Architecture et Systèmes

Stefan Schwoon

Cours L3, 2026/2027, ENS Paris-Saclay

Utilisateurs dans Unix

Utilisateurs :

- personnes naturelles
- utilisateurs virtuels (admin, “daemons”)

Gestion :

- locale (`/etc/passwd`) dans Unix
- à distance (serveur LDAP), `getent passwd`

$\Rightarrow$ On s’intéressera aux relations avec d’autres concepts (processus, fichiers, . . .)

Identifiants utilisateur et groupe

Un utilisateur . . .

- possède un identifiant numérique (uid) ;
- appartient à un groupe *primaire* ;
- appartient à quelques groupes *supplémentaires*.

commande utile : id

Groupes :

- chaque groupe possède son identifiant numérique (gid)
- gestion locale : voir `/etc/passwd` et `/etc/group`

Utilisateurs et processus

Attributs d'un processus relatifs aux utilisateurs :

- utilisateur réel (getuid)
- utilisateur effectif (geteuid)
- utilisateur sauvegardé (getresuid)
- groupe réel et effectif (getgid, getegid)
- groupes supplémentaires (getgroups)

Ces attributs déterminent des privilèges que possède un processus.

Ces identifiants sont hérités du processus père lors d'un `fork`.

Utilisateur réel et effectif

Utilisateur réel : l'utilisateur pour qui le processus travaille

Utilisateur effectif : pris en compte pour les privilèges

Normalement, ces deux sont identiques !

Exceptions : p.ex., `passwd` (pour changer de mot de passe)

- utilisateur réel : celui qui souhaite changer son mot de passe
- utilisateur effectif : root

Gérer l'utilisateur effectif

`setuid` (voir `showids.c`)

- modifier de façon permanente les privilèges d'un processus
- l'utilisateur effective doit être 0
- change les id réel, effectif et sauvegardé

`seteuid`

- modifier de façon temporaire les privilèges d'un processus
- on change l'utilisateur effectif seulement
- si l'utilisateur effectif est 0, aucune restriction
- sinon, on peut y mettre les valeurs réelles ou sauvegardées

Utilisateurs/groupes et fichiers

Tout fichier appartient à un utilisateur et à un groupe.

Les fichiers créés par un processus portent ses identifiants effectifs.

Droits d'accès sont déterminés par les uid et gid associés avec un processus :

- 9 bits "ugo" : (user,group,others) $\times$ (read,write,execute)
- 3 autres bits : setuid, setgid, sticky

Commandes: `chmod`, `chown`, `chgrp`, `umask`

Droits d'accès : exemple

Supposons qu'un processus souhaite lire dans un fichier.

Si son id effectif utilisateur égale le propriétaire du fichier, on vérifie le bit *(user,read)*.

Si le groupe du fichier est soit égale au groupe effectif du processus, soit dans ses groupes supplémentaires, on vérifie le bit *(group,read)*.

Sinon, on vérifie le bit *(others,read)*.

Pour écrire : c'est l'analogue avec *write*.

Le bit *execute* est utilisé lorsqu'on utilise une fonction de la famille `exec`.

Droits d'accès sur les dossiers

Les bits rwx ont une significance légèrement differente sur les dossiers :

read: on peut obtenir la liste des fichiers du dossier

write: on peut modifier la liste (créer, rénommer, supprimer des fichiers)

execute: on peut obtenir (avec `stat`) les méta-données des fichiers

Setuid et setgid

Les bits setuid/setgid sur les *fichiers*:

> Quand un processus exécute le fichier, ses identifiants effectifs deviennent ceux du fichier.

Le bit setgid sur les *dossiers*:

> Les fichiers crées dans ce dossier portent le gid du dossier (et pas du processus qui les a créés).

Entrées/sorties

Les opérations entrées/sorties transmettent des données entre la mémoire et d'autres processus ou des périphériques.

Les entrées/sorties se font par des fichiers.

plus qu'une collection de données;

structure de données abstraite avec au moins deux opérations (lecture, d'écriture)

Exemples

Un fichier peut représenter des réalisations différentes, p.ex.:

- fichiers sur un disque dur ;
- une zone de mémoire temporaire, p.ex. terminal, tube ;
- représentation des données dans le noyau (`/proc`) ;
- les connections réseau.

Les accès se font uniformement par les mêmes appels système, mais selon le type de fichier le noyau renvoie l'appel à un pilote pour réaliser l'opération.

Aspects de E/S

Stockage des données

système de fichiers

droits d'accès

(organisation physique d'un disque dur)

Gestion au niveau des processus

fonctions pour accéder aux fichiers/créer etc

fonctions pour manipuler les données d'un fichier

Système de fichiers

Unix gère un système de fichiers pour le stockage pérenne des données.

Ce système de fichiers est une arborescence :

- Les nœuds internes sont les dossiers (ou répertoires).
- Les feuilles sont des fichiers (ordinaires ou spéciaux).
- Sur certains nœuds on peut greffer un arborescence supplémentaire (p.ex. une partition, une clé USB) (mount point en anglais).

Voir `mount` pour une liste des systèmes greffés.

Organisation d'un système de fichiers

Les nœuds sont référencés par des chemins:

chemin absolu: en commençant par la racine `/` et suivant les dossiers, p.ex. `/home/schwoon/toto.txt`

chemin relatif on l'interprète en commençant dans un *dossier actuel*, p.ex. `schwoon/toto.txt` si on se trouve dans `/home`.

Dans un chemin relatif, `..` veut dire le dossier en-dessus, `.` le dossier actuel.

Ce dossier actuel est un attribut du processus (modifier avec `chdir`).

Le dossier actuel est hérité par les processus fils.

Les chemins absolus et relatifs sont acceptés par toutes les appels système qui gèrent les fichiers.

Inœuds

Un fichier consiste des données, et on y associe certains meta-données (nom du fichier, propriétaire, droits d'accès etc).

Les inœuds sont une structure de données pour stocker ces méta-données. Une partie d'un disque dur leur est reservée.

Parmi les données stockés dans un inœud, il y a le type de fichier, propriétaire, groupe, droits d'accès, nombre de pointeurs vers cet inœud, les blocs où les données du fichiers sont stockés, . . . , mais pas le nom du fichier.

Relation entre fichiers et inœuds

Un inœud représente une unité de données sur disque; un fichier est une reférence vers un inœud avec un nom.

Pour la plupart des fichiers normaux, cette relation est un-à-un.
Par contre, les dossiers typiquement possèdent plusieurs références.

`ls -i` donne les identifiants du inœud associé avec un fichier ;
`stat` affiche les méta-données d'un inœud.

Organisation d'un dossier

Un dossier est un fichier spécial.

Ses données consistent d'une liste de son contenu, avec pour chaque item :

son nom

son inœud

Du coup plusieurs entrées peuvent référencer le même inœud avec des noms différents.

Un inœud (et ses meta-données) est libéré lorsqu'on supprime son dernier lien (fonction `unlink` dans C).

Liens durs et faibles

Unix connait deux types de liens, tous les deux gérés par `ln`.

Lien dur : `ln foo bar` crée un nouveau fichier `bar` avec le même inœud que `foo`.

Lien faible : `ln -s foo bar` crée un fichier spécial `bar` qui ne contient qu'un pointeur vers un autre chemin (dans ce cas `foo`). Tout accès à `bar` est renvoyé vers ce chemin.

Processus et fichiers

Tout processus peux travailler avec plusieurs fichiers en même temps.
Les principes:

Ouverture: commencement du travail. Le noyau crée un accès au fichier au sein du processus.

Certains fichiers sont typiquement ouverts à la création d'un processus (les stdin/stdout/stderr).

Fermeture: destruction de l'accès (effectué automatiquement à la terminaison du processus).

Tableau des fichiers ouverts

Le noyau détient un tableau des fichiers (TFO) courramment utilisés (*ouverts*) par tous les processus.

Une ligne dans le TFO représente un accès à un fichier (réel ou virtuel). Elle contient des informations telles que :

- fichier/inœud référencé
- mode d'accès (lire et/ou écrire)
- position dans le fichier (si le fichier permet un accès aléatoire)

Gestion des fichiers dans les processus

Chaque processus possède des descripteurs de fichiers
(qui évoluent au fil de son exécution).

Au sein du processus, un descripteur est simplement un entier (0,1,2,. . .).
Tout descripteur pointe dans une ligne du TFO.

Plusieurs descripteurs (au sein de différents processus) peuvent pointer sur une même ligne du TFO. Les descripteurs sont hérités lors d'un `fork`.

Une ligne dans le TFO existe tant qu'il existe un processus qui pointe sur elle.

Descripteurs standard

Par défaut les trois premiers descripteurs d'un processus sont utilisés ainsi :

0 est l'entrée standard (stdin) (p.ex., `getch` ou `scanf` s'en servent)

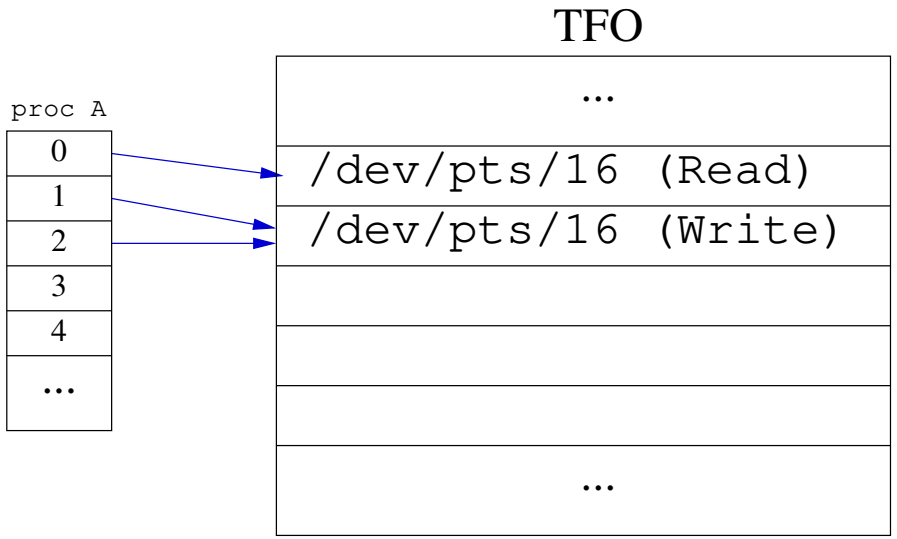
1 est la sortie standard (p.ex., `printf` s'en sert)

2 est la sortie erreur (on est censé y envoyer des messages d'erreur)

Normalement, ils sont liés à un terminal (fichier virtuel nommé `/dev/tty/16` ou similaire).

TFO et descripteurs

État initial typique d'un processus :

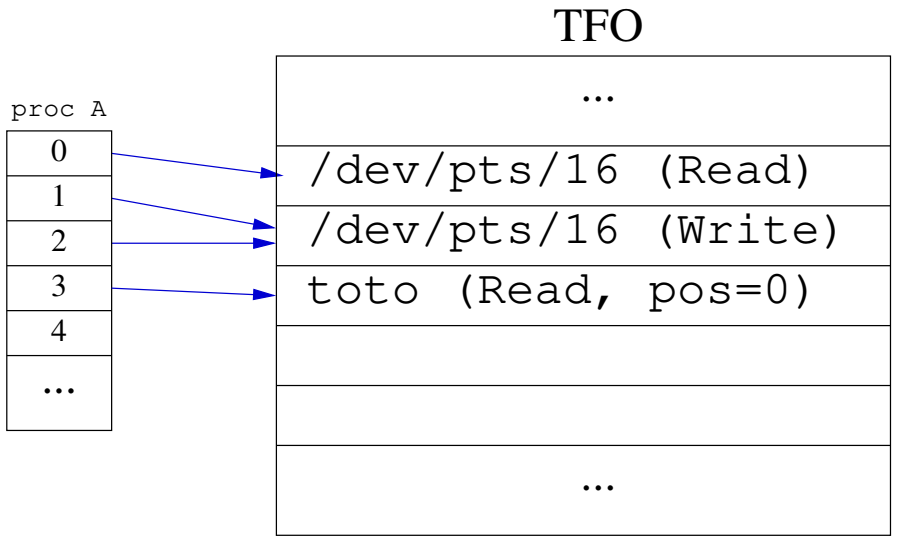


On considère les instructions suivantes :

```
f = open("toto",O_RDONLY);
p = fork();
g = open("tata",O_RDONLY);
if (p) close(f);
```

TFO et descripteurs

État après `open("toto",...)` :

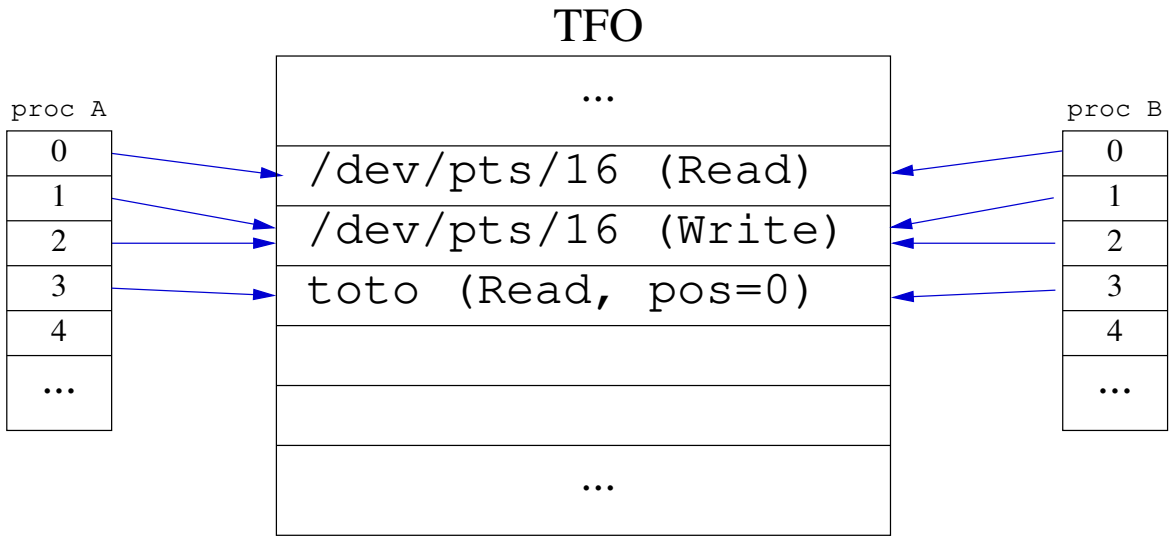


On considère les instructions suivantes :

```
f = open("toto",O_RDONLY);
p = fork();
g = open("tata",O_RDONLY);
if (p) close(f);
```

TFO et descripteurs

État après `fork()` :

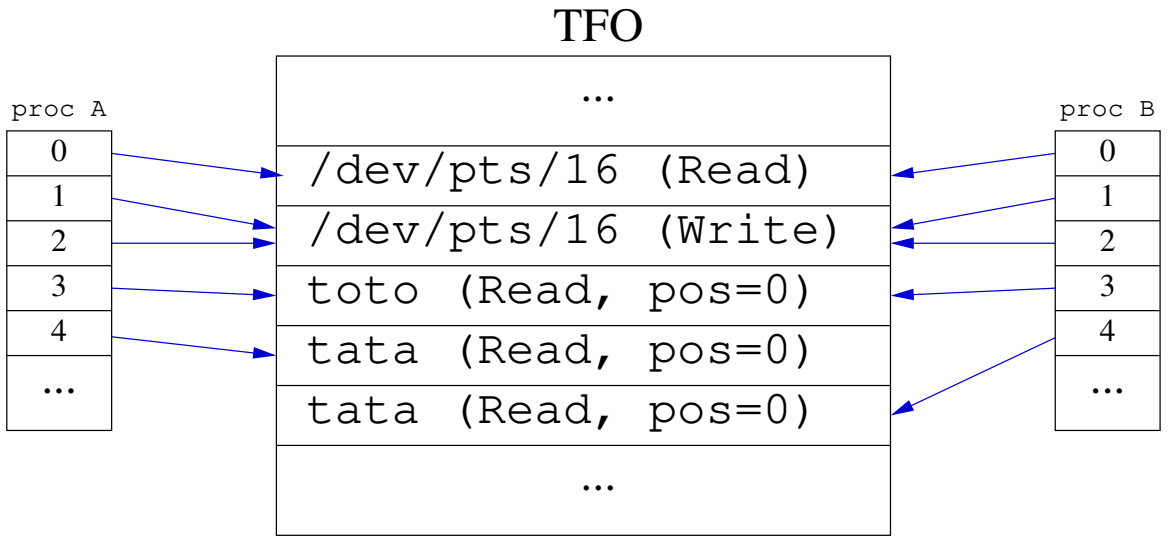


On considère les instructions suivantes :

```
f = open("toto",O_RDONLY);
p = fork();
g = open("tata",O_RDONLY);
if (p) close(f);
```

TFO et descripteurs

État après `open("tata",...)` :

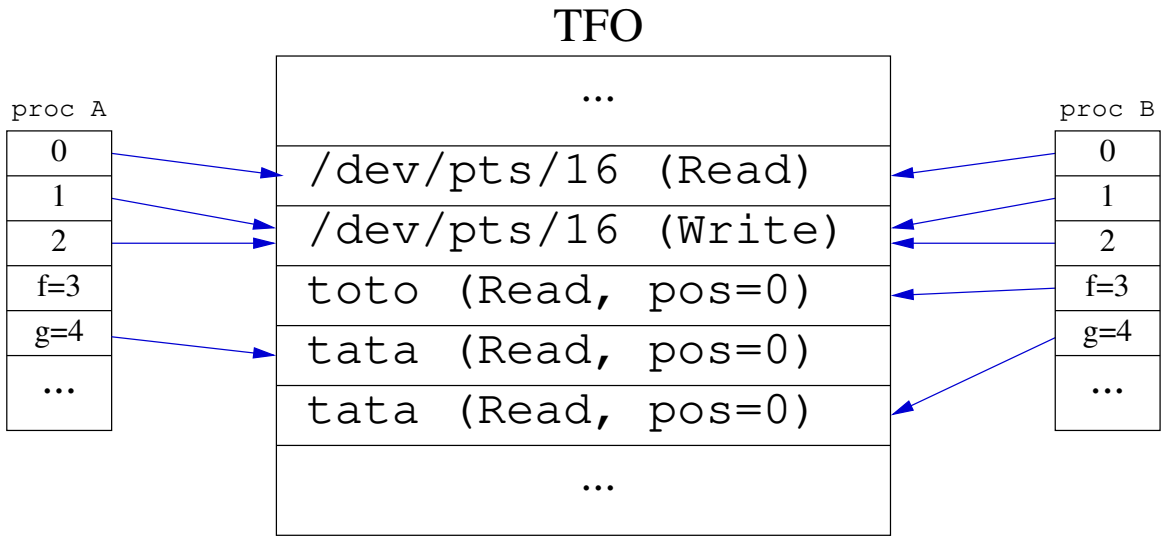


On considère les instructions suivantes :

```
f = open("toto",O_RDONLY);
p = fork();
g = open("tata",O_RDONLY);
if (p) close(f);
```

TFO et descripteurs

État après `close(f)` :



On considère les instructions suivantes :

```
f = open("toto",O_RDONLY);
p = fork();
g = open("tata",O_RDONLY);
if (p) close(f);
```

Quelques fonctions pour gérer les descripteurs

`open` ouvre un fichier existant (pour lire ou écrire).

`creat` crée un nouveau fichier (ou tronque un fichier existant).

`dup` et `dup2` dupliquent un descripteur au sein d'un processus.

`close` supprime le descripteur dans le processus actuel.

`pipe` génère un fichier virtuel avec deux descripteurs (lire/écrire).

Créer un descripteur

`open`: ouvrir un fichier existant. Exemples :

- `open("myfile",O_RDONLY)` : ouvrir fichier en lecture (alternatives: `O_WRONLY`, `O_RDWR`)
- `open("myfile",O_WRONLY | O_CREAT)` : ouvrir pour écrir, créer le fichier s'il n'existe pas
- `open("myfile",O_WRONLY | O_CREAT | O_TRUNC,0666)` : Comme avant, mais détruire ancien contenu s'il en existe ; en plus, spécifier droits d'accès.

`creat`: raccourci pour open avec `O_WRONLY`, `O_CREAT` et `O_TRUNC`

Lecture et écriture

`read/write(fd,p,n)` : lire/écrire *n* octets à partir de l'adresse *p* dans fichier *fd*

`read` et `write` renvoyent les nombres d'octets qui ont été réellement lus/écrits (ce nombre peut être inférieur à *n*). Un renvoi de -1 veut dire erreur.

Il est conseillé de vérifier les valeurs renvoyées en cas d'erreur.

`read` retourne avec 0 si "fin de fichier".

`read` bloque si aucune donnée n'est disponible actuellement, mais il peut encore en arriver dans le futur.

Manipuler des descripteurs

`dup` and `dup2`: recopier un descripteur vers un autre

`g = dup(f)`: créer un nouveau descripteur `g` avec le même comportement que `f`

`dup2(f,g)`: recopier `f` vers `g`, même si `g` est déjà utilisé

`pipe`: créer un conduit unidirectionnel (tube)

```
int p[2]; pipe(p);
```

Les données écrites dans `p[1]` apparaissent dans `p[0]`.

Les tubes

Un tube permet à deux processus d'échanger des données.

La lecture sur un tube soit renvoye toute de suite les données qui sont dédans, soit elle bloque jusqu'à ce que des données arrivent (ou que tous accès en écriture ont été fermés).

Écrire sur un tube qui n'a plus d'accès en lecture donne un signal `SIGPIPE`.

Utilisation dans le shell : `cmd1 | cmd2`

Le shell crée un tube, plus fork deux fois, ferme ses accès au tube, et attend les deux fils.

Le premier fils ferme l'accès en lecture, redirige la sortie standard vers le descripteur écriture, et fait un exec sur `cmd1`.

Le deuxième fils ferme l'accès en écriture, redirige l'entrée standard vers le descripteur lecture, puis fait exec sur `cmd2`.

Configurer une tube

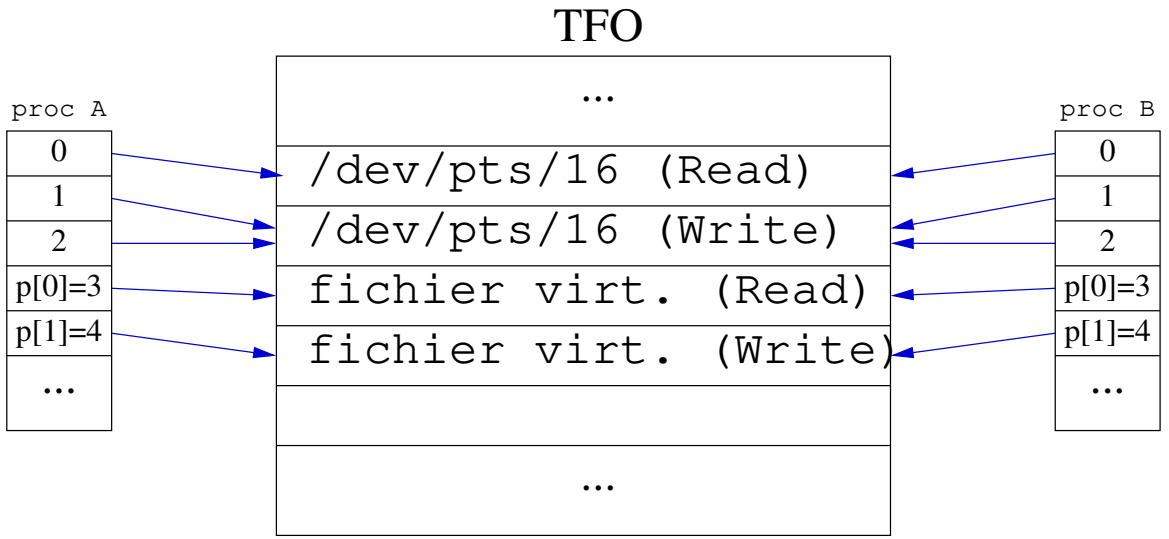
Processus A va créer un fils (proc B) et brancher son entrée standard sur une tube :

```
int p[2];
p = pipe();
if (!fork()) { dup2(p[0],0); close(p[0]); close(p[1]); }
 else { close(p[0]); }
```

Ainsi, tout ce qu'écrit proc A sur `p[1]` apparaît dans l'entrée standard de proc B.

Configurer une tube

État après `fork()` :

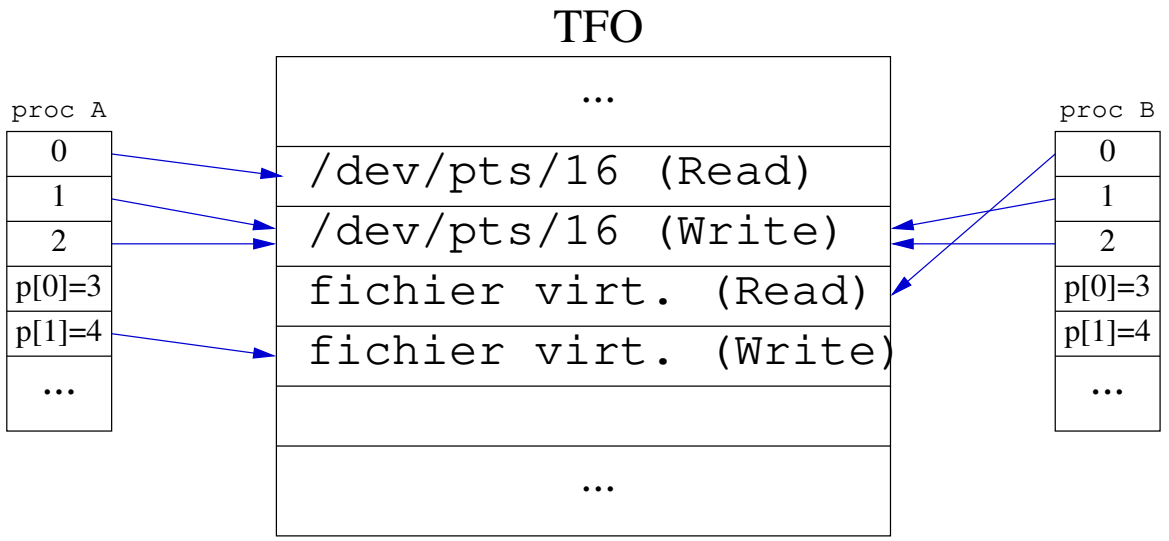


On considère les instructions suivantes :

```
int p[2];
p = pipe();
if (!fork()) { dup2(p[0],0); close(p[0]); close(p[1]); }
 else { close(p[0]); }
```

Configurer une tube

État final :



On considère les instructions suivantes :

```
int p[2];
p = pipe();
if (!fork()) { dup2(p[0],0); close(p[0]); close(p[1]); }
 else { close(p[0]); }
```

Modifier position de lecture/écriture

`lseek` modifie la position dans un fichier où la prochaine opération de lecture/écriture sera effectuée.

Pas disponible sur tous les types de fichier (p.ex. pas sur les tubes).

Syntaxe: `lseek(f,p,m)`, où `m` est l'une des valeurs suivantes :

`SEEK_SET`: se positionner à l'octet numéro *p* (premier octet à 0)

`SEEK_CUR`: avancer position par *p* octets

`SEEK_END`: position relative à la fin du fichier (*p* peut être negatif)

Renvoie la position obtenue (peut être utilisé pour déterminer la taille du fichier).

I/O: Descripteurs et streams

Sous C il existe deux familles de fonctions pour les entrées/sorties :

`open`, `write`, `read`, `...`

Appels système défini par POSIX

travaillent sur les *descripteurs* (0, 1, 2, . . .)

`fopen`, `printf`, `scanf`, `...`

Fonctions utilisateurs défini par le standard ANSI-C

travaillent sur des *stream* (stdin, stdout, stderr, . . .)

plusieurs modes de comportement en sortie : sans tamponnage, tamponnage par bloc, tamponnage par ligne

Ne pas mélanger les appels des deux familles !

Streams en sortie avec tamponnage

Un stream est associé avec un descripteur.

Dans un stream en mode tamponné, les opérations d'écriture ne sont pas directement transmises au descripteur mais on stocke les données dans une zone tampon.

Transmission entre zone tampon et descripteur :

- sur appel de `fflush` ;
- sur ferméture du stream ;
- en mode tamponnage par ligne : sur apparition d'une nouvelle ligne (`\n`) ;
- en mode tamponnage par bloc : quand le tampon est rempli.

Opérations

Le mode de tamponnage peut être modifié par `setvbuf`.

Il existe quelques raccourcis, voir la page man.

Important lorsqu'on :

- fait passer des donnés à un autre processus (par un pipe)
- printf sans nouvelle ligne

`fdopen` crée un stream au-dessus d'un descripteur.