# *BlenderOS*

Pierre DONAT-BOUILLUD

1er août 2010

## Introduction

*BlenderOS* est un système d'exploitation conçu spécialement pour exécuter Blender, logiciel open-source de création, édition d'images de synthèse, dans des conditions optimales, d'abord son moteur de rendu, et plus généralement les logiciels mettant en jeu des calculs portant sur le graphisme. Il est ainsi conçu pour une rapide vitesse d'exécution, et pour un support natif du calcul distribué. *BlenderOS* peut se voir ainsi ajouté des capacités de calcul ou de connectique à l'aide de système clients qui peuvent être installés sur d'autres systèmes d'exploitation, ou former avec plusieurs autres instances distantes une grappe formant une entité unique. Il est ainsi le terrain d'expérimentation des algorithmes les plus sophistiqués, et, on le souhaite, les plus performants.

Il est destiné à fonctionner sur des architectures 32 ou 64 bits, d'abord sur processeur Intel. L'implémentation de l'utilisation des ressources de la carte graphique à des fins de calculs pas forcément graphiques est envisagée au niveau du noyau. le système de clients de calcul permettra aussi d'accéder à des périphériques distants, ce qui assurera une compatibilité importante avec le matériel, tandis que les pilotes des périphériques cruciaux seront réécris pour une meilleure interaction avec Blender.

*BlenderOS* est aussi destiné à être lancé d'un cédérom, ou mieux, d'une clé USB, même si une installation sur disque est parfaitement possible, et même conseillé dans le cas d'une utilisation prolongée.

*BlenderOS* est inspiré, au moins dans quelques mécanismes bas niveau comme le chargement du noyau, par le noyau Pépin, dont le site à l'adresse suivante `http://a.michelizza.free.fr/pmwiki.php?n=TutoOS.TutoOS` décrit les étapes menant à un système basique, mais fonctionnel.

# Première partie
# Description

## 1 Architecture globale

Pour de meilleurs performances, *BlenderOS* utilise un noyau monolithique. *BlenderOS* désignera dorénavant tout aussi bien le noyau que le système d'exploitation, sans autre précision que l'apport éclairant du contexte.

## 2 Le chargement du noyau

Le noyau répond au standard *multiboot*. Il peut donc être lancé grâce à tout chargeur répondant à ce type de standard. *Grub* est le chargeur privilégié.

### 2.1 Le standard *multiboot*

Pour répondre à ce standard, le noyau doit être pourvu de deux caractéristiques :
- être un fichier exécutable 32 ou 64 bits standard
- l'image du noyau doit contenir un entête spécial, le *multiboot* header.

#### Le *multiboot* header

```
  global _start, start
extern kmain

%define MULTIBOOT_HEADER_MAGIC  0x1BADB002
%define MULTIBOOT_HEADER_FLAGS  0x00000003
%define CHECKSUM -(MULTIBOOT_HEADER_MAGIC + MULTIBOOT_HEADER_FLAGS)


_start:
        jmp start

; The Multiboot header
align 4
multiboot_header:
dd MULTIBOOT_HEADER_MAGIC
dd MULTIBOOT_HEADER_FLAGS
dd CHECKSUM
; ----- Multiboot Header Ends Here -----

start:
```

```
        push ebx
        call kmain

        cli ; stop interrupts
        hlt ; halt the CPU
```

Le *multiboot* header résultant doit être placé au début du noyau grâce à une option appropriée du linker.

*Grub* peut aussi fournir au noyau la quantité de mémoire utilisée.

La commande *mbchk* de vérifier le conformité du noyau avec le standard *multiboot* :

```
 \$ mbchk kernel
kernel: The Multiboot header is found at the offset 4104.
kernel: Page alignment is turned on.
kernel: Memory information is turned on.
kernel: Address fields is turned off.
kernel: All checks passed.
```

# 3 La gestion de la mémoire

La gestion de la mémoire est une fonction cruciale d'un système d'exploitation. Elle doit répondre à deux problématiques :

**être performante :** l'allocation de mémoire est une opération fondamentale, effectuée très régulièrement par l'ensemble des programmes exécutés sur le système

**éviter la fragmentation :** la fragmentation apparaît lorsque des blocs de mémoire sont libérés mais ne sont pas utilisés car trop petits pour les demandes actuelles. Il en résulte un progressif gaspillage d'espace, et une recherche plus lente d'espace libre.

Une autre problématique, mineure, est de faire en sorte que la gestion de la mémoire ne vampirise pas toute la mémoire disponible, que les structures de gestion soient raisonnablement négligeables face à l'ensemble de la mémoire. Une autre est de permettre un accés séquentiel rapide à la mémoire.

## 3.1 *Pré-requis :* connaître la taille de la mémoire disponible

*BlenderOS* répond au standard *multiboot* : le *bootloader* que celui-ci met à disposition peut passer au noyau la taille de la mémoire disponible, qu'il a lui-même obtenu du *BIOS*. Ainsi, *Grub* transmet au noyau via la structure `struct mb_partial_info` des informations, en particulier sur la mémoire.

```
1 struct mb_partial_info {
2         unsigned long flags;
3         unsigned long low_mem;
4         unsigned long high_mem;
5         unsigned long boot_device;
6         unsigned long cmdline;
7 };
```

La structure complète contient d'autres informations telles que la carte de description de la mémoire initialisée par le *BIOS*. Ces informations peuvent alors être utilisées ainsi :

```
1 void kmain(struct mb_partial_info *mbi)
2 {
3         printk("Grub example kernel is loaded...\n");
4         printk("RAM detected : %uk (lower), %uk (upper)\n", mbi->low_mem, mbi->high_mem);
5         printk("Done.\n");
6 }
```

**TODO :** préciser le système de gestion utilisée : les adresses mémoires concrètes...

## 3.2 Gestion de la mémoire physique

La mémoire est découpée en page de taille *PAGESIZE*. L'allocation d'une page consiste à la marquée comme utilisée *1*, ou libre *0*, d'une façon comme d'une autre. Parmi différentes structures de données, comme une pile, ou une liste chainée, *BlenderOS* utilise un système hybride formé d'une bitmap et d'un tas, tas en ce qu'il est stocké linéairement et qu'il est ordonné selon des adresses implicites. Chacun des bits de la bitmap représente l'état d'une page, de façon contiguë. Cependant, un tel système se heurte au temps de recherche d'une page libre. En effet, on parcoure séquentiellement la bitmap jusqu'à trouver une page libre ; aussi dans le pire des cas, le temps de recherche est de $O(p)$ où $p$ est le nombre de pages du système.

Pour remédier à cette lenteur, la bitmap est divisée en blocs de taille $a$, référencés à l'aide d'un tas (arborescence complète) de hauteur $h$ tel que chacun de ses nœuds indiquent si les nœuds enfants comportent des blocs utilisés. Les feuilles du tas sont des pointeurs vers les différents blocs de la bitmap ; elles correspondent aux différents niveaux du tas. On peut donc accéder à un bloc de bits dès la racine, pas forcément au niveau de la feuille.

Pour accélérer plus encore le processus, on effectue un parcours non plus bit à bit, mais mot par mot, où un mot correspond à la taille du plus grand registre du microprocesseur, 32 bits, ou 64 bits. Si ce mot, en 32 bits, vaut 0xFFFFFFFF, c'est que ces 32 bits correspondent à 32 pages utilisées. Dans le cas contraire, on parcoure ces 32 bits à la recherche des bits à 0.

Par ailleurs, il est probable, au moins au début avant fragmentation, que la page suivant la page couramment allouée soit une page libre. On garde donc

un pointeur sur la page couramment allouée, et on teste avant tout parcours dans le tas si cette page suivante est libre. Cela ne rajoute qu'une seule opération. Cette heuristique accélère en particulier l'allocation d'une série de pages lors de demandes groupées par un programme.

De même, chaque premier bit libre de chaque bloc est indiqué par un pointeur, à NULL s'il n'y en a pas, mis à jour à chaque opération d'allocation ou suite groupée d'allocations.

Ainsi, si dans le pire des cas, une allocation s'effectue en un temps $O(\ln p)$, dans le meilleur des cas, elle s'effectue en $O(1)$.

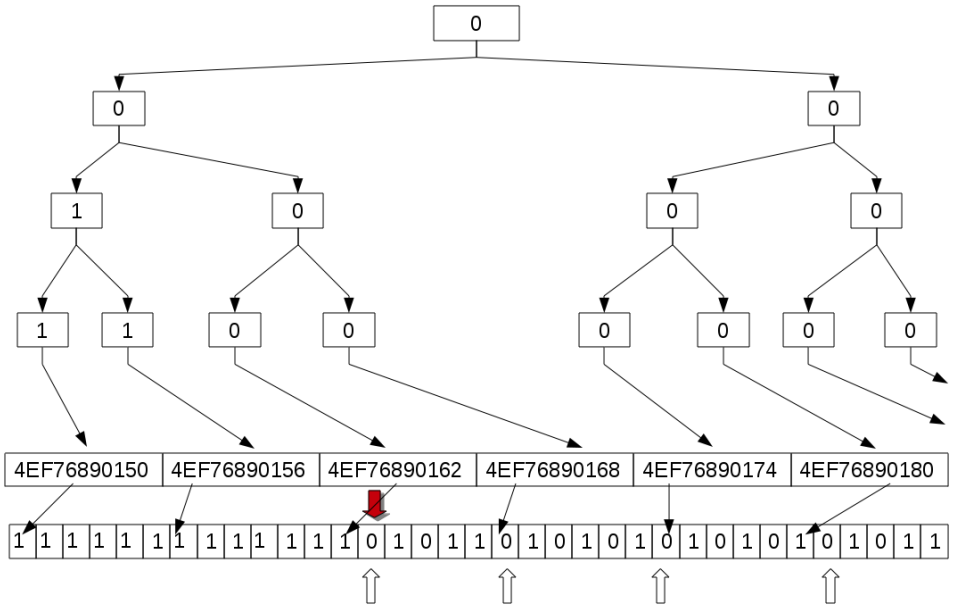


FIGURE 1 – La structure du gestionnaire de mémoire physique

**Une structure à construire de façon optimale** On cherche à minimiser le temps de recherche d'une page libre. On appelle $h$ la hauteur de l'arborescence complète, $k$ le nombre d'enfants d'un nœuds, et $a$ la longueur d'un bloc, c'est-à-dire le nombre de pages qu'il peut décrire ; $n$ représente la taille de la mémoire physique, $t$, la taille d'une page.

Calculons le nombre d'opérations que nécessite une recherche dans le pire des cas. L'arborescence $k$-aire de hauteur $h$ étant complète, elle compte $N_k =$

$\frac{k^h-1}{k-1}$ nœuds intérieurs et $N_h$ feuilles, soit autant de blocs, où :

$$N_h = k^h \tag{1}$$

Le nombre de pages $p$ total est :

$$p = \frac{n}{t} \tag{2}$$

D'autre part, on peut exprimer ainsi le nombre total de pages :

$$p = N_h \times a = k^h \times a \tag{3}$$

On en déduit l'égalité suivante :

$$a \times k^h = \frac{n}{t} \Leftrightarrow a = \frac{n}{t \times k^h} \tag{4}$$

Le pire des cas n'advient pas lorsque la page suivant la page précédemment allouée est libre. On parcoure alors tous les enfants d'un nœud jusqu'à arriver à chaque fois au dernier enfant, puis arrivé au bloc, on le parcoure jusqu'à la fin.

**TODO :** revoir l'étude avec le pointeur de premier bit libre par bloc. -> Abandonner ces pointeurs s'il n'y a que peu de blocs comme pour k=2.

La fonction qui modélise le nombre d'opérations dans le pire des cas est alors, en comptant la racine :

$$\Theta(h,k) = k \times h + a - 1 = k \times h + \frac{n}{t \times k^h} + 1 \tag{5}$$

de dérivée partielle par rapport à $h$ :

$$\frac{\partial \Theta}{\partial h} = k - \frac{n \times \ln k \times k^{-h}}{t}$$

$\Theta$ atteint alors un minimum en :

$$h = \ln_k \frac{n \times \ln k}{t \times k} \tag{6}$$

Optimisons désormais :

$$\Xi(k) = \Theta\left(\ln_k \frac{n \times \ln k}{t \times k}, k\right) = \frac{k}{\ln k} \times \left(\ln \frac{n \times \ln k}{t \times k} + 1\right)$$

La dérivée de $\Xi$ vaut :

$$\Xi'(k) = \frac{\ln k - 1}{\ln^2 k} \times \ln \frac{n \times \ln k}{t \times k} \tag{7}$$

Elle s'annule en $k = e \simeq 2.71$, le degré de l'arborescence étant une valeur entière, on peut choisir ou un arbre binaire, ou un arbre ternaire. L'arbre ternaire aura le degré le plus proche, le binaire, les opérations de navigation dans l'arbre les plus rapides, car obtenues par de simples décalages de bits et de *et* binaires. $\Xi$ s'annule aussi lorsque $g : k \mapsto \frac{\ln k}{k}$ atteint $\frac{t}{n}$. Or $g$ a pour maximum $\frac{1}{e}$ en $e$. On doit donc avoir pour que ce membre s'annule :

$$\frac{t}{n} \leqslant \frac{1}{e} \tag{8}$$

c'est-à-dire que :

$$\frac{t}{n} \lesssim 0.36\ \% \tag{9}$$

On appelle rapport critique de pagination $\mu_P$ ce rapport. Dans la majorité des cas, la taille d'une page est bien inférieure à la taille totale de la mémoire, et toujours en situation réelle, elle est inférieure à $\mu_P$. Par exemple, dans l'architecture x86, en 32 bits, la taille d'une page est habituellement de 4 Ko, alors que la taille de la mémoire peut atteindre les 4 Go, et dépasse toujours aujourd'hui pour les ordinateurs personnels les 512 Mo.

**Choix de $a$ et de $h$** On utilise les équations déjà déterminées :

$$h = \ln_k \frac{n \times \ln k}{t \times k} \tag{10}$$

$$a = \frac{k}{\ln k} \tag{11}$$

Par exemple, pour $k = 2$, $n = 4$ Go, et $t = 4$ Ko, on a $h \simeq 18.47$, on choisit $h = 18$, et $a \simeq 2.88$. On choisit $a = 3$. En tout, on obtient $\Theta(18, 3) = 39$ opérations dans le pire des cas. Pour $k = 3$, avec la même architecture de mémoire, $h = 12$, et $a = 3$, soit $\Theta(12, 3) = 39$. Il apparait donc plus tenant de choisir $k = 2$.

**TODO :** étude dans le cas moyen : considérer qu'une page est trouvée libre à peu près à la moitié d'un bloc. (en choisissant un politique first-ordered fit, c'est plus que probable : trouver une meilleure distribution -> exponentielle, de Poisson ? ?). Implémenter les deux stratégies, comparer les temps d'exécutions sur un programme réel.

**Allouer le gestionnaire de mémoire physique** Le gestionnaire de mémoire physique ne peut s'allouer lui-même : on retombe dans l'éternel problème du premier à voir le jour entre l'œuf et la poule ! Il est donc directement placé en mémoire, à un emplacement stable, juste derrière le noyau.

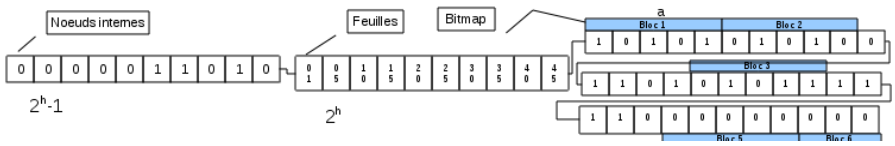


FIGURE 2 – L'emprise mémoire du gestionnaire de mémoire physique

### 3.3 Gestion de la mémoire virtuelle

### 3.4 *Le cœur, crucial, de la gestion de l'allocation côté utilisateur : malloc* et *free*

### 3.5 *Gestion du manque de mémoire :* mémoire swap

# Deuxième partie
# Installation à partir des sources

L'installation de *BlenderOS* se fait à partir des sources, qu'il faudra compiler. Il faut ensuite créer une image où le noyau et grub sont correctement placés. On se propose de détailler les étapes menant à une installation propre, efficace, et durable de *BlenderOS* à partir d'un système Linux. Un script permet aux néophytes d'accélérer le processus, au détriment des enseignements conséquents qui pourraient résulter d'une installation à la main.

## 4 Compilation des sources

Positionner-vous dans le répertoire contenant les sources. Un simple `make` permet de les compiler. Vous pouvez sélectionnez l'architecture à laquelle vous destinez votre futur système : 32 bits ou 64 bits.

# 5 Installation de grub et du noyau de *BlenderOS*

# 6 Installation des programmes complémentaires indispensables à l'administration et à l'exploitation du système

# A Le standard *multiboot*

Cette annexe est une revue condensée et traduite en français de la spécification *multiboot*. La spécification *multiboot* veut unifier le chargement du noyau par une bootloader du point de vue du noyau lui-même : elle définit un header qui permet entre autres au noyau d'indiquer sa présence et l'adresse où il veut être chargé, un état dans lequel la machine doit se trouver lorsque le noyau est chargé, et enfin une structure passant de précieuses informations au noyau.

*BlenderOS* suit la spécification multiboot, et peut donc être lancé par *Grub*.

## A.1 Le *multiboot* header

## A.2 Etat de la machine

Le registre EAX contient la valeur magique indiquant que le bootloader répond au standard *multiboot*.

## A.3 La structure *multiboot* information

L'adresse de la structure d'informations sur le boot passée par le bootloader est stockée dans le registre EBX. Elle peut se trouver n'importe où dans la mémoire.

| Offset | Contenu | Présence |
|---|---|---|
| O | flags | requis |
| 4 | mem_lower | présent si flags[0] est activé |
| 8 | mem_upper | présent si flags[0] est activé |
| 12 | boot_device | présent si flags[1] est activé |
| 16 | cmdline | présent si flags[2] est activé |
| 20 | mods_count | présent si flags[3] est activé |
| 24 | mods_addr | présent si flags[3] est activé |
| 28-40 | syms | présent si flags[4] ou flags[5] est activé |
| 44 | mmap_length | présent si flags[6] est activé |
| 48 | mmap_addr | présent si flags[6] est activé |
| 52 | drives_length | présent si flags[7] est activé |
| 56 | drives_adrr | présent si flags[7] est activé |
| 60 | config_table | présent si flags[8] est activé |
| 64 | bootloader_name | présent si flags[9] est activé |
| 68 | apm_table | présent si flags[10] est activé |
| 72 | vbe_control_info | présent si flags[11] est activé |
| 76 | vbe_mode_info | idem |
| 80 | vbe_mode | idem |
| 82 | vbe_interface_seq | idem |
| 84 | vbe_interface_off | idem |
| 86 | vbe_interface_len | idem |

TABLE 1 – Format de la structure d'information *multiboot*