Croisière au cœur d'un OS*
Étape 5 : Allocation de mémoire pour le noyau

Résumé

Grâce aux deux articles qui précèdent, nous savons allouer de la RAM et comment y associer des adresses virtuelles quelconques. Nous présentons ici un *allocateur* de mémoire virtuelle qui permet d'avoir accès à des primitives de type `malloc()/free()` (pour l'allocation et la libération) dans le noyau.

Introduction

Les articles 3 et 4 ont posé les bases de la gestion de la mémoire dans SOS en implantant la gestion de la mémoire physique et la pagination. SOS est maintenant capable d'allouer de la mémoire physique par pages de 4ko et de les *mapper* en mémoire virtuelle.

Néanmoins, d'une part le noyau a besoin d'allouer des objets (au sens de "blocs de données", pas au sens de la programmation orientée objet) de tailles diverses, en général différentes de 4ko et souvent plus petites : informations relatives à un système de fichiers, à un processus, à un thread, à une socket, etc. D'autre part il faut garantir que ces objets ne se recouvrent pas en mémoire, sinon il y a un bogue latent (écrasement de données). Afin de gérer les adresses de ces objets de façon cohérente, le noyau a besoin d'un allocateur de mémoire qui va leur attribuer un espace en mémoire virtuelle le temps de leur utilisation puis libérer cet espace. En pratique un tel allocateur prend souvent la forme de primitives du type `malloc()/free()`.

Cet article porte donc sur l'implantation d'un allocateur de mémoire virtuelle pour le noyau. Il s'agit du premier article qui s'intéresse à la gestion de ressources qui ne sont pas physiques (les adresses virtuelles). Il repose intégralement sur les deux articles précédents : nous aurons besoin d'allouer des pages de mémoire physique (article 3) et nous aurons besoin de les associer à des adresses virtuelles (article 4) de sorte que le code de l'article est indépendant de l'architecture `x86`.

C'est aussi le premier article qui aura une composante algorithmique très importante et non triviale. On pourra d'ailleurs reprocher la complexité du choix que nous avons fait, remettant probablement en cause le "Simple" du nom "SOS". En effet, l'allocateur de type *Slab*[1] que nous présentons n'est certainement pas le plus simple, mais il est assez répandu (Solaris 2.4 et suivants, Linux 2.4 et suivants, ...) et permet d'aborder quelques difficultés de réalisation.

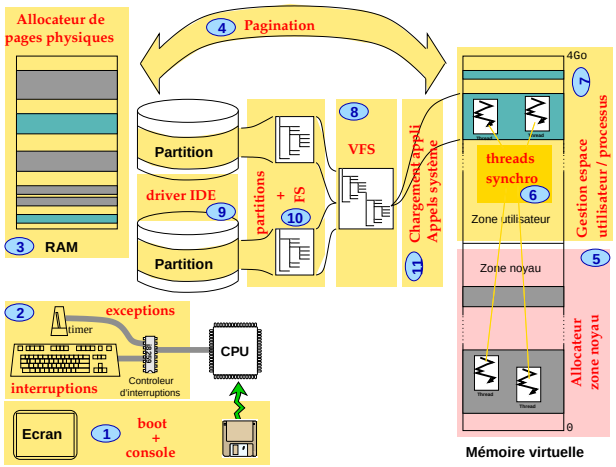


FIG. 1 – Programme des articles

Dans ce qui suit, nous commencerons par un rapide rappel du rôle de l'allocation de mémoire (section 1). Nous donnerons ensuite un aperçu de l'allocateur que nous avons implanté dans SOS (section 2) et qui est constitué de deux sous-systèmes. Nous présenterons ces deux sous-systèmes dans les sections 3 et 4 et nous expliquerons comment ils sont combinés dans l'allocateur traditionnel du type `malloc()/free()` (section 5). Nous décrirons ensuite comment les deux sous-systèmes interdépendants précédents sont initialisés "en parallèle" (section 6). Nous ferons une synthèse pour rappeler les points importants de cette présentation en section 7, avant de terminer par l'habituelle petite démo (section 8).

Contrairement à l'habitude et pour des raisons de place, nous illustrerons peu par des extraits de code. Pour suivre les explications, vous devrez donc avoir le code de SOS sous les yeux (essentiellement les deux fichiers `sos/kmem_vmm.c` et `sos/kmem_slab.c`).

1 Généralités sur l'allocation de mémoire

Le principe de fonctionnement élémentaire d'un allocateur de mémoire est toujours le même, et ceci malgré

*La version originale de cet article a été publiée dans GNU Linux Magazine France numéro 66 – Novembre 2004 (`http://www.linuxmag-france.org`) et cette version est diffusée avec l'autorisation de l'éditeur.

[1] Insistons sur "***de type***"...

les dizaines d'années de littérature sur le sujet.

Lorsqu'on lui soumet une requête d'*allocation* d'un objet de taille donnée (typiquement : appel à `malloc(taille)` sous Unix), il recherche un emplacement libre de taille suffisante dans l'espace des adresses et le marque comme occupé. Tant que l'objet n'a pas été libéré, les allocations qui suivront ne pourront pas utiliser cet emplacement, même partiellement, sinon il y aurait écrasement de données puisque 2 objets se recouvreraient.

Quand l'objet n'est plus utilisé, on peut libérer la portion de l'espace des adresses qui lui était allouée pour permettre l'allocation d'autres objets à cet emplacement. Dans certains allocateurs, cette étape de libération est explicite (par exemple : l'appel à `free(adresse)` sous Unix est nécessaire) ; dans d'autres cas elle est implicite (la plupart des langages de scripts, les langages Caml, Java, C#, etc.) et on parle de *glanage de cellules* ou de *ramassage de miettes* (*garbage collection*).

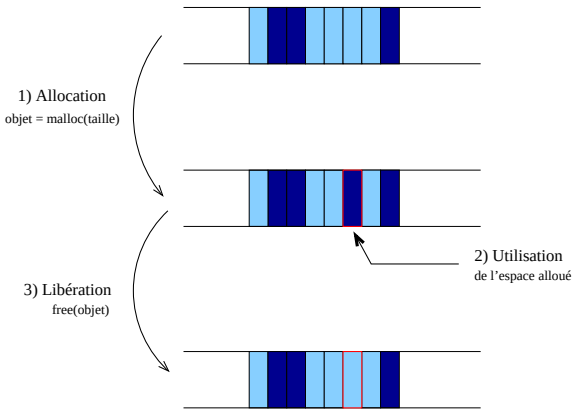


FIG. 2 – Allocation, utilisation et libération

Autrement dit, un allocateur de mémoire est chargé de gérer la ressource "espace des adresses en mémoire". Il s'agit d'un gestionnaire de ressource logique *par réservation* qui établit des règles de bonne conduite : on doit allouer un objet avant d'utiliser une de ses données ; on ne doit pas libérer un objet avant d'avoir terminé de l'utiliser ; on ne doit pas libérer un objet à une adresse ne correspondant à aucun objet encore alloué ; etc. Rien n'empêche de passer outre ces règles de bonne conduite, si ce n'est le bogue qui se manifestera tôt ou tard à cause d'un écrasement de données...

2 Présentation de l'allocateur noyau de SOS

Dans le cas de SOS, l'allocateur du noyau qui nous intéresse ici s'occupe de gérer l'*espace des adresses virtuelles du noyau* (i.e. les adresses virtuelles 0-1Go, voir l'article 4) et ne dispose pas d'un *garbage collector*. La gestion des adresses virtuelles dans la zone utilisateur (1Go-4Go) sera détaillée dans un article ultérieur.

2.1 Structure

Comme nous l'avons vu dans l'article 4, la mémoire virtuelle est découpée en *pages* de taille fixée (4 Ko par défaut sur l'architecture x86). Cependant le noyau ne manipule pas *que* des objets de 4ko. Pour l'allocation d'objets de plus grande taille, l'allocateur doit permettre l'allocation de *régions* de mémoire virtuelle, c'est-à-dire d'ensembles de pages contiguës en mémoire virtuelle (pas nécessairement contiguës en mémoire physique). Et pour l'allocation d'objets de plus petite taille, l'allocateur doit être capable de découper les pages en morceaux et de gérer ces morceaux de pages.

C'est pourquoi nous avons choisi de découper l'allocateur de SOS en deux sous-systèmes liés l'un à l'autre (la figure 3 donne une vue d'ensemble) :

kmem_vmm **:** l'allocateur de *régions* de mémoire virtuelle pour le noyau. Cet allocateur permet d'allouer des régions de mémoire virtuelle d'une ou plusieurs pages contiguës (les *ranges*, ou *intervalles* en français), en gèrant l'espace des adresses virtuelles du noyau à la granularité de la page (4ko).

kmem_slab **:** l'allocateur de mémoire, aussi appelé *cache*, qui va permettre l'allocation d'objets de taille quelconque. En fait, on ne devrait pas parler d'un allocateur mais de plusieurs allocateurs (ou plusieurs caches). En effet, chaque *cache* est spécialisé dans l'allocation d'objets d'une taille donnée, définie par le programmeur lors de la création du *cache*. En pratique le programmeur créera un cache non pas pour une taille d'objet donnée, mais plutôt pour un type d'objet donné (un cache pour les *inodes* servant aux systèmes de fichiers, un cache pour les buffers réseau, etc.). Afin de mieux comprendre la suite de l'article, notons que chaque *cache* utilise *kmem_vmm* pour allouer des régions virtuelles : les *slabs* (*galettes* en français). Ces *slabs* contiennent des espaces prêts à être alloués, dont la taille correspond à celle des objets définie lors de la création du cache. Plusieurs objets peuvent ainsi être contenus dans un slab. Le rôle d'un *cache* est ainsi de maintenir une liste des espaces libres/occupés dans ces slabs et d'allouer de nouveaux slabs (ou de libérer des anciens) en fonction des besoins.

Pour résumer, le principe fondamental est très simple. la première étape consiste à allouer en mémoire des gros objets (slabs). Ensuite on considère ces gros objets comme des tableaux de petits objets prêts à être alloués. Puis enfin on pioche dans ces tableaux pour allouer les objets.

On peut se demander où sont passés `malloc()` et `free()` dans tout ça... C'est tout simple : il suffira de définir un *cache* pour les objets de taille 8 octets, un autre *cache* pour ceux de 12 octets (par exemple), encore un autre pour ceux de 16 octets, et ainsi de suite. Et le tour est joué : quand on veut allouer un bloc de 8 octets, on pioche dans les *slabs* du *cache* contenant des

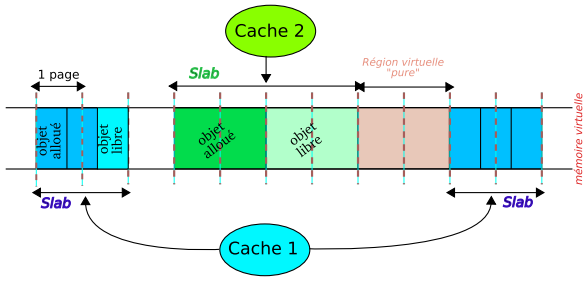


FIG. 3 – Exemple avec 4 régions virtuelles utilisées : 2 régions pour les 2 slabs d'un même cache (en bleu : slabs de 2 pages pour 3 objets), 1 région pour le slab d'un autre cache (en vert : slab de 4 pages pour 2 objets) et 1 région utilisée par autre chose que *kmem_slab* (en rouge). La région blanche (1 page) entre un slab bleu et le slab vert est marquée encore libre.

objets de 8 octets, etc.

2.2 Caractéristiques

L'allocateur (i.e. les deux sous-systèmes précédents) est inspiré du *Slab Allocator* [1, 2, 3] de Solaris (2.4 et suivants) et de Linux (2.4 et suivants).

Il est dit *ségrégationniste* puisqu'il réserve des portions de l'espace virtuel (ici : les *slabs*) pour l'allocation d'objets de taille donnée. Il existe aussi des allocateurs non ségrégationnistes (*first/best/worst fit*, etc.), c'est-à-dire qui allouent les objets à la volée exclusivement en fonction de l'espace disponible : ils diffèrent les uns des autres par l'algorithme de recherche des emplacements disponibles. D'autres allocateurs sont "entre les deux" (*buddy systems, Fibonacci,* etc.) c'est-à-dire que les emplacements libres dans lesquels on pourra allouer un objet ne sont pas contraints uniquement par la taille de ces emplacements, mais aussi par la taille des objets qui ont été alloués au voisinage de cet emplacement : l'historique des allocations/libération amène ainsi à préréserver certaines portions de l'espace mémoire pour l'allocation d'objets de taille donnée.

Nous ne nous lancerons pas dans une étude comparative de tous ces algorithmes car c'est au-dessus de nos connaissances (40 ans de littérature). Nous dirons seulement qu'on les compare en général suivant deux critères : leur rapidité d'allocation/libération, et la fragmentation de la mémoire qu'ils occasionnent.

Relativement à ces deux critères, l'allocateur que nous avons choisi est intéressant car il est très bien adapté au profil des allocations/libérations effectuées par un OS. En effet, dans un OS on aura peu de types d'objets différents et beaucoup d'objets pour un type donné. Par exemple, sous Linux (voir `cat /proc/slabinfo`), on a beaucoup d'objets de type `inode` (structures de gestion des systèmes de fichiers), beaucoup d'objets de type `dentry` (structures de gestion de l'espace de nommage), beaucoup d'objets de type `skbuff` (buffers réseau), etc. On définit donc naturellement une série de "caches" pour chacun de ces types d'objets.

2.3 Précautions de réalisation

Les deux sous-systèmes `kmem_vmm` et `kmem_slab` ont des rôles différents et complémentaires. Ils sont néanmoins interdépendants : `kmem_vmm` a besoin de `kmem_slab` pour allouer des structures de données nécessaires à son fonctionnement (`struct sos_kmem_range`) et `kmem_slab` a besoin de `kmem_vmm` comme nous l'avons indiqué dans la section 2.1. Une partie de la complexité de la conception de ces sous-systèmes est liée à cette interdépendance conduisant à deux double-récursivités : lors de l'allocation et lors de la libération (cf. sections 3.3.2 et 4.4 pour plus d'explication sur ces récursivités). Malheureusement la récursivité peut se traduire par une utilisation importante de la pile (nous en reparlerons dans le prochain article). Ceci n'est pas envisageable dans un OS tel que SOS puisque la taille de la pile noyau est réduite et non extensible. Il a donc fallu veiller à briser ces deux double-récursivités, sans quoi un débordement de pile eût été inévitable.

Dans ce qui suit, nous allons décrire les deux sous-systèmes `kmem_vmm` (section 3) puis `kmem_slab` (section 4). Pour des raisons de simplicité dans la description, lorsque nous présenterons `kmem_vmm` nous supposerons que `kmem_slab` fonctionne, et quand nous présenterons `kmem_slab` nous supposerons que `kmem_vmm` fonctionne. La mise en place ces deux sous-systèmes en "parallèle" (le problème de l'œuf et de la poule, lié à la double récursivité) sera vue en section 6.

3 Gestionnaire de régions de mémoire virtuelle (*kmem_vmm*)

L'allocateur `kmem_vmm` correspond au code de `sos/kmem_vmm.c` et au fichier d'en-tête `sos/kmem_vmm.h`. Il s'occupe de gérer les pages de mémoire virtuelle du noyau, i.e. entre 0 et 1G. Il s'agit d'un allocateur de type *first fit* : pour chaque requête d'allocation de n pages contiguës il utilise le **premier** espace disponible de taille suffisante, i.e. qui contient au moins n pages contiguës libres.

Cet allocateur doit donc maintenir une liste des régions libres et une liste des régions occupées (voir la figure 4), et devra proposer des fonctions d'allocation et de libération de régions qui manipuleront ces listes.

3.1 Définitions

Chaque région de mémoire est décrite par une structure de type `sos/kmem_vmm.c:struct sos_kmem_range` qui est opaque à l'utilisateur :

```
struct sos_kmem_range
{
  sos_vaddr_t base_vaddr;
  sos_count_t nb_pages;
```

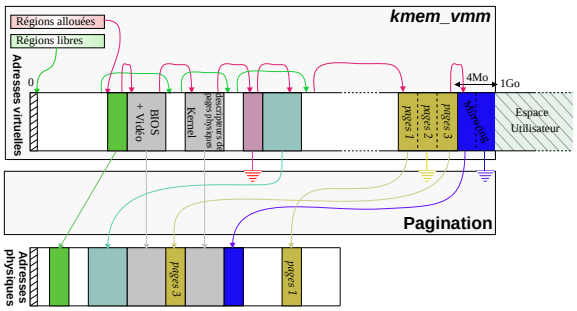


FIG. 4 – Gestion des adresses virtuelles de l'espace noyau par *kmem_vmm* et lien avec la pagination.

```
  /* The slab owning this range, or NULL */
  struct sos_kslab *slab;

  struct sos_kmem_range *prev, *next;
};
```

`base_vaddr` désigne l'adresse virtuelle de début de la région de mémoire, `nb_pages` le nombre de pages de la région et `slab` désigne le *slab* utilisant cette région de mémoire (voir la section suivante). Enfin, des pointeurs `next` et `prev` permettent de chaîner les régions dans les listes.

Tout le fonctionnement de `kmem_vmm` consiste en la gestion des deux listes de `sos_kmem_range`, à savoir une liste pour les régions occupées et une liste pour les régions libres :

```
static struct sos_kmem_range *kmem_free_range_list,
*kmem_used_range_list;
```

Dans ces listes, les régions sont classées par ordre d'adresses virtuelles de début croissantes et le rôle principal de `kmem_vmm` sera de garantir que ces régions ne se recouvrent jamais.

Ces deux listes couvrent l'intégralité de l'espace virtuel du noyau (i.e. les adresses 0-1Go). Au tout début, elles reflètent directement les régions déjà utilisées par les articles précédents (voir la figure 4) ainsi que nous le verrons dans la section 6.

3.2 Allocation d'une région de mémoire

L'allocation de `nb_pages` pages contiguës en mémoire virtuelle s'effectue dans la fonction `sos_kmem_vmm_alloc(nb_pages, flags)` qui appelle simplement la fonction `sos_kmem_vmm_new_range()`.

Cette fonction commence par chercher une région suffisamment grande (fonction `find_suitable_free_range()`) dans la liste des régions libres. Si la région trouvée est exactement de la taille voulue alors il suffit de la transférer vers la liste des régions utilisées en maintenant l'ordre des adresses de début des régions croissantes (fonction interne `insert_range()`). Si la région trouvée est plus grande, alors on la découpe en deux : une région de la taille voulue qu'on insère dans la liste des régions occupées et le reste qui demeure dans la liste des régions libres (voir la figure 5).

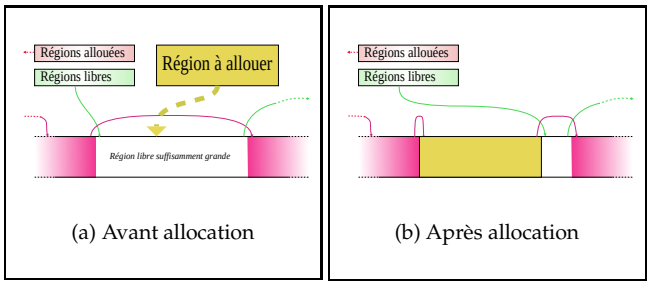


FIG. 5 – Configurations avant/après allocation d'une nouvelle région.

On remarquera que, dans le dernier cas, nous avons besoin d'allouer une structure de type `sos_kmem_range`. Nous n'avons que deux moyens pour ce faire : utiliser *kmem_vmm* ou utiliser *kmem_slab*. Puisque la structure `sos_kmem_range` est de petite taille, il serait ridicule d'allouer une région virtuelle (i.e. 1 ou plusieurs page-s) rien que pour elle, c'est pourquoi nous sommes amenés à utiliser tout naturellement l'allocateur `kmem_slab`. Mais cela peut poser un problème de récursivité car `kmem_slab` peut faire à son tour appel à `sos_kmem_vmm_alloc()` : nous y reviendrons dans la section 4.4.

Si le paramètre `flags` contient le drapeau `SOS_KMEM_VMM_MAP` alors une page physique est allouée puis mappée pour chaque page virtuelle de la région qui vient d'être allouée. Pour le moment le drapeau `SOS_KMEM_VMM_MAP` est **obligatoire** car SOS ne supporte pas le *demand paging* (allocation de pages physiques à la demande) à ce niveau : tout accès en dehors d'une page mappée conduit à une exception de type défaut de page, qui n'est pas encore gérée (article 6).

Ce *mapping* page physique/page virtuelle s'accompagne de la mise à jour du champ `kernel_range` dans le descripteur de page physique associé à la page virtuelle (nouvelle fonction `sos_physmem_set_kmem_range()` de physmem). Ce nouveau champ contient l'adresse de la structure `sos_kmem_range` où est mappée cette page, et a été rajouté spécialement pour accélérer la libération des régions (voir ci-dessous).

3.3 Libération d'une région de mémoire

3.3.1 Synopsis

La libération d'une région située à une adresse virtuelle `vaddr` s'effectue en appelant la fonction `sos_kmem_vmm_free(vaddr)`.

La première étape consiste à identifier la région de mémoire virtuelle (i.e. la structure `struct sos_kmem_range`) qui couvre `vaddr`. Il suffit pour cela de parcourir la liste des régions occupées (fonction interne `lookup_range()`). Dans le cas particulier où

une page physique est mappée à l'adresse virtuelle `vaddr`, nous avons implémenté une autre méthode plus rapide (voir la figure 6). Dans ce cas en effet, on récupère le descripteur de page physique associée qui contient le nouveau champ `kernel_range` (nouvelle fonction `sos_physmem_get_kmem_range()` de physmem) donnant directement l'adresse de la structure `struct sos_kmem_range` concernée. La recherche se fait alors en temps constant, contrairement au parcours de la liste des régions occupées, qui se fait en temps proportionnel au nombre de régions occupées.

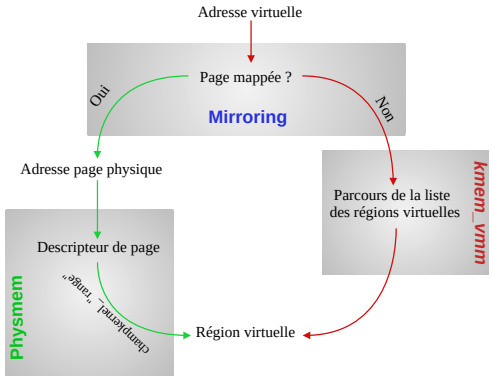


FIG. 6 – Récupération de la région virtuelle associée à une adresse virtuelle.

Ensuite, la libération proprement dite est effectuée par la fonction `sos_kmem_vmm_del_range(range)`. Cette fonction transfère la région à libérer vers la liste des régions libres (fonction `insert_range()`) en démappant les pages physiques associées s'il y en a. Si la nouvelle région libre précède ou suit immédiatement (relativement aux adresses virtuelles) d'autres régions libres, cette fonction veille à les fusionner.

3.3.2 Considérations de récursivité

La difficulté réside justement dans la libération des `struct sos_kmem_range` effectuée lorsque deux régions libres sont fusionnées. Si on les libère en faisant appel à la fonction de libération usuelle de `kmem_slab` (`sos_kmem_cache_free()`), alors celle-ci risque d'appeler `sos_kmem_vmm_del_range()` lorsque le slab de `struct sos_kmem_range` devient libre : il y a récursivité indirecte de `sos_kmem_vmm_del_range()`, phénomène que nous avions évoqué dans la section 2.3. Dans des cas pathologiques où il y a des dizaines ou des centaines de slabs de `sos_kmem_range` qui sont libérés en rafale (voir la figure 7), cela conduit à un débordement de la pile noyau (le prochain article détaillera pourquoi).

C'est pourquoi on *déroule* cette récursivité : pour libérer un `struct sos_kmem_range` on appelle une version allégée de `sos_kmem_cache_free()` (`sos_kmem_cache_release_struct_range()`) qui fait la même chose, sauf qu'elle n'appelle pas `sos_kmem_vmm_del_range()`. À la place, c'est directement `sos_kmem_vmm_del_range()` qui retient qu'il

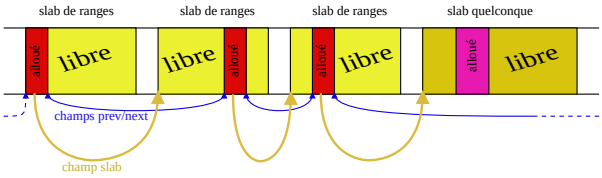


FIG. 7 – Dans cette configuration, la libération de l'objet en rose à droite va entraîner la libération de slabs en rafale.

faudra libérer une `struct sos_kmem_range`. Cela se fait en stockant cette structure dans une liste. Il suffit alors que `sos_kmem_vmm_del_range()` traite tous les éléments de cette liste au moyen d'une boucle. Elle sera mise en place par un `do ... while(range != NULL)` dans le code relatant les étapes présentées dans la section précédente.

4 L'allocateur de mémoire (*kmem_slab*)

L'allocateur d'objets `kmem_slab` repose sur deux notions fondamentales, à savoir les *caches* et les *slabs* :

Cache. Pour utiliser un tel allocateur, on commence par créer un *cache*. Chaque *cache* est destiné à l'allocation d'objets de taille fixe donnée au moment de la création du cache. Son rôle est de contenir une liste de *slabs*.

Slabs. Chaque slab est un ensemble de pages virtuelles contiguës et correspond à un tableau d'objets prêts à être alloués, ou déjà alloués.

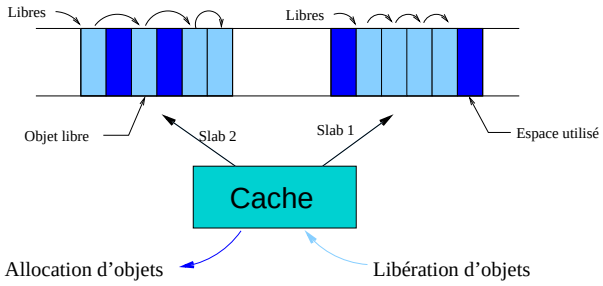


FIG. 8 – Un *cache* et ses *slabs*

Pour allouer un objet dans un cache (figure 9), on puise dans un des slabs du cache qui contient des objets disponibles si il en existe encore ; sinon on agrandit le cache en lui adjoignant un nouveau slab et on puise un objet dans ce nouveau slab.

Un exemple d'utilisation typique de `kmem_slab` est le suivant :

```
struct sos_kslab_cache *my_object_cache;
sos_vaddr_t object1, object2;

my_object_cache = sos_kmem_cache_create("my-object-cache",
                                        sizeof(struct my_object),
                                        1, 0, SOS_KSLAB_CREATE_MAP);


object1 = sos_kmem_cache_alloc(my_object_cache,
                               SOS_KSLAB_ALLOC_ATOMIC);
```

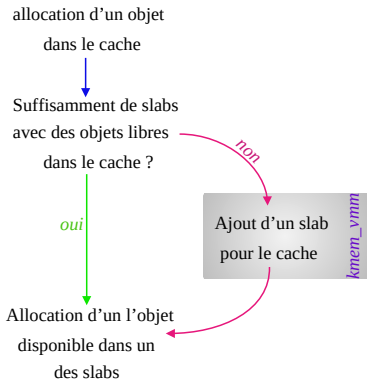


FIG. 9 – Allocation d'un objet dans un *cache*

```
object2 = sos_kmem_cache_alloc(my_object_cache,
                               SOS_KSLAB_ALLOC_ATOMIC);
sos_kmem_cache_free(object1);
sos_kmem_cache_free(object2);
[...]

sos_kmem_cache_destroy(my_object_cache);
```

Il convient donc d'implanter des fonctions permettent de créer et détruire des caches, d'ajouter et de supprimer des slabs dans des caches et d'allouer et de libérer des objets. Ces fonctions sont déclarées dans `sos/kmem_slab.h` et sont définies dans `sos/kmem_slab.c`.

4.1 Création et destruction de cache

La création et la destruction de *cache* sont des opérations accessibles à l'utilisateur de l'allocateur et implantées respectivement dans les fonctions `sos_kmem_cache_create()` et `sos_kmem_cache_destroy()`.

4.1.1 La structure `sos_kslab_cache`

À chaque cache est associée une structure de type `struct sos_kslab_cache`. Celle-ci stocke les informations relatives au *cache* et est opaque pour l'utilisateur de *kmem_slab*. Elle a la forme suivante :

```
struct sos_kslab_cache
{
  char *name;

  sos_size_t  original_obj_size;
  sos_size_t  alloc_obj_size;

  sos_count_t nb_objects_per_slab;
  sos_count_t nb_pages_per_slab;
  sos_count_t min_free_objects;

/* slab cache flags */
// #define SOS_KSLAB_CREATE_MAP  (1<<0) /* See kmem_slab.h */
// #define SOS_KSLAB_CREATE_ZERO (1<<1) /* " " " " " " " " */
#define ON_SLAB (1<<31) /* struct sos_kslab is included
                           inside the slab */
  sos_ui32_t  flags;

  /* Supervision data (updated at run-time) */
  sos_count_t nb_free_objects;

  /* The lists of slabs owned by this cache */
  struct sos_kslab *slab_list; /* head = non full, tail = full */

  /* The caches are linked together on the kslab_cache_list */
  struct sos_kslab_cache *prev, *next;
};
```

La signification de ses champs est la suivante :

name : le nom du cache ;

original_obj_size : la taille des objets spécifiée par l'utilisateur ;

alloc_obj_size : la taille qui est effectivement allouée pour chaque objet ($\geq$ `original_obj_size`) après prise en compte de contraintes d'alignement ;

nb_objects_per_slab : le nombre d'objets par *slab* ;

nb_pages_per_slab : le nombre de pages par *slab* ;

min_free_objects : le nombre minimum d'objets libres que doit contenir le cache à tout moment. C'est grâce à ce champ qu'on peut briser la double-récursivité à l'allocation (voir la section 4.4) ;

flags : divers drapeaux de configuration dont nous parlons plus loin ;

nb_free_objects : le nombre d'objets actuellement disponibles dans le cache ;

slab_list : la liste des *slabs* de ce *cache*. Les *slabs* dans lesquels des objets sont encore disponibles sont positionnés en **tête** de liste, tandis que les *slabs* entièrement alloués sont positionnés en **queue** de liste ;

prev, next : pointeurs de chaînage pour la liste globale des *caches* `kslab_cache_list`.

4.1.2 Création d'un cache

La fonction `sos_kmem_cache_create(name, obj_size, pages_per_slab, min_free_objs, cache_flags)` nécessite les paramètres suivants pour mettre en place le *cache* :

`name :` c'est le nom du *cache*, principalement pour deboguer et afficher des statistiques d'utilisation. La chaîne de caractères doit rester accessible durant toute la vie du cache car elle n'est pas recopiée dans un endroit sûr.

`obj_size :` la taille des objets que le cache doit gérer. Celle-ci sera arrondie aux 4 octets supérieurs si nécessaire. Si un alignement spécifique au delà de 4 octets est requis, alors l'utilisateur doit le prendre en compte dans la valeur `obj_size` qu'il fournit à la fonction.

`pages_per_slab :` le nombre de pages qui constituent chaque *slab*. Un petit nombre de pages par *slab* permet de limiter l'allocation de mémoire inutile, un grand nombre de pages évite de créer ou de détruire trop souvent des *slabs*.

`min_free_objs :` le nombre d'objets libres minimum que le *cache* doit contenir à tout moment. Cela est utile pour éviter les problèmes de récursivité avec `kmem_vmm`.

Enfin, le paramètre `cache_flags` permet de configurer deux comportements pour le *cache* (cumulables par un *ou logique*) :

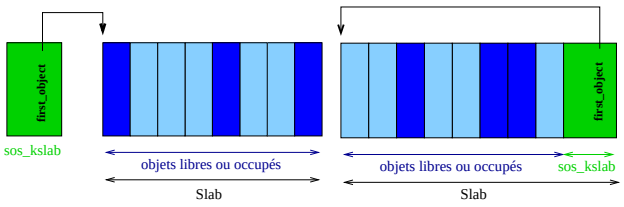
– Si le *flag* `SOS_KSLAB_CREATE_MAP` est positionné, alors les régions allouées sont automatiquement mappées en mémoire. Comme nous l'avons déjà

évoqué précédemment, ce paramètre est actuellement **obligatoire** puisque SOS ne supporte pas le *demand-paging* : tout accès en dehors d'une page mappée conduit à une exception de type défaut de page, qui n'est pas encore gérée (article 6).
– Si le *flag* `SOS_KSLAB_CREATE_ZERO` est positionné, alors les objets alloués seront remplis de zéros. Ce drapeau implique automagiquement le précédent.

L'implantation de `sos_kmem_cache_create()` est assez simple, bien qu'elle appelle de nombreuses fonctions internes intermédiaires. Elle commence par vérifier que les paramètres fournis sont cohérents. Puis elle alloue une nouvelle structure `sos_kslab_cache` pour le *cache* : on remarquera qu'elle utilise *kmem_slab* pour cela. Ensuite elle initialise cette structure à partir des paramètres (fonction interne `cache_initialize()`).

Cette initialisation consiste principalement à déterminer la configuration des *slabs*. Chaque *slab* correspond en effet à deux choses : une région de mémoire de taille établie à la création du *cache* (allouée par *kmem_vmm*) et une structure `struct sos_kslab` qui contient l'état du *slab* (nombre d'objets libres, liste des objets libres, etc.). Dans SOS, il y a deux façons de stocker une telle structure : soit dans un *cache* dédié (on dit qu'elle est "off slab", figure 10(a)), soit directement dans le *slab* (on dit qu'elle est "on slab", figure 10(b)). L'initialisation du *cache* décide laquelle de ces méthodes on choisit, suivant la taille des objets à allouer, la taille des *slabs*, etc. Cette décision sera consignée par la présence ou l'absence du drapeau `ON_SLAB` dans le champ *flags* de la structure `struct sos_kslab_cache`.

Pour terminer, si un nombre minimum d'objets est demandé (paramètre `min_free_objs`), alors un nouveau *slab* est immédiatement alloué. Pour cela, la fonction `cache_grow()` détaillée plus loin est utilisée. Enfin le nouveau *cache* est ajouté à la liste globale des *caches* du système.



(a) **En dehors** du *slab* (b) **Dans** le *slab*

FIG. 10 – Méthodes de stockage de la structure `sos_kslab`.

4.1.3 Destruction d'un cache

La destruction d'un *cache* s'opère grâce à la fonction `sos_kmem_cache_destroy(cache)`. Elle consiste en la destruction de tous les *slabs* par appel à *kmem_vmm*. Bien sûr, avant de le faire, elle s'assure qu'aucun objet n'y est alloué. La libération de la structure `struct sos_kslab_cache` termine la fonction, après l'avoir enlevée de la liste globale des caches du système.

4.2 Ajout et suppression de *slabs*

L'ajout et la suppression de *slabs* sont des opérations internes à l'allocateur. Elles interviennent au fur et à mesure de l'allocation et de la libération d'objets effectuées par l'utilisateur.

4.2.1 Ajout d'un *slab*

L'ajout d'un nouveau *slab* intervient à plusieurs moments. Le moment le plus courant est lors de l'allocation d'un nouvel objet par l'intermédiaire de la fonction `sos_kmem_cache_alloc()` : si tous les *slabs* sont pleins, alors un nouveau est ajouté. Un *slab* est également ajouté à un *cache* lors de sa création si un nombre minimum d'objets libres est spécifié (fonction `sos_kmem_cache_create()` vue précédemment).

L'ajout d'un *slab* s'effectue par la fonction interne `cache_grow(cache, flags)`. Elle procède de l'allocation d'une nouvelle région virtuelle par appel à *kmem_vmm*. Elle alloue éventuellement une nouvelle structure `sos_kslab` par appel à *kmem_slab* quand le *flag* `ON_SLAB` du *cache* n'est pas positionné (voir la section 4.1.2). Enfin, le *slab* est rajouté en tête de la liste des *slabs* du *cache* et le champ `slab` de la région virtuelle (voir la section 3.1) est mis à jour pour pointer sur le nouveau *slab* (fonction `sos_kmem_vmm_set_slab()` de *kmem_vmm*).

Un *slab* contient entre autres la liste des objets libres. C'est `cache_grow()` qui construit la liste initiale en y mettant tous les objets du *slab*. Il s'agit d'une liste chaînée classique, chaque élément de la liste (8 octets) étant stocké dans l'objet libre lui-même. C'est la raison pour laquelle la taille minimale des objets allouables dans un cache est de 8 octets. Les plus attentifs d'entre vous remarqueront que ceci implique que les *slabs* alloués doivent toujours être mappés en mémoire physique, même quand tous ses objets sont "libres" (car ils doivent contenir les champs de chaînage des objets libres). Ceci constitue LA différence majeure par rapport à [1] (voir les commentaires en tête de `sos/sos_kmem.h`). Et ceci remet en cause l'utilité du drapeau `SOS_KSLAB_CREATE_MAP` vu en section 4.1.2... Nous laissons à titre d'exercice un moyen de gérer l'ensemble des objets disponibles qui soit extérieur aux objets.

4.2.2 Suppression d'un *slab*

La suppression d'un *slab* intervient également à plusieurs moments :
– lors de la suppression d'un *cache* par la fonction `sos_kmem_cache_destroy()` : tous les *slabs* sont supprimés (voir la section 4.1.3).

– lors de la libération d'un objet par la fonction `sos_kmem_cache_free(vaddr)` : si le *slab* devient vide alors il est supprimé.
– dans la fonction `sos_kmem_cache_release_struct_range()` utilisée pour libérer un objet de type `sos_kmem_range` et utilisable exclusivement par *kmem_vmm* (voir la section 3.3.2).

La suppression d'un *slab* se déroule dans la fonction `cache_release_slab()`. Celle-ci enlève le *slab* de la liste des *slabs* du cache puis libère la région virtuelle associée. Si le *slab* n'est pas de type `ON_SLAB`, la structure `sos_kslab` est libérée par appel à *kmem_slab*.

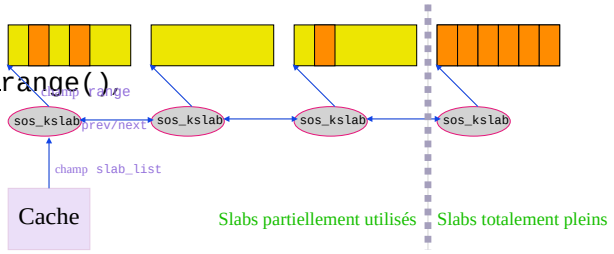


FIG. 11 – Organisation des listes de *slabs* dans un *cache*

4.3 Allocation et libération d'objets

L'allocation et la libération d'un objet sont des opérations accessibles à l'utilisateur par les fonctions `sos_kmem_cache_alloc()` et `sos_kmem_cache_free()`.

4.3.1 Allocation d'un objet

L'allocation d'un objet dans un *cache* est prise en charge par la fonction `sos_kmem_cache_alloc(cache, alloc_flags)`. Pour allouer un objet, on doit indiquer le cache concerné (paramètre `cache`). On peut également contrôler le comportement de l'allocation grâce au paramètre *alloc_flags* qui ne peut prendre que deux valeurs : 0 ou `SOS_KSLAB_ALLOC_ATOMIC` qui est relative à la gestion du swap dont nous reparlerons dans un prochain article.

La fonction puise un objet disponible (i.e. prêt à être alloué) dans un des *slabs* où il en reste encore. Et s'il n'en reste plus, ou plus exactement si le nombre d'objets libres dans le *cache* correspond au `min_free_objects` du *cache* (voir la section 4.4), alors un nouveau *slab* est alloué et un objet est puisé dedans.

Pour pouvoir déterminer très rapidement s'il existe des *slabs* contenant des objets disponibles, on maintient la propriété suivante dans toutes les fonctions qui modifient/ajoutent les *slabs* : les *slabs* complètement pleins sont en queue de la liste des *slabs* du *cache* ; tous les autres sont en *tête* de la liste (figure 11). Par exemple, si le *slab* devient plein après l'allocation de l'objet, on le transfère en queue de la liste. Ainsi, pour tenter de récupérer un *slab* avec des objets disponibles, il suffit de regarder dans le *slab* en tête de la liste.

4.3.2 Libération d'un objet

La libération d'un objet s'effectue en utilisant la fonction `sos_kmem_cache_free(vaddr_obj)`. Il est inutile de lui préciser le *cache* ou le *slab* contenant l'objet à libérer, l'adresse de celui-ci est suffisante.

Cette fonction courte appelle la fonction interne `free_object()`. La première étape consiste à déterminer quel *slab* couvre l'objet situé à l'adresse *vaddr*. Ceci s'effectue par appel à `sos_kmem_vmm_resolve_slab()` qui récupère d'abord l'adresse de la région virtuelle couvrant cette adresse (suivant le même principe que dans la section 3.3.1), puis qui utilise ensuite le champ `slab` de la région virtuelle (voir la section 4.2.1). Le reste de la fonction libère l'objet du slab, remet le *slab* en tête de liste des *slabs* du *cache*, ou le libère s'il devient totalement libre.

4.4 Considérations de récursivité

Le problème de récursivité à la libération a déjà été étudié (voir 3.3.2), mais il en reste un autre lors de l'allocation des `struct sos_kmem_range`.

Lors de la création d'une nouvelle région de mémoire (**i**) dans la fonction `sos_kmem_vmm_new_range()`, une nouvelle structure `struct sos_kmem_range` doit être allouée. Pour cela, (**ii**) l'allocateur par *slab* est utilisé, au travers de la fonction `sos_kmem_cache_alloc()` appliquée sur le *cache* des `struct sos_kmem_range` (variable `kmem_range_cache` au début de `kmem_vmm.c`).

Cependant, si il s'avère qu'il ne reste plus d'objets libres dans aucun *slab* de ce *cache*, alors (**iii**) un nouveau *slab* devra être ajouté à ce *cache*. Pour créer un nouveau *slab*, (**iv**) un appel à `sos_kmem_vmm_new_range()` est effectué pour allouer une nouvelle région de mémoire. Pourtant, nous sommes actuellement en train d'éxécuter cette fonction (étape **i**) : il y a récursivité indirecte. Le problème de cette récursivité est qu'elle est infinie puisqu'on se retrouve à nouveau dans l'obligation d'allouer un `struct sos_kmem_range` pour satisfaire **iv** puisqu'il est clair qu'il n'y a pas davantage de `struct sos_kmem_range` de disponibles qu'en **iii**. Le débordement de pile est inévitable.

Pour briser cette récursivité infinie, le *cache* des `struct sos_kmem_range` est créé avec le paramètre `min_free_objects` positionné à 2. Cela signifie que l'allocateur par *slab* n'attendra pas la pénurie d'objets `sos_kmem_range` (étape **iii**) pour allouer un nouveau *slab* : on déclenchera l'allocation de nouveaux *slabs* pour le cache des `struct sos_kmem_range` en avance, non pas quand il n'en restera plus du tout, mais dès qu'il n'en restera plus que 2 (par la fonction `sos_kmem_cache_alloc()`, voir la section 4.3.1). Il y a en quelque sorte *pré-allocation* de *slab* : il y aura toujours suffisamment de `struct sos_kmem_range`

disponibles pour allouer celui nécessaire à (**i**) et celui nécessaire pour pré-allouer un *slab* quand il ne reste plus que 2 `struct sos_kmem_range`.

5 L'allocateur de type `malloc/free`

L'allocateur par *slab* étant maintenant en place, il est possible de créer des *caches* pour chaque type d'objet du noyau. Mais on ne va pas créer un cache quand on sait qu'on ne va allouer qu'un petit nombre d'objets pour un type donné. C'est à cela que servent les primitives de type `malloc/free` : `sos_kmalloc()` et `sos_kfree()`. Elles sont déclarées dans `sos/kmalloc.h` et définies dans `sos/kmalloc.c`.

Elles reposent entièrement sur *kmem_vmm* et *kmem_slab*. Pour les objets de petites tailles, elles utilisent le sous-système *kmem_slab* : elles vont simplement puiser dans un jeu de *caches* prédéfini en haut de `sos/kmalloc.c` et créé par la fonction d'initialisation `sos_kmalloc_setup()`. Pour les plus gros objets, elles utiliseront directement des régions de mémoire virtuelle de *kmem_vmm*.

Pour les petits objets, nous définissons des caches pour les objets de tailles 8, 16, 32, 64, 128, 256, 1024, 2048, 4096, 8192, 16384 octets. Lorsqu'on veut allouer un objet de taille 3124 octets par exemple, on puisera dans le cache d'objets de 4096 octets, sachant qu'on perdra 972 de mémoire par objet alloué. Rien n'empêcherait de rajouter des caches pour d'autres tailles d'objets, qui ne doivent pas forcément être des puissances de 2 d'ailleurs.

6 Initialisation de l'allocateur

L'initialisation de l'ensemble de l'allocateur s'effectue au sein de la fonction `sos_kmem_vmm_setup()`. Elle va mettre en place conjointement l'allocateur de régions virtuelles (*kmem_vmm*) et l'allocateur par *slab* (*kmem_slab*). Les problèmes d'oeuf et de poule liés à l'interdépendance des deux sous-systèmes sont réglés par une initialisation en trois étapes.

La première étape est la pré-initialisation de l'allocateur par *slab* pour construire le cache pour les slabs, le cache pour les régions virtuelles et le cache pour... les caches. En effet, tous ces caches sont simultanément nécessaires pour que *kmem_vmm* et *kmem_slab* fonctionnent et ne pourront donc pas être créés "normalement" par *kmem_slab* puisqu'on est justement en train de l'initialiser. On n'a pas d'autre solution que de les créer totalement manuellement, en remplissant toutes les structures à la main (fonction `sos_kmem_cache_setup_prepare()`) et en allouant les slabs nécessaires à la main.

Une fois ces caches initialisés, *kmem_vmm* peut allouer les `struct sos_kmem_range` nécessaires pour être initialisé. Son initialisation (fonction `sos_kmem_vmm_setup()`) consiste à couvrir la zone du noyau (0-1Go) par des régions de sorte que ces régions reflètent la configuration initiale suivante (voir la figure 4 précédente et l'article 4) :

- la région de la mémoire vidéo ;
- la région contenant le code et les données du noyau, ainsi que le tableau des descripteurs des pages physiques (article 3) ;
- la région utilisée pour les *slab*s alloués manuellement par la fonction `sos_kmem_cache_setup_prepare()`.

La dernière étape de l'initialisation de l'allocateur par *slab* s'effectue par un appel à la fonction `sos_kmem_cache_setup_commit()`. Cette dernière va simplement mettre à jour les *slabs* qui avaient été créés manuellement en leur indiquant la région virtuelle qui les contient (champ `range` de `struct sos_kslab`). En effet, les `struct sos_kmem_range` viennent d'être créés par la fonction précédente et sont définis seulement maintenant.

On pourra remarquer que la zone de 4Mo pour le mirroring (i.e. $[1Go - 4Mo, 1Go[$, voir l'article 4) n'est pas couverte par *kmem_vmm*. En effet on a préféré que *kmem_vmm* couvre $[0, 1Go - 4Mo[$ plutôt que $[0, 1Go[$, i.e. s'arrêter juste avant le mirroring. Ceci garantit que *kmem_vmm* et *kmem_slab* ne toucheront jamais au mirroring, même en cas d'erreur grave de programmation.

7 Synthèse

Notre allocateur est hiérarchique à deux niveaux : l'allocateur de régions (*kmem_vmm*) pour la gestion de régions de pages de mémoire virtuelle contiguës, et l'allocateur par *slabs* (*kmem_slab*) pour l'allocation d'objets de taille homogène dans ces régions. Ces deux sous-systèmes reposent sur la gestion des pages de RAM (article 3) et la pagination (article 4) mais sont auto-suffisants en ce qui concerne la gestion de la mémoire virtuelle. Ainsi par exemple, *kmem_vmm* utilise *kmem_slab* pour allouer ses structures de fonctionnement `struct sos_kmem_range`, et pas un allocateur spécialisé à part (comme dans la plupart des implantations de [1]). Cette élégance dans la conception s'obtient cependant au prix d'une certaine lourdeur dans l'initialisation (section 6) et de quelques subtilités dans la réalisation pour briser toute récursivité (section 2.3) : à la libération de régions de mémoire (section 3.3.2) par déroulement de la récursivité et à l'allocation d'objets grâce à la pré-allocation de *slabs* (section 4.4).

En pratique, pour allouer des objets, on aura le choix entre créer ses propres caches et puiser dedans (section 4) ou appeler les fonctions `sos_kmalloc()/sos_kfree()` pour utiliser des caches génériques prédéfinis (section 5). On n'utilisera jamais directement, ou très peu, les fonctions de *kmem_vmm*.

Lorsqu'on crée un *cache*, on pourra contrôler

son comportement au moyen des drapeaux SOS_KSLAB_CREATE_MAP (obligatoire pour le moment) et SOS_KSLAB_CREATE_ZERO (section 4.1.2). Le paramètre min_free_objects, si fondamental pour éviter les problèmes de récursivité dans le cœur de l'allocateur, sera quasiment toujours fixé à 0 pour une utilisation normale. Et quand on alloue un objet dans un cache ou par sos_kmalloc(), on pourra contrôler le comportement de l'allocateur par le drapeau SOS_KSLAB_ALLOC_ATOMIC (section 4.3.1).

Par rapport à [1] et à ses implantations de référence (Solaris, Linux), nous avons fait quelques simplifications pour plus de lisibilité du code. Ces simplifications remettent en cause le sens habituel du mot "*cache*" pourtant au œur de notre allocateur, c'est pourquoi nous n'avons pas insisté sur l'origine de ce terme. Les plus curieux pourront se reporter d'abord à [1] puis aux commentaires en tête de sos/kmem_slab.h pour avoir la liste de ces simplifications. Elles sont presque toutes triviales à rajouter : avis aux amateurs !

8 Testons !

L'allocation de mémoire a une vocation utilitaire, donc la démo de ce mois-ci pouvait être une petite application simple qui s'écarte des fonctionnalités pures "OS". Malheureusement nous avons eu une panne d'idées ludiques. Tout ce que nous avons à proposer, c'est une bibliothèque de fonctions pour manipuler des grands entiers positifs, i.e. des entiers à plusieurs milliers ou millions de chiffres. La petite démo utilisera ces fonctions pour calculer le nombre 1000!, i.e. "factorielle 1000" (par définition : $1! = 1$ puis pour $n > 1$: $n! = (n-1)! * n$). La copie d'écran figure 12 donne les 55 chiffres les plus significatifs de 1000 ! (2568 chiffres au total).

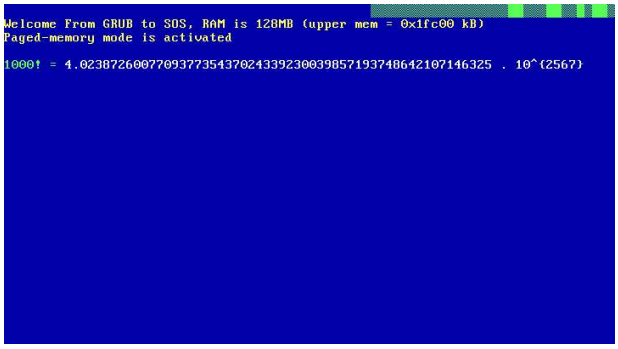


FIG. 12 – Aperçu de la petite démo

8.1 Représentation des données

Chaque nombre est représenté par une liste chaînée de chiffres décimaux, i.e. compris entre 0 et 9 (struct digit), ce qui est loin d'être optimal en terme de taux d'utilisation de la mémoire :

```
struct digit
{
  struct digit *prev, *next; /* To chain the digits */
  char value;                /* The value (0..9) of the digit */
};
```

On définit un type big_number_t désignant une liste d'éléments struct digit, c'est-à-dire un grand nombre :

```
typedef struct digit * big_number_t;
```

Dans cette liste, le chiffre le plus significatif du nombre est stocké en tête, le chiffre le moins significatif en queue.

Pour allouer les chiffres des nombres, nous utilisons sos_kmalloc()/sos_kfree(). Nous laissons à titre d'exercice la création et l'utilisation d'un cache dédié pour l'allocation des chiffres.

8.2 Fonctions de base

Nous définissons les opérateurs "addition" (bn_add()) et "multiplication" (bn_mult()). Leur implantation est loin d'être la plus efficace, en particulier pour la multiplication. Mais elle a l'immense mérite d'être extrêmement simple car elle reproduit ce que nous faisons avec un papier et un crayon.

Pour cela, nous utilisons un certain nombre de fonctions intermédiaires : bn_push_lsd(bn, v) pour rajouter le chiffre v à droite du nombre bn ($bn \leftarrow bn*10+v$, lsd signifie *least significant digit*), bn_push_msd(bn, v) pour rajouter le chiffre v à gauche du nombre bn ($bn \leftarrow v * 10^{\lfloor \log_{10}(bn) \rfloor + 1} + bn$, msd signifie *most significant digit*), bn_shift(bn, n) pour multiplier le nombre bn par 10^n, et bn_muli(bn, d) pour multiplier le nombre bn par le chiffre d. Nous donnons également la fonction de création d'un grand nombre à partir d'un entier habituel de la machine (int) : bt_new(int i), ainsi qu'une fonction d'affichage des premiers chiffres significatifs bn_print_console().

8.3 Le test

Pour calculer la factorielle (fonction bn_fact()), nous prenons soin de toujours bien libérer les résultats intermédiaires qui ne sont plus utiles. Au-delà d'optimiser l'utilisation de la mémoire (il y a quand même de la marge avant de saturer la mémoire par des nombres...), cela permet de stresser un peu l'allocateur dans son ensemble (à la fois *kmem_vmm*, *kmem_slab* et sos_kmalloc()/sos_kfree()) :

```
/* Result is the factorial of an integer */
big_number_t bn_fact(unsigned long int v)
{
  unsigned long int i;
  big_number_t retval = bn_new(1);
  for (i = 1 ; i <= v ; i++)
    {
      big_number_t I   = bn_new(i);
      big_number_t tmp = bn_mult(retval, I);
      sos_x86_videomem_printf(4, 0,
                              SOS_X86_VIDEO_BG_BLUE | SOS_X86_VIDEO_FG_LTGREEN,
                              "%d! = ", (int)i);
      bn_print_console(4, 8, SOS_X86_VIDEO_BG_BLUE | SOS_X86_VIDEO_FG_WHITE,
                       tmp, 55);
      bn_del(& I);
      bn_del(& retval);
      retval = tmp;
    }

  return retval;
}
```

À chaque itération, cette fonction affiche les 55 premiers chiffres de la factorielle. Si le port `0xe9` de bochs ou de qemu est activé (voir l'article 1), tous les chiffres de 1000! seront affichés sur le terminal où bochs/qemu a été lancé par la fonction `main()` (grâce à la fonction `bn_print_bochs()`).

Conclusion

Nous venons de décrire l'implantation de l'allocation de l'espace virtuel du noyau dans SOS. L'allocateur retenu est de type ségrégationniste simple et s'inspire de ceux utilisés dans les OS classiques tels que Solaris et Linux. Son implantation fait apparaître quelques subtilités liées au fait qu'il repose sur une hiérarchie de 2 sous-systèmes qui sont en réalité fortement interdépendants.

Cet article marque la fin d'une première "saison" dans notre saga sur la réalisation de *votre* OS. À partir de maintenant les briques de base de l'OS sont posées pour la gestion d'un élément essentiel : la mémoire. La prochaine "saison" débutera avec la gestion d'un autre élément essentiel d'un système informatique : le processeur. Nous attaquerons en effet la problématique du multitâche et de la synchronisation, avec la notion de *threads* noyau.

D'ici là, il va falloir vous armer d'un peu de patience car nous faisons une pause d'un mois. Le prochain épisode sera donc votre cadeau de début d'année 2005 ! Enfin : "cadeau"... ce sera à vous de juger.

Bon vent !

Thomas Petazzoni et David Decotigny
`Thomas.Petazzoni@enix.org` et `d2@enix.org`
Merci à Nessie pour sa relecture, ses remarques constructives et ses propositions.
Site de SOS : `http://sos.enix.org`
Projet KOS : `http://kos.enix.org`
Á Ludwig

Références

[1] J. Bonwick. The slab allocator : An object-caching kernel memory allocator. In *USENIX Technical Conference*, Summer 1994.

[2] Uresh Vahalia. *UNIX Internals : The New Frontiers*. Number ISSN 0131019082. Prentice Hall, 1995.

[3] Brad Fitzgibbons. The Linux Slab allocator. `http://www.cc.gatech.edu/people/home/bradf/cs7001/proj2/`.