Morgan Valentin

Questions OS Théorie 1ère année :

Chapitre 1 : 2

1. Expliquer ce qu'un **inode**. 2

2. Expliquer les **buffers cache** et donné les 5 cas pour leurs allocations. 3

3. Expliquer la **lecture asynchrone.** 7

4. Allocation des inodes en **mémoire** (centrale). 8

5. Désallocation des inodes en **mémoire** : 9

6. Comment peut-on retrouver les **données d'un fichier sur disque** ? 10

7. Expliquer les **directories** (algo ***namei***) 12

8. Allocation des inodes sur **disque**. 13

9. Désallocations des inodes sur **disque** : 15

10. Allocation et désallocations des **blocs** (sur disque) : 16

11. Expliquer comment fonctionne l'algorithme **Open**. 17

12. Expliquer différence entre **FIFO** et **PIPE**. 18

13. Expliquer où, quand et comment détecter une **erreur du mode d'ouverture**. 18

14. Expliquer les 2 cas de l'erreur « **no more inode** ». 19

15. Expliquer à l'aide des buffers, **2 processus qui font une lecture sur un fichier**. 19

16. Expliquer l'appel système **write**. 19

Chapitre 2: 20

1. Expliquer la différence entre la **table des processus** et la **u-area** : 20

2. Expliquer les parties **contexte** d'un processus. 22

3. Expliquer le **contexte switch**. 23

4. Appels systèmes. 24

5. Interruptions. 26

6. Expliquer le **fork** (de manière **théorique**). 27

7. Expliquer l'algorithme **sleep**. 28

8. Gestion et traitement des **signaux**. 29

Morgan Valentin

Chapitre 1 :

1. Expliquer ce qu'un **inode**.

La représentation interne d'un fichier est donnée par un inode, qui contient plusieurs informations d'ordre « administratif » (type de fichier, permission, propriétaire, taille, etc.).

Chaque fichier a un seul inode, même s'il peut avoir plusieurs noms (chacun de ceux-ci fait référence au même inode).

- ⇨ Chaque nom est appelé un **link**.

Remarque : Le inode ne spécifie pas le(s) pathname(s) du fichier.

I.I.T (Incore Inode Table) :

Ces copies des inodes en mémoire se trouvent dans une table appelée **In-Core Inode Table**, où le noyau les manipules.

Le noyau contient également 2 autres structures de données, la **File Table** et la **User File Descriptor Table**.

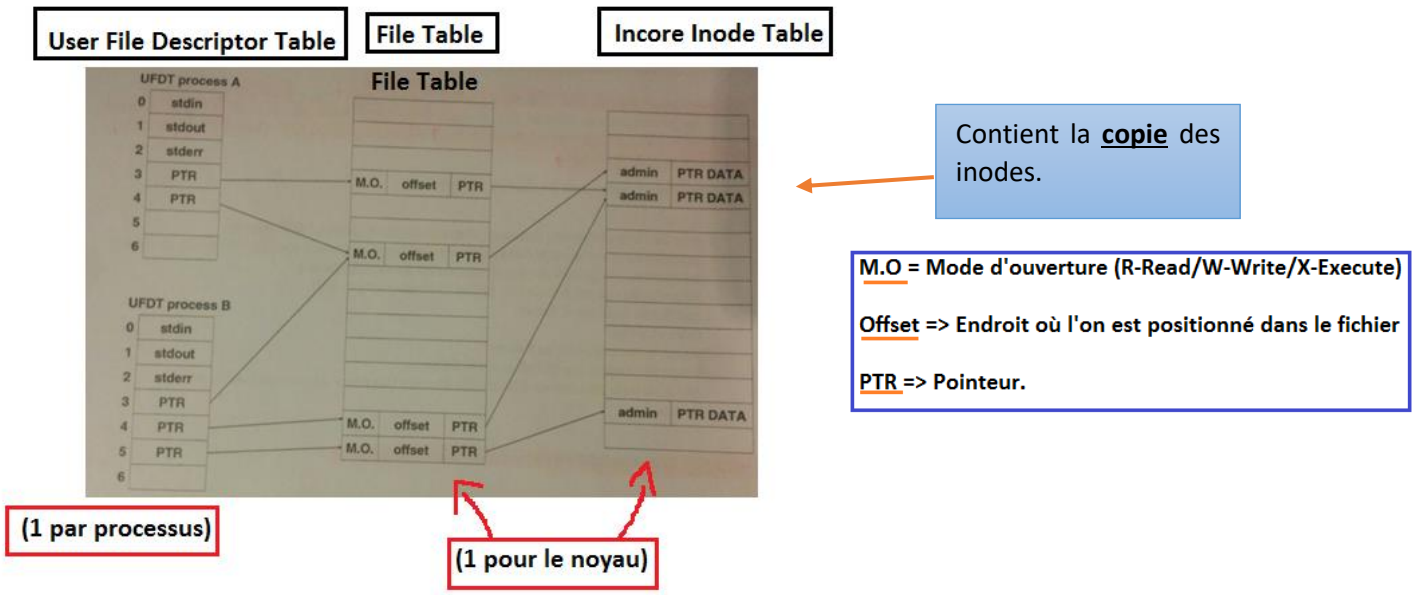
La **File Table** garde trace du déplacement en bytes dans le fichier où aura lieu la prochaine lecture ou écriture de l'utilisateur, ainsi que les droits d'accès sur le fichier pour le processus qui l'a ouvert.

Chaque entrée de cette table contient également un pointeur sur l'entrée de la **In-Core Inode Table** contenant le inode du fichier.

UFDT (User File Descriptor Table) :

La User File Descriptor Table identifie les fichiers ouverts par un processus.

- ⇨ Les 3 premières entrées de la **User File Descriptor Table** (0,1,2) font référence à des fichiers ouverts automatiquement par tout processus, les standard input, standard output et standard error.
- ⇨ Les 3 premières entrées sont traitées d'une manière différente des autres étant donné que ce sont des fichiers qui font correspondance avec les inputs et les outputs ave les claviers et les écrans. => Traitée différemment que les fichiers sur disque.



Morgan Valentin

2. Expliquer les **buffers cache** et donné les 5 cas pour leurs allocations.

Qu'est-ce que c'est ?

- C'est un espace mémoire qui est réservé et géré par le système pour contenir les blocs de données que les processus sont en train de lire et de traiter.

Vous voulez lire des données dans un fichier :

1) On va trouver le bloc de donnée.
2) On va l'amener dans un buffer en mémoire.
3) Et puis on va faire travailler votre programme (directement en mémoire et pas sur disque !).

A quoi sert le buffer cache ?

- Sert à amener les blocs de données en mémoire pour pouvoir les traiter.
- Cela permet plus de rapidité et permet de conserver des données en mémoire pendant un certain temps.

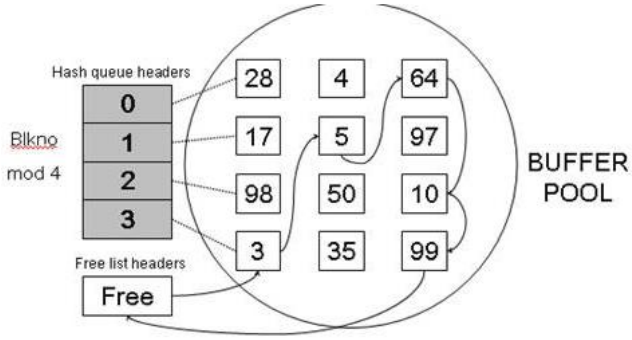
Syllabus : « Le noyau pourrait lire et écrire les données directement sur disque, mais les performances du système risqueraient d'en subir des conséquences assez fâcheuses. Dès lors, le noyau tente de minimiser la fréquence des accès disques en gardant un pool de buffers internes, appelé le buffer cache » (ou buffer pool).

Qu'est-ce qu'un buffer ?

Un buffer comporte 2 parties :

- Un tableau de mémoire contenant les données du disque. (Un bloc de donnée en mémoire)
- Un buffer header qui identifie le buffer grâce :
  - Au numéro du filesystem.
  - Au numéro du bloc sur le filesystem.
  - Au status (libre, locked, delayed-write)
  - Pointeurs :
    - Sur le buffer suivant et précédent dans la hash queue.
    - Sur le buffer suivant et précédent dans la liste libre (L.L).

Structure du buffer pool (buffer cache) :



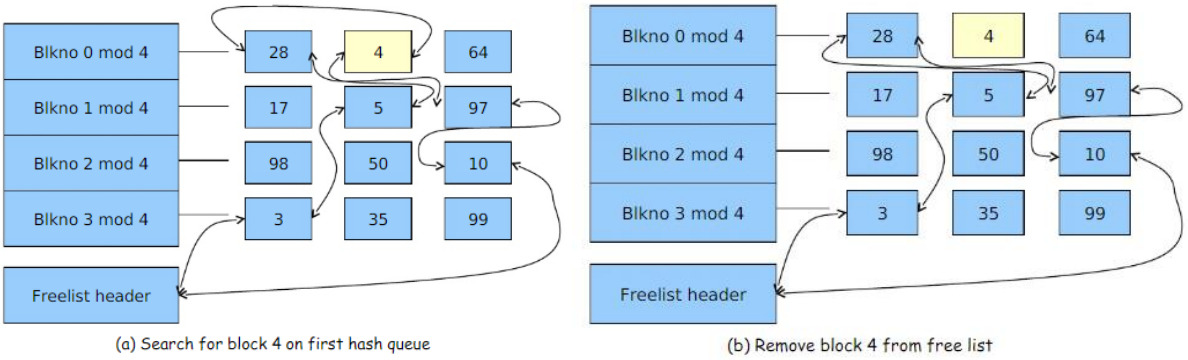
Syllabus : « Le buffer pool est géré par un algorithme basé sur le principe du **LRU** (**L**east **R**ecently **U**sed), ce qui veut dire qu'après avoir alloué un buffer à un bloc, le noyau ne peut plus utiliser ce buffer avant d'avoir employé tous ls autres. Le noyau gère une **liste des buffers libres** qui conserve cet ordre ».

Morgan Valentin

Allocation d'un buffer (getblk) :

5 cas possibles :

Premier cas : *Block dans la hash queue et un buffer libre.*



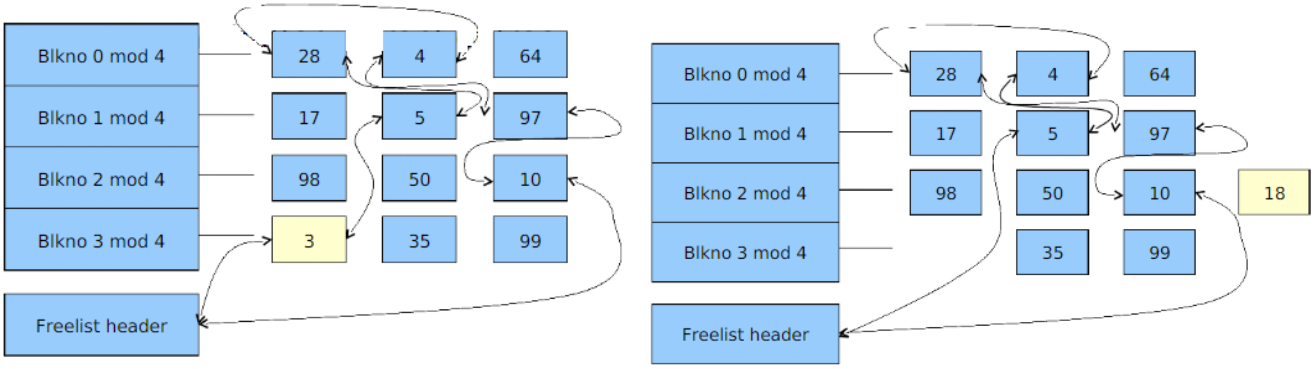
(a) Search for block 4 on first hash queue

(b) Remove block 4 from free list

a) Cherche bloc 4 sur la 1ère hash queue.

b) Retirer bloc 4 de la liste libre (L.L).

Deuxième cas *: Bloc pas dans la hash queue et un buffer libre.*



(a) Search for block 18 - Not in the cache

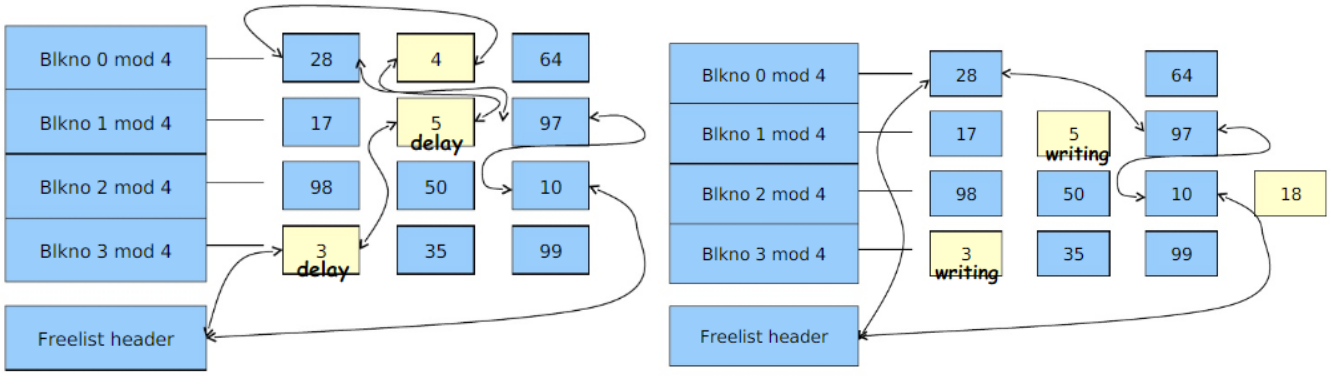
(b) Remove first block from free list, assign to 18

a) Cherche bloc 18 : Pas dans la buffer pool.

b) Retirer le 1er bloc de la liste libre (L.L) et assigner à 18.

Morgan Valentin

Troisième cas : *Bloc pas dans la hash queue et il y a des buffers marqué « delayed-write » en tête de la liste libre.*



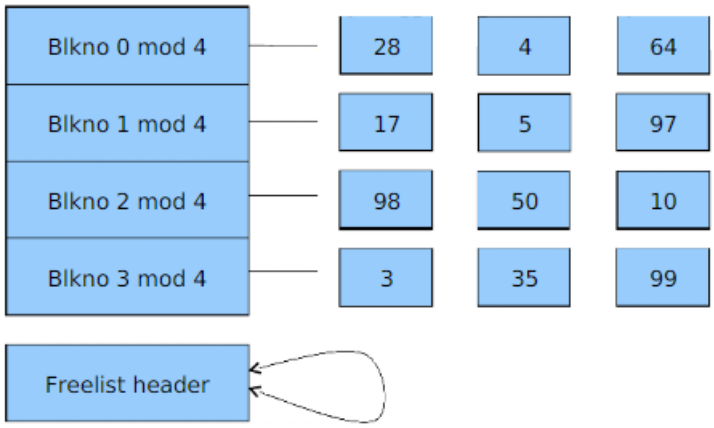
(a) Search for block 18, delayed write blocks on free list

(b) Writing block 3, 5, reassign 4 to 18

a) Cherche le block 18, lance une écriture asynchrone des blocs sur la liste libre (L.L).

b) Ecris blocs 3,5 et réassigne 4 à 18.

Quatrième cas : *Bloc pas dans la hash queue et liste libre (L.L) vide.*



(a) Search for block 18, empty free list

a) Cherche bloc 18, liste libre vide.

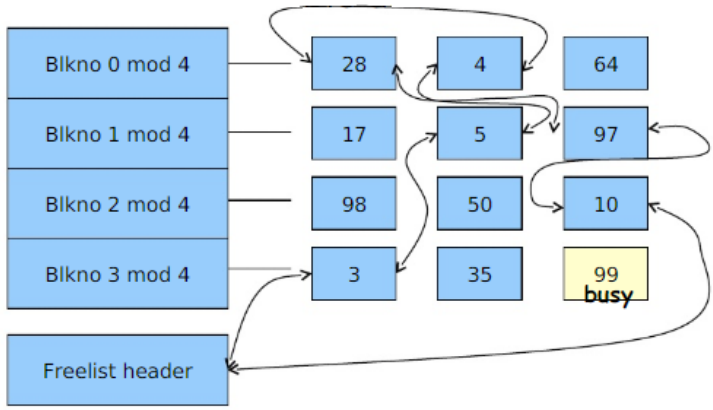
Notre processus part en « sleep » (état).

Sleep( ) => Fonction qui prends en paramètre un évènement => sleep (bufferLibre) par exemple.

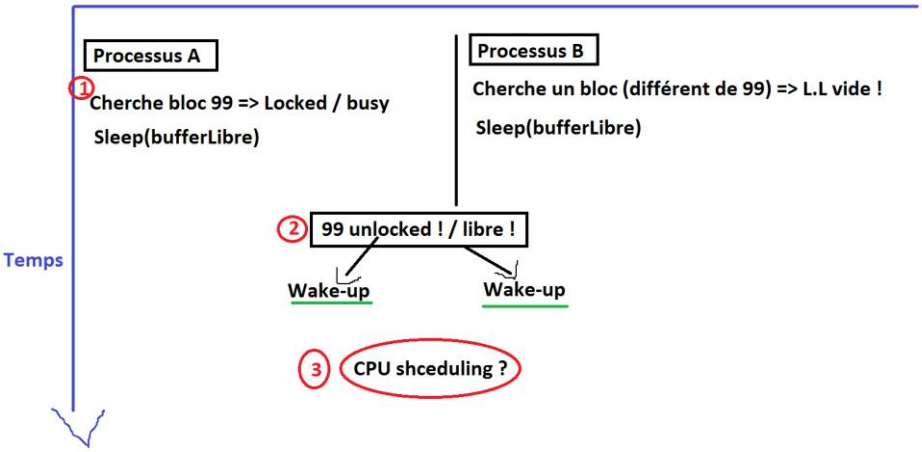
Donc, lorsqu'il y aura un buffer libre, le processus se « réveillera ».

[Morgan Valentin]

Dernier cas : Bloc dans la hash queue, mais locked / busy.



(a) Search for block 99, block busy



Etapes :

1) Le processus A va partir en sleep en attendant un buffer libre. Le processus B a besoin d'un buffer, mais vu que la liste libre est vide, il part en sleep en attendant un buffer libre.

2) Le buffer est délocké => Donc le buffer est libre !

Quand un évènement attendu se produit, les processus qui attendaient cette évènement-là, sont réveillés.

=>Processus A et B réveillé.

3) Qui va prendre le buffer ? => **On ne sait pas le dire !**

=> CPU scheduling qui va décider qui va passer en premier.

Deux cas possible :

1) Le meilleur cas, il passe le buffer libre au processus A. Parfait.
2) Il passe le buffer libre au processus B :
   - Donc le processus B va prendre le buffer libéré, va remplacer son contenu par le contenu d'un autre. Le processus B va continuer à travailler.
   - Le processus A va reprendre la main (plus en sleep) et il va remarquer que son bloc n'est plus dans la hash queue alors qu'il était !
   - Il va donc retourner en sleep ! car il n'y a plus de buffer libre.

Morgan Valentin

3. Expliquer la **lecture asynchrone.**

Principe:

Vu que la lecture disque est lente, on va faire une lecture à l'avance.

Si vous avez un très gros fichier, il y a plusieurs blocks. Et si vous voulez lire ce fichier **complètement**.

- On va lire le bloc 0, puis le 1, 2, 3 … => **Séquentiellement**.

Alors linux se dit : « Donc je vais déjà (même si l'utilisateur ne l'a pas demandé) amener la suite des fichiers dans un autre buffer comme ça si après il en a besoin, il sera déjà là ».

Quand il remarque qu'on lit 2 blocks successifs d'un même fichier, il va lancer un Read Ahead (une lecture à l'avance).

Remarque : C'est le système qui prend la décision => Lorsqu'il remarque que l'utilisateur lit 2 blocs successifs !

L'algorithme (lecture asynchrone) :

Input => Le numéro du bloc **demandé**.

=> Le numéro du bloc **suivant**.

Output => Le buffer qui contient le bloc **demandé**.

1) **Trouver un buffer pour le 1er block => getblk()**
2) **If (données du buffer PAS valide)**

   J'ai un buffer mais il contient pas les données souhaitées.

   a. **Lancer la lecture disque.**

      Prendre les données sur disque et les ramenées en mémoire.

3) **Trouver un buffer pour le 2ème block => getblk()**

   Je profite du temps de la lecture disque, plutôt que de me mettre directement en sleep pour déjà aller chercher un 2ème block au cas où l'utilisateur en aurait besoin. (décision vient du système !)

4) **If (données du buffer valide)**

   a. **Release (buffer) => Libéré le buffer (« délocker »)**

      Permet de vérifier que les données futur que vous allez demander sont en mémoire.
      getblk LOCK un buffer! On ne peut pas laisser un buffer « locké ».
      Qu'est-ce qui se passe si un buffer reste « locké » ?
      Si on n'en a pas besoin => « Locké » à vie => Inutilisable !

5) **Else**

   a. **Lancer la lecture disque.**

6) **Sleep (fin de la lecture du 1er block)**

   On attend que notre block soit amené complètement en mémoire centrale.

7) **Return (buffer)**

   On ne va pas vous faire attendre pour un bloc que vous n'avez pas demandé.

Morgan Valentin

4. Allocation des inodes en **mémoire** (centrale).

Qu'est-ce que c'est ?

C'est pour faire venir un inode en mémoire. Donc lui trouver un inode en mémoire dans la Incore Inode Table (I.I.T) et mettre le inode disque dans un inode mémoire.

L'algorithme :

Input => Le numéro d'inode (dans le file system)

Output => Inode locké.

While (pas fini)

1) **If (inode est dans la Incore Inode Table)**
    a. **If (inode est locké)**

        Cela signifie qu'un autre processus travaille sur le fichier.

        Comment ça se fait que je puisse tomber sur un cas comme ça ? (inode locké)

        Une opération d'écriture et de lecture n'est pas nécessairement faite en 1 fois !

        i. **Sleep (inode délocké)** => *Attendre que le inode soit libre.*
        ii. Continue => *Reviens vers la boucle principale.*

    b. **If (inode est dans la liste libre)**
        i. **Le retirer de cette liste**
        ii. **Incrémenter le compteur de référence du inode**

            C'est pour dire, le inode est libre mais y a déjà 1 process qui l'utilise et moi je vais utiliser le même inode, donc on sera 2.

            => Mettre le compteur à 2. « On est 2 process à utiliser le même inode / même fichier ».

        iii. **Return (inode)**
2) **If (liste des inodes libres vides)**

    Si la liste des inodes libres est vides => Incore Inode Table pleine. => Erreur de conception niveau programmation. Utilisation de trop de ressources.

    a. **Return (error)**
3) **Prendre un inode libre**
4) **Le recopier et l'initialiser**
5) **Return (inode)**

Morgan Valentin

5. Désallocation des inodes en **mémoire** :

Input => L'adresse d'un inode (pointeur sur un incore inode)

Output => Rien. (Vu que je libère un inode)

1) **Locker le inode (s'il ne l'est pas)**
2) **Décrémenter le compteur de références**

   Je supprime un fichier ou le ferme => je décrémente le compteur de référence.

3) **If (compteur de référence ==0)**

   Plus personne ne l'utilise. C'est-à-dire qu'on « ferme » le fichier.

   Attention : Fermé un fichier et supprimer un fichier libère tous les 2 l'inodes.

   a. **If (nombre de liens)**

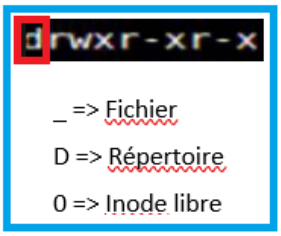
      Si le nombre de liens = 0 (rappel 1 lien = 1 chemin vers le fichier), On a fait un « rm » (remove/delete)

      i. **Libérer les blocs disques du fichier**

         Donc il faut se débarrasser des blocs de données vu qu'on a (précédemment) fait un « remove ».

      ii. **Mettre le type du fichier =0**

         Le 1er champ lorsqu'on fait un « ls -l » :



      iii. **Release (inode)**

   b. **If (accès au fichier en modification)**

      Si vous avez écris des données dans le fichier. On a alloué un nouveau bloc et on a mis son numéro dans le inode. Si j'ai supprimé des données dans le fichier => Blocs ont été supprimer.

      i. **Mettre à jour le inode disque.**

         Cela signifie recopier le inode sur disque.

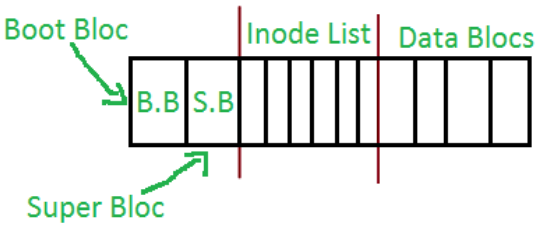
   c. **Mettre le inode sur la liste libre.**

Note supplémentaire :

Tout le travail se fait en mémoire centrale. Le inode dans Incore Inode Table a été modifié. On ferme le fichier. On voit qu'il a été modifié et on recopie le inode sur disque.

Morgan Valentin

6. Comment peut-on retrouver les **données d'un fichier sur disque** ?



Boot block :

- Lu pour initialiser le système d'exploitation.
- Chaque file system dispose d'un boot block.

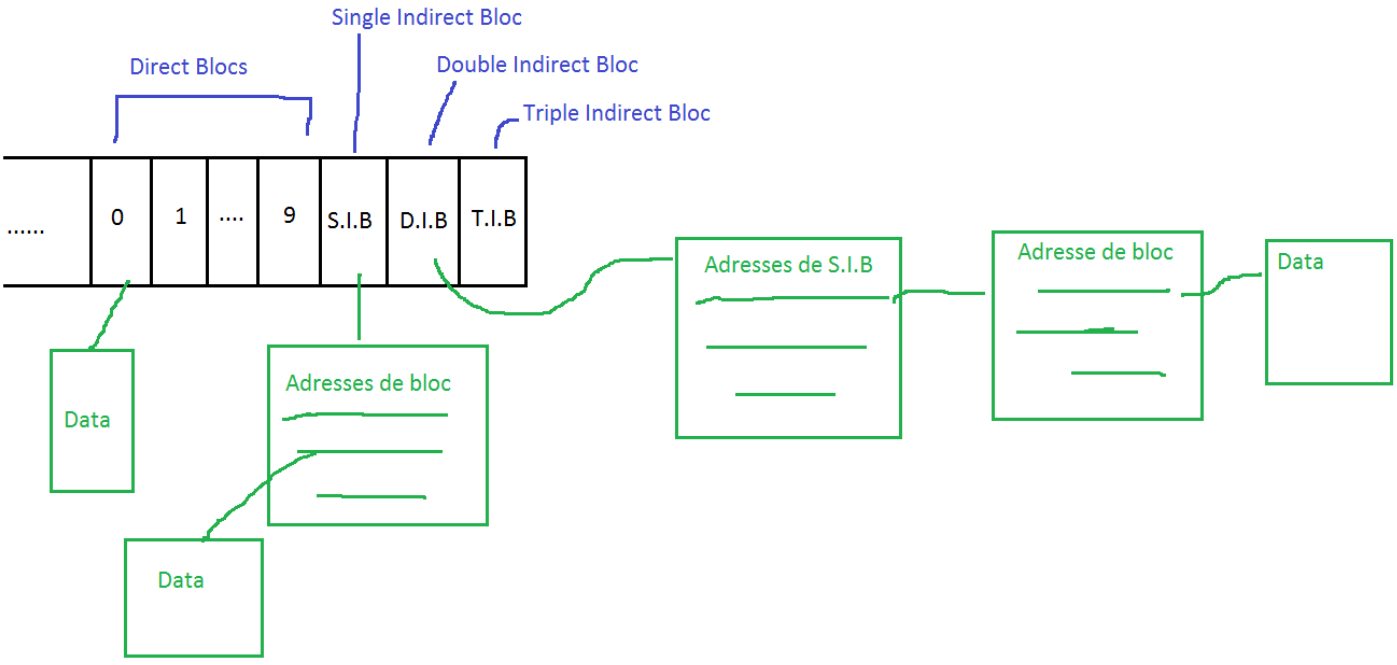
Super Block :

- Décrit l'état d'un file system, sa taille, le nombre de fichiers qu'il peut contenir, la position des blocs libres, …
- Contient :
  - Une liste des inodes libres dans le file system.
  - Une liste des blocs libres disponible dans le file system.
- N'est qu'un bloc => Pas plus grand qu'un data bloc.

Data Blocks :

- Un bloc de données ne peut bien sûr être alloué qu'a un et un seul fichier.

Structure des adresses de blocs (disque) :



Morgan Valentin

Vous avez de 0 à 9 (10 adresses) + 3 autres adresses donc **13** adresses au total. **Voir comme un tableau à 13 entrées**.

Les 10 premières => Direct blocs.

- Donc cela veut dire que pour les 10 premiers blocs de votre fichier, donc le début de votre fichier, les adresses des blocs disques sont stockées directement dans le inode.
- Donc on sait les retrouver directement. On va dans le inode, on trouve l'adresse des blocs disque. Et après on l'amène dans un buffer et on va lire les datas quand on en a besoin.

Un inode a une taille (pour l'instant on nous a dit **64 bytes**). Et on pourrait se dire que vu qu'il y a une taille limite, c'est possible qu'il n'y ait plus de place pour écrire les adresse des blocs disques => Besoin de plus gros fichiers.

Alors le système (pour ne pas être trop restrictif) propose le Single Indirect Bloc (S.I.B) et 2 autres (D.I.B et T.I.B).

Single Indirect Bloc (S.I.B) :

- C'est l'adresse d'un bloc mais le contenu n'est pas les datas de votre fichier mais des adresses de blocs.
- Donc c'est un bloc qui contient les adresses de blocs.

Double Indirect Bloc (D.I.B) :

- C'est l'adresse d'un bloc dont le contenu est une liste / suite d'adresses de S.I.B.
  - Remarque sur le nombre de lecture :
    - Je vais lire le block D.I.B qui va contenir les adresses de S.I.B. Je vais trouver le S.I.B qui m'intéresse, relire le S.I.B pour enfin lire le data bloc (bloc de donnée) qui m'intéresse.
    - Donc 3 lectures pour arriver au bloc !

Triple Indirect Bloc (T.I.B) :

- C'est l'adresse d'un bloc dont le contenu est une série / suite d'adresses de D.I.B.

Morgan Valentin

7. Expliquer les **directories** (algo ***namei***)

Un directory est un fichier spéciale car il contient une liste de fichier et/ou de sous-répertoire.

Il y a plusieurs données mais 2 où nous allons insister :

1) Le nom du fichier / répertoire.
2) Le numéro du inode.

Pathname absolu vs relatif :

Le **chemin absolu** d'un fichier, c'est le chemin depuis la racine (« / »).

Le **chemin relatif** d'un fichier, c'est chemin depuis le répertoire courant (où l'on est).

Le inode de la directory courante et le inode du slash (« / ») sont toujours disponible pour n'importe quel processus à n'importe quel moment.

⇨ Donc TOUT processus à TOUT moment connait TOUJOURS son répertoire courant et « / » (**u-area**).

Syllabus : « La u-area contient un pointeur sur le inode de la directory courante ainsi que sur le inode racine pour le processus ».

Algorithme namei (trouve inode à partir du nom du pathname)

Input => pathname

Output => inode locké

1) **If (pathname commence à la racine)**
   a. **Inotrav = inode racine ;**
2) **Else**
   a. **Inotrav = inode directory courante ;**
3) **While (il y a encore des composants dans le pathname)**
   a. **Lire le composant suivant ;**
   b. **Vérifier que inotrav est une directory et accès ok ;**
   c. **If (inotrav == root et composant == '..')**
      i. **Continue ;**
   d. **Lire directory (inotrav) ;**
   e. **If (composant dans directory)**
      i. **Prendre son numéro d'inode ;**
      ii. **Libérer inotrav ;**
      iii. **Inotrav = inode composant ;**
   f. **Else**
      i. **Return (pas d'inode) ;**
4) **Return (inotrav) ;**

Donc parcourir un répertoire n'est pas difficile. C'est simplement commencer au début, l'ouvrir, commencer au début, avancer, regarder / chercher le nom du fichier et du sous-répertoire qu'on chercher.

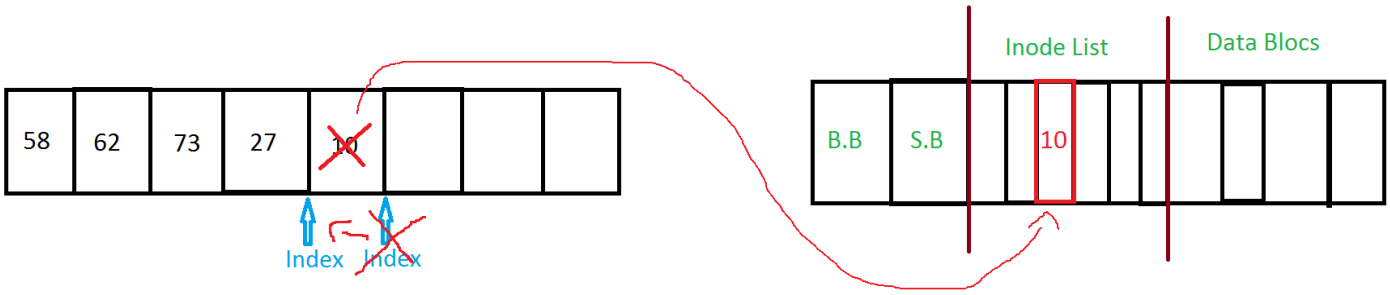
Soit on le **trouve** => On a son numéro d'inode et on fait ce qu'on veut avec.

Soit on ne le **trouve pas** => Message d'erreur.

Morgan Valentin

8. Allocation des inodes sur **disque**.

On va dans le super bloc dans une table et on va trouver un numéro d'inode. Et on va aller sur disque au numéro 10 (exemple) et on va prendre cette inode là et on est parti vers le Incore Inode Table, etc…



Le numéro n'est jamais stocké. Il représente une adresse.

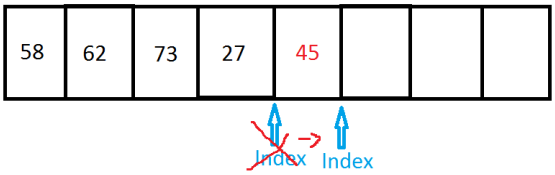
Voir algo (ialloc)

Que se passe-t-il si je supprime un fichier ?

Supprimer un fichier signifie **libérer un inode**.

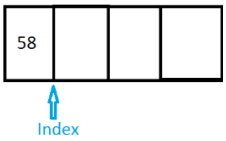
Admettons qu'on libère l'inode n°45 :

- Ajoute l'inode n°45 dans la table.
- On décale l'index.



Retrouver des fichiers perdus en linux, c'est très difficile car le dernier inode désallouer est le premier réallouer !

Passons au cas où il ne reste plus qu'un seul inode dans la inode liste (table).



⇨ Donc la liste des inodes libres du super bloc est vide.

- ➢ Si j'ai besoin d'un fichier ok mais si j'alloue le n°58, après ma liste est vide !
- ➢ Il faut donc penser à remplir cette liste…
- ➢ Quand on alloue le dernier inode de la liste, il a un nom : « Le remembered inode ».

Que fait-on ?

Il faut remplir cette liste.

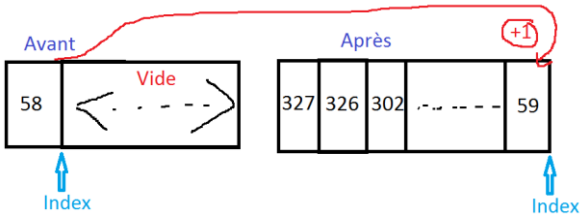
Il faut prendre comme indice de départ sur disque le remembered inode (inode n°58 dans notre cas). Le 58 est là, et bien on va parcourir sur disque toute la liste des inodes (physique, sur disque) jusqu'au bout. Et chaque fois qu'on trouve un inode libre, on va mettre son numéro dans la liste en interne sur le super bloc.

Morgan Valentin

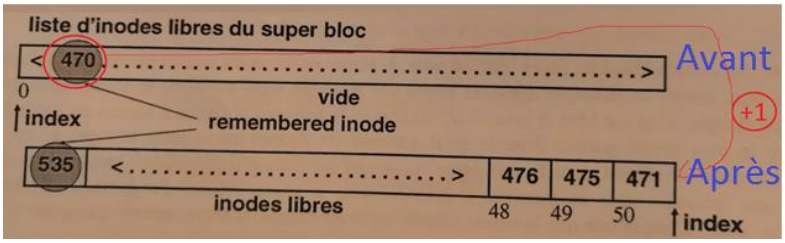
Comment sait-on si un inode est libre ?

⇨ Un inode est libre quand son champ type vaut 0 (rappel il peut valoir D (répertoire) ,- (fichier) ,0 (inode libre).

On va parcourir le super bloc jusqu'à ce que ma liste soit remplie.



Deuxième exemple où mon remembered inode est 470 et je parcours la liste de tout les inodes sur disque.



Syllabus : « Le noyau conserve le plus grand numéro d'inode qu'il a trouvé (le dernier sauvé dans le super bloc), lequel servira de point de départ pour la prochaine recherche sur disque (remembered inode) ».

Morgan Valentin

9. Désallocations des inodes sur **disque** :

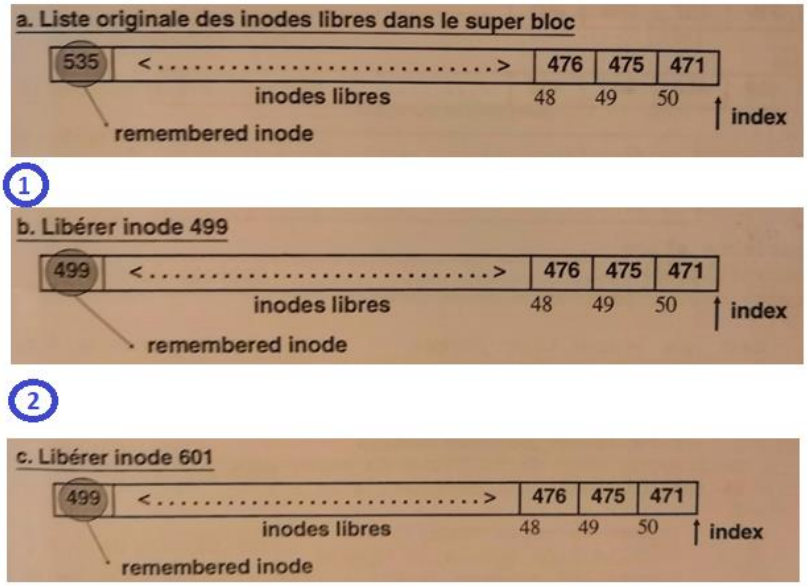
Algorithme ifree (libérer inode)

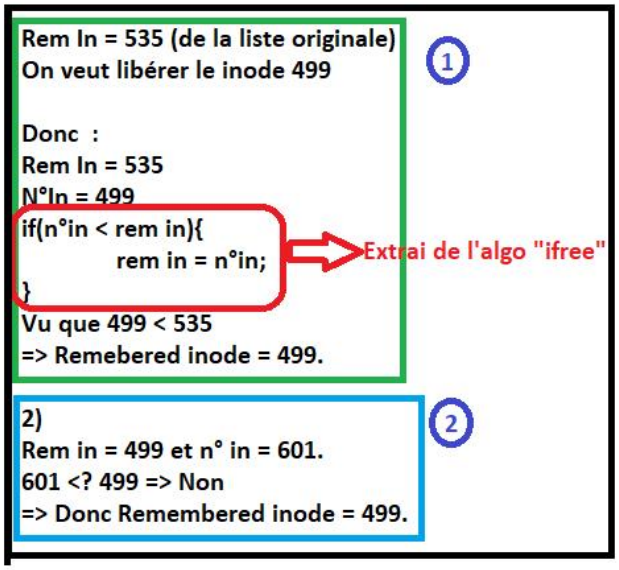
Input => numéro du inode dans le file system.

Output => Rien vu qu'on libère un inode.

1) **Incrémente le compteur d'inodes libres du file system ;**
2) **If (super bloc locké)**
    a. **Return ;**
3) **If (liste d'inodes du super bloc pleins)**
    a. **If (numéro d'inode < remembered inode)**
        i. **Rememberedinode = numéro inode;**
4) **Else**
    a. **Metre le numéro du inode dans la liste ;**
5) **Return ;**

Exemple de désallocation d'inode :





Morgan Valentin

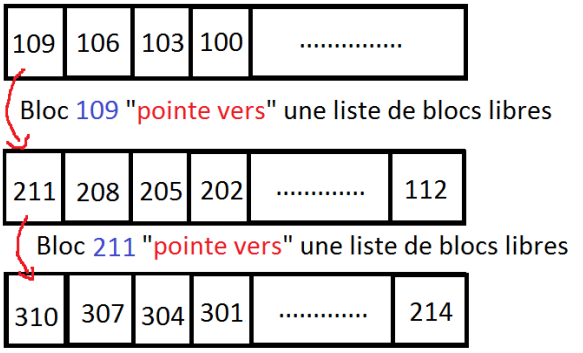
10. Allocation et désallocations des **blocs** (sur disque) :

Rappel :

Le super bloc contient :

1) Une liste des inodes libres (**algo ialloc / ifree**) dans la file system.
2) Une liste des blocs libres disponible (**algo alloc**) dans le file system.

Représentation de la liste des blocs libre :



Voir algo (alloc)

Libérer et affecter un bloc est plutôt simple. Il n'y qu'un seul cas où une réflexion est nécessaire : **lorsqu'il ne reste plus qu'un bloc**. Donc j'ai besoin d'un bloc pour mon fichier mais c'est **le dernier** du super bloc.

Remarque : C'est pas du tout la même chose que pour les inodes !

Syllabus : « Si le bloc alloué est le dernier du cache (bloc liste) du super bloc, le noyau le traite comme un pointeur sur un bloc contenant une liste de blocs libre. Il place leurs numéros dans la liste du super bloc et utilise ensuite le bloc qui contenait ces numéros pour y placer des données ».

Liste d'inodes vs liste de blocs (toujours dans super bloc) :

La liste des inodes est assez petite contrairement à la liste des blocs qui elle contient TOUS les blocs du disque. Si on devait parcourir tous les blocs du disque 1 par 1 pour voir s'ils sont libres ou occupés … Beaucoup trop long !

Traitement du cas du dernier bloc dans la liste des blocs du super bloc :

a) Configuration de base

b) On libère le bloc 949

On ajoute le bloc 949 dans la liste de blocs libres

c) On affecte le bloc 949

On retire le bloc 949 de la liste.
=> De retour à la configuration initiale

d) On affect le bloc 109 (dernier bloc)

On place les numéros des blocs pointer par le bloc 109 dans la liste du super block **et ensuite** on retire le block 109 de la liste.

[Morgan Valentin]

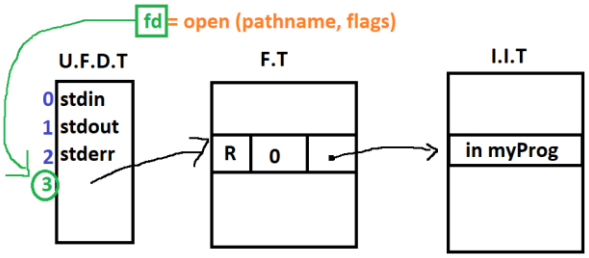
11. Expliquer comment fonctionne l'algorithme **Open**.

1ère synthaxe

fd = open (pathname, flags, mode) Ou 2ème synthaxe fd = open (pathname, flags)

pathname => Nom du fichier que vous voulez ouvrir
flags => Mode d'ouverture (R,W)
mode => Donne les permissions (si vient d'être crée par open

flags => 0 = R(read)
1 = W (write)
2 = R-W (read - Write)



Note : **fd** est un entier (U.F.D.T).

Algorithme open :

Input => nom du fichier + type d'ouverture + permissions

Output => file descriptor (fd)

1) **Convertir le nom du fichier en un inode ; (algo namei)**
2) **If (fichier n'existe pas ou accès interdit)**
    a. **Return (error) ;**
3) **Allouer une entrée de la File Table et l'initialiser ;**
4) **Allouer une entrée de la U.F.D.T et la faire pointer sur l'entrée de la File Table ;**
5) **If (type d'ouverture demande de tronquer le fichier)**
    a. **Libérer les blocs diques ;**
6) **Délocker le inode ;**
7) **Return (user file descriptor => l'entier fd)**

Que se passe-t-il lors de **l'ouverture d'un fichier** ?

=> Supposons qu'il existe !

On trouve son inode sur disque, on l'amène dans la Incore Inode Table. Donc on prend le inode disque, on le copie dans la Incore Inode Table, on établis un lien via la File Table, qui elle va pointer vers le inode.

Rappel, il y a 3 champs dans la File Table :

1) Le premier est le mode d'ouverture.
2) Le deuxième est l'offset (Où est-ce qu'on est dans le fichier => On appelle ça le « **déplacement** »).
3) Et le troisième c'est lien avec votre processus => User File Descriptor Table.

Cette UFDT a 3 champs déjà réservé 0,1 et 2 qui correspondent à stdin, stdout et stderr.

Et puis après cela, un pointeur pour chacun des fichiers ouverts vers une entrée de la File Table. Comme ça vous avez accès via l'offset au contenu de l'inode.

Morgan Valentin

12. Expliquer différence entre **FIFO** et **PIPE**.

Unnamed pipe (PIPE)	Named pipe (FIFO)
Accès grâce **pipe**	Accès grâce à **open**
Accesible par le processus descendant.	Accessible par tous les processus.
Accès en lecture et écriture via un tableau de 2 file descriptor.	Accesible via un **pathname**.
Noyau reprend le inode quand le processus a terminé.	Permanent.
	Problème : Risque d'erreur de nom lié au pathname (voir algo namei)
	Problème : temps de traitement supplémentaire à la création.
Données **temporaire**.	
Données lue ne **peuvent pas être relue**.	

13. Expliquer où, quand et comment détecter une **erreur du mode d'ouverture**.

Où :

- ⇨ Dans un champ de la U-Area réserver pour les valeurs de retour de l'appel système.
- ⇨ Du coup, il y a dans ce champ les erreurs possibles.

Quand :

- ⇨ Lorsque l'on revient en mode user (context switch), vu que c'est seulement à ce moment là où l'on voit les valeurs de retour de l'appel système.

Comment / Pourquoi :

- ⇨ Soit on avait pas les permission pour accéder au fichier.
- ⇨ Soit les permissions (mode) ont étés mal paramétrées lors de l'appel système **open**.

Rappel sur open :

Fd = open (pathname,flags,mode)

- Pathname => le chemin d'accès
- Flags => Le mode d'ouverture (on veut lire le fichier ou écrire)
- Mode => Donne les permissions **SI VIENT d'être CRÉER PAR OPEN**

Morgan Valentin

14. Expliquer les 2 cas de l'erreur « **no more inode** ».

Lorsque nous avons une erreur du type "**no more inode**", c'est que l'on cherche à allouer un inode et qu'il y en a plus.

Et on alloue des inodes soit en **mémoire**, soit sur **disque**.

Sur disque : (super bloc)

- ⇨ Lorsque la liste d'inode libre du super bloc est vide et que le processus veut trouver des inodes libres sur le disque mais qu'il n'y en a plus.

En mémoire : (inode incore table)

- ⇨ Quand il n'y a plus de place dans la Incore inode table (trop de fichiers ouverts en même temps).

15. Expliquer à l'aide des buffers, **2 processus qui font une lecture sur un fichier**.

Pour le premier processus le noyau va trouver le bloc dans la hashqueue et le buffer sera libre, le processus pourra donc y accéder mais le buffer sera alors **marqué busy.**

Pour le deuxième processus, vu que le buffer correspondant est marqué busy il dormira jusqu'à ce que le **buffer soit libéré**.

16. Expliquer l'appel système **write**.

On utilise le file descriptor (FD) renvoyé par le noyau lors d'un open ou un create.

Le noyau trouve ensuite l'entrée de la UFDT correspondant et suit le pointeur vers l'entrée de la FT.

Il suit ensuite le pointeur vers l'entrée de la IIT et bloque le inode avant d'écrire dans le fichier.

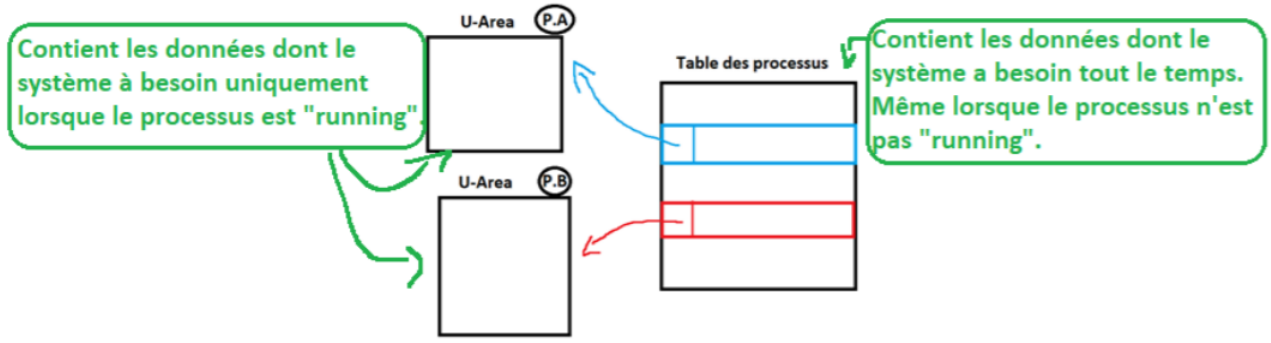
Le inode est donc ensuite bloqué pendant toute la durée du write et le noyau peut le modifier en allouant de nouveaux blocs.

Lorsque le write est terminé, le noyau met à jour la taille du fichier dans le inode, si modifié.

Morgan Valentin

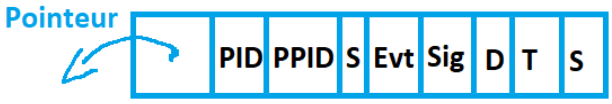
Chapitre 2:

1. Expliquer la différence entre la **table des processus** et la **u-area** :



Syllabus : « La table des processus contient des champs qui doivent toujours être accessibles au noyau, tandis que les champs de la u-area ne doivent être accédés que par le processus en cours d'exécution. »

Que y a-t-il dans la table des processus ?



- Un **pointeur** sur la u-area.
- Un champ **status** (running(R), writing(W), sleeping(S), ready to run, …)
  - Running => En cours d'exécution avec le processeur. « Pris en charge par le processeur ».
  - Sleeping => Attends quelque chose qui vient du système.
  - Waiting => Attends quelque chose qui vient d'un autre processus.
- **PID** (Process ID) + **PPID** (Parent Process ID).
  - Chaque processus a un numéro unique qui est PID (pas d'ordre spécifique)
  - Un processus est toujours le fils de son père. Et c'est toujours un processus père qui crée un ou plusieurs processus fils.
  - Un processus en crée un autre grâce à l'appel système ***fork().***
- Réception des **évènements** => Exemple l'évènement « sleep(buffer libre) ».
- Réception des **signaux**.
  - Un signal est un évènement qui vient d'un autre processus.

Morgan Valentin

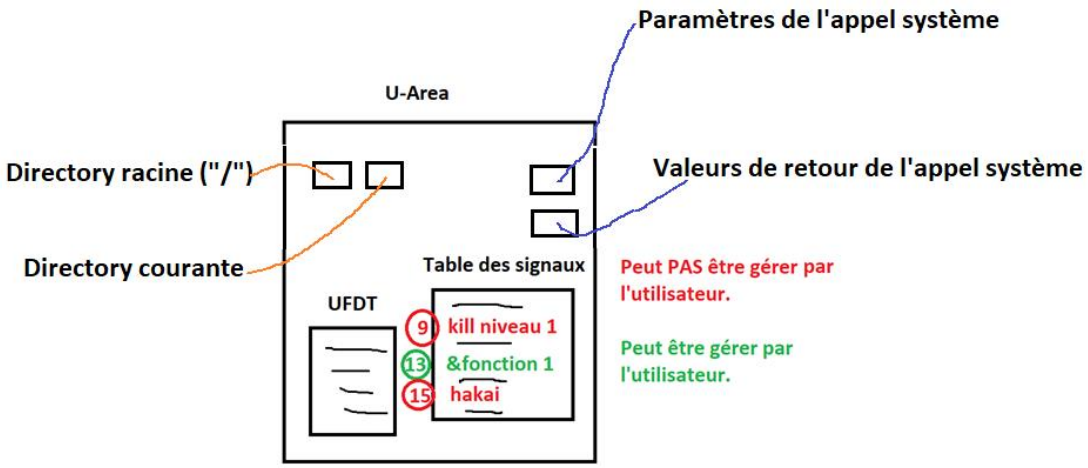
Prégion (Process region) :

Syllabus : « Un processus comprends **trois régions** : **texte**, **données** et **pile**. Plusieurs processus peuvent partager une région. (…) Chaque processus contient une table des régions par process privée (Prégion=Process region) ».
Processus divisé en 3 morceau :

- ❖ Une partie qui permet de retrouver vos données **DATA** (D)
- ❖ Une partie qui permet de retrouver les instructions du programme **TEXT** (T)
- ❖ Une partie qui permet de retrouver la User **STACK** (S) => Votre environnement de travail.

- Une chose à retenir sur la table des région => A chaque entrée, il y a un pointeur sur la table des pages de la région => Pagination!

U-Area :



Contient :

- La **UFDT** (User File Descriptor Table)
- Une **table des signaux**
- Un champ pour les **paramètres** de l'appel système et pour les **valeurs de retour** de l'appel système.
- Également la directory **courante** et la **racine** courante du processus.

Morgan Valentin

2. Expliquer les parties **contexte** d'un processus.

Il y a **3 types** de contextes :

- Contexte **User** => C'est-à-dire votre programme (avec les 3 régions -> D,T,S)
  - Processus divisé en 3 morceau :
    - Une partie qui permet de retrouver vos données **DATA** (D)
    - Une partie qui permet de retrouver les instructions du programme **TEXT** (T)
    - Une partie qui permet de retrouver la User **STACK** (S) => Votre environnement de travail.
- Contexte **Registre** (On s'en fou).
- Contexte **Système** :
  - Table des processus.
  - U-Area
  - Les données systèmes, dont celui-ci a besoin pour gérer votre processus.

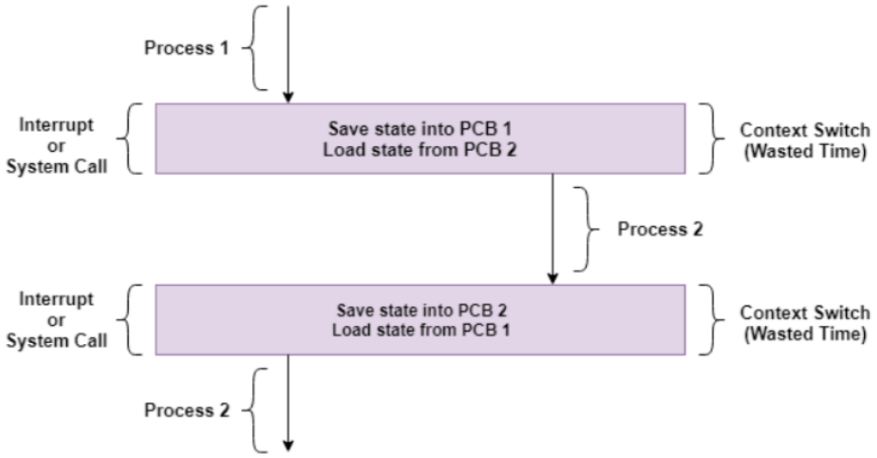
Note :

- Table des processus et la U-Area => Partie **statique** (toujours **présente**)
- La pile système => Partie **dynamique** (**N'existe que** lors d'un appel système)

Morgan Valentin

3. Expliquer le **contexte switch**.

A diagram that demonstrates context switching is as follows:



⇨ Un contexte switch se produit dans **plusieurs cas** :

- Lors d'un sleep()
- Lors d'un exit()
- Lors d'un appel système (Voir NOTE).
- Lors d'une interruption (Voir NOTE).

⇨ Procédure pour **effectuer** un contexte switch :

- **Décider** de faire un contexte switch
- **Sauver le contexte** de l'ancien processus
- Passer au **processus suivant** (sélection => On choisit en autre mot)
- **Restaurer le contexte** de ce nouveau processus.

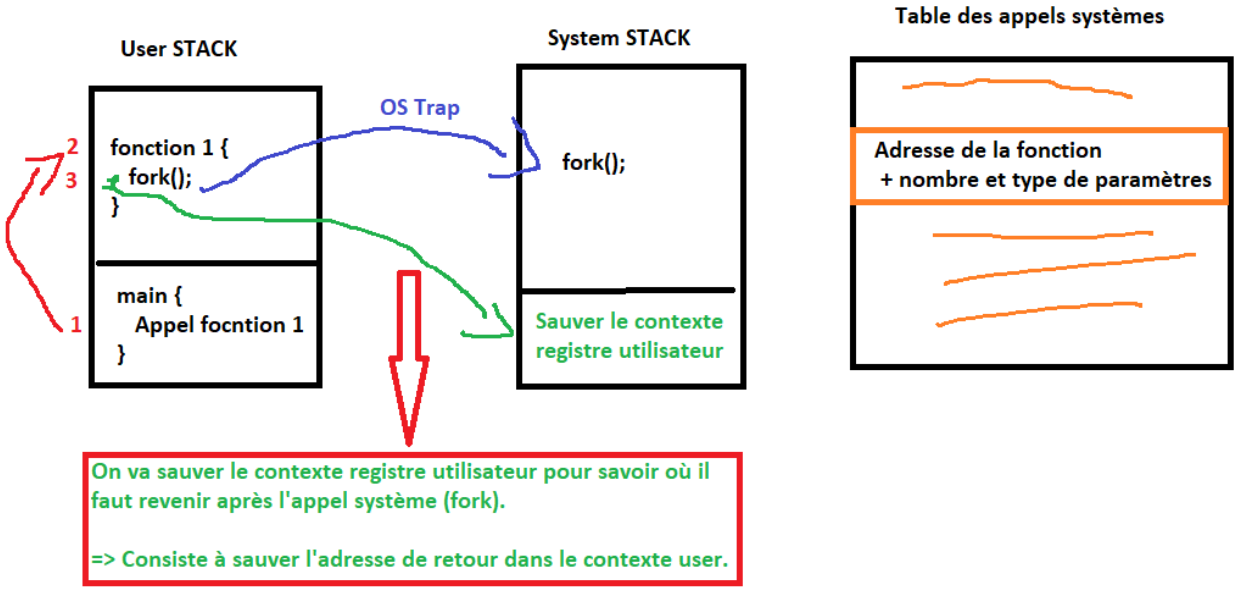
⇨ Note :

- Les contextes switch ont **toujours** lieu en **mode utilisateur** ou lors **d'un retour en mode utilisateur**, c'est-à-dire alors que la pile noyau (système) est **vide**.

Morgan Valentin

4. Appels systèmes

Un appel système = **appelez une fonction système** (et pas une fonction que vous avez écrites).



La pile système est la **partie dynamique** => Elle **n'existe pas tout le temps**. Dès qu'il y a un appel système, on change de pile.

Syllabus : « Les fonctions systèmes appellent une instruction (Operating System Trap = OS-Trap), qui change le mode d'exécution du processus en mode noyau (système) et lance l'exécution par le noyau du code. »

Où trouver les appels systèmes ?

⇨ Les appels systèmes sont répertorier dans une table interne au système => Table des appels système.

Table des appels systèmes :

- ✓ Une entrée par appel système.
- ✓ On trouve pour chaque appel système :
  - **L'adresse** de la fonction à exécuter.
  - Le **nombre** et le **type** des paramètres.
- ✓ Chaque appel système est caractérisé par un numéro. Et en fonction du numéro de l'appel système que vous allez exécuter, vous allez avoir accès à une entrée dans cette table.

Passer des paramètres à l'appel système :

⇨ On va initialiser les paramètres formel à l'aide de paramètres réels.

Paramètre formel : Ceux qui sont écris à l'entrée de la fonction et initialiser au moment où on appelle la fonction.

Paramètre réel : Paramètre passés lors de l'appel de la fonction.

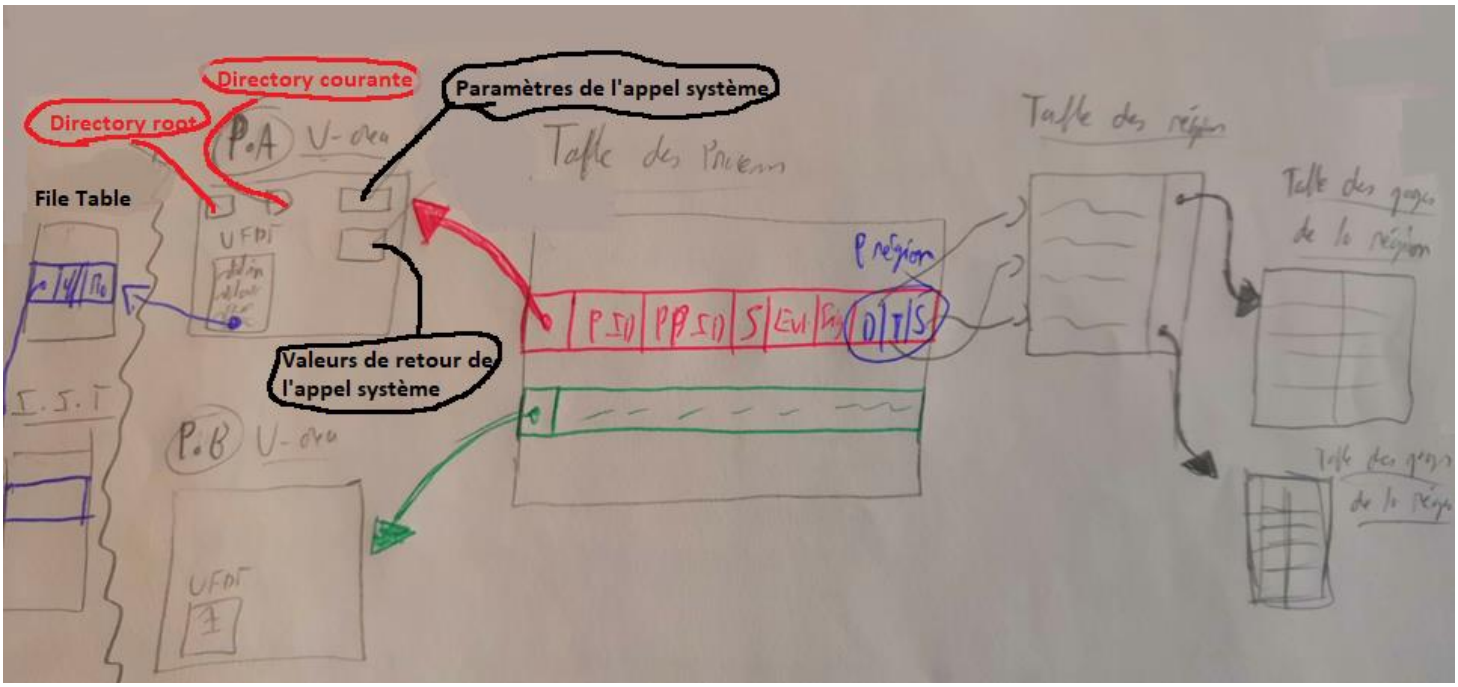
Morgan Valentin

Mécanisme de traitement d'un appel système :

1) J'appelle ma fonction
2) On passe les paramètres => Paramètres copier dans une zone spécifique dans la U-Area.
3) On fait un OS Trap.
4) Sauver le contexte utilisateur.
5) Grâce au numéro de l'appel système, on trouve dans la table des appels systèmes l'adresse de la fonction, le nombre et le type de paramètre.
6) Copier les paramètres réels dans la U-Area.
7) Exécuter l'appel système.
8) L'appel système va renvoyer les valeurs de retour dans une zone spécifique de la U-Area.
9) Quand on revient de l'appel système (passe en mode User), le processus User va retrouver dans une zone spécifique de la U-Area les valeurs de retours.

***Note** : Vos données font partie de l'espace user.*

Schéma globale :



Noter qu'il manque la table des signaux dans la U-Area => Elle devrait y être si on veut être complet.

Morgan Valentin

5. Interruptions.

Une interruption vient du système => On n'est pas maitre d'une interruption.

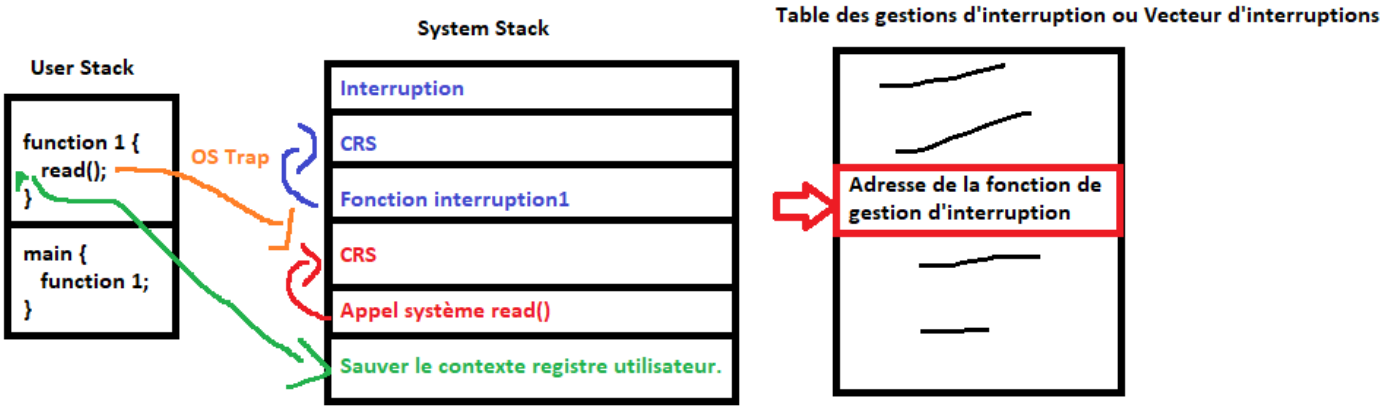
Lorsqu'une interruption se produit, il faut la gérer.

On sait qu'il y a quelque chose qui vient de se produire en interne, il faut mettre à jour les données du systèmes /structures de données du système.

Qu'est-ce qui se passe ?

- ⇨ **Comme pour les appels systèmes !**

Supposons que vous êtes en mode noyau et une interruption se produit. On va sauver le contexte registre => Pour savoir où revenir. Et on va exécuter la fonction de gestion de l'interruption.



En fonction de l'interruption (possède également un numéro), on va aller dans la table des gestions d'interruption / Vecteur d'interruptions et on va trouver l'adresse de la fonction pour gérer l'interruption. On l'exécute et quand on a fini, on revient et on reprend le programme.

Morgan Valentin

6. Expliquer le **fork** (de manière **théorique**).

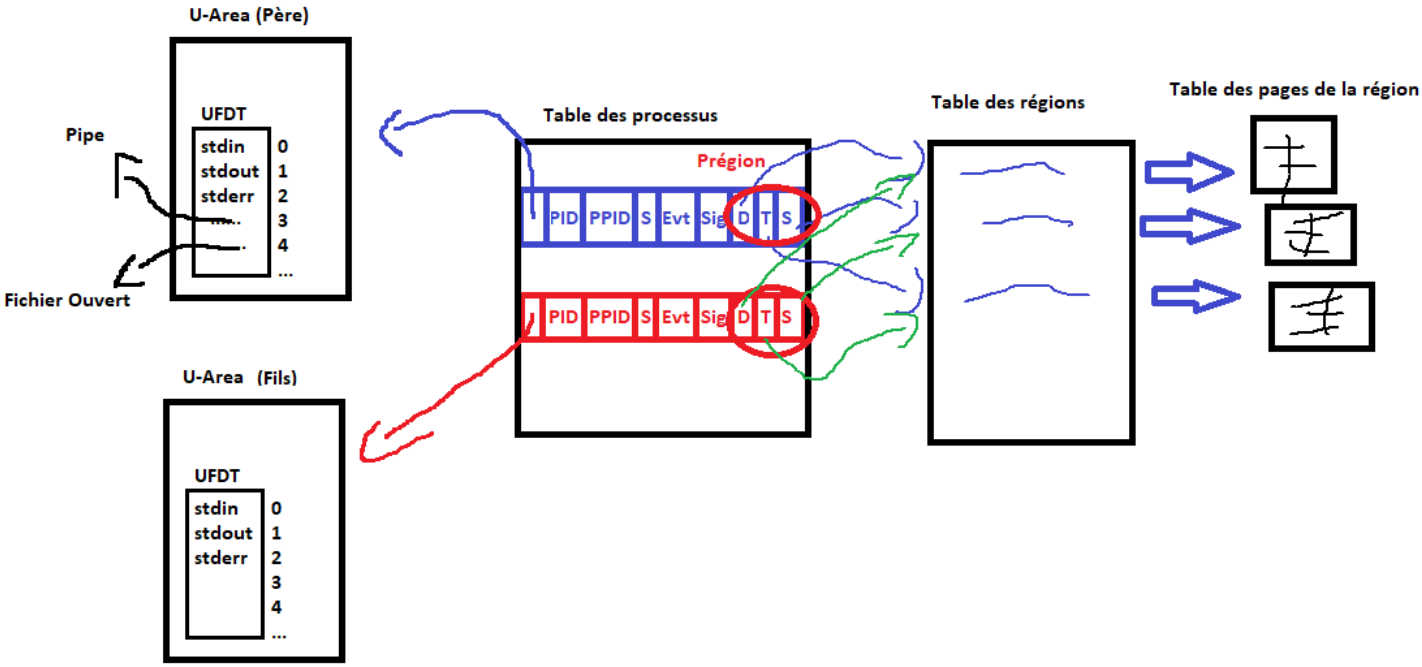
Input => Rien !

Output => PID du fils pour le père / 0 pour le fils.

Donc si fork :

- <-1 (négatif) => Erreur !
- =0 => Fils
- >0 (positif) => Père.

Explications à l'aide des différentes structures :



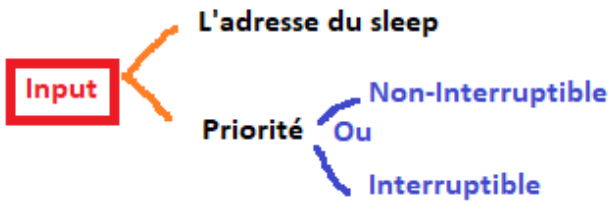
Si le fils veut modifier la région DATA, il fera une copie, puis il modifiera ce qu'il voudra.

Le seul moyen dont un utilisateur dispose pour créer un nouveau processus est l'appel système **fork**. Le processus qui lance l'exécution du **fork** est appelé le **parent** et le processus nouvellement crée durant le fork est appelé **l'enfant**. Au retour du **fork**, les deux processus ont des **copies identiques de leur contexte user**, à l'**exception de la valeur de retour PID**.

+ **Vous DEVEZ être capable d'expliquer le schéma ci-dessus** !

Morgan Valentin

7. Expliquer l'algorithme **sleep**.



L'adresse du sleep ?

Il y a dans la table des processus un champ particulier, qui permet de recevoir l'adresse du sleep. Donc lorsqu'un processus part en sleep, on va écrire dans la table des processus ce qu'on appelle « l'adresse du sleep ».

- ⇨ Attention, c'est le terme exacte mais ce n'est pas un pointeur !
- ⇨ C'est un entier qui est le numéro de l'évènement attendu par le processus.

Priorité ?

Il y a 2 types de priorité :

- Non-Interruptible => Le processus sera réveillé uniquement par l'occurrence de l'évènement attendu.
- Interruptible => Le processus sera réveillé par l'occurrence de l'évènement ou par un signal.

Quand un autre processus qui tourne à côté du notre processus sait que ce qu'on attend ne se produira pas, alors il enverra un signal.

L'algorithme sleep :

- Je pars en sleep car j'attends un évènement.
- L'évènement va être noter dans un champ dans la table des processus (« Evt »).
    - Donc le numéro de l'évènement attendu va être inscrit dans la table des processus.
- Et le processus part en sleep => Context switch (passe à un autre processus).
- En fonction de l'évènement attendu, le sleep peut avoir soit :
    - Priorité non-interruptible :
        - Alors on va attendre que l'évènement se produit (et il se réveil après).
        - Return (0) ;
    - Priorité interruptible :
        - 2 cas possibles :
            - ❖ Soit il a été réveillé par un signal.
                - Donc pas réveillé par un évènement.
                - Return (1) ;
            - ❖ Soit il a été réveillé par un évènement.
                - Return (0) ;
- Mais comme un signal pourrait être arrivé avant que le processus s'endorme, on va tester s'il n'a pas reçu de signal disant qu'il est inutile d'aller en sleep.

Morgan Valentin

8. Gestion et traitement des **signaux**.

Un signal est un évènement en mode User.

Envoyer un signal à un processus :

- ⇨ Peut être fait par l'appel système kill (PID, numéro du signal) ;

Chaque processus à une entrée dans la table des processus et une U-Area spécifique. Et dans la U-Area, il y a une table des signaux.

La fonction système (ou appel système) signal permet de dire ce qu'on veut faire lorsque l'on reçoit un signal.

Attention, l'appel système signal n'envoie pas de signal ! Il permet de les gérer.

signal (numéro du signal, x)

si x = 0 => Exit si je reçoi un signal.

si x = 1 => Ignorer si je reçoi un signal.

si x = &fonction user => gérer le signal.

PS: & = "adresse"

Gérer le signal :

Il faut savoir que certaine signaux ne peuvent pas être gérer comme les signaux 9 et 15 qui « tue » un processus.

A la réception d'un signal, si un processus n'a rien fait (pas préciser si ignorer / exit ou exécuter une fonction), il va **par défaut** **exit** le programme.

Note : Le gestion des signaux = communications entre processus en mode User.

On peut mettre dans la table des signaux l'adresse de la fonction User que vous utilisez.

- ⇨ « Catcher un signal » => Le prendre, le recevoir et quand je le reçoi, j'exécute la fonction dont l'adresse est dans la table des signaux.

Je suis le processus A (running) et je veux envoyer un signal au processus B :

Je vais écrire le numéro des signaux dans la table des processus (champ « sig »). Et lorsque le processus B reprendra la main, il va regarder s'il a reçu un signal. Et si c'est le cas, il va exit / ignorer ou exécuter une fonction (User) dont l'adresse se trouve dans la table des signaux.