Notes du cours de système d'exploitation (SYS2)

Nathan Furnal

24 mai 2021

Table des matières

1 Système d'exploitation

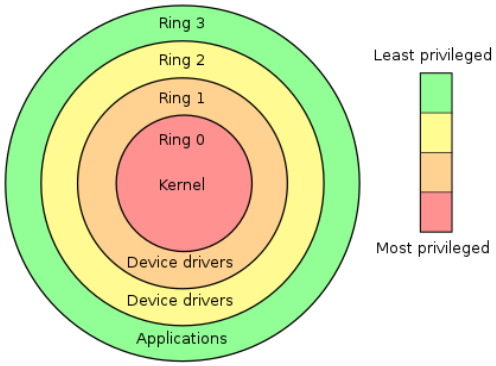
Un système d'exploitation joue le rôle à la fois de **gestionnaire de ressources** et de **machine étendue**.

Dans le cas de la machine étendue, il joue le rôle d'interface applicative, c'est-à-dire qu'il gère les périphériques et leurs interactions.

Dans le cas du gestionnaire de ressources, le système d'exploitation gère le CPU et la RAM. En pratique, quand plusieurs programmes demandent des ressources à l'ordinateur, l'OS va arbitrer quel programme reçoit quelle ressource à quel moment. La gestion passe par un **ordonnanceur** (scheduler) qui gère les processus par exemple, ce rôle est joué par le CPU. Tandis que la RAM charge des données en mémoire.

L'OS est du **logiciel**, le code de l'OS s'exécute en **mode privilégié**. Ce sont les logiciels, les applications qui peuvent basculer en mode privilégié, pas l'utilisateur. Ce mode dénote qu'un code a accès à la totalité des instructions et a accès à la totalité de la RAM. Alors que le code d'un utilisateur s'exécute en mode **non privilégié**.

Voici une représentation du niveau de privilège que les programmes peuvent avoir ou demander.



De ce fait, les applications et les programmes **doivent** passer par les services de l'OS pour accéder à des périphériques, elles n'y ont pas accès directement. Les limitations de l'OS se répercutent donc sur les programmes.

L'OS est matérialisé par du code : des appels système, des traitements d'interruption, des ordonnanceurs, des démons, etc. Et aussi par des données en RAM et sur le disque.

L'ordonnanceur désigne le composant du noyau du système d'exploitation choisissant l'ordre d'exécution des processus sur les processeurs d'un ordinateur. Alors que le chargeur est un composant du système d'exploitation dont le rôle est de charger des programmes en mémoire, afin de créer un processus. Ses principales responsabilités sont la lecture et l'analyse du fichier exécutable, la création des ressources nécessaires à l'exécution de celui-ci, puis enfin le lancement effectif de son exécution.

Finalement, le système d'exploitation gère le démarrage, les processus, l'accès au CPU, l'allocation de mémoire, les traitements d'interruption, les erreurs.

1.1 Rappel système de numération binaire

1. Puissance de 2 :
   KiB 2^{10} bytes et un byte = 8bits
   MiB 2^{20} bytes
   GiB 2^{30} bytes
   TiB 2^{40} bytes
2. Combien de blocs de 4KiB dans un espace de 32GiB ?
   32GiB/4KiB = $(32*2^{30})/(4*2^{10})$ = $(2^5 * 2^{30})$ bytes / $(2^2 * 2^{10})$ bytes = 2^{23} blocs.
3. Quelle est la représentation binaire de 43 ? et de `0x2B` ?
   — 43 codé sur 8 bits → 00101011
   — `0x2B` en binaire → 0010 1011 (donc 43 en décimal)
4. Quel est l'intérêt de la base hexadécimale ?
   La base hexadécimale permet de représenter l'information d'une manière plus compacte que le binaire.
5. Quelle est la différence de représentation de 11 et "11".
   La représentation en mémoire est différente, le premier est une valeur et le second, une chaîne de caractères. Un éditeur de texte pourra lire la chaîne de caractères mais essayera de traduire les valeurs binaires en ASCII, ce qui n'affichera pas correctement le contenu du fichier.

6. Comment visualiser du binaire ?
   On ne peut pas utiliser `vi` ou `ed` pour voir ces données, on doit utiliser des programmes comme `od`.
7. On peut trouver le modulo d'une division avec une puissance n de 2 en regardant les n bits de poids faibles. Les bits de poids forts restants sont le résultat de la division entière.

On utilise des puissances de 2 pour identifier les tailles et les blocs en mémoire. Cette représentation nous permet de calculer les restes d'une division sans avoir recours à l'ALU (unité informatique et logique), dans certains cas favorables.

2 Appels système

2.1 Définition

Avant tout,un appel système est du code l'OS. Il réalise un service pour l'OS et s'exécute en mode **privilégié**.

2.2 Privilèges

Quand on parle de privilège, c'est-à-dire qu'on a accès à la **totalité** de la RAM et qu'on a accès aux **instructions** d'accès au disque.

Il n'y a pas d'instruction spécifique pour passer en mode privilégié, le basculement de mode se fait au niveau du processeur (CPU).

— Les appels systèmes provoquent un basculement en mode privilégié.
— Les traitements d'interruption provoquent un basculement en mode privilégié.

2.3 Utilisation

Les services du système correspondent à un numéro (read = 0,...). Les numéros correspondant aux services du système, tout comme le mécanisme de basculement, changent selon l'architecture.

Pour utiliser un service un programme doit :

1. Fournir un numéro de service dans un registre qui dépend de l'architecture (`EAX` en 32 bits et `RAX` en 64 bits).
2. Fournir les paramètres nécessaires à l'appel système dans les registres `EBX`, `ECX` en 32 bits et `RDI`, `RSI` en 64 bits.
3. Provoquer le basculement dans le noyau et le saut de privilèges (via l'interruption logicielle `0x80` en 32 bits et via l'instruction `SYSCALL` en 64 bits).
4. Vérifier le statut d'erreur dans le registre `EAX` ou `RAX`.

Dans les exemples aux sections 32 bits et 64 bits, les trois premières étapes sont reprises.

On suit donc bien exactement les étapes décrites.

Les numéros des appels systèmes sont documentés dans des fichiers en langage `C`. Pour connaître les paramètres utilisables, on doit utiliser les manuels via la commande `man`, de niveau 2.

```
man 2 read
man 2 open
```

2.4 Intel 32bits

En 32 bits, `exit(0)` est traduit par :

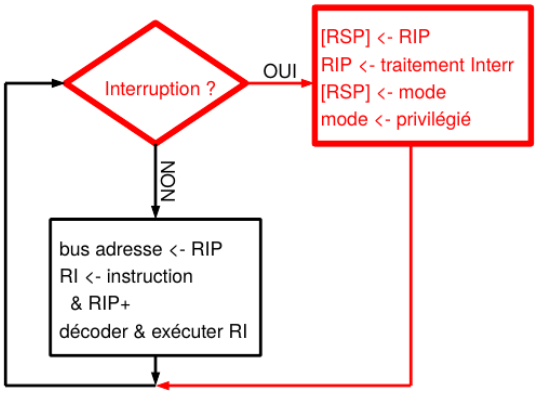
— `mov eax, 1`, le numéro du service exit en 32 bits.
— `mov ebx, 0`, le paramètre de l'appel système `exit`.
— `int 0x80`, l'interruption logicielle.

Pour rappel, `0x80` est le numéro de l'**interruption logicielle** qui assure le basculement et branchement sur le code de l'appel système en Linux.

2.4.1 Mécanisme d'interruptions

Le mécanisme d'interruptions provoque un basculement de mode du CPU.

D'abord le bus d'adresse reçoit `RIP` et `RI` reçoit l'instruction. Aussi, on incrémente `RIP` pour qu'il passe à l'instruction suivante. Puis, il faut **décoder** et **exécuter** l'instruction qui se trouve dans `RI`. Sans interruption, on ne fait que répéter les instructions précédentes. En pratique, à la fin de chaque instruction, le CPU vérifie s'il y a eu une interruption. Si c'est le cas, il met la valeur de `RIP` sur le sommet de la pile, pour pouvoir le récupérer plus tard (après l'interruption). On sauvegarde aussi le mode (privilégié ou pas) et puis `RIP` pointe vers l'interruption dans le code du noyau. Voici une représentation. Une instruction de `RI` se termine toujours à part si elle provoque une **erreur**, on parlera alors d'**exception**.



2.5 Intel 64bits

En 64 bits, `exit(0)` est traduit par :

```
1 mov rax, 60 ; n° du service exit en 64 bits
2 mov rdi, 0  ; paramètre de l'appel système exit
3 syscall
```

L'instruction `SYSCALL` assure le basculement et branchement sur le code de l'appel système.

2.6 Exemples d'appels systèmes

open Crée un descripteur en RAM pour un fichier qu'on souhaite lire ou écrire et y mémorise l'avancement dans le fichier.

read Transfère en RAM *n* bytes depuis un fichier.

write écrit *n* bytes depuis la RAM vers un fichier.

fork Clone un programme qui tourne en mémoire.

exit termine un programme qui tourne.

2.7 Questions

— Un appel système sur Linux en 32 bits provoque toujours la même interruption ?
Pour une architecture et un système donné, ici Linux 32 bits, on utilise bien toujours la même interruption. Pour Linux 32bits `0x80`, Pour Windows 32bits on utilise `0x21`. Donc la réponse est vraie.

— Donnez une brève définition d'un appel système.
C'est du code de l'OS qui réalise un service pour les applications et qui s'exécute en mode privilégié.

— Pourquoi un appel système est-il un passage obligé pour les programmes ?
Parce ça permet de protéger le système et empêcher les programmes de détruire des espaces importants. C'est un moyen centralisé de gérer les ressources, de manière sécurisée.

— `SYSCALL` est une instruction du processeur ?
Oui, c'est une instruction du processeur, celle d'interruption.

— `SYSCALL` est une instruction privilégiée du processeur ?
Non, `SYSCALL` n'est pas une instruction privilégiée, c'est l'instruction qui permet à du code non privilégié d'appeler l'OS.

3 Multiprogrammation et timeslicing

Cette section va concerner la **multiprogrammation** et le **timeslicing**. La multiprogrammation apparaît tôt dans l'histoire de l'informatique car la monoprogrammation était une énorme contrainte de temps et de ressources au niveau des CPU vu que le CPU était bloqué pendant les entrées et sorties lentes.

Multiprogrammation Action de charger plusieurs programmes en mémoire et qui s'exécutent de manière entrelacée alors qu'en monoprogrammation on a un seul programme en mémoire à tout moment.

Timeslicing Ajout de contrainte de temps à la multiprogrammation.

Quand on parle de **processus** ici-bas, ce sont les programmes chargés en mémoire. Ils s'exécutent en mode normal de manière habituelle et en mode **privilégié** quand on exécute du code système pour le compte du processus.

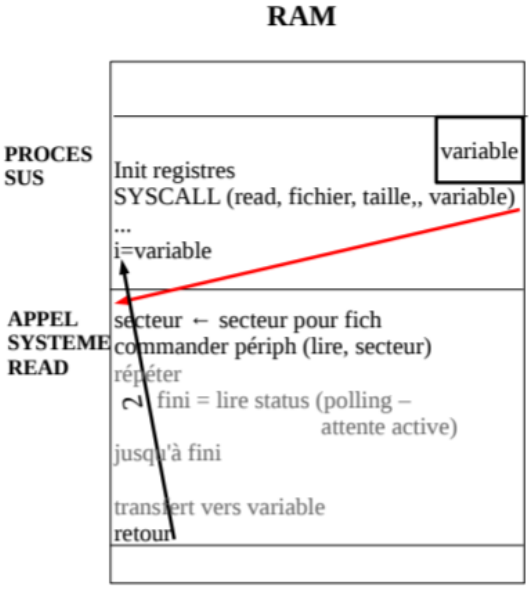
La **multiprogrammation** permet de ré-attribuer le CPU au moment d'une entrée/sortie (*input/output*).

Le **timeslicing** quant à lui, définit un temps maximum pendant lequel un processus peut utiliser le CPU sans interruption. Une fois ce délai passé, le processus n'a plus accès au CPU même s'il ne fait pas de demande d'entrée/sortie.

Nous allons d'abord voir ce qu'il se passe dans le cas de la monoprogrammation et puis les différentes stratégies pour mitiger ce problème.

3.1 Monoprogrammation

Voici un exemple de programme chargé en mémoire et exécuté en monoprogrammation.



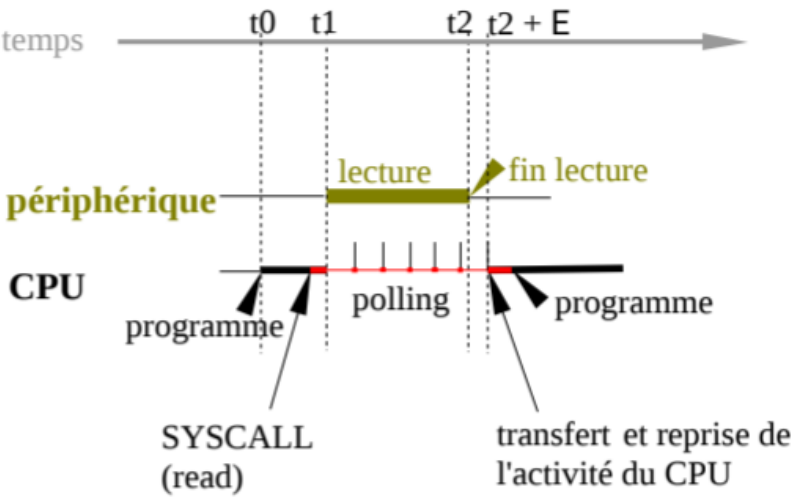
En ce qui concerne le code du **programme**, il est exécuté en mode normal, accède aux registres avec les paramètres nécessaires. Ensuite, l'opération `SYSCALL` provoque le basculement dans le noyau et donc le passage en mode privilégié.

Maintenant, le programme a perdu la main et donc on attend. On demande constamment au CPU si le basculement est terminé ou pas, cette opération s'appelle le **polling**. Une fois qu'on revient au mode normal (quand le **polling** indique que l'appel système est fini), on peut exploiter les données lues en RAM.

Entre temps, le code de l'appel système **read** qui a provoqué le basculement (via `SYSCALL`) va identifier le numéro de secteur nécessaire, commander au périphérique de lire le numéro de secteur et la donnée stockée à cet endroit. Puis, on rapatrie les données lues vers la variable et on rend la main au code utilisateur. Le **polling** se fait constamment durant cette période savoir si on peut reprendre le code utilisateur.

Dans ce cas de monoprogrammation, le CPU **attend** que le périphérique ait mis à disposition les données pour les transférer en RAM et reprendre son activité au profit du programme.

Ici, on voit la ligne du temps du processus et des appels systèmes. Le rouge indique le passage au mode privilégié.



3.2 Processeur canal ou DMA

Pour remédier au problème d'attente et d'usage inefficace du CPU ; on va associer au CPU un processeur externe relié par le bus (CANAL) et lui confier la gestion du périphérique et le transfert depuis la RAM.

On parle alors de processeur canal ou DMA (*Direct Memory Access*).

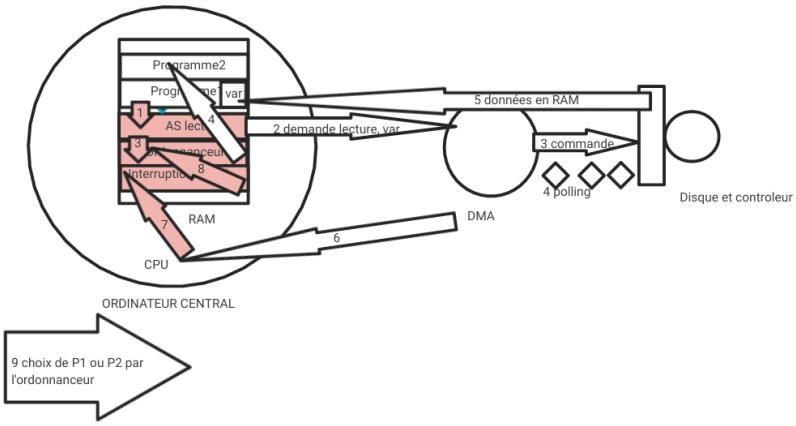
Ce processeur commande le périphérique et **accède à la RAM** pour transférer les données lues. Il génère une interruption pour prévenir de la fin de la lecture. Une fois que l'interruption est établie, le CPU est libéré au profit d'un deuxième processus chargé en RAM. Donc, ce processeur canal est relié au bus et accède à la RAM, il **interfère** avec le fonctionnement du CPU. De ce fait, il faut synchroniser le CPU et le canal (régler les accès à la RAM via le bus).

Grâce à cela, le CPU ne gère plus le polling et le transfert des données, ces tâches sont entièrement déléguées au DMA et permettent de rentabiliser l'usage du CPU. On peut maintenant exécuter plusieurs programmes de manière entrelacée et un programme qui demande une lecture est **bloqué** au profit d'un autre qui récupère le CPU.

Bien que cette stratégie permet d'améliorer l'usage du CPU, on a introduit de la complexité puisqu'il faut garder le compte d'où en est chaque processus. Pour ce faire on va :

- Mémoriser l'état des registres (`RIP`, `RAX`,...) du processeur via une **table des processus** dont le contenu est expliqué plus bas. On dira qu'on mémorise le **contexte** du CPU.
- Gérer qui est exécuté quand grâce à l'**ordonnanceur**.
- Protéger l'accès à la mémoire par la **segmentation** (sur Intel).
- Gérer l'attribution des ressources non partageables et les conflits dûs à l'**interblocage**.

Voici un exemple d'usage de multiprogrammation avec deux programmes, géré par le DMA :



3.3 Gestion des processus

Pour ce qui est du **contexte** et de la **table des process**, elle mémorise les attributions et cessions des processus et elle contient :

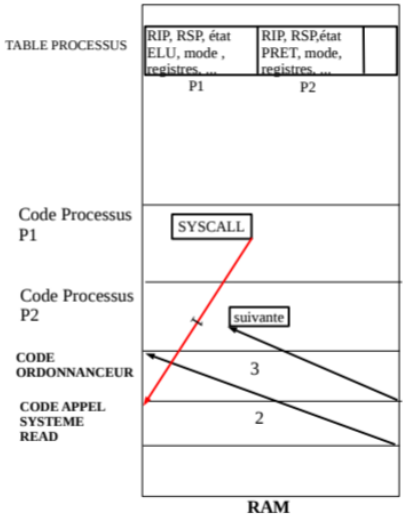
- Une valeur de `RIP` pour suivre où l'instruction se trouve dans le registre des instructions.

— Une valeur de RSP, le registre des pointeurs sur la **pile** (*stack*).
— Une état de **registres** du CPU (RAX, RBX,...).
— Un **mode** de fonctionnement du CPU (privilégié ou pas).
— Un **état** (élu, prêt, bloqué). Un processus est bloqué s'il est dans l'attente de fin d'une lecture/écriture. Prêt s'il est en attente et qu'il peut s'exécuter. Enfin, on dira le processus est **élu** s'il s'exécute et lui **seul**. Plusieurs demandes de lecture/écriture et donc plusieurs processus bloqués peuvent coexister.

Enfin, l'ordonnanceur choisit des processus **prêts** pour leur attribuer le CPU.

3.4 Déroulement d'une lecture

Maintenant, voyons le déroulement d'une lecture en multiprogrammation, avec les programmes P1 et P2. On utilise le mot "processus" pour un programme qui s'exécute en mémoire.



Dans le cas d'une demande de lecture on a :

1. P1 demande une lecture, ce qui provoque un SYSCALL.
2. On a un appel système.
   — état de P1 devient **bloqué**.
   — On sauvegarde le **contexte** de P1 dans la table des process.
   — On commande le périphérique et le processeur canal.
   — RIP prend l'adresse de l'ordonnanceur.
3. Ordonnanceur.
   — L'état de P2 devient **élu**, car c'est le seul prêt (l'autre est bloqué).
   — On utilise la table des process pour charger les registres et le mode de P2.
   — RIP prend la valeur de RIP de P2 qui était stockée dans la table des process.
4. Comme P2 à la main, il continue à s'exécuter.

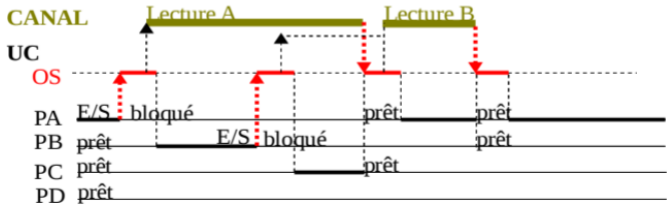
Maintenant, quand on arrive à la fin de la lecture :

1. Il y a une interruption due au processeur canal.
   — On met la valeur de RIP dans la pile (*stack*).
   — Le mode devient **privilégié**.
   — RIP prend l'adresse de la gestion d'interruption.
2. Traitement de l'interruption CANAL.
   — L'état de P2 passe à prêt.
   — On sauvegarde le **contexte** de P2 dans la table des process.
   — L'état de P1 passe à **prêt** car il a fini sa lecture.
   — RIP prend l'adresse de l'ordonnanceur.
3. Ordonnanceur
   — L'état de P1 ou de P2 dans la table passe à **élu** suivant le programme qu'on choisit.
   — On charge les registres et le mode de l'élu depuis la table des process.
   — RIP prend la valeur de RIP de l'élu, qu'on connaît aussi grâce à la table des process.
4. L'élu à la main et s'exécute.

La section suivante donne le découpage du temps pour les actions décrites au-dessus.

3.5 Multiprogrammation et occupation du CPU

Voici à quoi ressemble les tranches de temps de la multiprogrammation décrite au-dessus. Comme avant, le rouge indique le passage à du code de l'OS via un appel système.



Ce qu'on constate c'est qu'un processus qui fait de la lecture/écriture (entrée/sortie) rend la main et permettent donc d'optimiser l'usage du CPU. Par contre, les processus de calcul ne rendent jamais la main.

Donc, on voit que le canal est utile dans les cas d'entrées et de sorties mais qu'il n'est pas vraiment rentabilisé dans des cas de calculs. Une manière de mitiger ce problème est abordé dans la section suivante.

3.6 Time slicing

On a vu que dans les cas de calculs, le programme ne rend pas la main et bloque l'accès aux ressources. Si le calcul prend énormément de temps, on va limiter le temps qui est disponible pour ce process, pour rentabiliser le processus canal.

On va se servir de l'**interruption horloge** pour retirer le CPU au process qui l'utilise. C'est un mécanisme qui retire et assigne des tranches de temps aux processus.

En pratique, un processus garde le CPU pendant une tranche de temps de durée maximum t et après on lui retire au profit d'un autre. Dans ce cas, l'**ordonnanceur** est appelé par l'interruption horloge. On optimise l'utilisation du canal mais attention, on doit maintenant gérer les **changements de contexte** chaque fois qu'on passe de tranches en tranches.

Cette gestion de CPU permet l'exploitation interactive, comme partager du temps CPU entre utilisateurs, car elle minimise les temps de réponse. On parle alors de **time sharing**, qui est une notion différente du **time slicing** mais liée.

Les slides suivants décrivent les différents changements d'états pour les processus.

Élu → prêt Via une interruption, que ce soit une interruption horloge ou une interruption canal.

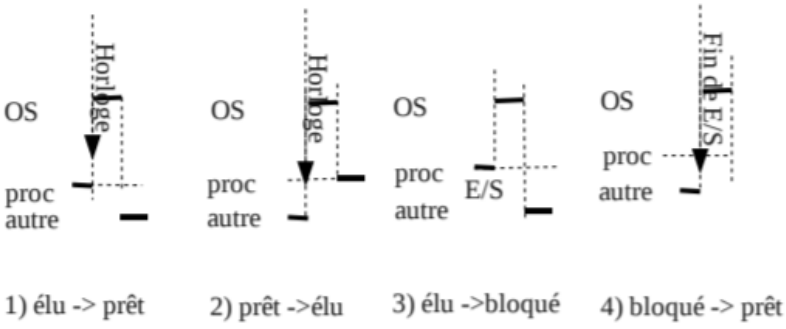
Prêt → élu Quand l'ordonnanceur choisi un processus.

Élu → bloqué Quand on demande une ressource indisponible (lecture, écriture, demande de ressource non partageable).

Bloqué → prêt Lors de l'interruption canal ou interruption liée à des ressources qui deviennent partageables.

On ne peut pas passer de prêt à bloqué car il faut demander des ressources ou être exécuté pour être bloqué, ce qui n'est pas le cas d'un processus prêt. Aussi, on ne peut pas passer de bloqué à élu car c'est l'ordonnanceur qui élit les processus et donc on doit d'abord être prêt avant de pouvoir être élu.

Voici un exemple de basculement d'états :



1. L'interruption horloge interrompt le processus et passe la main à l'autre. Donc l'élu devient prêt car son temps est fini.
2. Interruption du premier processus permet au second processus d'avoir la main, il passe de prêt à élu.

3. La demande d'E/S crée un appel système qui bloque le programme en cours, c'est donc un appel système qui fait basculer d'élu à bloqué. Dans ce cas il n'y a pas d'interruption à proprement parler, c'est l'appel système qui interrompt le processus.
4. C'est le canal qui va gérer le passage de bloqué à prêt, car les ressources utilisées lors de l'E/S sont libérées.

3.6.1 Préemption

Plus tôt, on a parlé de l'action d'allouer ou retirer le CPU au processus via le *time slicing*, c'est la préemption (le fait de prendre).

On parlera de :

système non préemptif Un processus qui se bloque (entrée/sortie) ou rend la main spontanément. Il n'y a pas de notion de *time slicing*.

système préemptif Un processus qui se bloque (E/S) ou rend la main spontanément ou bien lorsque son temps est écoulé (Windows NT, XP, Linux depuis le noyau 2.6).

système coopératif Un processus rend la main spontanément.

Il faut aussi savoir qu'un ordonnanceur préemptif ne peut pas exister sans interruption horloge et qu'il peut être non préemptif même en présence d'interruption horloge.

3.6.2 Précisions sur le time slicing

Du coup, on peut se demander quand à lieu l'ordonnancement exactement ?

1. Demande d'entrée ou sortie
2. Fin d'entrée ou sortie
3. En interruption d'horloge (préemption)
4. Nouveau processus
5. Fin d'un processus
6. ...

De manière générale, à chaque fin de traitement d'interruption et en cas d'appel système, on fait appel à l'ordonnanceur.

3.7 Questions

1. Si P est l'élu, alors RIP pointe vers une des instructions de P.
   Vrai
2. Combien de processus en mémoire ?
   Autant de processus qu'on veut tant qu'il y a de la place en mémoire.
3. Combien de **prêts** en même temps ?
   Le CPU ne s'arrête jamais, donc tous peuvent être prêts sauf un qui doit être élu.
4. Combien de **bloqués** en même temps ?
   Tous peuvent être bloqués sauf un qui doit être élu.
5. Un processus peut passer de l'état prêt à bloqué ?
   Faux.
6. Un processus peut passer de l'état bloqué à élu ?
   Faux.
7. Quels avantages et inconvénients à la time slicing par rapport à la multiprogrammation ?
   Le *time slicing* est utile quand on a des processus longs qui ne rendent pas la main. En limitant leur temps attribué, on peut mieux rentabiliser l'usage du canal. Mais l'usage de tranches oblige à conserver les contextes et donc les changements de contexte prennent du temps, ce qui représente un inconvénient. Dans le cas de la multiprogrammation, les entrées et sorties sont très rapide car il n'y a pas changement de contexte mais s'il y a des processus qui bloquent le CPU pendant longtemps, alors on ne rentabilise pas le canal.
8. Donnez des cas où l'état d'un processus bascule de élu à prêt et un cas de bloqué à prêt.
   Un process passe de élu à prêt par une interruption (horloge ou canal) et un processus passe de bloqué à prêt par une interruption canal.

4 Ordonnancement

L'ordonnanceur est du code de l'OS qui choisit le prochain processus élu parmi les processus prêts.

Ce choix est complexe et il faut prendre en compte plusieurs contraintes pour un "bon" algorithme d'ordonnancement. Ces contraintes sont souvent mutuellement exclusives donc optimiser une contrainte peut créer des ralentissements dans d'autres contraintes.

- Équité : Les mêmes processus prennent un temps similaire.
- Temps de réponse : Un processus ne bloque pas les réponses d'autres processus pendant trop longtemps. Important pour les systèmes interactifs.
- Temps d'exécution : Un processus s'exécute rapidement alors que les entrées et sorties pénalisent le temps d'exécution.
- Rendement/efficacité : Utilisation correcte du temps et des ressources du CPU.
- Équilibre : Utilisation correcte et équilibrée des différentes parties du système.

On voit donc qu'il faut trouver un compromis entre ces contraintes. Il n'existe malheureusement pas d'algorithme magique qui optimise l'utilisation du CPU dans toutes les situations. On peut donc adopter plusieurs stratégies décrites dans les points suivants, qui supposent généralement l'utilisation du *time slicing*.

4.1 Tourniquet

Avec cette stratégie, chaque processus **prêt** reçoit, à tour de rôle, un quantum du temps processeur.

Un processus perd le CPU :

- parce qu'il a terminé
- parce qu'il a besoin d'une entrée/sortie (élu $\rightarrow$ bloqué)
- parce qu'il a épuisé son quantum de temps.

Puis, l'ordonnanceur élit le suivant de la liste.

On doit être prudent dans le choix du quantum de temps. Voici un exemple avec 10 processus prêts en même temps et un changement de contexte qui prend **5ms**. Le dénominateur de l'efficacité est la somme du quantum et du changement de contexte.

quantum	temps max attente	efficacité CPU
5 ms	95 ms	50 % = 5/10
100 ms	+- 1 seconde	95 % = 100/105
1 sec	+- 10 secondes	99.5 % = 1000/1005

4.1.1 Priorité

Un soucis du tourniquet et qu'il place tous les processus sur un même niveau de priorité alors que certains processus peuvent être importants et devraient être traités au plus vite.

Pour remédier à ça, on associe des priorités aux processus et l'ordonnanceur élit les processus dans la liste, par priorité décroissante. Cette solution est avantageuse mais risque de laisser les process de priorité basse sur le côté, on parle alors de **famine**.

Une solution est décrite au point suivant, en utilisant des priorités dynamiques.

4.2 Priorités dynamiques

4.2.1 Solution par décrémentation

Une solution au problème de famine est de décrémenter la priorité d'un processus chaque fois qu'il a accès au processeur. Grâce à cette solution, on peut permettre à des processus basse priorité d'avoir quand même accès au processeur.

Par contre, on oublie la priorité initiale du processus et s'il est important et tourne souvent, on le pousse constamment vers le bas des priorités, ce qui n'est pas optimal non plus, une autre solution doit être choisie.

4.2.2 Solution par quantum utilisé

Du coup, une autre idée sera de favoriser les processus qui demandent beaucoup d'entrées et sorties en leur attribuant tout de suite le processeur, en ajustant leur priorité en fonction du temps déjà utilisé.

Un processus qui demande beaucoup d'entrées et sorties doit être favorisé car ils sont exécutés en parallèle avec le processeur.

$$\text{priorité} = \frac{\text{quantum}}{\text{temps du quantum utilisé}}$$

Par exemple, un processus qui a un quantum de 100ms et est bloqué par une demande d'entrée / sortie après 5ms. Alors, il aura une priorité 100/5 = 20. On voit que moins le temps a été utilisé, plus la priorité est haute.

4.3 Files multiples ou classes

On a vu que la question des priorités était problématique et on remarque que pour les processus qui utilisent beaucoup le CPU, la performance augmente quand le quantum est grand. Tandis que pour les processus qui font beaucoup d'entrées et sortie, on a un meilleur temps de réponse quand le quantum est petit.

De là, on tire la conclusion qu'on peut attribuer un grand quantum à un processus qui demande beaucoup de calcul et une **priorité** plus grande ainsi qu'un plus petit quantum pour les processus qui demandent beaucoup d'E/S.

Voilà un exemple avec 5 tourniquets, un par classe. Tous les processus reçoivent un quantum en fonction de leur classe et un processus de classe inférieure est élu quand aucun processus de classe supérieure n'est en état prêt.

Classe	Quantum	priorité	remarque
1	40 ms	5	pour un processus d'E/S
2	80 ms	4	...
3	160 ms	3	intermédiaire
4	320 ms	2	...
5	640 ms	1	pour un processus de calcul

Mais attention, comment connaître la classe d'un processus *a priori* ? On ne la connaît pas et on choisira de la fixer au départ.

Par exemple, on mettra tous les processus à la classe 1. Ce qui représente un désavantage car on voulait les classer et donc on choisira de les faire évoluer en fonction de leur usage du CPU.

Chaque fois qu'un processus consomme tout son quantum, il descend d'une classe. De ce fait, les processus qui font des E/S restent en classes élevées et les processus de calcul baissent de classe et ont un plus grand quantum.

Mais qu'en est-il des processus hybrides ? Par exemple un processus qui demande des E/S et qui fait des calculs. Pour remédier à ce problème, on augmente la classe de 1 chaque fois qu'un processus demande une E/S. C'est une bonne stratégie dont on peut cependant abuser car si un programmeur connaît cette stratégie alors il peut créer un processus qui demande des E/S après chaque calcul pour s'assurer de rester en classe haute.

4.4 Questions

— L'ordonnancement du tourniquet utilise le plus souvent un quantum fixe. Donnez un argument en faveur d'un petit quantum et un argument en faveur d'un grand quantum.
Le petit quantum permet de minimiser le temps d'attente mais a une faible efficacité d'utilisation du CPU, car le temps qu'on passe pour les changements de contexte est proportionnellement plus grand.
Le grand quantum permet de maximiser l'usage du CPU et donc son efficacité car le changement de contexte est proportionnellement plus faible. Par contre, on a un temps de réponse qui augmente et peut être problématique.

— Soit un quantum de 92ms et un temps d'ordonnancement de 8ms, calculez l'efficacité du processeur.
On mesure le temps utile sur le temps total donc :

$$\frac{92}{(92+8)} = 92\%$$

— Donner une haute priorité à un processus favorise sont temps de réponse.
Vrai.

— Avec un ordonnancement par tourniquet un processus s'interrompt pour une seule raison.
Faux, il y a aussi fin de tâche, demande de ressource (E/S) ou interruption matérielle.

— Un ordonnanceur à priorités statiques améliore le temps de réponse de tous les processus.
Faux, car il peut y avoir famine pour certains processus de basse priorité.

5 Mémoire

Cette section traite de l'usage de la mémoire.

5.1 Introduction

Dans une vue idéale de la mémoire, chaque processus doit disposer d'une mémoire :

— privée
— infiniment grande
— rapide
— non volatile
— réalisée dans une technologie "bon marché"

TYPE	TAILLE	RAPIDE	NON VOLATILE	BON MARCHÉ
cache	MiB	++	NON	-
mémoire vive	GiB	+	NON	+-
disques	TiB	-	OUI	+

Le meilleur compromis se trouve au niveau de la RAM (*Random Access Memory*) qui est utilisée pour les instructions et variables d'un programme.

On y accède aux "mots" via des adresse. Le "mot" est l'**unité adressable**. Chaque "mot" en RAM correspond à une adresse. En architecture Intel, le "mot" est le **byte**. En déposant l'adresse sur le bus d'adresse, on peut lire ou modifier le "mot" en RAM.

5.2 Sans abstraction

Sans abstraction de la mémoire, le programme manipule des adresses **physiques**.

```
1 MOV [8192], reg
```

Dans ce cas, 8192 correspondant à une vraie adresse en RAM. Le programme est maître des adresses qu'il utilise en RAM. Donc, deux programmes pourraient utiliser la même adresse physique, ce qui cause énormément de problèmes. Donc, ce mode d'adressage ne peut être utilisé qu'en monoprogrammation (on exécute un seul programme à la fois).

Si on utilise plusieurs programmes à des moments différents, on doit s'assurer qu'ils écrivent à des endroits différents en mémoire. Ce mode d'adressage ne permet pas la multiprogrammation.

5.3 Abstraction de la mémoire

Pour pallier les problèmes de la monoprogrammation, on va utiliser la **relocation statique**. Ce qui signifie que le programme utilise des adresses relatives ajustées au **chargement**.

5.3.1 Relocation statique

En relocation statique, la traduction des adresses relatives en adresses physiques est réalisée une seule fois au **chargement** du programme.

Toute référence à la mémoire est corrigée en y ajoutant l'adresse de chargement du programme. Le programme qui s'exécute utilise les **adresses physiques** que le chargeur (*loader*) aura adaptées. Par exemple, l'adresse 8192 deviendra 108192 pour le programme chargé en 100000 et 308192 pour le programme chargé en 300000.

Après le chargement du programme, toutes les adresses auront été converties en adresses **physiques**. En relocation statique, l'utilisation des adresses est absolue, on y fait directement référence.

5.3.2 Relocation dynamique

En relocation **dynamique**, les adresses du programme qui s'exécutent sont exprimées par rapport à un espace d'adressage propre au programme et chaque programme a son ou ses propres espaces d'adressage.

Aussi, les adresses ne sont pas modifiées au chargement du programme. Elles sont exprimées relativement à l'espace d'adressage. Aussi, en Intel 32bit, l'espace d'adressage est limité à 4GiB.

Ce qui est dynamique, c'est la traduction de chaque instruction à la volée au **moment de l'exécution**. Ce processus est géré par un dispositif **hardware**. Sur Intel c'est le **MMU** (*Memory Management Unit*) qui est responsable de la traduction des adresses à l'exécution.

En relocation dynamique, les adresses sont relatives, on les exprime par rapport à une base jusqu'à une certaine limite de taille. En pratique, deux registres **spéciaux** mémorisent l'adresse de chargement de la **base** et de la **limite**. Lors de chaque accès, l'adresse relative est convertie et on vérifie qu'on est bien dans les limites permises.

Cette vérification se fait par un dispositif **hardware**. Si on dépasse la limite, le **MMU** génère une exception (*Segmentation fault*).

Dans le cas dynamique, exécuter une instruction nécessite **deux accès** à la RAM et donc deux traductions. Il faut lire l'instruction depuis la RAM et écrire en RAM le contenu du registre.

On a donc une plus grande flexibilité mais les corrections d'adresses ralentissent l'exécution du programme.

En conclusion, la relocation dynamique :

— Permet la **coexistence** des programmes en mémoire et la protection de leurs espaces respectifs. (+)
— Le **swapping** et le déplacement sont facilement envisageables car il n'y a pas de corrections à apporter au programme étant donné qu'on modifie le registre de base. (+)
— Par contre, chaque accès mémoire nécessaire une **addition** et une **comparaison**, ce qui ralentit l'exécution. (-)

5.4 Taille et *swap*

Parfois, il arrive que la taille de la mémoire physique soit insuffisante pour accueillir l'ensemble des programmes qui tournent en même temps. Deux approches permettent d'éviter le problème.

1. Le *swapping* des programmes.
2. Le *swapping* des parties de programme.

Dans le premier cas, un programme sort de mémoire au profit d'un autre. Aussi, tous les programmes n'utilisent pas le même espace en RAM. Cette stratégie crée donc de la **fragmentation** de la RAM qui est coûteuse à défragmenter. On parle de fragmentation quand la mémoire présente une multitude d'espaces libres mais de taille insuffisante à accueillir un nouveau processus. On défragmente pour libérer des espaces contigus.

Dans le second cas, on utilise le fait qu'un programme ne doit pas entièrement résider en RAM à tout moment de son exécution. On va donc charger des morceaux de programme, appelés **pages**. Seulement une partie des pages des programmes résideront en RAM à un instant donné. C'est ce qu'on appelle la **pagination** ou mémoire virtuelle.

5.5 Segmentation (mode réel et protégé)

L'espace d'adressage physique d'un processus peut être séparé en segments (code, données,...).

La segmentation sur l'architecture Intel est une gestion de la mémoire à deux niveaux : le mode **réel** et le mode **protégé**.

5.5.1 Mode réel

Dans les architectures Intel, le mode réel n'est accédé qu'au moment du démarrage de l'ordinateur, pendant un court instant. Tout ce qui concerne les **rings** ne se déroule qu'en mode protégé. Ce dernier est utilisé sur la majorité des machines actuelles.

Le mode réel est le seul mode d'adressage des premiers processeurs 8086. Comme dit au-dessus, ce mode d'adressage réel subsiste dans les processeurs actuels, uniquement au démarrage. Le processeur accède à 1 MiB de RAM dans ce mode.

5.5.2 Mode protégé

La manière de traduire les adresses différencie les deux modes ; en mode protégé la traduction d'adresse se fait en plusieurs étapes :

$$\text{adresse logique} \rightarrow \text{adresse linéaire} \rightarrow \text{adresse physique}$$

La dernière étape est nécessaire si on utilise la pagination. Si on est en segmentation pure (pas de pagination), l'adresse linéaire est égale à l'adresse physique.

Pour bien comprendre ces traductions, on doit comprendre ce qu'est une adresse **logique**.

Une adresse **logique** est une adresse dans un segment, elle est composée de deux parties :

— Le **registre sélecteur** et l'**offset**.
— Les **registres sélecteurs** :

`CS` associé au segment de code

`DS` associé au segment de données

`SS` associé au segment de pile

...

Un programme utilise plusieurs segments et leur utilisation est parfois implicite :

— `jmp boucle` → `jmp CS:boucle`

— `mov eax, [ebx]` → `mov eax, [DS :ebx]`

— `push eax` → utilise `SS` :`ESP`

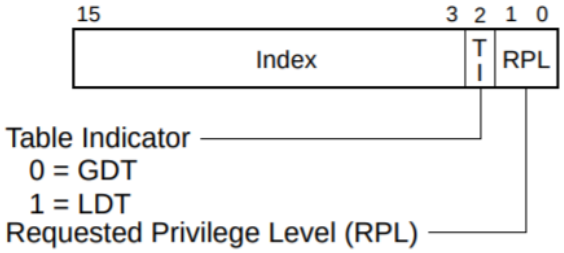
Plus précisément, le **sélecteur** de segment est codé sur 16 bits et on retrouve les registres mentionnés juste avant (`CS`, `DS`, `SS`, ...). On a aussi l'**offset** dans le segment, sur 32 bits.

Chaque segment est un espace d'adressage limité par une base et une limite, comme expliqué dans la section sur la relocation dynamique.

L'architecture Intel prévoit 6 registres sélecteurs de segment, ce sont : `CS`, `SS`, `DS`, `ES`, `FS`, `GS` qui sont tous des registres de 16 bits. Un programmeur peut utiliser plusieurs segments mais seulement 6 seront disponibles en même temps.

5.5.3 Sélecteur de segment

Voici une représentation du sélecteur de segment, il est composé d'un **index** sur 13 bits et donc au maximum 2^{13} descripteurs de segments sont possibles par table. Il y a aussi un indicateur de table sur 1 bit qui précise si on a affaire à la *global description table* (0) ou la *local description table* (1). Enfin, il y a le niveau de privilège demandé (*requested privilege level*) codé sur 2 bits et qui prend valeur entre 0 et 3, faisant référence aux **rings**. On a donc bien 16 bits au total.



Pour trouver l'adresse linéaire, on va simplement additionner la **base** et l'**offset**, on trouve la base grâce au sélecteur de segment.

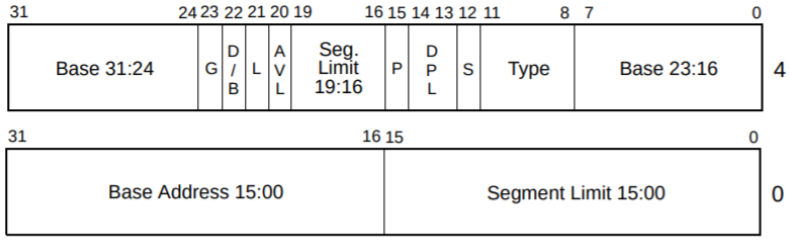
Dans le sélecteur, on a mentionné avoir 1 bit pour savoir si on utilise la table de description globale ou locale. Ce sont deux tables de 8 *bytes* en RAM qui rassemblent les **descripteurs** de segments. La table globale est partagée par tous les process tandis que la table locale appartient à un process. Ce sont les registres `gdtr` et `ldtr` qui contiennent les adresses des tables.

On a aussi mentionné les 2 bits de privilège, ils représentent le *requested privilege level* (*RPL*) ou le *current privilege level* (*CPL*) dans le cas du *code segment*. C'est donc le niveau de privilège demandé ou accordé.

Finalement, l'index de 13 bits sélectionne un des 8192 descripteurs possibles dans la table globale ou locale des descripteurs. Le processeur multiplie la valeur de l'index par 8 (le nombre de *bytes* dans un descripteur de segment) et ajoute le résultat à la **base** de l'adresse de *GDT* ou *LDT*, provenant des registres **GDTR** ou **LDTR**. On a donc $GDTR + \text{index} * 8$ ou $LDTR + \text{index} * 8$; pour trouver l'adresse du descripteur.

5.5.4 Descripteur de segment

Voici une représentation plus complexe, celle du descripteur. On décrira seulement quelques informations pertinentes. Toutefois, on doit comprendre ce que sont la **base**, la **limite** et la **granularité**. La base fait 32 bits, c'est là qu'on trouve l'adresse 0 d'un segment, elle peut être allouée dans un espace de 4 *gigabytes* (GiB). La limite fait 20 bits et spécifie la **taille** du segment. Le processeur interprète la limite de deux manières suivant que le drapeau de granularité soit levé ou pas. Si la granularité est à 0, le segment peut être de taille 1 *byte* à 1 MiB, par incréments en *byte*. Si la granularité est à 1, le segment peut être de taille KiB à 4 GiB, par incréments de 4KiB. Dans le premier cas on utilise le byte comme unité et dans le second on utilise la **page** (donc la pagination). Sans pagination, la base est une adresse physique.



L — 64-bit code segment (IA-32e mode only)
AVL — Available for use by system software
BASE — Segment base address
D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL — Descriptor privilege level
G — Granularity
LIMIT — Segment Limit
P — Segment present
S — Descriptor type (0 = system; 1 = code or data)
TYPE — Segment type

En présence de pagination, la **base** est une adresse dans l'espace d'adressage du programme.

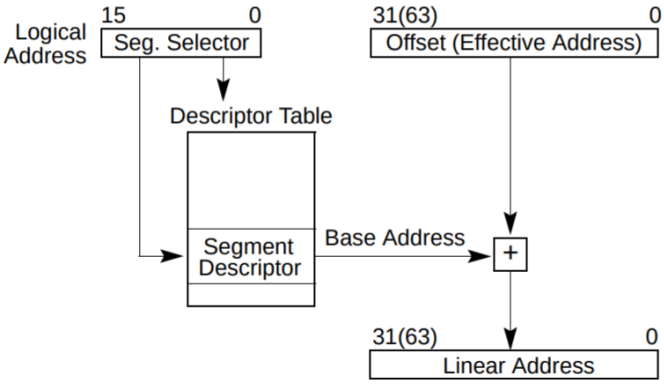
La taille maximum d'un est exprimée par la **limite** et la **granularité**, elle est de 4GiB en pagination.

On a aussi le *descriptor privilege level* (*DPL*) qui donne le niveau de privilège du segment, 0 étant le plus privilégié. Le *DPL* sert à contrôler l'accès au segment. On voit aussi dans la table, le **type** qui donne les droits qu'on a sur le segment : lecture, écriture, exécutable.

Enfin, il y a un drapeau (*P*) qui permet d'indiquer si un segment est présent ou non et qui est utilisé lors du *swap*.

5.5.5 Traduction d'adresse logique en adresse linéaire

Voici une image qui donne le processus de traduction d'adresse logique en adresse linéaire.



5.5.6 Performance de segmentation et protection

Quand on exécute du code et qu'on déplace des données, le **MMU** traduit l'adresse et chaque référence à la RAM requiert une lecture supplémentaire pour le descripteur de la table. Par exemple dans le cas suivant :

```
1 mov ebx, [0x1000]
```

Il y a 3 accès nécessaires, c'est un acte coûteux qu'on va essayer de réaliser le moins de fois possible.

Pour ça, on considère le principe de **localité**. On mémorise les derniers descripteurs de segments utilisés (64 bits sont utilisés). On divise un sélecteur de segment en une partie visible sur 16 bits et une partie cachée sur 8 *bytes* qui contiennent des informations en *cache*, ceci accélère la traduction. De plus, le descripteur est lu uniquement quand on modifie le sélecteur.

En ce qui concerne la protection, on a plusieurs parties de segments qui en sont responsable. Le niveau de privilège (**rings**), le type d'accès (lecture, écriture, exécutable) et la limite de taille. Les privilèges d'un code qui s'exécute sont marqués dans le sélecteur `CS` via le *CPL* allant de 0 à 3. Aussi, les privilèges nécessaires pour accéder à un segment de données se trouvent dans le descripteur de segment (*DPL*) allant lui aussi de 0 à 3.

Ces conditions de protections sont vérifiées et génèrent des erreurs lorsqu'elles sont enfreintes. Un privilège insuffisant générera une exception de type **general protection fault**, en général quand le *CPL* > *DPL*. Le dépassement d'une limite entraînera une exception de type **segmentation fault**.

5.6 Questions

1. Est-ce qu'un sélecteur de segment est un décalage par rapport à la RAM ?

   Un sélecteur n'est pas un décalage par rapport à la RAM ou par rapport au début du segment, ce n'est pas un décalage du tout, c'est l'*offset* qui est un décalage.

2. Comment est représentée une adresse logique ?

   Une adresse logique est une adresse dans un segment, elle est composée de deux parties. Le registre sélecteur et l'offset. Le sélecteur indique où se trouve la description qui lui est assignée. L'offset est un décalage.

6 Interblocage

Lorsqu'on a plusieurs processus, ils partagent de la mémoire, le CPU et aussi des **ressources**. Comme les mémoires de masse, l'imprimante, le graveur, les informations, les entrées de la table des processus, etc.

6.1 Ressources

On sépare les ressources en deux grandes classes : les ressources **partageables** et non partageables et les ressources **préemptibles**.

Ces propriétés sont liées à la nature des ressources mais aussi à la manière dont ces ressources sont gérées. Par exemple, une imprimante est par sa nature **non partageable** mais elle devient **partageable** par l'utilisation d'un **spooler** d'impression (qui sera expliqué plus tard).

— Une ressource est dite **non partageable** si elle doit être allouée de manière exclusive à un processus.
— Une ressource est dite **non préemptible** si elle ne peut être retirée à un processus sans dégâts.

Par exemple, si on utilise un graveur lors d'une exécution **alternée** de deux processus accédant à ces ressources, on peut créer une situation difficile.

Une demande de ressource **non partageable** et **non préemptible** attribuée à un autre, **bloque** un processus jusqu'à disponibilité de la ressource. Cette situation ne cause pas forcément un interblocage mais crée la possibilité de le faire.

L'**interblocage** apparaît quand plusieurs processus se bloquent mutuellement indéfiniment. On parle alors d'**étreinte mortelle** ou **interblocage** ou **deadlock**.

6.2 Exemple

Imaginons l'utilisation **simultanée** de deux ressources **non partageables** et **non préemptibles** comme le scanneur et le graveur de CD par deux processus :

processus A

— obtenir scanneur
— obtenir graveur
— scanner et graver
— rendre graveur
— rendre scanneur

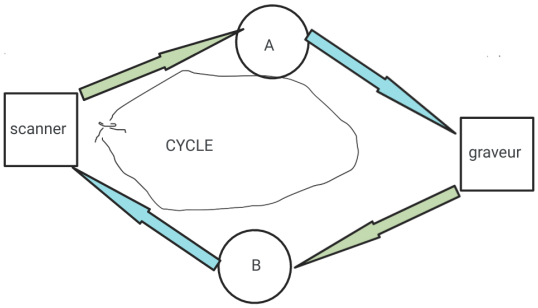
processus B

— obtenir graveur
— obtenir scanneur
— scanner et graver
— rendre scanneur
— rendre graveur

Voici une séquence possible d'exécution, elle entraîne un interblocage mais ce n'est pas nécessairement le cas, ça dépend du choix fait par l'ordonnanceur.

1. Le processus A obtient le scanneur
2. Le processus B obtient le graveur
3. Le processus A demande le graveur et ne peut l'obtenir, il est donc bloqué. Il possède déjà le scanneur.
4. Le processus B demande le scanneur et est donc bloqué. Il possède déjà le graveur.

Dans ce cas-ci, on crée une situation d'interblocage. Les processus intéressés sont bloqués de manière **irréversible**. Si on représente les demandes et les attributions par un graphe, on voit qu'on a un cycle et donc un blocage.



Une telle situation ne peut se présenter que dans les cas suivants :

— ressources **non partageables**
— ressources **non préemptibles**
— **plusieurs** de ces mêmes ressources sont nécessaires **simultanément** à **plus d'un** processus.

Il s'agit de conditions **nécessaires non suffisantes**. C'est-à-dire qu'il **faut** que ces conditions soient réunies pour créer un interblocage mais qu'elles ne suffisent pas, on doit encore avoir d'autres conditions en plus.

En clair, elles sont nécessaires car dans les cas suivants, on n'a pas de possibilité d'interblocage.

— Une ressource **partageable** ou **préemptible** peut toujours être obtenue.
— Dans le blocage d'utilisation d'une **ressource unique** le blocage est temporaire : le processus qui a la ressource se termine et cède la ressource à l'autre.

6.3 Solutions

Il y a plusieurs solutions à l'interblocage, on peut simplement l'ignorer, le reste du système continue de fonctionner correctement et s'il pose un problème, un administrateur peut tuer le processus en interblocage.

On peut aussi choisir de le **détecter** et le **corriger**, par exemple en analysant un graphe des attributions de ressources. On peut aussi l'éviter quand il se présente en ne permettant pas aux ressources d'entrée en situation d'interblocage. On peut aussi, l'éviter en gérant les processus de manière à ce qu'ils ne puissent tout simplement pas rentrer en situation d'interblocage.

Comme toujours, ces techniques ont des coûts, décrits dans les points suivants.

6.3.1 Ignorer l'interblocage

Certains OS comme Linux ou Windows font le choix d'ignorer la situation d'interblocage. En effet, si ces situations sont rares, pourquoi se donner la peine d'avoir un dispositif de gestion complexe ?

6.3.2 Détecter et reprendre l'interblocage

Une alternative à juste ignorer l'interblocage est de laisser les processus fonctionner et à corriger les problèmes lorsqu'ils apparaissent.

Détection

— On représente l'état d'allocation ou de demande par un graphe.
— On détecte les cycles dans le graphe.

Reprise

— Suppression du processus
— Préemption sur la ressource
— Technique de *rollback* (point de reprise)

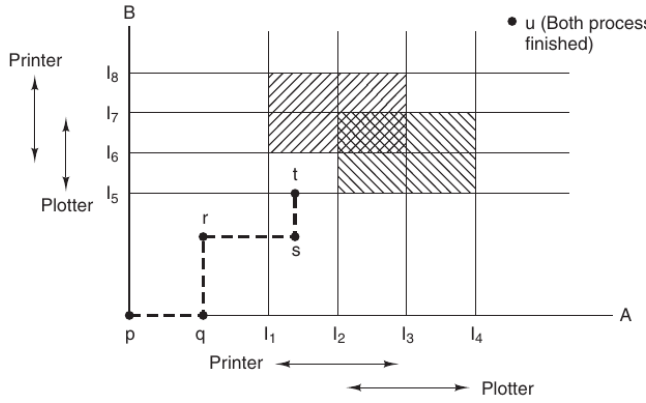
Si on représente par `R1` $\rightarrow$ `P1` le fait que la ressource `R1` est allouée à `P1`. Par `R2` $\rightarrow$ `P2` le fait que `P2` nécessite la ressource `R2`, cette situation peut être détectée par la présence d'un cycle dans le graphe correspondant à l'état des ressources demandées/obtenues.

6.3.3 Éviter l'interblocage

On peut réfléchir l'attribution des ressources mais cela nécessite des informations sur les **attributions futures**. Cette solution est basé sur la notion d'état **sûr**. C'est un état tel qu'il existe une séquence d'ordonnancement permettant à tous les processus de se terminer.

Voici comment cette solution est représentée dans le livre de Tannenbaum. Chaque point (p, q, r, s, t) représente un état des deux processus, chaque I est une instruction. Lorsqu'on est dans une case blanche, c'est un état sûr,

on peut donc continuer à attribuer des ressources au processus. Au point *t*, on rentre dans un état non-sûr car les cases à droite ou au-dessus peuvent créer un interblocage. Les zones hachurées représentent des zones où deux processus accèdent à la même ressource non-partageable et non-préemptible. Dans ce cas, la solution sûre est de continuer jusqu'à l'instruction 4, donc on bloque B pour laisser A se dérouler. Le *plotter* est libéré et le processus B peut continuer sans encombre.



6.3.4 Prévenir l'interblocage

La dernière solution propose de prévenir entièrement les cas d'interblocage en rendant une ressource partageable. C'est le cas du **spooler** et de son **répertoire**. Le *spooler* assure que plusieurs processus peuvent accéder à une ressource et gère leur accès. En cas de *spooling*, aucun processus n'a accès directement à l'imprimante à l'exception du **spooler** d'impression. Le **spooler** est un processus dédié aux impressions. Les processus qui demandent des impressions écrivent dans des **fichiers temporaires** déposés dans un **répertoire** dédié au spooler.

On peut aussi utiliser l'**allocation en bloc**, c'est-à-dire qu'on exécute un bloc d'instruction d'un processus en une fois, sans que les soucis d'accès multiples. Cette méthode n'est pas optimale car on doit attendre la fin d'exécution du bloc.

On peut aussi ordonner les demandes et accès aux ressources mais ça ne fonctionne pas toujours et rend la gestion difficile dans le cas d'un grand nombre de ressources.

6.4 Questions

1. Un interblocage ralentit le processeur ?

   Faux, car le processeur est préemptif, donc il ne peut pas rentrer en état d'interblocage. Ce sont les processus et les ressources occupées qui sont bloqués mais le système continue de tourner.

2. Un interblocage **peut ne pas** être détecté par l'OS.

   Certains OS ne s'occupent pas de ça donc c'est possible. Par exemple Windows et Linux ignorent les deadlocks.

3. Une ressource préemptible et non partageable peut occasionner un interblocage ?

   Faux, car d'un côté une ressource unique ne peut pas causer d'interblocage. De l'autre côté, une ressource préemptible ne cause pas d'interblocage non plus.

7 Système de fichiers

De manière générale, un système de fichiers est l'organisation d'une partition d'un disque. Tout d'abord il faut bien comprendre comment fonctionne un disque.

Un disque est une mémoire secondaire (mémoire de masse), il contient des **secteurs** (formatage de bas niveau). Localiser un *byte* sur un disque revient à spécifier le numéro du secteur dans lequel le *byte* se trouve et la position du secteur. La mémoire du disque est segmentée en secteurs et chaque secteur porte un numéro unique. On peut donc référer à un secteur par son numéro.

Revenons sur ce qu'est un **secteur** : c'est une **unité physique** de stockage et d'échange. Un secteur = 512 bytes de données utiles plus quelques bytes d'information redondante permettant de valider son contenu. Le secteur ne doit pas forcément faire 512 bytes, il peut aussi faire 2048 bytes dans certains supports physiques par exemple.

Par exemple, dans un disque à plateaux, chaque secteur à son adresse unique dans le disque. Une **piste** ou *track* est la circonférence parcourue par la **tête de lecture** pendant une rotation de disque, elle est identifiée par le couple cylindre-tête (*cylinder*, *head*). Un cylindre est l'ensemble des pistes parcourues sans déplacement de tête

de lecture. Sur une piste se trouvent plusieurs **secteurs** numérotés de 1 à 63. Un changement de cylindre nécessite le changement de position des têtes de lecture (lent).

À l'époque, on utilisait l'adressage **CHS** (*Cylinder*, *Head*, *Sector*) des secteurs, qui est défini sur 3 *bytes*. Le **C** sur 10 bits (0-1023), le **H** sur 8 bits (0-255) et le **S** sur 6 bits (1-63). Le **Master Boot Record** est le secteur 0-0-1. Il contient des informations essentielles sur le système d'exploitation. On n'utilise plus l'adressage **CHS** car il n'est plus adapté aux tailles de disque actuelles. En effet, il ne peut contenir que 2^{24} secteurs, pour un secteur de 2^9 bytes, ça correspond à une taille d'environ 8 gigabytes.

Aujourd'hui, on utilise l'adresse **LBA** (*Logical Block Address*) : le disque est une suite de secteurs numérotés depuis 0 et le **MBR** est le premier bloc (LBA 0).

De plus, un disque est subdivisé en **partitions** comme `C:` ou `D:` sous Windows et `/dev/sda1` sous Linux. On peut les créer grâce à des commandes ou des logiciels (**fdisk**, **parted**). Chaque partition peut être organisée selon un format différent (système de fichiers) grâce à l'opération de **formatage**.

Chaque partition est donc organisée en **système de fichiers** comme `FAT`, `NTFS`, `EXT`, etc. Attention, **chaque partition** à son propre secteur 0. Il y a donc une numérotation générale pour le disque et une numérotation par partition.

7.1 Système de fichiers

Un système de fichiers est une organisation de partition qui permet : stocker de grandes quantités d'information de manière **permanente**, de **nommer** l'information (fichiers) et de **partager** l'information. Ils permettent aussi de **localiser** l'information grâce à leur chemin et non pas via le numéro de secteur.

Le système de fichiers met en œuvre le code des appels systèmes comme `open`, `read`, `write`. Son rôle est de définir l'implantation des fichiers et des répertoires, d'**allouer de l'espace** aux fichiers et aux **répertoires** et de permettre leur localisation. Il gère aussi l'espace libre, il permet de définir des droits, etc.

En clair, un système de fichiers permet de stocker et retrouver les données de nos fichiers. Aussi, ils ont une structure hiérarchique en arbre où un répertoire est une nœud. Les répertoires ont eux-mêmes des données qui permettent de localiser ou décrire les fichiers qu'ils contiennent.

Un système de fichiers comprend aussi des **méta-données**, ce sont des données qui donnent de l'information sur d'autres données. Par exemple la taille du fichier, la date de création, la taille des secteurs utilisées, etc. On distingue donc :

données information contenue par un fichier ou un répertoire.

méta-données de fichier ou répertoire information à propos d'un fichier ou d'un répertoire (taille, droits, date de création,...).

méta-données du système de fichiers information à propos du système en tant que tel. Par exemple la taille des secteurs, la localisation de la racine, la taille des blocs, les blocs libres, etc.

Pour allouer les secteurs des fichiers et des répertoires, il existe deux approches : allocation contiguë ou par blocs. Les deux stratégies sont décrites aux points suivants.

7.2 Allocation contiguë ou par blocs

7.2.1 Allocation contiguë

En allocation contiguë, les *bytes* de données d'un fichier sont **consécutifs** sur la partition. Tout fichier démarre à une frontière de secteur et occupe les secteurs voisins. Sa lecture nécessite un seul positionnement initial de la tête de lecture.

Pour localiser un *byte*, on peut imaginer l'exemple suivant : un fichier démarre au secteur 24 et contient 2060 bytes.

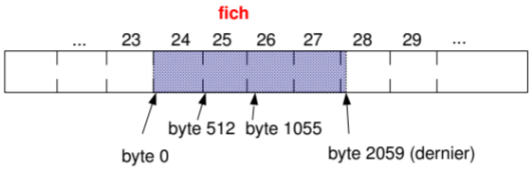
le *byte* n°1055 du fichier se trouve :

- dans le secteur : 26 = 24 + (1055 `DIV` 512)
- dans le byte : 31 = (1055 `MOD` 512)

le *byte* n°512 du fichier se trouve :

- dans le secteur : 25 = 24 + (512 `DIV` 512)
- dans le byte : 0 = (512 `MOD` 512)

On peut voir ces calculs illustrés par l'image ci-dessous :



En modifiant le système de fichiers, si on supprime des fichiers ou qu'on augmente la taille d'un fichier, on perd de l'espace disque. C'est de la **fragmentation externe**. De ce fait, il n'est plus possible de créer de nouveaux fichiers à moins de déplacer d'autres fichiers et les déplacements sont coûteux en écritures.

De ce fait, l'allocation contiguë est rapide en **lecture** mais très lente en réécriture dû à la fragmentation externe. Cette allocation convient aux systèmes de fichiers en **lecture** uniquement.

7.2.2 Allocation par blocs

Dans le cas de l'allocation par blocs, fichiers et partitions sont découpés en blocs de **taille fixe**. Un bloc correspond à un **nombre entier de secteurs** (1, 2, 4, 8,...). Un bloc est une **unité d'allocation** et une **unité d'accès** logique.

Donc, un fichier occupe un nombre **entier** de blocs et l'espace alloué au fichier n'est pas nécessairement contigu. Il faut donc des **métadonnées supplémentaires** pour décrire le chaînage des blocs de chaque fichier/répertoire.

La découpe en blocs se fait de la manière suivante :

- Les blocs commencent à la frontière d'un secteur.
- Les blocs sont numérotés.
- Le numéro de bloc de la partition P permet de calculer le numéro de son premier secteur. Ce numéro est l'**adresse** du bloc P.
  - L'adresse de P = $P*$ nombre de secteurs par bloc.

Les métadonnées du système de fichiers occupent également de l'espace disque, souvent en dehors des blocs. De ce fait, le bloc 0 n'est généralement pas aligné avec le secteur 0.

Donc pour trouver l'adresse d'un bloc on a que :

$$adresse\ P = adresse\ du\ bloc\ 0 + P \times nbr\ secteurs\ par\ bloc \tag{1}$$

Si on a une partition divisée en blocs et que chaque bloc contient 4 secteurs, que le bloc 0 est aligné sur le secteur 8 de la partition. Alors :

- le bloc 0 commence au secteur : $8 + (0 \times 4) = 8$.
- le bloc 3 commence au secteur : $8 + (3 \times 4) = 20$.

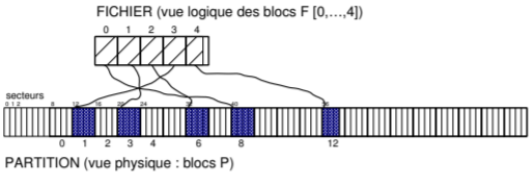
On doit garder une trace, une information sur ces données. C'est le rôle des métadonnées, elles conservent la **taille** du secteur, le **nombre de secteurs par bloc**, la taille du bloc, le **début du bloc 0**.

En allocation par blocs, on peut aussi utiliser la notation F et P qui permet d'identifier le numéro du bloc au sein du fichier et le numéro d'ordre au sein de la partition. Plus précisément :

F numéro d'ordre du bloc au sein du fichier (un fichier de 2000 bytes contient deux blocs [0, 1] de 1024 bytes). Il s'agit d'une vue logique.

P numéro d'ordre du bloc au sein de la partition (vue physique).

Voici un exemple avec $F = 4$ et $P = 12$:



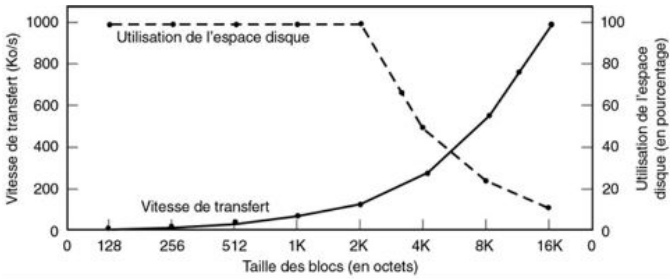
L'allocation par blocs permet un **ajout** et une **suppression** plus aisée des données car la taille des blocs est **fixe** et les fichiers **non contigus**. Cependant, cette facilité d'ajout et de suppression crée de la fragmentation de fichiers, qui s'éparpillent sur le disque.

On voit aussi apparaître une perte de place dans le dernier bloc du fichier qui n'est pas toujours rempli à 100% et ne peut être occupé que par un seul fichier. Il est en moyenne rempli à 50%. On parle de **fragmentation interne** pour désigner l'espace perdu dans le dernier bloc des fichiers. Ensuite, l'allocation par blocs à tendance à éparpiller les blocs d'un même fichier à travers le disque, ce qui rend la lecture plus lente. Cette dispersion porte le nom de

fragmentation des fichiers. La fragmentation interne induit une perte d'espace disque tandis que la fragmentation des fichiers ralentit la lecture, dans le cas de disques mécaniques à plateaux. Dans ce dernier cas, on doit réorganiser systématiquement les fichiers ou utiliser des outils de défragmentation.

1. Taille des blocs

   Les problèmes de fragmentation interne et de fichiers évoqués plus haut dépendent directement de la taille des blocs, qu'il faut définir. En effet, des blocs plus grands induisent plus de perte d'espace car un bloc ne peut être occupé que par un seul fichier, même partiellement. Tandis que des blocs plus petits rendent la lecture plus lente. Ce compromis est illustré par le graphique ci-dessous.



   Avec des blocs de 8 KiB, 50% de l'espace disque est perdu et avec des blocs de 64 KiB, 90% ! Dans la plupart des systèmes, on a des blocs de 2 ou 4 KiB. Pour des espaces de disques très grands, de l'ordre du *terabyte*, on peut recourir à des tailles de blocs plus grandes.

   En clair, l'allocation par blocs permet la souplesse nécessaire à la mise à jour de fichiers et à l'ajout ou suppression de fichiers mais est sujette à la **fragmentation interne** et **fragmentation des fichiers**.

7.3 Questions

1. En allocation de l'espace par blocs, une plus grande taille de bloc augmente la fragmentation externe.

   Faux, il n'y a pas de lien direct entre la taille des blocs et la fragmentation.

2. Quelle différence y a-t-il entre fragmentation (internelexterneldes fichiers) ?

   interne C'est une perte d'espace disque dans le dernier bloc car la taille du fichier est très rarement un multiple de la taille des blocs. De ce fait, le dernier bloc n'est jamais entièrement rempli et on perd de la place en mémoire car il est alloué exclusivement à un fichier.

   externe C'est une fragmentation de l'espace libre. C'est-à-dire que lors de la suppression de fichiers l'espace libre du disque est fragmenté et il n'est plus possible d'ajouter de nouveaux fichiers sans effectuer de déplacements, qui sont coûteux.

   des fichiers C'est une dissémination des blocs à travers le disque, la lecture devient donc plus lente car elle induit des déplacements de têtes fréquents pour les disques mécaniques.

3. Soit un fichier de 10360 *bytes* et une taille de bloc de 4KiB en allocation par blocs. Combien de blocs sont utilisés par ce fichier ? Quel pourcentage du dernier bloc est occupé ?

$$10360/(4*2^{10}) = 2.529 \tag{2}$$

   Donc on utilise 3 blocs et 52.9% du dernier bloc est occupé.

7.4 Répertoires

Un répertoire est un fichier particulier qui contient des métadonnées des fichiers comme les autorisations, le propriétaire du fichier, la taille du fichier, etc.

Pour localiser un fichier, on lit son répertoire parent et pour lire le répertoire en question, on lit aussi son parent. De cette manière, on remonte l'arborescence des fichiers et répertoires jusqu'à atteindre la *racine* qui doit avoir un emplacement **fixe connu** ou **calculable**. La plupart des systèmes de fichiers comme `EXT`, `FAT`, `NTFS` utilisent des **conventions**. En pratique, soit la racine est à une position calculable soit elle à une position fixe comme l'inode 2 pour `EXT`.

Par exemple, lire le fichier `/home/mba/test` demande de **localiser** et **lire** 3 répertoires.

1. `/`
2. `/home`
3. `/home/mba/`

C'est dans le dernier répertoire (3) qu'on trouve les informations qui permettent de localiser et finalement lire le fichier `test`.

7.4.1 Localisation

Dans le cadre de l'allocation par blocs, **début** et **longueur** ne suffisent plus à localiser un *byte N* si celui-ci est plus loin que le premier bloc du fichier ($F > 0$), comment établir la correspondance entre F et P dans ce cas ?

On aura besoin de métadonnées supplémentaires qui renseignent sur le chaînage des blocs au sein des fichiers. Ces métadonnées sont typiquement des métadonnées des fichiers. Ce n'est pas toujours le cas cependant car dans le système `FAT`, le chaînage est une métadonnée du système de fichiers via la **table d'index**.

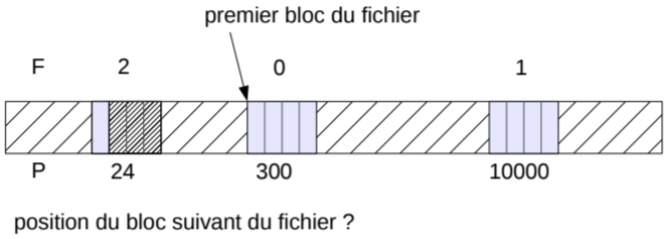
Lorsqu'on veut localiser le *byte N* d'un fichier, on peut s'y prendre de la manière suivante : tout fichier démarre à une frontière de bloc de la partition. Les blocs sont numérotés au sein du fichier depuis 0, le *byte N* d'un fichier se trouve dans le bloc *P* correspondant à son bloc n° **F**. Où :

$$F = N \ DIV \ \text{TailleBloc} \tag{3}$$

Mais à quel bloc **P** de la partition correspond **F** ? Le système de fichiers doit fournir les informations qui permettent de le localiser.

— F = n° du bloc dans le fichier
— P = n° du bloc de la partition

Par exemple dans le cas suivant, il n'est pas possible de définir simplement où se trouve le prochain bloc.



Suivant les systèmes de fichiers, il y a différentes approches. On peut utiliser des blocs **chaînés**, avoir un index **par fichier** (`EXT`, `NTFS`) ou bien une **table d'index** globale pour tous les blocs (`FAT`).

7.5 Appels système liés aux systèmes de fichiers

Un système d'exploitation mettra à disposition des applications, les services pour utiliser les systèmes de fichier.

Par exemple, on a les appels :

`open` Permet la **localisation** d'un fichier pour une session de lecture/d'écriture.

`close` Permet de clôturer la session de lecture ou d'écriture.

Comme la localisation est une opération coûteuse car elle demande plusieurs lectures de répertoires, on préférera travailler par sessions, avec une seule localisation pour plusieurs lectures/écritures.

Lorsqu'on utilise `open`, la localisation du fichier est mémorisée dans la Table des Descripteurs de Fichiers Ouverts (*TDFO*).

Lire un fichier se fait en général via des appels consécutifs à `read`. La position courante (prochain *byte* à lire ou écrire) est mémorisée dans la même table et est mise à jour par d'autres appels systèmes comme `read`, `write`, `lseek`.

L'appel système `close` quant à lui libère l'entrée de la table.

7.5.1 Appels systèmes pour les fichiers

Il existe plusieurs appels systèmes pour travailler sur les fichiers directement comme :

`read` Permet de lire des *bytes* (localiser les *bytes*)

`write` Permet d'écrire des *bytes* (localiser les *bytes*)

`lseek` Modifie la position de lecture ou d'écriture.

7.6 Questions

— L'appel système `open (/home/mba/test)` lit trois fichiers ?
Vrai, cet appel système lit trois **fichiers**, respectivement `/`, `/home/` et `/home/mba/`. Le fichier `test` est alors chargé en RAM mais pas lu.
— Localiser le répertoire `/` demande de lire son parent ?
Faux, le répertoire `/` est la racine, qui n'a pas de parent. Elle a une localisation prédéfinie ou calculable.

8 *File Allocation Table* (FAT)

8.1 Présentation et définition

La *FAT* est une famille de système de fichiers, nous nous concentrerons sur la `FAT32` qui date de 1996 et nous parlerons aussi de l'extension créée par Microsoft (`exFat`). C'est un système standardisé et extrêmement répandu.

8.1.1 Allocation

L'allocation en `FAT` se fait par blocs (*clusters*). La taille d'un *cluster* est définie au **formatage**. Typiquement, la taille d'un *cluster* vaut $2^9 B * 2^n$ où $n = 0..7$ et donc :

— 512B pour $n = 0$
— 1024B pour $n = 1$
— .
— .
— 64KiB pour $n = 7$

8.1.2 *Clusters*

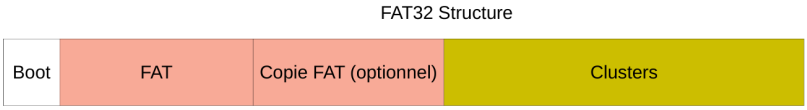
Voici une table des tailles de *clusters* en `FAT32`, avec une taille de **secteur** initiale de 512 *bytes*.

n	0	1	2	3	4	5	6	7
2^n = nb secteurs / cluster	1	2	4	8	16	32	64	128
taille du cluster	512B	1KiB	2KiB	4KiB	8KiB	16KiB	32KiB	64KiB

En `exFAT`, on élargit la taille des secteurs et la taille des *clusters*. On peut passer de 512 *bytes* à 4096 *bytes* pour des tailles de *clusters* allant jusqu'à 32MiB.

8.2 Structure

On peut se demander comment fonctionne le `FAT32` et comment sont gérées les données et métadonnées du système de fichiers, c'est-à-dire la structure d'une partition formatée en `FAT32`. C'est le but de cette section. Voici une vision schématique :



On remarque que le premier secteur de la partition, le secteur zéro est en général le secteur de *boot*. Cette zone de *boot* n'est pas forcément de 1 secteur, elle peut être plus grande et est renseignée dans ses métadonnées.

Ensuite, on trouve la table `FAT` en tant que telle, qui est l'index. Pour des questions d'intégrité des données, on peut aussi trouver une copie de la `FAT`.

Puis, il y a les *clusters*. Il faut noter que le premier *cluster* n'est pas 0 ou 1 mais bien 2, c'est une particularité du système de fichiers `FAT`. En général, on placera la racine au tout début des *clusters* et son emplacement est une métadonnée.

Auparavant, en `FAT12` et `FAT16`, la racine possédait un emplacement **fixe** entre la `FAT` (ou sa copie) et les *clusters*, avec une taille de maximum 256 descripteurs de fichier. En pratique, seulement 255 sont accessibles car 1 entrée est utilisée pour mémoriser le nom du volume.

À partir de `FAT32`, la racine devient un fichier comme un autre. Elle est placée dans les *clusters* et a une taille **dynamique**.

8.2.1 Secteur de *boot*

Le *boot sector* contient les métadonnées du système de fichiers. Ces métadonnées permettent de trouver l'adresse du *cluster* 2, c'est-à-dire le tout début des *clusters*, celle-ci dépend de la taille de la `FAT` et des *clusters*. Il y a donc un lien entre le nombre d'entrées de la `FAT`, celle des *clusters* et la partition.

Dans le secteur de *boot* on retrouve la taille des secteurs, le nombre de secteurs par *cluster*, etc.

8.2.2 Table `FAT`

Ce qui suit le secteur de *boot*, c'est la `FAT`, qui mémorise le chaînage des blocs de tous les fichiers.

La `FAT` est un index qui contient les numéros de *clusters*.

— `FAT16` numéros de *clusters* codés sur 16 **bits** (2 *bytes*)
— `FAT32` numéros de *clusters* codés sur 28 **bits**
— `exFAT` numéros de clusters codés sur 32 **bits**

Par cette structure, on peut localiser n'importe quel fichier du système de fichiers grâce au numéro de son premier bloc. Le numéro de premier bloc est indiqué dans le répertoire parent et on peut appliquer cette logique par liste chaînée jusqu'à retourner à la racine, pour n'importe quel fichier. C'est la `FAT` qui permet de retrouver la chaîne donnant accès à ces fichiers.

Un problème qui se pose, de part cette structure, est qu'il faut constamment vérifier dans la `FAT` l'endroit où on se trouve dans le chaînage des *clusters*. Dès qu'on arrive à la fin d'un *cluster* il faut lever la tête de lecture pour vérifier dans la `FAT` où se trouve le *cluster* suivant. Ce passage constant entre table d'index et *clusters* est coûteux et dû au fait que l'allocation en mémoire des espaces pour les fichiers n'est pas contiguë. Une solution est de charger la table d'index en RAM pour avoir de meilleures performances, ce qui devient problématique quand la `FAT` atteint une taille trop grande.

8.2.3 Chaînage des *clusters*

L'index contenu dans la table est en fait un tableau ayant autant d'entrées qu'il y a de **clusters** dans la partition. À chaque numéro de *cluster* est associé dans l'index, à l'indice correspondant au numéro de ce *cluster*, un état : libre, défectueux ou "chaîné".

Dans le cas d'un *cluster* chaîné, l'index contient le **numéro du cluster suivant** dans le chaînage ou une marque de fin si le *cluster* est le dernier du fichier. Aussi, un *cluster* appartient **au plus** à un fichier. Enfin, si on a la valeur 9 à l'indice 4 de l'index, ça veut dire que le cluster 4 est suivi par le cluster 9.

Grâce à cette structure, si un *cluster* indique son suivant au moyen de la table, chaque fichier peut être reconstitué entièrement à condition de connaître le numéro *P* du premier *cluster* du fichier sur disque. De plus, la taille du fichier ainsi que le premier *cluster* du fichier sont renseignés dans le répertoire parent.

À noter qu'on a pas forcément 2^{28} entrées (en `FAT32`) dans l'index. On a autant d'entrées que nécessaires et que de *clusters*. À chaque numéro de *cluster* doit correspondre son indice dans la table.

8.2.4 Racine

En `FAT32`, la racine est une chaîne de *clusters*, le **premier cluster est renseigné dans la zone boot** (souvent le cluster 2). Comme expliqué dans la section de secteur de boot.

8.2.5 *Clusters* libres

Un *cluster* libre est marqué comme tel, par la valeur 0, dans l'index. Attribuer un nouveau *cluster* à un fichier requiert de parcourir l'index.

On pourrait s'imaginer de chaîner tous les *clusters* libres comme s'ils appartenaient un seul même fichier, pour les trouver plus facilement. C'est ce qui est fait dans le système de fichiers `ext`.

En `exFAT`, la stratégie est similaire que celle utilisée en `ext2` : on utilise une table de bits, on parlera d'allocation **bitmap**. Cette approche, combinée avec une **pré-allocation** des *clusters*, permet de réduire le problème de fragmentation des fichiers.

8.2.6 Exemple de chaînage

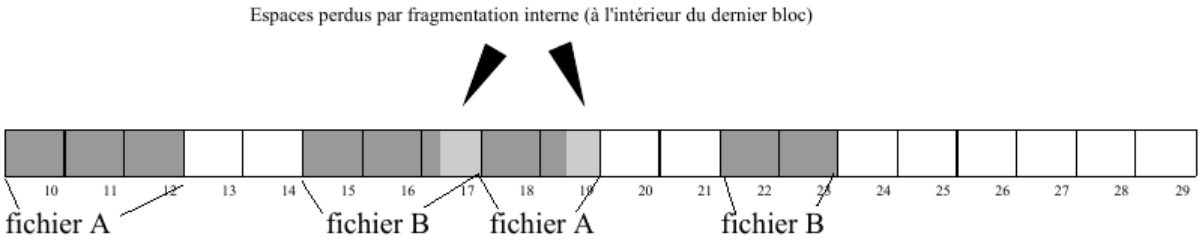


fig 3.2.1.2.1 :blocs sur le disque

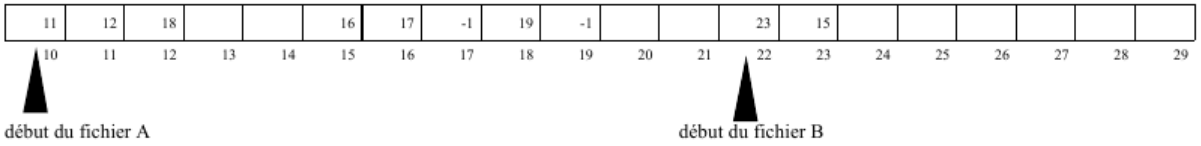


fig 3.2.1.2.2 : chainage des blocs par table d'index

Voici une explication issue du syllabus de SYS2, section 3.2.1.2.

Dans le dessin qui précède, nous avons représenté en haut les blocs du disque avec les données des fichiers en noir et en bas la portion de table d'index correspondante avec une entrée par bloc du disque. À partir de cette table nous pouvons reconstituer la séquence des blocs appartenant à un fichier à condition de connaître le numéro du premier bloc de celui-ci qui sera lu dans le répertoire parent. Dans l'exemple, le fichier A a pour premier bloc le bloc 10 et B le bloc 22.

Chaque bloc est chainé au suivant dans la table d'index dans l'entrée correspondant à sa position, le suivant du bloc 10 en position 10 etc. Chaque entrée de la table contient le numéro du bloc suivant qui n'est pas toujours le suivant sur le disque. Dans notre cas le bloc 10 est suivi par le 11. Le fichier A correspond donc à la séquence de blocs : 10 11 12 18 19, le fichier B à : 22 23 15 16 17. Les valeurs -1 en position 19 et 17 de la table d'index indiquent que ce bloc est le dernier d'un fichier.

Étant donnés les blocs d'un disque, localisés par leur numéro, et la table d'index dont l'emplacement sur le disque est connu, nous pouvons reconstruire l'entièreté d'un fichier en connaissant le numéro de bloc de début (10 pour A et 22 pour B). Nous avons déjà dit que le numéro du premier bloc d'un fichier se trouve dans la description du répertoire parent.

8.3 Table d'index

Voici les métadonnées nécessaires au chaînage des *clusters* en `FAT`.

La table d'index est métadonnée du système de fichiers.

La premier cluster d'un fichier il est renseigné dans le **répertoire parent**. C'est une métadonnée de fichier qui permet de retrouver les *clusters* suivant du fichier.

8.3.1 Valeurs réservées

En `FAT`, certaines valeurs sont réservées dans la table d'index, indiquant un état particulier.

0 *cluster* **disponible**

-1 **dernier** *cluster* du fichier (valeur sentinelle)

cluster **défectueux**

8.3.2 Localisation d'un *byte*

Pour localiser un *byte*, on a besoin de plusieurs métadonnées, notamment :

- Les métadonnées du système de fichiers :
  - Taille du secteur
  - Nombre de secteurs par cluster | taille du cluster
  - Début du cluster 2
  - Table d'index

— Les métadonnées du fichier :
  — Cluster associé au **premier bloc** du fichier
  — Longueur du fichier (et occupation du dernier bloc)

Ceci est la vue générale, maintenant on peut se demander comment localiser le byte N ? Pour ça, on doit traduire le byte N en une position dans un secteur, qu'on nommera le secteur x. Voici la manière de calculer la localisation d'un *byte*. Pour un rappel de la notation, voir l'allocation par blocs.

1. F : $F = N$ DIV TailleBloc
2. P obtenu en suivant le chaînage. P est le numéro du premier cluster du fichier.
3. **adresse** de P = début du cluster $2 + ((P-2) \times$ nbSecteursParCluster$)$
4. x = (adresse de P $+$ ((N MOD TailleCluster) DIV TailleCluster)
5. **position du byte N** dans $x \rightarrow N$ MOD TailleSecteur

Voici un exemple :

Si le *cluster* 2 est aligné sur le secteur 2048 et un *cluster* contient deux secteurs, soit 1KiB ou 1024B. Cherchons le *byte* $N = 3600$ du **fichier composé des blocs** (22-23-15-16-17).

1. $F =$ 3600B DIV 1KiB = bloc 3 du fichier
2. $P(F) =$ cluster 16, pour F = 3. (22(0)-23(1)-15(2)-16(3)-17(4))
3. adresse de P = 2048 + (16 - 2) × 2(secteur/bloc) = 2076
4. $x = 2076/($3600 MOD 1024B$)$ DIV 512B $= 2077$
5. byte N dans le secteur x = 3600B MOD 512B = 16B

Donc, le *byte* 3600 du fichier B est le *byte* 16 du secteur 2077.

À noter que les *clusters* `FAT` sont numérotés depuis le n°2, l'adresse du *cluster* 16 est donc :

$$P = 2048 + (16-2) \times 2\text{(secteur/bloc)} = 2076 \tag{4}$$

8.3.3 Exercice

Étant donnée la table d'index suivant (indices en bas), avec des secteurs de 512B et des *clusters* de 1KiB.

			8		4		10		-1	...
	2	3	4	5	6	7	8	9	10	11

Localisez le *byte* 2050 (n°secteur-position) dans le fichier qui démarre au bloc 6, sachant que le *cluster* 2 est aligné sur le secteur 40000.

1. F = 2050 DIV 1KiB = 2
2. P(F) = P(2) = 8 (6(0)-4(1)-8(2)-10(3))
3. adresse de P = 40000 + (P-2) × 2(secteur/bloc) = 40000 + 6*2 = 40012
4. x = 40012 + (2050B MOD 1024B) DIV 512B = 40012
5. byte N dans le secteur x = 2050B MOD 512B = 2B

Le byte 2050 du fichier est le byte 2 du secteur 40012.

8.4 Répertoires

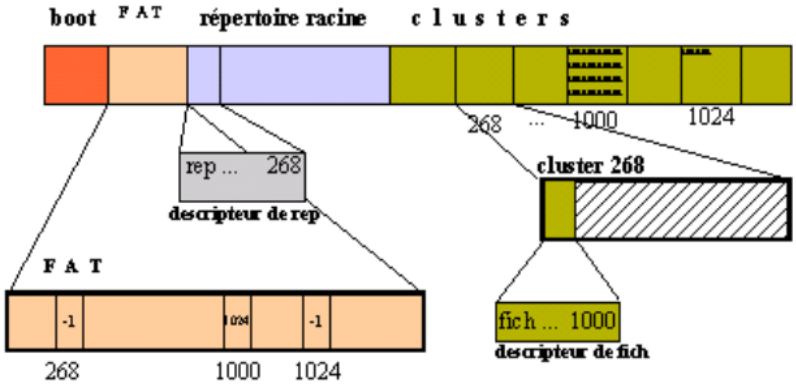
Tout fichier/répertoire est décrit dans son répertoire parent **sauf** la racine. Pour rappel, un répertoire est un fichier avec un type particulier et une certaine structure : c'est un fichier d'enregistrement (donc un répertoire).

— Ils sont composés de **descripteurs** de 32 *bytes* (un par fichier/sous-répertoire).
— Chaque **descripteur** contient les **métadonnées d'un fichier sous-répertoire**. Notamment **son nom** et son **premier cluster**.
— Chaque répertoire sauf la racine contient **deux** sous-répertoires (`.` et `..`), qui donne respectivement le répertoire courant et le répertoire parent. De ce fait, un répertoire n'est jamais vraiment vide.

Voici un exemple : soit une partition `FAT16` avec le fichier et le répertoire `/rep/fich`.

Le fichier contient 1030 caractères `'a'`. La taille des *clusters* est de 1KiB (1024B). Le répertoire `rep` est décrit dans le *cluster* 268. Le fichier `fich` occupe les *clusters* 1000 et 1024.

Voici un dessin qui représente la structure de ce système de fichiers :



Comme on est en `FAT16`, la racine est coincée entre la table `FAT` et les *clusters*. Dans la zone de la racine, on trouve **un seul** descripteur (32bits), qui correspond à ce que contient la racine, c'est-à-dire le répertoire `rep`, on trouve de l'information additionnelle comme le premier *cluster* de ce dossier (le *cluster* 268).

— En `FAT32` l'espace réservé à la racine serait remplacé par un chaînage de *clusters* démarrant au *cluster* 2.
— Les sous-répertoires `.` et `..` et du répertoire `rep` ont été oubliés dans le dessin précédent. Ils devraient précéder `fich` dans le *cluster* 268 qui décrit de répertoire.

Quand on utilise l'appel système `open`, le système doit localiser le fichier `/rep/fich`, pour ça il doit :

— Lire les enregistrements successifs du fichier racine `/` jusqu'au descripteur de `rep` et obtenir sa position.
— Lire les enregistrements successifs du fichier `rep` jusqu'au descripteur de `fich`.
— Ajouter la position de `fich` à l'entrée à la *TDFO* (Table des Descripteurs de Fichiers Ouverts).
C'est cette position enregistrée dans la *TDFO* une fois que la localisation est faite, qui va être utilisée par les autres appels systèmes comme `read`, `write`, `lseek`, etc.

Puis quand on utilise l'appel système `read` :

— Lire les données du fichier `fich` (*clusters* chaînés).

Le système d'exploitation doit savoir que le *cluster* 1024 ne contient que 6 `'a'`.

Le pendant de `open` est l'appel système `close` qui prend comme paramètre l'indice du fichier, et qui va (en général) effacer l'entrée de la *TDFO* correspondant au fichier et la rendre indisponible.

8.4.1 Descripteur

Ici-bas, voici un détail de l'entrée de répertoire en `FAT32`. On voit 10 bits réservés en gris foncé. En `FAT16` ils sont réservés pour l'utilisation future et rien n'y est stocké. Par contre, ils sont entièrement exploités en `FAT32` pour des améliorations.

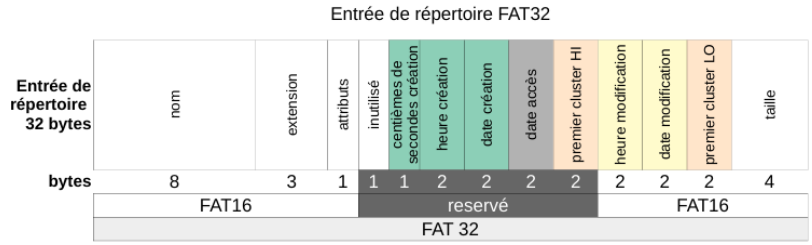
Ce qu'on y voit aussi, c'est que le nom est limité à 8 *bytes*, une extension de 3 *bytes* (`.txt`, `.py`, `.exe`, etc), un attribut sur 1 *byte* (droits, types, si c'est un répertoire ou pas).

Dans les champs à droit (pas dans la zone gris foncé), on retrouve deux champs pour indiquer la date l'heure. En `FAT16`, la date et l'heure sont des informations structurées et l'année est calculée par rapport à une date de référence. On a généralement 127 (2^7 - 1) années qu'on peut indiquer, par rapport au premier Janvier 1980 (Windows) ou 1970 (Unix).

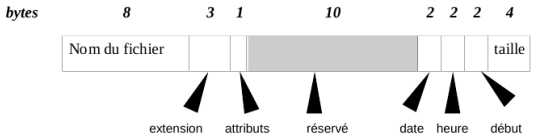
On a suffisamment de place pour décrire l'heure et la minute mais pas la seconde. Cette dernière est coupée en deux. Donc, la précision maximale qu'on peut obtenir pour un moment est limitée à deux secondes près et pas à la seconde près. Ceci est dû au fait qu'il n'y a que 2 *bytes* (16 bits donc) pour représenter l'heure. Avec 5 bits utilisés pour l'heure et 6 bits pour les minutes, il n'en reste que 5 bits pour les secondes, ce qui est insuffisant (60 secondes possibles mais seulement 32 valeurs sur 5 bits).

Dans la zone grisée qui est exclusive à `FAT32`, il y a plus de bits alloués à la description de temps, qui permet d'avoir une précision jusqu'au centième de seconde.

Puis, on trouve un descripteur pour le premier *cluster* du fichier ainsi que la taille du fichier. L'endroit où est localisé le premier *cluster* du fichier est codé sur 2 *bytes* en `FAT16` et 4 *bytes* en `FAT32` (2 *bytes* additionnels sont dans la zone réservée). En `exFAT`, cette limite est encore augmentée.



Donc, en FAT16, on a une image similaire avec une partie réservée dans la zone grisée.



En exFAT, la structure des entrées de répertoire est modifiée, comme en NTFS. On y étend l'utilisation d'entrées de répertoires spécifiques pour les métadonnées du système de fichiers.

8.4.2 Attributs

Voici les attributs qu'on retrouve dans la FAT, 1 *byte* est prévu avec 2 bits non utilisés. On a un attribut par bit, chacun valant donc 0 ou 1.

— ARCHIVE : Si le fichier est compressé ou pas.
— READ_ONLY : Si le fichier est en lecture seule ou pas.
— HIDDEN : Si le fichier est caché ou pas.
— SYSTEM : Si le fichier est un fichier système ou pas (non effaçable)
— DIRECTORY : Si le fichier est un répertoire ou pas.
— VOLUME_ID : Si le fichier est le nom d'un volume ou pas.
— 2b bits à 0, non utilisés.

On voit que ces attributs sont limités, on ne peut pas indiquer le propriétaire, si le fichier est exécutable, etc. Ceci est dû au fait que FAT était initialement prévu pour MS-DOS qui est un système d'exploitation mono-utilisateur et mono-tâche.

8.4.3 Noms

Le nom et l'extension s'écrivent sur 11 *bytes*, 8 pour le nom et 3 pour l'extension. On se rend compte que 8 *bytes* pour écrire un nom, c'est très peu.

Le **premier byte** du nom correspond à des valeurs particulières en FAT32, qui ont des significations :

— 0x00 : marque la **fin des entrées**
— 0xF5 : **entrée libre** (fichier supprimé). Suite au passage à UTF-8 plutôt que ASCII, la valeur 0x05 est utilisée.

En exFAT, on n'utilise plus le premier caractère (*byte*) avec une signification spécifique. Plutôt :

— Un nouveau champ **type** (sur un *byte*) indique, quand il vaut 0, la fin des entrées de répertoire.
— Un nouveau bit **inUseField** indique si une entrée est utilisée.

Cependant, depuis vFAT et FAT32, pour un nom plus long, on utilise **plusieurs entrées répertoire**. Chaque nouvelle entrée permet d'étendre le nom de 13 caractères. Cette entrée, qui doit être consécutive au nom, est marquée grâce à l'attribut ATTR_LONG_NAME ce qui permet de ne pas confondre l'entrée avec des nouveaux fichiers.

Attention, ATTR_LONG_NAME est **ignoré** par DOS, ce qui pose problème quand on travaille avec d'anciens outils.

En exFAT, on utilise au minimum trois entrées de répertoire par fichier et les fichiers dont le nom est supérieur à 15 caractères occupe des entrées supplémentaires, chacune permettant d'ajouter 15 caractères unicode au nom.

Aussi, en vFAT(16), la gestion des noms longs combinée avec le nombre réduit d'entrées du répertoire racine permettrait de saturer un système de fichiers vFAT16 avec +/- 30 fichiers de 1 *byte*. C'est bien les noms de fichiers fort long qui satureraient le système de fichier, même s'il n'y a effectivement pas de fichiers lourds.

8.4.4 Heure précision à deux secondes

Comme expliqué plus haut, l'heure et la date sont représentées sur 4 (2+2) *bytes*.

Le choix est fait d'utiliser une représentation structurée en année, mois, jour,... Mais si on représentait la date par un nombre de secondes sur 32 bits (4 *bytes*) à partir d'une date de référence (mentionnée dans la partie sur les descripteurs), on pourrait représenter une période plus longue avec une précision à 1 seconde plutôt que 2 secondes. On voit donc que la gestion de l'espace utilisable est importante.

8.4.5 Précision de l'heure en création `FAT32` - `exFAT`

À partir de `FAT32`, un *byte* supplémentaire est ajouté pour obtenir une précision au centième de seconde.

8.4.6 Premier *cluster*

Le numéro du **premier cluster** du fichier est codé sur 4 *bytes* en `FAT32`. Pour un fichier vide, le *cluster* vaut 0. En `FAT16`, on a que 2 *bytes* pour indiquer le numéro du premier *cluster* d'un fichier.

8.4.7 Taille fichier

En `FAT`, la taille d'un fichier est enregistrée sur 4 *bytes*. Ce qui correspond à 2^{32} bits et donc 4GiB. De ce fait, la taille maximale d'un fichier sur une partition `FAT` est de 4GiB.

En `exFAT` par contre, 8 *bytes* sont utilisés, pour une taille théorique de 16PiB. Cette taille dépasse la taille maximum de la partition `exFAT` pour des *clusters* de 32MiB.

Enfin, les entrées de type répertoire n'ont pas de taille associée, il faut suivre la chaîne pour obtenir le dernier *cluster* du répertoire.

8.4.8 Questions

— Que doit faire le système quand la taille d'un fichier augmente et que le dernier *cluster* du fichier est plein ?

D'abord il faut identifier un *cluster* vide *v*, on le trouve grâce au fait que `FAT` réserve la valeur 0 pour les *clusters* vides. Donc, on cherche un 0 dans la table d'index (dans son contenu, pas dans ses indices). Ensuite, là où le fichier s'arrêtait précédemment, indiqué par la valeur sentinelle -1, on remplace le -1 par l'indice du *cluster* libre *v*. Maintenant que ce *cluster v* est occupé, on lui donne la valeur -1 pour dénoter la fin du fichier. Plus précisément, à l'indice du *cluster v*, on met la valeur -1.

Enfin, le noyau doit mettre les métadonnées à jour dans le répertoire parent.

— Que doit faire le système pour créer un nouveau fichier vide ?

On doit rajouter une entrée dans le répertoire parent.

— Que doit faire le système pour lire un fichier entièrement ?

Il faut suivre le chaînage de tous les blocs, à partir du premier bloc, renseigné dans le parent. On s'arrête à la taille du fichier.

— Pourquoi le premier *cluster* d'un nouveau répertoire ne vaut pas 0 (n'est pas vide) ?

Parce qu'il contient les entrées `.` et `...`

— Que doit faire le système pour créer un nouveau répertoire ?

Il faut ajouter les deux entrées `.` et `...` Puis, comme pour un fichier classique, il faut rajouter une entrée dans le répertoire parent.

— Les *clusters* d'une `FAT` on tous la même taille ?

Les *clusters* d'une `FAT` on tous la même taille. La taille d'un *cluster* est définie au formatage. Par contre, la taille des *clusters* peut varier entre deux partitions avec des paramètres de formatage différents.

— En `FAT`, il y a une perte d'espace disque par fragmentation interne ?

Oui, car on utilise l'allocation par blocs.

— Un répertoire quelconque en `FAT` tient sur maximum un *cluster* ?

Faux, un répertoire peut être chaîné comme tout autre fichier et donc occuper plusieurs *clusters*.

— La *File Allocation Table* sert à trouver le premier *cluster* d'un fichier ?

Faux, le premier *cluster* n'est pas renseigné par la table, il se trouve dans le parent.

8.4.9 Grands répertoires

Localiser un fichier dans un répertoire demande de réaliser des comparaisons. Pour des répertoires de N entrées, il faut en moyenne exécuter $\frac{N}{2}$ comparaisons. Le système de fichiers `FAT` ne prévoit pas d'améliorations de performance sur ce point.

Quand on veut créer un nouveau fichier dans un répertoire, il faut vérifier que le nom du fichier n'existe pas déjà et donc on doit faire un parcours complet du répertoire.

D'autres systèmes de fichiers proposent une résolution à ce problème :

— `ext2` : répertoires groupés par un cylindre
— `exFAT` : Limitation de la taille d'un répertoire à 512MiB et l'utilisation d'un **hash-code** sur deux *bytes* dans les descripteurs de fichiers. On ne compare alors que ces deux *bytes* plutôt que le nom du fichier en entier.
— `NTFS` : utilisation d'un index sur le nom, pour un accès plus direct.

8.5 Intégrité

8.5.1 Incohérences

Des états incohérents d'un système de fichier peuvent survenir par exemple suite à une panne de courant. Notamment si celles-ci survient pendant un état instable du système. Par exemple lors d'une modification, un ajout, une suppression d'un *cluster* de fichier.

8.5.2 Ajout

L'ajout d'un *cluster* demande deux mises à jour :

— Le *cluster* libre pointe vers le *cluster* suivant du fichier ou la fin du fichier.
— Le *cluster* qui précède le nouveau, pointe vers le nouveau *cluster*.

8.5.3 Suppression

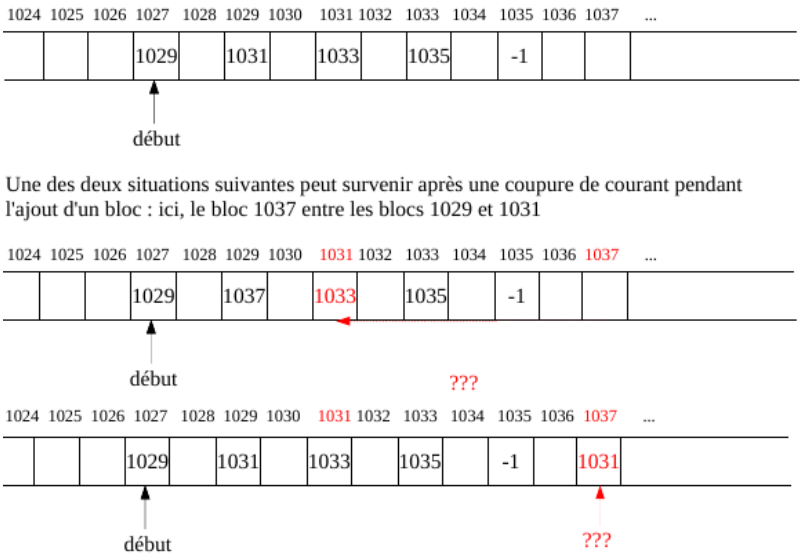
La logique est la même qu'au point précédent :

— Le *cluster* qui précède celui à supprimer pointe vers le *cluster* suivant.
— Le *cluster* à supprimer est marqué libre.

Dans ce cas, comme dans le cas de l'ajout, si une panne de courant survient entre les deux mises à jour, la `FAT` sur disque reste dans un état incohérent.

8.5.4 Exemple

Voici un cas d'incohérence suite à un ajout : on essaie de mettre à jour le chaînage après ajout mais une coupure de courant survient. Dans le cas ici, le bloc 1037 est chaîné dans le fichier mais il est libre, le lien vers 1031 est perdu ! Le système peut tenter de recomposer les chaînes avec des commandes de réparation car il existe une chaîne qui n'a pas de nom.



On peut réparer les erreurs qui surviennent dans certains cas avec une commande comme `chkdsk`. En suivant les étapes suivantes :

1. Parcourir la FAT à la recherche des *clusters* chaînés : ceux qui appartiennent à un fichier.
2. Chercher les *clusters* libres : ceux qui ont un 0 en FAT.

En théorie, un *cluster* normal doit soit être libre soit être chaîné. La comparaison entre les deux étapes décrites au-dessus permet de repérer des **chaînes perdues** de *clusters*.

8.5.5 Copie de la FAT

Si un *cluster* est défectueux dans un fichier, cela entraîne une perte d'une partie des données. Si un *cluster* est défectueux dans la FAT, le système de fichiers risque d'être **compromis**.

De ce fait, une **copie** de la FAT peut être maintenue pour garantir une plus grande fiabilité.

Donc, un *cluster* défectueux au niveau d'un fichier utilisateur est moins grave qu'au niveau de la FAT.

8.6 Table d'index - Tailles

8.6.1 Nombre de *clusters* maximum

L'entrée de l'index est un nombre entier (sur 2B ou 4B), le plus grand numéro de *cluster* dépend de la taille de ce nombre entier.

Sur 16 bits (2B) on peut représenter les valeurs allant de 0 à 2^{16} -1. Il y a donc au maximum 2^{16} *clusters* dans une FAT16.

Chaque type de FAT aura un nombre de *clusters* maximum différent.

8.6.2 Taille partition

En poussant le raisonnement plus loin, on peut définir la taille maximum d'un système de fichiers FAT avec :

$$\text{Taille Maximum} = \text{Nb max clusters} \times \text{Taille max cluster} \tag{5}$$

Ce qui donne pour FAT16 et FAT32 : en multipliant dans la taille maximum, le nombre de *clusters* par la taille maximal qu'un *cluster* peut atteindre. Ce sont des tailles théoriques, en pratique on conseille de ne pas dépasser une taille total de 32GiB en FAT32 et de 512TiB en exFAT. On utilise pas de très grands *clusters* car on perd de la place à cause de la fragmentation interne.

Nom	bits index	nb clusters	max taille
FAT16	16	2^{16}	4GiB ($2^{16} * 2^{16}$ B)
FAT32	28	2^{28}	16TiB ($2^{28} * 2^{16}$ B)
exFAT	32	2^{32}	128 PiB ($2^{32} * 2^{25}$ B)

8.6.3 Performance

Un constat qu'on a en utilisant les systèmes de fichiers FAT est que les *clusters* d'un même fichier sont éparpillés et que chaque accès à un nouveau bloc demande un **double accès disque** : table d'index et bloc de données.

Pour améliorer les performances, la table d'index est **chargée en RAM**. Donc, sa grande taille cause un ralentissement du système en démarrage !

8.6.4 Taille de l'index

À taille de partition égale, un plus petit *cluster* donne une plus grande table d'index.

On peut se poser la question de la taille de l'index d'une FAT donnée.

— Taille d'index FAT16 = nombre d'entrées index * 2B
— Taille d'index FAT32 = nombre d'entrées index * 4B
— Nombre d'entrées index = taille partition / taille clusters

Attention, ce calcul ne tient pas compte de la présence sur la partition de l'index même, il s'agit donc d'une approximation.

La taille de la table d'index pour une partition FAT32 de 500GiB avec des *clusters* de taille 4KiB est de 500MiB ! Avec des *clusters* de 16KiB on réduit l'index à 125MiB.

Une partition FAT32 de 500 GiB utilisant des clusters de 4KiB a besoin de 500GiB/4KiB entrées de table pour décrire chacun de ses *clusters*.

En FAT32, les entrées de table font 4B donc la taille de la table d'index peut être calculée par : 500GiB/4KiB x 4B = 500MiB.

8.6.5 Tailles maximum

Une table d'index peut devenir très grosse :

— La taille maximum d'un table `FAT16` est de 128KiB
— La taille maximum d'une table `FAT32` est 1GiB.

En `FAT16` la taille des entrées de l'index est 16 bits - 2B.

En `FAT32` la taille des entrées de l'index est 28 bits - 4B.

La plus grande table d'index est celle qui a le nombre maximum d'entrées.

Cela se vérifie pour le plus grand nombre de *clusters* en `FAT16` = 2^{16} *clusters*.

Cela se vérifie pour le plus grand nombre de *clusters* en `FAT32` = 2^{28} *clusters*.

Plus grand index de la `FAT16` $\rightarrow$ 2^{16} x 2B = 2^{17} B = 128 KiB

Plus grand index de la `FAT32` $\rightarrow$ 2^{28} x 4B = 2^{30} B = 1 GiB

En `exFAT`, on utilise 32 bits pour l'index.

8.6.6 Choisir le type de `FAT`

Il faut à la fois faire attention à la taille de la table d'index, chargée en RAM au démarrage et à la fois à la taille des *clusters*. Des *clusters* plus grands permettent d'éviter de la fragmentation de fichiers mais causent de la fragmentation interne. L'effet est inverse pour des petits *clusters*.

8.6.7 Critique

La table d'index `FAT32` qui est **chargée en RAM** au démarrage du système peut être grande ! Ce problème cause un ralentissement du démarrage du système.

C'est pour cette raison qu'on décourage de créer des partitions `FAT32` de taille supérieure à 32GiB même si la taille théorique est plus grande.

`exFAT` permet de plus grandes tailles de *clusters* et est conçu pour des fichiers de plus grande taille en essayant de mitiger la taille de la table d'index.

8.6.8 Exercice

— Calculer la taille de la **table** `FAT` avec *clusters* de 4KiB pour 32GiB en `FAT32` (28 bits). (N.B : les entrées de la table font 4B en `FAT32`).

$$\frac{32 * 2^{30}\text{B}}{4 * 2^{10}\text{B}} \times 4\text{B} = 32\text{MiB} \tag{6}$$

Si on multipliait la taille des *clusters* par 4, la table serait 4 fois plus petite.

— Calculer la taille minimum de **clusters** pour 2TiB en `FAT32` et la taille de la **table** `FAT` dans ce cas.

La taille maximale de la table d'index en `FAT32` est de 1GiB car elle vaut 2^{28} *clusters* fois les entrées de 4B donc 2^{30} B, soit 1GiB.

De ce fait, la taille des clusters est donnée par :

2^{28} = taille partition / taille clusters

taille clusters = 2TiB / 2^{28} = 2*2^{40} B / 2^{28} = 2^{13} B = 8KiB

Attention, il y a maximum 2^7 secteurs dans un *cluster*, pour un secteur de taille 2^9 B. De ce fait, la taille maximale d'un cluster est $2^7 \times 2^9$ B = 2^{16} B. Dans ce cas-ci, c'est bien une taille valide de *cluster*. Les limites sont expliquées dans la section sur les clusters.

Dans ce cas, on a pris tous les *clusters* possibles et on a donc une table de 1GiB.

— Pour une partition de 8GiB, quelle est la taille minimum de **clusters** en `FAT16`.

taille cluster = 8*2^{30} B / 2^{16} = 2^{17} B = 128KiB

Ce qui n'est pas une valeur possible. De plus on dépasse la taille théorique d'une partition en `FAT16` qui est de 2^{16} * 2^{16} B soit 4GiB. Avec 2^{16} B qui est la taille maximale d'un *cluster*, comme pour le `FAT32`.

Pour référence, la taille minimum de *cluster* est 2^9 B * 2^0 = 512B.

8.7 Conclusions

Le type de support et la taille attendue vont guider le choix des systèmes de fichiers à utiliser. Dans le cas de `FAT` et `exFAT`, c'est un bon choix pour les clés USB, disques externes, etc. Cependant il faut trouver un équilibre entre la taille de l'index et la fragmentation interne. Sans oublier que la table est chargée en RAM au démarrage, ce qui peut causer des ralentissements si elle est trop grande.

8.7.1 `FAT32`

En conclusion, le `FAT32` est un système de fichiers simple et reconnu par plusieurs systèmes d'exploitation ce qui le rend facile d'usage et il utilise plus efficacement l'espace que le `FAT16` car il permet de plus petits blocs à taille de partition égale.

Cependant, la taille des fichiers est limitée à 4GiB (`FAT12`, `FAT16`, `FAT32`) et la taille théorique des partitions à 16TiB bien qu'on recommande plutôt 32GiB.

Aussi, la performance n'est pas optimisée pour les grands répertoires et il existe de la fragmentation des fichiers puisque l'allocation est faite par blocs.

8.7.2 `exFAT`

Ce système de fichiers est plus performant que `FAT32` via de plus grands blocs et plus grands répertoires. Il dispose aussi d'une meilleure gestion de la fragmentation de fichiers par de plus gros *clusters*, l'utilisation d'une allocation *bitmap* et une pré-allocation des *clusters* aux fichiers.

Cependant, le problème de fragmentation interne est toujours présent, surtout à cause des plus grands *clusters* et à la pré-allocation. Aussi, `exFAT` perd de la place dans la représentation des répertoires (3 entrées minimum) alors qu'on est à minimum 1 entrée pour la représentation des répertoires en `FAT32`.

9 Crédits

Cours de Mme. Bastreghi, SYS2 2020-2021 à l'École Supérieure d'informatique