeur)" de la table sont appelés dans l'ordre inverse d'ajout.

Attention: les piles et les TLS sont dans le même espace virtuel $\Longrightarrow$ tout thread peut modifier la pile ou le TLS de ses collègues.

### 8.2.2 API

#### *8.2.2.1 Création*

**Synopsis**

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
void *(*func) (void *), void *arg);
void pthread_exit(void *ret);

**Fonction**

pthread_create crée un nouveau thread et son point d'entrée est la fonction func avec l'argument arg.
pthread_exit termine le thread avec le statut val.
Un "return x;" dans func est équivalent à pthread_exit(x).

**Retour** En cas de succès 0 sinon un numéro d'erreur (équivalent à errno).

**Exemple**

```
9   void * print(void *str)
10  { printf((char*)str); return NULL; }
11
12  int main(int argc, char*argv) {
13    pthread_attr_t att;
14    pthread_attr_init(&att);
15
16    pthread_t th;
17    pthread_create(&th,&att,print,"hel");
18
```

```
19    sleep(1);
20    printf("lo\n");
21
22    return 0;
23  }
```

#### 8.2.2.2 Attente

**Synopsis**

int pthread_join(pthread_t th, void**statut);

**Fonction**

Attend la fin “pthread_exit(x)” du thread th et délivre son statut (x) *statut.

**Retour** En cas de succès 0 sinon un numéro d’erreur (équivalent à errno).

**Exemple**

```
9   void * print(void *str)
10  { printf((char*)str); return NULL; }
11
12  int main(int argc, char*argv) {
13    pthread_attr_t att;
14    pthread_attr_init(&att);
15
16    pthread_t th;
17    pthread_create(&th,&att,print,"hel");
18
19    pthread_join(th,NULL);
20    printf("lo\n");
21
22    return 0;
23  }
```

#### 8.2.2.3 Sémaphore d’exclusivité mutuelle

**Synopsis**

int pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int pthread_mutex_lock(pthread_mutex_t *mutex);

int pthread_mutex_unlock(pthread_mutex_t *mutex);

**Fonction**

Crée un sémaphore mutex avec les fonction P (lock) et V (unlock).

**Retour** En cas de succès 0 sinon un numéro d’erreur (équivalent à errno).

**Exemple**

```
9   #ifndef NOMUTEX
10
11  pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
12  #define P() pthread_mutex_lock(&mutex)
13  #define V() pthread_mutex_unlock(&mutex)
14
15  #else
16
17  #define P()
18  #define V()
19
20  #endif
```

```
22  int a; // init. à 0 par défaut
23  void * add(void*signe)
24  {
25      int i;
26      for (i=0 ; i<(1<<23) ; i++) {
27          P();
28          int x=a;
29          if (signe!=0)
30              x = x + -2;
31          else
32              x = x + +2;
33          a=x;
34          V();
35      }
```

```
36      return NULL;
37  }

39  int main(int argc, char*argv) {
40    pthread_attr_t att;
41    pthread_attr_init(&att);
42
43    pthread_t tha,  thb;
44    pthread_create(&tha,&att,add, (void*)0);
45    pthread_create(&thb,&att,add, (void*)1);
46
47    pthread_join(tha,NULL);
48    pthread_join(thb,NULL);
49
50    printf("a=%d\n",a);
51
52    return 0;
53  }
```

**Expérimentation**

```
sh> gcc -DNOMUTEX mutex.c -lpthread && ./a.out
a=-7197708
sh> gcc -DNOMUTEX mutex.c -lpthread && ./a.out
a=-7081580
sh> gcc mutex.c -lpthread && ./a.out
a=0
sh> gcc mutex.c -lpthread && ./a.out
a=0
sh>
```

Sur un PC Linux bi-processeurs, La version avec le sémaphore est environ 25 fois plus lente.

$\Longrightarrow$ Quel rapport serait attendu?

$\Longrightarrow$ À quoi est dû le surplus?

### 8.2.3 Threads POSIX sous Linux

**Thread = Processus** Processus léger créé avec l'appel système sys_clone.

**Processus regroupés** Les threads partageant le même E.V sont regroupés dans un groupe. Appelons T$m$ le processus initial et T$a$ les autres.

**exit(s)** Dans un thread termine tous les thread du group.

**getpid() et getppid()** Dans un thread T$i$ donnent celui de T$m$.

$\Longrightarrow$ Les processus threads sont masqués

**syscall(SYS_gettid)** Renvoie le vrai PID du processus.

$\Longrightarrow$ Pour un T$m$, syscall(SYS_gettid)=getpid()

**ps -Af** Affiche tous les processus T$m$.

**ps -LAf** Affiche tous les processus T$m$ et T$a$.

**Gestionnaire de signal** Partagé par les T$i$.

**Envoi d'un signal** Il faut l'envoyer à syscall(SYS_gettid).

**Fork dans un T$i$** Le processus T$i$ uniquement est cloné, le père du clone est T$m$.

$\Longrightarrow$ Tous les T$i$ peuvent faire un wait sur ce fils.

$\Longrightarrow$ Tous attendront la mort du clone.

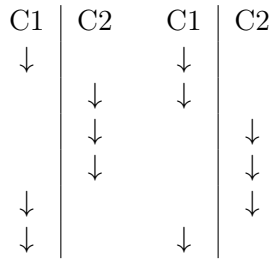
$\Longrightarrow$ Tous sauf 1 recevront -1 avec errno "no child".

## 8.3 Fonction réentrante et thread-safe

### 8.3.1 Problème

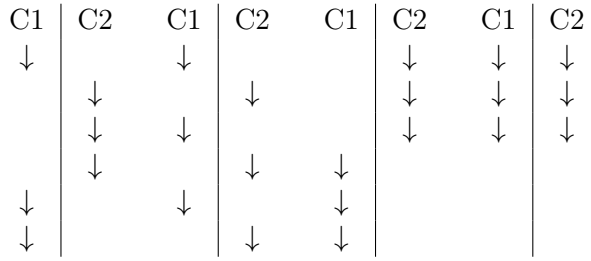
Soit un code $C_1$ manipulant des données $D_1$, et un code $C_2$ manipulant des données $D_2$. $C_1$ et $C_2$ peuvent s'exécuter en :

mode interruption




$C_1$ s'interrompt, $C_2$ s'exécute complètement puis $C_1$ reprend.

concurrence




$C_1$ et $C_2$ s'exécutent sans ordre préétabli, et éventuellement en même temps.

Il faut que quelque soit le scénario à la fin les données $D_1$ et $D_2$ contiennent les bons résultats des 2 codes et soient dans un état cohérent.

Si $D_1$ et $D_2$ sont disjoints les codes sont résistants à une exécution concurrente et en mode interruption.

### 8.3.2 Définition

**Fonction réentrante** f est réentrante si un second appel à f se déroulant pendant le premier appel donne un résultat correct pour les 2 appels.

Par exemple, f se déroule, un signal est attrapé et le gestionnaire du signal appel f.

**Fonction thread-safe** f est thread-safe si deux appels en parallèle donnent un résultat correct pour les deux appels.

Par exemple, 2 threads exécutent une fonction f en même temps,

### 8.3.3 Appels système et fonctions de la libc

**Appels système** Les appels système sont threadsafe.
Au niveau utilisateur, ils n'ont pas besoin d'être réentrants car si le gestionnaire de signal est appelé, il n'y a pas d'appel système en cours.

**Fonctions de la libc** Les fonctions de la libc sont réentrantes et thread-safe sauf mention contraire dans la page de man.
Dans ce cas il existe une fonction équivalente qui l'est.

### 8.3.4 Exemple

Les fonctions de lecture et écriture de la libc sont thread-safe.

Elles fonctionnent ainsi:

- Posent un verrou
- Font leur job
- Relâchent le verrou

De ce fait le code ci-contre pose autant de verrous qu'il lit de caractères.

```
1  #include <stdio.h>
2
3  int main(int argc, char*argv[])
4  {
5    char c; int len=0;
6    while ( fread(&c,1,1, stdin)==1 )
7      len += 1;
8    printf("len=%d\n",len);
9    return 0;
10 }
```

Toutes les fonctions de lecture et écriture des flux libc ont leurs équivalents sans verrou (ex: printf $\Longrightarrow$ printf_unlocked).

```
sh> gcc test.c && time ./a.out < 10m
len=10485760
real 0m0.233s
user 0m0.230s
sys 0m0.003s
sh> gcc -Dfread=fread_unlocked test.c && \
    time ./a.out < 10m
len=10485760
real 0m0.180s
user 0m0.170s
sys 0m0.005s
sh>
```

La capture d'écran ci-dessus montre que les poses et les relâchements du verrou prennent 1/3 du temps d'exécution.

De plus en optimisant, gcc inline fread_unlocked, ce qui donne:

```
sh> gcc -O2 -Dfread=fread_unlocked test.c && \
```

```
    time ./a.out < 10m
len=10485760
real 0m0.031s
sh>
```

# 9 Travaux pratiques

## 9.1 Shell (base)

Pour ces exercices n'oubliez pas:

**la complétion** touche tabulation

- 1 tab $\Longrightarrow$ complétion si pas de conflit.
- 2 tab en cas de conflit $\Longrightarrow$ les possibilités.

**le rappel de commandes** touches flèches: $\uparrow$ et $\downarrow$

**la modification de commandes rappelées** touches flèches: $\leftarrow$ et $\rightarrow$

Sur votre écran, il ne doit y avoir qu'un terminal xterm de taille raisonnable ($\sim$ 25 lignes, $\sim$ 100 colonnes) et si vous voulez de l'aide avec une police lisible à un mètre (<CTL-+CLIC-DROIT> et choisir Huge).

Dans ce document, "<C-R>" signifie la presion de la touche entrée du clavier, "<CTL-X>" signifie la pression simultanée des touches du clavier contrôle et x.

***Exercice 1***

1. Dans un terminal, placez vous dans le répertoire ~/sys/tp1.

   ```
   sh> mkdir -p ~/sys/tp1
   ```

   ```
   sh> cd ~/sys/tp1
   ```

2. Affichez le CWD.

   ```
   sh> pwd
   ```

3. Allez dans votre HOME.

   ```
   sh> cd
   ```

4. Affichez le CWD.

   ```
   sh> pwd
   ```

5. Retournez dans le répertoire ~/sys/tp1.

   ```
   sh> cd -
   ```

6. Exécutez la commande ls sans argument dans votre HOME et sans changez de CWD de 4 manières différentes.

   ```
   sh> ( cd ; ls )
   ```

   ```
   sh> ( cd ~ ; ls )
   ```

   ```
   sh> ( cd $HOME ; ls )
   ```

   ```
   sh> ( cd ../.. ; ls )
   ```

7. Ouvrez un terminal en background qui a le même CWD que ce terminal.

   ```
   sh> xterm &
   ```

8. Vérifiez que les terminaux ont le même CWD.

   ```
   sh> pwd # dans le 1er terminal
   ```

   ```
   sh> pwd # dans le 2nd terminal
   ```

9. Fermez le 2nd terminal sans cliquer avec la souris.

   en tapant <CTL-D> dans le $2^{nd}$ terminal

   ```
   sh> exit # dans le 2nd terminal
   ```

### *Exercice 2*

1. Ouvrez un terminal et placez vous dans votre HOME.

```
sh> cd
```

2. Sans changer de CWD, copiez le fichier /pub/ia/sys/shell/fich.dat dans le répertoire /tmp sous le nom f1.

```
sh> cp /pub/ia/sys/shell/fich.dat  /tmp/f1
```

3. Quelle est la taille en octet de /tmp/f1.

```
sh> ls -l /tmp/f1
```

4. Sans changer de CWD, déplacez le fichier /tmp/f1 dans /tmp/d1/d2/f1-de-d2 (il faudra créer le répertoire /tmp/d1/d2).

```
sh> mkdir -p /tmp/d1/d2
```

```
sh> mv /tmp/f1  /tmp/d1/d2/f1-de-d2
```

5. Sans changer de CWD, copiez l'arborescence /tmp/d1/d2 dans votre HOME sous le nom sys/tp1/exo1.

```
sh> mkdir -p sys/tp1
```

```
sh> cp -r /tmp/d1/d2  sys/tp1/exo1
```

6. Sans changer de CWD, déplacez l'arborescence /tmp/d1 dans votre HOME dans le répertoire sys/tp1/exo1 de votre HOME

```
sh> mv /tmp/d1  sys/tp1/exo1
```

7. Vérifiez que le répertoire sys/tp1/exo1 contient 2 fichiers: f1-de-d2 et d2.

```
sh> ls sys/tp1/exo1
```

8. Vérifiez que les fichiers sys/tp1/exo1/f1-de-d2 et sys/tp1/exo1/d1/d2/f1-de-d2 sont de même taille.

```
sh> ls -l sys/tp1/exo1{/,/d1/d2}/f1-de-d2
```

### *Exercice 3*

1. Ouvrez un terminal et placez vous dans votre HOME.

```
sh> cd
```

2. Visualisez le fichier /pub/ia/sys/shell/expr-arith.dat.

```
sh> less /pub/ia/sys/shell/expr-arith.dat # q pour quitter
```

```
sh> cat /pub/ia/sys/shell/expr-arith.dat # sans pageur
```

3. Calculez cette expression.

```
sh> echo $[$(cat /pub/ia/sys/shell/expr-arith.dat)]
```

4. Créez le fichier /tmp/expr1 contenant cette expression multipliée par elle même en utilisant la commande echo.

```
sh> echo > /tmp/expr1 "( $(cat /pub/ia/sys/shell/expr-arith.dat) ) * ( $(cat /pub/ia/sys/shell/expr-arith.dat) )"
```

5. Calculez l'expression de /tmp/expr1

```
sh> echo $[$(cat /tmp/expr1)]
```

6. Créez le fichier /tmp/expr2 contenant l'expression de /pub/ia/sys/shell/expr-arith.dat multipliée 3 fois par elle même en utilisant la commande echo et une variable.

```
sh> x="($(cat /pub/ia/sys/shell/expr-arith.dat))"
```

```
sh> echo > /tmp/expr2 $x \*$x \*$x
```

7. Calculez l'expression de /tmp/expr2

```
sh> echo $[$(cat /tmp/expr2)]
```

8. Créez le fichier /tmp/expr3 contenant l'expression qui est la somme des expressions de /tmp/expr1 et /tmp/expr2 en utilisant la commande cat.

```
sh> cat > /tmp/expr3 - /tmp/expr1 - /tmp/expr2 -
(<C-R>
<CTL-D>)+(<C-R>
<CTL-D>)<C-R>
<CTL-D>sh>
```

L'argument '-' est interprété par la commande cat comme le flux stdin.

9. Calculez l'expression de /tmp/expr3

10. détruisez les fichiers expr du répertoire /tmp

```
sh> rm /tmp/exp*
```

### *Exercice 4*

Créez le fichier a qui contient 1000 'A' en utilisant echo et cat.

1. Créez le fichier a qui contient 1 seul A.

```
sh> echo -n A > a
```

```
sh> cat a
```

```
sh> ls -l a
```

2. Créez le fichier a qui contient 10 fois le fichier a (on utilisera un fichier intermédiaire /tmp/1).

```
sh> cat a a a a a a a a a a > /tmp/1 ; mv /tmp/1 a
```

3. Répétez 2 fois la commande.

Utilisez l'historique

4. Vérifiez le contenu et la taille du fichier a.

```
sh> cat a
```

```
sh> ls -l a
```

### *Exercice 5*

1. Dans un terminal où CWD est /tmp, créez le répertoire essai dans votre HOME.

```
sh> cd /tmp
```

```
sh> mkdir ~/essai
```

2. Ouvrez un second terminal dont le HOME est ce répertoire essai.

```
sh> HOME=~/essai xterm &
```

3. Dans le second terminal allez dans votre HOME et vérifiez que c'est bien le répertoire essai.

```
sh> cd
```

```
sh> pwd ; echo ~ $HOME
```

4. De ce second terminal, allez dans /tmp, et créez le fichier f1 vide dans votre vrai HOME.

```
sh> cd /tmp
sh> > ~/../f1
```

5. Du premier terminal, allez dans /tmp, et vérifiez que le fichier f1 vide existe bien dans votre HOME, puis supprimez le.

```
sh> cd /tmp
sh> > ~/f1
sh> rm ~/f1
```

6. Fermez le second terminal sans cliquer avec la souris.

en tapant <CTL-D> dans le $2^{nd}$ terminal

```
sh> exit # dans le 2nd terminal
```

***Exercice 6***

1. Tapez la commande ls, Quel est son chemin absolu?

```
sh> ls
```

```
sh> which ls
```

2. Ouvrez un second terminal dont le PATH est vide.

```
sh> PATH=   /usr/bin/xterm &
```

3. Dans ce second terminal tapez la commande ls.

```
sh> ls # bash: ls: command not found
```

4. Dans ce second terminal tapez la commande ls sans utilisez le PATH.

```
sh> /bin/ls
```

5. Restaurez le PATH du second terminal, sans le saisir et sans utilisez de copier/coller.

   (a) Depuis le premier terminal, copiez la valeur du PATH dans /tmp/path

   ```
   sh> echo -n $PATH > /tmp/path
   ```

   (b) Dans le second terminal utilisez ce fichier pour restaurer le PATH.

   ```
   sh> PATH="$(cat /tmp/path)" # bash: cat: command not found
   ```

   (c) Dans le premier terminal trouvez le chemin absolu de cat.

   ```
   sh> which cat
   ```

   (d) Dans le second terminal, restaurez le PATH.

   ```
   sh> PATH="$(/bin/cat /tmp/path)"
   ```

   (e) Dans le second terminal, vérifiez votre PATH en tapant la commande ls.

   ```
   sh> ls
   ```

6. Fermez le second terminal sans cliquer avec la souris.

   en tapant <CTL-D> dans le $2^{nd}$ terminal

```
sh> exit # dans le 2nd terminal
```

***Exercice 7***

1. Créez dans votre HOME le répertoire bin. Copiez dans ce répertoire les commandes ls et cat sous les noms myls et mycat.
2. Lancez les commandes myls et mycat et ~/bin/myls et ~/bin/mycat
3. Ajoutez ce répertoire bin à votre PATH.
4. Lancez les commandes myls et mycat (elles doivent fonctionner).
5. Créez un nouveau terminal. Dans ce terminal, les commandes myls et mycat fonctionnent elles encore?
6. Créez un nouveau terminal à partir des menus/boutons du Window Manager. Dans ce terminal, les commandes myls et mycat fonctionnent elles encore?
7. Éditez le fichier ~/.bashrc et écrivez la ligne qui ajoute votre répertoire bin au PATH.
8. Créez un nouveau terminal à partir des menus/boutons du Window Manager. Dans ce nouveau terminal, lancez les commandes myls et mycat (elles doivent fonctionner).
9. Fermez les $2^{nd}$ et $3^{ième}$ terminaux sans cliquer avec la souris.

***Exercice 8***

1. Lancez un xpdf en avant plan (foreground), puis tapez <CTL-C> dans le terminal. Le <CTL-C> envoie une demande de terminaison au processus leader du groupe terminal.

   **sh>** **xpdf** /pub/ia/sys/shell/sys-poly.pdf

   Le xpdf se ferme, il est donc le leader du groupe terminal et obéissant.

2. Tapez <CTL-C> dans le terminal. Il ne se passe rien. Pourtant le Shell est bien le leader, il reçoit la demande de terminaison, mais il n'est pas obéissant et l'ignore.

3. Lancez un xpdf en arrière plan (background), puis tapez <CTL-C> dans le terminal.

   **sh>** **xpdf** /pub/ia/sys/shell/sys-poly.pdf **&**

   Le xpdf est en vie, il n'a pas reçu la demande de terminaison, c'est le Shell qui est le leader du groupe terminal et se contente d'écrire "↑C" sur la demande de terminaison. Vous pouvez exécuter des commandes (ex: ls).

4. Ramenez le xpdf en avant plan. La builtin commande **fg** ramène le dernier processus lancé en arrière plan en avant plan.

   **sh>** **fg**

   xpdf redevient le leader, et le Shell ne peut plus lire le flux stdin. Vous pouvez le terminer avec <CTL-C>, puis relancez le en avant plan.

5. Ramenez le xpdf en arrière plan. Ceci ce fait en 2 étapes:
   - Tapez <CTL-Z>. Le <CTL-Z> envoie la demande de se suspendre au processus leader du groupe terminal. xpdf étant le leader, il reçoit la demande et étant obéissant, il se suspend. Du coup le tty est libre, le Shell redevient le leader du groupe.
   - Comme le Shell est leader, on peut taper des commandes

     **sh>** **bg**

     La builtin commande bg réveille et fait passer en arrière plan le dernier processus suspendu. Le Shell reste le leader.

6. Vous avez créé un terminal en oubliant de le lancer en arrière plan

   **sh>** **xterm**

   Faites que vos deux terminaux soient utilisables.

### *Exercice 9*

Créez un terminal en mettant le titre de sa fenêtre et le label de son icône à la chaine de caractères "A".

1. Affichez l'aide de la commande xterm.

   ```
   sh> xterm -h
   ```

   L'option -h ou –help fonctionnent sur la plupart des commandes Unix. Cette aide en ligne est plus succincte que la page de man.

2. Paginez la avec le pageur less.

   ```
   sh> xterm -h | less
   ```

3. Allez en bas de l'aide.

   Tapez la touche >

4. Retournez en haut de l'aide.

   Tapez la touche <

5. Rechercher les options qui ont un rapport avec le titre de la fenêtre.

   Tappez /title<C-R> (/ indique la recherche de mots)

   Tapez la touche n (pour trouver la première occurence)

   Tapez la touche n (pour trouver la suivante et ainsi de suite jusqu'au bout du fichier)

   Vous devez trouver les options -T, -name, -title.

   Tapez la touche q pour quitter less.

6. Essayez de lancer xterm avec l'option -name et vérifiez que le titre de la fenêtre et de l'icône sont mis à jour.

   ```
   sh> xterm -name A &
   ```

7. Fermez le terminal

8. Fermez le terminal A sans cliquer avec la souris.

   en tapant <CTL-D> dans le terminal A

   ```
   sh> exit # dans le terminal A
   ```

### *Exercice 10*

1. Déterminez votre groupe terminal. Il est identifié par le fichier spécial tty correspondant à votre terminal (la commande tty renvoie le chemin de ce fichier).

   ```
   sh> tty
   ```

   On l'appellera "/dev/pts/N-I".

2. Créez un terminal A (les titres de fenêtre et d'icône sont "A")

   ```
   sh> xterm -name A &
   ```

3. Déterminez le fichier spécial correspondant au terminal A.

   ```
   sh> tty # dans le terminal A
   ```

   On l'appellera "/dev/pts/N-A".

4. Dans le terminal A changez la touche "retour-arrière"par m, et la touche fin de fichier <CTL-D> par d.

   ```
   sh> stty erase m # dans le terminal A
   sh> stty eof f # dans le terminal A
   ```

   Écrivez quelque chose dans le terminal A et essayez les touches "retour-arrière"et m.
   Lancez la commande cat et indiquez la fin du flux stdin en tapant sur la touche f.

   ```
   sh> cat # dans le terminal A
   aaa
   aaa
   fsh>
   ```

5. Restaurez les fonctions d'effacement de caractère et de fin du flux stdin du terminal A à leurs valeurs initiales.

   ```
   sh> stty erase 0x7f # dans le terminal A
   sh> stty eof 0x04 # dans le terminal A
   ```

   Essayez la touche "retour-arrière"et le <CTL-D> pour voir s'ils sont bien restaurés.

6. Créez un terminal B (les titres de fenêtre et d'icône sont "B") et déterminez le fichier spécial correspondant au terminal B. On l'appellera "/dev/pts/N-B".

7. Lancez une commande cat sur le terminal A pour que les caractères tapés sur ce terminal s'affichent sur le terminal B.

   sh> cat > /dev/pts/N-B

8. Indiquez à cat la fin de fichier.

   en tapant <CTL-D> dans le terminal A

9. Lancez une commande echo sur le terminal B qui écrit "hello world" sur le terminal A.

   sh> echo hello world > /dev/pts/N-A

10. Sur le terminal initial, lancez une commande cat pour que les caractères tapés sur le terminal A soient écrits sur terminal B. (Attention: pour évitez les conflits de lecture entre votre commande et le Shell de A, lancez un sleep 2000 (> 30mn) sur le terminal A).

11. Fermez les terminaux A et B sans cliquer avec la souris.

### *Exercice 11*

1. Créez 3 terminaux dont les titres de fenêtre et icône sont IN et OUT et ERR. Récuperez leurs fichiers terminaux, et lancez un sleep de 2000 secondes dans chacun d'eux.

2. Dans le terminal initial, lancez un bash qui lit ses commandes sur le terminal IN, écrit ses sorties sur le terminal OUT et écrit ses erreurs sur le terminal ERR.
   Attention: Le bash réouvre les fichiers tty, pour l'en empécher il faut que sont stdin soit un pipe dont l'autre extrémité est le fichier tty.

3. Fermez les terminaux IN, OUT et ERR sans cliquer avec la souris.

### *Exercice 12*

1. Affichez le fichier /pub/ia/sys/shell/unsort.dat (commande less ou cat).

2. Triez le fichier /pub/ia/sys/shell/unsort.dat (commande sort) dans le fichier /tmp/1.

3. Générez /tmp/2 à partir /tmp/1 en enlevant les lignes semblables (commande uniq).

4. Calculez le nombre de lignes enlevées (nombre de lignes du fichier fich: wc -l < fich).

5. Refaites le même calcul sans créer de fichiers intermédiaires en une seule commande (utilisez les pipes).

***Exercice 13***

Quelques commandes pour sélectionner des lignes, des parties de lignes ou des colonnes.

1. Affichez le fichier /pub/ia/sys/shell/select.dat (commande less ou cat) (les espaces sont formés d'un seul caractère de tabulation).

2. Commande **cut** sélection de colonnes
   Par défaut, cut commence une nouvelle colonne à chaque caractère '\t'. Ainsi "\t\t" est une ligne à 3 colonnes vides. Le caractère qui sépare les colonnes est modifiable par une option.

   (a) Affichez la $3^{ième}$ colonne du fichier /pub/ia/sys/shell/select.dat.

   ```
   sh> cut -f 3 /pub/ia/sys/shell/select.dat
   ```

   ou

   ```
   sh> cut -f 3 < /pub/ia/sys/shell/select.dat
   ```

   (b) Affichez les $1^{ère}$ et $3^{ième}$ colonnes du fichier /pub/ia/sys/shell/select.dat.

   ```
   sh> cut -f 1,3 /pub/ia/sys/shell/select.dat
   ```

   (c) Notez que l'on ne peut pas inverser des colonnes avec cut.

   (d) Affichez les GID du fichier /etc/passwd (colonne 4).

3. Commande **head tail grep** sélection de lignes

   (a) Affichez les 2 premières lignes du fichier /pub/ia/sys/shell/select.dat.

   ```
   sh> head -n 2 /pub/ia/sys/shell/select.dat
   ```

   (b) Affichez le fichier /pub/ia/sys/shell/select.dat moins ses 2 dernières lignes.

   ```
   sh> head -n -2 /pub/ia/sys/shell/select.dat
   ```

   (c) Affichez les 3 dernières lignes du fichier /pub/ia/sys/shell/select.dat.

   ```
   sh> tail -n 3 /pub/ia/sys/shell/select.dat
   ```

   (d) Affichez les lignes contenant Alice du fichier /pub/ia/sys/shell/select.dat.

   ```
   sh> grep Alice /pub/ia/sys/shell/select.dat
   ```

4. Commande **sed** sélection de lignes et de parties de lignes

   (a) Affichez les 3 premières lignes du fichier /pub/ia/sys/shell/select.dat.

   ```
   sh> sed -e 4,\$d /pub/ia/sys/shell/select.dat
   ```

   "4,\$" signifie les lignes 4 à la fin, "d" signifie delete.

   (b) Affichez le fichier /pub/ia/sys/shell/select.dat moins ses 3 premières lignes.

   ```
   sh> sed -e 1,3d /pub/ia/sys/shell/select.dat
   ```

   (c) Affichez les 3 dernières lignes du fichier /pub/ia/sys/shell/select.dat. Pour cela, il faut connaitre le nombre de ligne.

   ```
   sh> sed -e "$[ $( wc -l < /pub/ia/sys/shell/select.dat) - 2 ],\$d" /pub/ia/sys/shell/select.dat
   ```

   (d) Affichez les lignes contenant Alice du fichier /pub/ia/sys/shell/select.dat.

   ```
   sh> sed -e /Alice/\!d /pub/ia/sys/shell/select.dat
   ```

   "/Alice/" les lignes contenant Alice (selection par contenu),
   "d" détruire les lignes sélectionnées,
   "!d" détruire les lignes non sélectionnées,
   "\!d" \! pour échapper à l'expansion par le shell de !d (la commande de l'historique la plus récente commençant par d).

   (e) Affichez les lignes contenant Alice du fichier /pub/ia/sys/shell/select.dat ($2^{nde}$ solution).

   ```
   sh> sed -n -e /Alice/p /pub/ia/sys/shell/select.dat
   ```

   "-n" détruire les lignes non sélectionnées,
   "/Alice/" les lignes contenant Alice (selection par contenu),
   "p" afficher les lignes sélectionnées.

   (f) Affichez la $3^{ième}$ colonne du fichier /pub/ia/sys/shell/select.dat.

   ```
   sh> sed -e "s+[^\t]*\t[^\t]*\t\([^\t]*\)+\1+" /pub/ia/sys/shell/select.dat
   ```

   - pas de sélection de lignes comme "3,$" ou "/Alice/" donc l'action "s" s'applique à toutes les lignes.
   - "s+aaa+ccc+" remplacer aaa par ccc,
   - "[^\t]*" une suite de 0 ou n caractères sans tab,
   - "[^\t]*\t"[^\t]*" deux suites précédentes séparées par un tab,
   - "\(aaaa\)" numéroter la séquence aaaa,
   - "\1" fait référence à la séquence numérotée 1,
   - "\n" fait référence à la séquence numérotée n.

   (g) Affichez les $1^{ère}$ et $3^{ième}$ colonnes du fichier /pub/ia/sys/shell/select.dat.

   ```
   sh> sed -e "s+\([^\t]*\)\t[^\t]*\t\([^\t]*\)+\1\t\2+" /pub/ia/sys/shell/select.dat
   ```

   (h) Inverser les colonnes du fichier

   ```
   sh> sed -e "s+\([^\t]*\)\t\([^\t]*\)\t\([^\t]*\)+\3\t\2\t\1 /pub/ia/sys/shell/select.dat
   ```

   (i) Affichez les GID du fichier /etc/passwd (colonne 4).

5. Exercices d'application

   (a) Donnez la première ligne qui contient Alice du fichier /pub/ia/sys/shell/select.dat avec grep et head .

   (b) Donnez la première ligne qui contient Alice du fichier /pub/ia/sys/shell/select.dat avec sed et head .

   (c) Donnez la destination de la première ligne qui contient Alice du fichier /pub/ia/sys/shell/select.dat avec grep et head et cut.

   (d) Donnez la destination de la première ligne qui contient Alice du fichier /pub/ia/sys/shell/select.dat avec sed et head.

   (e) Donnez la destination de la dernière ligne qui contient Alice du fichier /pub/ia/sys/shell/select.dat avec ce que vous voulez.

***Exercice 14***

Quelques commandes pour déterminer la taille d'un fichier en octet. Seule la dernière est optimale et doit être utilisée dans la vrai vie. De plus elle est facilement extensible pour d'autres informations d'un fichier (UID, droits, ...).

L'objectif est de mettre dans la variable sz la taille du fichier /pub/ia/sys/shell/gen-projet.sh .

**wc -c**

1. Donnez la taille en octet du fichier /pub/ia/sys/shell/gen-projet.sh .
   ```
   sh> wc -c /pub/ia/sys/shell/gen-projet.sh
   ```
   La commande écrit le nom du fichier ce qui oblige à le supprimer avec une autre commande.
2. Donnez que la taille en octet du fichier /pub/ia/sys/shell/gen-projet.sh .
   ```
   sh> wc -c < /pub/ia/sys/shell/gen-projet.sh
   ```
3. D'où sz ⟸ taille de /pub/ia/sys/shell/gen-projet.sh :
   ```
   sh> sz=$(wc -c < /pub/ia/sys/shell/gen-projet.sh )
   sh> echo sz=$sz
   sh>
   ```

"**wc -c**"doit être bannie pour les fichiers réguliers car elle les lit et elle est donc réservée aux flux FIFO où il n'y a pas d'autre moyen de connaitre la taille.

**ls -l et cut**

1. Donnez le numéro de la colonne dans laquelle ls affiche la taille.
   ```
   sh> ls -l /pub/ia/sys/shell/gen-projet.sh
   ```
2. Réduisons le nombre de colonne (voir man de ls). Donnez le numéro de la colonne dans la quelle ls affiche la taille.
   ```
   sh> ls -l -g -G /pub/ia/sys/shell/gen-projet.sh
   ```
3. Quel est le séparateur de colonne?
   ```
   sh> ls -lgG /pub/ia/sys/shell/gen-projet.sh ¦ od -c
   ```
4. D'où sz ⟸ taille de /pub/ia/sys/shell/gen-projet.sh avec ls et cut
   ```
   sh> sz=$(ls -lgG /pub/ia/sys/shell/gen-projet.sh ¦
   cut à compléter)
   sh> echo sz=$sz
   sh>
   ```

**ls -l et sed** Il faut récupérer la $3^{ième}$ colonne de la sortie de ls, les colonnes étant séparées par des espaces.

```
sh> sz=$(ls -lgG /pub/ia/sys/shell/gen-projet.sh ¦ sed
-e "s/[^ ]* [^ ]* \([^ ]*\).*/\1/")
sh> echo sz=$sz
sh>
```

**ls -l et read** exemple à voir après le cours page 22.

```
sh> sz=$(ls -lgG /pub/ia/sys/shell/gen-projet.sh ¦ {
read a b c d ; echo $c ; })
sh> echo sz=$sz
sh>
```

**stat -c %s** Affiche que la taille (%s), %U affiche l'utilisateur, ....

```
sh> sz=$(stat -c %s /pub/ia/sys/shell/gen-projet.sh )
sh> echo sz=$sz
sh>
```

La méthode la plus efficace pour les fichiers réguliers.

***Exercice 15***

La commande find permet de parcourir des arborescences, de sélectionner des fichiers en combinant de nombreux critères et d'exécuter une action sur chacun de ces fichiers.

1. Affichez tous les fichiers de l'arborescence HOME.

```
sh> find $HOME -print
```

   ou

```
sh> ( cd ; find . -print )
```

   La seconde forme qui donne des chemins relatifs est en générale plus utile.

2. Affichez tous les fichiers réguliers de l'arborescence CWD.

```
sh> find . -type f -print
```

3. Affichez tous les répertoires de l'arborescence CWD.

```
sh> find . -type d -print
```

4. Affichez tous les fichiers réguliers de l'arborescence CWD d'une taille supérieure à 10k octets.

```
sh> find . -type f -size +10k -print
```

5. Affichez tous les fichiers réguliers de l'arborescence CWD d'une taille inférieure à 100 octets.

```
sh> find . -type f -size -100b -print
```

6. Affichez tous les fichiers réguliers de l'arborescence CWD dont les noms se terminent par ".c" ou ".h".

```
sh> find . -type f -name "*.[hc]" -print
```

   Le paramètre de l'option "-name" est une expression régulière qui porte sur le nom de base du fichier. L'option "-iname" est similaire mais est insensible à la casse (minuscule ou majuscule).

7. Recherchez parmi les fichiers .c de l'arborescence /pub/ia/sys/shell/projet, les fichiers et leurs lignes qui contiennent une occurrence de strcmp.

```
sh> find /pub/ia/sys/shell/projet -type f -name "*.c"
-exec grep -w -H -e strcmp {} \;
```

L'option "-exec" exécute la commande "**grep -w -H -e strcmp {}**" en remplaçant "{}" par le nom du fichier sélectionné. Si find sélectionne 10 fichiers, grep sera appelé 10 fois (une fois pour chacun des fichiers).
Le '; ' indique la fin de l'option -exec, on peut ajouter d'autres options après. Il est échappé par un \ pour pas que le Shell l'interprète comme la fin de la commande find.

***Exercice 16***

1. Quel est le nombre de fichiers .c ou .h dans l'arborescence /pub/ia/sys/shell/projet ?

2. Combien de lignes de C font l'ensemble de ces fichiers?

3. Combien de fois est utilisée la fonction atoi dans ces fichiers?

4. Faites une copie de l'arborescence /pub/ia/sys/shell/projet dans /tmp/projet.

```
sh> rm -rf /tmp/projet
sh> cp -r /pub/ia/sys/shell/projet /tmp/projet
sh>
```

   (a) Faites un backup .c.org et .h.org de tous les fichiers .c et .h de l'arborescence /tmp/projet.

   (b) Vérifiez qu'il y a autant de fichiers .org que de fichiers .c .h.

   (c) Changez dans tous les .c et les .h de l'arborescence /tmp/projet les occurrences de dupont par Dupont et de jean par Jean.

   aide

   La commande **sed -i -e "s/mot1/mot2/g" file** change dans le fichier file toutes les occurrences de mot1 par mot2.
   On peut mettre plusieurs options -e.

   (d) Recherchez les fichiers de l'arborescence qui contiennent jean ou dupont.

```
sh> find /tmp/projet -exec grep -w -l -e jean -e dupont {} \;
```

   Il ne doit pas y avoir de fichiers .h ni de fichiers .c.

   (e) Oups! ca bug pour les Jean! Restaurez les .c et les .h qui contiennent Jean du répertoire /tmp/projet à partir des .org.

   aide

   a) Faites un find qui lance grep en mode silence (option -q) et se termine par -print. Le -print est exécuté si tous les prédicats précédents sont vrais, en particulier le grep. Le -print est un prédicat toujours vrai.
   b) Ajoutez à la commande un nouveau -exec qui lance cp
   c) Enlevez l'option -print pour que la commande travaille silencieusement.

   (f) Recherchez les fichiers de l'arborescence qui contiennent Jean.

```
sh> find /tmp/projet -exec grep -w -l -e jean {} \;
```

   Vous devez trouver ∅fichier.

   (g) Recherchez les fichiers de l'arborescence qui contiennent Dupont.

```
sh> find /tmp/projet -exec grep -w -l -e Dupont {} \;
```

   Vous devez trouver liste.h et liste.c seulement.

5. Recopiez l'arborescence /pub/ia/sys/shell/projet dans /tmp/proj mais seulement les répertoires, les .c et les .h.

   aide

   - Reconstruisez l'arborescence des répertoires uniquement (find et mkdir).
   - Copiez les fichiers .c et .h (find et cp).

***Exercice 17***

La commande tar est à Unix ce qu'est winzip à Windows. Elle permet de créer des archives d'arborescences et de les restaurer.

La commande ci-dessous crée l'archive file à partir des fichiers $path_i$. Ceux ci peuvent être des répertoires ou des fichiers de n'importe quel type.

**tar -cf file $path_1$ $path_2$ $path_3$ ...**

La commande ci-dessous restaure tous les fichiers de l'archive file dans le CWD.

**tar -xf file**

La commande ci-dessous restaure les fichiers $path_i$ de l'archive file dans le CWD.

**tar -xf file $path_1$ $path_2$ ...**

De plus tar a l'option "-C dir" qui utilise le répertoire dir comme CWD pour les $path_i$. Il a aussi l'option -t qui est similaire à l'option -x ci-dessus mais qui ne fait qu'afficher les noms des fichiers de l'archive. Enfin tar lit ou écrit l'archive sur les flux stdin ou stdout si le nom de l'archive est -.

1. Créez une archive du répertoire /pub/ia/sys/shell/projet dans /tmp/projet.tar.

```
sh> ( cd $(dirname /pub/ia/sys/shell/projet ) ; tar -cf
/tmp/projet.tar projet )
```

   ou

```
sh> tar -C $(dirname /pub/ia/sys/shell/projet ) -cf
/tmp/projet.tar projet
```

2. Visualisez les noms des fichiers de l'archive.

```
sh> tar -tf /tmp/projet.tar
```

   ou en verbeux

```
sh> tar -tvf /tmp/projet.tar
```

3. Restaurez l'archive dans /tmp.

```
sh> tar -C /tmp -xf /tmp/projet.tar
```

4. Détruisez l'arborescence /tmp/projet et l'archive /tmp/projet.tar.

5. Recopier l'arborescence /pub/ia/sys/shell/projet dans /tmp mais sans utiliser de fichier temporaire (on rappelle que comme beaucoup de commande Unix, tar interprète le path "-" comme le flux stdin ou stdout).

6. Détruisez l'arborescence /tmp/projet et l'archive /tmp/projet.tar.

7. Dupliquez dans /tmp sans utiliser de fichier temporaire, l'arborescence /pub/ia/sys/shell/projet mais seulement les fichiers .c et les .h. Les répertoires dont l'arborescence ne contient ni .c ni .h ne sont pas recopiés. (combinez tar et find).

***Exercice 18***

1. Compilez le fichier /pub/ia/sys/shell/projet/src/util/bug.c et déterminez la $1^{ère}$ erreur.

   aide | On paginera avec less les flux stdout et stderr de gcc

2. Copiez le fichier /pub/ia/sys/shell/projet/src/inclue/data.h dans votre HOME.

3. Y-a-t-il dans l'arborescence /pub/ia/sys/shell/projet un fichier de taille nulle?

4. Quelle est la taille du plus grand fichier .c de l'arborescence /pub/ia/sys/shell/projet?

   aide | commandes: find stat sort uniq head

5. Quels sont les noms des fichiers .c de la taille trouvée à la question précédente?

6. Quels sont les utilisateurs qui ont des fichiers réguliers dans /pub/ia/?

   aide | commandes: find stat sort et uniq

7. Combien de fichiers réguliers dans /pub/ia appartiennent à l'utilisateur rrioboo?

8. Quelle est la taille cumulée en octets des fichiers de la question précédente?

9. Déterminez les fichiers .c de l'arborescence /pub/ia/sys/shell/projet qui incluent le fichier table.h directement.

10. Créez une archive /tmp/burger.tar qui contient les fichiers réguliers de l'arborescence /pub/ia/sys/shell/projet écrits par Patrice Burger (On sélectionnera les fichiers contenant la chaine "Patrice Burger").

11. Affichez le fichier /pub/ia/sys/shell/select.dat. Réaffichez le en écrivant la date sous le format "aaaa/mm/jj".

12. Affichez le fichier /pub/ia/sys/shell/select.dat en le triant par date croissante.

    aide | commandes: question précédente puis sort colonne 2 puis question precedente à l'envers

## 9.2 Shell (script)

Dans ces exercices, il est supposé:

- que vous avez un répertoire ~/bin dans votre HOME,
- que votre PATH est à jour pour permettre de lancer directement les exécutables présents dans ce répertoire ~/bin.

Comment créer ce répertoire et mettre à jour le PATH sont expliqués dans les premières questions du premier exercice.

De plus, sur votre écran, il ne doit n'y avoir qu'un éditeur et un terminal xterm visible simultanément.
Le terminal doit être de taille raisonnable (~ 25 lignes, ~ 100 colonnes) et si vous voulez de l'aide avec une police lisible à un mètre (<CTL-+CLIC-DROIT> et choisir Huge).
L'éditeur doit être de taille raisonnable (~ 35 lignes, ~ 80 colonnes), avec une police lisible à un mètre, et afficher en début de chaque ligne, de le numéro de ligne.

### *Exercice 1*

1. Créez le répertoire bin dans votre HOME, copiez dans ce répertoire le fichier "/pub/ia/sys/shell/punition1.sh ". C'est le script vu en cours. La commande "**sh> punition1.sh n word**"écrit n lignes de word sur le flux stdout.

```
sh> mkdir -p ~/bin
```

```
sh> cp /pub/ia/sys/shell/punition1.sh  ~/bin
```

2. Configurez votre Shell et ce script pour qu'il soit exécutable par le PATH. La commande "**sh> ( cd /tmp ; punition1.sh 3 essai )** "doit fonctionner.

```
sh> PATH="$PATH:~/bin"
```

```
sh> chmod 700 ~/bin/punition1.sh
```

3. Écrivez le script ~/bin/punition2.sh. Ses arguments sont n, m et word et il écrit n lignes contenant m fois word séparés par un blanc. On partira de punition1.sh et on fera 2 boucles imbriquées.

4. Écrivez le script ~/bin/punition3.sh. Il est similaire à punition2.sh mais il n'a qu'une seule boucle et il appelle la commande punition1.sh.

aide

```
1 str= ; i=0
2 while [ i -lt $m ] ; do
3   str="$str $w"
4   i=$[i+1]
5 done
6 punition1.sh "$n" "$str"
```

5. Écrivez le script ~/bin/punition.sh. Il est similaire à punition2.sh mais:
   - avec 0 argument il écrit 10 lignes contenant 3 fois "je ne bavarde pas en cours".
   - avec 1 argument (**punition.sh word**) il écrit 10 lignes contenant 3 fois l'argument word.
   - avec 2 arguments (**punition.sh m word**) il écrit 10 lignes contenant m fois l'argument word.
   - avec 3 arguments (**punition.sh n m word**) il écrit n lignes contenant m fois l'argument word.

   On l'implémentera sans boucle et en utilisant la commande punition3.sh.

aide

```
1 ...
2 if test $# = 0 ; then
3   n=10; m=3; w="je ne bavarde pas en cours"
4 elif test $# = 1 ; then
5 ...
6 punition3.sh "$n" "$m" "$w"
```

***Exercice 2***

L'exécutable /pub/ia/sys/shell/iacmp/iacmp compare les chaines de caractères passées en argument et les écrits sur le flux stdout triées par ordre alphabétique. Pour démarrer, il a besoin des variables d'environnement

**IacmpDir** positionnée à /pub/ia/sys/shell/iacmp,

**LD_LIBRARY_PATH** positionnée à /pub/ia/sys/shell/iacmp/lib.

1. Lancez le sans modifiez votre environnement avec les arguments "chat roux", "chat blanc", "chat noir".

```
sh> LD_LIBRARY_PATH=/pub/ia/sys/shell/iacmp/lib
IacmpDir=/pub/ia/sys/shell/iacmp /pub/ia/sys/shell/iacmp/iacmp "chat roux" "chat blanc" "chat noir"
```

2. Ouvrez un terminal et configurez votre environnement pour pouvoir lancer /pub/ia/sys/shell/iacmp/iacmp sans les affectations des variables d'environnement.

```
sh> export LD_LIBRARY_PATH=/pub/ia/sys/shell/iacmp/lib
```

```
sh> export IacmpDir=/pub/ia/sys/shell/iacmp
```

```
sh> /pub/ia/sys/shell/iacmp/iacmp "chat roux" "chat blanc" "chat noir"
chat blanc
chat noir
chat roux
sh>
```

Fermez ce terminal.

3. Écrivez le script ~/bin/iacmp qui est un wrapper de /pub/ia/sys/shell/iacmp/iacmp . Il configure l'environnement de /pub/ia/sys/shell/iacmp/iacmp puis le lance avec exec en lui transmettant ses arguments. Ainsi la commande ci-dessous doit fonctionner quelque soit le CWD.

```
1 sh> iacmp "chat roux" "chat blanc" "chat noir"
2 chat blanc
3 chat noir
4 chat roux
5 sh>
```

### *Exercice 3*

L'objectif de cet exercice est d'écrire une commande "**sortarg** $s_1$ $s_2$ ... $s_n$"qui réécrit les chaines de caractères $s_i$ triées par ordre croissant sur une ligne.

**Méthode 1** Écrivez le script ~/bin/sortarg1, son algorithme est:

1. Écrire les arguments 1 par ligne dans le fichier /tmp/1.

   | aide \| commande: echo |
   |---|

2. Trier le fichier /tmp/1 dans /tmp/2.

   | aide \| commande: sort |
   |---|

3. Écrire sur le flux stdout, le fichier /tmp/2 sans saut de ligne.

   | aide \| commandes: read et echo -n |
   |---|

Testez le avec la commande:

```
sh> sortarg1 "chat roux" "chat blanc" "chat noir"
chat blanc
chat noir
chat roux
sh>
```

**Méthode 2** Écrivez le script ~/bin/sortarg2, son algorithme est similaire à celui de sortarg1 mais il n'utilise pas de fichier intermédiaire.

| aide \| N'hésitez pas à piper les boucles |
|---|

**Méthode 3** Écrivez le script ~/bin/sortarg3, il trie les arguments de façon algorithmique. On supposera que les arguments $s_i$ n'ont pas d'espace[1].

| aide |
|---|
| tant que il y a des argument faire<br>trouver le plus petit argument et l'écrire<br>reconstituer l'ensemble d'argument moins celui écrit<br>fait |

[1] C'est possible aussi avec des espaces mais il faut utiliser les variables tableau.

***Exercice 4***

Réalisation de la boite à outils ~/bin/s3tool.sh.

1. Écrivez dans le fichiers ~/bin/s3tool.sh la builtin commande s36str données ci-dessous:

```
# usage: s36str a b c  => s36str a a b b c c
# usage: s36str k1 s1 k2 s2 k3 s3
# imprime les chaines s(i) en fonction de l'ordre
# des cles k(i). Les cles sont comparees de facon
# lexicographique.
function s36str() {
  if test $#-eq 3 ; then
    s36str "$1" "$1" "$2" "$2" "$3" "$3"
    return 0
  fi
  if   test ! "$1" \> "$3" -a ! "$3" \> "$5"; then
    echo $2 ; echo $4 ; echo $6
  elif test ! "$1" \> "$5" -a ! "$5" \> "$3"; then
    echo $2 ; echo $6 ; echo $4
  elif test ! "$3" \> "$1" -a ! "$1" \> "$5"; then
    echo $4 ; echo $2 ; echo $6
  elif test ! "$3" \> "$5" -a ! "$5" \> "$1"; then
    echo $4 ; echo $6 ; echo $2
  elif test ! "$5" \> "$1" -a ! "$1" \> "$3"; then
    echo $6 ; echo $2 ; echo $4
  else
    echo $6 ; echo $4 ; echo $2
  fi
}
```

2. Dans un nouveau terminal, ajoutez s36str aux builtin commandes de son Shell, puis testez la.

```
sh> s36str cc 33 bb 22 aa 11
bash: s36str: command not found
sh> source ~/bin/s3tool.sh
sh>
```

puis testez la.

```
sh> s36str cc 33 bb 22 aa 11
11
22
33
sh> s36str cc bb aa
aa
bb
cc
sh>
```

3. Ajoutez à s3tool.sh, la builtin commande s36int qui est similaire à s36str mais elle compare les clés numériquement. Testez s36int dans le nouveau terminal

```
sh> s36int 01 2 03
01
2
03
sh>
```

4. Ajoutez à la boites à outils la builtin commande isInt ci-dessous:

```
# usage: isInt n
# fonction: renvoie 0 si n est un entier sinon 1
function isInt() {
  n="$1"
  m=$(echo "$n" | sed -e "1s/[+-]\?[0-9]\+//")
  ! test "$m"
}
```

Testez la dans le nouveau terminal.

```
sh> isInt -01 ; echo $?
0
sh> isInt a01b ; echo $?
1
sh>
```

5. Fermez le nouveau terminal.

### *Exercice 5*

Réalisation de quelques commandes en utilisant la boite à outils s3tool.sh de l'exercice précédent.

Les commandes demandées doivent "sourcer" la boite à outils: source ~/bin/s3tool.sh ou . ~/bin/s3tool.sh.

Elles doivent être écrites dans le répertoire bin de votre HOME. Elles font tous les contrôles nécessaires sur les arguments et si un problème advient, elles écrivent un message explicite sur le flux stderr et s'arrêtent en renvoyant un statut de 1.

1. Écrivez le script s3s qui a trois arguments chaines de caractères **s3s $str_1$ $str_2$ $str_3$** et qui les réécrit triées par ordre croissant. Le tester en outre avec les commandes:

```
sh> s3s "chat roux" "chat blanc" ;  echo $?
s3s:Fatal:  2 trop ou pas assez d'arguments (3)
1
sh> s3s "chat roux" "chat blanc" "chat noir" ;  echo $?
chat blanc
chat noir
chat roux
0
sh>
```

2. Écrivez le script s3e qui a trois arguments entier **s3e $n_1$ $n_2$ $n_3$** et qui les réécrit triés par ordre croissant. Le tester en outre avec les commandes:

```
sh> s3e "chat roux" "chat blanc" ;  echo $?
s3e:Fatal:  2 trop ou pas assez d'arguments (3)
1
sh> s3e 01 2 "chat noir" ;  echo $?
s3e:Fatal:  arg 'chat noir' n'est pas un entier
1
sh> s3e 01 2 03 ;  echo $?
01
2
03
0
sh>
```

3. Écrivez le script s3f dont les 3 arguments sont des chemins de fichiers réguliers **s3f $f_1$ $f_2$ $f_3$** et qui les réécrit triés par ordre croissant de taille. Le tester en outre avec les commandes:

```
sh> s3f /lib /dev /dev/sda ; echo $?
s3f::Fatal:  arg '/lib' n'est pas un fichier regulier
1
sh> cd /pub/ia/sys/shell/s3/petit
sh> s3f grand moyen truc ; echo $?
s3f:Fatal arg 'truc' n'est pas un fichier regulier
1
sh> s3f grand moyen petit ; echo $?
petit
grand
moyen
0
sh> cd -
sh>
```

4. Écrivez le script s3d dont les 3 arguments sont des chemins de répertoire **s3d $d_1$ $d_2$ $d_3$** et qui les réécrit triés par ordre croissant du nombre de fichiers qu'ils contiennent. On comptabilisera tous les fichiers. Le tester en outre avec les commandes:

```
sh> s3d /lib /dev /dev/sda ; echo $?
s3d:Fatal:  arg '/dev/sda' n'est pas un répertoire.
1
sh> s3d /lib /dev /var/log ; echo $?
s3d:Fatal:  arg '/var/log' n'est pas lisible.
1
sh> ( cd /pub/ia/sys/shell/s3  ;  s3d grand moyen petit ;
echo $?)
petit
moyen
grand
0
sh>
```

5. Écrivez le script s3f1m dont les 3 arguments sont des chemins de fichiers réguliers **s3f1m $f_1$ $f_2$ $f_3$** et qui les réécrit triés par ordre croissant du second mot de la $1^{ère}$ ligne du fichier (le séparateur de mot étant le blanc ' '). Le tester en outre avec les commandes:

```
sh> s3f1m /etc/group /etc/group /etc/shadow ;  echo $?)
s3f1m:Fatal:  arg '/etc/shadow' n'est pas lisible.
1
sh> ( cd /pub/ia/sys/shell/s3/petit  ;  s3f1m petit moyen
grand ;  echo $?)
grand
moyen
petit
0
sh>
```

### *Exercice 6*

Copiez le fichier /pub/ia/sys/shell/hello.c chez vous et ajoutez `#!cce` en première ligne et première colonne. Rendez hello.c exécutable. L'objectif de cet exercice est que la commande

**sh> ./hello.c 2**

compile le fichier C sans la première ligne et lance l'exécutable généré si il n'y a pas eu d'erreurs de compilation.

1. Écrivez le script "~/bin/cce", il a un seul argument qui est le chemin du fichier C. On créera l'exécutable /tmp/N.a.out où N est le PID du processus cce, on détruira cet exécutable après son exécution.

   aide

   - La commande "**gcc -x c -**" compile le flux stdin en supposant que c'est du C.
   - la commande "**sed -e 1d file**" écrit sur le flux stdout le fichier file sans sa première ligne.
   - L'expansion de "$$" est le PID du processus.

2. Vérifiez que le script fonctionne

   **sh> bash ~/bin/cce hello.c**

   **sh> cce hello.c**

   **sh> ./hello.c**

3. Faites que le script cce transmette les paramètres à l'exécutable et que le statut renvoyé soit:
   2 La compilation a échoué et le programme n'a pas été lancé.
   1 L'exécution du programme a renvoyé un statut différent de ∅.
   ∅ L'exécution du programme a renvoyé le statut ∅.

```
sh> ./hello.c 2 good ;  echo statut=$?
good
good
0
sh> ./hello.c 2 good x ;  echo statut=$?
./hello.c:  too many args.
1
sh>
```

aide | L'expansion de "$? " est le statut de la dernière commande.

4. Ajoutez au script les options suivantes:
   - Si la variable d'environnement SILENCE n'est pas vide, les messages du compilateur ne sont pas affichés.
   - Si la variable d'environnement DEBUG n'est pas vide, le code C est compilé avec l'option -g et l'exécution est lancée sous le débogueur gdb. Dans ce cas les arguments ne sont pas transmis.

***Exercice 7***

L'objectif de cet exercice est de réaliser le script shzip qui crée des archives auto-décompressables. On crée l'archive archive.shz qui contient l'arborescence du répertoire source dir-src par:

```
sh> shzip archive.shz dir-src
sh>
```

On restaure quelque part l'archive par:

```
sh> mkdir dir-dest
sh> ./archive.shz dir-dest
sh>
```

Après ces commandes, le répertoire destination dir-dest contient les mêmes fichiers que le répertoire source dir-src.

1. base64 est un format qui permet de encoder/décoder avec 64 caractères imprimables n'importe quelle suite d'octets, il est très utilisé dans l'internet.
   La commande base64 est la commande Unix standard pour gérer ce format.
   - Créez le fichier "un" qui contient une ligne avec 3 caractères 'a'.
     ```
     sh> echo aaa > un
     ```
   - Encodez le fichier "un" sur le flux stdout.
     ```
     sh> base64 un
     ```
   - Encodez le fichier "un" dans le fichier /tmp/un.64
     ```
     sh> base64 un > /tmp/un.64
     ```
   - Décodez le fichier /tmp/un.64 sur le flux stdout.
     ```
     sh> base64 -d /tmp/un.64
     ```
   - Décodez le fichier /tmp/un.64 dans le fichier /tmp/un.
     ```
     sh> rm -f /tmp/un
     sh> base64 -d /tmp/un.64 > /tmp/un
     sh> cat /tmp/un
     sh>
     ```
   - Décodez le flux stdin dans le flux stdout.
     ```
     sh> cat /tmp/un.64 | base64 -d
     ```

2. Écrivez le début du script shzip qui vérifie les arguments et qui écrit "$0:DEBUG: arguments ok" si les arguments sont correctes et des messages d'erreur dans le cas contraire.

3. Ajoutez au script la builtin commande "**encodefile file**" qui écrit sur stdout le code pour générer le fichier file à partir de son codage en base64. Pour le fichier "un" précédent, le code ci-dessous doit être généré.

   ```
   base64 > un «EOF
   YWFhCg==
   EOF
   chmod 644 un
   ```

   En bas du script, appelez la commande sur les fichiers un et deux "**encodefile un**" et "**encodefile deux**" puis testez votre script:

   ```
   sh> echo bbbb >  deux
   sh> echo bbbb >>  deux
   sh> shzip ~/bin/archive.shz .
   sh> cat ~/bin/archive.shz # vérif.  à l'oeil
   sh> ( cd /tmp ;  rm -f un* deux* ;  bash ~/bin/archive.shz )
   sh> cat /tmp/un
   aaa
   sh> cat /tmp/deux
   bbbb
   bbbb
   sh>
   ```

4. Ajoutez au script la builtin commande "**encodedir dir** qui écrit dans l'archive le code pour générer tous ses fichiers réguliers. Les répertoires et les autres fichiers seront ignorés avec un message d'erreur de niveau "warning".

   En bas du script,

   - enlevez les commandes "**encodefile un**" et "**encodefile deux**"
   - donnez un chemin absolu à l'archive.

     aide

     ```
     1 case "$chemin" in
     2   /*) # chemin absolu
     3   *)  # chemin relatif
     4 esac
     ```

   - appelez encodedir en vous placant dans le répertoire source.

aide

```
cd repertoire-source
encodedir .
```

Testez votre script:

```
sh> mkdir 123
sh> mv un deux 123
sh> cat 123/un 123/deux >  123/trois
sh> shzip ~/bin/archive.shz 123
sh> cat ~/bin/archive.shz # vérif.  à l'oeil
sh> rm -rf /tmp/tt
sh> mkdir /tmp/tt
sh> ( cd /tmp/tt ;  bash~/bin/archive.shz )
sh> cat /tmp/tt/un
aaa
sh> cat /tmp/tt/deux
bbbb
bbbb
sh> cat /tmp/tt/trois
aaa
bbbb
bbbb
sh>
```

5. Modifiez la builtin commande "**encodedir dir** pour qu'elle traite aussi les répertoires (attention aux répertoires "." et "..").

   Testez votre script:

```
sh> mkdir 123/123
sh> cat 123/un 123/un 123/123/un-bis
sh> cat 123/deux 123/deux 123/123/deux-bis
sh> shzip ~/bin/archive.shz 123
sh> cat ~/bin/archive.shz # vérif.  à l'oeil
sh> rm -rf /tmp/tt
sh> mkdir /tmp/tt
sh> ( cd /tmp/tt ;  bash~/bin/archive.shz )
sh> cat /tmp/tt/un
aaa
sh> cat /tmp/tt/deux
bbbb
bbbb
sh> cat /tmp/tt/trois
aaa
bbbb
bbbb
sh> cat /tmp/tt/123/deux
bbbb
bbbb
bbbb
bbbb
sh>
```

6. Modifiez le script shzip pour ajoutez l'entête "#!/bin/bash" à l'archive générée et la rendre exécutable. Testez votre script:

```
sh> cat ~/bin/archive.shz # vérif.  à l'oeil
sh> rm -rf /tmp/tt
sh> mkdir /tmp/tt
sh> shzip ~/bin/archive.shz 123
sh> ( cd /tmp/tt ;  archive.shz )
sh>
```

7. Ajoutez au script shzip l'option facultative -z qui gzip les fichiers avant de les encoder.

### *Exercice 8*

L'objectif de cet exercice est de réaliser le script "**pidfiles dir**"qui affiche la liste des utilisateurs ayant des fichiers réguliers dans le répertoire dir avec le nombre de fichiers.

1. Écrivez le script ∼/bin/pidfiles, après les vérifications portant sur son argument, son algorithme est:

   (a) Cherchez les utilisateurs ayant des fichiers réguliers dans le répertoire dir.

   > aide | commandes: find et stat

   (b) Éliminez les doublons d'utilisateur.

   > aide | commandes: sort et uniq

   (c) Pour chaque utilisateur cherchez le nombre de fichiers.

   > aide | commandes: find et wc -l

   Testez le avec la commande:

   ```
   sh> pidfiles /pub/ia/
   iauge 200
   gberthelot 300
   ...
   sh>
   ```

2. Ajoutez au script l'option -s qui en plus du nombre de fichiers, indique aussi en octet, la taille cumulée de ces fichiers.

### *Exercice 9*

Écrivez le script "**sleepsort** $\mathbf{n_1}$ $\mathbf{n_2}$ ... $\mathbf{n_n}$"qui trie les entiers $n_i$ positifs par ordre croissant.

Son algorithme est de lancer en arrière plan pour chaque $n_i$ la séquence de commandes "**sleep** $\mathbf{n_i}$ **;** **echo** $\mathbf{n_i}$".

Pour un affichage, plus joli, vous pouvez, après avoir lancé les n "sleep", ajouter n commandes wait.

- Quel est son temps d'exécution?
- Quel est son ordre théorique en nombre d'instructions?
- Quel est son ordre en nombre d'instructions sur une machine Unix?

> aide | Pour réveiller les processus endormis, le noyau les insère dans une liste classée par ordre de réveil.

## 9.3 Flux

Pour ces exercices, vous n'avez besoin sur votre écran que d'un éditeur et un terminal xterm visible simultanément.
Le terminal doit être de taille raisonnable (∼ 25 lignes, ∼ 100 colonnes) et si vous voulez de l'aide avec une police lisible à un mètre (<CTL-+CLIC-DROIT> et choisir Huge).
L'éditeur doit être de taille raisonnable (∼ 35 lignes, ∼ 80 colonnes), avec une police lisible à un mètre, et afficher en début de chaque ligne le numéro de ligne.

Enfin ce document est accessible en ligne: /pub/ia/sys/syscall/sys-poly2.pdf

### *Exercice 1*

Écrivez le programme suplastbyte.c:

- Il a un seul argument appelé f.
- f est un chemin de fichier régulier.
- Il modifie f en lui enlevant le dernier caractère.

Implémentez ce programme en utilisant les flux noyau.

aide

Les seules opérations sur un fichier sont lire et écrire, Il n'y a aucun moyen d'enlever un ou plusieurs octets à un fichier, l'algoritme le plus simple est:

1. Ouvrir le fichier en lecture seule.
2. Calculer la taille t du fichier (lseek ou fstat)
3. Allouer un tampon de taille t-1 et lire d'un coup les t-1 premiers octets.
4. Fermer le fichier et le réouvrir en écriture seule plus troncature.
5. Écrire le tampon d'un seul coup.
6. Fermer le fichier et libérer le tampon.

Note: Sous un système Unix, le dernier point est fait par le système si vous l'omettez.

### *Exercice 2*

Écrivez le programme suplastbyte-libc.c qui est fonctionnellement identique à suplastbyte.c (exercice précédent), mais qui est implémenté en utilisant les flux libc.

### *Exercice 3*

1. Que fait la fonction lire du fichier /pub/ia/sys/syscall/lire.h ? Précisez particulièrement à quoi correspondent les 0, 1 utilisés dans les read et les write.

2. Écrivez le programme C /pub/ia/sys/syscall/lire.c qui 1) écrit "je suis la cmd lire" sur le fichier standard d'erreur, 2) appelle la fonction lire.
   - Générez son exécutable lire.
     ```
     sh> gcc -o lire lire.c
     sh>
     ```
   - Testez la.
     ```
     sh> ./lire < /pub/ia/sys/syscall/data.in
     je suis la cmd lire
     lire():  nbCarLus = 10
     aaaaaaaaa
     sh>
     ```

   aide

   ```
   1 #include "lire.h"
   2 int main()
   3 {
   4     fprintf(stderr,"je suis la cmd lire\n");
   5     lire();
   6     return 0;
   7 }
   ```

3. Écrivez le programme lire3exec.c qui appelle la fonction lire 2 fois, puis exécute la commande lire précédente (appel système execlp).

4. Testez lire3exec.c en redirigeant le fichier standard d'entrée sur /pub/ia/sys/syscall/data.in et le fichier standard de sortie sur data.out:
   ```
   sh> ./lire3exec < /pub/ia/sys/syscall/data.in  >  data.out ;
   cat data.out
   lire():  nbCarLus = 10
   lire():  nbCarLus = 10
   je suis la cmd lire
   lire():  nbCarLus = 10
   aaaaaaaaa
   bbbbbbbbb
   cccccccccc
   sh>
   ```

   Comment sont transmis les flux lors de l'appel execlp?

5. Écrivez le programme lire3a.c similaire à lire3exec.c mais qui redirige:
   - le read(0,...) de la fonction lire sur le fichier argv[1] (le $1^{er}$ argument de la commande),
   - le write(1,...) de la fonction lire sur le fichier argv[2] (le $2^{nd}$ argument de la commande).

   aide

   - Copiez lire3exec.c dans lire3a.c. **sh> cp lire3exec.c lire3a.c**
   - Avant le premier lire, ouvrez le fichier argv[1] en lecture, ouvrez le fichier argv[2] en écriture + troncature + création si besoin.
   - Puis faites que le flux 0 soit le fichier argv[1] (dup2), le flux 1 soit le fichier argv[2] (dup2),

   Compilez le, puis faites les tests ci-dessous.
   ```
   sh> ./lire3a
   ./lire3a:  usage:  ./lire3a infile outfile
   sh> ./lire3a /pub/ia/sys/syscall/data.in  /ho/la/la
   ./lire3a:  can not open /ho/la/la for writing :  No such file
   sh> ./lire3a /ho/la/la data.out
   ./lire3a:  can not open /ho/la/la for reading :  No such file
   sh> ./lire3a /pub/ia/sys/syscall/data.in data.out ;  cat
   data.out
   lire():  nbCarLus = 10
   lire():  nbCarLus = 10
   je suis la cmd lire
   lire():  nbCarLus = 10
   aaaaaaaaa
   bbbbbbbbb
   cccccccccc
   sh>
   ```

6. Écrivez le programme lire3b.c similaire à lire3a.c mais qui:
   - laisse les read et write du $1^{er}$ appel à lire sur les fichiers originaux.
   - redirige les read et write des autres appels à lire sur les fichiers argv[1] et argv[2].

aide

- Copiez lire3a.c dans lire3b.c. **sh> cp lire3a.c lire3b.c**
- Déplacez les dup2 après le premier appel à lire().

Compilez le, puis faites le test ci-dessous.

```
sh> echo 111111111 ¦  ./lire3b /pub/ia/sys/syscall/data.in
data.out
lire():  nbCarLus = 10
111111111
lire():  nbCarLus = 10
je suis la cmd lire
lire():  nbCarLus = 10
sh> cat data.out
aaaaaaaaa
bbbbbbbbb
sh>
```

7. Écrivez le programme lire3c.c similaire à lire3b.c mais qui:
   - laisse les read et write du $1^{er}$ appel à lire sur les fichiers originaux.
   - redirige les read et write du $2^{nd}$ appel à lire sur les fichiers argv[1] et argv[2].
   - laisse les read et write du $3^{ième}$ appel à lire sur les fichiers originaux.

aide

- Copiez lire3b.c dans lire3c.c. **sh> cp lire3b.c lire3c.c**
- Avant les dup2 sauvegardez les flux 0 et 1 (dup)
- Avant l'execlp restaurez les flux 0 et 1 (dup2).

Compilez le, puis faites le test ci-dessous.

```
sh> { echo 111111111 ; echo 222222222 ; } ¦  \
     ./lire3b /pub/ia/sys/syscall/data.in  data.out
lire():  nbCarLus = 10
111111111
lire():  nbCarLus = 10
je suis la cmd lire
lire():  nbCarLus = 10
222222222
sh> cat data.out
aaaaaaaaa
sh>
```

***Exercice 4***

L'exécutable /pub/ia/sys/shell/iacmp/iacmp compare les chaines de caractères passées en argument et les écrit sur le flux standard de sortie triées par ordre alphabétique. Pour démarrer, il a besoin des variables d'environnement:

| nom de variable | positionnée à |
|---|---|
| IacmpDir | /pub/ia/sys/shell/iacmp |
| LD_LIBRARY_PATH | /pub/ia/sys/shell/iacmp/lib |

1. Complétez le programme /pub/ia/sys/syscall/iacmp.c qui est un wrapper de /pub/ia/sys/shell/iacmp/iacmp. Il configure l'environnement pour /pub/ia/sys/shell/iacmp/iacmp puis le lance avec exec en lui transmettant ses arguments.

2. Une fois compilé, testez le avec la commande ci-dessous. Elle doit fonctionner quelque soit le nombre d'arguments.

```
1 sh> ./a.out "chat roux" "chat blanc" "chat noir"
2 chat blanc
3 chat noir
4 chat rouR
5 sh>
```

## *Exercice 5*

1. Écrivez dans le fichier readpasswd.c la fonction

```
1  // renvoie n>=16: OK/ERR (password tronqué)
2  //       n taille réelle du password entré,
3  //       pwd contient les 15 1ers chars du password.
4  //       pwd[15]=0
5  // renvoie 0<=n<16 : OK
6  //       pwd contient le password (pwd[n]=0)
7  // renvoie -1    : ERR
8  //       pas de fichier terminal, ....
9  int readpasswd(char pwd[16])
10 {
11 }
```

qui lit un mot de passe d'au plus 15 caractères sur le flux correspondant au terminal de son groupe terminal.

aide

- Rechercher le nom du fichier du terminal grâce à ttyname():

```
1  char* ttypath =
2       ttyname(0) ?  ttyname(0)
3     :  ttyname(1) ?  ttyname(1)
4     :  ttyname(2);
5  if ( ttypath==0 ) return -1;
```

- Ouvrir en lecture/écriture le fichier ttypath.
- Mettre le mode de ttypath en raw et sans écho:
  `system("stty raw -echo");`
  Il n'y a plus d'écho des caractères tapés et ils arrivent sans attendre le '\n'. Tous les <CTL-?>, le caractère d'effacement, ... sont désactivés et sont lus tels quels.
  Vous pouvez essayer cette commande dans un nouveau terminal. Il faut alors taper les commandes en aveugle, et vous pouvez restaurer son mode avec la commande "stty -raw echo".
- Écrire la chaine de caractère "passwd: ".

aide

- Boucle infinie
  - Lire un caractère c sur le flux ttypath
  - si le c est un \n ou un \r $\Longrightarrow$ fin de boucle.
  - Écrire un X.
  - si pwd n'est pas plein écrire c dans pwd à sa place.
- Écrire la chaine de caractère "\r\n".
- Restaurer le mode de ttypath:
  `system("stty -raw echo");`
- Fermer le fichier ttypath.
- Renvoyer la taille du mot de passe entré.

Notes: l'utilisation de system et stty n'est pas optimale et ne garantit pas que l'on restaure parfaitement le mode du terminal. Il faudrait mieux utiliser les "appels système" tcgetattr, tcsetattr, mais c'est un peu plus compliqué.

2. Faites un petit programme principal pour tester votre implémentation. "**sh> echo mon-pwd | ./a.out**" permet il d'entrer le mot de passe "mon-pwd"?

3. Modifiez readpasswd pour gérer le caractère backspace.

aide

Au niveau de l'affichage, sur un backspace (caractère '\b') si on est au début, on l'ignore, sinon on écrit "\b \b".

***Exercice 6***

1. Complétez le programme /pub/ia/sys/syscall/factoriel.c qui si il s'appelle 6, calcule factoriel 6 (6!), si il s'appelle 7, il calcule factoriel 7 (7!) et ainsi de suite. Son fonctionnement est le suivant:

   **lancé avec 1 argument** (le nom de la commande)
   - Il vérifie que le nom de base de cet argument est un nombre.
   - Il initialise la première variable d'environnement à ce nombre.
   - Il initialise la seconde variable d'environnement au nombre 1.
   - Il initialise la troisième variable d'environnement à l'argument.
   - Il exécute le programme sans aucun argument et avec un environnement contenant les 3 variables.

   **lancé avec 0 argument**
   - Il extrait n, fac et cmd des 3 premières variables d'environnement.
   - Si n est égal à 1, il écrit fac et s'arrête.
   - Si n n'est pas égal à 1, il positionne les 3 premières variables d'environnement à n-1, n*fac et cmd.
   - Puis il exécute le programme sans aucun argument et avec un environnement contenant les 3 variables.

   **lancé avec plus que un argument** il s'arrête.

2. Une fois compilé, testez le en calculant factoriel 5 et 6.

3. Ce programme respecte il les conventions d'Unix pour les paramètres?

4. Ce programme respecte il les conventions d'Unix pour les variables d'environnement?

5. Lors du calcul de factoriel 6, combien de processus ont été créés?

6. Lors du calcul de factoriel 6, combien de programmes ont été exécutés?

7. Lors du calcul de factoriel 6, combien de configurations de MMU ont été créées?

## 9.4 Communication inter-processus

Pour ces exercices, vous n'avez besoin sur votre écran que d'un éditeur et un terminal xterm visible simultanément.
Le terminal doit être de taille raisonnable (∼ 25 lignes, ∼ 100 colonnes) et si vous voulez de l'aide avec une police lisible à un mètre (<CTL-+CLIC-DROIT> et choisir Huge).
L'éditeur doit être de taille raisonnable (∼ 35 lignes, ∼ 80 colonnes), avec une police lisible à un mètre, et afficher en début de chaque ligne le numéro de ligne.

Enfin ce document est accessible en ligne: /pub/ia/sys/syscall/sys-poly2.pdf

### *Exercice 1*

Cet exercice porte sur les programmes sigsend et sigcatch dont les sources sont disponibles en /pub/ia/sys/syscall/sigsend.c et /pub/ia/sys/syscall/sigcatch.c .

1. Donnez l'algorithme du main de sigsend, complétez le et compilez le.
   ```
   sh> gcc -o sigsend sigsend.c
   ```
2. Donnez l'algorithme de sigcatch.

   **main**

   **lignes 27 à 32** Attache le ..... à tous les signaux de .... à .....
   **lignes 29 à 30** Affiche un message d'erreur pour les signaux qui ne sont pas .....
   **lignes 36 à 39** Boucle infinie: affichage d'un ....., attente d'un .....

   **Gestionnaire de signal (fonction callback)**

   **lignes 15 à 19**

   Compilez le.
   ```
   sh> gcc -o sigcatch sigcatch.c
   ```
3. Lancez 2 processus sigcatch dans des terminaux différents.
   ```
   sh> xterm -e ./sigcatch &
   ```
   ```
   sh> xterm -e ./sigcatch &
   ```
   Dans quel état sont les 2 processus sigcatch?
4. Envoyez quelques signaux qu'ils attrapent à chacun deux.
   ```
   sh> ./sigsend hup <pid-1>
   ```
   ```
   sh> ./sigsend 10 <pid-2>
   ```
5. Tapez <CTL-C> dans les terminaux des processus sigcatch.
   Se sont-ils terminés?
   Que fait le noyau quand <CTL-C> est tapé dans un terminal?
6. Tapez <CTL-Z> dans les terminaux des processus sigcatch.
   Se sont-ils suspendus?
   Que fait le noyau quand <CTL-Z> est tapé dans un terminal?
7. Envoyez le signal SIGSTOP à un des processus sigcatch.
   ```
   sh> ./sigsend 19 <pid-1>
   ```
   - A-t-il été attrapé?
   - Envoyez lui le signal SIGUSR1. L'a-t-il attrapé?
   - Tapez <CTL-C> dans le terminal du processus. A-t-il attrapé le signal SIGINT?
   - Envoyez lui le signal SIGCONT. L'a-t-il attrapé? Qu'en est il des signaux SIGUSR1 et SIGINT précédents?

   Le signal SIGSTOP suspend le processus, il ne peut être réveillé que par la réception d'un signal non ..... ou non .....
8. Recommencez l'expérimentation précédente en envoyant plusieurs fois le signal SIGUSR1.

   Combien de réceptions du signal SIGUSR1 sont traitées?
9. Terminez les deux processus avec la commande sendsig.

***Exercice 2***

Cet exercice porte sur le programmes mycat dont les sources sont disponibles en /pub/ia/sys/syscall/mycat.c .

Il met en évidence la plus part des comportements des appels système read et write en fonction des types de fichier et qui sont résumés dans la table ci-dessous (en supposant les arguments valides):

| | mode | bloquant | retour | | SIGPIPE |
|---|---|---|---|---|---|
| | | | 0 | -1 | |
| read | regu. | jamais | fin fichier | non | jamais |
| write | regu. | jamais | jamais | df | jamais |
| read | fifo | V & $ne \neq 0$ | V & $ne = 0$ | jamais | jamais |
| write | fifo | P & $nl \neq 0$ | jamais | spa & $nl = 0$ | $nl = 0$ |

ne : nombre d'écrivains. V: FIFO vide.
nl : nombre de lecteurs. P: FIFO pleine.
df : disque plein ou dépassement de quota.
spa : signal SIGPIPE attrapé ou ignoré

1. Donnez l'algorithme de /pub/ia/sys/syscall/mycat.

   **lignes 24 et 13 à 18** Attache le ..... au signal SIGPIPE. Le gestionnaire affiche un message, attend .... secondes puis termine le processus avec un statut de 0.

   **lignes 26 à 39** boucle infinie.

   **ligne 28** lecture d'un caractère c du flux ..... .

   **ligne 29 à 32** si une ..... de lecture s'est produite, on écrit un message d'erreur et on ..... le processus.

   **ligne 29 à 32** si la .... du flux d'entrée est détectée, on écrit un ..... sur le flux ....., puis on ..... 1/10 seconde.

   **ligne 37** Dans les autres cas, on écrit le caractère c sur le flux ..... .

   Compilez le.

   ```
   sh> gcc -o mycat /pub/ia/sys/syscall/mycat.c
   ```

2. Expérimentation de mycat sur des flux réguliers.

   (a) Lancez deux écrivains sur le fichier 1 puis deux lecteurs sur le même fichier dans quatre terminaux différents.

   ```
   sh> xterm -e bash -c "./mycat > 1" &
   ```

   ```
   sh> xterm -e bash -c "./mycat > 1" &
   ```

   ```
   sh> xterm -e bash -c "./mycat < 1" &
   ```

   ```
   sh> xterm -e bash -c "./mycat < 1" &
   ```

   (b) Tapez les entrées suivantes dans les fenêtres des écrivains en regardant leurs résultats dans les fenêtres des lecteurs.

   **aaaa**<C-R>**bbbb**<C-R> dans la fenêtre du $1^{er}$ écrivain.
   **cccc**<C-R>**dddd**<C-R>**eeee**<C-R> dans la fenêtre du $2^{nd}$ écrivain.
   **ffff**<C-R>**gggg**<C-R> dans la fenêtre du $1^{er}$ écrivain.

   (c) Stoppez les processus lecteurs avec <CTL-S> (vous pouvez les réveiller avec <CTL-Q>).

   (d) Soit P1 et P2 2 processus et un fichier régulier f déjà tamponné par le noyau:

   - Une lectures sur f n'est pas bloquante, si on est en fin de fichier, le noyau renvoie une valeur indiquant la fin de fichier.
   - Si P1 et P2 écrivent f à la même position, la donnée de f à cette position sera celle de la ..... .....
   - Si P1 et P2 lisent f à la même position, les données lues seront les mêmes si il n'y a pas eu ..... dans f à cette position entre les ..... .
   - Si P1 fait une lecture et P2 une écriture de f à la position l, P1 reçoit la donnée écrite par P2 si:
     - La lecture est faite ..... l'écriture.
     - Il n'y a pas eu d'autres ..... entre la lecture et l'écriture.

(e) Terminez les 4 processus.

3. Expérimentation de mycat sur des flux FIFO.

(a) Créez un fichier "fifo" correspondant à un pipe nommé.

(b) Lancez 2 écrivains et 2 lecteurs sur le fichier fifo dans quatre terminaux différents.

```
sh> xterm -e bash -c "./mycat > fifo" &
```

```
sh> xterm -e bash -c "./mycat > fifo" &
```

```
sh> xterm -e bash -c "./mycat < fifo" &
```

```
sh> xterm -e bash -c "./mycat < fifo" &
```

(c) Tapez les entrées suivantes dans les fenêtres des écrivains en regardant leurs résultats dans les fenêtres des lecteurs.

**aaaa**<C-R>**bbbb**<C-R> dans la fenêtre du $1^{er}$ écrivain.
**cccc**<C-R>**dddd**<C-R> dans la fenêtre du $2^{nd}$ écrivain.
**eeee**<C-R>**ffff**<C-R> dans la fenêtre du $1^{er}$ écrivain.

(d) Tuez les processus écrivains.

(e) Relancez un nouveau écrivain et tapez "ffff<C-R>" dans son terminal.

(f) Tuez les processus lecteurs.

(g) Soit P1 et P2 2 processus et un fichier fifo f:

- Une lectures sur f est bloquante, si f est .... et qu'il existe encore un ........ .
- De même, une écriture sur f est bloquante, si f est pleine et qu'il existe encore un lecteur.
- Une écriture sur une f, génère l'émission du signal .......... Si ce signal n'est ni attrapé, ni ignoré, ceci entraine la ........... du processus. Si ce signal est attrapé ou ignoré, l'écriture renvoie une ..... avec ..... positionné à EPIPE.
- Si P1 et P2 écrivent f, les données seront écrites dans f soit l'une derrière l'autre soit enchevêtrées.
- Si P1 et P2 lisent f, ils ne peuvent pas lire la ..... donnée.
- Si P1 fait une lecture de f et P2 une écriture dans f, P1 reçoit la donnée écrite par P2 si:
  - La fifo est ..... .
  - Il n'y a pas d'autres ........ entre la lecture et l'écriture.
  - Il n'y a pas d'autres ..... entre la lecture et l'écriture.

  Si ces conditions sont réalisées, l'ordre chronologique de la lecture et de l'écriture est ..... .

### *Exercice 3*

L'appel système int alarm(int ns) indique au système d'envoyer un signal SIGALRM au processus dans ns second.

1. Dans le fichier "mysleep.h", écrivez la fonction "void mysleep(int ns)" qui suspend l'exécution du processus pendant ns secondes (équivalent de la fonction sleep).

2. Testez la avec le programme mysleep.c ci dessous:

```c
#include <stdio.h>

#include "mysleep.h>

int main(int argc, char* argv)
{
  if (argc!=2) {
    fprintf(stderr,"%s: usage: %s sec\n",argv[0],argv[0]);
    exit(1);
  }
  mysleep( atoi(argv[1] );
  return 0;
}
```

### *Exercice 4*

L'appel système "pipe(int fd[2])" permet de créer un fichier FIFO non nommé. Le descripteur fd[0] donne accès au pipe en lecture et fd[1] en écriture.

Écrivez un programme qui calcule la taille d'un pipe.

| aide |
| --- |
| • Créez un pipe.<br>• Écrivez indéfiniment un caractère dans le pipe.<br>• Avant d'écrire dans le pipe, écrivez le nombre de caractères déjà écrits dans le pipe.<br>• Quand le pipe sera plein l'écriture sera bloquante. |

## 9.5 Création de processus

Pour ces exercices, vous n'avez besoin sur votre écran que d'un éditeur et un terminal xterm visible simultanément.
Le terminal doit être de taille raisonnable ($\sim$ 25 lignes, $\sim$ 100 colonnes) et si vous voulez de l'aide avec une police lisible à un mètre (<CTL-+CLIC-DROIT> et choisir Huge).
L'éditeur doit être de taille raisonnable ($\sim$ 35 lignes, $\sim$ 80 colonnes), avec une police lisible à un mètre, et afficher en début de chaque ligne le numéro de ligne.

Enfin ce document est accessible en ligne: /pub/ia/sys/syscall/sys-poly2.pdf

### *Exercice 1*

1. Écrivez le programme hello dont le fonctionnement est le suivant:
   - Le père crée un $1^{er}$ fils, puis un $2^{nd}$, puis écrit "hello" sur le flux standard d'entrée, puis se termine.
   - Le $1^{er}$ fils attends 2 secondes puis écrit un "\n" sur le flux standard d'entrée, puis se termine.
   - Le $2^{nd}$ fils attends 1 seconde puis écrit " world" sur le flux standard d'entrée, puis se termine.

2. Modifiez le programme hello pour que le père attende que ses 2 fils soient terminés avant de se terminer.

### *Exercice 2*

Écrivez le programme "**sleepsort $\mathbf{n_1}$ $\mathbf{n_2}$ ...** " qui écrit sur le fichier standard de sortie les entiers $n_i$ triés par ordre croissant. Son fonctionnement est le suivant:

- Le père crée un fils pour chaque entier $n_i$, puis il attend la terminaison de tous ses fils.
- Le fils$_i$ attend $i$ secondes, écrit $i$, puis se termine.

### *Exercice 3*

L'objectif de cet exercice est la réalisation d'une version parallélisée de la commande "wc -l".

1. Écrivez le programme wc-par dont le fonctionnement est le suivant:
   - Le père calcule la taille (appelée sz dans la suite) du fichier passé en argument (argv[0]), crée 4 fils, attend que ses fils se terminent, puis se termine.
   - Le $1^{er}$ fils ouvre le fichier, lit les octets [0..sz/4[ et écrit sur le flux standard de sortie le nombre de lignes (caractère '\n') de cette partie du fichier.
   - Le comportement du $2^{nd}$ est identique mais pour la partie [sz/4..sz/2[ du fichier.
   - Le comportement du $3^{ième}$ est identique mais pour la partie [sz/2..3*sz/2[ du fichier.
   - Le comportement du $4^{ième}$ est identique mais pour la partie [3*sz/2..sz[ du fichier.

   Testez votre implémentation et vérifiez que vos résultats sont les même que ceux produits par la commande "wc -l".

2. Modifiez le programme wc-par pour que le père affiche la somme des résultats des 4 fils.

   aide

   Le père créera un pipe, y lira 4 entiers qu'il sommera.
   Les fils écriront leurs résultats dans le pipe.

3. Faites des essais pour trouver le degré optimal de parallélisation sur votre machine.

   Faites vos essais avec un fichier de 200 mo placé dans /tmp.

### *Exercice 4*

Cet exercice porte sur le programme forkPN dont les sources sont disponibles en /pub/ia/sys/syscall/forkPN.c .

1. Indiquez ce que fait le programme forkPN.

   Il crée 2 ..... et 2 processus .... appelés pos et neg. Il lit des entiers sur le flux standard d'entrée:

   - si l'entier lu est supérieur ou égal à ∅, il l'envoie à son fils .... via le pipe ..... .
   - si l'entier lu est inférieur ou égal à ∅, il l'envoie à son fils .... via le pipe ..... .
   - si l'entier lu est ∅, il attend la ..... de ses .... avant de se terminer.

   Les fils quand à eux lisent des entiers sur leurs ..... respectifs les affichent et se terminent quand l'entier lu est ∅.

   Le père affiche ses messages sur le flux standard ..... et les fils sur le flux standard ..... .

   Compilez le.

   ```
   sh> gcc -o forkPN /pub/ia/sys/syscall/forkPN.c
   ```

   Puis lancez le comme indiqué ci-dessous pour avoir des entrées et sorties lisibles.

   - Créez un nouveau terminal, récupérez le nom du tty de ce terminal, puis lancez un sleep de 1000 secondes dans ce nouveau terminal.

     ```
     sh> xterm -e bash -c "tty ; sleep 1000" &
     ```

   - Dans votre terminal initial, lancez forkPN en redirigeant le flux standard de sortie sur le tty du nouveau terminal

     ```
     sh> ./forkPN > /dev/pts/N
     ```

     où N est écrit en haut du second terminal.

2. Faites forkPN-v1.c similaire à forkPN.c mais le père comme les fils s'arrêtent sur la fin de leur fichier de lecture. Le père attend toujours la fin des fils pour se terminer.

   aide

   Il faut que les fils détectent la fin de fichier lors de la lecture sur leur pipe.
   Pour cela, il faut que le pipe n'aie plus d'écrivains.

3. Faites forkPN-v2.c similaire à forkPN-v1.c mais si un fils meurt (commande: **sh> kill -9 pid-fils**) le père tue l'autre fils et écrit sur le fichier standard d'erreur:

   ```
   pere:pid: fin inattendue du fils pid
   pere:pid: tue fils pid
   pere:pid: mes fils sont morts (fin)
   ```

   aide

   Le père peut détecter la mort d'un fils lors du write dans le pipe si le fils était le seul lecteur du pipe.

   En effet dans ce cas, si le fils se termine, il n'y a plus de lecteur sur le pipe et donc sur un write dans ce pipe, le noyau enverra le signal SIGPIPE au père, et une erreur d'écriture.

   **Solution 1:** Tuer les 2 fils dans le gestionnaire de SIGPIPE (le retour de kill donne le processus qui était déjà mort).

   **Solution 2:** Tester si le write dans le pipe fait une erreur, dans ce cas on sait que ce pipe n'a plus de lecteur donc que le fils qui le lisait est terminé.

4. Faites forkPN-v3.c similaire à **forkPN.c** (l'original) mais en plus

   - Le père arrête de lire quand il reçoit le signal SIGINT (<CTL-C> dans le terminal).
   - Le père envoie alors le signal SIGUSR1 à ses fils.
   - Sur la réception du signal les fils se terminent en écrivant "fils:pid: fin due à la réception de SIGUSR1" sur le fichier standard d'erreur.
   - Le père attend la mort de ses fils et écrit: "`pere:pid mes fils sont morts (fin)`".

### *Exercice 5*

Cet exercice porte sur les programmes runcmd-pl et runcmd-vl dont les sources sont disponibles en /pub/ia/sys/syscall/runcmd-pl.c et /pub/ia/sys/syscall/runcmd-vl.c .

Ils sont fonctionnellement semblables: ils construisent une SD de n mégaoctets (n étant le premier argument de ces programmes), puis répètent 1000 fois: création d'un fils qui exécute la commande "echo -n hello" puis attente de la fin du fils.

Ils diffèrent par leur implémentation. En effet, runcmd-pl crée ses fils avec fork (processus lourd) tandis que runcmd-vl crée ses fils avec clone (processus léger).

1. Compilez puis lancez les 2 programmes.

```
sh> gcc -o runcmd-pl /pub/ia/sys/syscall/runcmd-pl.c
```

```
sh> gcc -o runcmd-vl /pub/ia/sys/syscall/runcmd-vl.c
```

```
sh> ./runcmd-vl 1
```

```
sh> ./runcmd-pl 1
```

2. Mesurez les temps d'exécution des 2 programmes, avec une SD de 1 mégaoctets, en redirigeant le flux standard de sortie sur le fichier /dev/null pour ne pas messurer le temps mis par l'affichage.

```
sh> time ./runcmd-vl 1 > /dev/null
```

```
sh> time ./runcmd-vl 1 > /dev/null
```

3. Recommencez ces mesures pour des SD de 10, 100, 300, 600 et 1200 mégaoctets.

4. Expliquez les mesures obtenues.

5. Quels programmes doivent être particulièrement précautionneux sur ce sujet?

### *Exercice 6*

L'objectif de cet exercice est la réalisation d'un minishell.

1. Écrivez un programme, qui lit des commandes sur le fichier standard d'entrée, les unes derrière les autres et les exécute en avant plan (il attend la fin de la commande courante pour en lire une nouvelle et l'exécuter). Les commandes n'ont pas d'arguments (ex: ls, cat, ./a.out, false ...). Le programme écrit un message compréhensible si la commande demandée n'a pas été trouvée.

   Enfin, sur la fin du fichier standard d'entrée ou la commande "exit", il se termine.

aide

Une boucle simple de lecture des commandes simplifiée peut être:

```c
// exécute la commande cmd
// renvoie 0: fin du minishell
// renvoie 1: lire la commande suivante
int shell(char * cmd)
{
   if ( strcmp(cmd,"exit")==0 )
     return 0;

   return 1;
}

int main()
{
  char cmd[1000],line[1000];

  while (1) {
    if ( feof( stdin ) ) break;
    printf("shell> ");
    if ( fgets(line,1000, stdin )==0 ) continue;
    if ( sscanf(line,"%s",cmd)!=1 ) continue;
    if ( shell(cmd)==0 ) break;
  }
  printf("\n");
  return 0;
}
```

2. Complétez le programme précédent avec les commandes "`$$`" (le pid du programme) et "`$?`" le statut de la dernière commande.

3. Complétez le programme précédent avec les commandes "`<chemin`" et "`>chemin`" (pas d'espace) qui redirige les fichiers standards

d'entrée et de sortie sur le fichier "chemin" pour la prochaine commande uniquement.

4. Complétez le programme précédent pour que "`&cmd`" lance la commande cmd en arrière plan.

5. Complétez le programme précédent pour que la commande "`fg`" fasse passer en avant plan, la dernière commande lancée en arrière.

6. Complétez le programme pour que les commandes `|cmd1` puis `cmd2` pipe `cmd1` sur `cmd2` (équivalent Shell: **sh> cmd1 ¦ cmd2**).

   On pourra tester avec ls et head ou tail.

7. Complétez le programme précédent pour que le <CTL-C> (qui génère le signal SIGINT), ne tue ni le minishell, ni les processus en arrière plan mais seulement le processus en avant plan si il existe.

   aide

   Pour cela 1) il faut que le processus minishell ignore le signal SIGINT 2) il faut que le processus minishell aie le traitement par défaut pour le signal SIGINT 3) il faut mettre le processus en arrière plan dans un nouveau groupe de processus "setpgid(0,0)".

8. Complétez le programme précédent pour que la touche <CTL-Z>, suspende le processus en avant plan uniquement et pas le minishell ni les processus en arrière plan.

   aide

   (a) le noyau envoie le signal SIGSTP à tous les processus du groupe de processus qui contrôle le tty.

   (b) le minishell doit attraper ce signal.

   (c) Le processus en avant plan doit avoir le traitement par défaut (stop) sur ce signal et appartenir au groupe qui contrôle le tty. Le fork met le fils dans le groupe du père.

   (d) Les processus en arrière plan n'appartiennent plus au groupe (question précédente), ils ne reçoivent donc pas le signal SIGTSTP.

   (e) La commande `fg` fait repasser en avant plan soit le processus en avant plan stoppé, soit le dernier processus lancé en arrière plan. Dans ce dernier cas il faut le remettre dans le groupe de processus qui contrôle le tty (getpgid et tcsetpgrp) pour que le <CTL-Z> soit de nouveau actif.

## 9.6 Thread

Pour ces exercices, vous n'avez besoin sur votre écran que d'un éditeur et un terminal xterm visible simultanément.
Le terminal doit être de taille raisonnable ($\sim$ 25 lignes, $\sim$ 100 colonnes) et si vous voulez de l'aide avec une police lisible à un mètre (<CTL-+CLIC-DROIT> et choisir Huge).
L'éditeur doit être de taille raisonnable ($\sim$ 35 lignes, $\sim$ 80 colonnes), avec une police lisible à un mètre, et afficher en début de chaque ligne le numéro de ligne.

Enfin ce document est accessible en ligne: /pub/ia/sys/syscall/sys-poly2.pdf

### *Exercice 1*

Écrivez le programme hello.c qui écrit "hello" sur le flux standard de sortie. Pour cela il crée 2 threads.

- Le $1^{er}$ thread écrit 'h' et 'o'.
- Le $2^{nd}$ thread écrit 'ell'.
- Le thread principal écrit '\n'.

1. Utilisez des sleep pour synchroniser les écritures.
2. Utilisez 2 sémaphores pour synchroniser les écritures (plus de sleep).

### *Exercice 2*

L'objectif de cet exercice est la réalisation du programme pm (**P**rise **M**ultiple) qui s'utilise de la manière suivante:

**sh> ./pm $\mathbf{in_0}$ $\mathbf{in_1}$ ... $\mathbf{in_n}$ -- $\mathbf{out_0}$ $\mathbf{out_1}$ ... $\mathbf{out_m}$** où les $\text{in}_i$ sont des flux d'entrée et les $\text{out}_i$ des flux de sortie. **pm** lit les flux $\text{in}_i$ caractère par caractère et diffuse les caractères lus sur tous les flux $\text{out}_j$.

1. /pub/ia/sys/syscall/pm-seq.c est un version séquentielle (très limitée) de pm. Compilez la et essayez la de la manière suivante:
   - Créez 5 pipes nommés in0, in1, out0, out1, out2.
     ```
     sh> mkfifo in0 in1 out0 out1 out2
     ```
   - Créez 5 terminaux.
     ```
     sh> xterm & # 5 fois
     ```
   - Lancez deux écrivains dans les premiers terminaux et 3 lecteurs dans les suivants.
     ```
     sh> cat > in0 # 1er terminal
     ```
     ```
     sh> cat > in1 # 2nd terminal
     ```
     ```
     sh> cat < out0 # 3ième terminal
     ```
     ```
     sh> cat < out1 # 4ième terminal
     ```
     ```
     sh> cat < out2 # 5ième terminal
     ```
   - Lancez la prise multiple pour connecter ces écrivains et lecteurs.
     ```
     sh> ./pm in0 in1 -- out0 out1 out2
     ```
   - Tapez "a<C-R>" dans le terminal 1 et "123<C-R>" dans le terminal 2.

     Les lectures sur les $\text{in}_i$ ne sont pas indépendantes.
   - Terminez un cat dans un terminal lecteur (<CTL-C>).

     pm est sensible à l'existence des lecteurs.

2. Écrivez la version pm-v1 qui a les caractéristiques suivantes:

- Le programme principal crée autant de threads (appelés lecteurs) que de flux d'entrée, puis attend indéfiniment.
- Un thread lecteur répète indéfiniment:
  - lecture d'un caractère c sur son flux d'entrée.
  - diffusion de c sur tous les flux de sortie.

  Sur un erreur de lecture, il se termine avec le message:
  "in%d:exit: pb lecture : %s\n"

  Sur un E.O.F en lecture, il affiche le message ci-dessous et attend 1 seconde avant de recommencer une lecture:
  "in%d:eof : attente\n"

  Sur un erreur d'écriture grave, il affiche le message ci-dessous et passe au flux de sortie suivant.
  "in%d:igno: pb écriture sur out%d: %s\n"

  Sur un erreur d'écriture FIFO sans lecteur, il affiche le message ci-dessous et attend 1 seconde avant de recommencer l'écriture:

  "in%d:att : pb écriture sur out%d: %s\n"

Testez votre implémentation. Que se passe-t-il si on termine un lecteur? Que se passe-t-il si on le relance?

3. Écrivez la version pm-v2 qui crée toujours un thread lecteur par flux d'entrée et un pipe et un thread écrivain par flux de sortie.
   Les threads lecteur écrivent dans les pipes et plus dans les flux de sortie. Les threads écrivain lisent leur pipe et écrivent leur flux de sortie.
   On adaptera les messages de pm-v1

4. Écrivez la version pm-v3 qui est similaire à pm-v2 mais dans le cas où l'écrivain n'est pas prêt, on n'attend pas qu'il soit prêt, mais on tamponne les N dernières données. Ainsi quand l'écrivain redémarre, il lira les N dernières données.
   Pour l'implémentation, on fixera N à 10.

   aide

   Il faut ajouter un compteur à chaque pipe, qui indique combien d'octets sont dans le pipe.

5. Vérifiez que votre implémentation des threads lecteur et écrivain de pm-v3 est thread-safe.

   aide

   - Lancez pm-v3 avec 2 écrivains et 3 lecteurs et vérifiez que le système fonctionne.
   - Terminez les lecteurs.
   - Envoyer sur in0 un fichier texte d'au moins 64 kilooctets.
   - Redémarrez les trois lecteurs. Il doivent tous écrire les 10 mêmes caractères.

   Si ce n'est pas le cas, rendez thread-safe vos threads lecteur et écrivain.

6. Écrivez la version pm-v4 qui est similaire à pm-v3 mais les pipes sont remplacés par des FIFO circulaires de N octets avec attente active en lecture.

   aide

   Une FIFO de N octets s'implémente avec la structure suivante:

```
1  #define NBOCTETS 10
2  typedef struct _Tfifo {
3    char t[NBOCTETS];    // les données enfilées
4    int sz;              // nombre d'octets enfilés
5    int lec;             // t[lec] est le plus vieux octet
6    int ecr;             // t[ecr] est le plus récent octet
7  } Fifo;
```

   lec et ecr s'incrémentent modulo NBOCTETS: lec = (lec+1)%NBOCTETS.

7. Écrivez la version pm-v5 qui est similaire à pm-v4 mais sans attente active.

   aide

   Il faut utiliser les fonctions POSIX pthread_cond_wait et pthread_cond_signal et ajouter à la structure Tfifo un pthread_cond_t.