ScratchOS : un OS virtuel
Sujet V0.1

Vincent Dugat

Janvier 2022

NOTES IMPORTANTES :

Ceci est une version du cahier des charges du projet qui, comme tout cahier des charges qui se respecte, est susceptible d'évoluer en fonction des questions et remarques. Des mises à jour seront faîtes, si besoin. Vérifiez que vous êtes bien notifié des posts sur le forum Moodle afin d'être averti des nouvelles versions.

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 les intervenants 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 au codage.
- Normalement à plusieurs on va plus loin. L'organisation de votre équipe de développement 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 (validation) 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 du codage est de 80. Ce score est à partager entre tous les membres d'une équipe (la note finale de chacun est sur 20 points. On fera la moyenne avec la note de pilotage de projet elle-même sur 20 points). Le total sera sur 20 points. Une seule note sera transmise à l'administration.
- Le partage du score entre les membres d'une équipe 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). La note d'un membre d'une équipe est plafonnée à 20 points.

1 Scratch OS, un système d'exploitation (très) simplifié UNIX like

On va créer, from scratch, un système d'exploitation avec des fichiers, des utilisateurs, une procédure de connexion, un interprète de commande. Le disque dur de la machine sera simulé par un fichier qu'on appellera le disque virtuel, qu'il faudra formater et qui contiendra les fichiers et le catalogue du disque. Le système d'exploitation proprement dit sera un programme C, résident en mémoire, gérant des tables lui permettant de gérer un système de fichiers, qui sera écrit sur un disque dur simulé, pour le projet, par un fichier.

Le système gérera diverses opérations sur les fichiers, une table des utilisateurs, des droits (simplifiés) de ces derniers sur les fichiers, une procédure de connexion et un interprète de commande.

Comme il est d'usage en système, la logique de fonctionnement sera divisée en couches.

1.1 Le DD virtuel et son formatage

Donc la première étape pour développer notre système est le stockage du système de fichiers. Nous allons utiliser un fichier qui jouera le rôle de disque dur. Ce fichier sera placé dans un répertoire. Le disque sera nommé d_0.

Afin de pouvoir utiliser ce fichier pour gérer le système de fichier de l'OS, nous définissons une structure contenant le nom du fichier disque virtuel du système et qui est stocké à partir du premier bloc du fichier disque. On ajoute un tableau de structures, assimilable à la table d'inodes d'un système UNIX, et de taille fixe. Chaque entrée du tableau d'inode correspond à un fichier du disque et donne son nom, son premier bloc, sa taille et d'autre informations. Le tout définit le catalogue du disque.

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

Le fichier header fournit contient des valeurs indicatives pour ces constantes. Vous pouvez faire le projet avec ces valeurs mais veuillez vous assurer toutefois que si on change leur valeurs, votre système reste opérationnel.

1.1.1 Le formatage du disque virtuel

Le fichier source $cmd\_format.c$, fourni sur Moodle en annexe de ce sujet, permet de formater le disque. Pour formater votre système, l'exécutable demande comme paramètre le nom du répertoire existant dans lequel sera créé le fichier disque, et la taille *disk_size* du disque. La taille est donnée en octets. Si le disque d_0 existe déjà dans ce répertoire, il est réinitialisé. S'il n'existe pas dans le répertoire, un fichier dont le contenu est mis à 0 est créé 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.

Exemple : *Syntaxe* : command nom_répertoire taille_fichier (octets)
./cmd format dir 500000 création du fichier $d0$ dans le répertoire dir, avec une taille de 500000 octets.

2 Travail à réaliser en langage C

Le travail à réaliser est divisé et organisé en couches comme il est d'usage pour un système d'exploitation. Chaque couche implémente des fonctionnalités utilisant celles de la couche inférieure et fournissant des services à la couche supérieure. Un fichier $os\_defines.h$, fourni sur Moodle, donne quelques structures de données à utiliser.

Dans tout le sujet les fonctions de lecture/écriture sur le disque virtuel 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 le système : écrire, lire, effacer. Les fonctions suggérées ici sont le minimum utile. Vous pouvez avoir besoin d'autres fonctions.

1. On considère que notre système est représenté par la variable globale *virtual_disk_sos ;*. Avant de pouvoir l'utiliser, il est nécessaire de l'initialiser à partir du nom du répertoire contenant le disque virtuel formaté.

   Ecrire la fonction *init_disk_sos* qui, à partir du nom du répertoire, initialise cette variable. Dans un premier temps, on n'initialisera pas la table d'inodes (couche 2). 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" proprement notre système, c'est à dire sauvegarde le catalogue sur le disque virtuel.

2. Ecrire la fonction *compute_nblock* qui calcule le nombre de blocs nécessaires pour stocker un nombre n d'octets.

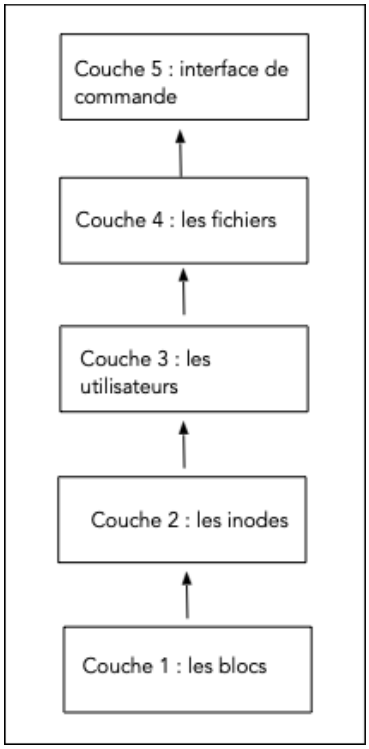

FIGURE 1 – Organisation en couches

3. Écrire la fonction *write_block* qui écrit un bloc *block*, à la position *pos* sur le disque du système.
4. Ecrire la fonction *read_block* qui lit un bloc *block* de données, à la position *pos* sur le disque du système. En cas d'échec de lecture, cette fonction doit renvoyer un code d'erreur que vous préciserez.

   Remarque : Les fonctions C $fseek$, $fwrite$ et $fread$ semblent les plus indiquées pour ce travail.
5. **Conseillé** : Ecrire des fonctions d'affichage à l'écran des blocs de type *block_t* en hexadécimal pour vérifier.

2.2 Couche 2 : gestion du catalogue, du super bloc et de la table d'inodes

Le catalogue du système de fichiers est une structure contenant les informations nécessaires pour gérer les fichiers. A ce niveau, certaines de ces informations seront complétées plus tard.

```
typedef struct inode_s{
  // type file vs dir
  char filename[FILENAME_MAX_SIZE]; // dont '\0'
  uint size; // du fichier en octets
  uint uid; //user id proprio
  uint uright; //owner's rights between 0 and 3 coding rw in binary
  uint oright; // other's right idem
  char ctimestamp[TIMESTAMP_SIZE]; // date création : 26 octets
  char mtimestamp[TIMESTAMP_SIZE]; // date dernière modif. : 26 octets
  uint nblock; // nblock du fichier = (size+BLOCK_SIZE-1)/BLOCK_SIZE
  uint first_byte; // start number of the first byte on the virtual disk
} inode_t;
```

Les opérations de bas niveau programmées dans la partie précédente permette de stoker des données binaires sur le système. 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 ci-dessus, est stockée à la position $INODES\_START$ sur le système. 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, soit une position égale à 0 si elle ne désigne pas de fichier existant dans le système.

Afin de faciliter l'accès à l'information on utilise un super bloc qui donne le nombre de fichiers de système, le nombre d'utilisateurs, le nombre de blocs utilisés et le numéro du premier octet libre dans le système. La taille du super bloc est fixe (4 blocs sur le système)

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 *write_super_block* qui écrit le super bloc au tout début du fichier disque.
2. Ecrire la fonction *read_super_block* qui lit le super bloc au tout début du disque.
3. Ecrire une fonction qui met à jour le champs $first\_free\_byte$ du super bloc.
4. Ecrire la fonction *read_inodes_table* permettant de charger la table d'inodes depuis le système et utilisez-la pour terminer l'initialisation de la question 1.
5. Ecrire la fonction *write_inodes_table* permettant d'écrire la table d'inodes sur le système à la suite du super bloc.
6. 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 système, les n premières entrées de la table d'inodes correspondent à ces fichiers. Le super bloc devra être mis à jour.
7. Ecrire la fonction *get_unused_inode* qui retourne l'indice du premier inode disponible dans la table.
8. 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.
9. Ecrire un programme *cmd_dump_inode* qui servira pour les tests sur les fichiers. Ce programme prends en argument le nom du répertoire contenant le fichier disque. Après avoir lu la table.

2.3 Couche 3 : Gestion des utilisateurs

Le système sera multi-utilisateurs. Ils seront gérés par une table des utilisateurs (un tableau), l'indice de chaque utilisateur dans le tableau servira de *user id* (UID). Il y a un utilisateur par défaut nommé "root" qui sera créé à l'installation du système, comme dans une installation de système Linux, en case 0 (un #define ROOT_UID 0 peut être utile).

2.4 Couche 4 : Gestion des fichiers et des droits

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

```
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 avec des droits (très) simplifiés.

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. 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 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 :

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

— 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, cette variable n'est pas modifiée et la fonction renvoie 0. Si le fichier est présent sur le système cette variable contient les données du

   fichier 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 fichier n'est pas présent sur le système.
4. Ecrire une fonction *load_file_from_host* qui prend en paramètre le nom d'un fichier de l'ordinateur que vous utilisez (nommé *host*) et l'écrit sur le système. Le nom du fichier sur le système 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 SOS et l'écrit sur l'ordinateur hôte. Le nom du fichier sera aussi le même.

Pour relier les fichier est les utilisateurs, les inodes intègrent l'uid de l'utilisateur créateur, ses droits (par défaut rw), et les droits des autres utilisateurs (par défaut rien). On pourra gérer ces droits par un simple entier entre 0 et 3 : si on représente chaque droit par une minuscule s'il est absent et par une majuscule s'il est présent on a :
rw = 0 aucun droit
rW = 1 droit d'écriture
Rw = 2 droit de lecture
RW = 3 droit d'écriture et lecture

Chaque inode contiendra aussi la date de création du fichier et la date de sa dernière modification (voir la fonction time_stamp.c). Vous pouvez maintenant compléter l'initialisation et la gestion de la table d'inode.

2.5 Couche 5 : l'interprète de commande

Pour utiliser toutes le système de fichiers mis en place nous créons dans cette partie un interprète de commandes basique mettant en place le système d'exploitation proprement dit.

Attention : L'implémentation des commandes demandées peut vous obliger à revenir sur les fonctions déjà écrites pour leurs ajouter des fonctionnalités.

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

   Avant d'entrer dans cette boucle, le système demandera un login et un mot de passe à l'utilisateur, calculera le haché du mot de passe, cherchera le login dans la table des utilisateurs, vérifiera le haché du mot de passe et si c'est conforme, démarrera l'interprète de commande. Dans la cas contraire, le système ré-affiche la demande de login, mot de passe. Au bout de trois erreurs consécutives, le système quitte.
2. Les commandes connues de l'interprètes sont : Commandes sur les fichiers :
   — ls [-l] : liste le contenu du catalogue. Un argument optionnel pour un affichage court (nom du fichier, taille) ou long (tout).

— cat <nom de fichier> : affiche à l'écran le contenu d'un fichier si l'utilisateur a les droits,
— rm <nom de fichier> : supprime un fichier du système si l'utilisateur a les droits,
— cr <nom de fichier> : crée un nouveau fichier sur le système, le propriétaire est l'utilisateur.
— edit <nom de fichier> : édite un fichier pour modifier son contenu si l'utilisateur a les droits,
— load <nom de fichier> : copie le contenu d'un fichier du système "hôte" sur le système avec le même nom (assimilé à une création),
— store <nom de fichier> : copie le contenu d'un fichier du système sur "hôte" avec le même nom,
— chown <nom de fichier> <login autre utilisateur> change le propriétaire d'un fichier si le demandeur a les droits
— chmod <nom de fichier> <droit> change les droits d'un fichier pour tous les autres utilisateurs si le demandeur a les droits
— listusers
— quit : sort de l'interprète de commande et du programme en sauvegardant le système de fichiers sur le disque.

Droits réservés à root : Remarque : root a tous les droits.
— adduser : ajouter un utilisateur. La commande demande le login et de créer un mot de passe.
— rmuser <login> : supprimer un utilisateur

Chaque commande devra gérer 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 disque des données et la fermeture des fichiers.

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

3 Installeur

A la manière des programmes d'installation des distributions Linux, il s'agit ici de créer un programme appelant certaines des fonctionnalités précédentes avec un *main* dédié. Ce programme sera chargé de créer et d'initialiser le système de fichiers, créer l'utilisateur root, demander à l'utilisateur de définir un mot de passe pour cet utilisateur, créer un fichier *passwd* contenant le login root et le haché du mot de passe. Sauvegarder le tout sur le disque virtuel.

Remarque : A la fin de l'installation, le système de fichiers ne contient qu'un fichier : le fichier passwd.

Le programme SOS supposera le système ainsi initialisé et en lira directement les données.

3.1 Programme Java

Le programme Java consiste à implémenter des outils d'analyse et de diagnostique du système de fichiers de SOS :

1. programmer une fonction d'analyse de la cohérence du système en vérifiant :
   — les informations du super bloc
   — les informations du catalogue. Par exemple que le bloc de début d'un inode ajouté à la taille correspond au début du fichier suivant.
   — les informations de la table des utilisateurs.

   Cette fonction pourra appeler des sous-fonctions.
2. Le problème de la fragmentation : les simplifications que nous avons introduites pour notre système génèrent une forte fragmentation après chaque suppression de fichier. Ecrire un programme de défragmentation permettant de compacter le stockage des fichiers en supprimant les trous occasionnés par la suppression d'un fichier. Le résultat devra être lisible par le programme C de SOS sans altération ou perte d'information.
3. Ecrire une interface graphique d'interaction pour gérer les différentes commandes précédentes. Les spécifications sont libres.