Croisière au cœur d'un OS*
Étape "9 et demi" : Pilotes de périphériques *bloc*

Résumé

Le mois dernier, nous nous étions intéressés aux périphériques spéciaux de type *caractère*. Nous vous proposons maintenant de présenter la seconde grande classe de périphériques : les périphériques *bloc*. Nous détaillerons leur principe de fonctionnement, l'implémentation de pilotes pour ces périphériques ainsi que leur intégration dans SOS. Cela aboutira à un pilote de disque dur de type *IDE* fonctionnel et utilisable depuis les applications utilisateur.

Introduction

Depuis le mois dernier, SOS est doté d'un certain nombre de pilotes de périphériques caractère permettant d'interagir avec le système au travers de la console *clavier/écran* ou au travers de la ligne série. Toutefois, SOS ne dispose toujours pas de pilote pour des périphériques de type disque dur qui permettraient de sauvegarder durablement des informations, et donc d'héberger un système de fichiers.

Les disques durs font partie de la catégorie des périphériques dits *bloc*. Ceux-ci sont caractérisés par une taille finie et des accès "bloc par bloc". Un *bloc* correspond à une zone du périphérique de taille fixée qui dépend du périphérique et/ou du système de fichiers. La taille typique d'un bloc est 512 octets. Il est possible de se déplacer dans ces périphériques afin d'accéder aux différents blocs, dans la limite de la taille du périphérique.

Comme les pilotes de périphériques caractère, les pilotes de périphériques bloc sont pris en charge par SOS au travers d'un sous-système spécifique : *blockdev*. Ce sous-système, décrit en section 1, s'occupe de l'intégration des pilotes de périphériques bloc dans le *VFS* en permettant en particulier d'accéder aux périphériques octet par octet plutôt que bloc par bloc. Á cette fin, ainsi que pour accélérer les transferts et assurer certaines propriétés de cohérence que nous préciserons par la suite, *blockdev* gère un système de caches : les *caches des blocs* et les *caches des pages*. Nous détaillerons tout cela en section 1.2 et suivantes.

Nous présenterons en section 2 un premier pilote de périphérique bloc, gérant les disques durs de type IDE. La section 3 présentera l'implémentation d'un pilote pour les partitions d'un disque dur. Nous terminerons par une petite démonstration utilisant ces périphériques bloc.

Le code source de cet article est disponible comme d'habitude sur le site de SOS : `http://sos.enix.org/`.

*La version originale de cet article a été publiée dans GNU Linux Magazine France numéro 80 – Février 2006 (`http://www.linuxmag-france.org`) et cette version est diffusée avec l'autorisation de l'éditeur.

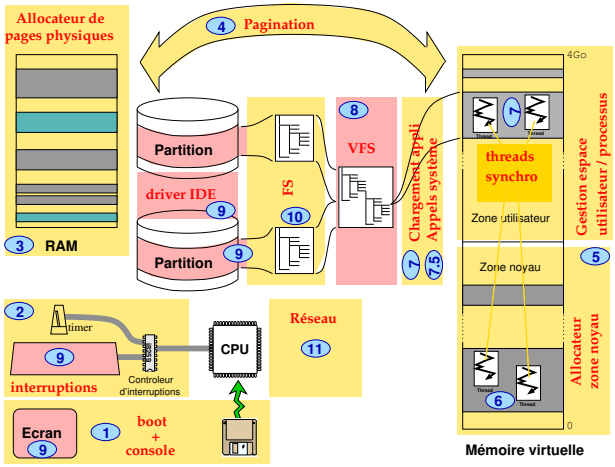

FIG. 1 – Programme des articles

1 Pilotes de périphériques bloc

Intéressons-nous tout d'abord à la façon dont les pilotes de type bloc fonctionnent et à la façon dont ils sont intégrés dans le *VFS*.

1.1 Description d'un pilote de périphérique bloc

Les pilotes de périphériques *bloc* doivent effectuer des actions assez similaires à celles de *chardev*, le sous-système gérant les pilotes de périphériques caractère décrit le mois dernier. Ils utilisent l'interface décrite dans le fichier `sos/blkdev.h`. Ici, les pilotes de périphériques doivent enregistrer une structure de type `sos_blockdev_operations` pour chaque instance de périphérique, et non plus pour une classe de périphérique entière comme c'est le cas pour *chardev*. Cette structure contient notamment les méthodes suivantes :

- **read_block**, une opération obligatoire qui permet de lire un bloc depuis le périphérique ;
- **write_block**, une opération optionnelle qui permet d'écrire un bloc sur le périphérique ;
- **ioctl**, une opération optionnelle qui permet d'accéder à des fonctionnalités spécifiques au périphérique. Par exemple, pour un disque dur IDE, une opération **ioctl** pourrait permettre d'activer ou désactiver le *DMA*.

Lorsqu'on implémente un pilote de périphérique bloc, on doit donc fabriquer un objet du type précédent et définir au moins sa méthode **read_block**. On doit ensuite enregistrer cette structure auprès de *blockdev* comme nous le verrons en section 1.6, en précisant en particulier le nombre de blocs dans le périphérique. Et c'est tout ! La couche *blockdev* s'occupe de tout le reste.

1.2 Systèmes de caches de blocs et de pages

Les périphériques bloc se prêtent bien à l'utilisation de caches car il est possible d'identifier complètement chaque octet du périphérique par un couple {`numéro de bloc, déplacement dans le bloc`}. On a donc intérêt à stocker en mémoire physique (RAM) les blocs les plus fréquemment utilisés pour accélérer l'accès aux données. En effet, le temps d'accès aux données stockées sur les périphériques de type disque dur demeure de plusieurs ordres de grandeur supérieur au temps d'accès à la mémoire physique. En réalité, passer par un cache mémoire ne constitue pas un luxe superflu, c'est un minimum nécessaire. En effet, comme *blockdev* doit satisfaire aux exigences du *VFS*, il doit émuler un accès octet par octet alors que le périphérique qu'il contrôle ne travaille qu'avec des blocs. Il est donc nécessaire de stocker en mémoire des blocs complets (au moins un) pour pouvoir accéder à chacun de leurs octets.

D'autre part, la sémantique Unix traditionnelle veut que, pour les fichiers comme pour les périphériques bloc, il y ait cohérence entre les données manipulées par `read()/write()` et les données projetées en mémoire par appels à `mmap()` en mode *partagé* (*shared mappings*). Ce dernier point signifie tout simplement que si on projette en mémoire en mode *partagé* le bloc 8 d'un périphérique à l'adresse `0x1234000`, alors si on écrit `"42"` à cette adresse en mémoire, un `read()` au début du bloc 8 devra permettre de lire `"42"`. Et inversement, si on fait un `write()` de `"24"` au début du bloc 8, on doit relire `"24"` à l'adresse `0x1234000` en mémoire. Par défaut, rien ne garantit que cette sémantique sera respectée avec les périphériques caractère. Ce sera au pilote de périphérique de s'en occuper si elle est désirable. À l'inverse, pour les périphériques bloc elle sera respectée et c'est le sous-système *blockdev* qui s'en chargera. Le pilote du périphérique *bloc* n'aura pas à s'en soucier.

Ces deux types d'utilisation de la mémoire par le sous-système *blockdev* prennent la forme de deux *caches* dans SOS : le "cache de blocs" et le "cache de pages" respectivement. La figure 2 présente l'architecture globale du sous-système *blockdev*, des caches et leurs relations avec le pilote de périphérique sous-jacent.

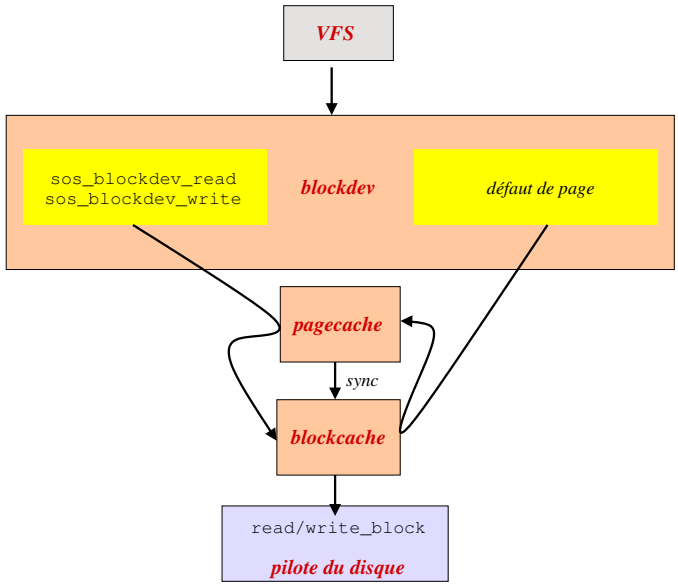

FIG. 2 – Interactions entre les sous-systèmes liés à la gestion des périphériques bloc.

1.3 Cache de blocs

Le cache de blocs est implémenté dans `sos/blkcache.c`. Le sous-système *blockdev* fait appel aux fonctions de ce sous-système *blockcache*, il n'accède pas directement au périphérique *via* les méthodes `read_block()/write_block()` de `struct sos_blockdev_operations`. C'est *blockcache* uniquement qui s'en chargera.

1.3.1 Principe

Dans notre cas, un cache de blocs est associé à chaque périphérique bloc de type disque. Il possède une taille fixée à sa création et est régi par une "politique" classique de type "*write-back*". Cette politique signifie qu'en cas de modification en mémoire d'un bloc, ce dernier ne sera pas immédiatement transféré sur disque (cas de la politique "*write-through*"). Il sera écrit sur disque le plus tard possible, c'est-à-dire seulement lorsqu'il devra être supprimé du cache pour laisser la place à d'autres blocs, ou par une fonction de resynchronisation forcée appelée régulièrement par exemple (équivalent au *thread [bp]dflush* de Linux). Cela aura pour effet de réduire le nombre d'écritures sur disque : un bloc modifié 10 fois avant d'être évincé du cache ne sera écrit qu'une seule fois sur disque.

D'autre part, pour limiter encore le nombre d'écritures sur disque effectuées, on utilise un mécanisme de marquage permettant de savoir si un bloc a été modifié en mémoire. Un tel bloc est dit "*sale*" (ou *dirty*). Un bloc non modifié en mémoire puis évincé du cache n'aura donc pas besoin d'être écrit sur disque puisque son contenu correspond encore à celui du disque.

1.3.2 Implémentation

Initialisation. La fonction `sos_blkcache_new_cache()` créé un cache de blocs, c'est-à-dire une structure de type `sos_block_cache` décrite plus bas. Cette fonction sera appelée par la couche *blockdev* lorsqu'un nouveau pilote de disque sera enregistré dans l'OS. On lui fournit la liste des opérations du pilote de périphérique (structure `sos_blockdev_operations` vue plus haut) ainsi que le nombre de blocs que le cache devra stocker en mémoire. L'espace nécessaire pour stocker les blocs sera immédiatement alloué en mémoire.

Préparation de l'accès à un bloc. La fonction `sos_blkcache_retrieve_block()` récupère un bloc du cache, ou plutôt l'adresse en mémoire où est stocké le bloc du disque demandé. C'est la fonction principale de *blockcache*, elle sera appelée par *blockdev* aussi bien pour les opérations de lecture que d'écriture sur le périphérique. Une fois qu'elle aura été appelée, on pourra lire ou modifier le contenu du bloc directement en mémoire. Les arguments de `sos_blkcache_retrieve_block()` sont le numéro du bloc auquel on désire accéder ainsi que le type d'accès (lecture seule, lecture/écriture, écriture seule). Le fonctionnement de cette fonction est le suivant (voir la figure 3) :

– Si le bloc demandé était déjà en mémoire, alors on le récupère immédiatement ;
– Sinon, le bloc n'est pas encore présent dans le cache. D'abord on trouve un emplacement disponible dans le cache s'il en existe, ou sinon on évince le bloc le plus anciennement utilisé du cache. Cette *politique de remplacement de cache* est dite "de type *LRU*" (*Least Recently Used*). L'éviction peut être accompagnée d'une écriture sur disque si le bloc évincé est sale, comme nous l'avons indiqué au début. Ensuite, de deux choses l'une :
  – On souhaite accéder au bloc en lecture ou en lecture/écriture : il faut récupérer le contenu du bloc depuis le disque à l'aide de la méthode `read_block()` du pilote de périphérique ;
  – On souhaite accéder au bloc en écriture seule : nul besoin de le récupérer depuis le disque puisque son contenu est appelé à être écrasé. Attention, cela signifie que la totalité du bloc devra être écrite en mémoire, sinon la partie du bloc qui ne sera pas écrite en mémoire sera indéterminée et ne reflétera pas le contenu de ce qui est sur disque.

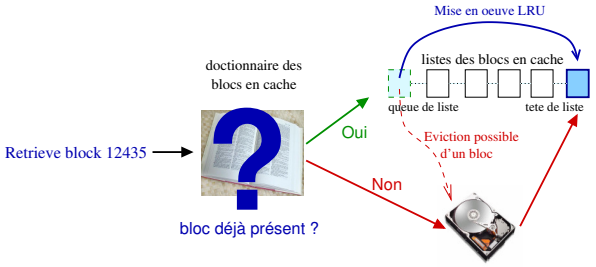

FIG. 3 – Principe de `sos_blkcache_retrieve_block()` : le bloc 12345 est demandé. Il est récupéré (carré bleu) soit à partir du cache quand il était déjà présent, soit à partir du disque. Les trois listes en jeu dans ce mécanisme ne sont pas détaillées sur la figure.

Fin de l'accès à un bloc. Une fois les données du bloc demandé récupérées ou modifiées en mémoire par le code de *blockdev*, ce dernier doit signifier au cache qu'il n'y accède plus. C'est le rôle de `sos_blkcache_release_block()`. Cette fonction s'occupe de relâcher le verrou sur le bloc pris par la fonction `sos_blkcache_retrieve_block()`, et éventuellement d'écrire le bloc sur le disque si on le lui demande (argument `force_flush` à `TRUE`). Si on ne lui demande pas, le bloc est inséré dans une liste spéciale (`dirty_list`, voir plus bas) pour être écrit sur disque quand il sera évincé du cache par un appel ultérieur à `sos_blkcache_retrieve_block()`. En fait, pour éviter les écritures inutiles sur disque, on précise à `sos_blkcache_release_block()` si le contenu du bloc en mémoire a été modifié ou pas (argument `is_dirty`). Ainsi, quand un bloc qui n'a pas été modifié en mémoire est évincé du cache, il ne sera pas transféré vers cette liste spéciale et sera simplement supprimé de la mémoire. Dans tous les cas, si écriture (immédiate ou différée au moment d'une éviction) il y a, celle-ci sera effectuée par appel à la méthode `write_block()` du pilote de périphérique bloc.

Resynchronisation forcée. La fonction de resynchronisation forcée `sos_blkcache_flush()` sert à écrire sur disque le contenu de tous les blocs de la liste marqués "sales" (*dirty*). Elle sera appelée par *blockdev* en réponse à une requête de synchronisation par le *VFS via* la fonction `sos_fs_sync()` par exemple.

Fonctionnement interne. Pour toutes ces fonctions, *blockcache* manipule un dictionnaire ainsi que 3 listes d'éléments du type suivant (interne à `sos/blkcache.c`), qui caractérisent un *bloc* :

```
struct sos_block_cache_entry
{
  /** The key of the hash map */
  sos_luoffset_t block_index;

  /** Kernel address where the
     data for the block are stored */
  sos_vaddr_t block_contents;

  /** Is the block available,
      dirty or not ? */
  enum { ENTRY_FREE, ENTRY_SYNC,
         ENTRY_DIRTY } state;

  /** Synchronization */
  struct sos_kmutex lock;

  /** Linkage structure to keep the
      cache block in the hash map */
  struct sos_hash_linkage hlink;

  /** Links to insert the bloc into
      the free/sync/dirty lists */
  struct bkcache_entry *prev, *next;
};
```

Les quatre premiers champs de cette structure ont été évoqués ci-dessus. Les trois autres servent à chaîner les blocs dans le dictionnaire et dans une des trois listes `free`, `sync` ou `dirty`. Le dictionnaire ainsi que les trois listes sont déclarés dans la structure `struct sos_block_cache` allouée par `sos_blkcache_new_cache()` lors de la création du cache :

```
struct sos_block_cache
{
  ...
  /* Dictionary block number -> block address */
  struct sos_hash_table * lookup_table;
  ...
  struct sos_block_cache_entry * free_list;
  struct sos_block_cache_entry * sync_list;
  struct sos_block_cache_entry * dirty_list;
  ...
};
```

Le dictionnaire `lookup_table` est une table de hachage qui établit la correspondance `numéro du bloc` → `adresse du bloc en mémoire` (quand le bloc est déjà dans le cache). Les 3 listes sont :

`dirty_list` : la liste des blocs en mémoire qui ont été modifiés et qui devront donc être écrits sur disque en cas d'éviction du cache ou de demande de *synchronisation* forcée (opération `sos_blkcache_flush()`). Les blocs dans cette liste ont leur champ `state` à `ENTRY_DIRTY`. Ils pourront être transférés vers la liste `sync_list` par `sos_blkcache_flush()` ;

`sync_list` : la liste des blocs en mémoire qui reflètent exactement le contenu du disque, c'est-à-dire qui n'ont pas été modifiés depuis qu'ils ont été transférés à partir du disque ou depuis qu'ils ont été précédemment écrits sur disque. Les blocs dans cette liste ont leur champ `state` à `ENTRY_SYNC`. Ils pourront être transférés vers la liste `dirty_list` par `sos_blkcache_release_block()` ;

`free_list` : la liste des blocs du cache qui ne sont pas encore associés à un bloc du disque. Lorsque le cache vient d'être créé, cette liste rassemble tous les blocs du cache. Elle sera raccourcie au

fur et à mesure du fonctionnement du cache par `sos_blkcache_retrieve_block()` qui transférera ses blocs vers les deux autres listes. Les blocs dans cette liste ont leur champ `state` à `ENTRY_FREE`.

Ces listes sont ordonnées suivant la politique "LRU" : à chaque fois que `sos_blkcache_retrieve_block()` récupère un bloc de ces listes, il le déplace en tête de la liste. Cela a pour effet de reléguer les blocs les moins fréquemment accédés en queue de liste, ils seront évincés du cache en premier en cas de besoin.

Considérations de synchronisation. Les fonctions de *blockcache* ne font ainsi que transférer des éléments d'une liste vers une autre, en faisant éventuellement accès au périphérique *via* appels aux méthodes de `sos_blockdev_operations`. Bien que le principe de fonctionnement de `sos_blkcache_retrieve_block()` décrit précédemment ne fasse que la mise en œuvre de cette idée générale très simple, son implémentation est relativement délicate et mérite qu'on s'y attarde un peu.

Lorsqu'un thread transfère un élément d'une liste vers une autre, il commence par verrouiller le mutex de cette entrée (champ `lock`). Ce verrouillage peut entraîner un bloquage du thread, ce qui signifie qu'un autre thread est en train d'utiliser l'entrée. Cet autre thread peut par exemple transférer le bloc associé à cette entrée vers le disque (ou depuis le disque), ce qui peut avoir pour effet de transférer l'entrée vers une autre liste. Ainsi, quand notre thread sera réveillé et donc qu'il possédera le mutex, l'élément qu'il espérait utiliser peut avoir été déjà transféré vers la bonne liste, ou avoir été transféré vers une autre liste. Pour fonctionner correctement, la fonction `sos_blkcache_retrieve_block()` se comporte donc de la façon suivante. Si on veut travailler avec tel élément E de la liste L :

1. On attend qu'il soit disponible (i.e. on verrouille le `mutex` de l'élément E),
   – *(si le mutex n'est pas immédiatement disponible, le thread est endormi et sera réveillé quand le mutex lui sera confié)*
2. Une fois qu'on possède ce mutex, on vérifie que l'élément est toujours dans l'état qui nous intéressait (i.e. qu'il est bien sur la liste L) :
   – Si oui, alors on est le seul à posséder l'élément, on peut donc l'utiliser sans soucis
   – Si non, alors on relâche le verrou, on re-cherche un élément qui nous intéresse, puis on recommence.

Ainsi, l'algorithme de `sos_blkcache_retrieve_block()` évoqué plus haut s'écrit plus précisément :

```
- DEBUT BOUCLE INFINIE

  # On espère que le bloc est déjà dans le cache
  - SI le bloc recherché est déjà dans le dictionnaire
    - DEBUT ALORS
      - verrouiller le mutex de l'élément trouvé
      - SI l'élément trouvé ne correspond plus au bloc
        qu'on cherchait
        - DEBUT ALORS
          - relâcher le mutex et recommencer la boucle
        - FIN ALORS
        - SINON on a le bloc donc on sort de la fonction !
    - FIN ALORS
  - FIN SI

  # Sinon c'est qu'il faut trouver de la place dans le cache
  # avant d'y transférer le bloc depuis le disque

  - SI la liste ``free'' n'est pas vide
    - DEBUT ALORS
      - choisir un élément de cette liste
      - verrouiller le mutex de cet élément
      - SI l'élément n'est plus dans la liste ``free''
           ou que le dictionnaire contient déjà le bloc recherché
        - DEBUT ALORS
          - relâcher le mutex et recommencer la boucle
        - FIN ALORS
        - SINON on sort de la boucle pour utiliser cet élément
      - FIN SI
    - FIN ALORS
  - FIN SI

  # Sinon on espère qu'il y a un élément dans la liste ``sync''
  - même principe que pour la liste ``free''

  # Sinon il y a nécessairement un élément dans la liste ``dirty''
  - choisir un élément de cette liste
  - verrouiller le mutex de cet élément
    - SI l'élément n'est plus dans la liste ``dirty''
         ou que le dictionnaire contient déjà le bloc recherché
      - DEBUT ALORS
        - relâcher le mutex et recommencer la boucle
      - FIN ALORS
      - DEBUT SINON
        # Mise en oeuvre du ``write-back''
        - on sauvegarde le bloc associé à cet élément sur le disque
        - on sort de la boucle pour utiliser cet élément
      - FIN SINON
    - FIN SI

- FIN BOUCLE INFINIE

# On dispose maintenant d'un emplacement
# pour stocker notre bloc, reste à transférer
# son contenu depuis le disque

- transférer le contenu du bloc depuis le disque
- enregistrer le bloc dans le dictionnaire
- insérer l'élément sur la liste ``sync''
- sortir de la fonction !
```

1.4 Cache de pages

Contrairement au cache des blocs, ce "cache" n'a pas pour rôle d'accélérer quoi que ce soit. Sa présence répond à une exigence de cohérence entre les données projetées en mémoire (*mmap*) en mode partagé, et les données accédées avec `read()` et `write()`. On continue de parler de "cache" quand même puisqu'il s'agit d'utiliser la mémoire pour refléter une portion d'un autre périphérique. Le sous-système *pagecache* chargé de la gestion de ce cache est implémenté dans `sos/fs_pagecache.c`.

Les fonctions qu'il définit sont destinées à être appelées par *blockdev* pour le moment. Dans le prochain article, nous verrons qu'elles seront appelées aussi par les systèmes de fichiers qui désirent assurer la cohérence entre *read/write* et *mmap*. C'est pourquoi ces fonctions ne raisonnent pas en termes de blocs mais plutôt en termes de déplacement (*offset*).

1.4.1 Principe

Le principe de *pagecache* est le suivant (voir la figure 4). Imaginons qu'un processus utilisateur ait demandé à projeter un périphérique bloc, en mode "partage" (*shared*), dans son espace d'adressage. Dans ce cas, à chaque faute de page dans la région virtuelle associée, une nouvelle page physique sera mappée dans la région virtuelle. Elle sera remplie avec le contenu provenant du disque *via* un appel aux fonctions du cache de blocs décrites précédemment. Le *pagecache* enregistrera l'adresse de cette page dans son dictionnaire interne `offset du bloc` → `adresse de la page`.

Par la suite (voir la figure 5), quand *blockdev* devra faire une lecture sur le disque, avant tout appel à *blockcache* il demandera si le *pagecache* dispose du bloc demandé dans son dictionnaire. Si *pagecache* dispose déjà du bloc en question, alors on utilisera le contenu stocké dans la page associée et mappée en espace noyau. Sinon on fera appel au cache de blocs *blockdev*.

Ainsi, une fois qu'un processus utilisateur mappe une page reflétant le contenu de quelques blocs de disque, les ap-

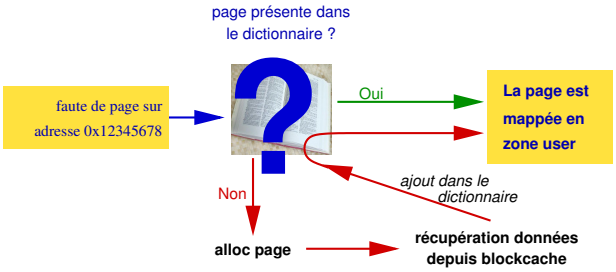

FIG. 4 – Scénario de prise en charge d'un défaut de page avec le *pagecache*.

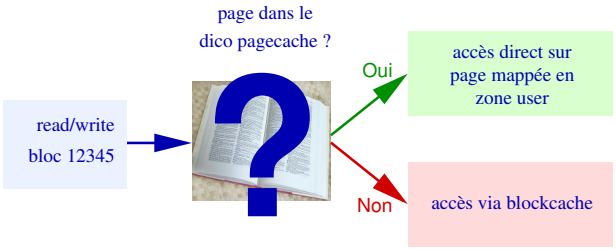

FIG. 5 – Scénario de prise en charge des lectures/écritures par *blockdev* en passant par *pagecache* et éventuellement *blockcache*.

pels à *read* et *write* accéderont directement à cette page sans passer par le cache de blocs. Il y aura d'emblée cohérence des données manipulées par *read*, *write* et *mmap*. Ce sera au *pagecache* d'assurer la synchronisation du contenu de ces pages avec le cache de blocs et, donc, avec le disque sous-jacent.

Contrairement au cache des blocs, les pages allouées dans ce cache ne sont soumises à aucune politique de remplacement de cache pour le moment. Une fois allouées suite à un défaut de page, elles resteront en mémoire jusqu'à ce que les processus qui les mappent les aient toutes démappées. Nous verrons certainement (à confirmer) dans un prochain article que c'est un thread spécial du système (*kswapd* sous Linux) qui s'occupe de mettre en œuvre la stratégie de remplacement de cache en écrivant sur disque en cas de besoin (i.e. si la mémoire commence à être remplie) les données modifiées, puis en démappant les pages associées.

Ce qui vient d'être présenté ne sera pas valable pour les projections de type "privé" (*private*). Pour ce type de projection, aucune cohérence ne doit être maintenue entre le contenu de la projection mémoire et le contenu du périphérique.

1.4.2 Implémentation

La fonction `sos_fs_pagecache_new_cache()` créé un nouveau "cache" de pages. Elle est appelée par *blockdev* lorsqu'un nouveau périphérique bloc est enregistré et s'occupe de créer une structure `struct sos_fs_pagecache` :

```
struct sos_fs_pagecache
{
  ...
  /* Dictionary page-aligned
     offset -> pagecache_entry */
  struct sos_hash_table  * lookup_table;

  /* Lists to look into in order to free a node */
  struct sos_fs_pagecache_entry * sync_list;
  struct sos_fs_pagecache_entry * dirty_list;
};
```

Cette structure renferme essentiellement le dictionnaire de correspondance `offset` → `description de la page` (voir plus bas), il s'agit d'une table de hachage. La structure contient aussi la liste des pages *sales* et *propres*. Elles permettent de savoir quelles pages sont à enregistrer sur disque lorsqu'on désire mettre à jour le disque avec ce qui a été modifié en mémoire. C'est la fonction `sos_fs_pagecache_sync()` qui s'occupera de cette mise à jour (voir plus bas).

Les éléments de description de chaque page renferment notamment les champs suivants :

```
struct pagecache_entry
{
  sos_vaddr_t    kernel_vaddr;
  sos_luoffset_t file_offset;
  ...
  struct sos_kmutex lock;
  ...
  /* Linkage in the
     dictionary & lists */
};
```

Autrement dit chaque page du cache de pages est associée à un déplacement dans le périphérique et est identifiée par une adresse virtuelle *en espace noyau*. En effet, le contenu d'une page enregistrée dans le cache de pages peut être utilisé par plusieurs processus différents. Il faut donc pouvoir accéder rapidement à son contenu depuis tous les espaces d'adressage. Une solution simple est de la mapper en zone noyau, c'est ce que nous faisons ici. Il eût aussi été possible de ne retenir que l'adresse physique de la page et de faire des mappings temporaires lorsque les différents processus utilisateur désirent accéder à son contenu par `read()/write()`. Mais cela s'avère plus lourd à l'exécution bien que moins consommateur en mémoire virtuelle dans la zone noyau. Et cela poserait d'autres problèmes plus subtils que nous ne détaillerons pas ici, comme par exemple la lecture de la table des partitions depuis le noyau.

Le reste des fonctions du sous-système *pagecache* se partage en 3 ensembles.

Accès par `read()`/`write()`. Les fonctions `sos_fs_pagecache_read()` et `sos_fs_pagecache_write()` seront appelées par les fonctions de lecture/écriture de *blockdev* pour accéder aux données depuis le cache de pages si elles y sont présentes. Si une partie seulement ou aucune des données recherchées n'est présente dans le cache, *blockdev* fera appel à *blockcache* pour récupérer le reste.

Accès par `mmap()`. Les fonctions `sos_fs_pagecache_ref_page()` et `sos_fs_pagecache_unref_page()` sont utilisées par *blockdev* pour la partie `mmap` du mécanisme. La première sera appelée par le gestionnaire de défaut de page associé à la région virtuelle mappant la ressource (méthode `page_in()` de `sos_umem_vmm_vr_ops`, voir l'article 7.5). Son rôle est de récupérer la page demandée dans le cache si elle y est déjà présente. Dans le cas contraire elle allouera une nouvelle page, transférera son contenu depuis le disque et l'ajoutera au cache. Dans ce cas le thread peut bloquer en attente du disque, donc les mêmes précautions qu'en section 1.3.2 seront prises pour éviter d'avoir 2 pages différentes qui *cachent* les mêmes blocs. La seconde fonction est appelée quand la page est démappée de la zone utilisateur.

Ces deux fonctions sont accompagnées de `sos_fs_pagecache_set_dirty()` servant à signaler que la page considérée est *sale*.

Resynchronisation forcée. Enfin, la fonction `sos_fs_pagecache_sync()` sert à synchroniser le disque avec le contenu des pages sales. Cette fonction ne fait qu'appeler une fonction de rappel (*callback*) `sync_fct()` sur toutes les pages de la liste des pages sales (`dirty_list`). Cette fonction de rappel est passée par *blockdev* en paramètre de `sos_fs_pagecache_new_cache()`. Dans le cas de *blockdev*, cette fonction de rappel est très simple et ne fait que des appels à `sos_blkcache_retrieve_block()` et `sos_blkcache_release_block()` du sous-système *blockcache*.

1.5 Intégration de *blockdev* dans le *VFS*

L'intégration de *blockdev* dans le *VFS* reprend les mêmes principes de base que *chardev* (voir l'article précédent). L'objectif est ***i)*** de gérer un dictionnaire `majeur/mineur → instance du périphérique bloc` (structure `sos_blockdev_instance` décrite plus bas) et ***ii)*** de dérouter les appels à *read/write* et *mmap* vers le code de *blockdev* et non vers le code du système de fichiers. Nous ne revenons pas sur la technique employée pour tout cela, c'est la même que celle de *chardev*, elle est résumée par la figure 6.

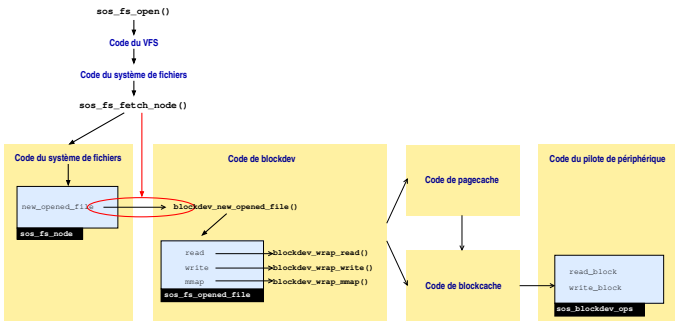

FIG. 6 – Principe de l'intégration de *blockdev* dans le *VFS*. En rouge : l'étape de déroutement mentionnée dans le texte.

La structure `sos_blockdev_instance` définie dans `sos/blkdev.c` et décrivant une instance de périphérique bloc, rassemble tout naturellement :

- **block_size**, la taille d'un bloc du périphérique, en octets ;
- **number_of_blocks**, le nombre de blocs du périphérique ;
- **operations**, le pointeur vers les opérations du pilote de périphérique, de type `sos_blockdev_operations` ;
- **blk_cache**, le cache de blocs du périphérique ;
- **map_cache**, le cache de pages du périphérique.

Ces champs sont initialisés par l'une des deux fonctions d'enregistrement du périphérique bloc `sos_blockdev_register_disk()/sos_blockdev_register_partition()` que nous décrivons plus bas.

Le premier rôle du code de *blockdev* est d'implémenter les fonctions *read/write* du VFS en utilisant de façon cohérente *blockcache* et *pagecache*. Son second rôle est de gérer les mécanismes liés à *mmap*, en particulier les défauts de pages, le tout en relation avec *pagecache* et *blockcache*.

Pour ce qui concerne *read/write* (revoir la figure 5), le principe est très simple. Il s'agit d'accéder à chaque bloc du périphérique en essayant de passer par *pagecache* d'abord (`sos_fs_pagecache_read/write`) et, si le bloc n'est pas présent, on fait appel à *blockcache* (`sos_blkcache_retrieve/release_block()`). Le sous-système *blockcache* se chargera des lectures/écritures nécessaires sur le disque.

Pour ce qui est de *mmap* (revoir la figure 4), tout ou presque se déroule dans la routine de traitement des défauts de page associée à la région virtuelle projetée (méthode *page_in* de la structure `sos_umem_vmm_vr_ops`, voir l'article 7.5). Son principe de fonctionnement est le suivant :

1. Lorsqu'aucune page n'était mappée à l'adresse fautive, de deux choses l'une :
   - Si la région virtuelle est de type "projection partagée" (*shared mapping*) : on récupère/alloue la page du *pagecache* correspondante par appel à `sos_fs_pagecache_ref_page()` ;
   - Sinon il s'agit d'une "projection privée" (*private mapping*) : on alloue une nouvelle page et on la remplit avec le contenu du disque (appels aux fonctions de *blockdev*).

   La page ainsi récupérée est mappée en zone utilisateur. On la projette en lecture seule afin de la marquer comme étant "sale" dès la première écriture (projection partagée) ou afin d'enclencher la copie sur écriture (projection privée).
2. Sinon, c'est que la page était déjà mappée en lecture seule et donc qu'une écriture est en train de se produire. Dans ce cas, de deux choses l'une :
   - Si la région virtuelle est de type "projection partagée" (*shared mapping*) : on indique à *pagecache* que la page est devenue "sale" (fonction `sos_fs_pagecache_set_dirty()`). Puis on la met en mode "lecture/écriture" pour que les écritures ultérieures ne déclenchent plus de défauts de page inutiles ;
   - Sinon il s'agit d'une "projection privée" (*private mapping*) et c'est donc le mécanisme de "copie sur écriture" (*COW*, *Copy-On-Write*) traditionnel qui est enclenché (voir l'article 7.5) afin de rendre les modifications invisibles depuis les autres processus et non sauvegardées sur disque.

La dernière fonction de *blockdev*, `sos_blockdev_sync()`, s'occupe de propager sur le disque physique toutes les modifications faites *via blockcache* et *pagecache* :

```
sos_ret_t
sos_blockdev_sync(struct sos_blockdev_instance * blockdev)
{
  sos_fs_pagecache_sync(blockdev->map_cache);
  sos_blkcache_flush(blockdev->blk_cache);
}
```

Elle est appelée toutes les 30 secondes par un thread noyau dédié, `bdflush`. On pourra cependant forcer une synchronisation manuellement en appelant `sync()` depuis une application utilisateur, ou en envoyant l'*ioctl* `SOS_IOCTL_BLOCKDEV_SYNC` à un périphérique bloc ouvert avec `open()`. On pourra aussi directement ouvrir le périphérique en mode `O_SYNC` si on veut que le disque reflète toujours l'état du cache suite aux lectures/écriture qu'on effectue.

1.6 Enregistrement des périphériques bloc et gestion des partitions

En réalité *blockdev* distingue deux types de périphériques bloc : les périphériques physiques en tant que tels (par

exemple : les disques durs) et les ressources logiques qu'ils contiennent, c'est-à-dire leurs *partitions*. Cela ne signifie pas que *blockdev* doit s'occuper d'identifier les partitions contenues dans un disque, cette tâche étant reléguée à un autre sous-système et détaillée en section 3. Cela signifie plutôt que *blockdev* définit une fonction pour enregistrer un périphérique physique de type disque (`sos_blockdev_register_disk()`) et une autre pour enregistrer ses partitions (`sos_blockdev_register_partition()`). En fait, `sos_blockdev_register_partition()` pourra être aussi appelée pour enregistrer les éventuelles sous-partitions d'une partition, les sous-sous-partitions d'une sous-partition, etc. La seule contrainte émise est que les partitions commencent toutes sur une frontière de bloc.

Les deux fonctions précédentes s'occupent d'enregistrer le périphérique dans le dictionnaire `majeur/mineur` → `instance du périphérique bloc` de *blockdev*. Lorsqu'un nouveau disque est enregistré, des caches de pages et de blocs tout neufs sont alloués. Par conséquent, dans SOS, chaque périphérique disque disposera de *son* cache de blocs et de *son* cache de pages.

Quand on enregistre une nouvelle partition, on doit préciser dans quel disque elle se situe. Le *cache de pages* et *le cache de blocs* seront alors ceux du disque (voir la figure 7). Ceci a pour intérêt principal de garantir *de facto* la synchronisation des accès au disque lorsque plusieurs opérations concurrentes ont lieu sur les différentes partitions, puisqu'on passe par les mêmes caches de pages et de blocs.

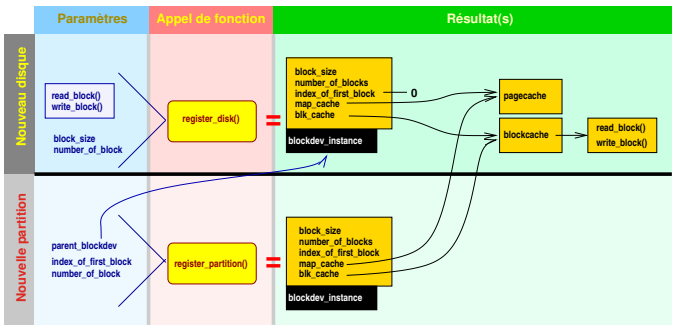

FIG. 7 – Relation entre disques et partition(s).

Comme nous l'avons dit, ce n'est pas *blockdev* qui devra identifier les partitions du disque, c'est une fonction tierce du noyau. Il en découle que le noyau doit être capable de lire le contenu du disque. Or le noyau n'a pas accès aux fonctions *read*/write classiques du *VFS*. En effet, celles-ci s'appliquent à des "fichiers ouverts", sachant qu'un fichier ouvert est toujours une ressource liée à un processus dans SOS. Le noyau ne peut donc pas créer pour lui-seul une ressource "fichier ouvert" puisqu'il n'est associé à aucun processus particulier. C'est pourquoi *blockdev* définit une petite interface de programmation simple : `sos_blockdev_kernel_read()` pour lire des portions du disque/d'une partition depuis le noyau, et `sos_blockdev_kernel_write()` pour écrire. Le code de ces fonctions est partagé avec les fonctions utilisées par le *VFS* et possède donc exactement les mêmes caractéristiques.

À cette fin, nous avons étendu l'interface de programmation `sos/uaccess.h` permettant d'accéder à la zone utilisateur depuis le noyau. Nous l'avons enrichie de deux fonctions de copie : `sos_memcpy_generic_from/to()`. Ces fonctions prennent en paramètre un nouveau type d'adresse, `sos_genaddr_t`, et effectuent les recopies indifféremment depuis/vers l'espace noyau ou l'espace utilisateur : elles appellent au choix les autres fonctions de `uaccess.h` (pour les transferts depuis/vers la zone utilisateur) ou le traditionnel `memcpy()` (pour les transferts depuis/vers le noyau). Toutes les fonctions susceptibles d'appeler les deux fonctions précédentes travaillent donc avec ce type d'adresse particulier qu'il convient d'initialiser rigoureusement grâce à la macro `SOS_GENADDR_DECL()`. Ceci concerne seulement 4 fonctions internes de *blockdev* et de *pagecache*.

1.7 Et dans Linux ?

1.7.1 Notion de *requête*

L'interface d'un pilote de périphérique dans SOS (`sos_blockdev_operations`) est naïve et différente de celle de Linux. Dans Linux et dans les systèmes d'exploitation performants, l'élément fondamental n'est pas la définition des fonctions de lecture/écriture, mais la notion de *requête* de lecture/écriture (ou de contrôle).

Un pilote de périphérique bloc joue alors le rôle d'un interpréteur de séquences de requêtes. Les requêtes de lecture/écriture au périphérique ne conduisent pas immédiatement à une lecture/écriture sur le périphérique. Elles sont "ordonnancées" afin de limiter le nombre de déplacements de la tête de lecture d'un disque dur par exemple. Cet ordonnancement est effectué par un *ordonnanceur d'entrée-sortie* (les fameux *Anticipatory Scheduler* et *CFQ Scheduler* du noyau Linux).

Pour illustrer, plutôt que d'effectuer immédiatement la séquence `<lecture bloc 9><lecture bloc 1><lecture bloc 12><lecture bloc 3><écriture bloc 2><lecture bloc 4>`, les requêtes peuvent être ré-ordonnancées dans l'ordre suivant : `<lecture bloc 1><lecture bloc 3><lecture bloc 9><lecture bloc 12><écriture bloc 2><lecture bloc 4>`. Le ré-ordonnancement doit obéir à certaines règles. En effet, avec ou sans ordonnancement, on doit finir par retrouver les mêmes données sur le disque. Avec de plus en plus de disques durs récents, c'est le disque dur lui-même qui s'occupe de réordonnancer les requêtes.

1.7.2 Unification progressive des caches

Dans les anciennes versions de Linux (2.2 et avant), une approche similaire à celle de SOS était adoptée. La gestion des périphériques bloc (et, par extension, des fichiers) reposait sur deux caches : le *buffer cache* et le *page cache*. Comme pour le *block cache* de SOS, le *buffer cache* de Linux avait pour rôle d'accélérer les accès au disque en conservant les blocs fréquemment accédés en mémoire. Le *page cache* de Linux avait un rôle similaire à celui de SOS tout en étant plus incontournable encore. En effet, dans Linux 2.2, *toutes* les opérations de lecture/écritures devaient passer par ce cache, ce qui assurait *de facto* la cohérence avec les données projetées en mémoire. Dans SOS, les lectures/écritures ne passent par ce cache *que* si les blocs accédés ont déjà été projetés en mémoire. Le premier reproche fait à Linux était que les données des disques pouvaient, dans les cas extrêmes, être doublement présentes en mémoire : une fois dans le

buffer cache, une autre dans le *page cache*. Dans SOS, ce "gaspillage" de la mémoire existe aussi mais devrait être atténué puisque les *block caches* ont une taille bornée dès la création du cache. En contrepartie, l'efficacité de SOS sera forcément moindre que celle de Linux. Le *block cache* de Linux au contraire peut grossir librement (mais en concertation avec le *page cache* tout de même) jusqu'à occuper le maximum de mémoire : il est normal que les caches occupent couramment plus de 80% de la mémoire. Linux privilégie donc l'efficacité car en général un cache est d'autant plus efficace qu'il dispose de davantage de mémoire.

L'autre problème de l'architecture de Linux 2.2 était lié à l'éviction des blocs ou des pages depuis ces deux caches en cas de besoin de place pour d'autres blocs/pages : doit-on évincer un élément du *buffer cache* ou une page du *page cache* ? Les critères de choix ne sont pas évidents. Dans SOS, nous avons choisi une solution simple dans laquelle le *block cache* est alloué dès le début et a une taille fixe. Il forme donc un système indépendant du *page cache*. Il n'y aura ainsi jamais de conflit d'intérêt entre les deux.

Linux 2.4 remédie aux deux limitations précédentes en *unifiant* progressivement le *buffer cache* et le *page cache* en un seul : le *page cache*. Linux 2.6 repose sur une version aboutie de cette unification. Il amène également une amélioration de la gestion des échanges entre périphériques et mémoire pour, entre autres, s'affranchir de la limite des 1 Go de mémoire qui existait auparavant. Dans SOS, nous avons choisi de ne pas faire cette unification pour des raisons de simplicité (très discutables) et surtout pour distinguer clairement le rôle des deux caches : l'un pour accélérer (*blockcache*), l'autre pour assurer une propriété de cohérence (*pagecache*). L'unification de ces deux caches dans SOS est donc laissée à titre d'exercice.

2 Pilote IDE

Dans cette section, nous présentons l'implémentation d'un pilote de périphérique simple permettant d'exploiter les disques durs IDE dans SOS.

2.1 Matériel

2.1.1 Norme *ATA*

ATA, pour *Advanced Technology Attachment*, est une interface de connexion pour les périphériques de stockage tels que les disques durs ou les lecteurs de CD-ROM. Bien que *ATA* soit toujours le nom officiel de ce standard, les termes *IDE* (*Integrated Drive Electronics*) ou *EIDE* (*Enhanced IDE*) sont souvent utilisés comme des synonymes et la différence entre ces acronymes demeure donc assez floue. Depuis l'introduction de la norme *Serial ATA*, le standard *ATA* est désormais souvent nommé *Parallel ATA*. D'autre part, la norme *ATA* a été étendue pour les périphériques comme les lecteurs et graveurs de CD-ROM ou DVD par la norme *ATAPI*, *Advanced Technology Attachment Packet Interface*. Les spécifications de la norme *ATA/ATAPI-4* sont disponibles en [1].

Dans SOS, le pilote de périphérique se limitera à la détection des périphériques *ATA* et *ATAPI* et à l'utilisation des disques durs *ATA*. Dans le reste de l'article, on utilisera le terme *IDE*, plus courant.

2.1.2 Contrôleur et adressage

Un ordinateur de type PC comporte classiquement deux contrôleurs IDE intégrés à la carte mère. Il est bien évidemment possible de disposer de contrôleurs supplémentaires par l'intermédiaire d'une carte PCI par exemple, mais le pilote IDE de SOS se limitera à la gestion des 2 premiers contrôleurs standards intégrés à la carte mère. Sur chacun de ces contrôleurs peuvent se connecter deux périphériques : un maître et un esclave.

Chaque disque dur comporte un certain nombre de secteurs qui ont toujours une taille de 512 octets dans le cas des disques durs IDE. Ces secteurs sont organisés sur le disque selon une certaine *géométrie* : un certain nombre de plateaux accessibles par une tête de lecture, un certain nombre de cylindres, comportant chacun un certain nombre de secteurs (voir par exemple [2] ou le chapitre 31 de [3] pour les détails sur le fonctionnement d'un disque dur). À l'origine, les secteurs étaient donc adressés en mode *CHS*, *Cylinder/Head/Sector*, c'est-à-dire en donnant le numéro du cylindre, puis le numéro de la tête de lecture, puis le numéro de secteur à l'intérieur de ce cylindre. Cette méthode d'adressage était limitée à 1024 cylindres, 255 têtes et 63 secteurs par tête, ce qui donnait une taille de disque dur maximale de 8 Go. Depuis, de nouvelles versions de la norme *ATA* ont introduit la méthode d'adressage *LBA* pour *Logical Block Addressing*, qui consiste à adresser les secteurs de 0 à n sans se soucier de la géométrie sous-jacente. Au départ, l'adressage *LBA* fonctionnait sur 28 bits, ce qui donnait une taille maximale de disque de 128 Go ($2^{28} = 268435456\ secteurs = 128\ Go$). Aujourd'hui, un mode *LBA* sur 48 bits est disponible, permettant d'utiliser des disques durs de 134217728 Go, soit 131072 Tera-octets, ou encore 128 Peta-octets.

2.1.3 Registres d'entrées-sorties

D'un point de vue logiciel, chaque contrôleur est programmable par l'intermédiaire d'un certain nombre de ports d'entrées-sorties. Sur PC, les registres du premier contrôleur sont accessibles à partir de l'adresse `0x1F0` (`IDE_CONTROLLER_0_BASE`), tandis que les registres du second contrôleur sont accessibles à partir de l'adresse `0x170` (`IDE_CONTROLLER_1_BASE`). À chaque contrôleur est également associé une *IRQ*, c'est-à-dire une interruption matérielle : `IDE_CONTROLLER_0_IRQ` pour le premier contrôleur et `IDE_CONTROLLER_1_IRQ` pour le second contrôleur.

À partir des adresses de base des ports d'entrées-sorties, des registres sont disponibles, dont une description complète est disponible à la section 7.2 de [1]. En particulier :

- le registre `ATA_DATA` permet de lire ou d'écrire les données d'un secteur ;
- le registre `ATA_ERROR` permet de disposer d'informations sur l'erreur survenue si il y a lieu ;
- le registre `ATA_SECTOR_COUNT` permet d'indiquer le nombre de secteurs concernés par la requête de lecture/écriture ;
- le registre `ATA_SECTOR_NUMBER` permet d'indiquer le numéro du premier secteur de la requête de lecture ou d'écriture ;
- les registres `ATA_CYL_LSB` et `ATA_CYL_MSB` permettent de fixer le numéro du cylindre du disque dur concerné par la requête ;

– le registre ATA_DRIVE permet de contrôler le disque à utiliser ;
– le registre ATA_STATUS permet de connaître l'état d'un contrôleur ;
– le registre ATA_CMD permet d'envoyer des commandes au contrôleur (identification, lecture, écriture, etc.). La liste des commandes disponibles et leur utilisation est décrite au chapitre 8 de [1].

Un pilote de périphérique doit donc utiliser ces registres pour commander les contrôleurs IDE, et implémenter à partir de ceci les méthodes read_block() et write_block() telles que définies dans la structure struct sos_blockdev_operations du fichier sos/blkdev.h.

2.1.4 Modes de fonctionnement

Il existe deux manières d'utiliser les contrôleurs IDE :
– La première méthode, la plus simple, est le *polling*. Elle consiste à envoyer les données au contrôleur et à attendre de manière active que celui-ci ait terminé l'opération. Pendant cette attente active, le processeur est monopolisé et ne peut pas être utilisé pour une autre tâche ;
– La seconde méthode consiste à utiliser une interruption signalant la fin d'une opération réalisée par le contrôleur. Après avoir envoyé la commande et les données au contrôleur par les ports d'entrée/sortie, il est donc possible d'utiliser le processeur pour une autre tâche : l'interruption levée par le contrôleur lorsqu'il aura terminé son opération permettra de réveiller l'application ayant effectué la requête.

Dans SOS, nous utilisons ces deux techniques : la première pour l'identification des périphériques, la seconde pour les requêtes de lecture et d'écriture.

De manière indépendante de ces deux techniques, un mécanisme supplémentaire peut ou non être utilisé : le *DMA* (pour *Direct Memory Access*). Sans *DMA*, le processeur est monopolisé pendant le transfert des données à lire ou à écrire depuis ou vers le contrôleur. En utilisant le *DMA*, le processeur programme une puce (par exemple le 8237A pour les vieux périphériques ISA PC) qui effectuera directement le transfert de la mémoire vers le contrôleur ou du contrôleur vers la mémoire. Pendant ces transferts réalisés par la puce spécialisée, le processeur pourra être utilisé pour exécuter d'autres tâches du système. Le *DMA* permet donc d'économiser de manière importante le processeur lors des transferts entre la mémoire et les périphériques. Ce mécanisme n'a toutefois pas été utilisé dans SOS afin de simplifier l'implémentation du pilote de périphérique, mais le lecteur intéressé pourra se reporter au chapitre 28 de [3] pour plus d'informations.

Il est important de noter que le pilote de périphériques IDE de SOS n'implémente qu'une fraction minime de la norme *ATA*. Il dispose donc de fonctionnalités limitées, n'a été testé que sur un émulateur de type *Qemu* ou *Bochs*, et n'a pas prétention à fonctionner sur toutes les plate-formes.

2.2 Structures de données

Dans le pilote IDE de SOS, chaque contrôleur est représenté par une structure ide_controller qui contient l'adresse de base du contrôleur, le numéro d'*IRQ*, l'état, un mutex permettant de contrôler les accès concurrents, un sémaphore pour réveiller les processus en attente de la terminaison d'une opération, et un tableau des périphériques attachés à ce contrôleur. Le tableau statique ide_controllers[] associe la structure précédemment décrite à chaque contrôleur du PC, avec leurs adresses de base, leurs *IRQ*s, etc.

D'autre part, chaque périphérique est représenté par une structure de type struct ide_device qui donne le type du périphérique, sa taille, sa géométrie (en terme de cylindres, têtes et secteurs par tête), et la méthode d'adressage à utiliser (*CHS* si support_lba est faux, *LBA* dans le cas contraire).

2.3 Initialisation

À l'initialisation du pilote de périphérique, implémentée dans ide_driver_init(), il s'agit tout d'abord de détecter les périphériques présents puis de les enregistrer dans le système.

La détection des périphériques est effectuée par la fonction ide_probe_controller(). Celle-ci appelle la fonction ide_probe_device() pour les deux périphériques qui peuvent potentiellement être attachés à ce contrôleur. Cette fonction essaie de déterminer le type du périphérique disponible et renvoie IDE_DEVICE_NONE si aucun périphérique n'est disponible, IDE_DEVICE_HARDDISK si il s'agit d'un disque dur, ou IDE_DEVICE_CDROM si il s'agit d'un lecteur de CD-ROM.

Pour cela, la fonction ide_probe_device() dialogue avec le contrôleur par l'intermédiaire des ports d'entrée-sortie :

1. Il commence par sélectionner le périphérique à détecter : maître ou esclave, en envoyant une commande au registre ATA_DRIVE ;
2. Il lit ensuite l'état du contrôleur via le registre ATA_STATUS ;
3. Si le contrôleur indique qu'il est occupé, alors il n'y a pas de périphérique disponible, et on retourne IDE_DEVICE_NONE ;
4. Si le contrôleur indique être prêt, c'est qu'un disque dur est attaché à ce contrôleur. On retourne alors IDE_DEVICE_HARDDISK ;
5. Sinon, on essaie de déterminer si il s'agit d'un lecteur de CD-ROM. Pour cela, il faut lire les registres de configuration du numéro de cylindre et voir si les valeurs lues correspondent à des *magic* connus : ATAPI_MAGIC_LSB et ATAPI_MAGIC_MSB. Si c'est le cas, alors il s'agit d'un lecteur de CD-ROM et on retourne IDE_DEVICE_CDROM.

Lorsqu'un disque dur a été détecté, la fonction ide_probe_controller() appelle ide_get_device_info() pour disposer d'informations complémentaires sur le disque dur : taille, géométrie, etc.

Cette fonction, qui envoie une commande au contrôleur IDE, fonctionne de la manière suivante :

1. Désactive les interruptions d'acquittement d'opération sur le contrôleur, en envoyant ATA_A_nIEN dans le port d'entrée-sortie ATA_DEVICE_CONTROL ;

2. Sélectionne le périphérique, maître ou esclave, par l'intermédiaire du port `ATA_DRIVE` ;
3. Envoie la commande `ATA_C_ATA_IDENTIFY` au contrôleur en utilisant le port `ATA_CMD` (voir section 8.12 de [1] pour une description complète de la commande) ;
4. Attend de manière active la terminaison de l'opération en lisant régulièrement le registre `ATA_STATUS` et en regardant si le bit `ATA_S_BSY` passe à 0 ;
5. Une fois l'attente terminée, on vérifie si le bit `ATA_S_DRQ`, indiquant si des données sont disponibles dans le contrôleur, est positionné ;
6. Lit les données disponibles dans le contrôleur par l'intermédiaire du registre `ATA_DATA`. Puisque 512 octets sont disponibles, une boucle réalisant 256 lectures de mots de 2 octets (en utilisant `inw()`) est utilisée ;
7. Extrait les informations utiles de ces données. Les données lues correspondent à une structure définie dans la norme *ATA* (voir section 8.12.8 de [1]). Cette structure est copiée dans la structure `struct ide_device_info` de SOS. Elle comprend de nombreuses informations sur le périphérique, avec notamment la géométrie du disque et des informations sur la méthode d'adressage utilisable (*LBA* ou *CHS*). La structure `struct ide_device` du périphérique est complétée grâce à ces informations.

Pour finir, la fonction `ide_probe_controller()` enregistre la fonction `ide_irq_handler()` comme gestionnaire de l'*IRQ* associée au contrôleur si des périphériques ont été détectés.

Une fois la détection des périphériques terminée sur les deux contrôleurs, la fonction `ide_driver_init()` enregistre les périphériques dans le système par l'intermédiaire de la fonction `ide_register_devices()`.

Celle-ci parcourt les contrôleurs et les périphériques de chaque contrôleur. Un message est affiché pour chaque périphérique présent. Dans le cas où le périphérique est un disque dur, des opérations supplémentaires sont effectuées :

– Enregistrement du disque dur en tant que périphérique bloc dans le sous-système *blockdev* grâce à la fonction `sos_blockdev_register_disk()`. Celle-ci permet d'associer le *majeur/mineur* du périphérique bloc aux opérations de la structure `ide_ops` et à un pointeur opaque pour le sous-système *blockdev* mais qui permettra aux opérations du pilote *IDE* de savoir à quel périphérique il faut s'adresser. Le majeur des disques durs est `SOS_BLOCKDEV_IDE_MAJOR`, défini dans `drivers/devices.h`. Quant au mineur, il est calculé par la macro `IDE_MINOR()`. Celle-ci calcule le numéro du mineur grâce au numéro du contrôleur et du périphérique, en laissant un espace de 15 mineurs entre deux périphériques pour enregistrer les partitions de ce périphérique. Ainsi, le deuxième périphérique du premier contrôleur aura pour mineur 16, et ses partitions pourront s'étendre du mineur 17 au mineur 31 (voir figure 8). La même technique est à l'œuvre dans Linux. Par défaut, on demande à `sos_blockdev_register_disk()` d'allouer un *block-cache* de 128 blocs (64 ko). On pourra s'amuser à comparer les performances du système en faisant varier cette taille de cache.

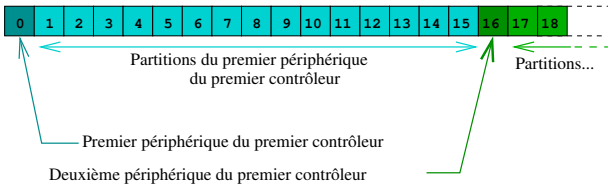

FIG. 8 – Affectation des mineurs pour les périphériques *IDE* et leurs partitions.

– Détection des partitions sur le disque dur en utilisant la fonction `sos_part_detect()` du pilote de partition, détaillée en section 3.

2.4 Fonctionnement

Le pilote *IDE* implémente les opérations `read_block()` et `write_block()` requises par le sous-système *blockdev* dans les fonctions `ide_read_device()` et `ide_write_device()`. Toutes deux sont de simples *wrappers* autour de la fonction `ide_io_operation()`. Elles ne font que convertir le pointeur opaque `void *blkdev_instance` en un pointeur sur le périphérique concerné par la lecture ou l'écriture, c'est-à-dire un pointeur sur `struct ide_device`. Ce pointeur opaque correspond à celui qui a été passé lors de l'enregistrement du périphérique à la fonction `sos_blockdev_register_disk()`.

2.4.1 Adressage du bloc

Pour commencer, la fonction `ide_io_operation()` calcule les paramètres à passer dans les registres `ATA_SECTOR_NUMBER`, `ATA_CYL_LSB`, `ATA_CYL_MSB` et `ATA_DRIVE` dans les variables `sect`, `cyl_lo`, `cyl_hi` et `head`. Ces paramètres permettront d'adresser le bloc qui doit être lu ou écrit :

```
if (dev->support_lba)
  {
    sect   = (block & 0xff);
    cyl_lo = (block >> 8) & 0xff;
    cyl_hi = (block >> 16) & 0xff;
    head   = ((block >> 24) & 0x7) | 0x40;
  }
else
  {
    int cylinder = block /
      (dev->heads * dev->sectors);
    int temp = block %
      (dev->heads * dev->sectors);
    cyl_lo = cylinder & 0xff;
    cyl_hi = (cylinder >> 8) & 0xff;
    head   = temp / dev->sectors;
    sect   = (temp % dev->sectors) + 1;
  }
```

La méthode de calcul diffère selon le mode d'adressage : *LBA* (première partie du code) ou *CHS* (seconde partie). En mode *LBA*, le calcul est très simple : il suffit de décomposer le numéro absolu du bloc en plusieurs parties : 8 bits de poids faible dans `sect`, les prochains 8 bits dans `cyl_lo`, les suivants dans `cyl_hi` et le reste dans `head`. Le `| 0x40` permet d'indiquer que l'accès se fait par le mode d'adressage *LBA*. Voir la figure 9 pour un exemple.

En mode *CHS*, le calcul est un peu plus complexe. On calcule tout d'abord le numéro du cylindre contenant le bloc désiré, ce qui permet de calculer les valeurs `cyl_lo` et `cyl_hi`. `temp` contient alors l'offset du bloc demandé dans le cylindre. Pour calculer le numéro de la tête `head`, on divise simplement cet offset du bloc dans le cylindre par le

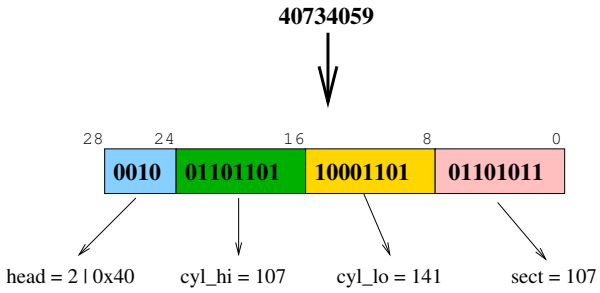

FIG. 9 – Calcul des paramètres pour adresser le bloc 40734059 en mode *LBA*.

nombre de secteurs par têtes. Et pour calculer le numéro du secteur, on prend le reste de cette division.

2.4.2 Lecture ou écriture

Une fois les paramètres calculés, le *mutex* du contrôleur est pris et sera conservé pendant toute la durée de l'opération afin de réguler les accès concurrents au contrôleur. Dès que le *mutex* est pris, l'opération proprement dite peut commencer. Elle se déroulera en plusieurs étapes, comme l'illustre la figure 10.

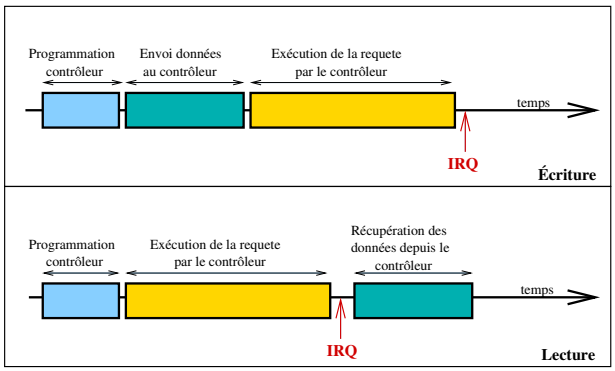

FIG. 10 – Déroulement des opérations lors de la lecture et de l'écriture sur un disque dur IDE.

```
1.   sos_kmutex_lock (& dev->ctrl->mutex, NULL);
2.
3.   outb(ATA_D_IBM | devselect, dev->ctrl->ioaddr + ATA_DRIVE);
4.   udelay(100);
5.
6.   outb(ATA_A_4BIT, dev->ctrl->ioaddr + ATA_DEVICE_CONTROL);
7.   outb(1,          dev->ctrl->ioaddr + ATA_ERROR);
8.   outb(0,          dev->ctrl->ioaddr + ATA_PRECOMP);
9.   outb(1,          dev->ctrl->ioaddr + ATA_SECTOR_COUNT);
10   outb(sect,       dev->ctrl->ioaddr + ATA_SECTOR_NUMBER);
11.  outb(cyl_lo,     dev->ctrl->ioaddr + ATA_CYL_LSB);
12.  outb(cyl_hi,     dev->ctrl->ioaddr + ATA_CYL_MSB);
13.  outb((ATA_D_IBM | devselect | head),
14.       dev->ctrl->ioaddr + ATA_DRIVE);
15.
16.  if (iswrite)
17.    outb(ATA_C_WRITE, dev->ctrl->ioaddr + ATA_CMD);
18.  else
19.    outb(ATA_C_READ, dev->ctrl->ioaddr + ATA_CMD);
20.
21.  if (iswrite)
22.    {
23.      sos_ui16_t *buffer = (sos_ui16_t *) buf;
24.      for (i = 0 ; i < 256 ; i++)
25.        outw (buffer[i], dev->ctrl->ioaddr + ATA_DATA);
26.
27.      if((inb(dev->ctrl->ioaddr + ATA_STATUS) & ATA_S_DRQ))
28.        {
29.          sos_kmutex_unlock (& dev->ctrl->mutex);
30.          return -SOS_EFATAL;
31.        }
32.    }
33.
34.  sos_ksema_down (& dev->ctrl->ack_io_sem, NULL);
35.
36.  if(inb(dev->ctrl->ioaddr + ATA_STATUS) & ATA_S_ERROR)
37.    {
38.      sos_kmutex_unlock (& dev->ctrl->mutex);
39.      return -SOS_EFATAL;
40.    }
41.
42.  if(!iswrite && !(inb(dev->ctrl->ioaddr + ATA_STATUS) & ATA_S_DRQ))
43.    {
44.      sos_kmutex_unlock (& dev->ctrl->mutex);
45.      return -SOS_EFATAL;
46.    }
47.
48.  if (! iswrite)
49.    {
50.      sos_ui16_t *buffer = (sos_ui16_t *) buf;
51.      for (i = 0 ; i < 256 ; i++)
52.        buffer [i] = inw (dev->ctrl->ioaddr + ATA_DATA);
53.    }
54.
55.  sos_kmutex_unlock (& dev->ctrl->mutex);
```

On commence par sélectionner le périphérique ligne 3, puis par envoyer les paramètres de la commande de lecture ou d'écriture lignes 6 à 13, et notamment les paramètres d'adressage du secteur demandé. La requête s'effectuant sur un secteur, on envoie 1 dans le registre `ATA_SECTOR_COUNT`. Une fois les paramètres envoyés, c'est au tour de la commande elle-même : `ATA_C_WRITE` dans le cas d'une écriture, `ATA_C_READ` dans le cas d'une lecture (lignes 16 à 19, voir sections 8.27 et 8.48 de [1] pour la description de ces commandes, et sections 9.7 et 9.8 pour le protocole d'utilisation).

Ensuite, s'il s'agit d'une écriture, il faut transmettre les données à écrire. Celles-ci sont transmises mot par mot à l'aide de l'instruction `outw()` sur le registre `ATA_DATA`. Dès la réception des données, acquittée par le contrôleur en positionnant le bit `ATA_S_DRQ` à 0 (ligne 27), le contrôleur effectuera l'opération d'écriture et lèvera une interruption à la fin.

Ensuite, ligne 34, nous bloquons sur le sémaphore `ack_io_sem` qui signale la terminaison d'une opération. En effet, dans le cas d'une lecture, nous attendons que les données deviennent disponibles, et dans le cas d'une écriture, nous attendons l'acquittement du contrôleur. Ce sémaphore sera débloqué par la fonction `ide_irq_handler()` qui est le gestionnaire d'interruption des contrôleurs *IDE*. Puisque nous bloquons sur un sémaphore et non de manière active, le processeur peut, pendant cette attente, être utilisé pour exécuter d'autres processus.

Une fois l'interruption survenue, la fonction `sos_ksema_down()` retourne. On regarde alors si l'opération s'est bien déroulée en vérifiant que le registre `ATA_STATUS` n'a pas le bit `ATA_S_ERROR` positionné et que le bit `ATA_S_DRQ` est positionné dans le cas d'une lecture (lignes 36 à 46). Pour une opération d'écriture, il ne reste rien à faire, mais dans le cas d'une opération de lecture, il faut maintenant récupérer les données depuis le contrôleur. Comme pour la fonction `ide_get_device_info()`, cela se fait en lisant 256 mots de 2 octets depuis le registre `ATA_DATA` (lignes 48 à 53).

L'opération de lecture/d'écriture étant maintenant terminée, nous pouvons relâcher le *mutex* sur le contrôleur, ligne 55, afin d'autoriser d'autres processus à utiliser ce contrôleur.

2.5 Tester dans *Qemu* et *Bochs*

Pour tester ce pilote *IDE* dans les émulateurs *Qemu* ou *Bochs*, il faut tout d'abord créer une image virtuelle de disque dur. En effet, le contenu de notre disque dur sera simplement stocké dans un fichier de la machine hôte, l'émulateur se chargeant de le faire passer pour un vrai

disque dur *IDE* auprès de l'OS s'exécutant dans la machine émulée.

Pour créer un disque dur virtuel de 10 Mo, on utilisera simplement la commande `dd` de la façon suivante :

```
$ dd if=/dev/zero of=disk10M.img bs=1M count=10
```

Pour l'utiliser dans *Qemu*, il suffira de passer les options "`-hda disk10M.img -boot a`" à ce dernier.

Pour l'utiliser dans *Bochs*, il faut ajouter une ligne dans le fichier de configuration `.bochsrc`. Cette ligne doit notamment préciser la géométrie du disque dur. Ici, la géométrie n'a pas de rapport avec une quelconque configuration physique du disque dur, mais *Bochs* nécessite néanmoins une telle information afin de pouvoir la fournir à l'OS s'exécutant dans la machine émulée. Pour un disque dur de 10 Mo, on pourra par exemple choisir une géométrie de 320 cylindres, 4 têtes et 16 secteurs par piste ($320 * 4 * 16 * 512 = 10\ Mo$). On ajoutera donc les lignes suivantes dans le fichier de configuration de *Bochs* :

```
ata0: enabled=1, ioaddr1=0x1f0, ioaddr2=0x3f0, irq=14
ata0-master: type=disk, path=disk10M.img, mode=flat, cylinders=320, heads=4, spt=16, translation=none
```

3 Détection des partitions

3.1 Le partitionnement sur PC

La plupart du temps, les disques durs sont découpés en plusieurs parties, les *partitions*. Ce *partitionnement* nécessite un support du système d'exploitation afin de reconnaître les différentes partitions disponibles et de les rendre utilisables au travers de périphériques bloc.

Sur PC, le premier secteur d'un disque dur partitionné est composé de trois parties : 446 octets pouvant servir à stocker du code de démarrage, une table des partitions comportant 4 entrées de 16 octets, et un marqueur de deux octets indiquant que ce secteur contient du code de démarrage à exécuter. Ici, c'est évidemment la table des partitions qui nous intéresse.

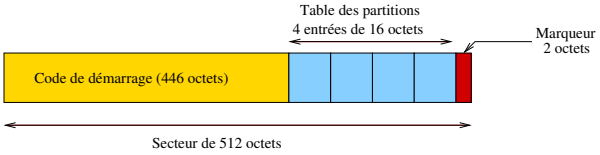

FIG. 11 – Composition du premier secteur d'un disque partitionné.

Ce premier secteur contient donc la description de quatre partitions, qui peuvent être de deux types : *primaires* ou *étendues*. En effet, quatre partitions n'étant parfois pas suffisantes, on a recours à une *partition étendue* qui peut contenir un nombre a priori illimité de *lecteurs logiques*. Pour cela, le début de la *partition étendue* comporte une mini-table des partitions de deux entrées. La première décrit le *lecteur logique* courant (comme si c'était une partition primaire) et la deuxième entrée pointe sur la prochaine mini-table des partitions, qui décrira à nouveau un *lecteur logique* et pointera sur la prochaine mini-table. La *partition étendue* est donc composée d'une liste chaînée de *lecteurs logiques* (voir figure 12). On pourra se faire une idée de cette structure en appelant la commande `fdisk -l` sous Linux.

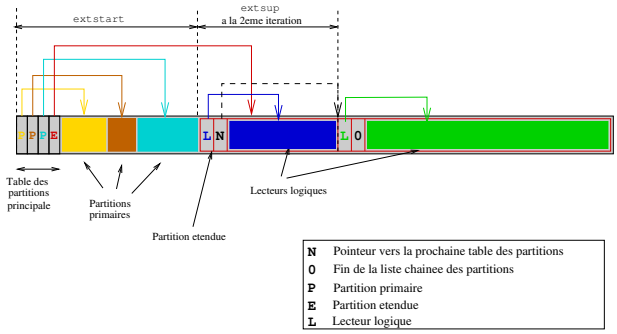

FIG. 12 – **Partitions primaires et étendues**. La table de partitions principale, à gauche, définit 4 partitions. Trois primaires, en jaune, marron, et bleu turquoise, et une étendue, en rouge. Au début de cette partition étendue rouge se trouve une nouvelle table des partitions décrivant un *lecteur logique*, en bleu, et pointant sur la prochaine table des partitions. Cette nouvelle table des partitions décrit un *lecteur logique*, en vert, mais ne pointe pas vers une nouvelle table des partitions. La liste chaînée des *lecteurs logiques* est terminée.

3.2 Implémentation du pilote

Dans SOS, le pilote chargé de la détection des partitions est implémenté dans le fichier `drivers/part.c`. Du point de vue de l'extérieur, ce pilote met à disposition deux fonctions. `sos_part_detect()` permet de détecter et d'enregistrer dans le sous-système *blockdev* les partitions pour un périphérique bloc dont le *majeur/mineur* est passé en paramètre, tandis que `sos_part_undetect()` permet de retirer les partitions enregistrées du sous-système *blockdev*. La fonction `sos_part_detect()` est la plus importante et c'est donc celle que nous décrirons.

Le pilote commence par récupérer l'instance du disque dur à lire par un appel à `sos_blockdev_lookup_instance()` puis alloue un buffer de la taille d'un bloc du disque dur (512 octets dans notre cas). Il lit ensuite le premier secteur du disque dur où se trouve la table des partitions principales. Pour simplifier le travail, la structure `struct partition_entry` reprend exactement la structure d'un descripteur de partition tel que stocké sur le disque dur :

```
typedef struct partition_entry
{
  sos_ui8_t  active;
  sos_ui8_t  start_dl;
  sos_ui16_t start_cylinder;
  sos_ui8_t  type;
  sos_ui8_t  end_dl;
  sos_ui16_t end_cylinder;
  sos_ui32_t lba;
  sos_ui32_t size;
} partition_entry_t;
```

On fait alors pointer `part_entry`, un pointeur vers un `struct partition_entry`, sur le 446ème octet de ce premier secteur (`PART_TABLE_OFFSET` vaut 446) :

```
struct partition_entry *part_entry;
sos_vaddr_t buffer;
sos_size_t size = block_size;

ret = sos_blockdev_kernel_read (blkdev, 0, buffer, & size);
[...]

part_entry = (struct partition_entry *) (buffer + PART_TABLE_OFFSET);
```

`part_entry` peut maintenant être utilisé comme un tableau de 4 éléments, chaque élément étant un descripteur de partition. On commence donc par parcourir ces

4 éléments. Si le descripteur indique une taille de partition égale à zéro, alors on passe au descripteur suivant. Si le descripteur indique une partition de type étendue (`PART_TYPE_EXTENDED`), alors on enregistre dans une variable locale le début de cette partition, stocké dans le champ `lba` de la structure `partition_entry`. Il nous servira plus tard pour lire la *partition étendue* et ses *lecteurs logiques*. Sinon, on est dans le cas d'une partition primaire, que l'on enregistre dans le sous-système *blockdev* grâce à la fonction `sos_blockdev_register_partition()` :

```
for (partnum = 0; partnum < 4; partnum++)
{
  if (part_entry [partnum].size == 0)
    continue;

  if (part_entry [partnum].type == PART_TYPE_EXTENDED)
    {
      extstart = part_entry [partnum].lba;
      continue;
    }

  ret = sos_blockdev_register_partition (disk_class, disk_instance + partnum +
                                         blkdev, part_entry[partnum].lba,
                                         part_entry[partnum].size, NULL);
  [...]
}
```

Une fois les partitions de la table des partitions principales traitées, nous traitons la *partition étendue* en parcourant la liste chaînée des *lecteurs logiques* avec le code suivant :

```
while (extstart != 0 && partnum < 15)
  {
    ret = sos_blockdev_kernel_read (blkdev, (extstart + extsup) * block_size,
                                    (sos_luoffset_t) buffer, &
                                    & size);
    [...]

    ret = sos_blockdev_register_partition (disk_class,
                                           disk_instance + partnum + 1,
                                           blkdev,
                                           extstart + part_entry[0].lba,
                                           part_entry[0].size, NULL);
    [...]

    extsup = part_entry[1].lba;
    if (extsup == 0)
      break;
  }
```

Notre boucle est exécutée si une partition étendue était présente dans la table des partitions principales (`extstart != 0`) et tant que l'on n'a pas atteint 15 partitions, y compris les partitions primaires. En effet, comme vu en section 2.3, il n'y a que 15 mineurs disponibles pour chaque périphérique IDE. Chaque itération permet de traiter un *lecteur logique*, c'est-à-dire un élément de la liste chaînée de ces lecteurs.

On commence par lire le premier secteur de la partition étendue courante, situé au secteur `extstart + extsup`. `extstart` est le secteur de début de la partition étendue et `extsup` est le nombre de secteurs entre le début de la partition étendue et la mini-table des partitions courante (voir figure 12). Une fois ce secteur lu, on peut enregistrer dans le sous-système *blockdev* la partition décrite par le premier descripteur de la mini-table des partitions courante (`part_entry` pointe toujours sur le 446ème octet du buffer de lecture). Cette partition est enregistrée avec le même majeur que le disque dur la contenant et avec un mineur égal à `disk_instance + partnum + 1`, comme indiqué en section 2.3. Enfin, on récupère le numéro du secteur contenant la prochaine mini-table des partitions grâce au deuxième descripteur de la table des partitions courante (`extsup = part_entry[1].lba`). Si cet offset devient nul, alors on a atteint la fin de la liste chaînée des lecteurs logiques donc on s'arrête.

Ainsi, à l'issue de cette fonction `sos_part_detect()`, toutes les *partitions primaires* et tous les *lecteurs logiques* (dans la limite de 15 partitions) auront été détectés.

3.3 Tester dans *Qemu* et *Bochs*

Pour tester ce pilote de partitions dans un émulateur tel que *Qemu* ou *Bochs*, il va falloir partitionner le disque dur virtuel que nous avons créé en section 2.5. Pour cela, nous utiliserons bien entendu le programme `fdisk`. Par rapport à une utilisation habituelle de ce dernier, il faudra néanmoins donner quelques informations supplémentaires : la géométrie du disque. En effet, sur un disque dur réel, `fdisk` récupère cette information en interrogeant le pilote du disque dur. Ici, il faudra spécifier explicitement la géométrie. Voici la séquence de commandes `fdisk` (il s'agit d'utiliser les fonctions `c`, `h` et `s` du menu *expert*) :

```
$ /sbin/fdisk disk10M.img
Vous devez initialiser cylindres.
Vous pouvez faire cela à partir du menu des fonctions additionnelles.
AVERTISSEMENT: fanion 0x0000 invalide de la table de partitions 4 sera
corrigé par w(écriture)
Commande (m pour l'aide): x

Commande pour experts (m pour de l'aide): c
Numbre de cylindres (1-1048576): 320

Commande pour experts (m pour de l'aide): s
Numbre de secteurs (1-63, par défaut 63): 16
AVERTISSEMENT: initialisation du décalage de secteur pour compatibilité DOS

Commande pour experts (m pour de l'aide): h
Numbre de têtes (1-256, par défaut 255): 4

Commande pour experts (m pour de l'aide): r

Commande (m pour l'aide): p

Disque disk10M.img: 0 Mo, 0 octets
4 têtes, 16 secteurs/piste, 320 cylindres
Unités = cylindres de 64 * 512 = 32768 octets
```

Une fois ces informations spécifiées, on peut utiliser normalement la commande `n` pour créer une nouvelle partition, et la commande `d` pour en supprimer. Pour quitter, on utilisera la commande `w` qui sauvegardera la nouvelle table des partitions.

Ensuite, on pourra par exemple observer la table des partitions à l'aide du programme `hexdump` :

```
$ hexdump -n 64 -s 446 disk10M.img
00001be 0100 0001 0383 9910 0010 0000 2670 0000
00001ce 0000 9a01 0383 3f50 2680 0000 2980 0000
```

Pour tester ce disque partitionné dans *Qemu* ou *Bochs*, il suffit de suivre les indications données en section 2.5.

Pour les moins courageux et pour ceux qui persistent à utiliser windows, nous fournissons une image de disque dur de 10 Mo pré-partitionnée compressée : `extra/hd10M.img.gz`.

4 Démonstration

La démonstration de ce mois-ci n'est une nouvelle fois pas très originale : nous vous proposons de tester les pilotes de périphériques bloc en utilisant les commandes `dd` et `hexdump`, maintenant disponibles dans le *shell* SOS. Une autre commande existe, `spawn blktest`, qui effectue une série de tests de lectures/écritures sur toutes les partitions de `/dev/hda`. Vous pouvez également utiliser la commande `partdump` pour lister les partitions d'un disque dur. Libre à vous d'ajouter de nouvelles commandes et/ou d'améliorer le système !

Soyez-en avertis, les auteurs de ne sauraient être tenus responsables de toute perte de donnée si vous testez SOS sur machine réelle !

Conclusion

SOS dispose maintenant de deux fonctionnalités fondamentales d'un système d'exploitation : il sait intégrer des pilotes de périphérique caractère (décrits dans l'article précédent) et des pilotes de périphériques bloc. Dans cet article, nous avons détaillé la couche de gestion des périphériques bloc et leur intégration avec le VFS et la mémoire virtuelle. Nous avons vu que ce sous-système reposait principalement sur deux caches : *blockcache* pour accélérer les accès aux périphériques et *pagecache* pour assurer une propriété de cohérence entre les opérations `read()/write()` et `mmap()` en mode *partagé*. Nous avons ensuite présenté l'implémentation d'un pilote de périphérique bloc omniprésent dans le monde PC : le pilote des contrôleurs de disques durs IDE. Enfin, nous avons décrit comment lire la table des partitions d'un disque.

Nous faisons maintenant une pause d'un ou deux mois avant la prochaine étape de notre croisière aux longs cours. Nous reviendrons avec un article consacré à un système de fichiers... ou deux. D'ici là, digérez bien cet article ainsi que le code qui l'accompagne, et amusez-vous bien à massacrer tous les disques durs qui vous tombent sous la main !

The end.

Thomas Petazzoni et David Decotigny
`thomas.petazzoni@enix.org` et `d2@enix.org`
Site de SOS : `http://sos.enix.org`
Projet KOS : `http://kos.enix.org`

Á Maurice

Références

[1] T13. *Information Technology - AT Attachment with Packet Interface Extension (ATA/ATAPI-4)*. `http://www.t13.org/project/d1153r18-ATA-ATAPI-4.pdf`, 1998.

[2] Page sur le fonctionnement d'un disque dur. `http://fr.wikipedia.org/wiki/Disque_dur`.

[3] Hans-Peter Messmer. *The indispensable PC Hardware book*. Number ISBN 0201596164. Addison-Wesley Professional, 2001.