Croisière au cœur d'un OS*
Étape "9" : Pilotes de périphériques *caractère*

Résumé

Après une longue pause, SOS est de retour dans GLMF. Dans le précédent article, nous avions étudié la mise en place d'une infrastructure pour le support de systèmes de fichiers : le *VFS*. Ce mois-ci, nous allons détailler l'implémentation de plusieurs pilotes de périphériques de type *caractère*, leur intégration dans SOS, et leur mise à disposition aux applications utilisateur par l'intermédiaire du *VFS*.

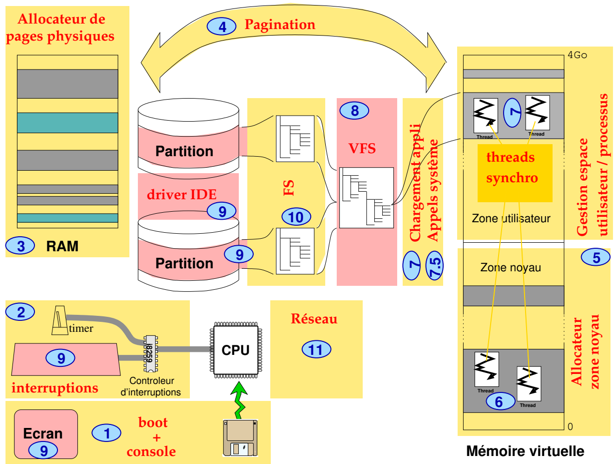

FIG. 1 – Programme des articles

Introduction

Jusqu'ici, les interactions de SOS et de ses applications utilisateur avec le matériel étaient limitées à un pilote console rudimentaire et une sortie de déboguage *port 0xe9*. Dans cet article, nous vous proposons d'étudier l'implémentation de pilotes de périphériques de type *caractère* qui permettront d'utiliser le clavier, le port série et la console. Au travers de la description de ces pilotes de périphériques, nous présenterons différents mécanismes permettant de dialoguer avec le matériel : ports d'entrées-sorties et IRQs.

Ces pilotes de périphériques proposeront une interface simple d'accès aux périphériques qui ne sera pas directement utilisable depuis les applications utilisateur. Afin de les rendre accessibles à ces applications, SOS propose une méthode similaire à celle d'Unix : l'accès aux périphériques se fait aux travers de fichiers spéciaux utilisables avec les appels système classiques `open()`, `read()`, `write()`, `close()`. Nous présenterons donc l'infrastructure permettant de rendre accessibles les différents périphériques par l'intermédiaire du VFS étudié dans l'article précédent.

En pratique, à la fin de cet article, les applications utilisateur pourront récupérer des frappes de touche du clavier et afficher des caractères sur une console ou un port série. Grâce à ceci, nous proposerons en guise de démonstration un petit shell, accessible *via* la console clavier/écran ou *via* le port série, et qui permettra d'exécuter des commandes simples.

1 Infrastructure d'accès aux périphériques

1.1 Types de périphériques

Dans les systèmes Unix comme dans SOS, on distingue deux grandes classes de périphériques : les périphériques de type *caractère* et les périphériques de type *bloc*.

Les périphériques de type *caractère* peuvent être assimilés à un flux infini d'octets. Par exemple, un port série est un périphérique caractère : on peut y lire ou écrire un nombre infini d'octets, octet par octet. Dans la plupart des périphériques caractère, il n'est pas possible de se déplacer (appel système `seek()`). En effet, il est difficilement concevable de rembobiner le flux d'octets à destination ou en provenance d'un port série ou d'une carte son. Toutefois, certains périphériques caractère spéciaux permettent de rembobiner, comme par exemple `/dev/zero`, un périphérique caractère dans lequel tout ce qui est écrit part aux oubliettes, et duquel on ne lit que des zéros.

A contrario, les périphériques de type *bloc* 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. Puisqu'on peut identifier chaque bloc du périphérique d'une manière unique (par sa position dans le périphérique), on peut utiliser la mémoire physique (RAM) de l'ordinateur comme zone tampon des blocs les plus fréquemment accédés. Ceci n'est en général pas possible avec les périphériques de type caractère.

Ces deux types de périphériques fonctionnant de manière différente, ils seront gérés par deux sous-systèmes indépendants : *chardev* et *blockdev*. En dehors de la première section, dans cet article, nous nous limiterons à l'étude des périphériques de type *caractère* et à la description du sous-système *chardev*. Les périphériques de type *bloc* et leur intégration dans SOS par l'intermédiaire du sous-système *blockdev* seront détaillés dans le prochain article de la série.

*La version originale de cet article a été publiée dans GNU Linux Magazine France numéro 79 – Janvier 2006 (`http://www.linuxmag-france.org`) et cette version est diffusée avec l'autorisation de l'éditeur.

1.2 Interaction de l'utilisateur avec les pilotes de périphériques

La figure 2 présente les différents sous-systèmes impliqués dans la mise à disposition des périphériques *caractère* aux applications utilisateur.

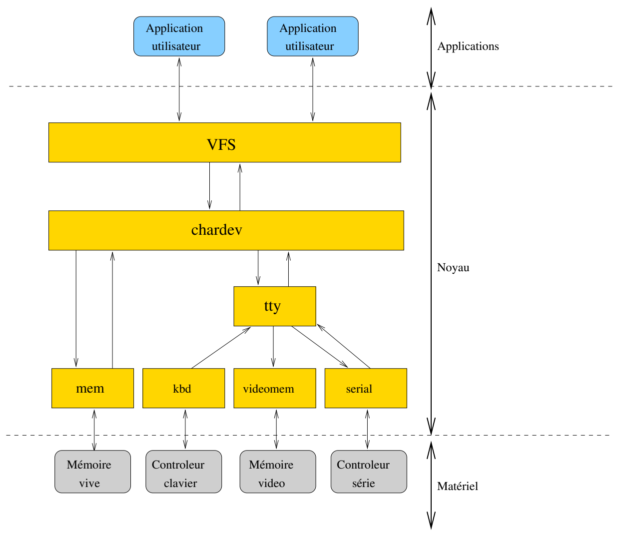

FIG. 2 – Schéma des différents sous-systèmes en jeu dans la mise à disposition des périphériques *caractère* aux applications utilisateur.

Pour faire le lien entre les nœuds des fichiers spéciaux présents dans les systèmes de fichiers et les fonctions implémentées par les pilotes de périphériques, on utilise un système de clefs : les fameux "majeurs"/"mineurs" Unix. Le "majeur" est un entier représentant la "classe" de ressources que le pilote sait gérer (par exemple : les disques IDE, les ports série, etc.). Le mineur est un entier qui représente l'"instance" de la ressource à laquelle on accède (par exemple : la première partition du disque IDE primaire du premier contrôleur, le premier port série de la machine, etc.). Dans SOS, le couple majeur/mineur définit la structure `sos_fs_dev_id_t` déclarée dans `sos/fs.h` :

```
struct sos_fs_dev_id_t
{
  sos_ui32_t device_class;    /**< aka "major" */
  sos_ui32_t device_instance; /**< aka "minor" */
} dev_id;
```

Le couple majeur/mineur ainsi que le type de périphérique (caractère/bloc) sont stockés dans le système de fichiers. C'est la fonction `mknod()` qui s'occupe de créer les nœuds pour ces fichiers spéciaux dans le système de fichiers :

```
sos_ret_t sos_fs_mknod(const struct sos_process * creator,
       const char * _path,
       sos_size_t _pathlen,
       sos_fs_node_type_t type /* only block/char allowed */,
       sos_ui32_t access_rights,
       const struct sos_fs_dev_id_t * devid)
{
  sos_ret_t retval;
  struct sos_fs_pathname path;
  struct sos_fs_nscache_node * nsnode;
  struct sos_fs_node * fsnode;

  path.contents = _path;
  path.length   = _pathlen;

  retval = fs_create_node(& path, creator, access_rights,
  type, & nsnode);
  fsnode = sos_fs_nscache_get_fs_node(nsnode);
  fsnode->dev_id.device_class    = devid->device_class;
  fsnode->dev_id.device_instance = devid->device_instance;

  sos_fs_nscache_unref_node(nsnode);
  return SOS_OK;
}
```

Le système de fichiers ne fait donc bien que *stocker* un nœud de fichier spécial sur le disque. Ce n'est pas lui qui implémente les pilotes de périphériques sinon il faudrait par exemple un pilote de port série dans le code du système de fichiers FAT, un autre dans le code de ext2, etc. Les pilotes de périphériques sont globaux à l'ensemble du système et c'est le système d'exploitation qui s'occupe de faire le lien entre les fonctions `read()`/`write()`/... que l'utilisateur appelle sur le VFS, et les fonctions du pilote de périphérique associées au majeur/mineur. Ce lien est précisément établi par le système d'exploitation au moyen du couple majeur/mineur. Dans le cas où aucun pilote de périphérique n'est associé au couple majeur/mineur, les fonctions `read()`/`write()`/... du VFS renverront `SOS_ENOSUP`. À l'inverse, si un pilote est implémenté dans le système d'exploitation, rien n'oblige un quelconque système de fichiers à avoir un fichier spécial pour le majeur/mineur associé.

Pour tout cela, le système d'exploitation gère deux dictionnaires `majeur/mineur` → `fonctions du pilote de périphérique` : un dictionnaire pour les périphériques caractère, l'autre pour les périphériques bloc.

1.3 Sous-système *chardev*

1.3.1 Description d'un pilote de périphérique caractère

D'un point de vue pratique, les pilotes de périphériques de type *caractère* doivent utiliser les structures et fonctions décrites dans `sos/chardev.h`. Un tel pilote doit enregistrer *une classe de périphérique* en l'associant à un ensemble d'opérations défini par la structure `sos_chardev_ops`. Ces opérations seront utilisées par le sous-système *chardev* lors des appels au *VFS* effectués par une application utilisateur sur un périphérique caractère de la *classe* concernée. L'interface `struct sos_chardev_ops` qu'un pilote de périphérique caractère doit fournir est constituée des opérations :

- **open** est une opération obligatoire, appelée à chaque fois qu'un périphérique de la classe considérée est ouvert par une application ;
- **close** est une opération optionnelle, appelée si elle existe lors de la fermeture d'un périphérique ;
- **seek** est une opération optionnelle qui permet de se déplacer dans le périphérique. Pour la plupart des périphériques caractère, elle ne sera pas implémentée puisqu'elle n'a pas de sens ;
- **read** est une opération optionnelle qui permet de lire un certain nombre d'octets depuis le périphérique caractère ;
- **write** est une opération optionnelle qui permet d'écrire sur le périphérique caractère ;
- **mmap** est une opération optionnelle appelée lorsqu'une application souhaite projeter en mémoire le périphérique caractère. Pour la plupart des périphériques caractère (clavier, console, port série), cela n'a pas de sens, mais pour d'autres (comme `/dev/zero` ou `/dev/mem`), cette opération a un sens ;

– **ioctl** est une opération optionnelle qui permet au pilote de périphérique d'implémenter des fonctions spécifiques et de les mettre à disposition des applications utilisateur. Ainsi, un pilote de port série pourra mettre à disposition des opérations **ioctl** pour changer la vitesse du port série, pour activer ou non l'utilisation du bit de parité, etc.

Un pilote de périphérique caractère est enregistré dans le système par un appel à la fonction `sos_chardev_register_class()`. Celle-ci ajoute la description de la classe de périphérique caractère (`struct sos_chardev_class`) dans le dictionnaire du sous-système *chardev* (`registered_chardev_classes`) :

```
struct sos_chardev_class
{
  sos_ui32_t                 device_class;
  struct sos_chardev_ops    *ops;
  void                      *custom_data;
  ...

  struct sos_chardev_class *next, *prev;
};


sos_ret_t
sos_chardev_register_class (sos_ui32_t device_class,
                            struct sos_chardev_ops *ops,
                            void * custom_data)
{
  struct sos_chardev_class *chardev;

  /* Allocate and initialize a new device description */
  chardev = (struct sos_chardev_class *)
    sos_kmalloc(sizeof(struct sos_chardev_class), 0);

  chardev->device_class = device_class;
  chardev->custom_data  = custom_data;
  chardev->ops          = ops;
  chardev->ref_cnt      = 1;

  /* insert it into the list */
  list_add_tail (registered_chardev_classes, chardev);

  return SOS_OK;
}
```

1.3.2 Intégration dans le *VFS*

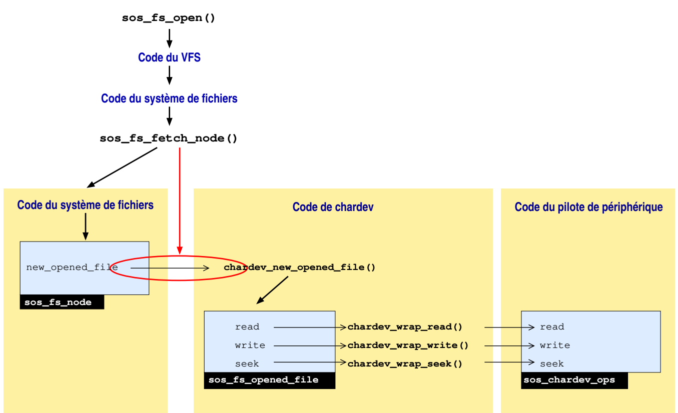

FIG. 3 – Principe de l'intégration de *chardev* dans le *VFS*. En rouge : l'étape de déroutement mentionnée dans le texte.

Lorsque l'utilisateur veut accéder à un fichier, qu'il soit quelconque ou spécial de type caractère, il commence par faire un appel à `open()`. Ceci créé la ressource "fichier ouvert" associée au fichier (voir la figure 3). Dans le cas d'un fichier spécial de type caractère, ce n'est pas le code du système de fichiers qui doit piloter le périphérique. C'est donc la fonction `chardev_helper_new_opened_file()` de `sos/chardev.c` qui créé la ressource "fichier ouvert" pour le fichier spécial de type caractère, et non pas une quelconque fonction du système de fichiers.

Comment le noyau sait-il qu'il doit appeler `chardev_helper_new_opened_file()` ? Tout débute dès la phase de récupération du nœud depuis le disque, dans la fonction `fs_fetch_node()` de `sos/fs.c` étudiée à l'article précédent. Lorsque le système de fichiers récupère un nœud de type "fichier spécial" de pilote de type périphérique caractère, le *VFS* force les méthodes `sync()`, `new_opened_file()` et `close_opened_file()` de `struct sos_fs_node` à pointer vers des fonctions spécifiques du sous-système *chardev*, dont `chardev_helper_new_opened_file()` fait partie. Ceci est fait par l'intermédiaire de la fonction `sos_chardev_helper_ref_new_fsnode()` de `sos/chardev.c`, appelée par `fs_fetch_node()` :

```
/* sos/fs.c */
sos_ret_t fs_fetch_node(struct sos_fs_manager_instance *fs,
                        sos_ui64_t storage_location,
                        struct sos_fs_node ** result_fsnode)
{
  ...
  if (SOS_FS_NODE_DEVICE_CHAR == (*result_fsnode)->type)
    sos_chardev_helper_ref_new_fsnode(*result_fsnode));
  ...
}

...

/* sos/chardev.c */
static sos_ret_t
sos_chardev_helper_ref_new_fsnode(struct sos_fs_node * this)
{
  this->sync              = chardev_helper_sync;
  this->new_opened_file   = chardev_helper_new_opened_file;
  this->close_opened_file = chardev_helper_close_opened_file;

  return SOS_OK;
}
```

Le rôle de `chardev_helper_new_opened_file()` est :

1. De récupérer la description du périphérique caractère associé au majeur du fichier spécial. Ceci se fait en consultant le dictionnaire du sous-système *chardev* (voir la section précédente).
2. De créer la structure "fichier ouvert" propre à un fichier de type périphérique caractère. Il s'agit d'une structure qui hérite de la structure "fichier ouvert" traditionnelle (`struct sos_fs_opened_file`, *via* le champ `super`) et qui contient un champ supplémentaire renvoyant aux fonctions définies par le pilote de périphérique (`sos_chardev_ops`, *via* le champ `class`) :

   ```
   struct sos_chardev_opened_file
   {
     struct sos_fs_opened_file  super;
     struct sos_chardev_class  *class;
   };
   ```

   Cette technique permet de laisser le champ `custom_data` de `struct sos_fs_opened_file` disponible pour que le pilote de périphérique puisse y stocker ce qu'il veut.
3. D'appeler la méthode `open()` du pilote de périphérique. Le pilote sera ainsi averti qu'une application va accéder au périphérique.
4. D'indiquer les adresses des fonctions `read()`/ `write()`/`seek()`/`ioctl()`... de `struct sos_fs_ops_opened_file` et `sos_fs_ops_opened_chardev` qui seront appelées par le *VFS*. Ces fonctions s'occuperont d'appeler les méthodes `read()`/`write()`/`seek()`/`ioctl()`... du pilote de périphérique.

Par exemple, la fonction correspondant au `read()` du *VFS* et appelant le `read()` du pilote de périphérique est écrite de la façon suivante :

```
static sos_ret_t chardev_wrap_read(struct sos_fs_opened_file *this,
   sos_uaddr_t dest_buf,
   sos_size_t * /* in/out */len)
{
  struct sos_chardev_opened_file *chardev_of
    = ((struct sos_chardev_opened_file*)this);

  struct sos_chardev_class * chardev = chardev_of->class;
  return chardev->ops->read(this, dest_buf, len);
}
```

Pour résumer, le *VFS* appelle la fonction `fs_fetch_node()` qui court-circuite le code du système de fichiers lorsqu'il rencontre un nœud de fichier spécial de périphérique caractère. Le but est de détourner les appels `read()`/`write()`... du *VFS* vers les fonctions du pilote de périphérique caractère (pour un fichier normal, ces fonctions seraient celles du système de fichiers). Ce sont les fonctions de type `chardev_wrap_read()` dans `sos/chardev.c` qui réalisent ce détournement.

2 Pilote *mem* et *zero*

Dans l'article 7 et demi (numéro 72 de GLMF), nous émulions les périphériques caractère `/dev/zero`, `/dev/null`, `/dev/mem` et `/dev/kmem` à l'aide d'un appel système particulier : `fakemmap()`. Maintenant qu'une infrastructure permettant d'intégrer proprement des périphériques caractère au *VFS* est disponible, ces quatre périphériques particuliers ont été modifiés pour utiliser cette nouvelle infrastructure.

2.1 *mem*

`/dev/mem` (représentation de la mémoire physique) et `/dev/kmem` (représentation de la mémoire du noyau) sont toujours implémentés dans `drivers/mem.c`. Ces deux périphériques spéciaux font partie de la même classe, `SOS_CHARDEV_MEM_MAJOR`, et disposent chacun d'un numéro d'instance : `SOS_CHARDEV_KMEM_MINOR` pour `/dev/kmem` et `SOS_CHARDEV_PHYSMEM_MINOR` pour `/dev/mem`. La fonction `sos_dev_mem_chardev_setup()` enregistre cette classe de périphérique et l'association à l'ensemble d'opérations défini dans la structure `dev_mem_fs_ops`. Cinq opérations sont implémentées :

- `open()`, dans `dev_mem_fs_open()`, qui, en fonction de l'instance du périphérique caractère à ouvrir positionne le champ `custom_data` de la structure `fsnode`. Ce champ contient la taille du périphérique caractère considéré, à savoir `SOS_PAGING_BASE_USER_ADDRESS` (début de l'espace virtuel réservé à l'espace utilisateur, donc fin de l'espace virtuel réservé au noyau) pour `/dev/kmem` et `ram_pages << SOS_PAGE_SHIFT` (taille de la mémoire physique en octets) pour `/dev/mem`. Cette information permettra à l'opération `seek()` de vérifier si le déplacement dans le périphérique est valide ou non ;
- `seek()`, dans `dev_mem_fs_seek()`, qui change la position courante dans le périphérique. En effet, `/dev/mem` et `/dev/kmem` sont les rares périphériques caractère pour lesquels cette opération a un sens. Les classiques `SOS_SEEK_SET`, `SOS_SEEK_CUR` et `SOS_SEEK_END` sont gérés par cette fonction. Cette dernière vérifie également que le déplacement n'a pas lieu en dehors des limites du périphérique, grâce à la taille stockée dans le champ `custom_data` par l'opération `open()`. La macro `GET_DEV_SIZE` est utilisée pour la récupérer ;
- `read()` et `write()` sont implémentées à partir de la fonction `dev_mem_fs_access()`, permettant la lecture et l'écriture. Pour `/dev/mem`, cette fonction mappe temporairement la page de la mémoire physique à écrire ou lire dans l'espace virtuel du noyau. Pour cela, une page virtuelle est allouée au début de la fonction en utilisant `sos_kmem_vmm_alloc()`, puis à chaque itération de la boucle principale (qui s'exécute pour chaque page à copier), la page de la mémoire physique à lire ou écrire est mappée dans cette page virtuelle en utilisant `sos_paging_map()`. En ce qui concerne `/dev/kmem`, les choses sont plus simples, puisque la mémoire virtuelle du noyau est déjà accessible. La seule vérification, c'est tester si l'adresse à laquelle on souhaite lire ou écrire est bien mappée (appel à `sos_kmem_is_valid_vaddr()`). Si elle n'est pas mappée, alors la boucle se termine et la lecture ou l'écriture ne sera que partielle.
- `mmap()` est implémentée en utilisant les primitives `sos_dev_physmem_map()` et `sos_dev_kmem_map()` déjà présentées à l'article 7 et demi.

2.2 *zero*

Comme dans l'article 7 et demi, les périphériques `/dev/zero` et `/dev/null` sont implémentés dans `drivers/zero.c`. Cette classe de périphérique a le numéro `SOS_CHARDEV_ZERO_MAJOR`, et les deux périphériques ont respectivement les numéros d'instance `SOS_CHARDEV_NULL_MINOR` et `SOS_CHARDEV_ZERO_MINOR`.

La fonction `sos_dev_zero_subsystem_setup()` a été modifiée pour enregistrer cette classe de périphérique, et l'associer aux opérations listées dans la structure `sos_zero_fs_ops`. L'implémentation de ces opérations est très simple :

- `open()` vérifie simplement que le numéro d'instance du périphérique à ouvrir est bien valide (soit `SOS_CHARDEV_NULL_MINOR` soit `SOS_CHARDEV_ZERO_MINOR`);
- `seek()` met à jour la position courante dans le fichier. Aucune vérification n'est nécessaire car `/dev/zero` et `/dev/null` sont des fichiers n'ayant ni début, ni fin. Une histoire sans queue ni tête en somme ;
- `read()` ne retourne rien dans le cas de `/dev/null` car on ne peut pas lire depuis ce périphérique. Pour `/dev/zero` en revanche, le buffer destination est rempli de zéros ;
- `write()` implémente les oubliettes : il ne fait que mettre à jour la position courante dans le fichier. Les données à écrire sont purement et simplement ignorées ;
- `mmap()` implémente la projection en mémoire virtuelle utilisateur. Cette opération réutilise simplement la fonction `sos_dev_zero_map()` déjà présente dans les articles précédents, lorsque l'astuce `fakemmap()` était utilisée.

3 Pilote *tty*

Les terminaux sont des périphériques caractère classiquement utilisés pour dialoguer avec le système. Les terminaux peuvent être par exemple un couple clavier/écran ou une ligne série. Afin de factoriser le code générique de gestion d'un terminal, SOS propose le sous-système *tty*. Comme le montre la figure 4, ce sous-système repose sur le pilote clavier et le pilote écran pour proposer une console clavier/écran (qui sera `/dev/tty`) et sur le pilote série pour proposer une console série (qui sera `/dev/ttyS`).

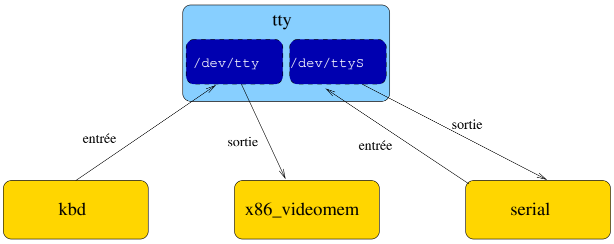

FIG. 4 – Le sous-système *tty* permet de mettre à disposition du système plusieurs terminaux, ces terminaux reposant sur d'autres pilotes de périphériques.

Le sous-système *tty* est implémenté dans `drivers/tty.c`. Chaque terminal est représenté en interne par une structure `struct tty_device` :

```
struct tty_device {
  sos_ui32_t        instance;
  unsigned int      open_count;
  char              buffer[TTY_BUFFER_LEN];
  unsigned int      buffer_read;
  unsigned int      buffer_write;
  struct sos_ksema  sem;
  struct sos_kwaitq wq;
  sos_ret_t         (*write)(char c);
  unsigned int      param;
  struct tty_device *next, *prev;
};
```

`open_count` permet de compter le nombre d'ouvertures d'un terminal. `buffer` est un buffer utilisé pour stocker les caractères lus depuis le terminal. Ce buffer étant circulaire, la structure comporte également un pointeur de lecture, `buffer_read`, et un pointeur d'écriture, `buffer_write`. Un sémaphore `sem` est utilisé pour protéger le terminal des accès concurrents, et une file d'attente `wq` permet de mettre en attente les threads qui font une lecture alors qu'il n'y a pas encore assez de caractères dans le buffer. `write` est un pointeur vers la fonction à utiliser pour écrire des caractères sur ce terminal.

La fonction `tty_init()` enregistre une nouvelle classe de périphérique : `SOS_CHARDEV_TTY_MAJOR`. À l'initialisation, aucun terminal n'existe : la liste `tty_device_list` est vide.

Pour créer un terminal, il faut utiliser la fonction `tty_create()` en spécifiant un numéro d'instance (le *mineur*) et la fonction à utiliser pour l'écriture de caractères sur le terminal. Le terminal nouvellement créé est ajouté à la liste globale, et un pointeur vers la `struct tty_device` est retourné.

La fonction `tty_open()` est appelée lors de l'ouverture d'un terminal. Elle recherche la structure `tty_device` correspondant à l'instance demandée, en utilisant la fonction interne `tty_device_find()`. Si un terminal correspondant est trouvé, alors le champ `custom_data` de la structure `sos_fs_opened_file` est positionné pour pointer vers la structure `tty_device` afin de faciliter le travail des autres opérations. De plus, le compteur d'ouvertures `open_count` est utilisé pour réinitialiser le buffer du terminal et les pointeurs de lecture/écriture lors de la première ouverture.

3.1 Configuration du terminal

Sous Unix, les terminaux sont très configurables, au moyen des fonctions dites *termios*. La page de manuel associée liste les différentes options disponibles dans un noyau tel que Linux. Évidemment, SOS n'implémente pas toutes les options et tous les modes, mais seulement deux :

- Le mode non-canonique est un mode dans lequel une lecture sur un terminal retourne lorsque le premier retour chariot a été reçu. En mode canonique, on retourne dès que suffisamment de caractères ont été lus ;
- L'option d'écho permet de préciser si les caractères saisis dans le terminal doivent être immédiatement affichés, ou bien si l'application s'en chargera.

Pour configurer ces modes, le pilote de terminal implémente l'appel `ioctl()` dans la fonction `tty_ioctl()`. `SOS_IOCTL_TTY_SETPARAM` permet d'activer un mode ou une option, et `SOS_IOCTL_TTY_RESETPARAM` permet de désactiver un mode ou une option. Afin de permettre aux applications utilisateur de configurer les terminaux, les définitions nécessaires sont disponibles dans le fichier `drivers/devices.h`. Celui-ci est conçu pour être inclus à la fois dans le noyau et dans une application utilisateur.

3.2 Lecture depuis un terminal

Pour l'implémentation de la lecture, deux fonctions sont utilisées : `tty_read()` implémente l'opération de lecture proprement dite, tandis que `tty_add_chars()` doit être appelée par le pilote du périphérique d'entrée à chaque fois qu'un nouveau caractère est disponible. Le cœur de la fonction `tty_read()` est constitué du code suivant :

```
1.
2.  while (1)
3.  {
4.   if (t->buffer_read == t->buffer_write)
5.    sos_kwaitq_wait (& t->wq, NULL);
6.
7.   sos_memcpy_to_user
8.     (dest_buf, (sos_vaddr_t) & t->buffer[t->buffer_read],
9.      sizeof (char));
10.
11.  dest_buf++;
12.
13.  t->buffer_read++;
14.  if (t->buffer_read == TTY_BUFFER_LEN)
15.   t->buffer_read = 0;
16.
17.  count++;
18.
19.  if (t->param & SOS_IOCTLPARAM_TTY_ECHO)
20.   t->write (c);
21.
22.  if (count == *len
23.      || (c == '\n'
24.          && t->param & SOS_IOCTLPARAM_TTY_CANON))
25.   break;
26. }
27. *len = count;
```

À l'entrée dans la boucle, la fonction bloque sur la liste d'attente si aucun caractère n'est disponible dans le buffer d'entrée, c'est-à-dire si le pointeur de lecture (`buffer_read`) est égal au pointeur d'écriture (`buffer_write`), lignes 4 et 5. Ceci permet de bloquer le thread ayant demandé cette lecture. Dès qu'un caractère devient disponible, le thread est réveillé et poursuit son exécution ligne 7. Il recopie alors ce caractère depuis le buffer du terminal vers le buffer de destination. Ce dernier étant situé en espace utilisateur, la fonction `sos_memcpy_to_user()` doit donc être utilisée pour effectuer la copie. Le pointeur dans le buffer destination est incrémenté ligne 11. Ligne 13, c'est le pointeur de lecture dans le buffer du terminal qui est incrémenté. Lignes 14 et 15, on teste si l'on est arrivé au bout du buffer du terminal, et si c'est le cas on revient au début : c'est le principe du buffer circulaire. Lignes 19 et 20, on affiche le caractère saisi sur la sortie du terminal, si le paramètre `SOS_IOCTLPARAM_TTY_ECHO` est positionné. Enfin, lignes 22 et 23, on sort de la boucle si suffisamment de caractères ont été lus (`count == *len`) ou si l'on a rencontré un retour chariot et que le paramètre `SOS_IOCTLPARAM_TTY_CANON` est positionné.

De l'autre côté, la fonction `tty_add_chars()` est utilisée par le pilote du périphérique d'entrée pour ajouter des caractères dans le buffer du terminal et réveiller un éventuel thread en attente de caractères (bloqué dans `tty_read()`).

```
1. void tty_add_chars (struct tty_device *t, const char *s)
2. {
3.  while (*s)
4.  {
5.   t->buffer[t->buffer_write] = *s;
6.   t->buffer_write++;
7.   if (t->buffer_write == TTY_BUFFER_LEN)
8.    t->buffer_write = 0;
9.   s++;
10. }
11.
12. sos_kwaitq_wakeup (& t->wq, SOS_KWQ_WAKEUP_ALL, SOS_OK);
13.}
```

En réalité, la fonction `tty_add_chars()` permet d'ajouter plusieurs caractères en une seule fois : le pilote du périphérique d'entrée lui passe une chaîne `s` terminée par un caractère nul. Cette fonction stocke chaque caractère dans le buffer du terminal, à l'endroit indiqué par le pointeur d'écriture `buffer_write` (ligne 5). À chaque ajout de caractère, elle met à jour ce pointeur et le fait revenir à zéro quand on atteint la fin du buffer (lignes 6 à 8). Ainsi, les caractères les plus anciens du buffer circulaire peuvent être écrasés par les caractères nouvellement ajoutés s'ils n'ont pas pu être récupérés à temps par la fonction `tty_read()`. En fin de fonction, ligne 12, les éventuels threads en attente de caractères sont réveillés. Ce processus est illustré dans la figure 5.

3.3 Écriture sur un terminal

L'écriture sur un terminal est beaucoup plus simple. La fonction `tty_write()` parcourt les caractères du buffer utilisateur. À chaque itération, la fonction recopie le caractère en mode noyau en utilisant `sos_memcpy_from_user()`, puis l'affiche dans le terminal en utilisant le pointeur de fonction `write()` disponible dans la structure `tty_device`.

4 Terminal clavier/écran

Le terminal clavier/écran utilise deux pilotes de périphériques séparés : le pilote clavier pour l'entrée, et le pilote console pour la sortie.

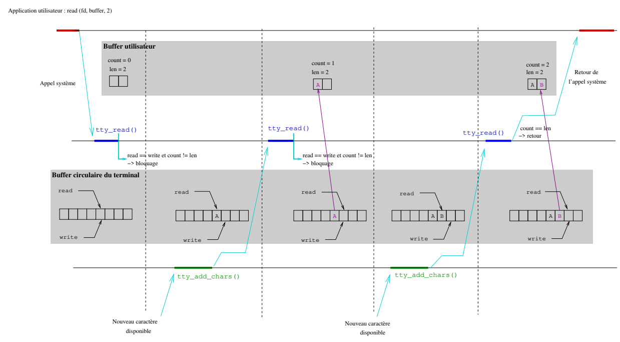

FIG. 5 – Une application utilisateur effectue un appel système `read()` pour lire 2 octets. Au début, aucun caractère n'est disponible dans le buffer du terminal, donc la fonction `tty_read()` bloque. Au moment où un nouveau caractère devient disponible dans le pilote de périphérique d'entrée, celui-ci appelle `tty_add_chars()` qui ajoute le caractère dans le buffer du terminal et réveille le thread bloqué dans `tty_read()`. Une fois réveillé, ce thread transfère le nouveau caractère depuis le buffer du terminal vers le buffer de l'application. Le thread constate ensuite qu'il n'y a plus de caractères disponibles et qu'il n'a pas encore lu les deux caractères demandés par l'application. Il bloque donc à nouveau. Le même processus se déroule, sauf que cette fois-ci, lorsque le thread est réveillé, après avoir transféré le nouveau caractère, il constate qu'il a bien lu les deux caractères demandés : l'appel système se termine et l'application peut utiliser les deux caractères lus.

4.1 Pilote clavier

Le pilote clavier de SOS est implémenté dans `drivers/kbd.c`. Il est initialisé grâce à la fonction `sos_kbd_init()` à laquelle il faut passer la structure `struct tty_device` du terminal connecté au clavier. Cette information est stockée dans une variable globale qui sera utilisée plus tard comme paramètre à `tty_add_chars()` pour ajouter des caractères dans le buffer du terminal clavier/écran.

Par ailleurs, cette fonction d'initialisation enregistre un gestionnaire pour l'interruption `SOS_IRQ_KEYBOARD`. Cette interruption est déclenchée lors de l'appui ou du relâchement d'une touche du clavier. C'est donc dans ce gestionnaire `kbd_irq_handler()` que se trouve le cœur du pilote clavier.

Ce gestionnaire d'interruption commence par récupérer le *scancode* de la touche qui a été pressée ou relâchée. Pour cela, elle lit un octet sur le port d'entrée/sortie `KBD_DATA_PORT` (`0x60`). Sur *x86*, on accède aux ports du bus d'entrées/sorties à l'aide d'instructions spécialisées comme `inb()` et `outb()`. Ici, notre pilote clavier effectue donc l'opération :

```
scancode = inb (KBD_DATA_PORT)
```

Ce *scancode* ne correspond pas au code ASCII de la touche qui a été pressée. Il s'agit d'un code qui est lié à la position géographique de la touche sur le clavier, indépendant du *keymap*. Ainsi, dans la figure 6 représentant l'association entre un *keymap* `azerty` et les *scancodes*, on peut voir que le *scancode* `16` est associé à la touche *A*. Sur un clavier `qwerty`, ce sera la touche `Q` qui aura le scancode `16`, qui sera donc associé au code ASCII `Q`.

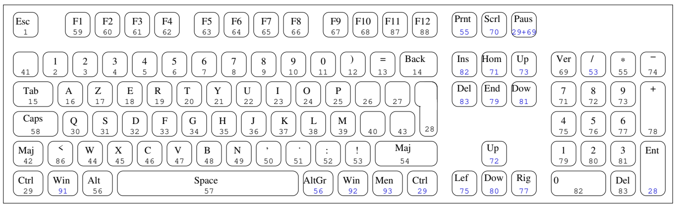

FIG. 6 – Les *scancodes* et les touches associées sur un clavier *Azerty*. Les *scancodes* en bleu sont des *scancodes étendus*.

Une fois le *scancode* récupéré, notre gestionnaire d'interruption détermine s'il s'agit d'un *scancode* étendu. Les *scancodes* étendus sont utilisés pour quelques touches spéciales (par exemple pour les touches *Windows* et *Menu*). Au lieu d'une seule interruption, l'appui sur une de ces touches en génère deux. La première enverra le *scancode* `0xE0` (`KBD_EXTENDED_SCANCODE`, `224` en décimal) et la deuxième enverra le *scancode* réel. Notre gestionnaire d'interruption ignore de tels *scancodes* étendus en utilisant la variable globale `is_ext`. Lorsque le *scancode* spécial `0xE0` est lu, la variable `is_ext` est positionnée à vrai. Lorsqu'un autre *scancode* est lu mais que `is_ext` est vrai, `is_ext` est repassée à faux et on ignore le scancode. Le code simplifié de cette opération est le suivant :

```
void kbd_irq_handler (int irq_level)
{
 scancode = inb (KBD_DATA_PORT);

 if (scancode == KBD_EXTENDED_SCANCODE)
  {
   is_ext = TRUE;
  }
 else if (is_ext)
  {
   is_ext = FALSE;
  }
 else
 {
  /* Gestion d'un scancode normal */
 }
}
```

Ce problème de gestion de *scancode étendu* a donc été simplifié à l'extrême dans SOS. Un pilote clavier complet devra prendre en compte ces touches spécifiques.

Une fois ce problème écarté, il faut traiter les *scancodes* normaux. Ceux-ci sont de deux types : les *makecodes* et les *breakcodes*. Les premiers sont envoyés lorsqu'une touche est pressée, les seconds lorsqu'une touche est relâchée. Les macros `KBD_IS_MAKECODE()`, `KBD_IS_BREAKCODE()` et `KBD_BREAKCODE_2_MAKECODE()` permettent respectivement de déterminer si un *scancode* est un *makecode* ou un *breakcode*, et de convertir un *breakcode* en *makecode*.

Lors de l'appui sur une touche, il faut convertir le *makecode* dans le code ASCII correspondant. Pour cela, nous utilisons une table stockée dans le fichier `drivers/kbdmapfr.c`. Cette table associe à chaque *scancode* une séquence de caractères. Dans la plupart des cas, un *scancode* correspond à un unique caractère (une lettre ou un chiffre, par exemple), sauf dans le cas des touches spéciales comme les touches de fonctions ou les flèches de direction. Pour ces touches spéciales, des séquences spécifiques de caractères sont ajoutées au buffer du terminal. Par exemple, lorsque la touche `F1` est pressée, c'est la séquence de caractères `\eOP` qui est ajoutée au buffer du terminal. Les séquences utilisées par SOS sont similaires à celles de Linux et sont définies par des normes, telles *VT100* et *VT220* par exemple. Pour plus d'informations, vous pouvez consulter le site *VT100.net* [1] ou une page expliquant la relation entre ces normes et les émulations de terminaux sous Unix [2].

En réalité, ce fichier `drivers/kbdmapfr.c` contient deux tables : une utilisée lorsque la touche *Shift* n'est pas enfoncée, et une utilisée lorsque *Shift* est enfoncée. Le code suivant permet de déterminer la séquence de caractères associée à la touche qui a été pressée ou relâchée :

```
1.  if (KBD_IS_MAKECODE(scancode))
2.   {
3.    if ((scancode == KBD_LEFTSHIFT_SCANCODE) ||
4.        (scancode == KBD_RIGHTSHIFT_SCANCODE))
5.          kbd_current_translate_table
6.            = kbd_shift_translate_table;
7.    else if (kbd_current_translate_table[scancode])
8.      key = kbd_current_translate_table[scancode];
9.   }
10.
11. else
12.  {
13.   scancode_t makecode = KBD_BREAKCODE_2_MAKECODE(scancode);
14.
15.   if ((makecode == KBD_LEFTSHIFT_SCANCODE) ||
16.       (makecode == KBD_RIGHTSHIFT_SCANCODE))
17.         kbd_current_translate_table
18.           = kbd_regular_translate_table;
19.  }
```

Lignes 1 à 9, le cas de l'appui d'une touche (réception d'un *makecode*) est traité. Si cette touche est *Shift*, alors on change la table de conversion courante pour la table de conversion avec *Shift* : `kbd_current_translate_table` est une variable globale qui désigne la table de conversion courante. Si la touche n'est pas *Shift*, alors on utilise la table de conversion courante pour déterminer la séquence de caractères correspondant à la touche pressée. Lignes 11 à 18, on traite les *breakcodes*. En réalité, seul le relâchement de la touche *Shift* est pris en compte afin de positionner à nouveau la table de conversion courante vers la table de conversion normale.

Une fois la séquence de caractères déterminée, il suffit de les ajouter dans le buffer du terminal :

```
if (key)
  tty_add_chars (tty, key);
```

La figure 7 présente un exemple de séquence de touches, ainsi que le traitement associé dans le pilote clavier de SOS.

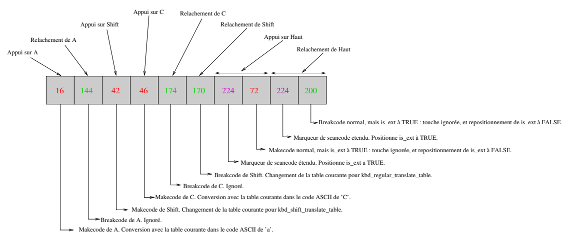

FIG. 7 – Exemple de séquence de touches, avec le traitement associé dans le pilote clavier de SOS. Les codes en rouge sont des *scancodes*, ceux en vert sont des *breakcodes*, et en violet sont indiqués les marqueurs de *scancode étendu*.

Ce code suffit donc pour gérer les lettres, chiffres, majuscules et minuscules, ainsi que les touches de fonction et les flèches de direction. Néanmoins, un pilote clavier complet devra gérer les touches *Control*, *Alt*, *AltGr*, et permettre de changer de *keymap*. Avis aux amateurs !

4.2 Pilote console

Un pilote console minimaliste est disponible dans SOS depuis le premier article dans le fichier drivers/x86_videomem.c. Ce pilote permet d'afficher des caractères ou des chaînes à une position donnée de l'écran. Il ne gère pas lui-même le défilement, l'affichage caractère après caractère en avançant le curseur, etc. Ce pilote a donc été amélioré grâce à une couche ajoutée au dessus du pilote existant.

Désormais, l'écran est séparé en deux parties : les CONSOLE_ROW_START premières lignes sont réservées pour l'affichage d'informations par le noyau en utilisant les fonctions du pilote de bas-niveau existant. Le reste de l'écran est utilisé pour la console accessible aux applications utilisateur et supporte le défilement (*scrolling*).

La fonction sos_screen_init() initialise cette console de haut niveau en réalisant les opérations suivantes :

– Initialisation de row et col, des variables globales qui seront utilisées pour savoir à quelle position le prochain caractère doit être affiché ;
– Configuration du curseur, par l'intermédiaire de commandes *VGA* (voir [3] pour les détails) ;
– Initialisation de la zone de l'écran réservée à cette console, afin qu'elle soit sur fond gris clair ;
– Affichage du curseur à la position courante.

La fonction sos_screen_putchar(), qui affiche un caractère passé en paramètre à la position courante de la console, constitue le cœur de ce pilote de haut niveau. Elle fonctionne en trois étapes :

– Traitement du caractère proprement dit. Les cas du *retour chariot* (\n) et de l'*effacement d'un caractère* (\b) disposent d'un traitement spécifique ;
– Défilement d'une ligne vers le haut quand on atteint le bas de l'écran ;
– Mise à jour de la position du curseur en utilisant la fonction sos_screen_set_cursor().

La gestion de cette console a donc été simplifiée à l'extrême. En particulier, les séquences d'échappement ANSI ne sont pas interprétées, le multi-console n'est pas prévu, ni la possibilité de remonter dans l'historique de la console. Voilà encore quelques exercices intéressants pour les lecteurs motivés !

4.3 Création du terminal

Le terminal clavier/écran est créé dans drivers/console.c, qui contient une unique et très simple fonction sos_console_setup(). Cette fonction créé ce terminal clavier/écran **1)** en initialisant la console par un appel à sos_screen_init(), **2)** en créant le *tty* par un appel à tty_create(), puis **3)** en associant le clavier à ce terminal par un appel à sos_kbd_init().

```
int sos_console_setup (void)
{
  sos_ret_t ret;
  struct tty_device *tty;

  ret = sos_screen_init();
  if (SOS_OK != ret)
    return ret;

  ret = tty_create (SOS_CHARDEV_CONSOLE_MINOR,
                    sos_screen_putchar, & tty);
  if (SOS_OK != ret)
    return ret;

  return sos_kbd_init (tty);
}
```

Cette fonction sos_console_init() est simplement appelée par la fonction principale d'initialisation de SOS dans le fichier sos/main.c.

5 Terminal série

Nous fournissons un pilote de ligne série minimal dans drivers/serial.c.

Sur PC, les lignes série sont pilotées par une puce appelée *UART* pour *Universal Asynchronous Receiver/Transmitter*, chargée de convertir des octets (8 bits en parallèle) en des bits transmis en série et vice-versa. L'*UART* présent dans les PCs actuels répond généralement au doux nom de *16550A*, référence que vous avez peut-être aperçue lors du démarrage de votre noyau Linux. Pour plus d'informations sur le fonctionnement de cet *UART* et sa programmation, vous pouvez consulter [4] ou le chapitre 32.2 de [5].

Le niveau d'interruption et les ports d'entrée/sortie associés au contrôleur série sont standards dans le monde PC, du moins pour les 2 premiers contrôleurs. Il faudrait modifier le code pour, par exemple, pouvoir piloter des cartes contrôleur série supplémentaires branchées sur le bus PCI.

Comme pour le pilote x86_videomem, le pilote série est constitué de deux niveaux : un bas niveau, utilisable directement pour le déboguage, et un niveau plus haut s'interfaçant avec l'infrastructure *chardev* pour faire fonctionner un port série comme un périphérique caractère.

5.1 Pilote série bas-niveau

La partie bas-niveau du pilote série est théoriquement capable de gérer 4 ports série, dont les adresses d'entrées-sorties sont stockées dans le tableau _serial_config[]. Ces adresses d'entrées-sorties sont les adresses de base d'une plage de registres permettant de commander chaque port série. Ces registres, au nombre de 8, sont définis en tête du fichier drivers/serial.c : UART_RX, UART_TX, UART_DLL, UART_IER, etc.

5.1.1 Initialisation

L'initialisation de la partie bas-niveau est effectuée dans la fonction sos_serial_subsystem_setup() qui peut être appelée très tôt dans l'initialisation du système. En effet, elle ne nécessite pas d'allocation mémoire et ne dépend d'aucune infrastructure qui aurait besoin d'être initialisée auparavant. Ce pilote bas-niveau ne dépend pas non plus de l'initialisation des interruptions : il fonctionne uniquement en mode *polling*, c'est-à-dire en effectuant des attentes actives. Il pourra donc être utilisé dès le tout début du fonctionnement de SOS. Cela peut être utile pour déboguer le système sur machine réelle quand elle plante très tôt.

Cette fonction sos_serial_subsystem_setup() initialise seulement le premier port série et notre pilote se limitera d'ailleurs à ce premier port série. Pour l'initialiser, il réalise les opérations suivantes :

1. Désactivation des interruptions : serial_outb (serial_port + UART_IER, 0);
2. Positionne le *DLAB* ou *Diviser Latch Access Byte* à 1 dans le registre UART_LCR. Positionner ce *DLAB* permet de changer la signification des deux premiers registres. Au lieu que le registre 0 soit le buffer d'émission/réception (UART_RX et UART_TX), il sera le *Diviser Latch Low Byte* (UART_DLL). Et au lieu que le registre 1 soit le registre de configuration des interruptions (UART_IER), il devient le *Diviser Latch High Byte* (UART_DLH). Il est nécessaire

de positionner ce *DLAB* pendant l'initialisation pour pouvoir configurer la vitesse du port série ;

3. Configure la vitesse du port série à *115.200 bps* :

```
#define SERIAL_BAUDRATE_MAX 115200

unsigned int speed = 115200;

div = SERIAL_BAUDRATE_MAX / speed;

serial_outb (serial_port + UART_DLL, div & 0xFF);
serial_outb (serial_port + UART_DLH, div >> 8);
```

Cette vitesse de *115.200 bps* est tout d'abord convertie en un diviseur pour l'horloge du port série. Cette horloge oscille à 1.8432 Mhz et comporte un compteur interne qui ramène la fréquence à 115.200 Hz. Ainsi, un diviseur (`div`) de 1 configurera une vitesse de 115.200 bps, tandis qu'un diviseur de 24 correspondra à une vitesse de 4.800 bps. La valeur de ce diviseur pouvant excéder 8 bits, elle est découpée dans deux registres vus plus haut, le *Diviser Latch High Byte* pour les 8 bits de poids fort, et le *Diviser Latch Low Byte* pour les 8 bits de poids faible.

4. Configure la ligne série sans bit de parité, avec des mots de 8 bits et un bit de stop. Dans le même temps, cette opération va remettre à 0 le bit *DLAB* puisqu'on écrit une nouvelle valeur dans le registre `UART_LCR` :

```
unsigned char status = 0;
int word_len = UART_8BITS_WORD;
int parity = UART_NO_PARITY;
int stop_bit_len = UART_1_STOP_BIT;

status |= parity | word_len | stop_bit_len;
serial_outb (serial_port + UART_LCR, status);
```

5. Active le buffer *FIFO* du contrôleur série avec une taille de 14 octets : `serial_outb (serial_port + UART_FCR, UART_ENABLE_FIFO);`
6. Vide le buffer d'entrée du contrôleur série.

5.1.2 Écriture

L'écriture est implémentée dans la fonction `serial_putchar()` :

```
static int
serial_putchar (unsigned short port, char c)
{
  /* Perhaps a timeout is necessary.  */
  int timeout = 10000;

  /* Wait until the transmitter holding register is empty.  */
  while ((serial_inb (port + UART_LSR) & UART_EMPTY_TRANSMITTER) == 0)
    if (--timeout == 0)
      /* There is something wrong. But what can I do?  */
      return -1;

  serial_outb (port + UART_TX, c);
  return 0;
}
```

Elle commence par attendre (attente active ou *polling*) que le registre de transmission `UART_TX` soit disponible, et une fois que c'est le cas, envoie le caractère `c` sur ce registre.

Les fonctions `serial_prints()` et `sos_serial_printf()` utilisent `serial_putchar()` pour envoyer une chaîne de caractères sur la ligne série.

5.1.3 Lecture

La lecture est implémentée dans la fonction `serial_getchar()`.

```
char
serial_getchar (unsigned short unit)
{
  /* Wait until data is ready.  */
  while ((serial_inb (_serial_config[unit].iobase + UART_LSR)
          & UART_DATA_READY) == 0)
    ;

  /* Read and return the data.  */
  return serial_inb (_serial_config[unit].iobase + UART_RX);
}
```

Cette fonction attend de manière active (*polling*) qu'un caractère à lire soit disponible en surveillant le bit `UART_DATA_READY` du registre `UART_LSR`. Une fois que le caractère est disponible, elle va le lire dans le registre `UART_RX` et le retourne.

À la différence du pilote de clavier, il n'y a pas besoin de faire de traduction particulière type scancode/code ASCII : le caractère reçu est d'emblée au format ASCII.

5.2 Pilote série haut-niveau

5.2.1 Initialisation

La fonction `sos_ttySX_subsystem_setup()`, qui initialise le pilote série de haut-niveau, doit être appelée après l'initialisation du pilote de bas-niveau. Cette fonction nécessite que le système soit pleinement fonctionnel : elle alloue de la mémoire, a besoin de la gestion des interruptions, etc.

Elle commence par créer un nouveau terminal en utilisant `tty_create()`. La fonction d'écriture spécifiée pour ce terminal est `sos_serial_putchar()`. Le pointeur sur la structure `struct tty_device` est stocké dans un champ de la structure `_serial_config`, de manière à en disposer lors de l'appel à `tty_add_chars()` lors de la réception de caractères. Une fois ce terminal créé, la fonction `sos_serial_int_handler()` est enregistrée comme gestionnaire de l'IRQ `SOS_IRQ_COM1`. Ce pilote de haut-niveau ne fonctionne donc plus en mode *polling*, mais utilise les interruptions. Ce fonctionnement est en effet requis par l'architecture du sous-système *tty*. Enfin, cette fonction d'initialisation active le déclenchement d'une interruption lors de la réception d'un caractère sur le port série en écrivant `1` dans le registre `UART_IER` du premier port série.

5.2.2 Écriture

La fonction `sos_serial_putchar()` est la fonction d'affichage d'un caractère utilisée par le terminal. Elle utilise simplement `serial_putchar()` pour envoyer le caractère sur le port série 0. Cette fonction gère spécifiquement le cas du caractère `\b`. Au lieu de l'envoyer directement sur la ligne série, on envoie à la place une séquence de commandes ANSI pour indiquer au terminal se trouvant de l'autre coté de la ligne série de bien vouloir effacer le dernier caractère envoyé.

5.2.3 Lecture

La fonction `sos_serial_int_handler()` est appelée lorsqu'un caractère est reçu sur la ligne série, puisqu'il s'agit du gestionnaire d'interruption associée à l'*IRQ* du contrôleur série. Elle commence par lire le caractère reçu *via* le registre `UART_RX` puis elle construit une chaîne de caractères terminée par un caractère nul et l'envoie au terminal *via* `tty_add_chars()`. Cette chaîne de deux caractères est simplement constituée du caractère reçu par le port série et du caractère nul.

6 Démonstration

Ce mois-ci, la petite démonstration que nous vous proposons est un *shell* rudimentaire. Implémenté sous la forme d'une application utilisateur, il permet d'exécuter quelques commandes de base. Dans la démonstration, deux instances de ce *shell* fonctionneront : l'une au travers de la console clavier/écran, l'autre au travers de la ligne série. Pour accéder à la ligne série dans *Qemu*, il faut l'exécuter avec l'option `-serial vc`. La console série sera alors accessible en tapant `Ctrl-Alt-3`, et on utilisera `Ctrl-Alt-1` pour revenir à la console normale.

Les deux *shells* sont lancés par le processus `init` (voir le fichier `userland/init.c`). Avant d'exécuter ces *shells*, le processus `init` créé un certain nombre de fichiers spéciaux dans le répertoire `/dev`. En effet, dans un système d'exploitation complet, ces fichiers spéciaux sont stockés dans un système de fichiers sur disque dur. Dans SOS, nous n'avons pas encore de *vrais* systèmes de fichiers "stockables" sur disque, il est donc nécessaire de re-créer ces fichiers spéciaux dans le système de fichiers virtuel `virtfs` (voir l'article 8) à chaque lancement du système. Des fichiers spéciaux pour `/dev/mem`, `/dev/kmem`, `/dev/zero`, `/dev/null`, `/dev/tty`, `/dev/ttys` et autres sont donc créés à ce moment là en utilisant l'appel système `mknod()`. Les majeurs et mineurs sont partagés entre le noyau et les processus utilisateur au travers du fichier `drivers/devices.h`.

```
mknod("/dev/tty", S_IRUSR | S_IWUSR,
      S_IFCHR, SOS_CHARDEV_TTY_MAJOR, SOS_CHARDEV_CONSOLE_MINOR);
mknod("/dev/ttyS", S_IRUSR | S_IWUSR,
      S_IFCHR, SOS_CHARDEV_TTY_MAJOR, SOS_CHARDEV_SERIAL_MINOR);
```

Une fois ces fichiers spéciaux créés, le processus `init` ouvre l'entrée standard (descripteur 0), la sortie standard (descripteur 1) et la sortie d'erreur (descripteur 2) puis lance deux *shells* :

```
/* Set up the shell on the console */
TEST_EXPECT_CONDITION(open("/dev/tty", O_RDWR), RETVAL == 0);
TEST_EXPECT_CONDITION(open("/dev/tty", O_RDWR), RETVAL == 1);
TEST_EXPECT_CONDITION(open("/dev/tty", O_RDWR), RETVAL == 2);

if (fork() == 0)
  exec ("shell");

close (2);
close (1);
close (0);

/* Set up the shell on the serial port */
TEST_EXPECT_CONDITION(open("/dev/ttyS", O_RDWR), RETVAL == 0);
TEST_EXPECT_CONDITION(open("/dev/ttyS", O_RDWR), RETVAL == 1);
TEST_EXPECT_CONDITION(open("/dev/ttyS", O_RDWR), RETVAL == 2);

if (fork() == 0)
  exec ("shell");

close (2);
close (1);
close (0);
```

Le *shell* proprement dit est implémenté dans le fichier `userland/shell.c`. Son fonctionnement est très simple : la fonction `main()` affiche une invite, lit l'entrée standard caractère par caractère et les stocke dans un buffer. Dès qu'un caractère *retour à la ligne* est lu, alors le buffer est passé à la fonction `command_exec()` qui se charge d'analyser la commande et d'exécuter la fonction l'implémentant. Les commandes disponibles sont `uname`, `ls`, `touch`, `mkdir`, `cat`, `hexdump` et `edit`. Contrairement au fonctionnement classique d'un Unix, ces commandes sont toutes des commandes *internes* au *shell* : elles sont exécutées directement dans le processus du *shell*. Dans un prochain article, SOS disposera d'un système de fichiers dans lequel nous pourrons stocker des programmes exécutables et le *shell* pourra alors utiliser des commandes externes.

Dans l'implémentation de ces diverses commandes, notons un point intéressant. Lorsque la fonction `main()` démarre, elle effectue un appel `ioctl()` sur l'entrée standard pour la passer en mode *canonique* (voir section 3.1) afin de pouvoir lire caractère par caractère :

```
ioctl (0, SOS_IOCTL_TTY_RESETPARAM, SOS_IOCTLPARAM_TTY_CANON);
```

Inversement, la commande `edit`, qui permet de manière rudimentaire de créer un fichier texte, désactive le mode canonique puis active l'écho à son lancement, et réactive le mode canonique puis désactive l'écho lorsqu'elle se termine :

```
/* Activate echo and activate again canonical mode */
ioctl (0, SOS_IOCTL_TTY_SETPARAM, SOS_IOCTLPARAM_TTY_ECHO);
ioctl (0, SOS_IOCTL_TTY_SETPARAM, SOS_IOCTLPARAM_TTY_CANON);

[...]

/* Desactivate echo and remove canonical mode */
ioctl (0, SOS_IOCTL_TTY_RESETPARAM, SOS_IOCTLPARAM_TTY_ECHO);
ioctl (0, SOS_IOCTL_TTY_RESETPARAM, SOS_IOCTLPARAM_TTY_CANON);
```

Avec ce petit *shell*, SOS commence à ressembler à un système d'exploitation utilisable !

Conclusion

Dans cet article, nous avons présenté comment les pilotes de périphériques *caractère* pouvaient être intégrés dans le système. Nous avons détaillé comment ils devaient interagir avec le *VFS* pour être utilisés depuis les applications utilisateur.

Nous avons ensuite détaillé quelques exemples de pilotes de périphériques. Des pilotes de périphériques caractère virtuels tout d'abord : `/dev/zero` et `/dev/null`. Puis des périphériques liés à la mémoire physique (`/dev/mem`) et virtuelle du noyau (`/dev/kmem`).

Nous avons ensuite présenté une infrastructure générique, le sous-système `tty`, pour la gestion des terminaux. Nous avons montré comment utiliser cette infrastructure pour contrôler la console (i.e. le clavier/écran) et la ligne série.

Enfin, nous avons terminé en présentant le programme de démonstration du mois : un *shell* très simple capable de vous obéir au doigt et à l'œil... enfin seulement au doigt pour le moment.

Le prochain article de la série SOS permettra d'étudier les pilotes de périphériques de type *bloc* par l'implémentation du sous-système *blockdev* ainsi que celles d'un pilote pour disques durs *IDE* et de l'infrastructure de gestion des partitions du disque.

En attendant, à vos claviers, on peut commencer à faire des choses intéressantes (?) *dans* SOS maintenant !

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 Henri

Références

[1] *VT100.Net*.
http://www.vt100.net.

[2] Phil Gregory. Terminal function key escape codes.
http://aperiodic.net/phil/archives/2005/07/.

[3] Ralf Brown. Ralf brown's interrupt (and port) list.
http://www.cs.cmu.edu/~ralf/files.html, 2000.

[4] Craig Peacock. Interfacing the serial / rs232 port.
http://www.beyondlogic.org/serial/serial.htm.

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