Stockage RAID et gestion de fichiers
Sujet V1.5

Vincent Dugat

Janvier 2019

NOTES IMPORTANTES :

Modalités du projet :

— La partie pilotage de projet et ses documents sera traitée dans le cours "Initiation à la gestion de projet" et gérée par l'intervenant de ce cours.
— Vous aurez un tuteur pour la partie développement et organisation du code.
— Le code est à programmer en langages C et Java selon les modalités décrite dans ce document.
— La page Moodle est commune à la partie Pilotage de projet et codage.
— Normalement à plusieurs on va plus loin. L'organisation de votre équipe et son efficacité en tant que telle, est un point important du projet.

Modalités de correction :

— Vous aurez une note de pilotage de projet et une de codage. La note de codage inclus la recette du code.
— La présence à la recette est obligatoire.
— Les deux notes comptent pour 50% chacune dans la note finale.
— Le total de tous les points est de 40. Ce score est à partager entre tous les membres d'une équipe (la note finale est sur 10 points qui seront ajoutés à la note de pilotage de projet elle-même sur 10 points). Le total sera sur 20 points.
— Le partage du score est par défaut équitable. Ce principe peut être modifié par le tuteur de développement ou à la demande de l'équipe (il est alors souhaitable qu'il y ait consensus).
— Il est obligatoire de commencer par le RAID 5, C et Java. Le reste peut être choisi librement.

1 Les technologies RAID : pour un stockage sûr, sécurisé et performant des données.

Le stockage de données pérennes sur disque est un besoin de plus en plus important au fur et à mesure que se développent les technologies numériques. Que ce soit pour des données professionnelles ou des données personnelles, les pannes matérielles inévitables des disques de stockage ont toujours un impact négatif sur la mémorisation d'informations importantes. Ces systèmes sont très employés pour les serveurs, le cloud computing et autres fermes d'ordinateurs et banques de données.

1.1 Introduction aux technologies RAID.

Les technologies RAID (pour *Redundant Array of Independent Disks*) permettent de répartir les données sur plusieurs disques durs afin d'améliorer la tolérance aux pannes, la sécurité des données, ou les performances d'accès. Ces technologies sont aussi capable de combiner ces différents critères pour offrir un espace de stockage sûr, résistant aux pannes et rapide d'accès. Les technologies RAID sont identifiée par un numéro indiquant les critères supportés (https://en.wikipedia.org/wiki/Standard_RAID_levels

Nous nous intéressons dans un premier temps à la technologie RAID 5 qui permet à la fois un fonctionnement résistant aux pannes et une grande performance lors de la lecture de fichiers.

1.2 Le RAID 5

Une introduction synthétique à ces technologies est disponible à l'url http://fr.wikipedia.org/wiki/RAID_(informatique).

L'objectif de cette partie est de développer en langage C une version simplifiée de la technologie RAID 5. La partie C est guidée. Une version Java de ce code sera à développer par la suite. D'autres types de systèmes RAID sont proposés aux équipes de trois et quatre personnes (voir en fin de documents).

Le RAID 5 nécessite au minimum 3 disques sur lesquels seront réparties les données. Un fichier de données est découpé en un ensemble de blocs de taille fixe de T octets, appelés *secteurs*. En fonction du nombre N de disques, les blocs sont regroupés par paquets de $N-1$ blocs. A partir de ces $N-1$ blocs, un bloc P appelé *bloc de parité*, de taille T est calculé en appliquant l'opération booléenne XOR (noté $\oplus$) sur les données du paquet de blocs.

On a ainsi la définition de P : $P = B_0 \oplus B_1 \oplus ... \oplus B_{N-1}$. L'opérateur XOR possède une propriété intéressante permettant d'utiliser ce bloc P pour reconstruire un bloc B_i manquant suite à une panne de disque. En effet, si $P = B_0 \oplus B_1 \oplus ... \oplus B_{N-1}$ alors $B_0 = P \oplus B_1 \oplus ... \oplus B_{N-1}$, $B_1 = B_0 \oplus P \oplus ... \oplus B_{N-1}$ et ainsi de suite.

Les $N-1$ blocs de données B_i et le bloc de parité P sont stockés dans un tableau S contenant N blocs de même taille qui constituent une bande. Cette bande est ensuite répartie sur les N disques en faisant en sorte que les blocs de parités de deux bandes successives ne se retrouvent pas sur le même disque.

Par exemple, étant donné un fichier de taille f telle que $kN < f <= (k+1)N$. Ce fichier sera décomposé en $k+1$ bandes, les k premières bandes contenant $N-1$ blocs de données et 1 bloc de parité, la dernière bande ayant $f\%N$ blocs de données, $N-1-(f\%N)$ blocs arbitraires et 1 bloc de parité. Sur l'illustration de la figure 1, le fichier est décomposé en 12 blocs, regroupés en 4 bandes, et stockés sur les disques de façon à ce que le bloc de parité de la première bande se trouve sur le dernier disque, celui de la seconde bande sur l'avant dernier disque, et ainsi de suite en gérant circulairement la position du bloc de parité.

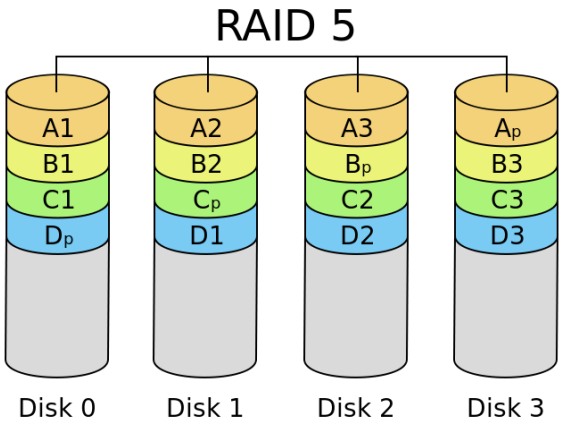

FIGURE 1 – Système RAID 5 avec $N = 4$ disques

1.3 Système de gestion de fichier (très) simplifié pour un stockage en RAID 5.

Afin de développer notre système de stockage RAID 5 simplifié, l'ensemble de N disques le composant sera représenté par un répertoire, chaque disque sera représenté par un fichier nommé d_i pour i allant de 0 à $N-1$ dans ce répertoire. Tous les disques seront de même taille.

Afin de pouvoir retrouver un fichier sur le RAID5, un tableau de structures contenant les noms des fichiers stockés sur le système et leurs propriétés (taille en octets du

fichier, indice du premier bloc sur le RAID5 et nombre de blocs) est stocké à partir du premier bloc du système. Ce tableau, assimilable à la table d'inodes d'un système UNIX, est de taille

fixe. On considère ici qu'un nom de fichier fait au plus de $FILENAME\_MAX\_SIZE$ caractères et que l'on peut avoir au maximum MAX_FILES fichiers.

Les fichiers auront une taille maximale de MAX_FILE_SIZE

Pour effectuer des tests, nous vous conseillons les valeurs suivantes de ces constantes. Veuillez vous assurer toutefois que si on change leur valeurs, votre système doit rester opérationnel.

```
#define BLOCK_SIZE 4
#define FILENAME_MAX_SIZE 32
#define INODE_TABLE_SIZE 10
#define MAX_FILE_SIZE (50*1024)
```

Dans notre système simplifié, on considère que le nombre de disque est variable. Cependant la mise au point se fera avec 4 disques.

On rappelle que tous les disques ont la même taille.

Le fichier source $cmd\_format.c$, fourni sur Moodle en annexe de ce sujet, permet de formater le système RAID5 à quatre disques. Pour formater votre système, l'exécutable demande comme paramètre le nom du répertoire existant dans lequel sera créé le système RAID5, le nombre de disque, et la taille $disk\_size$ des disques. Toutes les tailles sont données en octets. Si les disques d_i existent déjà dans ce répertoire, ils sont réinitialisés. S'ils n'existent pas dans le répertoire, des fichiers dont le contenu est mis à 0 sont créés à la bonne taille. L'opération de formatage du système doit être faite obligatoirement et une seule fois à la création du système. Tout formatage d'un système existant effacera ses données.

Afin de pouvoir réparer un système suite à la défaillance d'un disque, ce programme peut prendre aussi un paramètre supplémentaire qui contiendra le numéro du disque à formater. Seul ce disque sera alors ré-initialisé.

Exemple :

Syntaxe :
command nom_répertoire nb_disks taille_fichier (octets)
ou
command nom_répertoire nb_disks taille_fichier disk_id

./cmd format dir 4 500000 création des fichiers $d0$, $d1$, $d2$ et $d3$ dans le répertoire dir, chacun d'une taille de 500000 octets.
./cmd format dir 500000 2 ré-initialise (ou créé) le fichier $dir/d2$.

2 Travail à réaliser en langage C (10 points).

Le travail à réaliser est divisé et organisé en couches implémentant chacune des fonctionnalités utilisant celles de la couche inférieure et fournissant des services à la couche supérieure. Un fichier $raid\_defines.h$, fourni sur Moodle, donne les structures de données à utiliser.

Dans tout le sujet les fonctions de lecture/écriture sur les disques virtuels commencent par : write..., les fonctions d'affichage à l'écran commencent par print...

2.1 Couche 1 bas niveau : les blocs et les fonctions utilitaires

Cette partie a pour objectif l'écriture de fonctions permettant de gérer les blocs sur un système RAID 5 : écrire, lire, effacer et réparer un système RAID 5.

1. On considère que notre système RAID5 est représenté par la variable globale *virtual_disk_t r5Disk ;*. Avant de pouvoir l'utiliser, il est nécessaire de l'initialiser à partir du nom du répertoire contenant les disques virtuels formatés.

   Ecrire la fonction *init_disk_raid5* qui, à partir du nom du répertoire, initialise cette variable. Dans un premier temps, on n'initialisera pas la table d'inodes (couche 3). Lorsque notre système sera "éteint", il sera nécessaire de s'assurer de l'absence de risque de perte de données. Pour cela, écrire une fonction qui "éteint" notre système RAID5.

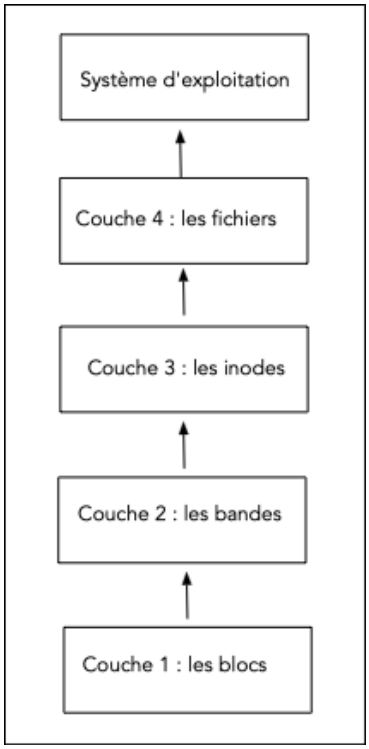

FIGURE 2 – Organisation en couches

2. Ecrire la fonction *compute_nblock* qui calcule le nombre de blocs nécessaires pour stocker un nombre n d'octets.
3. Écrire la fonction *write_block* qui écrit un bloc *block*, à la position *pos* sur le disque *disk_id* du système RAID 5.
4. Ecrire la fonction *read_block* qui lit un bloc *block* de données, à la position *pos* sur le disque *disk_id* du système RAID5. En cas d'échec de lecture, cette fonction doit renvoyer un code d'erreur que vous préciserez.
5. Ecrire la fonction *block_repair* qui, en cas d'erreur de lecture, reconstruit le bloc erroné.
   Remarque : Les fonctions C $fseek$, $fwrite$ et $fread$ semblent les plus indiquées pour ce travail.
6. Ecrire des fonctions d'affichage à l'écran des blocs de type *block_t* en hexadécimal pour vérifier.

2.2 Couche 2 : les bandes (de blocs)

1. Ecrire la fonction *compute_nstripe* qui calcule le nombre de bandes nécessaires pour stocker un nombre n de blocs.
2. Ecrire la fonction *compute_parity* qui, à partir d'un ensemble de *nblocks* blocs d'une bande calcule le bloc de parité selon la méthode spécifiée dans la section 1.2.
3. Ecrire la fonction compute *parity_index* qui à partir d'un numéro de bande calcule le numéro du disque sur lequel sera stocké le bloc de parité selon le principe indiqué sur la
   figure 1. Dans cette figure, les blocs de données sont référencés par X_i avec $X \in \{A, B, C, D\}$ et $0 < i < 4$ et les blocs de parité sont référencés par X_p avec $X \in \{A, B, C, D\}$.
4. Ecrire la fonction *write_stripe* qui écrit une bande de blocs et un bloc de parité, à la position *pos* sur le système RAID5.
5. Ecrire la fonction *write_chunk* qui, à partir d'un tableau de n octets $buffer$, l'écrit sur le système RAID5 à partir de la position $start_byte$ indiquant la position dans la listes complète des blocs de données sur le système. Les blocs de parité, non comptabilisés dans le nombre de blocs déjà présents dans le système ni dans le nombre de blocs couvrant le buffer, devra être écrit selon le principe illustré sur la figure 1.

6. Ecrire un programme principal de test nommé *cmd_test*1 permettant de tester votre fonction *write_chunk*. Ce programme devra écrire, dans le système RAID5, un buffer de 256 octets représentant toutes les valeurs possibles que peut prendre une variable de type *unsigned char*.
7. Ecrire la fonction *read_stripe* qui lit une bande de blocs et un bloc de parité, à la position *pos* sur le système RAID5. Cette fonction doit lire tous les blocs présents (données et parité) et renvoyer un code d'erreur, que vous préciserez, si un des blocs n'a pu être lu. Précisez ce que doit faire cette fonction si plusieurs blocs ne peuvent être lus.
8. Ecrire la fonction*read_chunk* qui lit, à partir de la position *start_byte*, un tableau buffer de n octets, en reconstruisant les blocks qui ne peuvent être lus. Cette fonction n'écrit dans le biffer que les informations, pas les blocs de parité.
9. Vérification : Ecrire un programme principal de test nommé *cmd_test*2 permettant de tester votre fonction *read_chunk*. Ce programme devra lire depuis notre système RAID5 le buffer de 256 octets que vous avez écrit précédemment, et devra afficher ces 256 octets sur le flux standard de sortie. On écrira le numéro de l'octet et la valeur lue sous forme hexadécimale.

2.3 Couche 3 : gestion du catalogue, du super bloc et de la table d'inodes

Les opérations de bas niveau programmées dans la partie précédente permette de stoker des données binaires sur le système RAID5. Afin de pouvoir structurer ces données sous forme de

fichiers, il est nécessaire de mettre en place un système de gestion adapté. Pour cela, une table d'inodes [1] de taille fixe $MAXFILES$, faisant l'association entre un nom de fichier et ses propriétés, comme présenté dans l'extrait de code 1, est stockée à la position $INODES\_START$ sur le système RAID5. Une entrée de cette table contient soit un nom de fichier et des données (non nulles) caractérisant sa taille, la position et le nombre de bloc utilisés par le fichier sur le système RAID5, soit une position égale à 0 si elle ne désigne pas de fichier existant dans le système RAID5.

Afin de faciliter l'accès à l'information on utilise un super bloc qui donne le type de raid, le nombre de blocs utilisés et le numéro du premier octet libre dans le système. La taille du super bloc est fixe (3 blocs + le bloc de parité = 4 blocs sur le RAID)

L'objectif de cette partie est de rajouter des fonctions permettant de gérer cette table d'inodes et le super bloc.

1. Ecrire la fonction *read_inodes_table* permettant de charger la table d'inodes depuis RAID5 et utilisez-la pour terminer l'initialisation de la question 1.
2. Ecrire la fonction *write_inodes_table* permettant d'écrire la table d'inodes sur RAID5.
3. Ecrire la fonction *delete_inode* qui, à partir d'un indice dans la table d'inodes supprime l'inode correspondant et compacte la table de façon à ce que, si n fichiers sont stockés sur le RAID5, les n premières entrées de la table d'inodes correspondent à ces fichiers.
4. Ecrire la fonction *get_unused_inode* qui retourne l'indice du premier inode disponible dans la table.
5. Ecrire la fonction *init_inode* qui initialise un inode à partir d'un nom de fichier, de sa taille et de sa position sur le système RAID5.
6. Ecrire un programme *cmd_dump_inode* qui servira pour les tests sur les fichiers. Ce programme prends en argument le nom du répertoire contenant les fichiers de disque. Après avoir lu la table.
7. Ecrire la fonction *write_super_block* qui écrit le super bloc au tout début du RAID.
8. Ecrire la fonction *read_super_block* qui lit le super bloc au tout début du RAID.
9. Ecrire une fonction qui met à jour le champs $first\_free\_byte$ du super bloc.

2.4 Couche 4 : Gestion des fichiers

Les fonctions des couches précédentes peuvent maintenant être utilisées pour gérer des fichiers. Un fichier sera décrit avec la structure :

1. Cette notion empruntée à UNIX est très simplifiée ici.

```
typedef struct file_s{
  uint size; // Size of file in bytes
  uchar data [MAX_FILE_SIZE] ; // only text files
} file_t ;
```

On ne va gérer que des fichiers texte.

1. Ecrire la fonction *writefile* prenant comme paramètres un nom de fichier (chaîne de caractères) et une variable de type $file\_t$ contenant le fichier à écrire sur le système RAID5. Si le nom de fichier n'est pas présent dans la table d'inodes, alors un nouvel inode est créé pour ce fichier et ajouté en fin de table. Le fichier est écrit sur le système RAID5 à la suite des fichiers déjà présents. Si le nom de fichier est présent dans la table d'inodes, alors c'est une mise à jour et deux cas sont à traiter :
   — le fichier a une taille inférieure ou égale à la taille du fichier déjà présent. Il suffit alors de mettre à jour les données et la table d'inodes (si il est plus petit il y aura un "trou" sur le disque).
   — le fichier a une taille supérieure à la taille du fichier déjà présent. Il faut alors supprimer l'inode correspondant puis ajouter le fichier en fin de disque (l'ancien fichier laisse un "trou" sur le disque).
2. Ecrire la fonction $read\_file$ prenant en paramètres un nom de fichier (chaine de caractères) et une variable de type $file\_t$ qui contiendra le fichier lu. Si le fichier n'est pas présent sur le système RAID5, cette variable n'est pas modifiée et la fonction renvoie 0. Si le fichier est présent sur le système RAID5 cette variable contient les données du
   chier lu et la fonction renvoie 1.
3. Ecrire la fonction $delete\_file$ prenant en paramètre un nom de fichier et qui supprime l'inode correspondant à ce fichier. Cette fonction retourne 1 en cas de suppression et 0 si le
   chier n'est pas présent sur le système RAID5.
4. Ecrire une fonction $load\_file\_from\_host$ qui prend en paramètre le nom d'un fichier de l'ordinateur (nommé *host*) et l'écrit sur le système RAID. Le nom du fichier sur le RAID sera le même que sur l'hôte.
5. Ecrire une fonction $store\_file\_to\_host$ qui prend en paramètre le nom d'un fichier du sytème RAID et l'écrit sur l'ordinateur hôte. Le nom du fichier sera aussi le même.

2.5 Couche 5 : le système d'exploitation

Pour utiliser toutes les fonctionnalités mises en place nous créons dans cette partie un système d'exploitation basique en ligne de commande.

1. Programmer un interprète de commande "Unix like" selon la boucle :
   (a) Lecture de la commande (avec ses paramètres)
   (b) Interprétation
   (c) Exécution
   (d) Affichage du résultat
2. Les commandes connues de l'interprètes sont :
   — ls [-l] : liste le contenu du catalogue. Un argument optionnel pour un affichage court ou long.
   — cat <nom de fichier> : affiche à l'écran le contenu d'un fichier,
   — rm <nom de fichier> : supprime un fichier du RAID,
   — create <nom de fichier> : crée un nouveau fichier sur le RAID,
   — edit <nom de fichier> : édite un fichier pour modifier son contenu,
   — load <nom de fichier> : copie le contenu d'un fichier du système "hôte" sur le RAID avec le même nom,
   — store <nom de fichier> : copie le contenu d'un fichier du système RAID sur "hôte" avec le même nom,
   — quit : sort de l'interprète de commande et du programme

Chaque commande devra la suite d'opérations nécessaires pour lister le catalogue, afficher, créer, supprimer, etc. un fichier. Après l'affichage du résultat ou d'un éventuel message d'erreur, le prompt sera affiché de nouveau. La commande "quit" permet de quitter l'interprète et le programme après la sauvegarde sur le RAID des donnée et la fermeture des fichiers.

La fonction *main* lancera cet interprète. Le main prendra en argument le répertoire contenant le RAID.

2.6 Simulation de panne : Diagnostic et réparation du système RAID 5

Une panne dans RAID5 est simulée par la suppression d'un fichier d_i dans le répertoire de travail. Si on supprime un tel fichier disque, les fonctionnalités de lecture doivent fonctionner sans incidents. Pour réparer l'incident la procédure est la suivante :

- Ecrire une fonction *repair_disk* qui prend en paramètre un numéro de disque et qui le répare pour que l'ensemble des disques de RAID5 soient cohérents.
- Ecrire le programme *cmd_repair* qui prend en paramètre un répertoire de travail et un numéro de disque. Ce programme répare le disque demandé en utilisant le fonction précédente.
- Ecrire un programme *dump_raid5* qui prend en paramètre le nom du répertoire et une taille en octets, et affiche les blocs du système RAID 5, bande par bande en hexadécimal en indiquant l'indice du bloc de parité.

2.7 Le problème de la fragmentation

Les simplifications que nous avons introduites pour notre système RAID 5 génèrent une forte fragmentation du système après chaque suppression de fichier. La question BONUS de ce projet consiste à écrire un programme de défragmentation permettant de compacter le stockage des fichiers en des bandes consécutives afin de ne pas perdre de place après avoir effacé un fichier.

2.8 Programme Java du RAID 5 (10 points)

Le programme Java consiste à implémenter le même cahier des charges que le programme C. Il s'agit donc d'implémenter les 5 couches su système avec les différences suivantes :

1. On ajoute une interface graphique d'interaction au niveau du système d'exploitation (couche 5). Les spécifications sont libres.
2. On délaissera le problème de la fragmentation, du diagnostique et réparation de panne.

2.9 Autres systèmes RAID

Pour les équipes de trois ou quatre personnes il va être nécessaire de gagner quelques points supplémentaires.

En vous aidant de votre expérience du RAID 5, et en réutilisant autant que possible les fonctionnalités déjà implémentées, programmer en Java ou en C au choix les systèmes suivants au choix également. https://en.wikipedia.org/wiki/Standard_RAID_levels

2.9.1 RAID 0 (4 points)

https://en.wikipedia.org/wiki/Standard_RAID_levels

2.9.2 RAID 1 (4 points)

https://en.wikipedia.org/wiki/Standard_RAID_levels

2.9.3 RAID 1+0 ou 0+1 (6 points)

https://en.wikipedia.org/wiki/Nested_RAID_levels

2.9.4 RAID 50 (10 points)

https://en.wikipedia.org/wiki/Nested_RAID_levels