OS : théorie

Inode

« metadata » d'un fichier

Une inode est une structure de données capable de donner la représentation interne d'un fichier. Chaque fichier a une inode **mais l'inode ne spécifie pas le pathname du fichier**.

Il contient une description de l'aspect disque du fichier de données et d'autres informations d'ordre « administratif ». Les inodes disques contiennent les champs suivants :

- Type du fichier (régulier, directory, character ou block device, FIFO (pipes))
- Permissions d'accès sur le fichier (lecture, écriture et exécution pour le propriétaire du fichier, les membres du groupe du propriétaire ou les autres utilisateurs)
- Identification du propriétaire du fichier (individuel et groupal)
- Nombre de links sur le fichier
  exemple : ln, ln -s (= symlink (lien symbolique = entrée spéciale de répertoire qui permet de référencer de manière quasi transparente d'autres entrées du répertoire, typiquement, des fichiers ordinaires ou des répertoires. Il se comporte comme un *alias* d'un fichier ordinaire ou d'un répertoire))
- Taille du fichier
- Temps de dernier accès au fichier
- Temps de dernière modification sur le fichier
- Une table pour les adresses disques des données (les blocs disques qui contiennent les données)

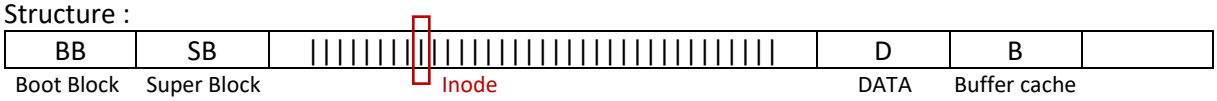
Les inodes existent sous forme statique sur disque et le noyau les lit dans un *in-core inode* pour les manipuler. Ces copies des inodes en mémoire se trouvent dans une table appelée **in-core inode table**, **structure unique pour le système**.

Le noyau contient également deux autres structures de données, la **file table** (structure globale du noyau) et la **user file descriptor table** (**une par processus**).

UFDT = User File Descriptor Table
↳ Identifie les fichiers ouvert par un processus.

File System (F.S.)

Un File System peut être vu comme une portion logique d'un disque physique.
Structure :



- *BB :* Boot Block (peut contenir le bootstrap code qui initialise l'OS)
- *SB :* Super Block (contient toutes les descriptions de l'état du File System*)
  * état, taille, nombre de fichiers qu'il peut contenir, position des blocs libres, etc.
- Liste d'inodes (taille définie en fonction de la taille moyenne souhaitée pour un fichier)
  L'un des inodes est le root inode du file system ; c'est par lui que la structure de directory du file system est accessible après exécution de l'appel système mount.
- Bloc de données (un bloc de données ne peut être alloué qu'à un seul fichier)

Buffer cache : contient les données des blocs disque récemment utilisé.
Il est utile pour minimiser la fréquence d'accès disque.

Un **delayed-write** est différent d'une **écriture asynchrone**.
Delayed-write : le noyau attend le plus longtemps possible avant de lancer l'écriture.
Ecriture asynchrone : le noyau commence l'opération immédiatement mais n'en attend pas la fin.

In-core inode table

La copie in-core du inode contient le inode disque plus :

- Le statut du in-core inode, lequel indique si :
  - Le inode est bloqué (locké)
  - Un processus attend que le inode soit débloqué
  - Le in-core inode diffère du inode disque (différence provoquée par une modification du contenu de ce inode)
  - Le contenu du fichier en mémoire est différent du fichier disque (ceci provenant d'un changement dans les données de ce fichier)
  - Le fichier est un mount point
- Le numéro de device logique du file system contenant le fichier
- Le numéro de inode du fichier (la position du inode du fichier dans le tableau des inodes)
- Un compteur de références, indiquant le nombre d'instances (copies) du fichier qui sont actives
- Des pointeurs sur d'autres in-core inodes

Le noyau lie les inodes dans des hash queues et une liste des inodes libres de la même manière qu'il le fait pour les buffers du buffer cache. Une hash queue est identifiée par le numéro de device logique et le numéro de inode. Il ne peut exister qu'une seule copie in-core d'un inode disque, mais les inodes peuvent se trouver simultanément dans une hash queue et dans la liste des inodes libres.

Un inode est actif lorsqu'un processus l'alloue (par exemple, en ouvrant le fichier). Un inode est dans la liste des inodes libres quand son compteur de références vaut 0, c'est-à-dire qu'il peut être réalloué pour un autre fichier. Dès lors, on peut se représenter la liste des inodes libres comme un cache d'inodes inactifs, pouvant être réalloués, mais se trouvant malgré tout en mémoire.

Changer le contenu d'un fichier entraîne automatiquement un changement dans le inode, mais pas l'inverse.

Super bloc

Le **super bloc** est un bloc qui contient toute la description de l'état du file system.

Il contient les champs suivants :

- La taille du file system
- Le nombre de blocs libres dans le file system
- Une liste des blocs libres disponibles dans le file system
- L'index du prochain bloc libre dans la liste des blocs libres
- La taille de la liste des inodes
- Le nombre d'inodes libres dans le file system
- Une liste des inodes libres dans le file system
- L'index du prochain inode libre dans la liste des inodes libres
- Des lock fields pour les listes des blocs libres et des inodes libres (exemple : breada (p. 1.9))
- Un flag indiquant si le super bloc a été ou non modifié

Appels systèmes (A.S.)

La première chose que doit faire un processus pour accéder aux données d'un fichier est *« open »*.
Sa syntaxe est $fd = open(pathname, flags, mode);$

fd : file descriptor
pathname : nom du fichier
flags : type d'ouverture (Read/Write)
mode : donne les permissions

Un appel système est une fonction primitive fournie par le noyau d'un OS à travers duquel un processus actif peut obtenir des services du système.

Pipes (fichiers spéciaux)

L'implémentation traditionnelle des pipes utilise le File System pour le stockage de données.
Il y a 2 sortes de pipes :

- Named pipes (FIFO)*
- Unnamed pipes (pipe ordinaire)

* exemple : la caisse au Colruyt (donnée lue → donnée partie du pipe)

Un pipe ordinaire est un pointeur sur un tableau d'entiers qui contiendra les 2 File Descriptor permettant l'accès au pipe de lecture et d'écriture. Un FIFO est un fichier semblable à un pipe ordinaire mais qui dispose d'une entrée de directory et est accessible via un pathname.

Structure de données du noyau

2 structures : table des processus, U_{area}.

Table des processus

Contient des champs qui doivent toujours être accessible au noyau.
Elle contient :

- L'état des processus et sa taille (nombre de processus exécuté simultanément),
- Champ permettant au noyau de localiser le processus et sa U_{area} en mémoire centrale ou secondaire (utilisé par le noyau lors d'un context switch ou pour le swapping),
- User ID déterminant les privilèges du processus (nombre entier et unique),
- Process ID spécifiant les relations inter-processus (nombre entier et unique),
- Un descripteur d'évènement pour le sleep,
- Running **(paramètres liés au CPU Scheduling)**,
- Un champ signal,
- Des timers.

U_{area}

Doit être accédé uniquement par le processus en cours d'exécution.
Elle contient :

- User ID réel et effectif **(détermine les privilèges du processus)**,
- Le tableau de gestion des signaux (il y a des fonctions par défaut appelé par les signaux),
- Champ contrôle pour identifier le terminal associé au processus (s'il y en a un),
- Champ erreur (enregistre les erreurs rencontrées pendant l'exécution d'appels systèmes),
- Champ valeur de retour (pour les résultats des appels systèmes),
- Paramètres I/O (décrit la quantité de données à transférer, les adresses des données en espace utilisateur, des déplacements dans les fichiers pour les I/O, …),
- La directory courante et la racine courante du processus,
- La User File Descriptor Table qui gère l'accès aux fichiers ouvert par le processus.
  La UFDT fait référence au système pour accéder au disque dur (données et inodes).
- Champ permission masque les modes par défaut placés sur les fichiers que le processus crée,
- Des timers.

Sleep

= en attente d'un évènement.
L'algorithme sleep fait passer un processus de « exécuté en mode noyau » à « endormi en mémoire ».

2 sortes de sleep :

- non-interruptible = peut être réveillé par un évènement* qu'il attend.
- Interruptible = réveillé par l'évènement*,
  = réveillé par un signal,
  = réveillé par l'évènement* qui ne se produit pas.

* évènement : buffer

Lorsque processus réveillé, noyau effectue un « longjmp » *(si réveillé par un signal qu'il ne gère pas, sinon 1)*.

Wakeup

L'algorithme wakeup fait passer un processus de « endormi » à « prêt à tourner », en mémoire ou swappé.
Le réveil d'un processus à lieu pendant un appel système ou lors d'une interruption.
Le noyau bloque toutes les interruptions et il met les processus endormi sur l'adresse du sleep dans l'état « prêt à tourner ».

Contexte d'un processus

Le contexte d'un processus comprend :

- Le contenu de son espace d'adresses utilisateurs (**contexte utilisateur**),
- Le contenu des registres hardwares (**contexte registre**),
- Des structures de données du noyau se rapportant au processus (**contexte système**).

Contexte utilisateur

Il est constitué de l'espace d'adresses accessibles au processus pendant son exécution en mode utilisateur. Les parties de l'espace d'adresses virtuelles d'un processus qui, périodiquement, ne résident pas en mémoire centrale pour cause de swapping ou de pagination font également partie du contexte utilisateur.

La partie **texte** d'un processus contient les instructions machine qui seront exécutées par le hardware. Cette partie est très fréquemment accessible en lecture uniquement, de manière à pouvoir être partagée. Lors de l'exécution d'un programme par le système d'exploitation, la partie texte est amenée en mémoire depuis le fichier sur disque, sauf s'il s'agit d'un fichier partagé se trouvant déjà en mémoire.

La partie **data** contient les données du programme. Elle peut se diviser en 3 parties :

- **Les données read-only initialisées** (données initialisées par le programme et accessibles en lecture uniquement pendant l'exécution du processus),
- **Les données read-write initialisées** (données initialisées par le programme, mais qui peuvent être modifiées en cours d'exécution),
- **Les données non initialisées** (sont mises à 0 avant le début du processus. Peuvent être modifiées en cours d'exécution. Souvent appelées **bss**. L'avantage est que le système ne réserve pas de place dans le fichier sur disque pour eux. Le gain réside dans le total d'espace disque alloué mais aussi dans le temps requis pour lire le fichier en mémoire avant l'exécution du processus).

Le **heap** (tas) est utilisé pour les allocations dynamiques de mémoire durant l'exécution du process.
Le **stack** (pile) est utilisé dynamiquement en fonction de l'évolution de l'exécution du process. Un cadre de pile contient les éléments requis pour l'exécution d'une fonction.

Beaucoup de systèmes laissent un espace libre entre le heap et le stack, pour que les 2 portions puissent grandir dynamiquement en cours d'exécution du processus. C'est souvent dans cet espace libre que se fait l'allocation des segments de mémoire partagée.

Contexte registre

Il contient :

- Le PC qui spécifie l'adresse de la prochaine instruction à exécuter par le CPU (il s'agit d'une adresse virtuelle, dans le noyau ou en espace mémoire utilisateur),
- Le PS (Process Status Register) qui spécifie le statut hardware du process (on y trouve des champs indiquant l'overflow, le carry, le mode d'exécution du processus (noyau ou utilisateur), …),
- Le pointeur de pile contient l'adresse actuelle de la prochaine entrée dans la pile (noyau ou utilisateur, selon le mode d'exécution),
- Des registres généraux contenant des données générées par le process en cours d'exécution.

Contexte système

Formé d'une partie statique et d'une partie dynamique. Un process dispose d'une seule partie statique pendant toute sa durée de vie, mais il peut avoir un nombre variable de parties dynamiques. La partie dynamique pourrait être vue comme une sorte de « pile de niveaux de contexte », que le noyau empile et dépile en fonction des événements qui se produisent.

Partie statique :

- L'entrée de la table des processus (qui définit l'état du process et contient de l'information de contrôle toujours accessible au noyau),
- La U_{area} du process (qui contient de l'information accessible uniquement dans le contexte du process),
- Les entrées des $P_{région}$, table des régions et table des pages nécessaires pour la traduction des adresses logiques en adresses physiques.

Partie dynamique :

- La pile du noyau qui contient les cadres de pile des procédures du noyau, lors de l'exécution en mode noyau d'un process (bien que tous les process exécutent le même code noyau, ils disposent chacun d'une copie privée de la pile du noyau, spécifiant leur utilisation propre des fonctions communes du noyau),
- Un ensemble de niveaux, visualisées comme une pile (chacun des niveaux contient l'information nécessaire pour revenir au niveau précédent, y compris le contexte registre du niveau précédent).

Création de processus

$process\ id = fork();$

Le seul moyen dont un utilisateur dispose pour créer un nouveau processus est l'appel système fork.

a.) Le noyau alloue une entrée pour le nouveau processus dans la table de processus.
b.) Il affecte un id unique au processus enfant.
c.) Il fait une copie logique du contexte parent.
d.) Il incrémente les compteurs dans la file table et dans la in-core inode table.
e.) Il renvoie l'id de l'enfant au parent et non pas à l'enfant.

User id d'un process

Le noyau associe 2 user à un processus :

- user id réel,
- user id effectif (« set uid »).

Le user id effectif est utilisé pour assigner la propriété des fichiers nouvellement créé et pour vérifier les permissions d'accès aux fichiers.
Le user id réel identifie l'utilisateur responsable du processus en cours d'exécution.

1. Comment allouer/désallouer un inode ?

Lors d'une demande d'accès à un inode, le noyau à partir du numéro d'inode trouve la hash queue* correspondante. Si l'inode demandé ne s'y trouve pas, le noyau alloue un inode.

* hashqueue : chaque buffer se trouve dans une hash queue en fonction du numéro du bloc qu'il contient. Si son statut est libre, il peut aussi se trouver chainé dans la liste des buffers libres.

2. Interruption (comment fonctionne un appel système d'une interruption ?)

Le système gère les interruptions quel que soit le type. Si le CPU travaille à un niveau moins important que l'interruption, il accepte l'interruption avant de passer aux autres instructions. Le CPU gère le niveau d'interruption de manière à ce que les autres interruptions de ce niveau, ou le plus bas, pose encore problème à l'instruction.

3. Read (commande)

Permet de lire son entrée standard et affecte les valeurs saisies dans les variables passé en argument.

4. Context Switch

Consiste à stocker et restaurer l'état d'un processus de sorte que l'exécution reprenne sans problème.

5. Qu'est-ce qu'un file system et un process ?

Un **file system** peut se composer de plusieurs unités disques physiques, chacune contenant un ou plusieurs file system. Un file system peut être vu comme une portion logique d'un disque physique. Le noyau considère chaque file system comme un device logique identifié par un logical device number.

Un **processus** est un programme en cours d'exécution. La seule façon de créer un nouveau processus en Unix est d'appeler l'appel système **fork** *(appel système permettant la création d'un nouveau processus)*. Sur un système multitâche, plusieurs processus peuvent être exécutés en même temps et un même programme peut être divisé en plusieurs processus, lesquels peuvent être exécutés en même temps et un même programme peut être divisé en plusieurs processus, également exécutables en même temps. Par exemple, plusieurs utilisateurs peuvent employer le même éditeur de texte simultanément.

6. Où, quand et comment détecter une erreur sur le mode d'ouverture quand un processus veut lire les données d'un fichier ?

Il s'agit du mode d'ouverture indiqué dans la file table.

7. Comment sont gérés les appels systèmes et les interruptions ?

https://perso.liris.cnrs.fr/pchampin/enseignement/se/noyau.html

Le système est entièrement responsable de la gestion des interruptions. Si le CPU travaille à un niveau plus bas que celui de l'interruption, il l'accepte avant de passer à l'instruction suivante. Il passe alors au niveau de l'interruption de manière qu'aucune autre interruption de ce niveau ou plus bas ne puisse interrompre la gestion de l'interruption en cours. Il préserve également l'intégralité des données structures du noyau.

- Il sauve le contexte courant du processus en cours d'exécution et crée un nouveau niveau de contexte.
- Il détermine la cause de l'interruption, via un nombre reçu lors de l'interruption, lequel sert de déplacement dans le vecteur d'interruption. Le contenu de l'entrée de cette table comprend l'adresse de la routine de gestion de l'interruption ainsi que la possibilité de lui donner des paramètres.
- Le noyau appelle la routine de gestion de l'interruption. La pile noyau pour le nouveau niveau de contexte est logiquement distincte de celle de l'ancien niveau.
- Lorsque la routine est terminée, le noyau restaure le contexte registre et la pile noyau du niveau de contexte précédent. Il reprend alors l'exécution du niveau contexte restaurée à l'endroit où il avait été interrompu.

8. Quand on a une erreur de type "no inode" d'où ça provient ?

2 cas (parler de la liste des inodes libres dans le super bloc et parler de la in-core inode table).

S'il n'y a plus d'inode libre dans le filesystem, il est impossible de créer un nouveau fichier jusqu'à la libération d'un inode (→ ***« ialloc »***).

9. Quelle est la différence entre la u_{area} et la table des processus ? Pourquoi utilise-t-on 2 tables ?

Deux structures de données du noyau décrivent l'état d'un processus : l'entrée de la **table des processus** et la **u_{area}**. 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. Dès lors, le noyau n'alloue de l'espace pour la u_{area} que lors de la création d'un processus ; il n'y a pas besoin de u_{area} pour les entrées non occupées de la table des processus.

10. Qu'est-ce qu'un processus zombie ?

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. On parle aussi de processus défunt *(en anglais "defunct")*.
La seule manière d'éliminer ces processus zombies est de causer la mort du processus père, par exemple au moyen du signal SIGKILL

11. Comment se font les lectures asynchrones ? (sylla 1.9)

Les modules de haut niveau du noyau peuvent anticiper les besoins d'un processus pour un bloc de données supplémentaire (lors de la lecture séquentielle d'un fichier, par exemple). Le noyau va effectuer la lecture de ce bloc en **asynchrone**.

1. Le noyau vérifie que le premier bloc est dans le buffer cache. Si ce n'est pas le cas, il appelle le disk driver pour le lire.
2. Si le second bloc n'est pas dans le cache, le noyau signal au disk driver de le lire en asynchrone.
3. Le processus attend éventuellement la fin des I/O pour le premier bloc. Lorsque le premier buffer est prêt, il est renvoyé, sans s'occuper de la fin des I/O pour le second bloc.
4. Lorsque celle-ci se produit, le contrôleur de disque lance une interruption. Le gestionnaire d'interruptions reconnaît qu'il s'agissait d'une lecture effectuée en asynchrone et libère le buffer du deuxième bloc.

Si cette dernière opération n'était pas faite, ce buffer resterait bloqué et donc inaccessible à tous les processus. Il n'est en effet pas possible qu'un processus débloque ce buffer car aucun n'a encore demandé les données qu'il contient. Ces données ne sont donc valides pour personne dans l'état actuel des choses. Ce buffer étant libéré, si plus tard un processus demande les données qu'il contient, il les trouvera dans le buffer cache.

12. Expliquer les gestions de signaux et comment ils s'exécutent (sylla 2.21 → 2.23)

Le noyau **gère** les signaux uniquement lorsqu'un processus passe de mode noyau en mode utilisateur. Dès lors, un signal n'a aucun effet immédiat sur un processus en cours d'exécution en mode noyau. Si le processus tourne en mode utilisateur, et que le noyau reçoit une interruption qui cause l'envoi d'un signal au processus, le noyau reconnaîtra et gèrera le signal au retour de l'interruption. Un processus n'est donc jamais exécuté en mode utilisateur avant d'avoir géré les signaux en attente. Le noyau gère les signaux dans le contexte du processus qui les reçoit ; il est donc nécessaire qu'un processus tourne pour pouvoir gérer les signaux.

3 cas de gestion des signaux peuvent se présenter :

- Le processus termine à la réception du signal (exit)
- Le processus ignore le signal
- Le processus exécute une fonction utilisateur particulière lors de la réception du signal

L'action par défaut est d'appeler exit en mode noyau, mais un processus peut spécifier une action spéciale à réaliser lors de la réception de certains signaux, et ce via l'appel système **signal**.

La **u_{area}** contient un tableau de champs gestionnaires de signal, un pour chaque signal défini dans le système. Si une fonction utilisateur doit être appelée à la réception d'un signal, le noyau stocke l'adresse de cette fonction dans le champ correspondant au numéro du signal traité. Une spécification pour la gestion d'un signal n'a aucune incidence sur la gestion des autres signaux.

Lorsqu'un processus reçoit un signal qu'il a précédemment décidé d'**ignorer**, il continue comme si de rien n'était. Vu que le noyau, dans ce cas, ne remet pas à 0 le champ de la u_{area} signifiant que le signal est ignoré, le processus l'ignorera encore s'il se produit de nouveau.

Si un processus reçoit un signal qu'il désire **gérer**, il exécute la fonction de gestion spécifiée par l'utilisateur, au moment où il repasse en mode utilisateur, après que le noyau ait effectué les étapes suivantes :

1. Le noyau accède au contexte registre sauvé de l'utilisateur, y retrouve le PC et le pointeur de pile qu'il y avait sauvegardé en vue du retour en mode utilisateur
2. Il efface le champ de gestion du signal dans la u_{area} et le remet à la valeur par défaut (exit du processus)
3. Il crée un nouveau cadre de pile sur la pile utilisateur, y écrit les valeurs du PC et du pointeur de pile qu'il vient de retrouver, et alloue du nouvel espace si nécessaire. La pile utilisateur a ainsi la même allure que si le processus avait appelé une routine utilisateur (celle qui s'occupe du traitement du signal) au point où il se trouvait avant de reconnaître le signal).
4. Le noyau modifie le contexte registre sauvé de l'utilisateur : il met le PC à la valeur de l'adresse de la fonction gestionnaire du signal et fait en sorte que le pointeur de pile prenne en compte l'accroissement de la pile utilisateur

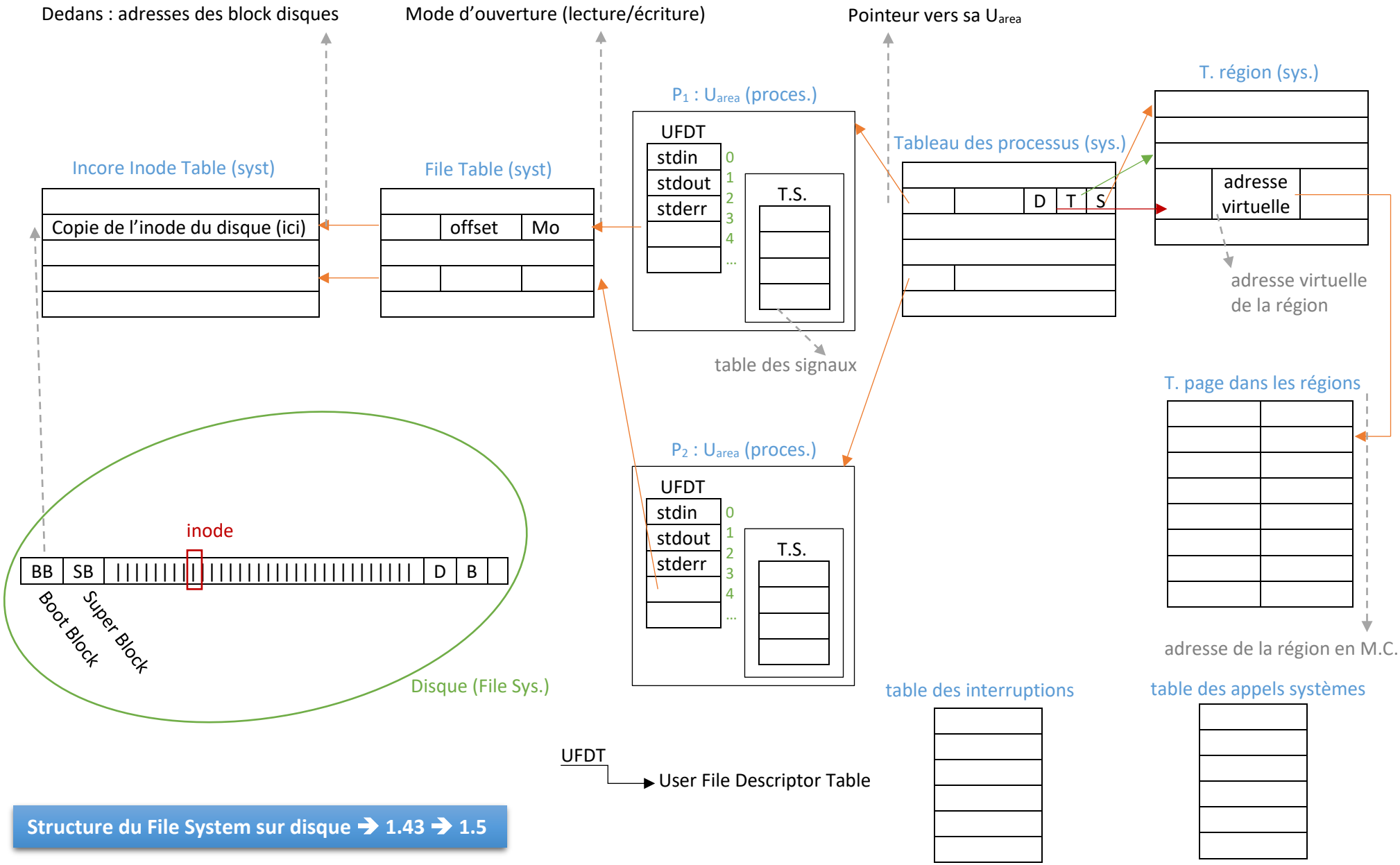
Après être revenu en mode utilisateur, le processus va exécuter la fonction de gestion du signal. Quand il en revient, il retourne à l'endroit dans le code utilisateur où s'était produit l'appel système ou l'interruption, mimant ainsi le retour véritable d'un appel système ou d'une interruption.

Exemple : Ctrl + C
➔ Envoi un signal au processus pour quitter le processus.

Melvin Campos Casares
EPHEC 2017-2018

Résumé M.C. (Mémoire Centrale)

Un pipe est un fichier



Structure du File System sur disque → 1.43 → 1.5