# Projet Rock'n'Roll : MarioBrOS

# Florentin GUTH Lionel ZOUBRITZKY

#### 16 mai 2017

# Table des matières

1	_	ganisation générale
	1.1	Makefile
	1.2	Boot
<b>2</b>	$\mathbf{Sys}$	tème de fichiers
	2.1	Côté hardware
	2.2	Côté interface utilisateur
	2.3	Implémentation et difficultés rencontrées
3	Ges	etion de la mémoire
	3.1	Organisation de la mémoire
	3.2	Paging
	3.3	Tas
4	Mu	ltitasking
	4.1	Processus
	4.2	Shell
		La commande run
5	Bug	gs et améliorations possibles

# 1 Organisation générale

### 1.1 Makefile

Les principales cibles de compilations sont les suivantes :

- make, make disk, make diskq: crée une image disque et lance QEMU;
- make diskb : crée une image disque et lance Bochs;
- make clean : supprime tous les fichiers générés.

La compilation (pour le disque) s'effectue en plusieurs étapes :

- créer une image disque vide EXT2,
- installer *GRUB* dessus,
- y copier kernel.elf, l'exécutable du kernel une fois compilé,
- compiler les programmes utilisateurs (dossier *progs*) en exécutables et les copier,
- lancer l'émulateur.

#### $1.2 \quad Boot$

MarioBrOS fonctionne aussi bien sous Bochs que QEMU. Le fonctionnement du boot est assez simple :

- GRUB est d'abord lancé et détecte notre noyau kernel.elf,
- on rentre alors dans le fichier loader.s qui permet de récupérer la taille de la mémoire,
- on lance ensuite la fonction kmain qui initialise tous les composants et rentre dans la boucle principale.

# 2 Système de fichiers

Le système de fichier implémenté est EXT2.

#### 2.1 Côté hardware

Les fonctions readPIO, readLBA et writeLBA permettent d'accéder au disque dur en utilisant le système d'adressage LBA 28 bits.

Nous avons remarqué que l'émulateur *Bochs* n'allumait pas aux moments voulus le bit DRQ du port de contrôle ATA, qui signifie que le disque est prêt à effectuer des opérations IO, contrairement à ce que la spécification demandait. Par conséquent, la fonction poll qui attend que ce bit soit allumé pour continuer à effectuer les opérations de lecture / écriture a été modifiée pour fonctionner sous *Bochs*.

#### 2.2 Côté interface utilisateur

Les fonctions openfile, read, write et close de l'interface POSIX ont étés implémentées, avec la syntaxe et les flags présentés dans la bibliothèque Unix de OCaml. Les permissions ne sont pas vérifiées, seul le mode d'ouverture (O\_RDONLY, O\_WRONLY, O\_APPEND, etc...) importe.

Les fonctions du *shell* ls, mkdir, rm sont aussi implémentées. rm peut supprimer récursivement des dossiers sans risque de débordement de pile pour une profondeur quelconque de sous-dossiers.

#### 2.3 Implémentation et difficultés rencontrées

La structure des inodes et des blocs est dictée par le standard de EXT2. Les champs de ces structures sont maintenus à jour de façon à ce que le disque puisse être monté sur un autre système d'exploitation et refléter les opérations réalisées sous MarioBrOS, à l'exception des champs concernant les dernières dates de modification, création, etc. qui ne sont pas actualisés.

Les données des fichiers en EXT2 sont contenues dans des blocs, pointés par l'inode soit directement (douze direct block pointers), soit par une simple indirection (un simple indirect block pointer qui pointe vers un bloc d'adresses de blocs de données), soit par une double indirection, soit par une triple. L'implémentation permet en théorie de manipuler tous les fichiers de ce type, mais du fait de la taille du disque employé (64 Mo), nous n'avons pas pu tester la manipulation effective de gros fichiers.

Les file descriptors (fd) ont été conçus pour répondre à trois contraintes : pouvoir effectuer la manipulation effective de fichiers, ne pas être réutilisables après avoir été fermés et ne pas être limités en nombre (autrement que par la taille de la mémoire physique).

Pour cela, l'OS dispose d'une file descriptor table (fdt) qui est un tableau dynamique de fd. Chaque fd est lui-même un pointeur alloué par l'OS vers un entier, cet entier servant d'index dans la fdt. Chaque entrée de la fdt est constituée d'un inode, de la taille du fichier, de la position du curseur du fd et de la valeur du fd pointant sur l'index de cette entrée.

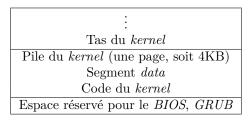
Pour dilater la fdt, il suffit d'allouer un bloc du double de sa taille et de recopier dedans la fdt actuelle. Pour la contracter, ce qui se produit lorsqu'elle est occupée pour moins d'un quart, un nouveau bloc est alloué de taille moitié de la fdt actuelle, et les entrées sont recopiées dans la nouvelle. Si une entrée doit changer d'index, le fd correspondant (qui est un de ses champs) est modifié en changeant l'index sur lequel il pointe.

Enfin, à la fermeture d'un fd, celui-ci est libéré de la mémoire et la valeur du champ fd correspondant à son index dans la fdt est mis à 0, qui ne correspond à aucun pointeur valide, et donc à aucun fd valide. À chaque appel à une fonction comme read, write, etc., la correspondance entre le fd donné et le champ fd dans la fdt correspondant à son index est vérifiée, et la fonction considère le fd invalide si ce n'est pas le cas. Cette sécurité n'est cependant pas complète : si le pointeur fd est ré-alloué par la suite par openfile, il pourra alors resservir.

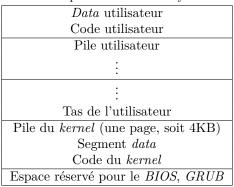
#### 3 Gestion de la mémoire

#### 3.1 Organisation de la mémoire

La mémoire (virtuelle) se décompose comme suit :



1.1 Espace mémoire du noyau



1.2 Espace mémoire de l'utilisateur

Table 1 – Organisation de la mémoire virtuelle

La segmentation est utilisée en modèle *flat*, c'est-à-dire que tous les segments vont de 0 à 4GB. Celle-ci ne sert qu'à gérer le *user-mode*.

#### 3.2 Paging

En ce qui concerne le *paging*, tout de 1MB à la fin de la pile du *kernel* est *identity-mapped* afin de pouvoir activer le *paging* de manière transparente (et ce qui permet aux processus de pouvoir exécuter du code noyau de manière transparente également), la page 0 n'étant pas *mappée* pour détecter les erreurs.

Lorsqu'un utilisateur demande l'accès à une page virtuelle, on cherche la première page physique libre et on *mappe* la première vers la dernière. Ainsi, cela permet au tas de grandir automatiquement. Il en va de même pour la pile de l'utilisateur : lorsqu'il effectue un *Page Fault* pour cause de *Stack Overflow*, on peut *mapper* la page manquante et recommencer (désactivé pour faciliter le *debug*).

Il est également possible pour le *kernel* de demander l'accès à une page physique, ce qui est souvent utile : par exemple, pour charger le code d'un processus en mémoire (alors que l'on est dans le *page directory* du *kernel*), on demande l'accès aux pages physiques correspondant aux adresses virtuelles Oxffff0000 - Oxfffffff du processus (le code utilisateur étant *linké* à l'adresse Oxffff0000) et on y copie alors le code voulu.

#### 3.3 Tas

Le tas (mem\_alloc et mem\_free) suit une structure de blocs standard. Initialement, il s'agit d'un seul bloc libre de la taille d'une page, et lorque l'on demande des tailles plus grandes, celui s'agrandit en demandant l'accès aux pages virtuelles contigües.

Les blocs libres sont doublement chaînés entre eux pour faciliter l'allocation. Lors de la libération d'un bloc, celui-ci est éventuellement fusionné avec ses voisins (physiques) si ceux-ci sont libres, afin d'éviter la fragmentation de la mémoire.

```
/* Unique end header for all blocks */
^{21}
   typedef struct end_header
22
23
      size_t size : 31; /* Shifted right one bit */
24
     bool used : 1; /* 0 for free, 1 for used, strongest bit because of little-endianness */
25
   } __attribute__((packed)) end_header_t;
26
27
   /* Used block have a first header identical to the end one */
28
   typedef end_header_t header_used_t;
   /* Free blocks also have the address of the adjacent free blocks in the doubly linked list */
31
   typedef struct header_free header_free_t;
32
   struct header_free
33
34
      size_t size : 31;
35
      bool used : 1;
36
                                 /* Pointer to the previous free block in the doubly chained list
      header_free_t *prev;
37
      → */
                                  /* Pointer to the next free block in the doubly chained list */
      header_free_t *next;
38
    } __attribute__((packed));
39
40
    /* Structure of a free block (size always comprises headers):
41
    * - header_free_t
42
     * - free memory (unspecified)
43
     * - end_header_t
44
     */
45
46
   /* Structure of an used block (size always comprises headers and padding, big enough to be
47
    \hookrightarrow freed):
     * - header_used_t
48
    * - maybe some padding bytes, set to 0 (always different than the header used since its last
    \hookrightarrow bit is 1)
    * - used memory (domain of the user)
50
    * - end_header_t
51
    */
52
```

LISTING 2 – Structure du tas

# 4 Multitasking

#### 4.1 Processus

La structure suivie est similaire à celle du piconoyau présenté lors du premier cours, en ceci que l'on retient pour chaque processus ses registres, son page directory et l'état de son tas.

Un ordonnanceur va alors sélectionner les processus selon leur priorité, à chaque tick de l'horloge (interruption *timer*), ou après un appel système qui laisse le processus courant en attente.

On dispose de l'appel système hlt (effectué par le processus idle en continu) qui permet au système d'être toujours sensible aux interruptions. Pour cela, on ignore les interruptions *timer* lorsque l'on est en train de répondre à un appel système, tandis que tout autre interruption provoque l'arrêt de la boucle d'attente et le retour à l'ordonnanceur, qui sélectionne un nouveau processus, etc....

#### 4.2 Shell

À l'heure où nous mettons sous presse, le *shell* n'est pas encore un processus utilisateur à part entière. Nous espérons cependant qu'il en sera un le jour de la présentation.

Le *shell* premet d'éxécuter des commandes (la liste étant accessible avec help), dispose d'un curseur (qui n'est pas visible depuis que l'on *boot* sur le disque pour une raison inconnue) et d'un historique. En tant que processus, toutes les commandes ou presque utilisent des appels systèmes.

#### 4.3 La commande run

Cette commande permet de lancer un exécutable ELF situé dans le dossier progs (run foo va ainsi lancer le programme progs/foo.elf). Ces exécutables correspondent à un ficher C situé dans progs/src qui a été compilé à l'aide d'une bibliothèque pour pouvoir utiliser les appels systèmes.

```
progs/src/lib.s[10-21]
    global fork
10
11
    fork:
12
      push ebx
13
      push edi
      mov eax, 1
14
      mov ebx, [esp+12]
15
      int 0x80
16
      mov edi, [esp+16]
17
      mov [edi], ebx
18
      pop edi
19
      pop ebx
20
      ret
```

Listing 3 – Exemple d'appel système (côté utilisateur)

# 5 Bugs et améliorations possibles

Il y a certainement quelques fuites de mémoire de temps à autre, notamment au niveau du *shell* (c'est plus faute de temps qu'autre chose). De plus, plus de gestion des erreurs (mauvaises commandes, manque de mémoire, . . . ) serait nécessaire.

Pouvoir exécuter le *shell* comme un processus aurait été un énorme plus, avec des *buffers* d'entrée, de sortie, d'erreur et les opérateurs pipe, les indirections...

Les appels systèmes new\_channel, send et receive sont codés (depuis le piconoyau) mais pas portés dans *MarioBrOS*, encore une fois par manque de temps.

Enfin, dans l'état actuel des choses, appeler la commander run ne rend pas la main au shell... C'est bizarre que la première commande marche sans la seconde, mais c'est un problème qui serait réglé par le fait que le shell soit un processus à part enitère.