Projet 3 Minix - Defragmentation de fichiers

Release 1

Fabien Duchêne and Emery Kouassi Assogba

March 27, 2013

CONTENTS

1 Introduction **1**

2 Le livre de Minix **3**

3 Mission **5**
- 3.1 Objectif . . . 5
- 3.2 Minix 3 . . . 5
- 3.3 Programme attendu . . . 7
- 3.4 Calendrier . . . 8
- 3.5 Présentation du projet . . . 8

4 Consignes détaillées **9**
- 4.1 Utilisation de Minix dans les Salles . . . 9
- 4.2 Stratégie de tests . . . 11
- 4.3 Rapport final . . . 12
- 4.4 Remise de votre travail . . . 12
- 4.5 Critères d'évaluation . . . 13

CHAPTER

ONE

INTRODUCTION

Le système de fichiers Minix est basé sur des *inodes* qui contiennent des pointeurs vers les différentes zones du disque allouées pour un fichier. Pour chaque zone d'un fichier, un pointeur correspondant est stocké dans l'*inode* ou dans l'une de ses zones indirectes, ce qui permet au système de disperser un fichier sur le disque dur en fonction des espaces qu'il y trouve. Par exemple, si l'on crée un fichier pour y écrire 10000 octets, et que le système commence à écrire dans un espace sur le disque de 4000 octets seulement, il devra trouver un autre emplacement libre sur le disque pour enregistrer le reste du fichier. Le fichier devient donc fragmenté, ce qui peut augmenter le coût de lecture du fichier.

Si vous n'avez pas compris ce qui suit, lisez **avant toute chose** le livre Minix (dont les références sont données au chapitre suivant).

Les systèmes courants de défragmentation parcourent un disque dur et ramènent toutes les zones occupées en début de disque, tout en garantissant que chaque fichier est stocké dans des zones contigües. Pour des raisons de temps, nous ne nous intéresserons qu'à la défragmentation d'un seul fichier passé en argument d'un appel système `defrag()`. L'objectif de ce projet sera, pour un fichier passé en argument, de vérifier s'il est fragmenté, et, s'il l'est, de copier l'entièreté de ses blocs vers une région suffisamment grande du disque.

CHAPTER

TWO

LE LIVRE DE MINIX

Pour ce projet nous vous conseillons le livre de Andrew Tanenbaum, “Operating Systems Design and Implementation (3rd Edition)” (ISBN 0131429388) aussi apellé “The Minix book”.

La réalisation du projet demande de bien comprendre l’architecture du système, il est donc nécessaire, **avant de modifier le code**, d’avoir lu les sections du livre décrites ci-dessous. Notez que cette liste est à la fois incomplète et trop complète: à vous d’identifier les parties qui s’avèreront cruciales pour votre projet.

A noter également: la quantité de lecture est assez importante, il faut donc prévoir commencer la lecture au plus tôt.

- `Section 1.3`: Concepts.
- `Section 1.4`: Appels sytème.
- `Section 3.1`: Principes d’E/S (Hardware).
- `Section 3.2`: Principes d’E/S (Software).
- `Section 3.4`: Vue d’ensemble des I/O dans Minix.
- `Section 5.1`: Les fichiers.
- `Section 5.2`: Les répertoires.
- `Section 5.3`: Implémentation du système de fichiers.
- `Section 5.6`: Vue globale du système de fichiers de Minix 3.
- `Section 5.7`: Implémentation du système de fichiers Minix 3.

Il est utile de regarder (sans le modifier dans un premier temps) le code de Minix au fur et à mesure de la lecture. Notez que certaines indications du livre peuvent différer de la version de Minix installée dans les salles, car celle-ci est plus récente.

CHAPTER

THREE

MISSION

Cette section décrit plus en détails ce que vous devez faire pour ce projet et comment.

3.1 Objectif

L'objectif de ce projet est que vous vous plongiez dans un projet complexe, que vous compreniez le fonctionnement d'un OS et que vous le modifiez. Pour ce faire vous devrez implémenter deux nouveaux appels systèmes dans Minix 3.

3.2 Minix 3

Minix 3 est un OS micro-kernel compatible POSIX. L'approche micro-kernel est basée sur les idées de modularité et de confinement des fautes en découpant le système en petits processus autonomes.

Le système d'exploitation Minix est structuré comme suit. Un noyau minimal fournit un gestionnaire d'interruption, un mécanisme pour démarrer et arrêter des processus, un ordonnanceur et un mécanisme de communication entre processus. Comparé à un système monolithique (ex: Linux), les fonctionnalités standards du système d'exploitation (drivers, système de fichiers, gestion de la mémoire de haut-niveau, etc.) forment des processus user-space indépendants encapsulé dans leur espace d'adressage privé.

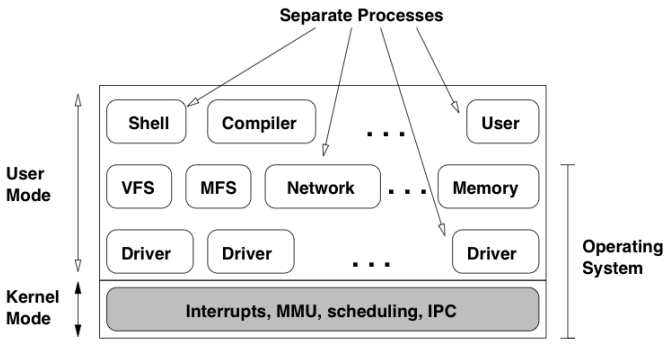

Figure 3.1: La structure du système d'exploitation Minix 3.

Même si dans Minix, les serveurs et drivers sont des processus utilisateurs, ils peuvent être structurés logiquement en 3 niveaux. Le niveau le plus bas des processus utilisateurs comprennent les drivers qui contrôlent chacun un périphérique. Au dessus de ce niveau, se trouvent les processus serveurs. Parmis ceux-ci on trouve le serveur virtuel de gestion des systèmes de fichiers, l'implémentation des systèmes de fichiers proprement dite, le serveur de processus, le

serveur de réincarnation, etc. Tout au dessus de ces serveur se trouvent les processus utilisateurs classiques comprenant les shells, compilateurs, etc.

3.2.1 Le système de fichier Minix

Le système de fichiers est implémenté dans le niveau des serveurs dans Minix. Depuis la version 3.1.3, Minix comprend un serveur virtuel de gestion de systèmes de fichiers (VFS). Ce serveur est une couche d'abstraction aux implémentations des systèmes de fichiers. VFS offre donc une interface commune pour que les applications puissent accéder à différents types de systèmes de fichiers de manière uniforme (les différences entre les systèmes de fichiers sont donc cachées). Cette interface permet également d'ajouter plus facilement de nouveaux systèmes de fichiers.

3.2.2 Etapes principales de l'exécution d'un appel système

Afin de démontrer le fonctionnement général du système de fichiers virtuel de Minix, considérons l'appel système `stat()` (pour plus d'info, voir `man 2 stat`) avec comme argument `/usr/src/vfs.c`. Considérons également qu'il y a une partition montée sur le répertoire `/usr` qui est géré par un processus FS séparé. La figure suivante montre les principales étapes de cet appel système dans Minix.

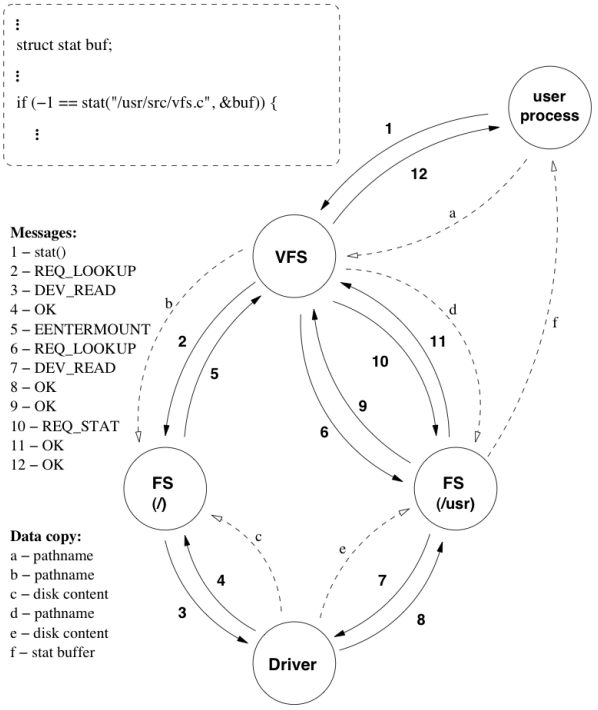

Figure 3.2: Messages échangés et données copiées pendant l'execution de l'appel système `stat()` (les lignes pleines montrent les messages échangés pendant l'appel système, les lignes en pointillés montrent les données copiées).

1. Le processus utilisateur appelle la fonction `stat()` de la librairie POSIX qui construit le message de requête et l'envoie au processus VFS.

1. Le VFS copie le chemin vers le fichier du processus appelant.

2. Le VFS effectue une recherche sur le chemin du fichier. Comme ce chemin est absolu, le serveur FS de la racine doit effectuer la recherche.

2. Le FS de la racine copie le chemin vers le fichier du processus appelant (ici le VFS).

3. Pendant la recherche, le FS de la racine demande au driver de lire le bloc correspondant à `/usr` du disque dur.
4. Le driver lit le bloc et renvoie OK.
3. Le driver copie le bloc dans la cache du FS.
5. Le FS examine l'inode du dossier `/usr` et découvre que c'est une autre partition qui est montée sur ce dossier. Il envoie le message `EENTER_MOUNT` au VFS.
6. Le VFS regarde dans sa table de montage virtuel sur quel processus FS est responsable de la partition `/usr`. La recherche doit se continuer dans ce processus. Il envoie donc un message avec le reste du chemin.
2. Le FS responsable de `/usr` copie le chemin vers le fichier du processus appelant (ici le VFS).
7. Le processus qui se charge de la partition `/usr` continue la recherche. Il a besoin d'information du disque, il demande donc au driver de lire des blocs et de les transférer dans la cache.
8. Le driver lit le bloc et renvoie OK.
5. Le driver copie le bloc dans la cache du FS responsable de `/usr`.
9. Le FS termine la recherche et transfère les détails de l'inode au VFS.
10. Le VFS a toutes les informations nécessaires pour exécuter l'appel système `stat()`. Il demande dès lors au FS responsable de `/usr` d'exécuter l'opération `stat()`.
11. Le processus FS remplit le buffer. Faisons l'hypothèse que toutes les informations se trouvent déjà dans la cache.
6. Le FS copie la structure dans l'espace d'adressage du processus utilisateur.
12. Le VFS reçoit la réponse du FS et renvoie la valeur à la libraire POSIX.

Note: Des informations complémentaires sur le fonctionnement de la communication entre le VFS et les FS se trouvent à l'adresse suivante: The VFS-FS protocol.

Les informations sur l'implémentation du FS, c'est-à-dire le *Minix File System* (MFS), se trouve dans les sections 5.6 et 5.7 du livre de référence.

3.3 Programme attendu

Vous devez implémenter deux appels système qui ont les spécifications suivantes:

```
/**
 * entrée: *file* Une chaîne de caractères indiquant le chemin d'accès
 *                vers un fichier, absolu ou relatif.
 * valeur de retour: La valeur de retour est le nombre de
 *                fragments non contigus qui constituent le fichier file.
 *                Un fichier non fragmenté contient un fragment.
 *            Si une erreur se produit, -1 est retourné, et errno
 *                reçoit le code correspondant à l'erreur.
 */
 int nfrags(const char * file);

/**
 * entrée: *file* Une chaîne de caractères indiquant le chemin d'accès
 *              vers un fichier, absolu ou relatif.
 * valeur de retour: La valeur de retour est le nombre de
 *                fragments qui constituaient le fichier file avant
 *                la défragmentation.
```

```
 *                Si defrag retourne une valeur positive, alors
 *                l'exécution de nfrags sur le même fichier donnera 1.
 *            Si une erreur se produit, -1 est retourné, et errno
 *                reçoit le code correspondant à l'erreur.
 */
int defrag(const char * file);
```

Ces appels système devront être implémentés dans le serveur VFS. Un support à cet appel système ne devra être implémenté que dans le serveur MFS. Vous aurez besoin que le premier fonctionne pour pouvoir réaliser le second. Votre programme devra au mieux s'intégrer dans le code de Minix et devra respecter les convention de codage (voir Coding Style).

Note: Afin de gagner du temps, nous ferons les hypothèses suivantes:

- Aucun fichier n'a de trou (sparse file). Les *sparse files* sont des fichiers dont certains pointeurs de zone ont la valeur `NO_ZONE`. Cela signifie que l'entièreté de la zone contient des zéros, et qu'il n'est donc pas nécessaire de la stocker sur disque. Dans notre cas, nous supposerons que tous les fichiers possèdent toutes leurs zones effectivement sur le disque.
- Seuls les fichiers *normaux*, et non ouverts ailleurs doivent être gérés. Vous devrez donc vérifier avant toute chose, dans votre appel système, que le fichier n'est pas un dossier, un fichier spécial (fichier d'accès à des périphériques dans `/dev`), ou un fichier normal déjà ouvert ailleurs (car il y aurait un risque de mettre le système de fichiers dans un état incohérent). Si le fichier n'était pas ouvert ailleurs au moment de la vérification, vous pouvez supposer que rien ne tentera d'y accéder jusqu'à la fin de votre appel système.

3.4 Calendrier

- 27 mars: remise de l'énnoncé du projet.
- 27 mars-16 avril: lecture du livre de Minix, pré-projet.
- 17 avril: permanence en salle Intel, présentation de l'architecture que vous comptez déployer.
- 24 avril: permanence en salle Intel, implémentation du problème.
- 8 mai: dernière permanence en salle Intel, finalisation du projet.
- 10 mai, 23h55: remise de l'implémentation et du rapport (sur iCampus, section travaux).
- semaine 13 au 17 mai: présentation par groupe du projet devant les assistants.

3.5 Présentation du projet

Durant cette présentation, vous devrez effectuer, par groupe, une brève (+- 15 mins) présentation de votre projets au corps enseignant, slides à l'appui. La présentation sera suivie d'une séance de questions/réponses. Cette évaluation permettant de juger de votre conaissance du projet, sa note sera prise en compte dans la note finale du projet, veuillez donc à bien la préparer.

CHAPTER

FOUR

CONSIGNES DÉTAILLÉES

Cette section détaille les consignes précises qui doivent être suivies lors de ce projet. Tout non-respect de ces consignes pourra éventuellement vous coûter de nombreux points.

4.1 Utilisation de Minix dans les Salles

Minix est disponible sur les machines des salles Siemens et Intel via *Qemu* un outil de virtualisation. Pour se faire un *Makefile* a été développé afin de vous faciliter la tâche.

4.1.1 Initialisation du répertoire de travail

Afin d'utiliser ce *Makefile*, vous devez dans un premier temps créer un dossier où sera stocké votre projet:

```
davidof:~ fduchene$ mkdir ~/minix # à changer par ce que vous voulez
davidof:~ fduchene$ cd ~/minix
davidof:~/minix fduchene$ curl -L http://goo.gl/OefSy -o Makefile
```

Une fois le *Makefile* téléchargé, il faut initialiser le dossier. L'initialisation consiste en trois étapes:

1. La création des disque dur virtuels `minix_local.cow` et `additianal_disk.img`. Le premier correspond au disque principal qui est une version *Copy-on-Write* du disque principal. Le deuxième (qui apparaitera comme `/dev/c0d1` dans Minix) servira à créer un nouveau système de fichier *mfs* afin de tester le bon fonctionnement de vos appels systèmes.
2. La copie des sources de Minix dans le dossier `./src`. Les sources que vous devez modifier sont copiées, le reste est référencé via un lien symbolique.
3. La création d'un dossier `./test` qui contiendra, vos scripts/codes qui permettront de tester le bon fonctionnement de vos appels systèmes.

L'ensemble de ces étapes est réalisé en exécutant:

```
davidof:~/minix fduchene$ make init
Creation du disque virtuel: minix_local.cow... done.
Creation du disque virtuel: additional_disk.img... done.
Creation du dossier: src... done.
Creation du lien symbolique: src_orig... done.
Creation du dossier: test... done.
davidof:~/minix fduchene$ ls
additional_disk.img  Makefile  minix_local.cow  src  src_orig  test
```

Note: Le *Makefile* fourni les cibles suivantes:

```
davidof:~/minix fduchene$ make
Utilisez 'make <target>' où <target> est :
  init          pour initialiser le répertoire du projet
  run           pour executer la machine virtuelle (en console)
  run_x11       pour executer la machine virtuelle (en fenêtre)
  patch         pour générer le patch
  dist          pour générer une archive comprenant le patch, le
                rapport et le dossier test
  clean         supprime l'archive et le patch
  mrproper      supprime les disques virtuels
```

Si vous voulez, par exemple, régénérer le disque dur virtuel `additional_disk.img`. Il vous suffit de le supprimer et d'ensuite appeler `make init`. Il en va de même pour tous les fichiers/dossiers qui se trouvent dans ce répertoire.

4.1.2 Exécution de la machine virtuelle

Une fois le dossier initialisé vous pouvez simplement modifier les sources de Minix dans le dossier `./src`. Une fois les sources modifiées vous devez démarrer la machine virtuelle contenant Minix. Pour ce faire:

```
davidof:~/minix fduchene$ make run # ou make run_x11
...
Hit a key as follows:

    1  Start MINIX 3
    3  Start Custom MINIX 3
1
```

Au démarrage vous devez taper `1` afin de démarrer le noyau *MINIX 3*. Une fois la machine virtuelle lancée, vous pouvez vous loguer en tapant comme login `root`. A partir de ce moment vous devez configurer la machine virtuelle, afin qu'elle puisse communiquer avec l'hôte (à n'exécuter qu'une seule fois!):

```
# echo "export HOST_USERNAME=<votre_login_des_salle>" >> .profile
# echo "export HOST_MINIXPATH=<chemin_vers_projet>" >> .profile
# . .profile
```

Pour mettre à jour à la fois les sources de Minix et le dossier `test` de la machine virtuelle:

```
# ./update_minix
```

Ce script utilise `rsync` sur `ssh` pour synchroniser les deux répertoires avec la machine hôte.

Une fois la machine mise à jour, vous pouvez re-compiler le système Minix:

```
# cd /usr/src/tools
# make libraries hdboot
```

Pour plus d'information sur comment recompiler le système: Rebuilding the System.

Une fois que le système a été recompilé, vous devez redémarrer la machine:

```
# shutdown
...
MINIX will now be shut down ...
d0p0s0> menu
Hit a key as follows:
```

```
    1  Start MINIX 3
    3  Start Custom MINIX 3
3
```

La machine vas dès lors redémarrer sur votre nouveau noyau (*Custom MINIX 3*). Pour stopper proprement la machine, une fois que vous avez terminé, vous devez taper `off` à la place de `menu`.

Note: Il y a évidement moyen de faire tourner Minix sur vos machines personnelles. Cependant, cela est fortement déconseillé. En effet, aucun support ne sera fourni en dehors des machines des salles. Vous risquez également de ne pas respecter le format de la soumission, ce qui vous fera perdre des points inutilement.

N'oubliez pas que vous pouvez sans problème accéder aux machines des salles, même hors de l'UCL, en passant par *Permeke* et ensuite en vous connectant aux machines:

```
$ ssh -C fduchene@permeke.info.ucl.ac.be # -X si run_x11

sirius:~ fduchene$ ssh volcan10 # volcanXY ou intelXY // -X si run_x11
fduchene@volcan10's password:

davidof:~ fduchene$ cd minix
davidof:~/minix fduchene$ make run
```

4.1.3 Pré-projet

Afin de découvrir le code de Minix et donc pouvoir répondre aux questions, nous vous proposons d'implémenter ce pré-projet:

```
struct pid_s {
    pid_t me;
    pid_t parent;
};

/**
 * sortie: *pids* Une structure contenant le pid du processus appelant
 *            et le pid du parent.
 * valeur de retour: Renvoi toujours un code de succès.
 */
int getpidinfo(struct pid_s * pids);
```

L'objectif principal est de pouvoir identifier les fichiers à modifier afin d'y inclure ce nouvel appel système. Une bonne façon de comprendre comment un appel système est instrumenté est de tracer son exécution (par ex, regardez ce qui se passe entre l'appel de `read()` par un processus user-space et son retour de fonction) et d'identifier ce qui se passe (où est défini l'appel système, comment est-il exécuté, comment est-il implémenté).

Note: Il n'y a rien a rendre pour ce pré-projet. Celui-ci a pour but de vous faciliter le travail pour le projet.

4.2 Stratégie de tests

Dans ce projet, nous ne vous demandons pas d'effectuer des tests unitaires pour chaque appel système. Nous vous demandons simplement de tester leur bon fonctionnement à l'aide de scripts/code que vous placerez dans le dossier

`test/`. Il faudra que ce dossier comporte au moins un *Makefile*, qui, à l'exécution de `make`, exécute tous les tests. Si nécessaire, vous pouvez également inclure un README pour une explication plus détaillée (en plus du rapport).

Nous listons des outils qui peuvent vous être utiles pour les tests que vous réaliserez dans le projet:

- `dd`: Permet de générer des fichiers de taille arbitraire à partir d'une source. La source peut par exemple être `/dev/zero` (source infinie de zéros) ou `/dev/urandom` (source infinie de nombre aléatoires).
- `/sbin/mkfs.mfs`: Crée un système de fichiers MFS sur une partition ou un disque. Pour initialiser le disque dur additionnel: `/sbin/mkfs.mfs /dev/c0d1`.
- `/sbin/fsck.mfs`: Vérifie l'état du système de fichiers. Utile pour vous assurer que vous avez laissé le système de fichiers dans un état cohérent après une défragmentation.
- `mount`: Permet de monter une partition ou un disque dans un dossier. Pour monter le disque additionnel (après l'avoir préalablement initialisé) dans le dossier `/mnt/disk`: `mount /dev/c0d1 /mnt/disk`.
- `diff`: Permet de lister les différences entre deux fichiers. Si par exemple vous faites une copie d'un fichier avant de le défragmenter, vous pouvez vérifier que la copie reste identique à la version défragmentée.
- `du`: Permet de contrôler l'espace disque occupé par un fichier.
- `df`: Permet de contrôler l'occupation d'une partition complète.

4.3 Rapport final

Le rapport final fera au **maximum 4 pages** en format *IEEEtran 10pt 1 colonne* (LaTeX, Word). Il présentera brièvement (mais efficacement!) votre solution. Il contiendra au moins:

1. La description de votre architecture: [2-2,5 page]
   - Fichiers modifiés et la raison de la modification.
   - Explication détaillée de l'implémentation (choix, efficacité, etc.).
   - ...
2. La description de votre stratégie de tests. Est-ce que vos tests assurent que vos appels systèmes fonctionnent dans tous les cas ? [1 page]
3. Si votre programme ne fonctionne pas comme il le devrait, une explication de la cause et ce qu'il faudrait faire pour résoudre le problème. Eventuellement, des pistes d'améliorations possibles. [0,5 page]

4.4 Remise de votre travail

L'entièreté du travail, rapport et implémentation, doit être remis sur iCampus, section Travaux pour le 10 mai à 23h55 (heure d'iCampus faisant foi).

Une cible du *Makefile* permet de générer directement l'archive à soumettre sur iCampus. Pour ce faire, vous devez au préalable avoir copié votre rapport dans le dossier du projet et l'avoir renommé `rapport.pdf`. Pour générer l'archive:

```
davidof:~/minix fduchene$ make dist
Génération du patch: minix_3.1.8r3_nfrags_defrag_fduchene.patch... done.
Génération de l'archive: projet_minix_fduchene.tar.gz... done.
```

Ceci générera une archive `projet_minix_${USER}.tar.gz` qui contiendra le `patch` de vos modifications, votre rapport, ainsi que votre dossier `test`. Verifiez cependant que tout le contenu demandé se trouve dasn votre archive avant de soumettre.

4.5 Critères d'évaluation

- Architecture des appels système.
- Respect des conventions Minix.
- Qualité du code source.
- Stratégie de tests et tests effectués.
- Rapport.
- Respect des consignes.