Système d'exploitation

Le 30 août 2022, SVN-ID 425

Table des matières

1 Fondement **2**
- 1.1 Organisation . . . 2
- 1.2 Système de protection . . . 2
- 1.3 Fichiers et systèmes de fichiers . . . 3
- 1.4 Processus . . . 6
- 1.5 Système Unix . . . 7

2 Shell interactif **11**
- 2.1 Séquence d'instructions . . . 11
- 2.2 Variables et environnement . . . 11
- 2.3 Expansions . . . 12
- 2.4 Redirections . . . 13

3 Shell script **15**
- 3.1 Mon premier script . . . 15
- 3.2 Instructions de contrôle . . . 15
- 3.3 Quelques indispensables . . . 18
- 3.4 Création de Builtin commande . . . 21
- 3.5 Exemple complet . . . 22
- 3.6 Astuces et pièges . . . 23

4 Appel système **25**
- 4.1 Organisation . . . 25
- 4.2 Format général d'un appel système . . . 28

5 Flux **29**
- 5.1 Algorithmes . . . 29
- 5.2 Les flux noyau . . . 30
- 5.3 Les flux libc . . . 34
- 5.4 Mapping . . . 36
- 5.5 Comparaison . . . 37

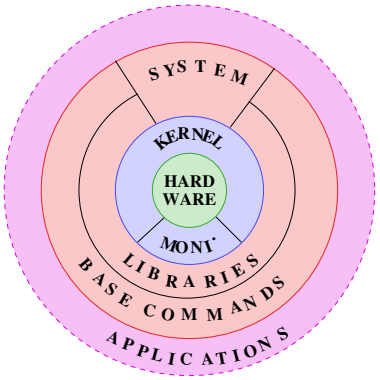
6 Quelques fonctions système **38**
- 6.1 Exec . . . 38
- 6.2 Exit . . . 38
- 6.3 Environnement . . . 39
- 6.4 Divers . . . 39

7 Communication inter-processus **40**
- 7.1 Signaux . . . 40
- 7.2 FIFO . . . 42
- 7.3 SHM et Sémaphore . . . 44

8 Processus **46**
- 8.1 Processus Unix . . . 46
- 8.2 Thread POSIX . . . 49
- 8.3 Fonction réentrante et thread-safe . . . 51

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
700	---	rw-	---	110	200
644	rw-	r--	r--	0	0
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

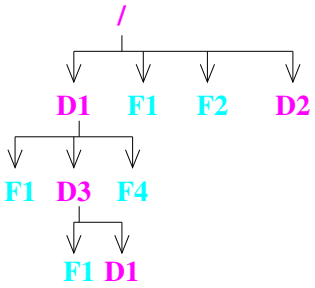
Complétez les colonnes accès de la table ci-dessous. Le répertoire D contient le fichier F.

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

1.3 Fichiers et systèmes de fichiers

1.3.1 Système de fichiers

1.3.1.1 Définition

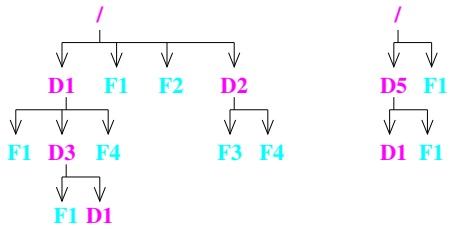


Arborescence de fichiers :

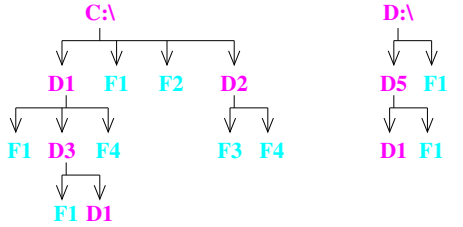
- les noeuds : fichiers répertoires
- les feuilles : fichiers ou répertoires vides
- la racine le haut de l'arbre

Support physique : disques durs, RAM

1.3.1.2 Plusieurs systèmes de fichiers (Windows)

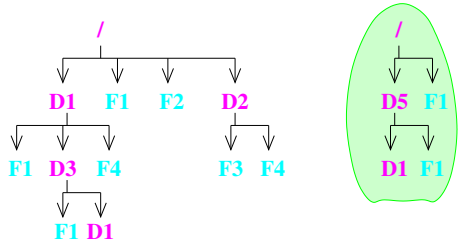


Chaque disque et/ou partition a un système de fichiers
$\Longrightarrow$ nombreux systèmes de fichiers.

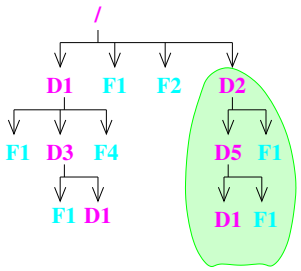


Sous Windows, les systèmes de fichiers sont visibles
$\Longrightarrow$ identifiés par une lettre suivie de ' :'.

1.3.1.3 Plusieurs systèmes de fichiers (Unix)

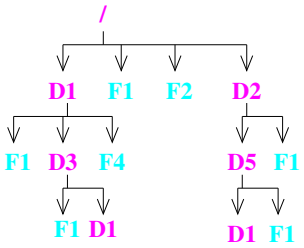


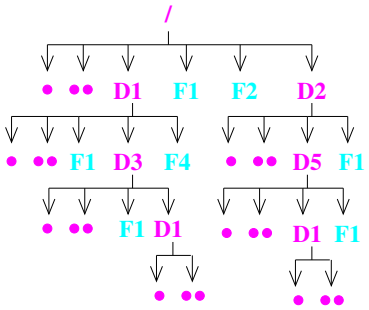
Chaque disque et/ou partition a un système de fichiers
$\Longrightarrow$ nombreux systèmes de fichiers.



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

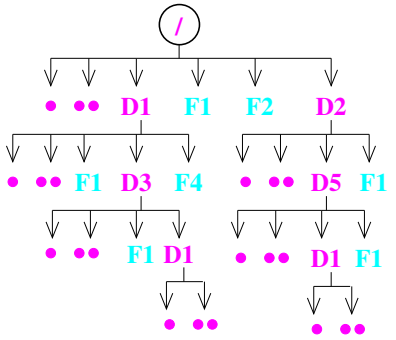
Suite de noms (nom=chaine de caractères sans '/') séparés par au moins un '/' et précédés et terminés par $\emptyset$ ou plusieurs '/' :
$[/]nom_0/nom_1/.../nom_n[/]$
Un chemin est valide si

- 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.



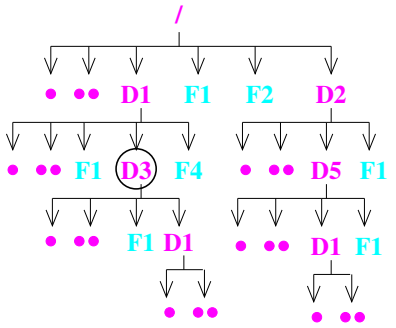
/D1/D3/D1
/D1/D3/D1/
/D1/D3/D1/.
/D1/D3/D1.
///D1//D3////D1

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

1.3.3.3 Chemins relatifs

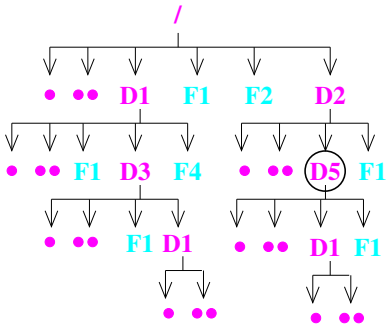
Tout programme qui tourne a un répertoire de travail associé qui s'appelle CWD ou WD (Current Working Directory).
Un chemin relatif ne commence pas par un '/', il part du répertoire CWD.

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

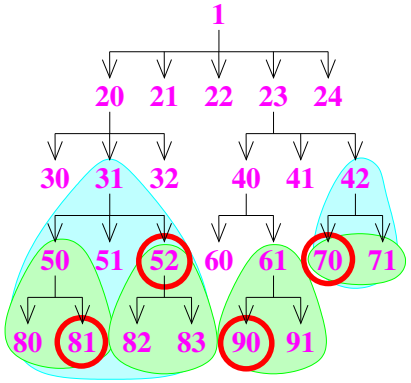
Soit un système de fichiers sans lien, indiquez si les propositions suivantes sont vraies ou fausses.

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

En avant plan (fg)

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

En arrière plan (bg)

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

avec

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

sleep 60 &

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

644 et 60 1^{er} argument, sa signification dépend de la commande.

gnu/bee 2^{nd} 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.

1.5.2.2 Job contrôle

<CTL-C>	demande au processus en avant plan de se terminer.
<CTL-Z>	demande au processus en arrière plan de suspendre son exécution.
fg	• fait passer en avant plan le dernier processus lancer en arrière plan. • si ce processus était suspendu, il le relance.
bg	relance le processus suspendu en arrière plan

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.

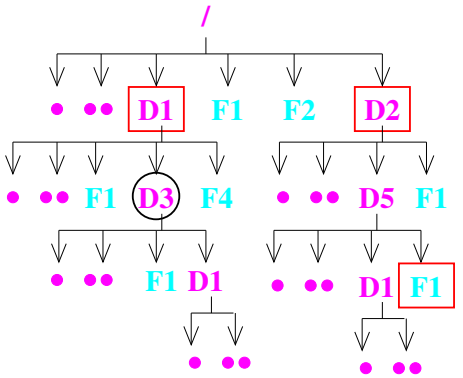
  Si il y a conflit pour compléter, une deuxième tabulation affiche les possibilités.

1.5.3 Les commandes de base

1.5.3.1 Liste

La figure 1 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
   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>

Les commandes de bases en mode interactif :

cd	:	change le CWD.
pwd	:	affiche 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
which	:	affiche le path absolu d'une commande
less	:	un pageur
chmod	:	modification des permissions 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
sort	:	tri de fichier
stat	:	affiche les meta-données (taille, type, ...) d'un fichier
tar	:	archiveur

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
sleep N	:	attente de N secondes

FIGURE 1 : les commandes essentielles

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. L'opérateur **!** (non) permet d'inverser la condition logique.

```
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 ;  }
```

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

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 :
   $DIRgnu $\Longrightarrow$ **DIRgnu** non définie
   ${DIR}gnu $\Longrightarrow$ beegnu
3. La forme parenthésée permet de nombreuses extensions :
   ${DIR :-undef} $\Longrightarrow$ 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>
```

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

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

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 '/'.

[$m_0m_1m_2$...**]** 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-zA-Z]*/*.[024]*
sh> gcc src/*.c
sh> wc -l prj/*.[ch]
sh> wc -l prj/*{.c,.h}
```

```
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 2.

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.

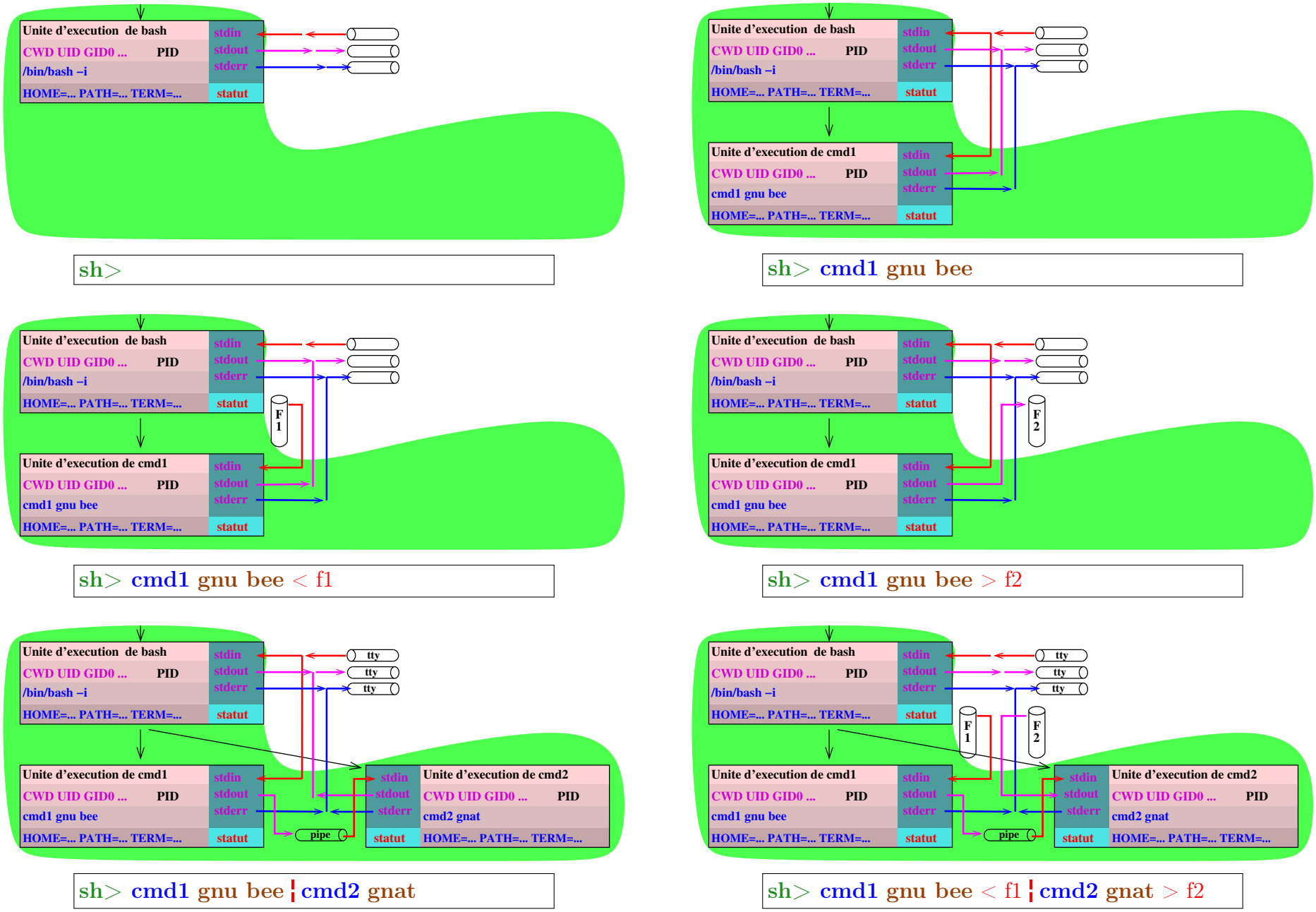


FIGURE 2 : Principe des redirection

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 :

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

3.2.1.3 Exemple 2

Écrire oui si on est en 2016 ?

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

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

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

3.2.2 Choix conditionnel

3.2.2.1 Cas d'une constante

```
case mot in
  motif )
    ...
    ;;
  motif | motif )
    ...
    ;;
  ...
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

```
if cmd-cond1 ; then
  ...
elif cmd-cond2 ; then
  ...
elif cmd-cond3 ; then
  ...
else
  ...
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.

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

Écrire 10 lignes contenant chacune 5 fois silence.

```
for i in {1..10} ; do
  for j in {19..15} ; do
    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 | done
4 | 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
```

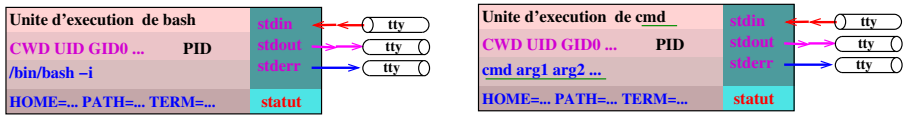
ou

```
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

⟹ pas de création de processus.
⟹ l'instruction qui suit l'exec ....

exit n Termine le Shell avec le statut n. Si n est omis, le statut est ∅.

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.

inst. \ var.	$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.

```
#!/bin/bash
#
# usage: punition [n] word

test $# = 1 && set -- 5 "$1"

i=0
while test $i -lt $1 ; do
   echo $2
   i=$[i+1]
done
```

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

```
#!/bin/bash
#
# usage: punition n word1 word2 ...

n=$1
shift
i=0
while test $i -lt $n ; do
   echo $*
   i=$[i+1]
done
```

3.4 Création de Builtin commande

```
#!/bin/bash
...
# usage: plus n1 n2 n3 ...
function plus( )
{
   s=0
   while test $# -gt 0 ; do
      s=$[$s+$1]
      shift
   done
   echo $s
}
...
plus 10 12 100
...
s=$(plus 1 2 3)
...
```

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

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

OK : test -d "bee old"

⟹ double quoter $(cmd) ("$(cmd)")

```
1 set -- "a" "b" "c"
2 for s in $* ; do
```

OK : 3 fois la boucle

```
1 set -- "a" "b" "c d"
2 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

```
x=0 ; a=0
sed /^#/d f1 | \
while read a b ; do
  x=$a
   ...
done
```

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

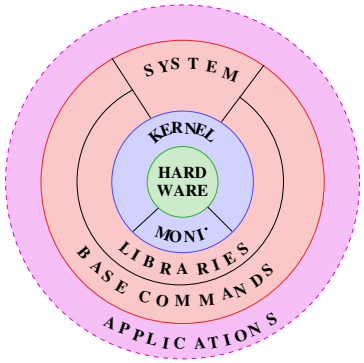
x

a

4 Appel système

4.1 Organisation

4.1.1 Couches Principales



matériel CPU, RAM, contrôleurs et périphériques.

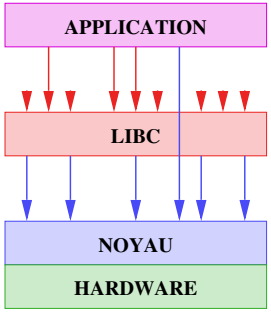
moniteur Petit programme en ROM, qui tourne au démarrage de la machine.

noyau Gère et donne accès au matériel

système Couche de standardisation

applications

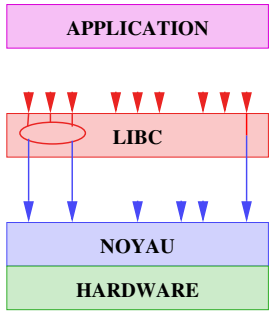
4.1.2 Appel système et libc



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



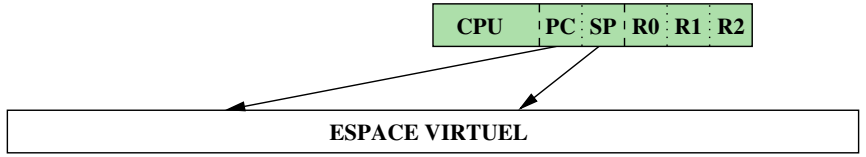
libc : module standalone service n'utilisant pas les appels système (ex : module strxxx)

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



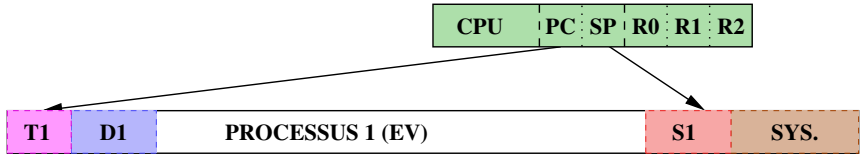
CPU Quelques registres

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

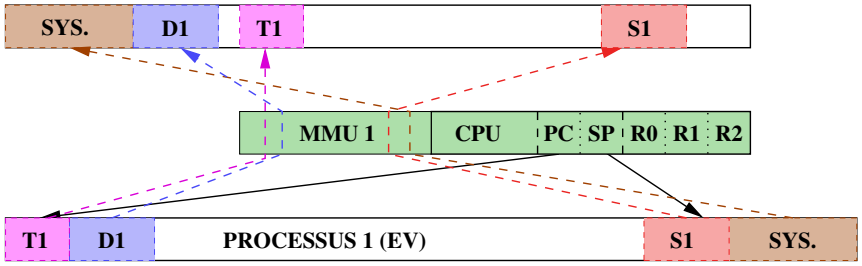
- Si le processeur est en mode système : il peut accéder à tout l'espace virtuel (sauf les trous).
- Si le processeur est en mode user : il ne peut pas accéder à l'espace système ($\Longrightarrow$ "privilege violation").

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

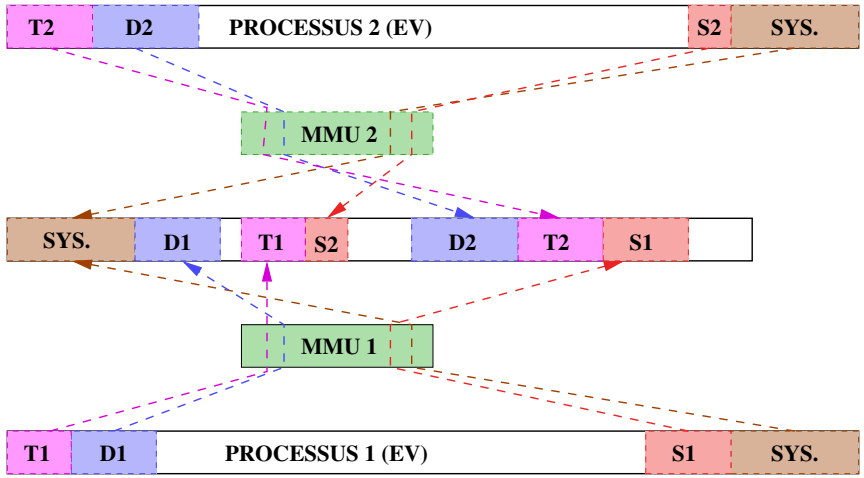
Fonction En fonction des valeurs contenues dans les registres :

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

MMU i La MMU configurée pour le processus i.

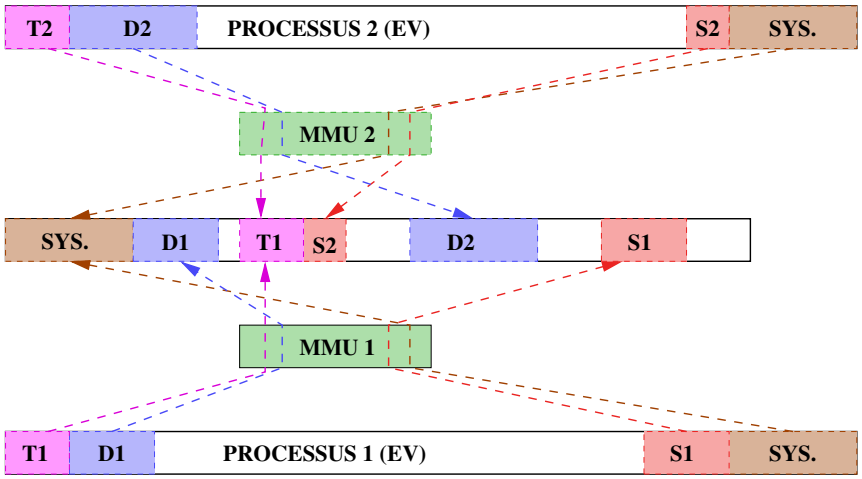
4.1.3.5 Quelques configurations

Partage de l'espace système



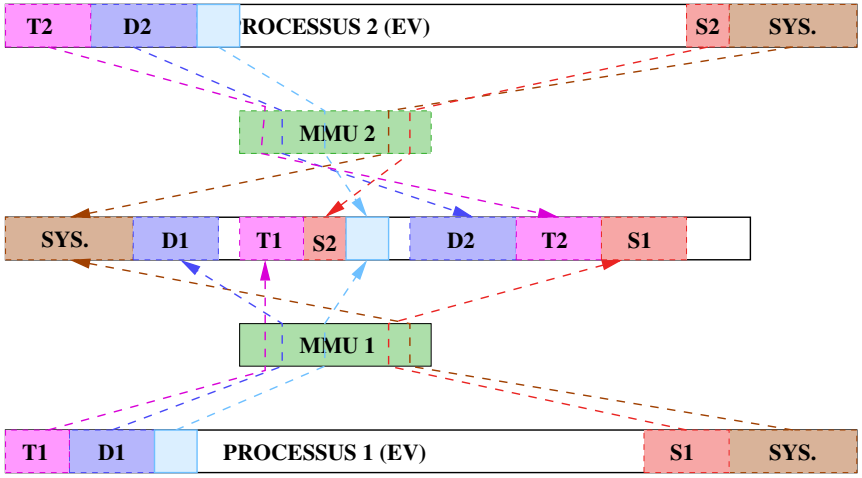
Le segment système est partagé entre les deux processus.

2 processus du même exécutable



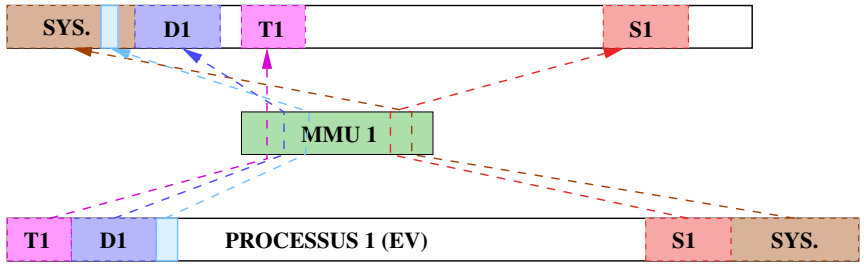
Les segments système et text sont partagés entre les deux processus.

Mémoire partagée



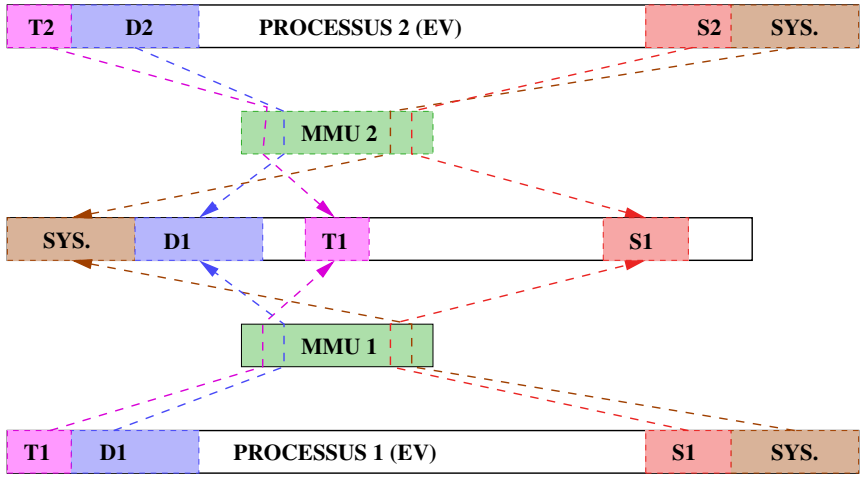
Le segment système et un bout de segment données sont partagés entre les deux processus. Les 2 processus peuvent s'échanger des données au travers de ce segment.

Accès direct à un tampon système



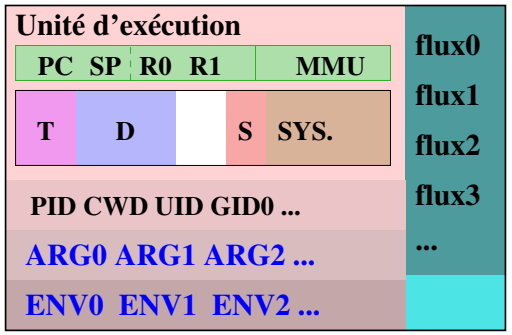
Un tampon de donnée système est mappé dans l'espace utilisateur (segment donnée). Le processus en mode utilisateur peut accéder à cette partie de l'espace système.

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.

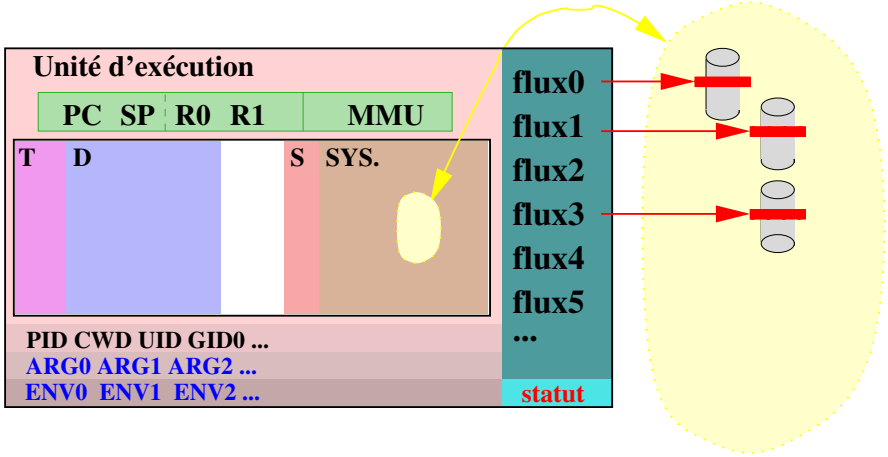
4. Après une lecture sur un flux avec un statut EOF, une seconde lecture donne toujours EOF.

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

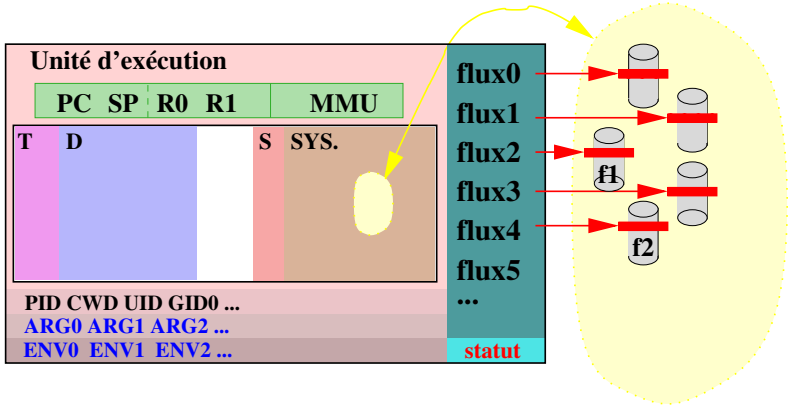


type Entier

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.

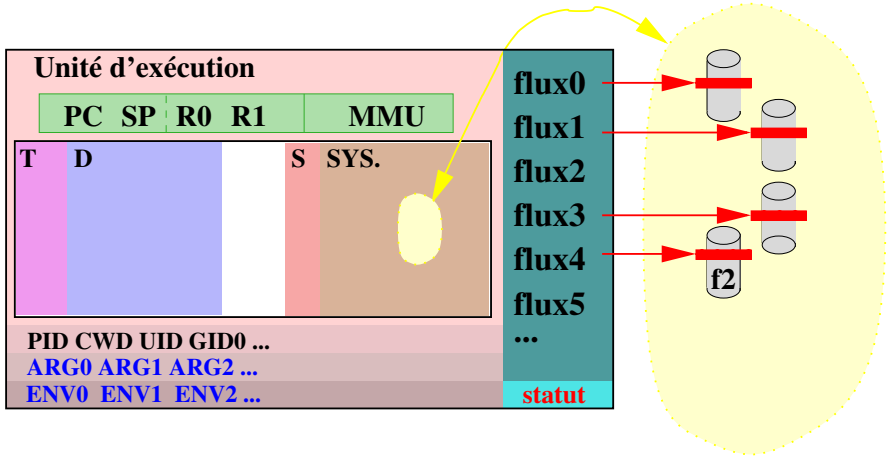
Exemple 1 open("f1",O_RDONLY) recherche le 1^{er} fd libre $\Longrightarrow$ 2.

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

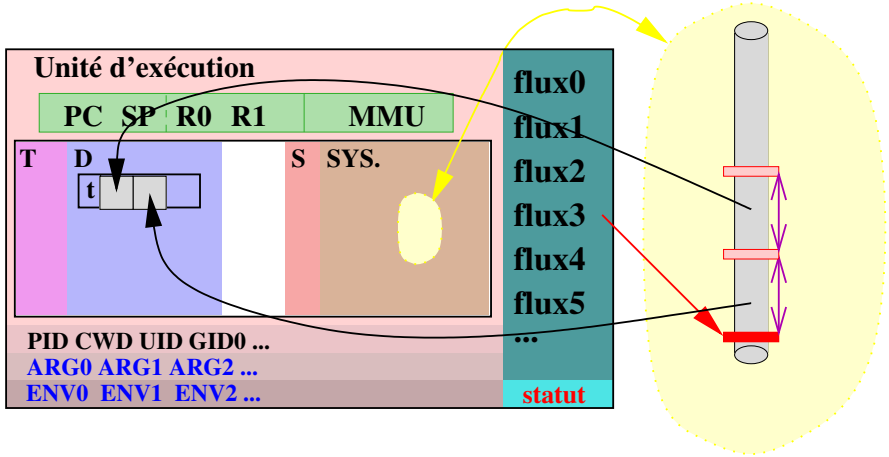


Synopsis int close(int fd) ;

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

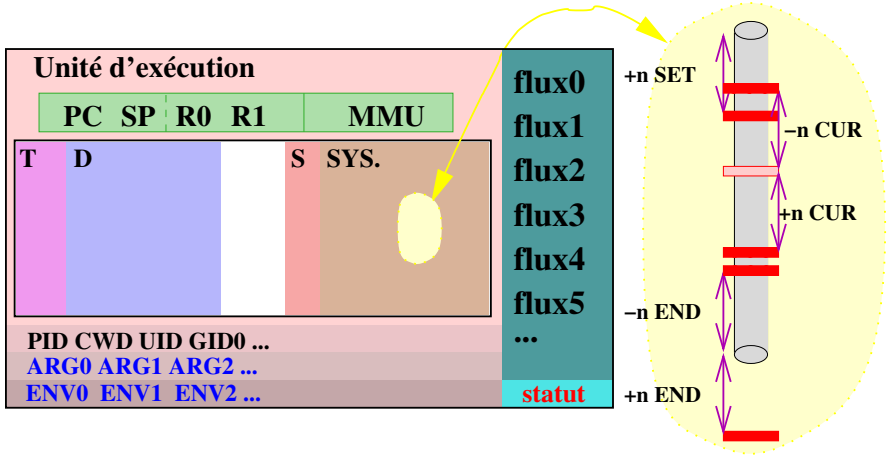
Synopsis size_t write(int fd, void *buf, size_t count) ;

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.

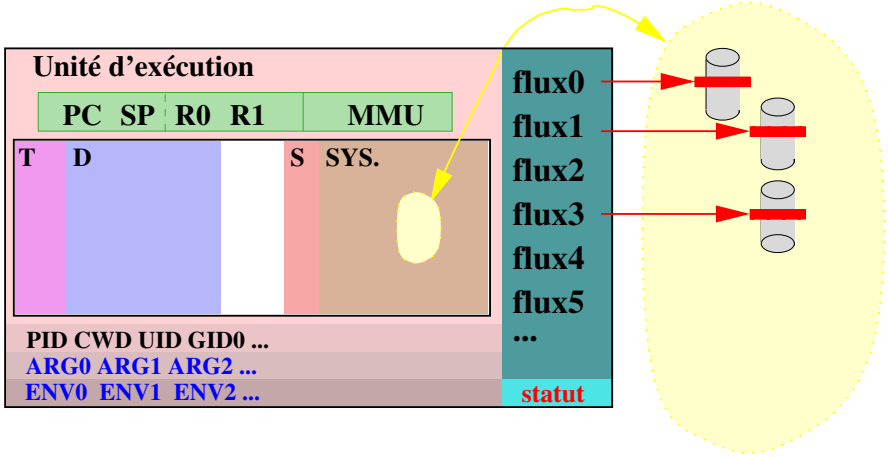
Exemple "pos=lssek(3,+10,SEEK_END)" avance à partir de la fin.

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    }
```

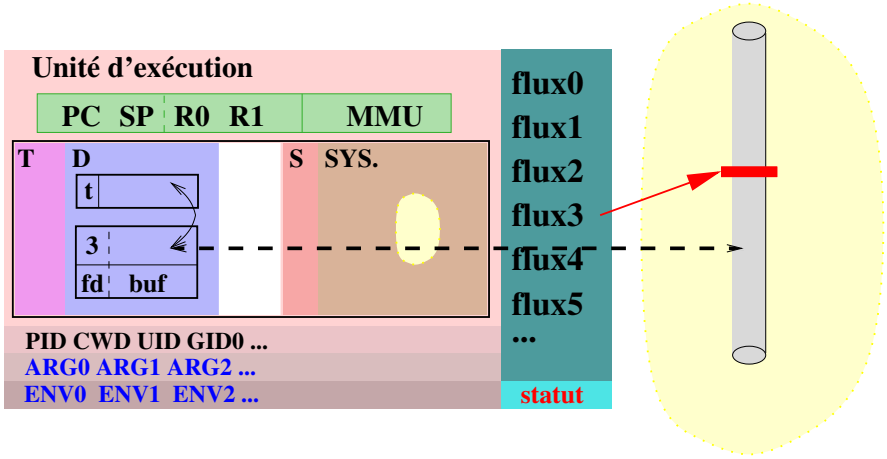
Terminaison

```
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.

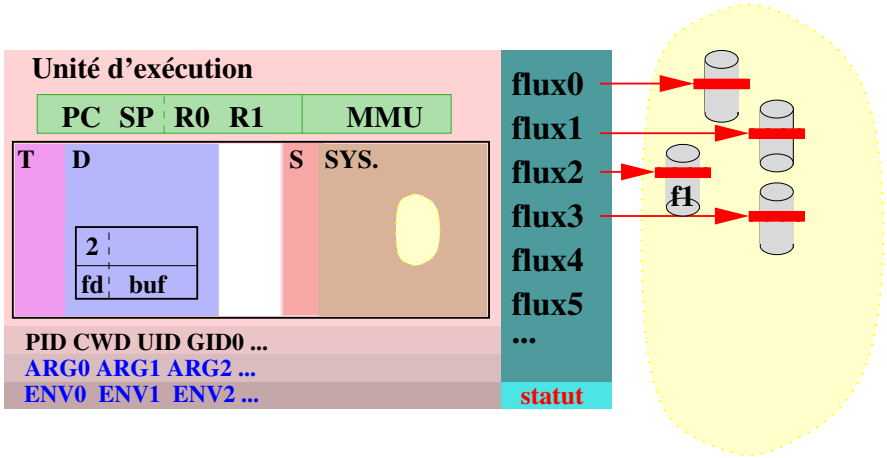
long ftell(FILE ∗f)

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.

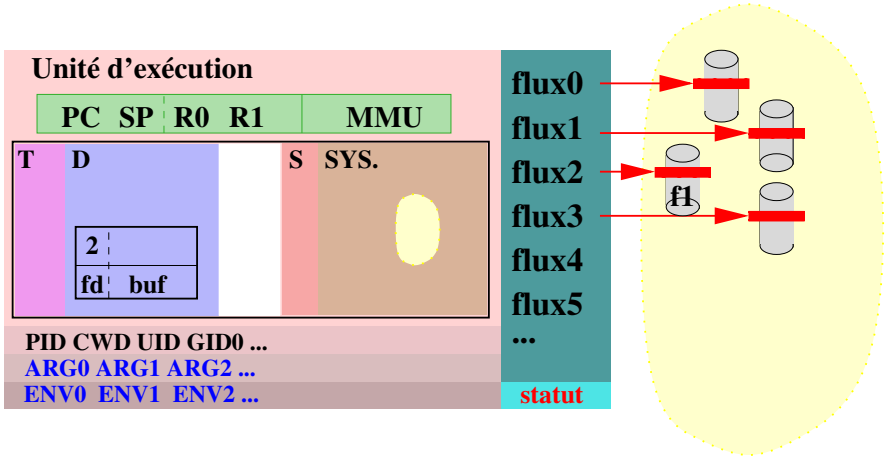
Exemple "fopen("f1","r")" crée un flux noyau (2) attaché au fichier f1, alloue un FILE∗ et l'associe au flux noyau.

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.

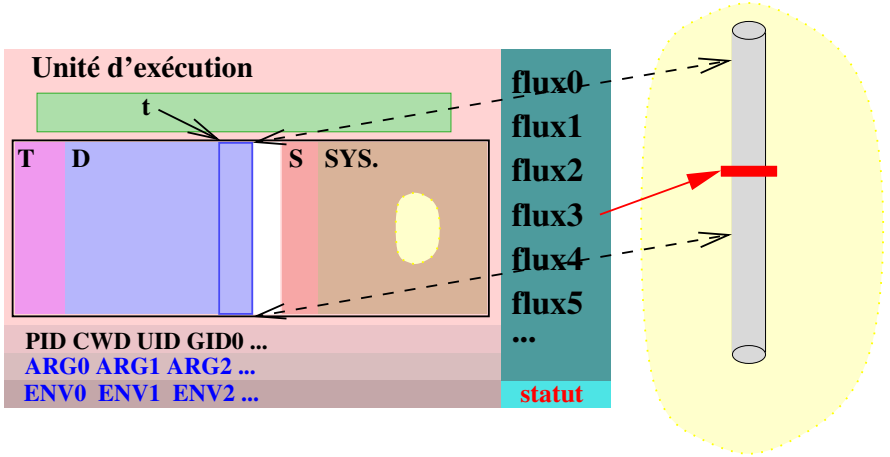
```
printf("hello world"); // tamponné
printf("hello world\n"); // non tamponné
```

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.

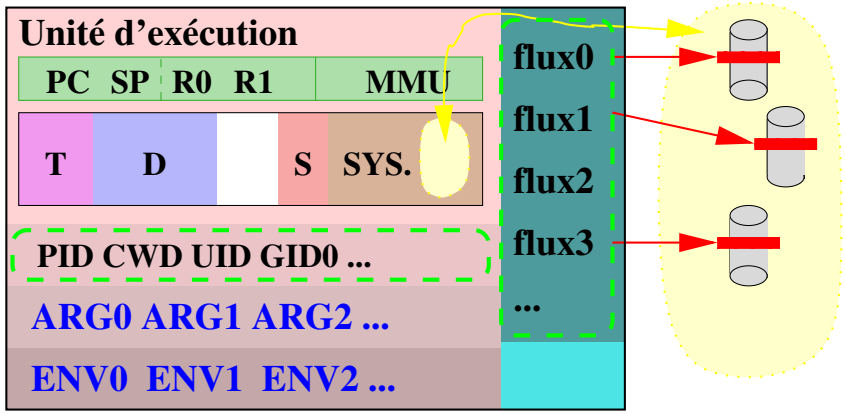
Mmap est le plus complexe, ceci est du au contraintes d'alignement, et l'impossibilité d'ajouter des octets à un fichier mappé.

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 μs).

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

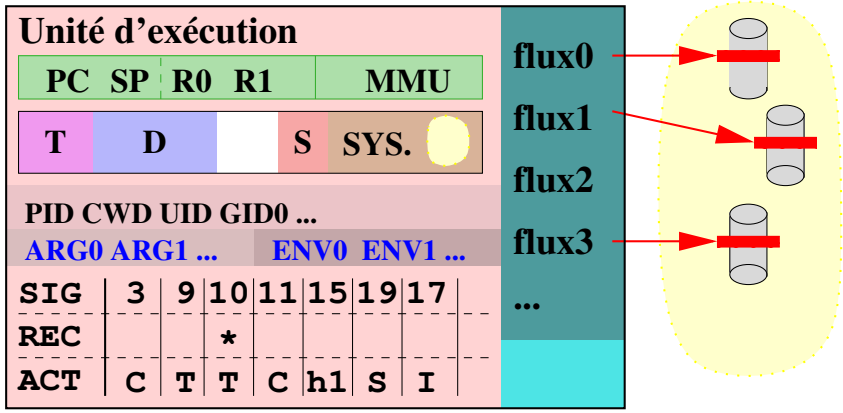
7.1.1 Qu'est un signal ?

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 4 traitements possibles pour un processus qui reçoit un signal :

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

7.1.1.2 Implémentation



- 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 ?

7.1.1.3 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.

7.1.1.4 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 $*$)

7.1.1.5 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 ∅ 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 : μ-secondes).

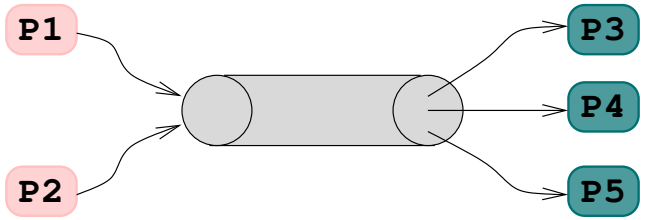
Retour Pause renvoie toujours -1.

Alarm et ualarm renvoie ∅ 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

```c
// ho
int main()
{
  char c;
  int m2s=open(
    "fifo", O_WRONLY);
  int s2m=open(
    "fifo2",O_RDONLY);

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

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

  return 0;
}
```

```c
// ell
int main()
{
  char c;
  int m2s=open(
    "fifo", O_RDONLY);
  int s2m=open(
    "fifo2",O_WRONLY);

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

  return 0;
}
```

7.2.3.2 Solution 2

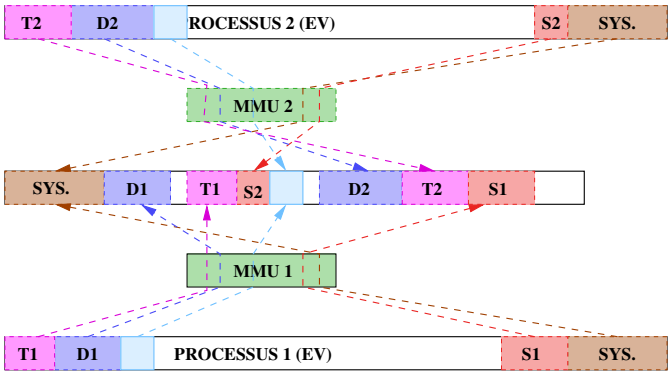
```c
// ho
int main()
{
  char c;

  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).

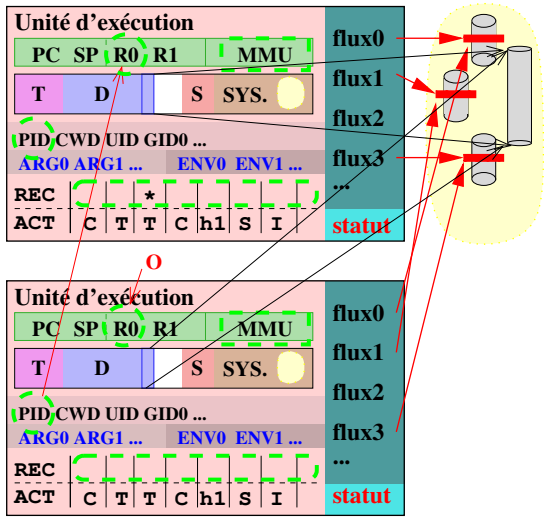
Exemple

```
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  ...
```

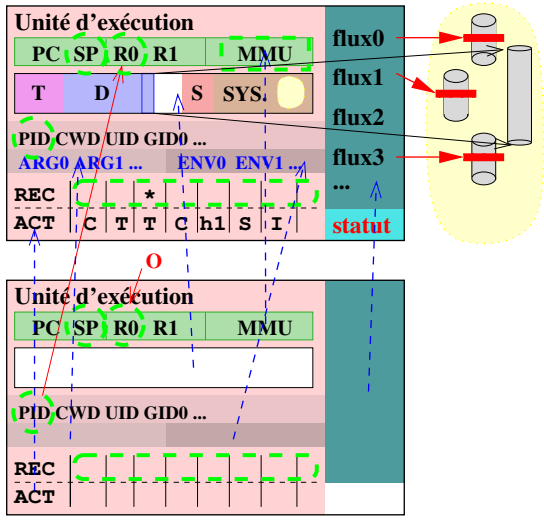
8.1.1.2 Principe

Les mécanismes de duplication des entités d'un processus lourd sont présentés sur la figure 3 (page 46) 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 3 : 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 3 (page 46) 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)

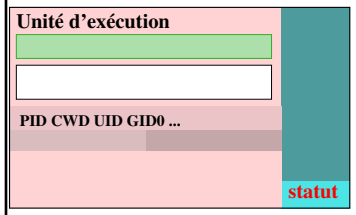
Fonction Attend un événement sur un processus fils, et le code dans status. wait ne traque que la terminaison d'un fils. waitpid traque la terminaison, la suspension ou l'activation du fils.

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 3, page 46) 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

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;
```

```
            a=x;
            V();
        }
        return NULL;
}

int main(int argc, char*argv) {
  pthread_attr_t att;
  pthread_attr_init(&att);

  pthread_t tha, thb;
  pthread_create(&tha,&att,add, (void*)0);
  pthread_create(&thb,&att,add, (void*)1);

  pthread_join(tha,NULL);
  pthread_join(thb,NULL);

  printf("a=%d\n",a);

  return 0;
}
```

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 Tm le processus initial et Ta les autres.

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

getpid() et getppid() Dans un thread Ti donnent celui de Tm.

$\Longrightarrow$ Les processus threads sont masqués

syscall(SYS_gettid) Renvoie le vrai PID du processus.

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

ps -Af Affiche tous les processus Tm.

ps -LAf Affiche tous les processus Tm et Ta.

Gestionnaire de signal Partagé par les Ti.

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

Fork dans un Ti Le processus Ti uniquement est cloné, le père du clone est Tm.

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

$\Longrightarrow$ Tous attendront la mort du clone.

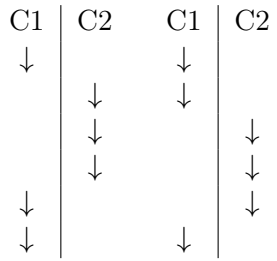
$\Longrightarrow$ Tous sauf 1 recevront -1 avec errno "no child".

8.3 Fonction réentrante et thread-safe

8.3.1 Problème

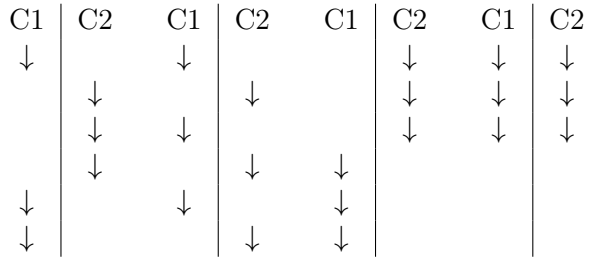
Soit un code C_1 manipulant des données D_1, et un code C_2 manipulant des données D_2. C_1 et C_2 peuvent s'exécuter en :

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>
```