Algorithmique et programmation

Projet de programmation
Stockage sécurisés RAID 5

2013

NOTES IMPORTANTES :

Modalités de remise du projet :
Vous devez déposer sous moodle une archive au format tar.gz (et uniquement à ce format là) nommée : prenom.nom-RAID5.tar.gz (remplacez prenom et nom par VOTRE prénom et VOTRE nom). Merci de ne pas mettre d'espaces ou d'accents dans le nom de votre archive. Cette archive devra contenir UNIQUEMENT :
- les fichiers sources de votre projet,
- un fichier *Makefile* permettant de compiler les différents programmes réalisés,
- éventuellement un document au format pdf ou texte contenant les informations que vous souhaitez communiquer sur votre travail (Spécifications, description des algorithmes, références bibliographiques, ...)

Modalités de correction :
Attention, un outil de détection de la copie sera utilisé sur les archives déposées. Toute copie avérée entrainera la note de 0 au projet et une convocation par le directeur des études et le responsable de l'unité d'enseignement.
Plusieurs programmes doivent être écrits dans le cadre de ce projet. Si un programme remis ne peut être compilé sur le système d'exploitation utilisé en TP, la note de 0 sera attribuée au projet. La notation prendra en compte la présence de spécifications correctes, les vérifications demandées, la compilation le passage des tests et la qualité du code (fond et forme). Le barème de notation est le suivant :
- Respect des consignes de remise de devoir : 1 points
- Système de gestion de bas niveau du système **RAID5** : 7 points
- Gestion de la table d'*inodes* : 2 points
- Lecture, écriture et suppression d'un fichier sur le système **RAID5** : 3 points
- Diagnostic et réparation du système **RAID5** : 2 points
- Question bonus : 3 points
- Fonctionnalités supplémentaires : 2 points.

1 Les technologies RAID : pour un stockage sur, 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.

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

L'objectif de ce projet est de développer en langage C une version simplifiée de la technologie RAID 5.

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. Ces blocs sont repérés par leur position dans l'ensemble des blocs de données. 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 regroupés 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 octets telle que $k(N-1)T < f <= (k+1)(N-1)T$. 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 $\lceil f/T \rceil \%(N-1)$ blocs de données correspondant aux données du fichier,1 bloc de parité, les blocs de *padding* pour finir la bande contiennent des données arbitraires (ainsi que la fin du bloc contenant la fin des données du fichier, le cas échéant). 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.

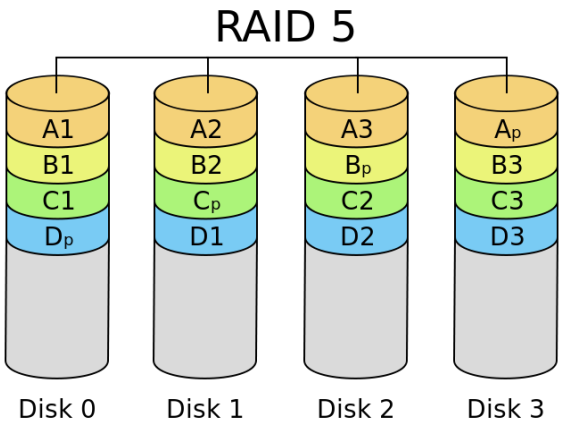

FIGURE 1 – Système RAID 5 avec 4 disques

1.2 Système de gestion de fichier simplifié pour un stockage en RAID 5.

Afin de développer notre système de stockage RAID 5 simplifié (appelé **RAID5** dans la suite du document), l'ensemble de disques le composant sera représenté par un répertoire, chaque disque sera représenté par un fichier nommé di, $i \in [0, N-1]$, dans ce répertoire (i.e. $d0$ $d1$ $d2$ $d3$). Tous les disques seront de même taille.

Les blocs composant les fichiers et les blocs de parités seront lu et écrits sous forme de tableaux d'octets. On considère que les blocs ont une taille fixe de $BLOCK\_SIZE$ octets. Le premier bloc d'un fichier sera toujours écrit sur le premier disque.

Une défaillance de **RAID5** est simulé en supprimmant 1 fichier *di* du répertoire représentant les disques (cela correspondrait à un disque qui ne s'allume plus).

Afin de pouvoir retrouver un fichier sur **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 **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 $FILENAME\_MAX\_SIZE$ caractères et que l'on peut avoir au maximum MAX_FILES fichiers. Les fichiers auront une taille maximale de MAX_FILE_SIZE

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

```
#define NUMBER_OF_DISKS 4
#define BLOCK_SIZE 4
#define FILENAME_MAX_SIZE 32
#define MAX_FILES 100
#define MAX_FILE_SIZE (50*1024)
```

Dans notre système simplifié, on considère que le nombre de disque est fixé à $NUMBER\_OF\_DISKS$ et 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**. Pour *formater* votre système, l'exécutable demande comme paramètre le nom du répertoire, existant, dans lequel sera créé le système **RAID5**, et la taille *disk_size* des disques. Toutes les tailles sont données en octets. Si les disques *di* existent déjà dans ce répertoire, ils sont ré-initialisés. 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.

./cmd_format dir 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 crée) le fichier $dir/d2$.

2 Travail à réaliser.

Ce projet a pour objectif la programmation de l'ensemble des fonctions permettant de gérer le système **RAID5**. Ces fonctions concernent d'une part des fonctions de bas niveau, permettant d'écrire ou de lire un tableau d'octets vers et depuis le système **RAID5**, et des fonctions de haut niveau, permettant de gérer les fichiers stockés sur le système.

Parmi les fichiers fournis en annexe de ce sujet sur moodle, vous trouverez le source du programme *cmd_format.c*, un script shell de test *test_script.sh* ainsi que des répertoires nommés *ref_**. **N'effacer pas** ces répertoires qui seront utilisés par le script de test *test_script.sh* pour vous permettre de valider les différentes étapes de votre développement. Lors de la correction de votre devoir, nous remplaceront le contenu de ces répertoires par nos propres fichiers pour nous assurer du bon fonctionnement de votre programme.

Attention, le script de test ne fonctionnera que si vous avez utilisé les valeurs des constantes indiquées dans le sujet. Vous pouvez les changer pour faire votre développement et vos tests personnels mais devez compiler avec ces constantes pour que la procédure de test automatique fonctionne.

2.1 Système de gestion de bas niveau du système RAID 5

Le système de gestion bas niveau de **RAID5** doit être implanté sous forme d'un module *r5_vdisk*. Dans ce module, les types de données que vous utiliserez pour programmer les différentes fonctions sont les suivants :

Code 1 – Types de données utilisés

```
typedef unsigned int uint;
typedef unsigned char uchar;

/* Type of a block of data */
typedef struct block_s {
 uchar data[BLOCK_SIZE];/* Bytes of the block */
} block_t;

/* Type of the pseudo-inode structure */
typedef struct inode_s{
 char filename[FILENAME_MAX_SIZE];
 uint size;       /* size of the file in bytes */
 uint block_id;   /* start block number on the virtual disk */
 uint nblock;     /* number of blocks of the file */
} inode_t;

/* Type of the inode table */
typedef inode_t inode_table_t[MAX_FILES];

/* Type of the virtual disk system */
typedef struct virtual_disk_s {
 inode_table_t inodes;              /* The inode table */
 int ndisk;                         /* Number of disks */
 FILE *storage[NUMBER_OF_DISKS];    /* Disks are just files */
} virtual_disk_t;
```

1. On considère que notre système **RAID5** est représenté par la variable globale *virtual_disk_t g_disk;*. Avant de pouvoir l'utiliser, il est nécessaire de l'initialiser à partir du nom du répertoire contenant les disques virtuels formatés. Écrire la fonction *init_disk_raid5* qui, à partir du nom du répertoire, initialise cette variable globale. Dans un premier temps, on n'initialisera pas la table d'inodes. 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**.
2. Spécifier et écrire la fonction *compute_nblock* qui calcule le nombre de blocs nécessaires pour stocker un nombre n d'octets.
3. Spécifier et écrire la fonction *compute_nstripe* qui calcule le nombre de bandes nécessaires pour stocker un nombre n de blocs.
4. Spécifier, écrire et vérifier la fonction *compute_parity* qui, à partir d'un tableau de *nblocks* blocs calcule le bloc de parité selon la méthode spécifiée dans la section 1.1.
5. Écrire 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 parités sont référencés par X_p avec $X \in \{A\ B\ C\ D\}$.
6. Écrire la fonction *write_block* qui écrit un bloc *block*, à la position *pos* sur le disque *disk_id* du système **RAID5**.
7. Écrire la fonction *write_stripe* qui écrit une bande de blocs et un bloc de parité, à la position *pos* sur le système **RAID5**.
8. Spécifier et écrire 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_block* 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.
9. Écrire un programme principal nommé *cmd_test1* permettant de tester votre fonction *write_chunk*. Ce programme devra écrire, dans notre système **RAID5**, un buffer de 256 octets représentant toutes

les valeurs possibles que peut prendre une variable de type *unsigned char*. Pour vous assurer que ce programme fonctionne correctement, le script shell *test_script* fourni en annexe de ce sujet sur moodle, devra être exécuté. Si votre fonction d'écriture est correcte, ce programme devra afficher, en première ligne sur votre flux standard de sortie le résultat **cmd_test1** [OK].

10. Spécifier et écrire 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.
11. Spécifier et écrire 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 possibles (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.
12. Spécifier et écrire la fonction *block_repair* qui, en cas d'erreur de lecture, reconstruit le bloc erroné.
13. Spécifier et écrire la fonction *read_chunk* qui lit, à partir de la position *start_block*, un tableau *buffer* n octets, en reconstruise les blocks qui ne peuvent être lus.
14. Écrire un programme principal nommé *cmd_test2* 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 dans la question 9, et devra afficher ces 256 octets sur le flux standard de sortie, le numéro de l'octet et la valeur lue. L'affichage se fera selon le format suivant : `"%d %d \n"`. Pour vous assurer que ce programme fonctionne correctement, le script shell *test_script.sh* fourni en annexe de ce sujet sur moodle devra être exécuté. Si votre fonction de lecture est correcte et que votre affichage respecte le format ci-dessus, ce programme devra afficher, en deuxième ligne sur votre flux standard de sortie le résultat **cmd_test2** [OK].

2.2 Gestion de la table d'*inodes*

Les opérations de bas niveaux 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 MAX_FILES, faisant l'association entre un nom de fichier et ses propriétés, comme présenté dans l'extrait de code 1, est stockée à la position 0 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**.

L'objectif de cette partie est de rajouter au module *r5_vdisk* des fonctions permettant de gérer cette table d'inode.

1. Écrire la fonction *read_inodes* permettant de charger la table d'inodes depuis **RAID5** et utilisez là pour terminer l'initialisation de la question 1.
2. Écrire la fonction *write_inodes* permettant d'écrire la table d'inodes sur **RAID5**. Appelez cette fonction, en en justifiant la raison dans un commentaire, à l'endroit nécessaire pour assurer le bon fonctionnement de notre gestion de fichier.
3. Spécifier, écrire et vérifier 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 **RAID5**, les n premières entrées de la table d'inodes correspondent à ces fichiers.
4. Spécifier et écrire la fonction *get_unused_inode* qui retourne l'indice du premier inode disponible dans la table.
5. Écrire 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. Écrire 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

1. attention, dans notre système, nous utilisons ce terme de façon quelque peu abusive, reportez vous à votre cours de système afin d'identifier clairement les différences entre les inodes UNIX et nos pseudo-inodes

d'inodes depuis le système **RAID5**, pour chaque inode valide, votre programme écrit les champs de l'inode sur la sortie standard en suivant le format :
`"file:[\%s] start at block %d, size %d (%d blocks)\n"`.
Ce programme devra ensuite écrire le nombre d'inodes valide sous la forme `"%d inodes read\n"`.

2.3 Lecture, écriture et suppression d'un fichier sur le système RAID 5

Le module *r5_vdisk* écrit dans les deux questions précédentes peut maintenant être utilisé pour stocker des fichiers, et y accéder, dans le système **RAID5**. Le module *r5_file* que vous allez écrire dans cette partie à pour objectif de fournir les fonctions simplifiées de gestion de fichier. Dans ce module, les fichiers seront représentés par le type de données suivant :

Code 2 – Types de données de représentation des fichiers

```
typedef struct file_s{
 uint size;                    /* Size of the file */
 uchar data[MAX_FILE_SIZE]; /* Bytes of the file */
} file_t;
```

1. Écrire la fonction *write_file* prenant comme paramètres un nom de fichier (chaine de caractères) et une variable de type *file_t* contenant le fichier à écrire dur le système **RAID5**. Si le nom de fichier n'est pas présent dans la table d'inodes, alors un 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 présent. Si le nom de fichier est présent dans la table d'inodes, deux cas sont à traiter :
   - le fichier à une taille inférieure ou égale à la taille du fichier 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 à une taille supérieure à la taille du fichier présent. Il faut alors supprimer l'inode correspondant puis ajouter le fichier en fin de disque (l'ancier fichier laisse un "trou" sur le disque).
2. Écrire 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 fichier lu et la fonction renvoie 1.
3. Écrire la fonction *remove_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 fichier n'est pas présent sur le système **RAID5**.
4. Pour tester ces fonctions écrire trois programmes *cmd_write*, *cmd_read* et *cmd_remove* qui prennent un répertoire de travail comme premier argument et les arguments suivants :.
   - *cmd_write* prends en plus deux arguments : un nom de fichier sur votre système "réel" et un nom de fichier sur le système **RAID5**. Le programme lit le fichier sur le système "réel" et l'écrit sur **RAID5**.
   - *cmd_read* prends en plus deux arguments : un nom de fichier sur votre système "réel" et un nom de fichier sur le système **RAID5**. Le programme lit le fichier sur **RAID5** et l'écrit sur le système "réel".
   - *cmd_remove* prends en plus un argument : un nom fichier sur **RAID5**. Le programme supprime le fichier sur **RAID5**.

   Pour vous assurer que ce programme fonctionne correctement, le script shell *test_script.sh* fourni en annexe de ce sujet sur moodle devra être exécuté. Si vos fonctions de lecture, d'écriture et de suppression de fichiers sont correctes, le script shell devra afficher, en troisième et quatrième ligne sur votre flux standard de sortie les résultats **files** [OK] et **files_disk** [OK].

2.4 Diagnostic et réparation du système RAID 5

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

- le disque manquant est “remplacé” et ré-initialisé par la commande *cmd_format*.
- les données du disque sont reconstruites à partir du contenu des autres disques.

1. Écrire 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.
2. Écrire le programme *cmd_repair* qui prend en paramètre un répertoire de travail et un numéro de disque. Ce programme répare le disque demandé.

Pour vous assurer que ce programme fonctionne correctement, le script shell *test_script.sh* fourni en annexe de ce sujet sur moodle devra être exécuté. Si votre fonction de réparation du système **RAID5** est correcte, le script shell devra afficher, en cinquième ligne sur votre flux standard de sortie le résultat **repair** [**OK**].

3 Question BONUS

Les simplifications que nous avons introduites pour notre système **RAID5** 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.

Annexe

Vous aurez sans doutes besoin d'utiliser les fonctions *C* suivantes :
fopen, fclose, fread, fwrite, fseek, strtol.
Pour savoir comment les utiliser, consultez les pages de manuel correspondantes.

Vous aurez besoin de lire les arguments passés en ligne de commande : pour cela la fonction main doit être déclarée comme suit
int main(**int** argc, **char** **argv).

argc contient le nombre d'arguments passés par l'utilisateur lors du lancement du programme. Ce nombre d'argument inclut la commande appelée.

argv est un tableau de chaines de caractères contenant une entrée par mot de la ligne de commande. argv[0] contient le nom du programme appelé.

L'exemple fourni dans cmd_format peut être analysé pour servir de base à l'écriture des autres programmes demandés.

```
// si on appelle
#./cmd_format hello 500 2
// le main de cmd_format.c
int main(int argc, char ** argv){
// argc = 4
// argv[0] = "./cmd_format"
// argv[1] = "hello"
// argv[2] = "500"
// argv[3] = "2"
}
```