I/O

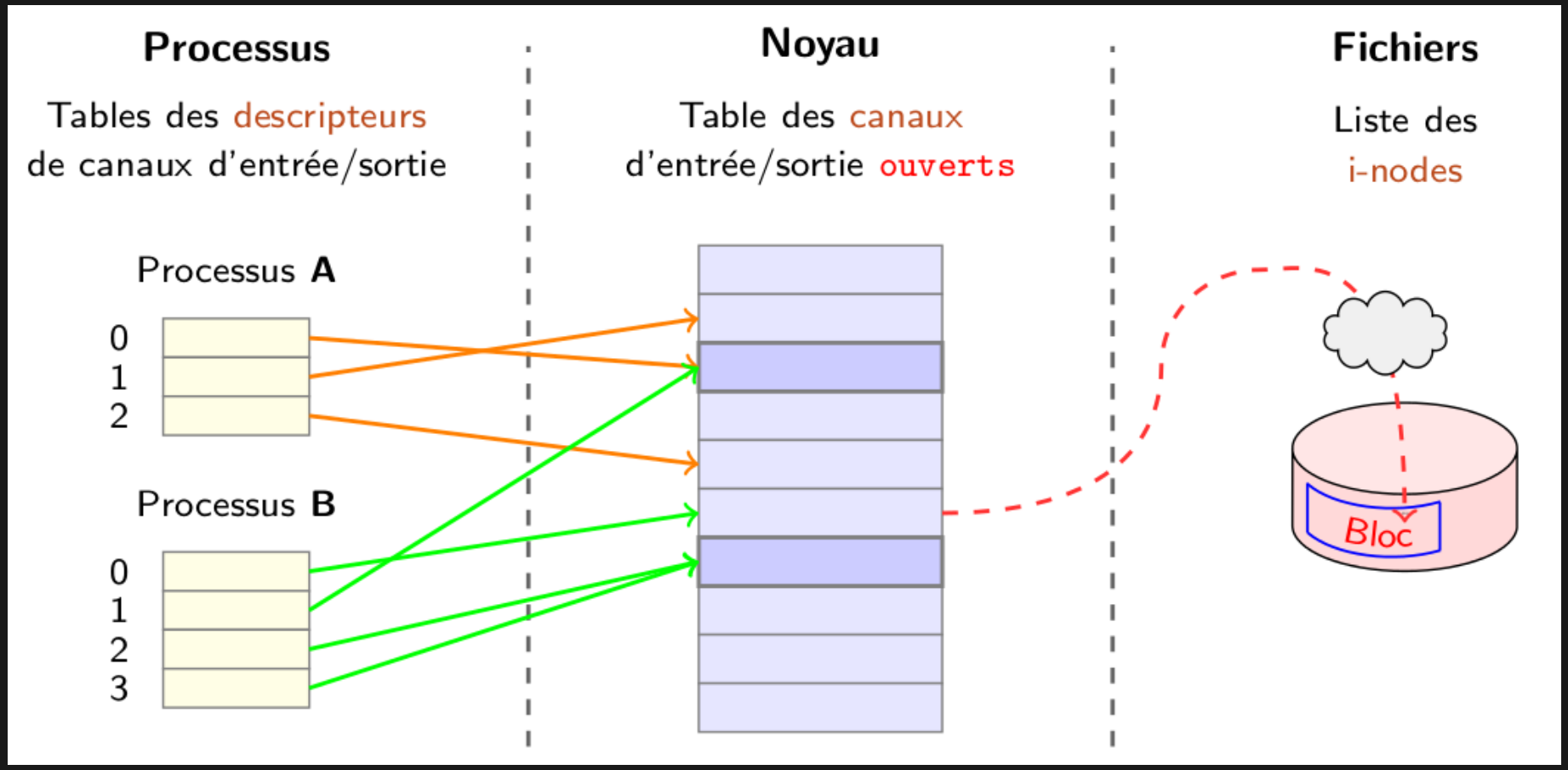
Guillaume Chanel

Remerciements à Jean-Luc Falcone

septembre 2019

CANAUX

CANAUX



Merci à Jacques Menu

DESCRIPTEUR DE CANAL

- Un descripteur de canal d'entrée/sortie est représenté par un entier non-négatif qui correspond à un index dans la *table des canaux ouverts* par le processus.
- Trois descripteurs sont créés au lancement d'un processus:

0	Entrée standard
1	Sortie standard
2	Erreur standard

OUVERTURE DE CANAL

A l'ouverture d'un canal d'entrée/sortie:

- un élément est ajouté à la table des canaux ouverts du noyau.
- un descripteur y est associé (indice).
- un pointeur vers l'élément est introduit dans la table des canaux ouverts du processus à l'indice associé.

Remarques

Un processus peut ouvrir plusieurs canaux vers un même inode:

- Plusieurs éléments distincts seront créés dans la tables du noyau
- Plusieurs descripteurs distincts seront créés.

TABLE DES CANAUX OUVERTS

Un élément de la tables des canaux ouverts contient (entre autre):

- Mode d'accès aux données (`O_RDONLY, O_WRONLY, O_RDWR`)
- L'état du canal (`O_NONBLOCK, O_ASYNC, O_APPEND`)
- Système de fichier et numéro d'inode
- Compteur de références pointant sur l'élément
- Pour les fichiers normaux: la **position actuelle**
- Le bit *close-on-exec*

OPÉRATIONS SUR LES CANAUX

Les mêmes fonctions sont utilisés quelque soit le type de fichier (fichier de données, socket, pipe, périphériques, etc.):

Opération	Appel système
Ouverture	`open`
Lecture de données	`read`
Ecriture de données	`write`
Contrôle du fonctionement	`fcntl`
Fermeture	`close`

OUVERTURE, FERMETURE ET CONTRÔLE

OUVRIR UN CANAL (open)

L'appel système suivant ouvre un canal:

```
int open(const char *pathname, int flags, mode_t mode);
```

- `pathname` est le nom du fichier
- `flags` est un champ de bit indiquant le mode d'accès au fichier
- `mode` indique les permissions si le fichier est créé par `open` (cf. `O_CREAT`).
- Retourne soit le descripteur créé, soit -1 en cas d'erreur.
- Le descripteur retourné est le plus petit descripteur possible.

FLAGS

Les flags suivants peuvent être passés à la fonction open:

`O_RDONLY`	Lecture seule
`O_WRONLY`	Ecriture seule
`O_RDWR`	Lecture et écriture
`O_NONBLOCK`	Canal non-bloqué par une lecture/écriture
`O_APPEND`	Ecrit à la fin
`O_CREAT`	Crée le fichier s'il n'existe pas
`O_TRUNC`	Efface le fichier s'il existe
`O_EXCL`	Erreur si `O_CREATE` est spécifié et que le fichier existe

Remarques

Les trois premiers flags sont particuliers:

- On ne peut pas les combiner entre eux
- On doit en passer au moins un.

MODE (PERMISSIONS)

Lorsque que le flag O_CREAT est passé, il faut spécifier les permissions:

```
int fd = open("/tmp/foo.txt", O_RDWR | O_CREAT | O_OEXCL, 0640);
```

Si O_CREAT est absent, le mode est ignoré.

OUVERTURE MULTIPLE

Si un fichier est ouvert plusieurs fois par un ou plusieurs processus:

- Plusieurs canaux seront créés
- Dans le cas d'un fichier normal, chaque canal aura sa position dans le fichier et éventuellement ses propres buffer
- Toutes les opérations s'effectuent indépendamment et en parallèle

Attention

Le développeur est responsable de coordonner l'accès aux fichiers (cf. verrous).

FERMER UN CANAL (`close`)

- L'appel système suivant permet de fermer un canal ouvert `fd`:

```
int close(int fd);
```

- Le descripteur est liberé et pourra être recyclé (attention)
- Retourne 0 en cas de succès et -1 en cas d'erreur
- Le compteur de référence de l'élément de la table des canaux ouverts est décrémenté
  - s'il atteint 0, l'élément est effacé et les ressources libérées
- Si fd est la dernière indirection vers un nom de fichier effacé par un `link`, le fichier est effectivement effacé.

FERMETURE AUTOMATIQUE D'UN CANAL

- Lors de la terminaison d'un processus par `exit` ou `abort`, le noyau ferme tous les descripteurs (équivalent à `close`).
- Egalement lors d'un appel à `execv()` si le bit *close-on-exec* est égal à `1`.

MANIPULATION DES DESCRIPTEURS (`fcntl`)

On peut manipuler finement un descripteur avec:

```
int fcntl(int fd, int cmd, ... /* arg */ );
```

- où `fd` est un descripteur et `cmd` la commande.
- permet de réserver un fichier (cf. verrous)

LECTURE/ECRITURE

LECTURE: BAS NIVEAU (`read`)

Pour lire les données d'un descripteur, on peut utiliser:

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

- Essaie de lire jusqu'à `count` bytes depuis le descripteur `fd`
- Copie les bytes lus dans `buf`
- Retourne le nombre de bytes **effectivement lus** en cas de succès
- Retourne -1 en cas d'erreur (cf. `errno`)
- Le nombre de bytes retournés peut être plus petit que ce qui est demandé
- **La position avance** d'autant de bytes

ECRITURE: BAS NIVEAU (`write`)

Pour écrire des données sur un descripteur, on peut utiliser:

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

- Essaie d'écrire jusqu'à `count` bytes sur le descripteur `fd`
- `buf` contient les bytes à écrire
- Retourne le nombre de bytes **effectivement écrits** en cas de succès
- Retourne -1 en cas d'erreur (cf. `errno`)
- Le nombre de bytes écrits peut être plus petit que ce qui est demandé
- **La position avance** d'autant de bytes

EXEMPLE (EXAMPLES/COPY.C)

```
Include example there (see script)
```

ACCÈS ALÉATOIRE (`lseek`)

- La position d'un fichier ouvert est à 0
- Elle avance à chaque lecture/écriture
- Si le type de fichier le permet, on peut changer la position avec:

```
off_t lseek(int fd, off_t offset, int whence);
```

- Avance la position du descripteur `fd`
- L'argument `whence` permet d'interpréter l'offset
- En cas de succès retourne la position en byte depuis le début du fichier.
- En cas d'échec, retourne -1 (cf. `errno`)

ACCÈS ALÉATOIRE (2)

La nouvelle position, dépend de `whence` et d'`offset`

whence	Nouvelle position
SEEK_SET	`offset`
SEEK_CUR	position courant + `offset`
SEEK_END	fin du fichier + `offset`

Remarque

Ecrire plus loin que la fin du fichier crée des trous, remplis de `\0`.

EXEMPLE (EXAMPLES/SEEKSTRUCT.C)

```
Include example there (see script)
```

VERROUS

SCÉNARIO

- Un fichier contient le montant disponible de comptes bancaires.
- Les comptes sont écrits séquentiellements

Fonction de retrait

```
int withdraw( int fd, int account, unsigned int amount ) {
    unsigned int before = GET_AMOUNT( fd, account );
    unsigned int after;
    if( before < amount )
        return -1;
    after = before - amount;
    SET_AMOUNT( fd, account, after );
    return after;
}
```

PROBLÈME

- Le compte #10 est crédité de 300 CHF.
- Les deux processus *A* et *B* sont exécutés en **parallèle**.
- *A* appelle:

```
withdraw( accounts, 10, 200 )
```

- *B* appelle:

```
withdraw( accounts, 10, 150 )
```

- Qu'est qui pourrait se passer ?

VERROUS (*LOCKS*)

- Les processus peuvent poser des verrous sur des fichiers pour se coordoner.
- Il y a deux types de stratégie:
  - verrouillage facultatif
  - verrouillage obligatoire

VERROUILLAGE FACULTATIF (*ADVISORY LOCKS*)

- Un verrou facultatif n'est pas pris en compte par les opérations de lecture/écriture.
- Le programmeur peut:
  - Poser un verrou
  - Vérifier si un verrou existe
  - Enlever un verrou
- Un verrou peut porter sur une partie d'un fichier
- Standard POSIX

VERROUILLAGE OBLIGATOIRE (*MANDATORY LOCKS*)

Un verrou obligatoire bloque les opérations de lecture/écriture

Attention

- Non standard (dépend des OS et des sytèmes de fichiers).
- Risque de blocage si un programmeur oublie de dévérouiller.
- Un programme peut être bloqué par son propre verrou.
- Sous GNU/Linux, doit être activé explicitement sur le système de fichiers.

EXEMPLE VERROU

Fonction de retrait

```
int withdraw( int fd, int account, unsigned int amount ) {
    LOCK( fd, account );
    unsigned int before = GET_AMOUNT( fd, account );
    unsigned int after;
    if( before < amount )
        return -1;
    after = before - amount;
    SET_AMOUNT( fd, account, after );
    UNLOCK( fd, account );
    return after;
}
```

TYPES DE VERROUS

Les verrous peuvent être:

partagés plusieurs verrous peuvent être posés en même temps (*shared*)

exclusifs un seul verrou peut être posé à la fois.

Ces deux types permettent de résoudre le *Readers-Writers-Problem*.

STRUCTURE D'UN VERROU (`struct flock`)

man fcntl

```
struct flock {
    ...
    short l_type;    /* Type of lock: F_RDLCK, F_WRLCK, F_UNLCK */
    short l_whence;  /* How to interpret l_start: SEEK_SET, SEEK_CUR, SEEK_END */
    off_t l_start;   /* Starting offset for lock */
    off_t l_len;     /* Number of bytes to lock */
    pid_t l_pid;     /* PID of process blocking our lock (F_GETLK only) */
    ...
};
```

TYPE DE VERROUS (CONSTANTES UTILISÉES)

`F_RDLCK`	plusieurs verrous peuvent être posés en même temps (verrou partagé, *shared*)
`F_WRLCK`	un seul verrou peut être posé à la fois (verrou exclusif, *exclusive*)
`F_UNLCK`	pas de verrou (débloque un verrou existant).

POSITION DU VERROU

`l_whence`	fonctionne comme pour `lseek`
`l_start`	donne le début du verrou (dépend de `l_whence`)
`l_len`	Nombre de "bytes" verouillés

Remarque: peu de lien entre le vérrou et les données vérouillées

- `l_start` ne peut pas pointer avant le début du fichier.
- Le verrou peut dépasser le fichier
- Il n'y a pas vraiment de lien entre la taille du fichier et la plage réservée par le verrou (sauf le début)
- On peut aussi interpréter les offsets+tailles comme des "enregistrements".

POSER/ENLEVER UN VERROU (`fcntl`)

L'appel système suivant permet de poser/enlever les verrous:

```
int fcntl(int fd, int cmd, struct flock *lock);
```

- `fd` spécifie le descripteur du fichier.
- Le comportement varie selon la commande (`cmd`) utilisée
- Le pointeur `lock` permet de passer ou de recevoir les paramètres du verrou.

COMMANDE F_SETLK

- La commande F_SETLK permet de poser/enlever un verrou.
- L'action dépend de `lock->l_type`:

  `F_RDLCK` essaie de poser un verrou partagé

  `F_WRLCK` essaie de poser un verrou exclusif

  `F_UNLCK` enlève un verrou précédent

- Retourne immédiatement
- Si un conflit empêche de poser le verrou retourne -1 et produit les erreurs EACCES ou EAGAIN.

COMMANDE F_SETLK: LOCK

```
int LOCK( int fd, int account ) {
    struct flock fl;
    fl.l_type = F_WRLCK;
    fl.l_whence = SEEK_SET;
    fl.l_start = account;
    fl.l_len = 1;
    if (fcntl(fd, F_SETLK, &fl) == -1) {
        if (errno == EACCES || errno == EAGAIN) {
            //Attendre ou faire autre chose
            return LOCK( fd, account );
        } else return -1;
    } else return 0;
}
```

COMMANDE F_SETLK: UNLOCK

```
int UNLOCK( int fd, int account ) {
    struct flock fl;
    fl.l_type = F_UNLCK;
    fl.l_whence = SEEK_SET;
    fl.l_start = account;
    fl.l_len = 1;
    return fcntl(fd, F_SETLK, &fl);
}
```

COMMANDE F_SETLKW

- La commande F_SETLKW permet de poser/enlever un verrou.
- L'action dépend de `lock->l_type`:

  F_RDLCK essaie de poser un verrou partagé

  F_WRLCK essaie de poser un verrou exclusif

  F_UNLCK enlève un verrou précédent

- **Attends en cas de conflit**
- Si un signal est capturé l'appel se termine avec un retour de -1 et produit l'erreur EINTR.

COMMANDE F_SETLKW: LOCK

```
int LOCK( int fd, int account ) {
    struct flock fl;
    fl.l_type = F_WRLCK;
    fl.l_whence = SEEK_SET;
    fl.l_start = account;
    fl.l_len = 1;
    return fcntl(fd, F_SETLKW, &fl);
}
```

COMMANDE F_GETLK

- La commande F_GETLK permet d'obtenir des informations sur un verrou.
- La structure `*lock` doit être remplie avec les informations concernant un verrou que l'on souhaite poser.
- Elle est modifiée après l'appel:
  - Si le verrou peut être placé, `lock->l_type` vaut F_UNLCK
  - Si le verrou ne peut être placé, `*lock` contient des informations sur un des verrous déjà en place.
- Elle retourne immédiatement.

PROBLÈMES POSSIBLES

Attention

- Tous les verrous posés par un processus sont automatiquement enlevés lorsque le descripteur est fermé.
- Tous les descripteurs d'un processus sont fermés à sa terminaison, donc tous les verrous sont automatiquement enlevés.
- Ne sont pas réentrants.
- Nécéssite la coopération de tous les processus…

LOCK FILES

- Approche alternative: représenter les verrous comme des fichiers
- On peut utiliser `open` avec les flags `O_CREAT | O_EXCL`.
- Il ne faut pas oublier des les enlever.
- Survit à la terminaison du processus.

Attention

- En cas de crash, le fichier subsistera…
- Problème affectant (entre autre) *Mozilla Firefox*

LOCK FILES: EXEMPLE

```c
#define LOCK_FILE ".lock"

int LOCK() {
    int fd = open( LOCK_FILE, (O_CREAT|O_EXCL), 0600 );
    if( fd < 0 && errno == EEXIST ) {
        //ATTENDRE
        return LOCK();
    }
    return fd;
}

int UNLOCK( int fd ) {
    close(fd);
    return unlink(LOCK_FILE);
}
```

FICHIERS TEMPORAIRES

FICHIERS TEMPORAIRES (*TEMPORARY FILES*)

- Un programme peut utiliser des *fichier temporaires*, par exemple pour:
  - Soulager la mémoire vive
  - Téléchargement partiel
  - Communication entre processus

Problème

- Il faut créer un nom unique pour éviter d'écraser d'autres fichiers existant.
- Il faut être sûr que le fichier soit effacé lorsque l'on quitte les processus.

CRÉER ET OUVRIR UN FICHIER TEMPORAIRE (`mkstemp`)

La fonction `mkstemp` permet de créer et d'ouvrir un fichier temporaire:

```
int mkstemp(char *template);
```

- Il faut passer un modèle pour le nom du fichier (`*template`), terminé par 6 "X".
- En cas de succès, la chaîne `*template` est modifiée avec le vrai nom du fichier.
- Le fichier est créé avec les permissions `0600`
- Retourne un descripteur de fichier ouvert en cas de succès (-1 sinon).

EXEMPLE mkstemp

```c
char name[15] = "";
int fd = -1;
strncpy( name, "/tmp/ed.XXXXXX", sizeof name );
fd = mkstemp( name );
if( fd < 0 ) {
    //Gerer l'erreur
}
else {
    printf( "The temporaray filename is %s\n", name );
}
```

EFFACER AUTOMATIQUEMENT

- **Rappel:**
  - un `link` n'efface pas un fichier, seulement son nom
  - Le fichier est effacé s'il n'y a pas de descripteur ouvert.
- On peut alors immédiatement appeler un `link` sur un fichier temporaire créé.
- Le processus peut toujours interragir avec le fichier via le descripteur.
- A la fermeture du descripteur (ou à la terminaison du processus) le fichier est effacé.
- Aucun autre processus ne peut y accéder.

EXEMPLE mkstemp/unlink/link (PSEUDOCODE)

```
int fd = mkstemp( name );
unlink( fd );           // Enleve le nom
DOWNLOAD_INTO( fd );
PLAY_WITH( fd );
close( downloadFD );    // Efface les donnees
```

RÉPERTOIRES TEMPORAIRES (`mkdtemp`)

On peut aussi créer des répertoires temporaires:

```
char *mkdtemp(char *template);
```

- Le template suit les mêmes règles que pour `mkstemp`.
- Le répertoire créé a les permissions `0700`.
- En cas de succès retourne le template (modifié).
- En cas d'écher retourne `NULL`.