

Comment marche un FileSystem

On peut y accéder par block grâce à :

- **lba** (block address).
- Une taille de block (512B, 4KB, 2KB)
- Une queue pour stocker les opérations à faire

Quelles sont les opérations que l'on peut exécuter :

- Read
- Write
- Flush (Toutes les opérations dans la queue doivent être terminées)

Disques

Différents types (protocoles) de disque :

- **SCSI**
- Il y a un vrai protocole avec commandes. Ça sert aussi pour les imprimantes scanners...
- **Sata**
- Mieux que IDE, utilise un **bus en série**. Il utilise SCSI et reste limité par ça. Il a une seule queue de taille 32. L'unicité de la queue est chiant pour le multi processing.
- **IDE (ATA)**
- Historiquement utilisé, il n'y a pas de queue, utilise un **bus parallèle**. (C'est très lent). Comme SCSI c'est cool, IDE peut lire des commandes SCSI.
- **NVME**
- Protocole moderne (2010), plus de problème de seek. 64000 queue avec un bus en série. Tous les vendeurs de disques, OS... se sont mis d'accord pour avoir un seul driver. L'envoi des requêtes est efficace (une seule mémoire).

Les différents médiums de stockage :

- HD (Rotation)
- Flash (SSD)

Sur **NVME** on peut définir des **namespaces**. Un namespace est un set de partitions.

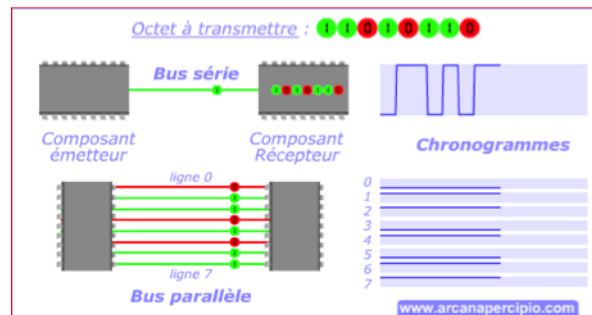


Figure 1: Bus série / parallèle

Différence entre bus parallèle et série

Un fichier

Un fichier a :

- Un nom
- Un format
- Un type
- Dossier
- Socket
- Name Pipe
- Symbolic link
- Char device
- Block device
- Attributs:
- date
- owner
- ACLs
- archive (NTFS only)
- hidden (NTFS only)

Type de fichier

Sur MS-DOS seul les fichiers `com`, `exe` et `bat` peuvent être exécutés.

Sur Mac OS un fichier garde l'information du logiciel qui a créé le fichier.

Operations sur les fichiers

- open(2)/creat(useless now)
- read(2)/write(2)
- lseek(2) (opération sur un fd et pas sur le fichier)
- fstat(2)
- other
- append modes
- truncate
- ...

Virtual File System

Le but est de créer une abstraction pour nos différents disques. (C'est notre tree sur linux par exemple)

On peut utiliser un arbre unique avec un dossier racine.

Windows utilise une lettre pour préciser le disque.

`mount` est un syscall qui permet de mapper un filesystem à un autre endroit.

NFS:

Network File System

Différents niveaux d'organisation

- Table de partition
- Mbr: master boot record
- Gpt: GUID partition table
- Partition
- Volumes
- Découpage logique (Lvm sur linux)
- Filesystems
- ext2/3/4, FAT, ntfs, ffs, ufs...

FAT: File Allocation Table, c'est une liste chaînée des blocks.

MBR

Master Boot Record, c'est le premier block du disque il contient la table de partition et le code pour charger l'OS et un magic number.

On a utiliser ca pendant 20 ans..

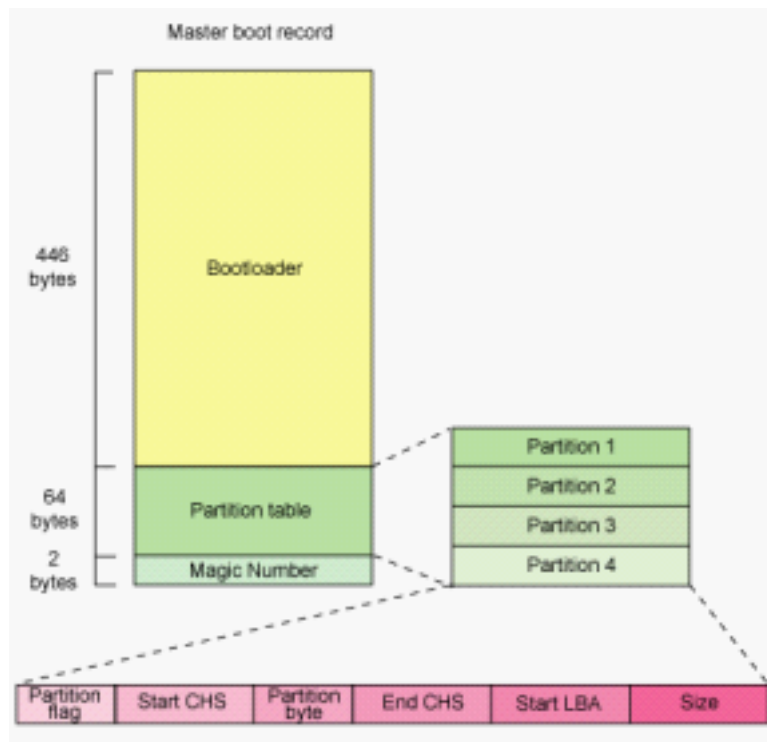


Figure 2: MBR Schema

C'est nul car:

- Il y a que 4 partition
- Le CHS n'a aucun sens aujourd'hui.
- Les sizes sont trop petites.

GUID

GUID : Identifiant de tres grande taille

GPT est une partie de **UEFI**.

On veut un nouveau format qui permet de régler ces problemes...

- Il y a deux tables de partitions pour faire de la redondance.
- Les partition sont de table variable

GUID Partition Table Scheme

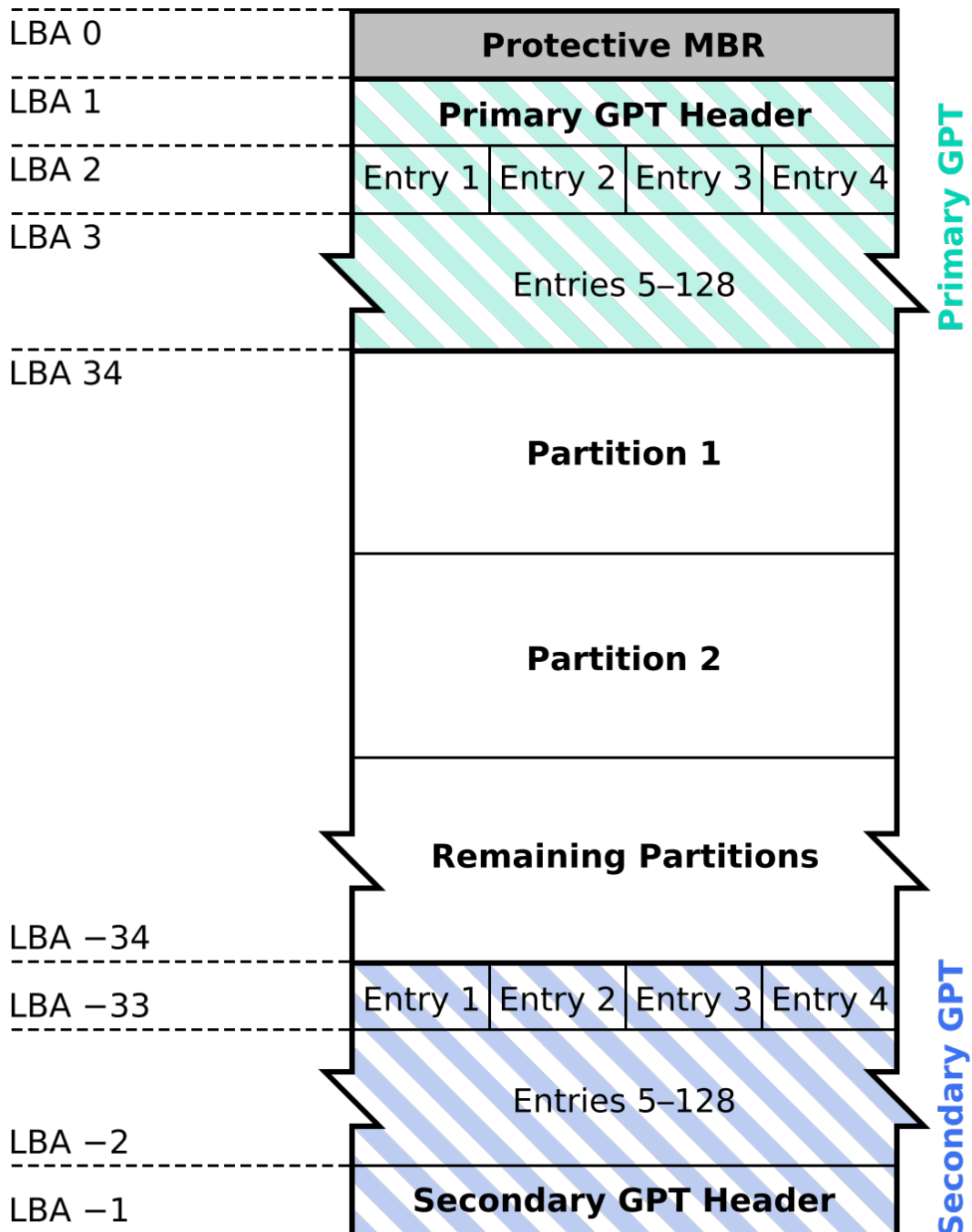


Figure 3: GUID partition table scheme

Superblock

C'est un header qui fait la taille d'un block il contient:

- L'**inode** de la racine (root) du filesystem
- La date de la dernière fois qu'il a été monté
- Indication du début du filesystem, il est impossible de retrouver ce qu'il y a dans le filesystem si le superblock est cassé

Il est donc critique, s'il n'est plus là on a perdu notre filesystem.

Il nous faut donc de la **redondance**, il y en a généralement un au début et un à la fin du filesystem.

Mais il peut ne pas être au début ex: **ISO** il est au 16^{ème} block. L'intérêt est de pouvoir stocker d'autres super block avant (pour le booter par exemple).

Let's discover

Dans un super block il y a par exemple:

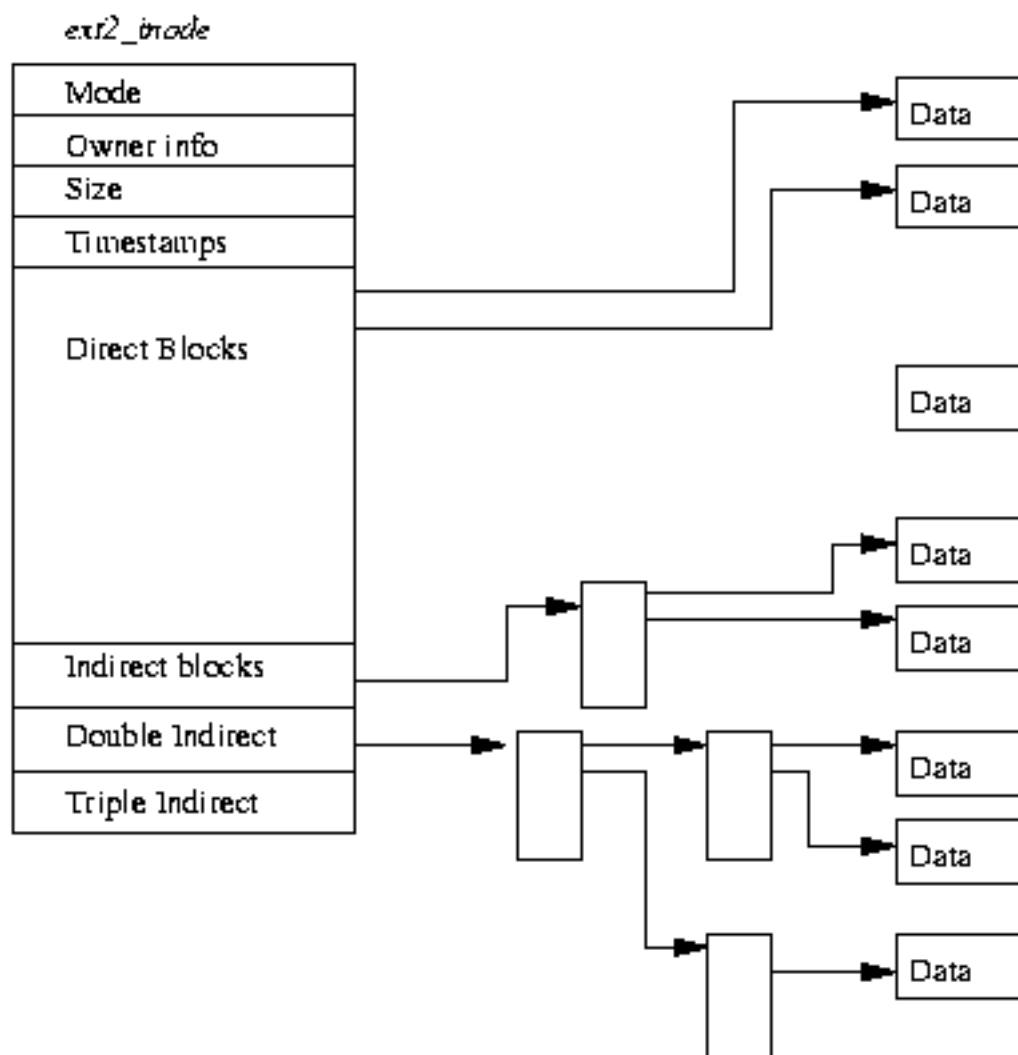
- Des informations fixes:
- Le nombre d'**inode**
- le premier **inode**
- Le nombre de block
- Des informations variables:
- Le nombre de block libre / utilisé:
- ...

Pour implémenter nos block (pouvoir les retrouver rapidement) on utilise un btree.

Dans un inode il y a par exemple :

- Le mode
- l'uid de l'utilisateur
- Date: modifié, créée ...
- Un tableau de block (les blocks utilisés par le fichier) composé de
- Pointeur sur data
- Pointeur sur indirect block (2 pointeurs sur data)
- Pointeur sur double indirect block (4 pointeurs sur data)
- Pointeur sur triple indirect block (8 pointeurs sur data)

Mapping private :

**Figure 4:** Ext2 Inode

L'espace mémoire n'est accessible seulement par le program ayant mappé l'espace mémoire

Mapping shared :

Il faut commit les changement fait sur l'espace mémoire pour tout le monde.

Linux Kernel IO Architecture

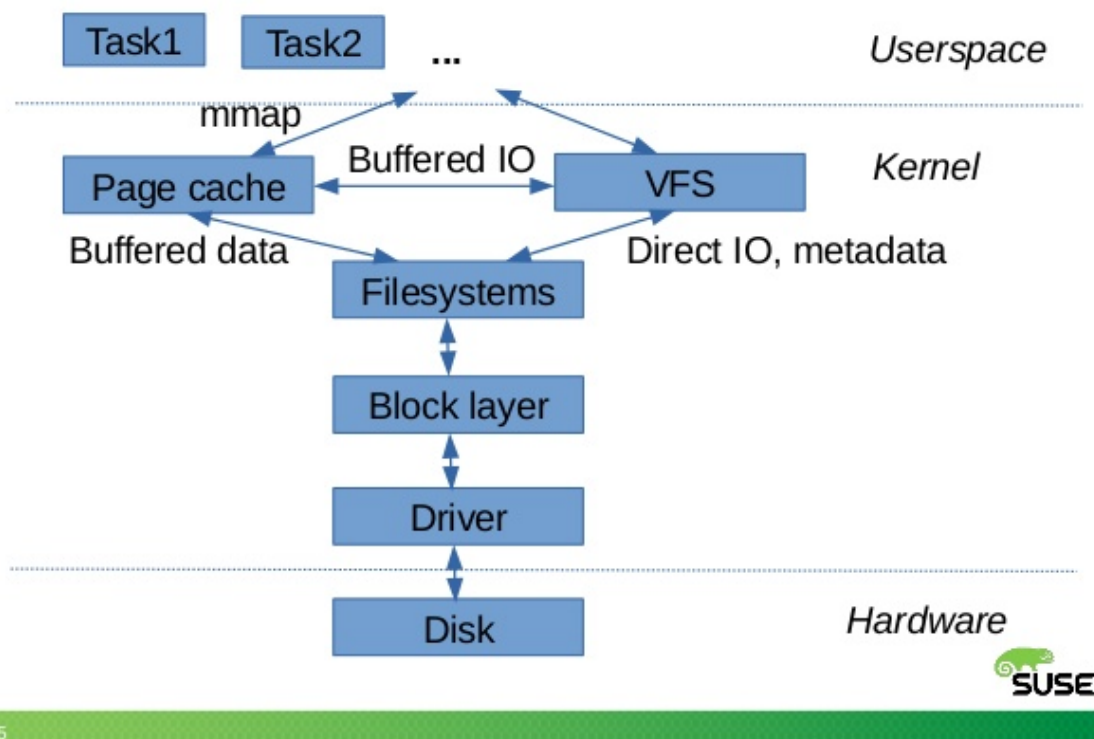


Figure 5: Linux Kernel IO Architecture

Block layer

Requêtes d'IO:

- Point de départ, longueur, lecture, écriture, ...
- La vie d'une requête possible :

IO (Input / Output) Scheduler

- **NOOP :**
 - Pas de scheduling, on prend les requêtes dans l'ordre.
 - Ca sert pas à grand chose.
- **Deadline :**
 - Préfère les lectures aux écritures (on avantage le userland).
 - Tri les requêtes de manière à réduire les déplacements (seek).
 - A pour but de dispatch les requêtes avec un temps de réponse inférieur à une **deadline**. Les tâches qui ont une deadline qui est dépassé vont passé prioritaire.
- **CFQ :**
 - Préfères les requêtes **synchrones**. Elles sont **bloquantes** donc on veut les finir plus rapidement.
 - Tri dans le but d'être juste avec les tâches.
 - Supporte les priorités d'IO.

Scheduling

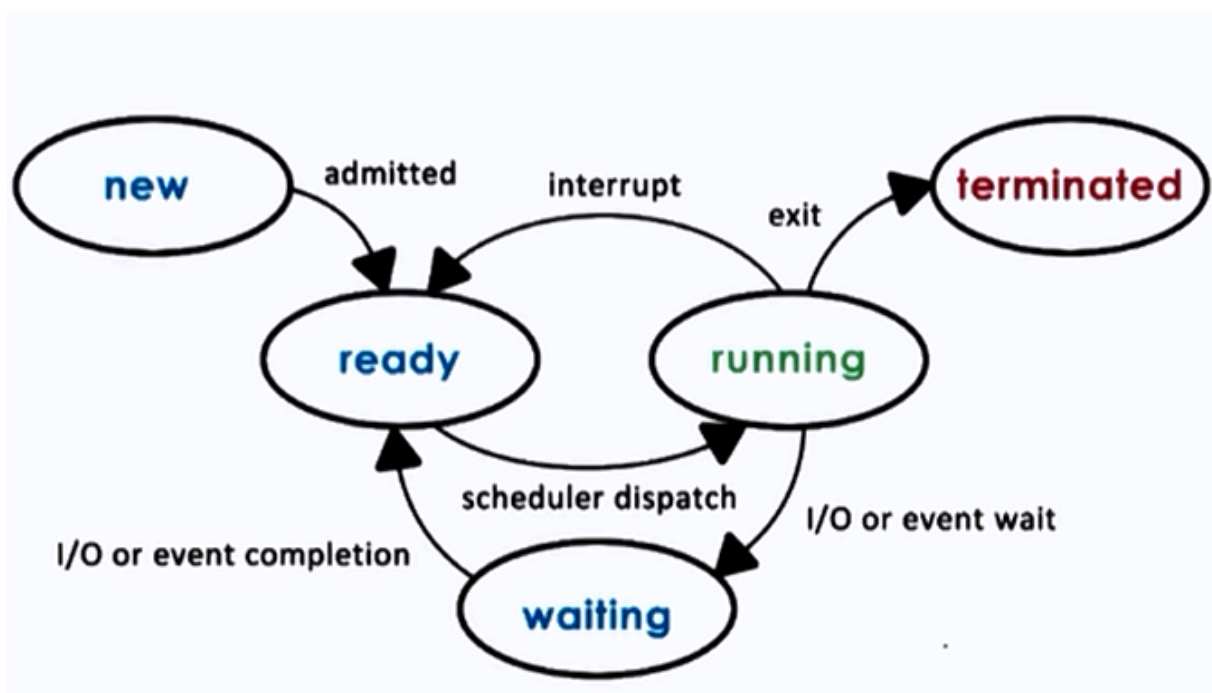


Figure 6: Process lifecycle

Qu'est qu'un algo de scheduling efficace ?

On a envie que notre algo de scheduling ne prenne pas trop de temps à choisir son process. Plus de temps de sélection alors, moins de temps pour l'exécution.

On va essayer d'être:

- **Équitable**
- **Rapide** $\Theta(1)$ (En vrai on sera en constant amorti, ne change pas en fonction du nombre de process)
- Ne jamais oublier un process dans un coin (No starvation) On veut éviter les **famines** (starvation) (un process qui ne tourne jamais).

Quelles métriques nous intéresse pour pouvoir mesurer la performance de nos algos ?

- Temps passé en ready (waiting time)
 - Exemple on a une tâche qui fait de la lecture genre un shell on veut avoir un process réactif.
- Temps pour passer de **new** à **running**
 - Temps qu'une nouvelle tâche va prendre pour s'exécuter (Response time), on cherchera à le minimiser
- Turn around time
 - Temps entre **new** de l'application et **terminated**.
- Throughput
 - Le nombre de tâche finies par unité de temps.
- Execution Time
 - Temps passé en running

Regarder en temps CPU, découper en parties, CPU en user (exécuter), en temps Kernel (scheduler, passer en waiting) Maximiser le CPU à passer en mode user.

Tâche doit être interactive (dans sa globalité elle fait beaucoup de syscalls), répondre rapidement.

Il y a d'autres applications qui vont très peu interagir. Ce sont des applications qui vont faire peu de syscalls.

On a donc globalement deux types de tâches.

:::info Une tâche peut être qualifiée interactive, va faire des calculs pendant un certain temps puis re-être interactive (Ex: compilateur) :::

Le goulot d'étranglement ralentit, empêche d'aller plus vite C'est souvent l'IO, mais ça peut être aussi le CPU / RAM pour un compilateur. Ça peut être le réseau aussi pour certaines applis.

- IO Bound (Syscall bloquants)

- Pour les tâches interactives.
- Minimiser le waiting time
- Minimier le quantum
- CPU Bound
 - Pour les tâches calculatoires.
 - L'augmentation du waiting time importe peu
 - L'augmentation du quantum importe peu

Quels sont les métriques à minimiser pour les tâches IO bound, CPU bound.

- Import les process IO bound
 - **Waiting time**
- Import les process CPU bound
 - **Throughput** (Débit plus rapide)

Le but du jeu est de minimiser l'exécution time.

TLB cache de pagination, tâche cpu bound lui faire tourner longtemps sur le CPU pour rentabiliser le temps de cache.

Le waiting time d'un CPU bound peut être élevé tourne moins souvent, IO bound (à check) plus souvent et moins longtemps comme ça c'est répartie

Comment le kernel sait si un process est IO ou CPU bound ?

Il suffit de compter les syscall et avoir des euristiques dessus.

On va ajouter des queues en fonction de leur niveaux de priorité pour l'état **ready**.

CPU bound basse priorité, IO bound une haute priorité.

- Running → Ready (quantum finit) : La priorité va baissé mettre dans une queue plus basse,
- Waiting → Ready : La priorité va monter.

Parenthèse pipeline et prédiction de branche

Pour chaque instruction on fait ces 5 étapes Fetch - Decode - Execute (add/sub/mov) - Memory (Read write memory) - Write back (si on read c'est bien de marquer la valeur)

Améliorer le temps de ces instruction on peut subdiviser chaque tâche :

- Bonne idée
 - Améliore la fréquence des micro processeur (Ex : intel pentium 35 étages)
- Mauvaise idée

Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

Figure 7: Pipelining

- Certaines instructions sont conditionnel on ne peut donc pas executer les instructions suivantes tant qu'elle n'est pas fini.
- Chaque jump va ralentir énormément (Trash les étages inutiles)