UE Programmation Unix

Contrôle Continu

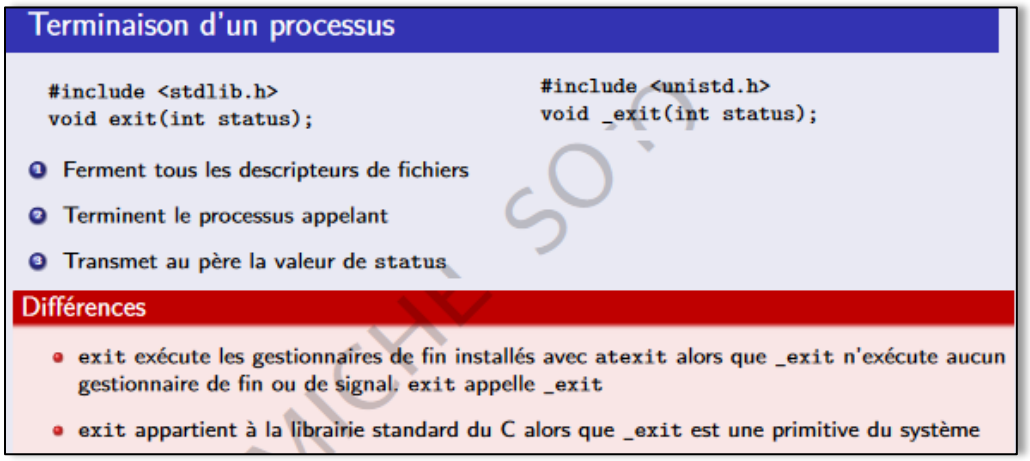
Partie 1
CONNAISSANCE DU COURS

Question 1 Vrai / Faux — 3-4 points

Gestion des Processus

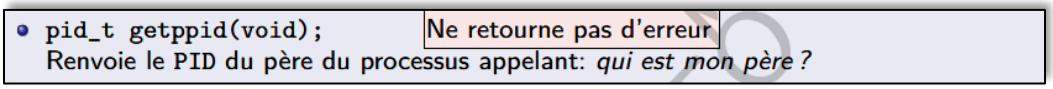
- **La primitive `_exit` exécute les gestionnaires de fin installés avec `atexit`.**

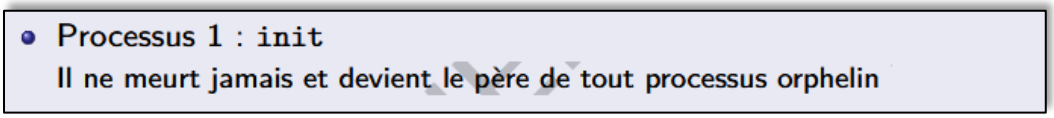
  Faux



- **Si le père d'un processus est terminé et que ce processus fait appel à la primitive `getppid` alors cette dernière retourne la valeur –1.**

  Faux





- **Un `fork` ne provoque pas la copie des données du processus père.**

  Vrai

  - copy-on-write (COW)
    - En réalité, le père et le fils partagent l'espace mémoire du père en lecture seule
    - La copie n'est effectuée que lorsque le père ou le fils tentent de modifier cet espace partagé et seule la page concernée est copiée

- **Soit un processus F fils d'un processus P. Si P ou F s'apprête à modifier une de ses variables, le mécanisme de `copy-on-write` effectue, avant la modification, une copie intégrale de P (moins les caractéristiques qui ne sont jamais héritées).**

  Faux

  - copy-on-write (COW)
    - En réalité, le père et le fils partagent l'espace mémoire du père en lecture seule
    - La copie n'est effectuée que lorsque le père ou le fils tentent de modifier cet espace partagé et seule la page concernée est copiée

- **Un processus zombie ne devient pas orphelin quand sont père se termine.**

  Vrai

  En informatique, sous les systèmes de type UNIX et similaires, **zombie** (on utilise plutôt l'orthographe anglaise) est un terme désignant un processus qui s'est achevé, mais qui dispose toujours d'un identifiant de processus (`PID`) et reste donc encore visible dans la table des processus.

  La seule manière d'éliminer ces processus zombies est de causer la mort du processus père, par exemple au moyen du signal `SIGKILL` . Les processus fils sont alors automatiquement rattachés au processus n°1, généralement `init` .

- **Par défaut, les fichiers ouverts sont fermés après l'appel à une primitive `exec`.**

  Faux

Gestion des Signaux

- **La primitive `signal` est utilisée pour envoyer un signal à un processus.**

Faux

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

Envoie un signal à un processus ou un groupe de processus

```
void (*signal(int signo, void (*func)(int)))(int);
```

Installe le gestionnaire spécifié par func pour le signal signo.

- **Un processus peut attendre l'arrivée d'un signal en utilisant la primitive `wait`.**

Faut

Attendre un signal

```
#include <unistd.h>
int pause(void);
```

Suspend le processus appelant jusqu'à l'arrivée d'un signal quelconque
Retourne -1 et errno = EINTR si un signal a été capté et que le gestionnaire du signal s'est terminé

- Un processus père peut attendre et s'informer de la terminaison de ses fils grâce aux fonctions wait et waitpid
  - Ces primitives fonctionnent en tandem avec les fonctions exit et _exit

- **Tout signal peut être capté.**

Faut

2) Envoi et réception des signaux

Un signal dit **pendant** est envoyé par un processus ou le noyau. Le signal est dit délivré lorsque le processus réalise l'action:

- réaliser l'action par défaut,
- ignorer le signal,
- réaliser l'action définie par l'usager (hnadle), on dit que le signal est **capté** dans ce cas.

Source : http://www.doritique.fr/Articles/View_Article.php?num_article=45

- **Par défaut, plusieurs instances d'un même signal peuvent être pendants**

Faut

Définition

Signal ***pendant***
Le signal émis n'a pas encore été pris en compte

ATTENTION

Une seule occurrence d'un même type de signal peut être pendante

- **Un signal reçu par un processus indique de façon certaine que l'évènement associé à ce signal s'est produit.**

Faut

On peut envoyer un signal via la primitive `kill()`

- **Après l'exécution du code le processus bloque seulement SIGALRM.**

```
sigemptyset(&set);
sigaddset(&set, SIGALRM);
sigprocmask(SIG_SETMASK, &set, NULL);
```

Vrai

`int sigemptyset(sigset_t *set);` Initialise à VIDE l'ensemble de signaux pointé par set

`int sigaddset(sigset_t *set, int signo);` Ajoute le signal signo à l'ensemble de signaux pointé par set

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

- Si oldset ≠ NULL, le masque courant du processus est retourné à travers oldset.
- Si set ≠ NULL alors how indique comment le masque courant est modifié :
  - SIG_SETMASK → *nouveau_masque* = {set}

- **La primitive `sigsuspend` permet de changer de manière provisoire le masque des signaux du processus appelant.**

Vrai

```
int sigsuspend(const sigset_t *sigmask);
```

- Remplace temporairement le masque courant des signaux bloqués par sigmask

- **Pendant l'exécution du gestionnaire d'un signal capté, ce même signal est ignoré.**

Vrai

Installation d'un gestionnaire avec `signal`

`signal` est l'interface historique. Son comportement varie selon les systèmes et les versions de système.

Pendant l'exécution d'un gestionnaire

- sur SysV : le handler peut être interrompu pour le même signal
- sur BSD : le signal qui a provoqué l'exécution du gestionnaire est masqué

Utiliser les fonctions POSIX !

Installation d'un gestionnaire avec `sigaction`

```
int sigaction(int signo, const struct sigaction *act, struct sigaction *oldact);
struct sigaction {
                void (*sa_handler)();
                sigset_t sa_mask;
                int sa_flags;
}
```

- `sa_mask` : signaux additionnels à bloquer au cours de l'exécution du gestionnaire
  Si le gestionnaire retourne, le masque des signaux est restauré à sa valeur précédente
  - `signo` est **toujours masqué**
  - d'autres signaux peuvent être ajouter en plus de `signo` (avec `sigemptyset`, `sigaddset`, etc)

Gestion des Fichiers

- **Le système gère une table `file table` par processus.**

Faut

Tables en mémoire exploitées par le noyau

`u_ofile`

- Descripteurs standards dans la `u_ofile`
  - 0 (`stdin`) : flux correspondant à l'entrée standard
  - 1 (`stdout`) : flux correspondant à la sortie standard
  - 2 (`stderr`) : flux correspondant à la sortie d'erreur standard
- Exemple

```
...
fprintf(stderr,"erreur numero %d", errno);
…
```

- Chaque entrée associée à un fichier ouvert pointe vers une entrée de la table des ouvertures de fichiers: `file table`

Tables en mémoire exploitées par le noyau
`file table`

- Contient les informations sur tous les fichiers ouverts dans le système par l'ensemble des processus à un instant donné.
  - Une entrée utilisée par ouverture de fichier
- Permet à plusieurs processus de même filiation de partager un fichier ouvert

- **Le système gère une table `u_ofile` par processus**

Vrai

- **`u_ofile`:** table des descripteurs de fichiers d'un processus
  - Chaque processus possède la sienne
    - Vision PAR PROCESSUS des fichiers ouverts

- **Avant l'exécution de la toute première instruction d'un processus, la première entrée libre de sa `u-ofile` est l'entrée n°3**

Vrai

- Descripteurs standards dans la `u_ofile`
  - 0 (`stdin`): flux correspondant à l'entrée standard
  - 1 (`stdout`): flux correspondant à la sortie standard
  - 2 (`stderr`): flux correspondant à la sortie d'erreur standard

- **La table des `i-nodes` est une copie en mémoire centrale de la `i-list` située sur le disque.**

Vrai

Tables en mémoire exploitées par le noyau
i-node table

- Permet la localisation physique du fichier.
- L'i-node de la i-list du fichier ouvert est chargée dans une entrée de cette table lors de sa première ouverture.
  - Vision GLOBALE des fichiers ouverts
  - Une entrée utilisée par fichier ouvert

La primitive `stat` retourne les statistiques d'utilisation d'un fichier

Vrai

- Obtention des attributs d'un fichier

```
#include <sys/stat.h>
#include <unistd.h>
    int stat(const char *file_name, struct stat *buf);
    int fstat(int filedes, struct stat *buf);
    int lstat(const char *file_name, struct stat *buf);
```

struct stat

```
dev_t     st_dev;       /* device file resides on */
ino_t     st_ino;       /* the file serial number */
mode_t    st_mode;      /* file mode */
nlink_t   st_nlink;     /* number of hard links to the file */
uid_t     st_uid;       /* user ID of owner */
gid_t     st_gid;       /* group ID of owner */
dev_t     st_rdev;      /* the device identifier (special files only)*/
off_t     st_size;      /* total size of file, in bytes */
time_t    st_atime;     /* file last access time */
time_t    st_mtime;     /* file last modify time */
time_t    st_ctime;     /* file last status change time */
long      st_blksize;   /* preferred blocksize for file system I/O*/
long      st_blocks;    /* actual number of blocks allocated */
```

- **La primitive `open` a pour unique fonction d'ouvrir un fichier.**

Faut

Autres fonctions de la primitive open :

Available Values for `oflag`

Value	Meaning
O_RDONLY	Open the file so that it is read only.
O_WRONLY	Open the file so that it is write only.
O_RDWR	Open the file so that it can be read from and written to.
O_APPEND	Append new information to the end of the file.
O_TRUNC	Initially clear all data from the file.
O_CREAT	If the file does not exist, create it. If the O_CREAT option is used, then you must include the third parameter.
O_EXCL	Combined with the O_CREAT option, it ensures that the caller *must* create the file. If the file already exists, the call will fail.

- **Plusieurs processus ne peuvent pas écrire dans un même tube ordinaire.**

Faut

Plusieurs lecteurs / écrivains peuvent, respectivement, lire/écrire dans un même tube.

- **La pose, par un processus P, d'un verrou *exclusif* en mode *advisory* sur une zone de fichier interdit l'accès a cette zone a tout autre processus que P**

Faut [?]

- Le type du verrou
  - partagé (*shared*) : plusieurs verrous de ce type peuvent cohabiter
  - exclusif (*exclusive*) : un verrou de ce type ne peut cohabiter avec aucun autre verrou (*exclusif* ou *partagé*)

- Le mode opératoire du verrou
  - coopératif/consultatif (*advisory*) : pas d'influence sur le E/S
  - impératif (*mandatory*) : influence sur les E/S

- **Si un verrou *partagé advisory* est posé sur zone d'un fichier aucun autre processus ne peut accéder à cette zone.**

Faut

- **Si un verrou *partagé mandatory* est posé sur zone d'un fichier aucun autre processus ne peut accéder à cette zone.**

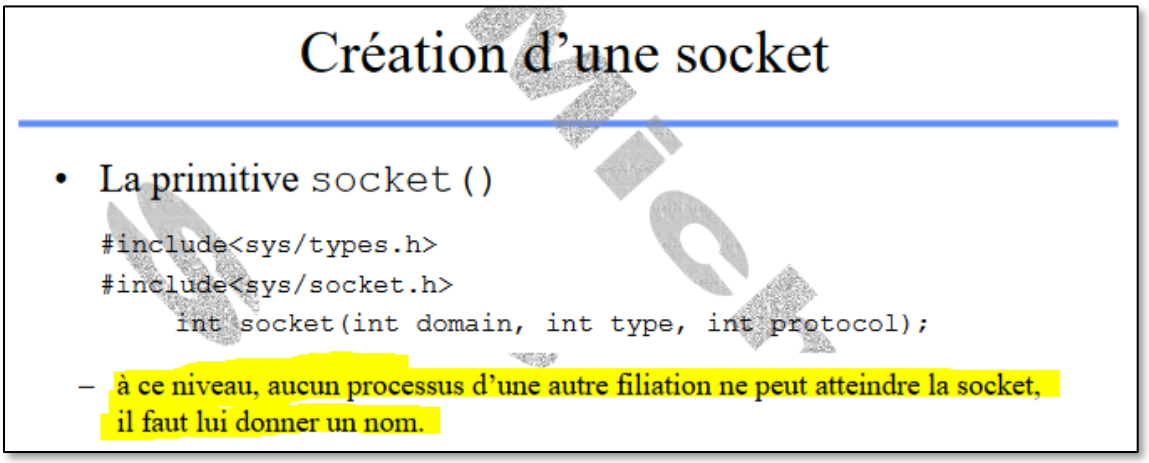
  ?

Les sockets

- **Il n'est pas possible d'utiliser une socket avant de l'avoir nommée.**

  Faut

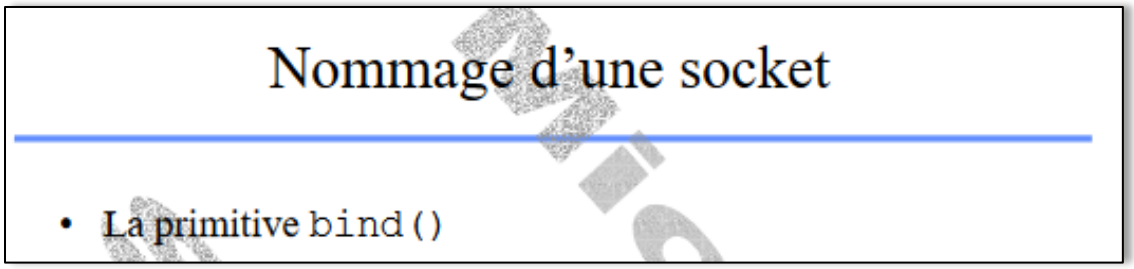
  Apres l'appel à la primitive socket, je me retrouve avec une socket qui a été créé mais quant à l'utilisation de cette socket, appart des processus de la même filiation, personne ne peut utiliser cette socket.



- **Le nommage d'une socket est une opération qui associe une chaine de caractère a une socket.**

  Vrai

  Le but de la primitive `bind` est d'associer une adresse à un objet socket. A partir de ce moment-là on va pouvoir utiliser cette socket avec un spectre de filiation qui dépasse le simple cadre de la machine sur laquelle elle a été crée et qui dépasse le simple cadre de la filiation des processus dans laquelle cette socket a été créé.



- **Une connexion sur un serveur s'établit et se poursuit sur la socket qui a fait l'objet du `bind`.**

  Faut

Autres

- **Tout thread se termine quand le processus qui l'a créé se termine.**
- **Tous les threads se terminent lorsque le processus auquel ils appartiennent se termine.**
- **Sous Linux un `fork` effectué par un thread a pour effet de cloner tous les threads du processus.**

Autres

- **L'appel aux primitives `wait`, `sigsuspend`, `read` est bloquant par défaut pour le processus appelant.**

  Faut

  `read() - pas bloquant [ ? ]`

```
pid_t wait(int *status);
```
- Bloque le processus père si *tous* ses fils sont en cours d'exécution

```
int sigsuspend(const sigset_t *sigmask);
```
- Met en sommeil (suspend) le processus jusqu'à l'arrivée soit :

Question 2 Questions de cours 2-5 points

Gestion des Processus

- **Expliquez <u>très brièvement</u> pourquoi la primitive fork ne retourne pas son identité au processus fils.**

```
#include <unistd.h>
pid_t fork(void);
```

Crée un nouveau processus
Retourne :

- Dans le processus créé et appelé *processus fils*: **ZÉRO**
- Dans le processus appelant et appelé *processus père*: le **PID du fils créé**
- En cas d'erreur, chez le processus père: -1

- **Citez 2 façons dont un processus zombie peut disparaître du système.**

> Les états les plus connus sont l'état **R** (en cours d'exécution), **S** (en sommeil), **T** (stoppé) ou encore **Z** (zombie). Ce dernier est particulier, car il désigne un processus qui, bien qu'ayant **terminé son exécution**, reste présent sur le système, en **attente** d'être **pris en compte** par son père.

1er façons dont un processus zombie peut disparaître du système

Le processus père est prévenu lorsque son fils vient de finir sa tâche et va récupérer, à l'aide des primitives wait() ou waitpid(), le code de retour de son fils terminé. Le père cumulera alors les statistiques de son fils avec les siennes et supprimera son entrée de la table des processus, le fils pourra alors totalement être effacé du système.

2em façons dont un processus zombie peut disparaître du système

> Si le processus père n'a pas été conçu pour réceptionner le code de retour de chaque processus fils qu'il crée (à l'aide des primitives **wait()** ou **waitpid()**), les processus fils resteront à l'état zombies pendant toute la durée d'exécution du processus père.
>
> On ne peut pas se débarrasser des processus zombies ... Ils sont déjà morts... La commande kill n'a aucun effet sur eux.

Le seul recours possible est de directement mettre un terme au processus père, avec par exemple la commande kill. Les processus

fils zombies seront alors adoptés par init et ce dernier se chargera de les supprimer de la table des processus.

Source : https://www.it-connect.fr/les-processus-zombies/

- **Complétez le programme suivant afin qu'il affiche le contenu de sa variable d'environnement PATH**

```
main(int argc, char *argv[], char **arge){
...
} // main
```

```
#include <stdio.h> // printf()
#include <string.h> //strstr()
int main(int argc, char *argv[], char **arge) {
        int i;
        for(i = 0 ; arge[i] != NULL ; i++) {
                if( strstr(arge[i], "PATH") != NULL)
                        printf(" %s \n", arge[i]);
        } // for


} // main
```

Source : http://www2.cs.uidaho.edu/~krings/CS270/Notes.S10/270-F10-20.pdf

- **Donnez le résultat de l'exécution du programme suivant. Justifiez votre réponse.**

```
main(){
  int i;
  for (i = 1 ; i <= 2;i++){
     execl("ls","ls",”-l”,NULL);
  }
}
```

```c
#include <unistd.h> // execl
int main() {
        int i;
        for(i = 1 ; i <= 2 ; i++) {
                execl("ls", "ls", "-l", NULL);
        }
} // main
```

```
$ nano test.c
$ gcc test.c -o test
$ ./test
$
```

Il ne se passe rien

```c
#include <unistd.h>
int execl  (const char *path, const char *arg0, ... /* (char *)0 */);
int execlp (const char *file, const char *arg0, ... /* (char *)0 */);
int execle (const char *path, const char *arg0, ... /* (char *)0 */,
                                                    char *const envp[]);
int execv  (const char *path, char *const argv[]);
int execvp (const char *file, char *const argv[]);
int execve (const char *path, char *const argv[], char *const envp[]);
```

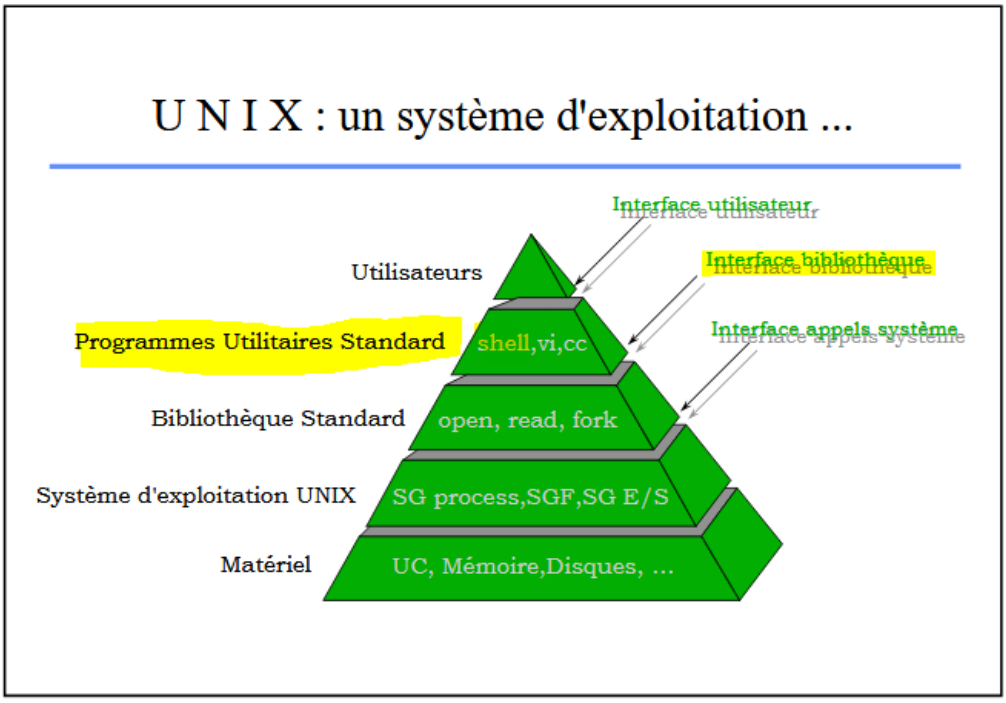
- La transmission des arguments se fait soit :
  - par liste (suffixes l)
  - par un tableau de pointeur sur des chaines de caractères (suffixes v)
- Le fichier à exécuter est soit :
  - recherché en utilisant la variable d'environnement PATH (suffixe p)
  - indiqué dans le paramètre path (absence de suffixe p)
- L'environnement peut :
  - être modifié (suffixe e)
  - être conservé (absence de suffixe e)

Il ne se passe rien car `execl` n'utilise pas la variable `PATH` et donc ne trouve pas l'exécutable de `ls` qui se trouve dans `/bin`

Version qui marche

```c
#include <unistd.h> // execl
int main() {
        int i;
        for(i = 1 ; i <= 2 ; i++) {
                execl("/bin/ls", "ls", "-l", NULL);
        }
} // main
```

- **Le shell at-il un status particulier du point de vue du système ? Justifiez votre reponse**



Shell

Un interpréteur de commandes :ne fait pas partie du système d'exploitation (c'est un processus comme les autres qui l'utilise).

Source : https://www.fil.univ-lille1.fr/~sedoglav/SHELL/Cours01-2x3.pdf

- **Sur quelle propriété du système repose la technique du double fork évitant de l'accumulation de processus zombie pour les commandes lancées en arrière-plan ?**

Double fork to avoid zombie process

http://thinkiii.blogspot.com/2009/12/double-fork-to-avoid-zombie-process.html

La technique du double fork évitant l'accumulation de processus zombie repose sur une propriété du système :

- Processus 1 : `init`
  Il ne meurt jamais et devient le père de tout processus orphelin

Gestion des Fichiers

- **Un processus fait un appel à la primitive `open()`**
  Quelles sont les tables du système qui sont concernées par cet appel.
  - `u_ofile`
  - `file table`
  - `i-node table`

Tables en mémoire exploitées par le noyau
u_ofile

- **u_ofile:** table des descripteurs de fichiers d'un processus
  - Chaque processus possède la sienne
    - Vision PAR PROCESSUS des fichiers ouverts
- Lors de l'ouverture d'un fichier, le noyau lui associe une entrée dans cette table
- Chaque entrée associée à un fichier ouvert pointe vers une entrée de la table des ouvertures de fichiers: file table

Tables en mémoire exploitées par le noyau
file table

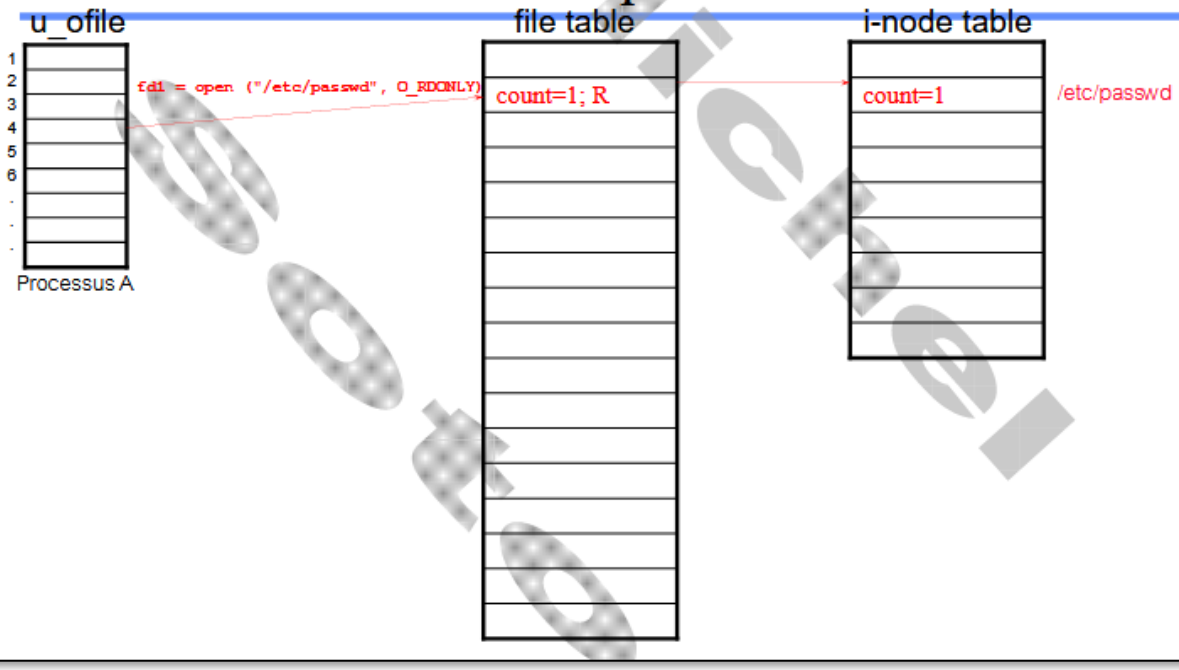
- Contient les informations sur tous les fichiers ouverts dans le système par l'ensemble des processus à un instant donné.
  - Une entrée utilisée par ouverture de fichier
- Permet à plusieurs processus de même filiation de partager un fichier ouvert

```
struct file{
  char f_flag   <-- mode d'ouverture écriture, lecture, pipe
  cnt_t f_count <-- nombre de processus qui accèdent au fichier
  struct inode *f_inode <-- pointeur vers la table des i-nodes
    .........
}
```

Tables en mémoire exploitées par le noyau
i-node table

- Permet la localisation physique du fichier.
- L'i-node de la i-list du fichier ouvert est chargée dans une entrée de cette table lors de sa première ouverture.
  - ➢ Vision GLOBALE des fichiers ouverts
  - ➢ Une entrée utilisée par fichier ouvert

u_ofile, file table, i-node table
exemples



- **Citez les types de fichier gérés par les systèmes de la famille UNIX ?**

Types de fichiers (cours p. 24 – 25)

Le type du fichier détermine les opérations possibles.
Le type de fichier est encodé dans le champs `st_mode` de la structure `stat`.

→Fichier régulier ou ordinaire

non structuré
contenu : texte, binaire, image, document, etc.

→Répertoire

nœud de l'arborescence
Contenu : fichiers réguliers, répertoires

→FIFO (tube)

communication unidirectionnelle entre processus d'une même machine.

→Socket AF_UNIX

→Lien symbolique

Contenu : un nom de fichier.

→Fichier spécial

- Périphérique

- **Mode bloc : disque**
- **Mode caractère : clavier, écran**

- **Ennoncez les differences qui existent entre un fichier régulier et un tube ordinaire.**

Différences entre un fichier régulier et un tube ordinaire

Contrairement à un fichier, tube ordinaire :

→ **Supprimé lorsque aucun processus ne l'utilise**
→ **Impossibilité d'ouvrir un tube (pas de `open()`)**
→ **Opération interdite : `lseek`**
→ **Lecture destructrice**
→ **Communication entre processus ayant un ancêtre commun**

- **Soient deux fichiers appartenant à Francois du groupe users tels que décrits ci-dessous. `a.out` est un programme qui ouvre en lecture/écriture (`O_RDWR`) le fichier `Donnees`.**

```
-rwsr-xr-x    1      francois      useres 11687  déc    2      14:12  a.out
----rw-r--    1      francois      useres    44  déc    2      14:09  Donnees
```

Francois et Pierre sont du même groupe. Le programme `a.out` peut-il ouvrir le fichier `Donnees` s'il est exécuté par :

a. **Francois ? Non car les droit de Francois pour `Donnees` sont `---`**

b. **Pierre ? Non car même si les droit de Pierre pour `Donnees` sont `rw-` ,le droit SUID est ajouter à l'exectuable `a.out` et les droit du propriétaire (Francois) sont `---`**

Le mode d'accès (O_RDWR) ne correspond pas aux droits d'accès du propriétaire.

Utilisation [modifier | modifier le code]

Pour voir quels droits sont attribués à un fichier, il suffit de taper la commande ls -l nom_du_fichier :

```
# ls -l toto
-rwxr-xr--    1 user     group     12345 Nov 15 09:19 toto
```

La sortie signifie que le fichier toto (de taille 12345) appartient à « user », qu'on lui a attribué le groupe « group », et que les droits sont rwxr-xr--. On remarque qu'il y a en fait 10 caractères sur la zone de droits. Le premier - n'est pas un droit, c'est un caractère réservé pour indiquer le type de fichier. Il peut prendre les valeurs suivantes :

- `d` : répertoire
- `l` : lien symbolique
- `c` : périphérique de type caractère
- `b` : périphérique de type bloc
- `p` : pipe (FIFO) pour "tube" ou "tuyau" en anglais ou pipeline aussi en 'français'.
- `s` : socket
- `-` : fichier classique

Droit SUID

Ce droit s'applique aux fichiers exécutables, il permet d'allouer temporairement à un utilisateur les droits du propriétaire du fichier, durant son exécution. En effet, lorsqu'un programme est exécuté par un utilisateur, les tâches qu'il accomplira seront restreintes par ses propres droits, qui s'appliquent donc au programme. Lorsque le droit SUID est appliqué à un exécutable et qu'un utilisateur quelconque l'exécute, le programme détiendra alors les droits du propriétaire du fichier durant son exécution. Bien sûr, un utilisateur ne peut jouir du droit SUID que s'il détient par ailleurs les droits d'exécution du programme.

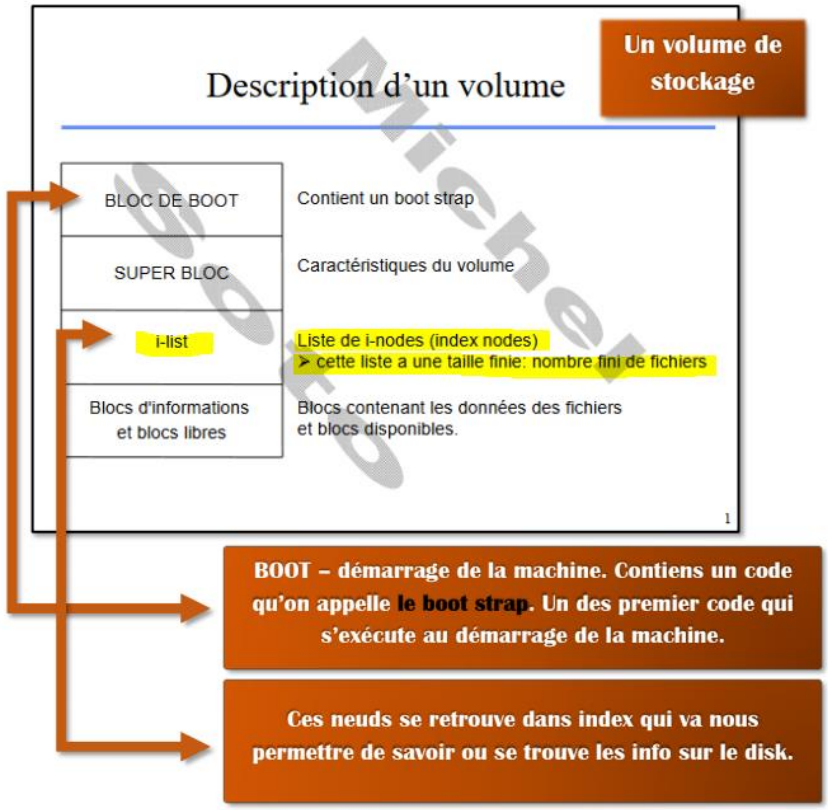
Notation [modifier | modifier le code]

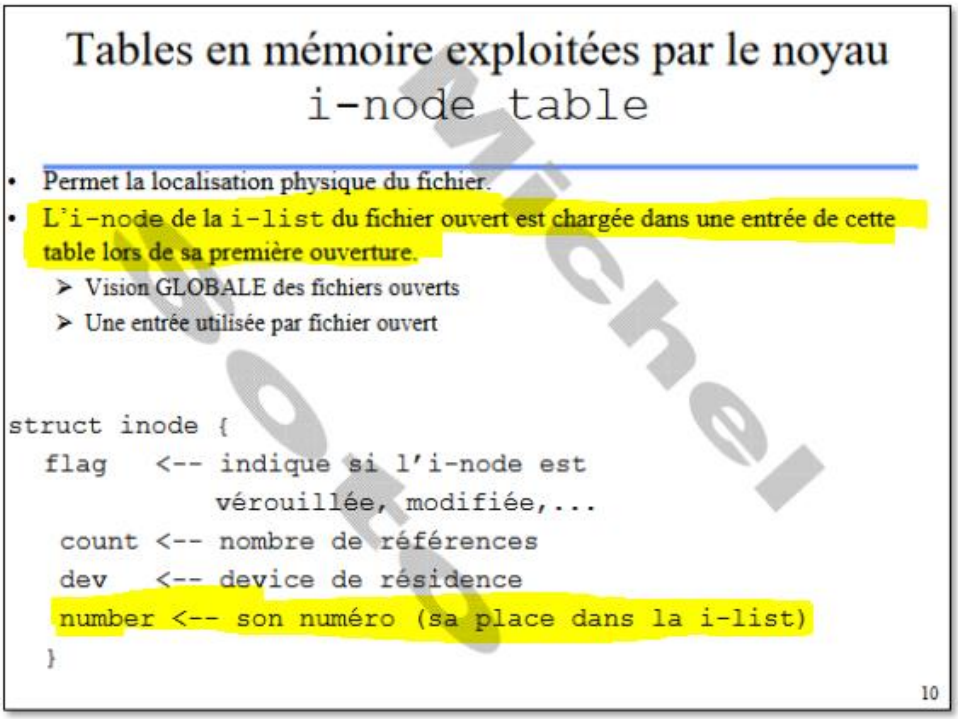
Son flag est la lettre s ou S qui vient remplacer le x du propriétaire. La majuscule ou la minuscule du 's' permet de connaitre l'état du flag x (droit d'exécution du propriétaire) qui est donc masqué par le droit SUID 's' ou 'S': C'est un s si le droit d'exécution du propriétaire est présent, ou un S sinon. Il se place donc comme ceci :

```
---s------  ou  ---S------
```

Un fichier avec les droits `-rwxr-xr-x` auquel on ajoute le droit SUID aura donc la notation `-rwsr-xr-x`

Qu'est-ce que qu'une i-list et a quoi sert-elle ?





Les sockets

- **Que demande l'utilisateur lorsqu'il écrit**

```
sd = socket(AF_INET, SOCK_STREAM, 0)
```

dans un programme ?

La primitive socket

Tous processus qui aura besoin de communiquer avec un processus distant qui se trouve sur une autre machine (machine qui est connecté bien sûr à un réseau qui permet de l'atteindre), va devoir créé une socket (socket c'est une prise en anglais, une prise a laquelle on se branche..) et donc la création d'une socket passe par une primitive du même nom qui permet de le faire.

- La primitive `socket()`

```
#include<sys/types.h>
#include<sys/socket.h>
     int socket(int domain, int type, int protocol);
```

  - retourne un descripteur de fichier,
  - si `protocol = 0` → le système choisit le protocole

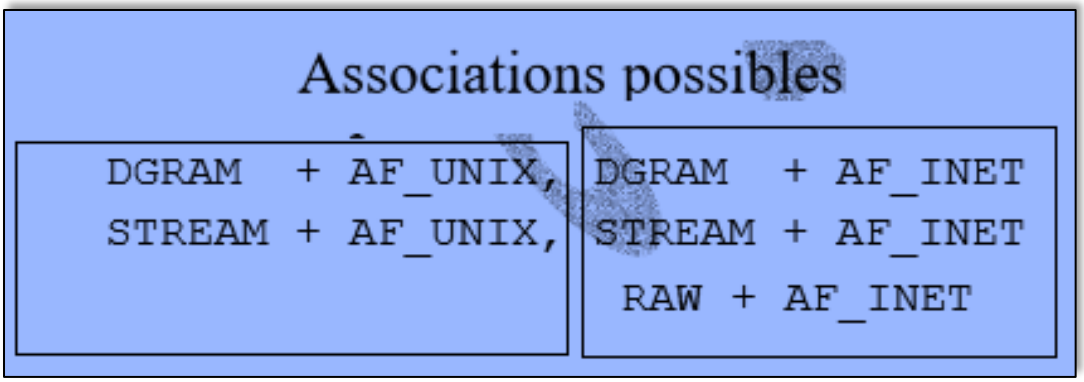
On a 3 paramètres :

1. **Domaine - le domaine ça va faire référence en fait au type d'adresse : adresses internet, adresses UNIX, adresse IPX (existe plus ajd)..**

Domain	Description
AF_INET	IPv4 Internet domain
AF_INET6	IPv6 Internet domain (optional in POSIX.1)
AF_UNIX	UNIX domain
AF_UNSPEC	unspecified

2. **Type – de quelle type de socket on a besoin ? ce Type il fait référence a la qualité de service qu'on a besoin pour notre application. En gros, ajd dans l'architecture TCP/IP y'a 2 type de service : le service connecté et le service non-connecté. A nous de voir laquelle des 2 correspond au besoin de notre application.**

Type	Description
SOCK_DGRAM	fixed-length, connectionless, unreliable messages
SOCK_RAW	datagram interface to IP (optional in POSIX.1)
SOCK_SEQPACKET	fixed-length, sequenced, reliable, connection-oriented messages
SOCK_STREAM	sequenced, reliable, bidirectional, connection-oriented byte streams



3. **Protocol** - **Le Protocol qui va être utiliser pour la communication entre les processus qui vont communiquer à travers les sockets. On peut mettre une valeur si on veut, mais si on mais 0 c'est le system qui choisit pour nous le Protocol qui correspond au type de service qu'on a demander (2em paramètre). En pratique on met toujours 0.**

   Par exemple : si on mai SOCK_STREAM, le Protocol qui va être choisi automatiquement par le system ça va être TCP.

Quand je fais appel a la primitive socket je récupère un entier. Cette entier on l'appelle « un descripteur de socket ».

Quand on crée une socket, le descripteur qui va être retourner si tous va bien c'est un entier et cette entier a la même sémantique que l'entier qui est retourner par un `open()` . c'est le numéro de l'entrée de la `u_ofile` du processus qui a été utiliser pour la création de cette socket.

- Soit le programme suivant :

```
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/socket.h>
main(void) {
int fdA, sd ;
/* 1 */ fdA = open("fichA", O_WRONLY) ;
/* 2 */ dup2(fdA, 1) ;
/* 3 */ printf("coucou \n") ;
/* 4 */ sd = socket(AF_INET, SOCK_STREAM, 0) ;
/* 5 */ dup2(sd, 0) ;
/* 6 */ sleep(2) ;
}/* main */
```

a}Representez par un schema l'etat de la u_ofile avant l'execution de la ligne 1
b) Representez par un schema l'etat de la u_ofile après l'execution des lignes 1 a 5. Justifiez votre reponse

c) Quel est le resultat obtenu après de l'execution de ce programme. Justifiez votre reponse.

Question 3 (3 points – 10 mn)

Soit le programme suivant :

```
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>

main(void) {
    int fdA, sd;
```

```
    /* 1 */ close(1);
    /* 2 */ fdA = open("fichA", O_RDONLY);
    /* 3 */ dup2 (fdA,3);
    /* 4 */ sd=socket(AF_INET, SOCK_STREAM, 0);
    /* 5 */ dup2 (sd, 0);
    /* 6 */ write(1,"Hello",5);
} /* main */
```

a) Représentez par un schéma l'état de la u_ofile avant l'exécution de la ligne /* 1 */
b) Représentez par un schéma l'état de la u_ofile après l'exécution de la ligne /* 5 */
c) Quel sera le résultat de l'exécution de la ligne /* 6 */ ? justifiez votre réponse.

Question 3 (4 points – 12 mn)

Soit le programme suivant :

```
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/socket.h>

main(void) {
    int fdA, sd;
```

```
    /* 1 */ fdA = open("fichA", O_WRONLY);
    /* 2 */ dup2 (fdA,1);
    /* 3 */ printf("Hello\n");
    /* 4 */ sd=socket(AF_INET, SOCK_STREAM, 0);
    /* 5 */ dup2 (sd, 0);
} /* main */
```

a) Représentez sur un schéma l'état de la u_ofile avant l'exécution de la ligne /* 1 */
b) Représentez sur un schéma l'état de la u_ofile après l'exécution de la ligne /* 5 */
c) Quel sera le résultat du programme après son exécution ?

Question 2 **(3 points – 3 mn)**

La commande `./a.out x`, où x est un entier, lance le programme ci-dessous. En supposant que le tube soit créé avec succès, indiquez tous les comportements possibles de ce programme en fonction de x? Justifiez votre réponse.

```
#include <stdio.h>
#include <stdlib.h>

int p[2], n, i, n_iter, n_tot=0;
void main(int argc, char **argv){
    pipe(p);
    n_iter = atoi(argv[1]);
    for (i=0; i < n_iter; i++) {
      n = write(p[1], "c", 1);
      n_tot = n_tot + n;
    }// for
    printf ("%d caractères\n", n_tot);
}// main
```

Question 2 **(4 points)**

a) A quoi sert le bit `set-uid` (bit `u`) du champ `st_mode` lorsqu'il est positionné sur un fichier contenant un exécutable ? Citez une commande du système qui possède le bit `set-uid` positionné sur son exécutable. Précisez à quoi sert cette commande.

b) Qu'est-ce qu'un thread joignable ?

c) Citez 2 façons dont un processus zombie peut disparaître du système.

a