

Programmation Système (INF 355)

Marc Zeitoun

Université Bordeaux 1

Année 2008-2009, Licence semestre 5

Organisation de l'UE

Enseignants :

- ▶ Aurélien Esnard, Raymond Namyst, Pierre-André Wacrenier (2 groupes), Marc Zeitoun (cours).
- ▶ Nom.Prenom@labri.fr.
- ▶ **Cours** : 15 séances 1h20
- ▶ **TD** : 10 séances 1h20
- ▶ **TP** : 15 séances 1h20
- ▶ **Web** : <http://dept-info.labri.fr/ENSEIGNEMENT/prs/>

Modalités de contrôle





▶ 1ère session

- ▶ **CC** 1 projet + 1 DS 1h30
- ▶ **Examen** 1h30
- ▶ Note finale $\max(\text{Ex}, 3/4\text{Ex} + 1/4\text{CC})$

▶ 2ème session

- ▶ **Examen 2** 1h30
- ▶ Note finale $\max(\text{Ex}2, 3/4\text{Ex}2 + 1/4\text{CC})$

Un choix bibliographique

- ▶ Programmation système sous Unix (ce cours) :
 - ▶  **Richard W. Stevens.** *Advanced Programming in the Unix Environment.* Addison-Wesley, 1992.
 - ▶  **Jean-Marie Rifflet et Jean-Baptiste Yunès.** *UNIX : Programmation et communication.* Dunod - 01 Informatique, 2003.
- ▶ Pour aller plus loin, nombreux livres sur les principes des systèmes, e.g.
 - ▶  **Andrew Tanenbaum.** *Systèmes d'exploitation.* Pearson, 2003. ou sur telle ou telle implémentation...
 - ▶  **Marshall Kirk McKusick et al.** *Conception et implémentation du système 4.4BSD* Addison-Wesley, 1997.

Objectifs du cours

- ▶ Apprendre la programmation **avec le** système.
 - ▶ Comment créer des applications qui communiquent avec l'OS...
 - ▶ ... ou avec d'autres applications, en se synchronisant correctement.
- ▶ Comprendre les principes du fonctionnement d'un système.
 - ▶ Que se passe-t-il quand je tape 1s ?
 - ▶ Quand je tape Ctrl-C, Ctrl-S ou Ctrl-Z ?
 - ▶ Comment sont gérés les processus, sont stockées les données sur disque ?
 - ▶ Les algorithmes et la programmation **du** système : pour plus tard.
- ▶ Comprendre des problèmes typiques posés par le partage de ressources.
 - ▶ Comment garantir qu'un fichier n'est écrit que par un processus à la fois ?

Prérequis

- ▶ Maîtrise du langage C.
 - ▶ En particulier les pointeurs, l'allocation dynamique...
 - ▶ Également les manipulations bit à bit (|, &, ~).
- ▶ Utilisation courante d'Unix au niveau utilisateur.
 - ▶ Gestion des fichiers, des processus.

Introduction

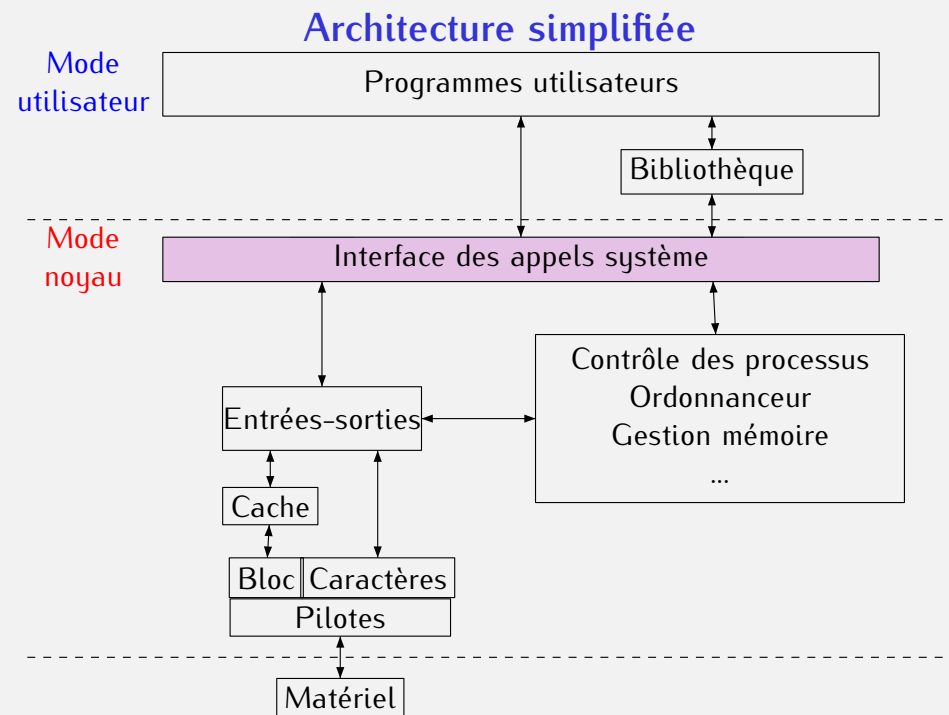
Interaction avec le système

- ▶ Un utilisateur a, sans nécessairement en avoir conscience, une interaction avec le système.
- ▶ Il l'utilise :
 - ▶ lui demande la création de processus, les suspend, les termine,
 - ▶ crée, modifie, efface des fichiers,
 - ▶ imprime, communique avec sa webcam, etc.
- ▶ Le système crée les processus de l'utilisateur, leur alloue de la mémoire.
- ▶ Pour fournir l'illusion que plusieurs processus tournent en parallèle (même en mono-processeur), le système interrompt et reprend les processus.

Rôle d'un système d'exploitation

Un système d'exploitation (Operating System, OS) est un programme

- ▶ permettant aux applications tournant sur la machine de partager les ressources : CPU, fichiers, mémoire...
- ▶ permettant de communiquer avec l'extérieur : terminaux, disque...
- ▶ garantissant l'intégrité de la machine.
- ▶ masquant au programmeur les détails matériels
 - ▶ Facilite l'écriture de logiciel.
 - ▶ Facilite leur portabilité.
- ▶ implémentant certaines abstractions, comme le modèle des processus.

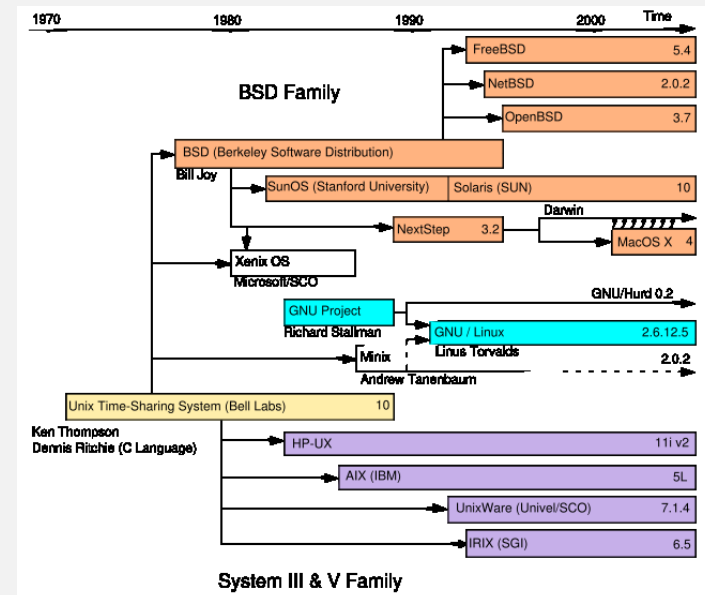


Modes d'exécution et appels système

- ▶ Le processeur possède deux modes de fonctionnement.
 - ▶ mode **utilisateur** (ou *user*, ou protégé) : accès au matériel impossible, accès interdit à certaines zones mémoires et certaines instructions.
 - ▶ mode **noyau** (ou *kernel*, ou moniteur, ou privilégié).
- ▶ Les interruptions provoquent le passage en mode noyau. Au retour de l'interruption, retour au mode utilisateur.
- ▶ Les **appels système** provoquent une interruption et font ainsi passer en mode noyau.
- ▶ On réalise donc des opérations privilégiées en faisant un **appel système**.
 - ▶ C'est la seule façon de réaliser des opérations privilégiées.
 - ▶ Cette interface bien définie permet au système de ne basculer en mode noyau que de façon **contrôlée**.

☹ Un appel système est coûteux (plusieurs centaines de cycles).

Généalogie Unix (très simplifiée)



Norme POSIX IEEE 1003.1

- ▶ Pourquoi un cours basé sur Unix et non Windows ?
 - ▶ code disponible,
 - ▶ principes généraux identiques,
 - ▶ normalisation.
- ▶ POSIX.1 : Portable Operating System Interface IEEE 1003.1.
- ▶ Norme de l'IEEE spécifiant les services principaux d'un OS.
- ▶ Définit les comportements d'une collection de fonctions permettant l'accès au système.
- ▶ Peut-être implémentée
 - ▶ soit par un appel système,
 - ▶ soit par un appel de bibliothèque.
- ▶ Plusieurs mises-à-jour : POSIX.1-1990, POSIX.1-1996, POSIX.1-2001.


Norme POSIX IEEE 1003.1-2001

- ▶ Mise à jour POSIX.1-2001 téléchargeable sur www.unix.org (aussi appelée SUSv3, Single Unix Specification v.3)
- ▶ Normalement en accord avec C99.
- ▶ 2 niveaux de conformité :
 - ▶ Conformité POSIX.
 - ▶ Extension XSI : optionnel.
- ▶ Constantes permettant de contrôler quelle norme respecte un source
 - ▶ `_POSIX_C_SOURCE`. La valeur 200112L correspond à POSIX.1-2001.
 - ▶ `_XOPEN_SOURCE`. Une valeur de 600 permet la compatibilité C99, et POSIX.1-2001 avec extensions XSI (X/Open System Interfaces).

Limites

- ▶ Plusieurs constantes (limites) sont définies par l'implémentation.
- ▶ Par ex. la longueur maximale d'un nom de login, le nombre maximal de fichiers qu'un processus peut ouvrir simultanément.
- ▶ POSIX essaie de fournir des façons portables de tester ces constantes.
- ▶ Les limites peuvent être détectées
 - ▶ à la compilation \rightsquigarrow fichier en-tête `limits.h`, ou
 - ▶ à l'exécution : `sysconf`, `pathconf`.

Appels système et erreurs

- ▶ Il est important de tester si les appels système réussissent ou pas.
- ▶ Pour les fonctions renvoyant un entier, l'échec d'un appel système se matérialise par une valeur retour de `-1`.
- ▶ La variable `extern int errno` est affectée par les appels système ayant échoué. Sa valeur, non nulle, détermine le type de l'erreur.
- ▶  Un appel système réussi laisse `errno` à sa valeur précédente.
- ▶ La fonction `void perror (const char *s)` ; affiche un diagnostic dépendant de la valeur de `errno`.
- ▶ Le programme `strace` permet de lister les appels systèmes d'une commande.
- ▶ Les appels système sont décrits en section 2 du manuel, 3p pour POSIX.

Les entrées/sorties

Les entrées-sorties

- ▶ Les lectures/écritures sont appelées entrées/sorties.
- ▶ Une entrée/sortie provoque une interruption. Le processus appelant est mis en attente (**sommeil**) le temps que sa demande soit traitée.
- ▶ Pour accélérer les entrées sorties, les systèmes utilisent un mécanisme de cache : lorsqu'un processus fait une écriture, elle est réalisée en RAM et non sur le périphérique réel, dans une zone appelée **buffer cache**.
- ▶ Le buffer cache est périodiquement **synchronisé** avec le périphérique.
- ▶ Ce mécanisme est transparent pour le programmeur, à part qu'au retour d'une écriture, les données ne sont pas nécessairement écrites sur le périphérique.

Descripteurs

- ▶ Avant de réaliser une E/S, un processus doit acquérir les droits en mode
 - ▶ lecture, ou
 - ▶ écriture, ou
 - ▶ lecture-écriture.
- ▶ Il le fait en demandant une **ouverture** du fichier, dans un des 3 modes.
- ▶ En cas de succès, le système fournit au programmeur un identifiant entier, appelé **descripteur** permettant les accès ultérieurs.

Descripteurs

- ▶ Un descripteur obtenu par un processus lui est propre.
- ▶ Il donne au processus le droit d'accès à une entrée dans une « **table des fichiers ouverts** » (**TFO**).
- ▶ Un même fichier peut être ouvert plusieurs fois (dans des modes éventuellement différents) par un ou plusieurs processus.
- ▶ 3 descripteurs spéciaux :
 - ▶ **STDIN_FILENO**, entrée standard,
 - ▶ **STDOUT_FILENO**, sortie standard,
 - ▶ **STDERR_FILENO**, sortie erreur standard.

Ouverture d'un fichier

- ▶ L'ouverture d'un fichier par un processus est demandée par l'appel

```
#include <sys/stat.h> // pour le mode d'ouverture.
#include <fcntl.h>

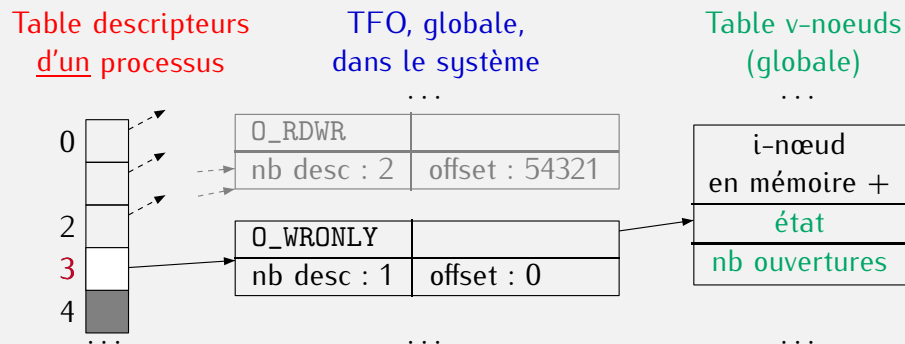
int open(const char *path, int flag, ... /* mode */ );
```

- ▶ 1er argument : chemin d'accès au fichier.
- ▶ 2ème argument : mode d'ouverture. **OU** bit à bit contenant
 - ▶ l'une des constantes `O_RDONLY`, `O_WRONLY`, `O_RDWR`.
 - ▶ des options supplémentaires parmi
 - ▶ `O_APPEND` les écritures se feront en fin de fichier,
 - ▶ `O_CREAT` création du fichier s'il n'existe pas,
 - ▶ `O_TRUNC` tronque le fichier s'il existe,
 - ▶ `O_EXCL` échec si création demandée et le fichier existe,
 - ▶ `O_SYNC` attente que l'écriture soit réalisée physiquement.
 - ▶ + ...
- ▶ 3ème argument optionnel : mode de création (modifié par `umask`).

Ouverture : structures mises en jeu

- ▶ En cas de succès, `open` renvoie un descripteur entre 0 et `OPEN_MAX`.
- ▶ Cet entier identifie l'ouverture du fichier par le processus.
- ▶ Il permet au noyau d'accéder à une entrée de la TFO précisant
 - ▶ le mode d'ouverture du fichier,
 - ▶ le nombre de descripteurs pointant sur l'entrée.
 - ▶ le décalage du **curseur** de lecture/écriture par rapport au début (offset),
 - ▶ un pointeur vers un **i-nœud** en mémoire.
- ▶ Un **i-nœud** est une structure maintenue par le système.
- ▶ Cette structure contient des informations sur le fichier (propriétaire, droits, etc.) et le moyen d'accéder aux données.

Ouverture : structures mises en jeu



Ouverture d'un fichier

L'ouverture d'un fichier par `open()` provoque

1. l'allocation d'un **descripteur** de fichier.
2. l'allocation d'une entrée dans la **table des fichiers ouverts (TFO)**.
3. l'allocation éventuelle d'une entrée dans la table des **v-nœuds**, avec récupération du **i-nœud** sur disque s'il n'est pas dans le cache.
4. les vérifications de **droits d'accès** au fichier selon le mode demandé.
5. le rejet éventuel d'opérations illégales (ouverture d'un répertoire, ou ouverture en écriture d'un exécutable ouvert en cours d'exécution).
6. la création éventuelle du fichier s'il n'existe pas et `O_CREAT` est demandé.
7. si le fichier régulier existe et peut être ouvert en mode écriture ou lecture-écriture, et si `O_TRUNC` est demandé, le fichier est tronqué
↪ libération éventuelle des blocs.
8. l'initialisation du fichier ouvert (offset, pointeur sur v-nœud, mode d'ouverture et compteur de références).

Atomicité

- ▶ Pour un processus P1, ouvrir un fichier en mode `O_CREAT | O_EXCL` n'est pas équivalent à
 1. tester s'il existe
 2. si oui, ne rien faire. Sinon, l'ouvrir avec le mode `O_CREAT`,
- ▶ Dans le second cas, scénario suivant possible :
 - ▶ P1 constate que le fichier n'existe pas.
 - ▶ P2 crée le fichier et y écrit.
 - ▶ P1, qui a déjà fait le test, ouvre le fichier bien qu'il existe.
- ▶ Mode `O_CREAT | O_EXCL` : [test et création] **atomique** : garantie de **non interruption** entre **test** et **création**.

Fermeture d'un fichier

- ▶ Lorsqu'un processus a fini d'utiliser un fichier, il utilise

```
#include <unistd.h>

int close(int descr);
```
- ▶ L'appel désalloue
 - ▶ le descripteur,
 - ▶ éventuellement l'entrée de la TFO si le compte de descripteurs pointant sur l'entrée passe à 0,
 - ▶ éventuellement l'i-nœud en mémoire si le nombre d'ouvertures passe à 0,
 - ▶ éventuellement les blocs du fichier si le nombre de liens et le nombre d'ouvertures est 0.
- ▶ Un processus qui se termine ferme implicitement tous ses descripteurs.
- ▶ On verra certains cas où il est **crucial** de ne pas oublier `close()`.

Tête (ou curseur) de lecture/écriture

- ▶ Pour lire ou écrire dans un fichier, tout se passe comme si on avait une « **tête (ou curseur) de lecture-écriture** ».
- ▶ La tête sera déplacée automatiquement à chaque lecture ou écriture. (tout se passe comme sur une bande magnétique).

Déplacement du curseur de lecture/écriture

- ▶ Le curseur de lecture et/ou écriture est associé à un **fichier ouvert**.
- ⚠ Un même fichier peut être ouvert plusieurs fois, avec des curseurs à des places différentes.
- ▶ La position du curseur est mémorisée par un entier ≥ 0 dans l'entrée de la TFO.
- ▶ À l'ouverture, il est positionné à :
 - ▶ la taille du fichier en mode `O_APPEND`;
 - ▶ 0 sinon.
- ▶ Sur les fichiers ordinaires, on peut demander un déplacement du curseur.
- ▶ Ce n'est pas possible sur tout fichier (ex. terminal).

Déplacement du curseur de lecture/écriture

- ▶ Pour déplacer, si possible, le curseur associé à un descripteur `descr` :

```
#include <unistd.h>
```

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

- ▶ Si `whence` vaut
 - ▶ `SEEK_SET`, déplacement 2^{ème} argument ≥ 0 relatif au **début** de fichier.
 - ▶ `SEEK_CUR`, déplacement 2^{ème} argument relatif à la **position courante**.
 - ▶ `SEEK_END`, déplacement 2^{ème} argument relatif à la **fin de fichier**.
- ▶ L'appel retourne la nouvelle position.
- ▶ On peut positionner le curseur après la fin de fichier.

Lecture

- ▶ La lecture sur un fichier peut se faire par la fonction

```
#include <unistd.h>
```

```
ssize_t read(int descr, void *buf, size_t nbytes);
```

- ▶ Lecture via le descripteur `descr` de `nbytes` octets, à placer dans `buf`.
 1. Lecture à partir de la position (`offset`) correspondante dans la TFO.
 2. Avancement du curseur du nombre d'octets lus,
- ▶ Lecture atomique : ce qui est lu n'est pas entrelacé avec une autre E/S.
- ▶ Retourne le nombre d'octets effectivement lus.
- ▶ `read` renvoie donc moins que `nbytes` si la fin de fichier est atteinte.
- ▶ Comportement sur tubes/sockets : vu plus tard.

Écriture

- ▶ L'écriture dans un fichier par la fonction

```
ssize_t write(int desc, const void *buf, size_t nbytes);
```

- ▶ demande l'écriture des `nbytes` premiers octets de `buf` via `descr`.
 - ▶ à partir de la position courante,
 - ▶ avec écrasement des anciennes données (n'insère pas) ou
 - ▶ extension du fichier si la fin est rencontrée.
- ▶ Retourne le nombre d'octets effectivement écrits.
- ▶ Un retour inférieur à `nb` peut se produire.
- ▶ **Exercice** : commandes `cat`, `cp`.
- ▶ **Remarque** : on peut écrire des données autres que chaînes (codages d'entiers, structures, etc.). Attention au codage (big/little endian, alignement).

Taille I/O et efficacité

- ▶ Si on doit faire plusieurs lectures ou écritures, la taille du buffer influe notablement sur les performances.
- ▶ Les écritures en mode synchrone sont beaucoup plus lentes.
- ▶ **Exercice** Mettre ceci en évidence.

Atomicité

- ▶ Ouvrir un fichier en mode `O_APPEND` n'est pas équivalent à
 - ▶ l'ouvrir sans le mode `O_APPEND`,
 - ▶ avant chaque écriture, déplacer le curseur en fin de fichier.
- ▶ Dans le second cas, le scénario suivant possible :
 - ▶ Fichier ouvert en mode `O_WRONLY` par 2 processus P1, P2.
 - ▶ P1 déplace le curseur en fin de fichier pour y écrire.
 - ▶ P2 idem.
 - ▶ P2 écrit en fin de fichier.
 - ▶ P1 écrit, mais **pas en fin de fichier!** (écrase l'écriture de P2).
 - ▶ P1 et P2 pensent avoir écrit en fin de fichier. Ce n'est pas le cas pour P1.
- ▶ En mode `O_APPEND`, [déplacement du curseur + et écriture] **atomique** : garantie de **non interruption** entre déplacement du curseur et écriture.

Duplications et redirections

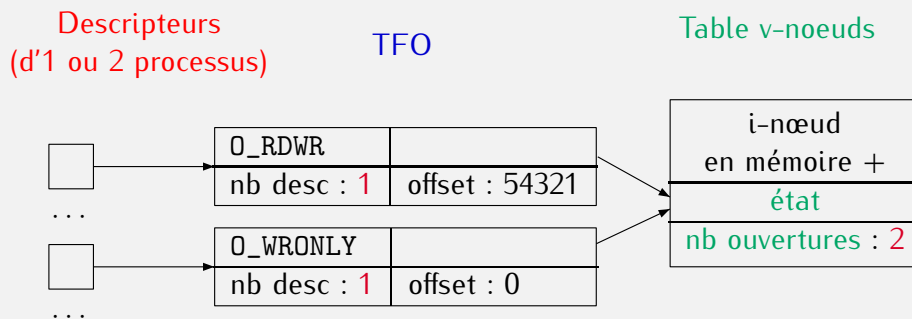
- ▶ L'objectif de la TFO est de créer une indirection, pour permettre à plusieurs processus de partager le même curseur.
- ▶ Plusieurs moyens de créer ce partage.
 - ▶ Quand un processus en crée un autre, il lui « transmet » ses descripteurs.
 - ▶ On peut demander **explicitement** de dupliquer un descripteur par `dup2`.
 - ▶ ...

```
int dup2(int old_descr, int new_descr);
```

- ▶ `old_descr` doit correspondre à un descripteur ouvert.
- ▶ La fonction
 1. ferme le descripteur `new_descr` s'il est ouvert,
 2. le fait pointer vers la même entrée de la TFO que `old_descr`.
 3. retourne la valeur de `new_descr` (-1 si erreur).
- ▶ Les appels `dup` et `fcntl` permettent aussi de dupliquer.

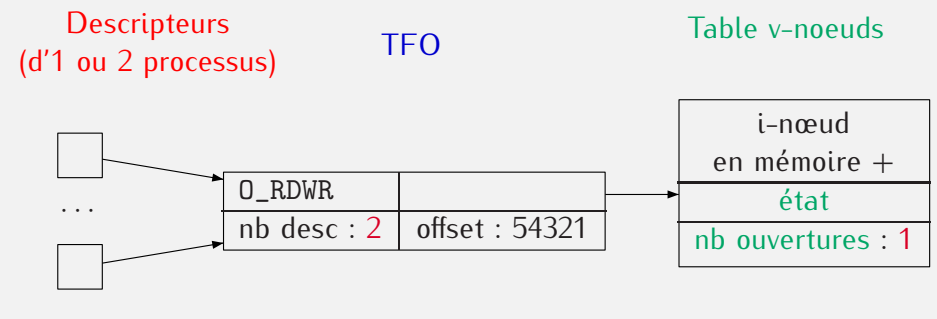
Duplications et redirections

- ▶ Deux ouvertures **indépendantes** du même fichier.



Duplications et redirections

- ▶ Une ouverture et une **duplication**.



Manipulation de l'entrée TFO

- ▶ L'appel `fcntl` permet de manipuler les attributs associés à un descripteur ou à une entrée de la TFO.

```
#include <unistd.h>
#include <fcntl.h>
```

```
int fcntl(int descr, int commande, ...);
```

- ▶ Le second argument donne le type d'opération à réaliser :
 - ▶ Récupération/changement du mode d'ouverture : `F_GETFL/F_SETFL`.
 - ▶ Manipulation d'un attribut du descripteur lui-même (vu plus tard).
 - ▶ Duplication à la `dup`.
 - ▶ Pose de verrous.

Entrées-sorties sur répertoires

- ▶ On ne peut pas utiliser `open()` et `read()` sur les répertoires. À la place

```
#include <dirent.h>
```

```
DIR *opendir(const char *dirname);
struct dirent *readdir(DIR *dirp);
void rewinddir(DIR *dirp);
```

- ▶ La structure `dirent` contient un champ `d_name` : le nom de l'entrée dans le répertoire.
- ▶ Curseur déplacé à l'entrée suivante après une lecture réussie.
- ▶ Remise du curseur au début du répertoire par la fonction `rewinddir`.

Les processus : généralités

Quelques attributs des processus

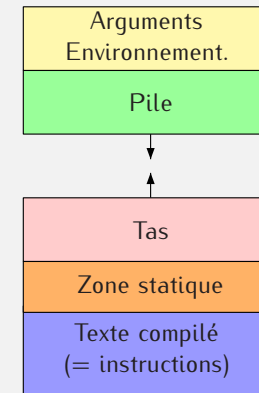
- ▶ Plusieurs attributs sont associés à chaque processus
 - ▶ Masque (`umask`).
 - ▶ Table des descripteurs.
 - ▶ Répertoire courant.
 - ▶ Propriétaire et groupe réels/effectifs.
- ▶ On peut créer de nouveau processus (par exemple en tapant une commande externe sous le shell).
- ▶ Comment sont initialisés ces attributs pour un processus nouveau ?
- ▶ Lorsqu'un processus crée un autre processus, chacun des attributs ci-dessus est copié pour le nouveau processus.

Les processus : caractéristiques

- ▶ Un processus est un programme en cours d'exécution.
- ▶ Le système alloue une structure pour le contrôler, et mémoriser
 - ▶ ses caractéristiques, par exemple
 - ▶ son identité (pid),
 - ▶ son état (prêt, mode kernel/user, endormi, zombi, créé, swappé),
 - ▶ ses propriétaires (réel, effectif),
 - ▶ son répertoire courant, etc.
 - ▶ les ressources qui lui ont été attribuées, par exemple
 - ▶ les fichiers ouverts.
 - ▶ ses demandes non encore satisfaites.

Les processus : organisation mémoire (rappel)

- ▶ La mémoire d'un processus a schématiquement plusieurs zones.
- ▶ Certaines (texte, statique) sont de taille fixée à la compilation.
- ▶ D'autres (pile, tas) ont une taille évoluant au cours de l'exécution.



Arguments et environnement

- ▶ Au lancement d'un processus, une routine de démarrage initialise ses **arguments** et l'**environnement**, avant sa première instruction.
- ▶ L'environnement est une liste de chaînes de caractères de la forme **VARIABLE=valeur**. Par exemple, **HOME=/Users/toto**.
- ▶ Lorsqu'un processus *P* crée un processus *F*, l'environnement de *P* est transmis par défaut à *F*.

Arguments et environnement

- ▶ Un processus utilise son environnement pour adapter son comportement.
 - ▶ `man` utilise `MANPATH` pour chercher les fichiers des pages de manuel.
 - ▶ `COLUMNS` affecte l'affichage de plusieurs commandes (shell, pager, etc.)
- ▶ Un programme peut avoir accès à son environnement en utilisant
 - ▶ Les fonctions de bibliothèque `clearenv`, `putenv`, `setenv`, `unsetenv`.
 - ▶ la variable `extern char **environ;`, cf. `<unistd.h>`.
 - ▶ la forme de `main` mentionnée par C99 comme extension **non portable** :

```
int main (int argc, char *argv [], char *envp []);
```

Allocation mémoire

- ▶ Les fonctions malloc/calloc/realloc et free, si elles sont disponibles, et bien implémentées, permettent d'allouer de la mémoire sans avoir à gérer les désallocations, récupérations, etc.
- ▶ Elles peuvent être implémentées en utilisant directement les fonctions

```
#include <unistd.h>
```

```
int brk(void *end_data_segment);  
void *sbrk(intptr_t increment);
```

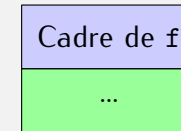
- ▶ Ni POSIX, ni C99 🙄

Appel de fonction (rappel)

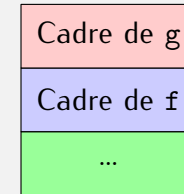
```
void f(void)  
{  
    int x = 3, y = 6, z;  
    z = g(x,y);  
}
```

```
int g(int x1, int x2)  
{  
    int y = 15;  
    return y - x1 * x2;  
}
```

- ▶ Schéma conceptuel de la pile :



Avant appel de g



Pendant l'appel à g

- ▶ Un cadre d'appel de fonction permet de mémoriser (entre autres)
 - ▶ les valeurs des arguments et variables automatiques.
 - ▶ la valeur retour transmise à la fonction appelante.

Sauts non locaux

- ▶ On ne peut pas, en C, faire un goto vers une étiquette d'une autre fonction, mais...
- ▶ On peut, dans une fonction, enregistrer l'état de la pile, via une variable globale de type jmp_buf.

```
int setjmp (jmp_buf buffer);
```

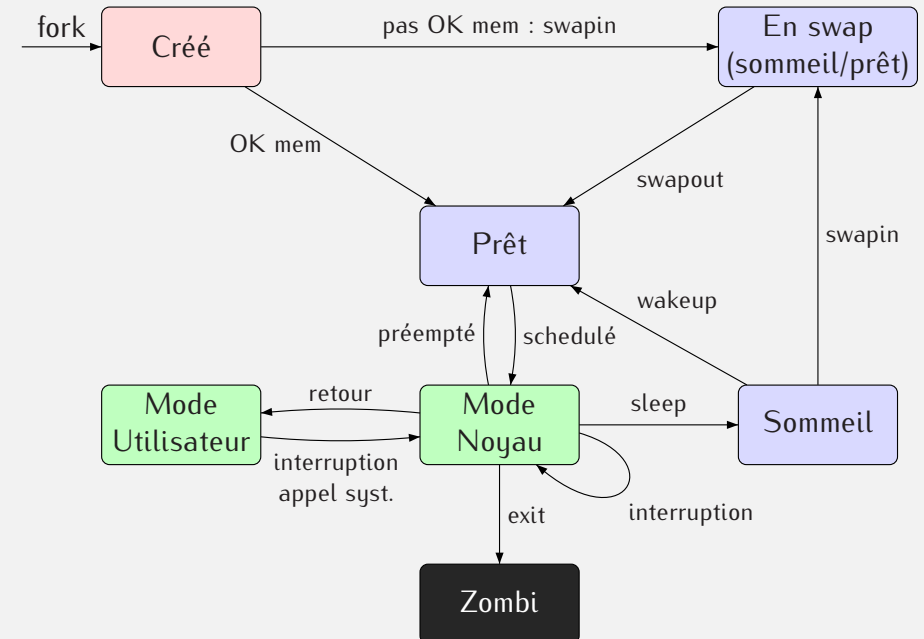
- ▶ D'une fonction dont le cadre est « après » celle ayant appelé setjmp, on peut revenir à l'état enregistré :

```
void longjmp (jmp_buf buffer_sauvegarde, int ret);
```

qui provoque le branchement au setjmp correspondant, avec comme valeur de retour ret.

- ⚠ Ne pas utiliser longjmp vers une fonction terminée.

Les processus : schéma simplifié des états



Identificateurs

- ▶ Chaque processus est **identifié** par un **numéro entier unique**, son pid.
- ▶ Deux processus ne peuvent pas avoir le même numéro.
- ▶ Chaque processus a un **processus père** : celui qui a demandé sa création.
- ▶ Tout processus orphelin est adopté par le processus `init`, de pid 1.

Identificateurs

- ▶ Un processus récupère son identificateur et celui de son père par

```
pid_t getpid(void);  
pid_t getppid(void);
```

- ▶ Tout processus a un propriétaire réel et un propriétaire effectif.

```
uid_t getuid(void);  
uid_t geteuid(void);
```

- ▶ Le propriétaire réel correspond normalement au login de l'utilisateur ayant lancé le processus.
- ▶ Le propriétaire effectif sert pour les vérifications de permissions.
- ▶ Idem pour le groupe, `getgid()/getegid()`.
- ▶ Ces fonctions réussissent toujours.

Pourquoi deux propriétaires ?

- ▶ Ne pas confondre le propriétaire d'un processus et celui d'un **fichier**.
- ▶ De plus, un processus a **2 propriétaires** : réel et effectif. Celui qui détermine les **droits** du processus est le **propriétaire effectif**.
- ▶ Le propriétaire réel est fixé au login, hérité lors de la création d'un fils.
- ▶ Le propriétaire effectif peut changer à l'exécution d'un programme binaire, dont le fichier a le **set-uid** bit positionné.
- ▶ Dans ce cas, le propriétaire effectif prend la valeur du propriétaire du **fichier** exécutable.
- ▶ Ceci permet de donner au processus des droits pendant la durée d'exécution du programme.

Création de nouveaux processus

On verra plus tard que

- ▶ on peut créer des processus avec l'appel système `fork()`.
- ▶ un processus peut exécuter un programme par l'un des appels système `exec(v/1) [p/e]`.

Terminaison

- ▶ Un processus se termine normalement en appelant `exit`, `_exit` ou `return` dans le 1er cadre de la fonction `main`.
- ▶ La fonction `exit()`
 1. appelle les fonctions enregistrées par `atexit`,
 2. vide les buffers de la bibliothèque standard.
 3. continue comme `_exit()`.
- ▶ La fonction `_exit()`
 1. ferme les descripteurs,
 2. termine le processus.
- ▶ Un processus peut se terminer anormalement, sur réception d'un signal.

Les threads

Limitation du modèle classique des processus

Un processus fils est une **copie** indépendante de son père.

- 😊 Synchronisation facile : chaque processus a une copie des ressources, sauf ressources système, comme entrée TFO).
 - ▶ Pas de risque d'écrasement des données de l'autre processus.
- 😞 Partage les descripteurs mais pas de la mémoire
 - ▶ Communication réduite
 - ▶ communication par tubes, tubes nommés,... OK.
 - ▶ communication par mémoire partagée difficile.
 - ▶ Communication nécessite au moins une copie,
 - ▶ Communication nécessite des protocoles parfois compliqués.

Les threads (coprocessus/activités)

- ▶ On peut voir un processus avec plusieurs threads comme exécutant plusieurs "procédures" en parallèle.
- ▶ Thread = un flux séquentiel d'instructions dans processus.
- ▶ Chaque thread d'un processus peut s'exécuter en parallèle.
- ▶ Différence **2 processus père-fils** versus **2 threads** d'un même processus :
il y a plus de partage entre les threads.

Les threads (coprocessus/activités)

- ▶ Pour un processus, le noyau maintient : *id, environnement, cwd, code, données statiques, descripteurs, tables de signaux, pile, descripteurs...
- ▶ Idéalement, un thread partage avec les autres threads du processus tout sauf
 - ▶ son identifiant (unique au sein du processus),
 - ▶ ses registres, sa pile, son compteur de programme,
 - ▶ des informations concernant son ordonnancement,
 - ▶ sa propre valeur d'errno,
 - ▶ son ensemble de signaux pendants et bloqués,
 - ▶ + ... ,
- ▶ En particulier, partage de la mémoire (zone statique).
- ▶ Processus = environnement d'exécution
- ▶ Thread = activité au sein d'un même environnement.

Les threads

- ▶ Lors d'un changement de thread actif, le noyau recharge les registres du processeur, la pile.
- ⇒ gain potentiel à la création de thread, versus `fork()`, l'appel système de création de processus.
- ▶ Le partage mémoire facilite la communication.
 - 😊 Partage de données importantes sans recopie.
 - ☹ Synchronisation difficile : à n'utiliser que si utile.

Exemples d'utilisation

- ▶ Partage de données volumineuses,
- ▶ Parallélisme.
- ▶ Exemples d'algorithme qui se parallélisent bien :
 - ▶ tri fusion,
 - ▶ traitement d'images (quad-trees),
 - ▶ analyse numérique matricielle.
- ▶ Gestion des entrées utilisateurs (interfaces graphiques/programme principal),
- ▶ Entrées multiples (multiplexage)
 - ▶ un thread peut bloquer sans bloquer tout le processus
- ▶ simplification des protocoles de communication.

API threads POSIX

- ▶ Standard IEEE POSIX 1003.1c (payant).
- ▶ En-tête `#include <pthread.h>`, éventuellement bibliothèque `pthread`.
- ▶ Fonctions retournent en général 0 si OK.
- ▶ Assez grosse API (> 50 fonctions).
- ▶ Lorsqu'un programme commence, il n'a qu'un seul thread.
- ▶ Idem après un `fork()` pour le fils.
 - ⚠ Attention, certaines variables (mutex) utilisées pour la synchronisation des threads sont héritées par le fils.
- ▶ Idem après un `exec*`.

Création de thread

```
int  
pthread_create (pthread_t *thread_id,  
               const pthread_attr_t *attr,  
               void * (*routine)(void *),  
               void *arg);
```

- ▶ `thread_id` : identificateur (unique pour ce processus) rempli par l'appel.
- ▶ `attr` : permet de changer les attributs (politique d'ordonnancement, si le thread est ou non joignable...)
NULL ⇒ attributs par défaut.
- ▶ `routine` : fonction de démarrage du thread.
- ▶ `arg` : argument de cette fonction.
- ▶ Le masque des signaux hérité,
- ▶ L'ensemble de signaux pendants vide.

Identification des threads

```
pthread_t pthread_self (void);
```

- ▶ Renvoie l'identificateur du thread appelant.

```
int pthread_equal (pthread_t thr1, pthread_t thr2);
```

- ▶ Renvoie vrai si et seulement si les arguments désignent le même thread.

Terminaison d'un thread

```
void pthread_exit(void *status);
```

- ▶ Termine le thread appelant en permettant à un thread faisant un `pthread_join()` de connaître la valeur de sortie `status` (sauf si le thread est détaché).
- ▶ `pthread_join()` : permet à un processus d'attendre la terminaison d'un autre thread (équivalent du `wait()` qui sera vu plus tard).
- ▶ Pas de libération des ressources du processus, sauf si dernier thread.
- ▶ Un `return x` est un `pthread_exit(x)` implicite...
- ▶ ...sauf pour le 1er thread `main()` où ça équivaut à `exit(x)`.

Destruction de thread par un autre

- ▶ Un thread peut accepter d'être terminé par un autre

```
int pthread_setcancelstate(int state, int *oldstate);  
state soit PTHREAD_CANCEL_ENABLE, soit PTHREAD_CANCEL_DISABLE.
```

- ▶ Terminaison si un autre thread l'a demandé :

```
void pthread_testcancel(void);
```

- ▶ Terminaison sans test, mode asynchrone :

```
pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS,  
                     NULL);
```

- ▶ Demande de terminaison du thread `thr` :

```
int pthread_cancel(pthread_t thr);
```

Détachement

- ▶ Par défaut, POSIX spécifie qu'un thread est **joignable**.
- ▶ Ceci signifie que les ressources allouées pour l'exécution d'un thread, comme sa pile, ne sont libérées que lorsque
 - ▶ le thread s'est terminé, et
 - ▶ un appel à `pthread_join()` pour ce thread a été effectué.
- ▶ L'appel à `pthread_join()` permet d'accéder au code retour effectué par `pthread_exit()`.
- ▶ Après sa création, un thread peut se **détacher** = devenir non joignable

```
int pthread_detach (pthread_t thread);
```
- ▶ Si on n'a pas besoin du code retour d'un thread, on peut le détacher.
- ▶ Dans ce cas, ressources libérées dès la terminaison de l'activité (et on ne peut pas attendre sa terminaison par `pthread_join()`).

Détacher dès la création : attributs

- ▶ Un thread a certains attributs, et l'un spécifie s'il est ou non joignable.
- ▶ Un thread peut décider de se détacher : il devient non joignable.
- ▶ Un thread détaché ne peut pas redevenir joignable.
- ▶ Pour créer un thread joignable ou détaché, on peut utiliser l'argument 2 de `pthread_create()`.
 - ▶ Déclarer une variable **attribut** de type `pthread_attr_t`,
 - ▶ L'initialiser : `pthread_attr_init(&attribut);`.
 - ▶ Pour créer un attribut détaché :

```
pthread_attr_setdetachstate(&attribut, PTHREAD_CREATE_DETACHED);
```
 - ▶ pour créer un attribut joignable :

```
pthread_attr_setdetachstate(&attribut, PTHREAD_CREATE_JOINABLE);
```
 - ▶ Libérer les ressources : `pthread_attr_destroy(&attribut);`.

Attente d'un thread

```
int pthread_join (pthread_t thr_id, void **get_status);
```

- ▶ La fonction suspend l'exécution du thread appelant jusqu'à ce que le thread `thread_id` se termine
- ▶ Si `get_status` est non NULL, la valeur passée à `pthread_exit()` par le thread terminé est mémorisée à cette adresse.

Fonctions réentrantes

Fonction **réentrante** : fonction pouvant être utilisée par plusieurs tâches en concurrence, sans risque de corruption de données.

Exemple de fonction **non** réentrante.

```
int f(void)
{
    static int x = 0;
    x = 2*x+7;
    return x;
}
```

Problème :

- ▶ l'instruction `x = 2*x+7` se compile en plusieurs instructions machine.
- ▶ un thread peut être interrompu au milieu de ces instructions.
- ▶ Le problème peut se poser avec un seul thread interrompu par signaux.

Fonctions réentrantes

- ▶ Une fonction **non réentrante** ne peut être partagée par plusieurs processus ou threads que si on assure l'**exclusion mutuelle** : au plus un thread pourra exécuter son code à un moment donné.
- ▶ Une fonction réentrante peut être interrompue à tout instant et reprise plus tard sans corruption de données.

Exclusion mutuelle

- ▶ 1985–87 5 morts par irradiations massives dues à la machine Therac-25.
Cause. Conflit d'**accès aux ressources** entre 2 parties logicielles.
- ▶ 2003 Panne d'électricité aux USA & Canada, General Electric.
Cause. À nouveau : mauvaise gestion d'**accès concurrents** aux ressources dans un programme de surveillance.
- ▶ Les threads d'un même processus partagent le même espace d'adressage (variables globales, statiques).
- ▶ Il est nécessaire d'éviter l'accès simultané à une même variable.
- ▶ Section critique : portion de code où il est nécessaire que le thread qui l'exécute soit le seul dans cette section.

Fonctions réentrantes

Pour écrire des fonctions réentrantes

- ▶ Pas de données statiques utilisées d'un appel à l'autre.
- ▶ Pas de retour de pointeur statique : données fournies par l'appelant.
- ▶ Travail sur variables locales,
- ▶ Pas d'appel de fonctions non réentrantes
 - ▶ POSIX.1 donne une liste de fonctions réentrantes.
 - ▶ Typiquement `malloc()`, `free()`, les fonctions E/S de la libc ne sont pas garanties réentrantes.
- ▶ N'utiliser qu'en connaissance de cause des fonctions non réentrantes dans les programmes utilisant des signaux, ou multi-threadés.

Exclusion mutuelle

- ▶ Une fonction réentrante peut elle-même poser problème.

```
typedef struct {
    int montant_actuel;
    int interets;
} compte;

void crediter(void *argument)
{
    compte *x = (compte *)argument;

    x->montant_actuel += x->interets;
}
```

- ▶ Fonction est appelée par 2 threads indépendamment, scénario possible
 - ▶ 1er thread lit le montant actuel, disons 100, sans pouvoir compléter +=
 - ▶ 2ème thread idem.
 - ▶ 1er thread incrémente : le montant actuel vaut 110.
 - ▶ 2ème thread incrémente : le montant actuel vaut 110 **au lieu de 120.**

Exclusion mutuelle

- ▶ On doit interdire l'accès au corps de `crediter()` par 2 threads ou plus.
- ▶ Une **section critique** est une section qu'on veut être exécutable par au plus un thread à un instant donné.
- ▶ La propriété d'**exclusion mutuelle** pour une **section critique** dit qu'au plus un thread est dans cette section à un instant donné.
- ▶ Typiquement, un accès à des ressources partagées pose la question de l'exclusion mutuelle.

Exclusion mutuelle

- ▶ Il existe des protocoles
 - ▶ garantissant l'exclusion mutuelle à une section critique,
 - ▶ en assurant que si un processus ne veut pas entrer en section critique, il ne bloque pas les autres,
 - ▶ ... mais faisant une **attente active**.
- ▶ **Exemple** : algorithme de Peterson.

```
int req[2], turn;

while(true)
{
    nc();        // section non critique
    req[i] = true;
    turn = 1-i;
    while (req[1-i] && turn != i)
        ;
    sc();        // section critique
    req[i] = 0;
}
```

Synchronisation par mutex

- ▶ La bibliothèque fournit des moyens d'assurer l'exclusion mutuelle : mutex (verrous), variables conditions, sémaphores.

- ▶ Création/destruction d'un verrou

```
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
```

ou

```
pthread_mutex_t m;
pthread_mutex_attr a;
// ici initialiser a
pthread_mutex_init(&m, const pthread_mutex_attr &a);
```

Synchronisation par mutex

- ▶ Destruction d'un verrou :

```
int pthread_mutex_destroy (pthread_mutex_t *mutex);
```

- ▶ Verrouillage (blocage si déjà verrouillé).

```
int pthread_mutex_lock (pthread_mutex_t *mutex);
```

- ▶ Verrouillage (échec si déjà verrouillé).

```
int pthread_mutex_trylock (pthread_mutex_t *mutex);
```

- ▶ Déverrouillage

- ▶ seul, le thread qui a verrouillé le mutex a le droit de le déverrouiller.
- ▶ Si des threads sont bloqués en attente du verrou, l'un d'entre eux obtient le verrou et est débloqué (lequel ? dépend de l'ordonnancement → attributs).

```
int pthread_mutex_unlock (pthread_mutex_t *mutex);
```

Les processus : création et recouvrement

Création de processus

- ▶ Un processus P demande à en créer un autre grâce à l'appel système

```
pid_t fork(void);
```
- ▶ En cas de succès
 - ▶ Création d'un nouveau processus F , « fils » de P (appelé « père »).
 - ▶ Le fils reçoit une copie de la mémoire du père, en particulier
 - ▶ du texte du programme,
 - ▶ de l'état de la pile, et de la valeur du compteur de programme,
 - ▶ Tout se passe donc comme si le fils était un clone du père, et effectuait comme lui le retour de la fonction `fork()`.
- ▶ La fonction `fork()` retourne donc une fois dans chacun des 2 processus
 - ▶ dans le père, elle retourne le pid du fils.
 - ▶ dans le fils, elle retourne 0.
- ▶ **Remarque** Tout se passe comme si la mémoire du fils était une copie de celle du père. Pour être efficace, la copie n'est faite que quand nécessaire.

Création de processus

- ▶ Les liens de parenté des processus se représentent comme un arbre.
- ▶ La commande `pstree` permet de le visualiser.
- ▶ La commande shell `ps` donne des informations sur les processus.
- ▶ Père et fils diffèrent sur
 - ▶ valeur retour du `fork()`,
 - ▶ pid, ppid,
 - ▶ statistiques temporelles, verrous, signaux pendants, +...
- ⚠ Les deux processus sont concurrents, on ne peut pas faire d'hypothèse sur celui qui exécutera la prochaine instruction.
- ▶ **Exemples** : identités, boucle, interaction bibliothèque.

Redirections

- ▶ Les descripteurs sont dupliqués dans le fils, ils référencent les mêmes entrées que ceux du père.
- ▶ Ceci permet au père et au fils de communiquer.
- ▶ On peut aussi rediriger l'entrée, la sortie ou la sortie erreur du fils juste après l'appel à `fork()`.
- ▶ **Exemple** : redirection de l'entrée standard.

Terminaison

- ▶ Un processus se termine normalement en appelant `exit`, `_exit` ou `return` dans le 1er cadre de la fonction `main`.
- ▶ La fonction `exit()`
 1. appelle les fonctions enregistrées par `atexit`,
 2. vide les buffers de la bibliothèque standard.
 3. continue comme `_exit()`.
- ▶ La fonction `_exit()`
 1. ferme les descripteurs,
 2. termine le processus.
- ▶ Un processus peut se terminer anormalement, sur réception d'un signal.

Terminaison

- ▶ Lorsqu'un processus se termine, il passe à l'état « **zombi** ».
- ▶ Si son père est 1, il est retiré de la liste des processus.
- ▶ Sinon, il reste zombi (et consomme des ressources internes!) jusqu'à ce que son père le libère.
- ▶ Le père le libère en récupérant des informations sur la terminaison par

```
pid_t wait(int *status);
```
- ▶ L'appel `wait` retourne immédiatement si le processus n'a pas de fils, ou a déjà un fils zombi.
- ▶ Sinon, l'appel est **bloquant** : mise en **sommeil** du processus appelant jusqu'à ce qu'un fils se termine (ou réception d'un signal).
- ▶ Au retour de l'appel `wait`, l'entier pointé par `status` contient des informations sur la terminaison du fils.

Terminaison

- ▶ Macros appliquées à l'entier `status` acquis par `wait/waitpid` :
 - ▶ `WIFEXITED(status)` : vrai si le processus s'est terminé normalement,
 - ▶ `WEXITSTATUS(status)` : le code retour du processus,
 - ▶ `WIFSIGNALED(status)` : vrai si le processus a été terminé par un signal,
 - ▶ `WTERMSIG(status)` : le numéro du signal.
 - ▶ +...
- ▶ Un double `fork()` permet de ne pas avoir à attendre un fils.
Comment ?

Terminaison

- ▶ L'appel `wait()` ne permet pas de préciser quel processus on attend, et l'appel est bloquant si le processus appelant a des fils, tous non zombis.
- ▶ `waitpid()` permet de lever ces limitations

```
pid_t waitpid(pid_t pid, int *status, int options);
```

 - ▶ `pid` peut être :
 - ▶ `-1` : attente de n'importe quel processus,
 - ▶ `> 0` : attente du processus de pid `pid`.
 - ▶ `= 0, < 0` : vu plus tard.
 - ▶ Option fréquente : `WNOHANG` : non bloquant.
 - ▶ L'appel permet aussi de récupérer des informations sur les fils stoppés (option `WUNTRACED`).

Recouvrement

- ▶ Un processus exécute un programme exécutable par l'une des 6 fonctions

`exec(l/v)(-/p/e).`

- ▶ Le processus « repart à 0 » avec le texte d'un autre programme.
- ▶ Il garde son pid, ppid, son propriétaire et groupe réel, son répertoire courant, son umask, ses signaux pendants, +...
- ▶ Il change de propriétaire/groupe **effectif** si le **setuid/setgid** bit du binaire qu'il exécute est positionné.
- ▶ Chaque descripteur a un flag **close_on_exec**. S'il est positionné, le descripteur est fermé au travers d'exec.
 - ▶ On positionne ce flag sur le descripteur d par
`fcntl(d, F_SETFD, FD_CLOEXEC);`

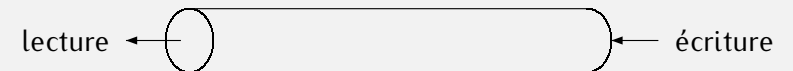
Les fonctions de recouvrement

- ▶ `execl, execlp, execl, execv, execvp, execve.`
- ▶ Les fonctions **execl*** spécifient les arguments comme une **liste**.
- ▶ Les fonctions **execv*** spécifient les arguments comme un **vecteur**.
- ▶ Dans les 2 cas, on spécifie :
 1. le fichier binaire à exécuter,
 2. les arguments, sous forme **liste** ou **vecteur**,
- ▶ Les fonctions `exec(v/1)p` permettent de chercher le binaire dans les répertoires spécifiés par la variable d'environnement **PATH**,
- ▶ Les fonctions `exec(v/1)e` permettent de passer en dernier argument un **environnement** `char *env[]`.

Communication entre processus : les tubes

Communication entre processus : les tubes

- ▶ Un tube est un canal de communication FIFO (**F**irst **I**n, **F**irst **O**ut).
- ▶ Les caractères lus sont les caractères les plus anciennement écrits.
- ▶ Le programmeur ne peut pas déplacer la tête de lecture/écriture.
- ▶ Elle est gérée par le système pour assurer l'aspect FIFO.



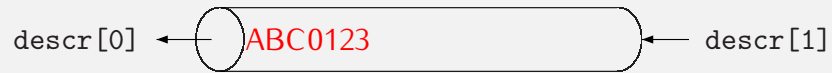
- ▶ L'appel `pipe` permet de créer un tube.

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

ouvre un tube et initialise 2 descripteurs :

- ▶ `descr[0]` référence l'« extrémité » **lecture** du tube,
- ▶ `descr[1]` référence l'« extrémité » **écriture** du tube.

Les tubes



```
char buf[4];
int descr[2];
pipe(descr);
write(descr[1], "ABC", strlen("ABC"));
write(descr[1], "0123", strlen("0123"));
read(descr[0], buf, 4); // buf contient {'A','B','C','0'}
read(descr[0], buf, 4); // buf contient {'1','2','3','0'}
read(descr[0], buf, 4); // Belle au bois dormant
```

Les tubes

Un processus est

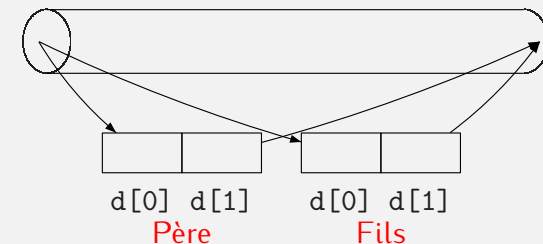
- ▶ **écrivain** (potentiel) sur le tube **d** s'il a le descripteur **d[1]** ouvert.
- ▶ **lecteur** (potentiel) sur le tube **d** s'il a le descripteur **d[0]** ouvert.

Les tubes

- ▶ La lecture sur tube vide
 - ▶ renvoie 0 s'il n'y a plus d'écrivains,
 - ▶ provoque la mise en sommeil du processus sinon.
- ▶ L'écriture sur un tube qui n'a plus de lecteurs provoque la génération d'un signal SIGPIPE (qui normalement termine le processus).
- ▶ Un tube a une taille maximale finie. Écriture sur tube plein avec lecteurs \Rightarrow sommeil.
- ▶ Écriture de taille \leq PIPE_BUF : sans entrelacements avec d'autres E/S sur le tube.

Les tubes

- ▶ Exemple : On veut transmettre des informations du père vers le fils.



```
int d[2];
pipe(d);
fork();
fermetures...
```

Tubes nommés

- ▶ Les tubes sont bien adaptés pour transmettre de l'information entre processus ayant des descripteurs en commun.
- ▶ Pour des processus sans lien de parenté : tubes nommés.

```
int mkfifo(const char *chemin, mode_t mode);
```

- ▶ Ces tubes apparaissent dans le système de fichiers.
- ▶ `mode` contient uniquement des bits de permissions, utilisés avec la valeur d'`umask` pour la création.
- ▶ *stat* donne le type `S_ISFIFO`.

Tubes nommés

- ▶ L'appel `open` est **bloquant** pour un tube nommé.
- ▶ Un processus demandant l'ouverture en lecture attend l'arrivée d'écrivains, et vice-versa.
- ⚠ Quel blocage peut-il arriver si deux processus veulent dialoguer dans les deux sens?
- ▶ On peut rendre l'ouverture non bloquante (`O_NONBLOCK` dans le mode d'ouverture).
Dans ce cas, une demande en écriture sans lecteurs échoue.

Les signaux

Qu'est-ce qu'un signal ?

- ▶ Un signal est un événement indiquant au processus qui le reçoit que quelque chose est arrivé.
- ▶ Analogies dans la vie courante :
 - ▶ Le réveil sonne.
 - ▶ Le téléphone sonne.
 - ▶ Quelqu'un frappe à la porte.
 - ▶ Une fenêtre arrive à l'écran pour annoncer qu'on a du courrier.
 - ▶ Un automobiliste fait un appel de phares...
- ▶ Un signal peut changer le déroulement d'un processus.
- ▶ Les signaux sont dits *asynchrones* : reçus à n'importe quel moment.
- ▶ Les signaux sont codés par des entiers et identifiés par des **noms**.

Principaux signaux (1)

Signal	Action par défaut	Événement déclenchant
SIGABRT	A	Généré par <code>abort()</code> .
SIGALRM	T	Expiration d'une alarme.
SIGCHLD	I	Fils terminé ou stoppé (ou continué [XSI]).
SIGCONT	C	Continue l'exécution, si stoppé.
SIGFPE	A	Opération arithmétique erronée.
SIGHUP	T	Hangup, fin de session.
SIGINT	T	Interruption au terminal (C-c).
SIGKILL	T	Termine le processus (*).
SIGPIPE	T	Écriture sur tube sans lecteurs.

(*) ne peut être ni capté ni ignoré

- ▶ T : termine le processus normalement.
- ▶ A : termine le processus anormalement (eg, avec fichier core).
- ▶ I : ignoré.
- ▶ C : continue un processus stoppé.

Principaux signaux (2)

Signal	Action par défaut	Événement déclenchant
SIGQUIT	A	Quit au terminal (C-\).
SIGSEGV	A	Référence mémoire invalide.
SIGSTOP	S	Stoppe l'exécution (*).
SIGTERM	T	Termine l'exécution.
SIGTSTP	S	Envoi Stop au terminal (C-Z).
SIGTTIN	S	Processus en background essayant de lire.
SIGTTOU	S	Processus en background essayant d'écrire.
SIGUSR1	T	À la disposition du programmeur.
SIGUSR2	T	À la disposition du programmeur.

(*) ne peut être ni capté ni ignoré.

- ▶ T : termine le processus normalement.
- ▶ A : termine le processus anormalement (eg, avec fichier core).
- ▶ I : ignoré.
- ▶ S : stoppe le processus.

Signaux générés

- ▶ Un signal est **généré** à un processus lorsqu'un événement survient. Par exemple :
 - ▶ Envoi d'un signal par un autre processus.
 - ▶ Écriture sur tube sans lecteurs. SIGPIPE
 - ▶ Terminaison d'un fils. SIGCHLD
 - ▶ Expiration d'une minuterie. SIGALRM
 - ▶ Erreur arithmétique (ex : division par 0). SIGFPE
 - ▶ Activité reliée au terminal (ex. *job control*) SIGTSTP, SIGINT.
- ▶ Si on envoie à un processus n occurrences d'un signal s ,
 - ▶ le processus détectera qu'il a reçu **au moins une** occurrence du signal...
 - ▶ mais ne peut détecter avec certitude n occurrences.

Signaux délivrés. Traitants - Handlers

- ▶ Comme pour un signal de la vie courante, un processus peut choisir
 - ▶ d'ignorer le signal.
 - ▶ d'effectuer l'action par défaut associée au signal.
 - ▶ d'effectuer une action spécifiquement choisie, appelée **traitant** ou **handler**.
- ▶ Exceptions : SIGKILL, SIGSTOP.
- ▶ Un signal est **délivré** lorsque le processus effectue l'action choisie.
- ▶ La délivrance des signaux intervient lorsque le processus passe du mode actif noyau au mode actif utilisateur.
- ▶ Conséquence : pendant qu'un processus est en mode noyau (par ex., en train de réaliser une entrée-sortie), il ne prend pas en compte les signaux.

Appels système interruptibles ou non

- ▶ Il y a deux sortes d'appels système
 - ▶ **Appels interruptibles** : les appels systèmes qui peuvent bloquer un temps arbitrairement long.
 - ▶ Par exemple `wait`, `read` dans un tube vide, `write` dans un tube plein...
 - ▶ Si un signal non ignoré est généré pendant que le processus est bloqué, l'appel échoue (erreur : `EINTR`).
 - ▶ **Appels non interruptibles** les autres (attente normalement bornée)
 - ▶ Par exemple `getpid`, `umask`, `read/write` dans un fichier.
 - ▶ Ceux-ci sont non interruptibles.
- ▶ **Retenir** : les appels systèmes **bloquants**, tel que `read` sur un tube vide avec écriture, sont interruptibles par un signal.

Signaux pendants

- ▶ Un signal est dit **pendant** lorsqu'il a été généré mais pas encore délivré.
 - ▶ signal généré = l'événement arrive
 - ▶ signal délivré = l'action choisie est appelée.
- ▶ Chaque processus a un ensemble de signaux pendants.
- ▶ Un processus peut **masquer** (ou **bloquer**) les signaux qu'il souhaite.
- ▶ Exceptions : `SIGKILL`, `SIGSTOP`.
- ▶ Si l'action associée à un signal bloqué est différente d'ignorer, et le signal reste pendant jusqu'à ce qu'il soit débloqué.

Signaux bloqués : masque

- ▶ Le masque d'un processus est l'ensemble des signaux bloqués.
- ▶ Hérité de son père.
- ▶ La délivrance des signaux bloqués est différée jusqu'à ce qu'ils soient supprimés du masque.
- ▶ `SIGKILL` et `SIGSTOP` ne peuvent pas être masqués.
- ▶ Pour installer un masque : 2 étapes.
 1. Construction de l'ensemble des signaux.
 2. Installation du masque par un appel adapté
 - ▶ `sigaction`
 - ▶ `sigprocmask`
 - ▶ `sigsuspend`

Interface POSIX : installer un handler

- ▶ Pour installer un handler, on utilise

```
#include <signal.h>
int sigaction(int sig, const struct sigaction *act,
              struct sigaction *old_act);
```

- ▶ La structure `sigaction` a la forme suivante.

```
struct sigaction
{
    void (*sa_handler)(int);
    int sa_flags;
    sigset_t sa_mask;
}
```

- ▶ `sa_handler` : un pointeur de fonction à un argument entier
 - ▶ `SIG_DFL` : traitant par défaut.
 - ▶ `SIG_IGN` : ignorer le signal.
- ▶ Si `act.sa_handler == f`, l'appel `sigaction(sig,&act,&old_act)` installe le traitant `f()` pour le signal `sig` et récupère l'ancien dans `old_act`.

Installer un traitant : sigaction (2)

- ▶ Le masque installé durant le traitant est l'union
 - ▶ du masque courant,
 - ▶ du signal ayant provoqué son appel,
 - ▶ de l'ensemble spécifié par `sa_mask`.
- ▶ `sa_flags` peut valoir, en particulier
 - ▶ `SA_NOCLDSTOP` et `sig==SIGCHLD` : ne pas générer `SIGCHLD` si un fils est stoppé.
 - ▶ `SA_NOCLDWAIT` : si `sig` est `SIGCHLD`, ne pas faire passer les fils zombies (XSI).
 - ▶ `SA_RESTART` : redémarre les appels systèmes interruptibles, interrompus avant d'avoir pu commencer (XSI).
 - ▶ `SA_RESETHAND` : rebascule le traitant à `SIG_DFL` lors de la délivrance.
 - ▶ `SA_NODEFER` : ne pas masquer dans le handler le signal délivré.

Installation d'un masque

- ▶ Création d'un ensemble de signaux : on dispose d'un type `sigset_t` et

```
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
int sigismember(const sigset_t *set, int signum);
```

- ▶ L'installation d'un masque se fait par

```
int sigprocmask(int how, const sigset_t *set,
               sigset_t *old_set);
```

- ▶ `set` permet de déterminer que masque installer, si non `NULL`.
- ▶ Dans `old_set`, on récupère l'ancien ensemble, si non `NULL`.
- ▶ `how` peut valoir
 - ▶ `SIG_BLOCK` : ajouter `set` au masque actuel.
 - ▶ `SIG_UNBLOCK` : enlever `set` du masque actuel.
 - ▶ `SIG_SETMASK` : installer comme masque exactement `set`.

Attente d'un signal

- ▶ On peut calculer l'ensemble des signaux pendants :

```
int sigpending(sigset_t *set);
```

- ▶ L'appel `int pause(void)` ; permet de passer en sommeil jusqu'à réception d'un signal. À ne pas utiliser...
- ▶ `int sigsuspend(const sigset_t *sigmask)` ;
 - ▶ fait passer le processus appelant en sommeil,
 - ▶ installe le masque spécifié,...
 - ▶ ... le tout de façon **atomique**.

Exemple : fonction sleep

- ▶ Un processus peut demander à recevoir le signal `SIGALRM` au bout d'un certain temps.
- ▶ Les fonctions

```
unsigned int alarm(unsigned int seconds);
int setitimer(int which, struct itimerval *value,
              struct itimerval *old);
```

permettent de manipuler des chronomètres qui, à expiration, envoient des signaux (`SIGALRM`, `SIGVTALRM`, `SIGPROF`).

Envoi de signaux

- ▶ À un processus ou un groupe de processus :

```
int kill(pid_t pid, int sig);
```

- ▶ Pour que P_1 puisse envoyer un signal à P_2 ,
 - ▶ le propriétaire effectif de P_1 doit être 0 (root), ou
 - ▶ le propriétaire réel ou effectif de P_1 doit être le même que celui de P_2 (+ saved-set uid).
- ▶ Exception : SIGCONT.
- ▶ On peut désigner un groupe de processus avec un argument négatif.

- ▶ À soi-même

```
int raise(int sig);  
void abort(void);
```

Génère SIGABRT. Masque est ignoré, traitant SIG_IGN ignoré.

```
alarm, setitimer
```

Gènèrent SIGALRM + ... au bout d'un certain temps.

Signaux et threads

- ▶ Chaque thread possède son masque et ses signaux pendants.
- ▶ Un thread peut positionner son propre masque :

```
int pthread_sigmask(int how, const sigset_t *set,  
                   sigset_t *old);
```

- ▶ On peut envoyer un signal à un thread spécifique :

```
int pthread_kill(pthread_t thr_id, int signal);
```

Signaux et fork/exec

- ▶ Lors d'un `fork()`, le processus fils hérite
 - ▶ du masque du père,
 - ▶ des handlers installés.

mais **pas des signaux pendants !**

- ▶ Lors d'un `exec()`, le processus garde
 - ▶ le masque avant l'exec.
 - ▶ les signaux pendants.

mais **pas les handlers !**

Sauts non locaux

- ▶ Au retour d'un saut non local venant d'un traitant, quel est le masque ?
 - ▶ Celui avant l'entrée dans le traitant ?
 - ▶ Celui au moment du `setjmp` initial ?

- ▶ Les fonctions POSIX

```
int sigsetjmp(sigjmp_buf env, int savemask);
```

```
void siglongjmp(sigjmp_buf env, int val);
```

permettent de préciser si le masque au moment du 1er passage est sauvegardé aussi.

Problèmes dûs à l'aspect asynchrone

- ▶ Un handler peut interrompre un programme de façon asynchrone.
- ▶ C'est problématique lorsque le programme manipule des données statiques, typiquement :
 - ▶ variable statique manipulée par le programme,
 - ▶ soit explicitement : `static int x;`
 - ▶ soit implicitement : par exemple, `gethostbyname()` renvoie un pointeur vers une variable allouée en zone statique.
 - ▶ manipulation d'une zone mémoire sur le tas.
 - ▶ soit explicitement `malloc()`.
 - ▶ soit implicitement : par exemple, `printf()` appelle `malloc()`.

Fonctions sûres (appelées réentrantes)

Corruption des données possible si

1. Le programme est interrompu par un handler **pendant** des manipulations en zone statique ou sur le tas, et
2. Le handler lui-même manipule ces données.

Conséquences.

- ▶ Un handler ne doit utiliser que des fonctions sûres, dites **réentrantes**.
- ▶ Avoir en mémoire que même une affectation peut correspondre à plusieurs instructions machines \rightsquigarrow **non atomique**.

Optimisations du compilateur

- ▶ Le compilateur peut détecter des situations où une variable ne change pas de valeur, et optimiser le code.

```
static sig_atomic_t i;
while (!i)
    ;
```

- ▶ (Le type `sig_atomic_t` garantit des affectations atomiques).
- ▶ Un compilateur peut remarquer que la variable `i` n'est pas changée, et optimiser la boucle.

```
while (true)
    ;
```

- ▶ Pour désactiver cette optimisation :

```
static volatile sig_atomic_t i;
while (!i)
    ;
```

Points importants dûs au caractère asynchrone

- ▶ Risques d'inter-blocage. S'en protéger en **masquant** correctement ou en **synchronisant**.
- ▶ Prendre en compte la perte possible de signaux.
- ▶ N'utiliser que des fonctions réentrantes.
- ▶ Attention aux allocations dynamiques, et aux variables statiques. Exemple : `errno`.

Pour la prochaine fois 😊

- ▶ **Projet!**
- ▶ Pourquoi le signal délivré est-il ajouté au masque dans le traitant ?
- ▶ Pourquoi est-il important que l'installation du masque par `sigsuspend()` et le passage en sommeil forment un tout atomique ?
- ▶ Que se passe-t-il si un processus reçoit un signal quand il est dans un traitant pour un autre signal ?
- ▶ Que se passe-t-il si un processus reçoit un signal quand il est dans un traitant pour le même signal ?
- ▶ Si un programme est bloqué sur un appel système, quels signaux lui sont-ils délivrés ?
- ▶ Que se passe-t-il si deux signaux arrivent rapidement, avant que le système ne puisse déclencher le traitant du 1er ?

Le système de gestion des fichiers (SGF)

Fichiers

- ▶ Un fichier est une entité sur laquelle on peut réaliser certaines opérations, typiquement lecture et écriture.
 - ▶ Fichiers sur disque, contenant une séquence d'octets,
 - ▶ Périphériques physiques : bandes magnétiques, terminaux, imprimante,...
- ▶ POSIX.1 définit plusieurs types de fichiers
 - ▶ fichiers normaux,
 - ▶ répertoires,
 - ▶ tubes,
 - ▶ fichiers en mode bloc,
 - ▶ fichiers en mode caractère.
 - ▶ lien symbolique (pas POSIX 1003.1, mais POSIX 1003.1-2001).

Le système de fichiers

- ▶ Un périphérique tel qu'un disque peut être vu comme une succession ordonnée de blocs.
- ▶ Le système doit donner à l'utilisateur une vue arborescente,
 - ▶ avec des répertoires,
 - ▶ la possibilité de nommer (référencer) un fichier,
 - ▶ la possibilité de protéger ses données envers les autres utilisateurs (en lecture, en écriture ou en exécution), etc.
- ▶ Pour cela, à chaque fichier est associé une structure appelée **i-nœud**.

Système de gestion des fichiers (SGF)

- ▶ Chaque fichier dans l'arborescence est identifié par
 - ▶ le système de fichiers sur lequel il réside,
 - ▶ un identifiant unique sur ce système, appelé i-nœud.
- ▶ On peut greffer la racine d'un système de fichier dans un autre (mount).
- ▶ Un i-nœud contient des informations sur le fichier et ses premiers blocs.
- ▶ Pour les autres blocs, on utilise des blocs d'indirection.
 - ▶ simple,
 - ▶ double,
 - ▶ triple.
- ▶ Combien de blocs au maximum, avec blocs de 4K et adressage 32 bits?
- ▶ Pourquoi les blocs directs, d'indirection simple et double? **Efficacité.**
- ▶ Peut-on faire des trous dans des fichiers? **Oui.**
- ▶ **Note** : Le nom de fichier (qui ne doit pas contenir / ou le caractère nul) n'est pas stocké dans le i-nœud.

Récupération des attributs d'un i-nœud

Les attributs d'un i-nœud sont stockés dans une structure `stat`.

```
#include <sys/stat.h>
```

```
struct stat {
    dev_t      st_dev;      // ID du périphérique
    ino_t      st_ino;      // Numéro du i-noeud
    mode_t     st_mode;     // Type et permissions
    nlink_t    st_nlink;    // Nb de liens physiques
    uid_t      st_uid;      // UID propriétaire
    gid_t      st_gid;      // GID propriétaire
    off_t      st_size;     // Taille totale en octets
    time_t     st_atime;    // Heure dernier accès
    time_t     st_mtime;    // Heure dernière modification
    time_t     st_ctime;    // Heure dernier changement éta
};
```

Les types `dev_t`, `ino_t`, etc. encapsulent un type arithmétique.

Un premier appel système : stat

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
```

```
int stat(const char *path, struct stat *buf);
int lstat(const char *path, struct stat *buf);
```

- ▶ Renvoient 0 en cas de succès, -1 sinon.
- ▶ 1er argument : chemin d'accès au fichier.
- ▶ En cas de succès, remplit la structure à l'adresse 2ème argument avec les attributs du i-nœud trouvé.
- ▶ `stat` suit les liens symboliques, contrairement à `lstat`.

Champs de la structure stat

- ▶ Un appel à `(l)stat` provoque la lecture d'un i-nœud.
- ▶ Les champs de la structure `stat` contiennent les informations du i-nœud autres que les blocs de données. Ex.
 - ▶ `st_size` est la taille du fichier. Pour les répertoires : en général un multiple de 1K. Pour les liens symboliques : le nombre de caractères de la cible.
 - ▶ Les temps correspondent à
 - ▶ `st_atime` = dernière lecture des données.
 - ▶ `st_mtime` = dernière écriture des données.
 - ▶ `st_ctime` = dernier changement de droits, propriétaires, etc.
 - ▶ On peut changer les 2 premiers (si on a les droits)!

```
#include <utime.h>
```

```
int utime(const char *path,
          const struct timeval times[2]);
```

Types de fichiers

- ▶ Le champ `st_mode` donne les permissions et le type de fichier.
- ▶ Macros `S_ISXXX()` pour tester le type du fichier, où `XXX` est
 - ▶ **REG** fichier ordinaire sans interprétation de contenu.
 - ▶ **DIR** répertoire contient typiquement couples (nom,i-nœud).
 - ▶ **FIFO** tube utilisé pour communication entre processus.
 - ▶ **CHR** fich. spécial car. périphérique caractère (eg, terminal).
 - ▶ **BLK** fich. spécial bloc périphérique bloc (eg, disque).
 - ▶ **LNK** lien symbolique pas POSIX < 2001.
 - ▶ **SOCK** socket pas POSIX < 2001.

Permissions : propriétaire et set-uid bit

- ▶ Chaque **processus** possède (entre autres)
 - ▶ un propriétaire et groupe propriétaire **réels**.
 - ▶ un propriétaire et groupe propriétaire **effectifs**, et éventuellement des groupes supplémentaires.
 - ▶ Ces derniers déterminent les droits.
- ▶ Chaque **fichier** a
 - ▶ un propriétaire (champ `st_uid` de la structure `stat`) et
 - ▶ un groupe propriétaire (champ `st_gid` de la structure `stat`).
- ▶ Lorsqu'un processus exécute un programme, le propriétaire effectif du processus est basculé, si le fichier exécuté a le **set-uid** bit positionné, au propriétaire du fichier.
- ▶ 3 catégories de permissions : r/w/x.

Permissions

Pour ouvrir un fichier en lecture, écriture, ou lecture-écriture, il faut

- ▶ l'accès en exécution sur les répertoires du chemin traversé pour l'accès au fichier (donné explicitement ou non).
- ▶ les permissions appropriées sur le fichier lui-même.
 - ▶ Permission en lecture pour lire un fichier ou lister un répertoire,
 - ▶ Permission en écriture pour modifier un fichier ou le contenu d'un répertoire (ajouter ou supprimer un fichier).

Permissions

L'autorisation dépend du propriétaire, groupe propriétaire du **fichier** et du propriétaire, groupe propriétaire **effectifs** du **processus**.

- ▶ si le propriétaire **effectif** du processus est root (uid 0), OK.
- ▶ sinon, on compare le propriétaire **effectif** du processus avec le propriétaire du fichier. S'ils sont identiques, les droits correspondants s'appliquent.
- ▶ sinon, idem avec le groupe,
- ▶ sinon, on regarde les permissions pour les autres.

Rem `int access(const char *path, int amode)` ; teste les permissions vis-à-vis du propriétaire **réel**.

Représentation des permissions

- ▶ Des champs de bits sont utilisés pour manipuler les permissions

$$S_I \begin{bmatrix} R \\ W \\ X \end{bmatrix} \begin{bmatrix}USR \\ GRP \\ OTH \end{bmatrix} \text{ et } S_{IRWX} \begin{bmatrix}U \\ G \\ O \end{bmatrix}$$

- ▶ Par exemple, `S_IWGRP` : la permission en **écriture** pour le **groupe**.
- ▶ `S_IRWXO` est une abréviation pour `S_IROTH|S_IWOTH|S_IXOTH`.
- ▶ Set-uid et s-gid bit : `S_ISUID`, `S_ISGID`.
- ▶ On utilise les opérateurs bit-à-bit pour tester ou modifier une permission du champ `buf.st_mode`.
- ▶ Le sticky-bit `S_ISVTX` (non documenté POSIX, extension XSI) permet de modifier la gestion des permissions dans un répertoire.

Propriétaires

- ▶ A la création, un fichier a comme propriétaire le propriétaire effectif du processus qui l'a créé.
- ▶ Le groupe propriétaire d'un fichier créé peut être
 - ▶ soit le groupe propriétaire effectif du processus créateur,
 - ▶ soit le groupe du répertoire qui le contient.
 - ▶ Le comportement dépend du système ou des options de montage.
- ▶ La modification peut se faire par les fonctions

```
#include <unistd.h>
```

```
int chown(const char *path, uid_t owner, gid_t group);  
int lchown(const char *path, uid_t owner, gid_t group);
```

- ▶ Si `pathconf(path, _PC_CHOWN_RESTRICTED)`, ou `_POSIX_CHOWN_RESTRICTED` sont non nuls, seul root peut appeler `chown`.
- ▶ `lchown` (ne suit pas les liens symboliques) est une extension XSI.

Masque de création

- ▶ Chaque processus a une valeur de masque qui lui est associée.
- ▶ Lorsqu'un fichier (ou répertoire) est créé, les bits de permission présents dans le masque sont basculés off.
- ▶ Le masque est modifiable en utilisant l'appel

```
#include <sys/stat.h>
```

```
mode_t umask(mode_t cmask);
```

où `cmask` est construit par | à partir des constantes définies précédemment.

- ▶ La fonction
 - ▶ positionne le nouveau masque à `umask`,
 - ▶ renvoie la valeur de masque précédent.

Changement des permissions

- ▶ Un processus de propriétaire effectif xyz (ou root) peut changer les permissions des fichiers de propriétaire xyz.
- ▶ Ceci indépendamment des permissions qu'il a sur le fichier.
- ▶ Le changement de permissions peut se faire par la fonction `chmod`.

```
#include <sys/stat.h>
```

```
int chmod(const char *path, mode_t mode);
```

Liens

- ▶ Un même i-nœud peut apparaître dans plusieurs répertoires, associé à un ou plusieurs noms.
- ▶ Exemple

```
espelette> ls -lid / /bin/..
2 drwxr-xr-x 21 root root 4096 jan 30 09:20 /
2 drwxr-xr-x 21 root root 4096 jan 30 09:20 /bin/..
```
- ▶ Les répertoires / et /bin/.. correspondent au le même i-nœud.
- ▶ On dit que ce sont des liens d'un même i-nœud.
- ▶ A part le cas de . et .., on ne peut pas créer de liens d'un répertoire (pour conserver l'aspect arborescent)...

Liens

- ▶ La création d'un lien se fait par l'appel

```
#include <unistd.h>

int link(const char *path_ori, const char *path_new);
```

où path_new est le nouveau nom du lien.
- ▶ path_ori doit exister, avec permissions OK d'accès au répertoire de path_ori et d'accès/création dans le répertoire de path_new.
- ▶ La suppression se fait par

```
int unlink(const char *path_ori);
```
- ▶ Ne suit pas les liens symboliques.
- ▶ Remarque Permission en écriture sur le fichier lui même non requise.

Liens

- ▶ La création/suppression d'un lien affecte de façon atomique
 - ▶ le répertoire dans lequel le lien est créé ou supprimé.
 - ▶ le compte du nombre de liens (champ st_nlink).
- ▶ Blocs désalloués si st_nlink == 0 + aucun processus n'a le fichier ouvert
- ▶ Pour les répertoires, on utilise rmdir.

Renommage de fichiers

- ▶ L'appel rename permet de renommer un fichier

```
#include <stdio.h>

int rename(const char *old, const char *new);
```
- ▶ Si old n'est pas un répertoire, new n'est pas un répertoire.
- ▶ Si old est un répertoire, new est un répertoire vide ou n'existe pas (et n'a pas old comme préfixe).
- ▶ Permissions en écriture nécessaires pour les répertoires modifiés.
- ▶ Ne suit pas les liens symboliques.
- 😊 Pour renommer un fichier régulier sur un même sgf, pas besoin de copier les blocs de données, mais juste d'une écriture sur les répertoires impliqués.

Liens symboliques

- ▶ ... on ne peut pas créer de liens d'un répertoire...
- ▶ ... ni créer un lien d'un système de fichier vers un autre (pourquoi?).
- ▶ Pour pallier ces deux limitations, on utilise des liens symboliques.
- ▶ Ils sont vus par le système comme des fichiers spéciaux.
- ▶ Leur contenu désigne un fichier cible, qui existe ou non.
- ▶ De nombreux appels système POSIX-2001 « suivent » les liens symboliques.
- ⚠ Fonctions qui suivent les liens symboliques et opèrent récursivement sur les répertoires...

Liens symboliques

- ▶ La création de lien symbolique se fait par

```
#include <unistd.h>

int symlink(const char *path1, const char *path2);
```
- ▶ La lecture du contenu d'un lien symbolique se fait par

```
ssize_t readlink(const char *path,
                 char *buf,
                 size_t bufsize);
```
- ▶ Le contenu du lien symbolique est le fichier pointé.
- ▶ La fonction `readlink` place ce contenu dans le buffer `buf`,
- ⚠ Pas terminé par `'\0'`.
- ▶ La fonction retourne le nombre d'octets lus, jamais plus que `bufsize`.

Création/suppression de répertoires

- ▶ La création et suppression de répertoires se fait par

```
#include <sys/stat.h>

int mkdir(const char *path, mode_t mode);
int rmdir(const char *path);
```
- ▶ Le répertoire créé est vide (avec liens `.` et `..`).
- ▶ L'argument `mode` de `mkdir` est le mode de création modifié par le masque du processus.
- ▶ La suppression demande que le répertoire soit vide.
- ▶ La suppression de blocs alloués au répertoire suit la même règle que pour un fichier.

Structure stat : extensions XSI

- ▶ XSI introduit des champs additionnels
 - ▶ `dev_t st_rdev` : ID du périphérique correspondant à un fichier bloc/carac. (`st_dev` donne l'ID du système de fichiers contenant ce fichier spécial).
 - ▶ `~` macros `major` et `minor` en général fournies par l'implémentation `sys/stat.h` ou `sys/sysmacros.h`, mais pas POSIX.
 - ▶ `blksize_t st_blksize` : taille préférée de bloc pour E/S.
 - ▶ `blkcnt_t st_blocks` : nombre de « blocs » alloués... mais unité non précisée!

Le répertoire courant

- ▶ A chaque processus est associé un **répertoire courant**.
- ▶ Le programmeur nomme les chemins dans l'arborescence
 - ▶ Soit de façon absolue, en commençant par la racine.
 - ▶ Soit de façon relative. Dans ce cas, le 1er composant du chemin est cherché à partir du répertoire courant.
 - ▶ L'appel

```
int chdir(const char *path);
```

permet de positionner le répertoire courant du processus.

```
char *getcwd(char *buf, size_t size);
```

permet de récupérer le répertoire courant dans le buffer buf ayant size octets alloués.

⚠ longueur max d'un chemin donnée par `pathconf(".", _PC_PATH_MAX)`.

Système de gestion de fichiers : résumé

- ▶ Présentation d'une **implémentation simplifiée** de SGF (s5fs).
- ▶ Principes se retrouvant dans les implémentations actuelles.
- ▶ Appels système pour accès et modification i-nœuds.
- ▶ Nombreuses améliorations dans les systèmes ultérieurs
 - ▶ Récupération après crash (systèmes journalisés),
 - ▶ Mécanismes de sécurité plus fins (ACL),
 - ▶ Limitations plus larges sur les tailles (noms de fichiers, etc.),
 - ▶ Performance.