Croisière au cœur d'un OS*
Étape 3 : Gestion de la mémoire physique

Résumé

Les deux précédents articles ont permis d'aboutir à un système minimaliste, capable de démarrer et de réagir à des interruptions (exceptions ou IRQs). Cet article et les deux suivants vont se focaliser sur une partie importante d'un système d'exploitation : la gestion de la mémoire. Cet article s'attaque précisément à la gestion de la *mémoire physique*.

Introduction

Les OS courants disposent d'un mécanisme dynamique d'allocation de mémoire pour le noyau. SOS bénéficie d'un tel mécanisme, qui repose sur trois composants : gestion de la mémoire physique, pagination, allocateur d'objets de taille arbitraire. Cet article est consacré à la première étape : la gestion de la *mémoire physique*. Pour une fois, les notions abordées ici ne sont pas spécifiques à l'architecture `x86`. D'ailleurs on trouvera l'essentiel du code dans le répertoire `sos/` et non pas dans `hwcore/`.

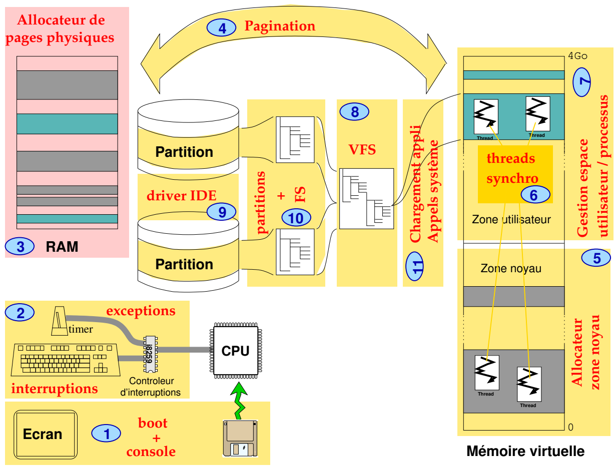


FIG. 1 – Programme des articles

*La version originale de cet article a été publiée dans GNU Linux Magazine France numéro 64 – Septembre 2004 (`http://www.linuxmag-france.org`) et cette version est diffusée avec l'autorisation de l'éditeur.

1 Gestion de la mémoire physique

Jusqu'ici, l'ensemble du code et des données utilisées par SOS était alloué "en statique", c'est-à-dire par le compilateur dans le fichier exécutable du noyau. Ces allocations correspondaient soit aux variables globales, soit aux variables locales déclarées "`static`", soit encore au code. Toutefois, comme dans de nombreux systèmes d'exploitation, SOS pourra allouer dynamiquement de la mémoire supplémentaire selon ses besoins. Cela permettra par exemple de créer des processus, de stocker les données nécessaires à la gestion des systèmes de fichiers, etc...

Le principe général de l'allocation dynamique de mémoire est assez simple. Il s'agit de connaître les emplacements libres et les emplacements occupés de la mémoire et d'écrire un jeu de fonctions permettant d'allouer et de libérer ces emplacements. Pour éviter tout écrasement du code ou des données allouées, la contrainte élémentaire est que les emplacements alloués ne peuvent pas être ré-alloués pour une autre utilisation sans avoir été libérés entre-temps.

La première étape pour permettre l'allocation dynamique de mémoire est de gérer la mémoire physique, c'est-à-dire la mémoire physiquement présente dans l'ordinateur.

Pour des raisons liées à la pagination (que nous étudierons au prochain article), nous décidons de découper la mémoire physique en zones de 4 Ko, appelées *pages physiques*. Nous avons donc à implanter ici le mécanisme d'allocation le plus simple qui soit : il s'agit seulement de savoir quelles pages physiques sont occupées ou non et de permettre l'allocation et la libération de ces pages.

En toute rigueur, nous devrions aussi parler de l'allocation et de la libération de zones plus grandes ou plus petites que 4 ko. Ceci sera en fait pris en charge par les deux mécanismes que nous présenterons dans les deux prochains articles. Ces deux mécanismes utiliseront évidemment le présent gestionnaire de mémoire physique.

2 Implantation

Notre implantation de la gestion de la mémoire physique utilise un jeu de macros facilitant la manipulation de listes chaînées. Nous commencerons par rappeler le principe de ces listes, puis nous décrirons rapidement le jeu de macros et enfin nous détaillerons notre implantation de la gestion de la mémoire physique proprement dite.

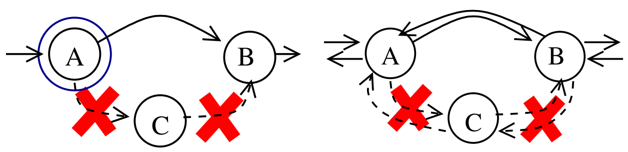
2.1 Gestion des listes

2.1.1 Rappels sur les listes chaînées

Le système de manipulation de listes chaînées que nous utilisons gère des *listes doublement chaînées circulaires*.

Une *liste (simplement) chaînée* est une suite d'éléments dans laquelle chaque élément connaît l'élément qui le suit. Un peu comme une série de wagons pour lesquels l'attache d'un wagon est fixée au suivant : il faut être dans un wagon pour accéder au wagon suivant. Sauf que dans ces listes, seul l'élément suivant est connu et non le précédent (*ie* une liste simplement chaînée ne peut être parcourue que dans un sens). Insérer un nouvel élément C entre 2 éléments A et B revient à connaître A, et à remplacer le lien $A \rightarrow B$ par deux liens $A \rightarrow C$ et $C \rightarrow B$. Supprimer de la liste cet élément C, situé entre A et B, revient à retrouver à nouveau A et à faire l'opération inverse de la précédente (figure 2(a)). Tout le problème avec ces listes est que toute opération d'insertion/suppression d'éléments à un endroit précis nécessite de connaître l'élément qui *précède* l'endroit où on veut effectuer l'opération. Or, pour retrouver cet "élément qui précède", on est obligé de parcourir la liste depuis le début puisqu'elle ne peut être parcourue que dans un seul sens. Ceci peut prendre un temps proportionnel à la longueur de la liste, qui peut être très longue.

Une *liste doublement chaînée* est une suite d'éléments dans laquelle chaque élément connaît à la fois l'élément qui le suit dans la liste, mais aussi à l'élément qui le précède. Ainsi, la liste peut cette fois être parcourue dans les deux sens. Ajouter ou supprimer un élément (figure 2(b)) impliquera donc de modifier l'accès au suivant de l'élément précédent et l'accès au précédent de l'élément suivant, pour les deux éléments entourant l'élément à ajouter ou à supprimer. L'intérêt de ces listes est que ceci se fait en un temps indépendant du nombre d'éléments dans la liste, au prix d'un doublement de l'espace occupé pour stocker les liens entre les éléments (2 liens au lieu d'un).

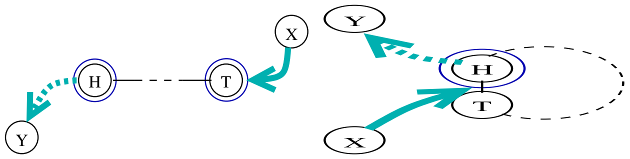


(a) Simplement chaînée : on doit obligatoirement connaître A

(b) Doublement chaînée : il suffit de connaître A ou B ou C

FIG. 2 – Suppression d'un élément C

Une *liste circulaire* est une liste simplement ou doublement chaînée dont le dernier élément est rattaché au premier élément : la liste n'est plus linéaire mais forme une boucle. Une telle liste permet notamment de représenter une file le plus simplement possible. Une file est une liste pour laquelle un nouvel élément ne peut être ajouté qu'en fin de liste et un élément déjà inséré ne peut être supprimé qu'en début de liste. Pour réaliser ces opérations, il faut donc connaître l'élément en tête de liste et l'élément en fin de liste (figure 3(a)). Avec une liste circulaire, on est capable d'accéder efficacement (*ie* en temps constant) à partir d'un seul élément à la fois à l'élément en tête et à celui en fin de liste (figure 3(b)). Un exemple concret d'une file est le tampon des caractères tapés au clavier, ou encore le tampon des paquets venant du réseau.



(a) Liste linéaire : il vaut mieux connaître la tête **et** la fin de la liste

(b) Liste circulaire : connaître l'un des deux suffit

FIG. 3 – Implantation d'une file à l'aide d'une liste

Les *listes doublement chaînées circulaires* cumulent les avantages des listes doublements chaînées et des listes circulaires : insertion/suppression d'éléments à un emplacement précis en temps constant (listes doublement chaînées), implantation des files sans nécessiter de définir de structures de données supplémentaires (listes circulaires).

2.1.2 Utilisation des macros

Afin de faciliter la manipulation des listes doublement chaînées circulaires, nous les avons implantées sous forme de macros (fichier `sos/list.h`). Cette implantation permet d'avoir une gestion générique pour tous les types de données, tout en s'affranchissant des *transtypages* (*castings*) inutiles et en conservant les vérifications du typage par le compilateur. Son utilisation ne sera pas limitée à la gestion de la mémoire physique.

Par défaut, ces macros de manipulation de listes supposent que les structures passées en paramètre possèdent les champs `prev` et `next` pour chaîner les éléments. Il est aussi possible de donner un autre nom à ces champs : chaque macro possède un équivalent se terminant par `_named` qui permet de spécifier des noms de champs *précédent* et *suivant* différents de ces `prev/next`.

2.1.3 Exemple

Avant de détailler les macros, illustrons leur utilisation au travers d'un exemple.

La première étape consiste à définir le type des éléments à chaîner (`struct integer`), ainsi qu'un pointeur vers le premier élément de la liste (`list`) :

```
struct integer {
  int val;
  struct integer *prev, *next;
};

struct integer *list;
```

Avant de commencer à utiliser la liste, il convient de l'initialiser en indiquant par exemple qu'elle est vide :

```
list_init(list);
```

On peut ensuite insérer un élément pointé par `element` en tête ou en queue. Dans les deux cas, l'insertion se fera en temps constant, car la liste est circulaire et doublement chaînée :

```
list_add_head(list, element); /* En tête */
list_add_tail(list, element); /* En queue */
```

Supprimer cet élement est tout aussi simple et s'effectue également en temps constant puisque la liste est doublement chaînée :

```
list_delete(list, element);
```

En supposant que les champs de chaînage *précédent* et *suivant* étaient nommés `prev_in_stuff` et `next_in_stuff`, le code précédent aurait dû être :

```
list_init(list, prev_in_stuff, next_in_stuff);
list_add_head(list, element, prev_in_stuff, next_in_stuff);
list_delete(list, element, prev_in_stuff, next_in_stuff);
```

Des macros existent également pour parcourir une liste :

```
struct integer *list;
struct integer *current;
int nb_elts;

list_foreach(list, current, nb_elts)
{
  /* Faire quelque chose avec 'current' */
}
```

Dans l'extrait précédent, les variables `current` et `nb_elts` sont mises à jour à chaque itération.

2.1.4 Rôle des macros disponibles

Nous détaillons ci-dessous l'ensemble des macros disponibles pour la manipulation des listes chaînées. Comme nous l'avons dit précédemment, chacune de ces macros comporte un équivalent dont le nom se détermine par `_named` pour spécifier les noms des champs de chaînage.

list_init(list) Initialisation d'une liste à la liste vide

list_singleton(list,item) Initalisation d'une liste à l'élément unique `item`

list_is_empty(list) Indique si la liste est vide ou non

list_get_head(list) Indique l'élément en tête de liste

list_get_tail(list) Indique l'élément en queue de la liste

list_insert_after(list,after_this,item) Insère l'élément `item` *après* l'élément `after_this` dans la liste `list`

list_insert_before(list,before_this,item) Insère l'élément `item` *avant* l'élément `before_this` dans la liste `list`

list_add_head(list,item) Insère un élément en tête de la liste

list_add_tail(list,item) Insère un élément en queue de la liste

list_delete(list,item) Retire un élément de la liste (ne vérifie pas si l'élement était réellement dans la liste)

list_pop_head(list) Récupère l'élément en tête de la liste et le retire de la liste

list_foreach_forward(list,iterator,nb_elements) Permet de boucler sur tous les éléments d'une liste, à partir de l'élément de tête jusqu'à l'élément de queue

list_foreach_backward(list,iterator,nb_elements) Comme la précédente, sauf que le parcours se fait de l'élément de queue vers l'élément de tête

list_collapse(list,iterator) Boucle analogue à `list_foreach_forward()`, sauf qu'à l'issue de chaque itération l'élément courant est retiré de la liste

Á noter que quand nous disons "retire", cela signifie que la liste ne contient plus de référence à l'élément retiré de la liste. Mais l'élément retiré de la liste reste toujours présent en mémoire.

2.2 Gestionnaire de mémoire physique `physmem`

L'ensemble du système de gestion de mémoire physique est implanté dans le fichier `sos/physmem.c`. Le fichier `sos/physmem.h` contient les prototypes des fonctions définies et également quelques macros tout à fait **essentielles** pour le reste du noyau SOS.

2.2.1 Macros fondamentales

Le fichier `sos/physmem.h` définit tout d'abord les trois constantes fondamentales à toute la gestion de la mémoire :

```
/** The size of a physical page (arch-dependent) */
#define SOS_PAGE_SIZE  (4*1024)

/** The corresponding shift */
#define SOS_PAGE_SHIFT 12 /* 4 kB = 2^12 B */

/** The corresponding mask */
#define SOS_PAGE_MASK  ((1<<12) - 1)
```

La première macro définit la taille d'une page physique. Les macros suivantes permettent d'accélérer certaines opérations arithmétiques, en les transformant en opérations booléennes élémentaires puisque la taille d'une page physique est une puissance de 2 :

- `x << SOS_PAGE_SIZE_SHIFT` est équivalent à `x * SOS_PAGE_SIZE`
- `x >> SOS_PAGE_SIZE_SHIFT` est équivalent à `x / SOS_PAGE_SIZE` (arrondi entier inférieur)
- `x & SOS_PAGE_SIZE_MASK` est équivalent à `x modulo SOS_PAGE_SIZE`

Dans certains OS, dont Linux, ces valeurs tendent à ne plus être des constantes. C'est-à-dire que ces noyaux sont appelés à gérer des pages de mémoire de tailles variables. Ce ne sera pas le cas dans SOS.

Deux autres macros très pratiques pour toute la suite sont également définies dans ce même fichier d'en-tête :

```
#define SOS_PAGE_ALIGN_INF(val) [...]
#define SOS_PAGE_ALIGN_SUP(val) [...]
```

Elles reposent elles-mêmes sur des macros définies dans `sos/macros.h`. Pour une adresse donnée `val`, leur objectif est de calculer l'adresse de la page inférieure (resp. supérieure) ou égale la plus proche. Pour l'adresse `0x1234` par exemple, ces macros renverront respectivement `0x1000` et `0x2000`.

2.2.2 Descripteurs de pages physiques

À chaque page physique est associée une structure de type `physical_page_descr`, qui définit un *descripteur de page physique*. Cette structure contient l'adresse physique de la page, un compteur de références à la page, et des pointeurs *précédent* et *suivant* pour chaîner la structure dans une liste :

```
struct physical_page_descr
{
  /** The physical base address for the page */
  sos_paddr_t paddr;

  /** The reference count for this physical page.
      > 0 means that the page is in the used list. */
  sos_count_t ref_cnt;

  /** The other pages on the list (used, free) */
  struct physical_page_descr *prev, *next;
};
```

Deux listes chaînées sont en effet créées : une liste pour les pages physiques libres (`free_ppage`) et une liste pour les pages physiques occupées (`used_ppage`).

Tous les descripteurs de page sont par ailleurs stockés dans un unique tableau : le descripteur i du tableau est associé à la page $[i * 4\text{Ko},\ (i+1) * 4\text{Ko}[$. Ceci permet de connaître très simplement le descripteur associé à n'importe quelle adresse physique, comme le montre le code de la fonction interne suivante :

```
inline static struct physical_page_descr *
get_page_descr_at_paddr(sos_paddr_t ppage_paddr)
{
  return physical_page_descr_array
         + (ppage_paddr >> SOS_PAGE_SHIFT);
}
```

La figure 4 résume le fonctionnement global de ce système : la partie de la mémoire physique en cyan en bas renferme le tableau des descripteurs, détaillé (sous la forme d'un zoom) par le tableau en haut de la figure.

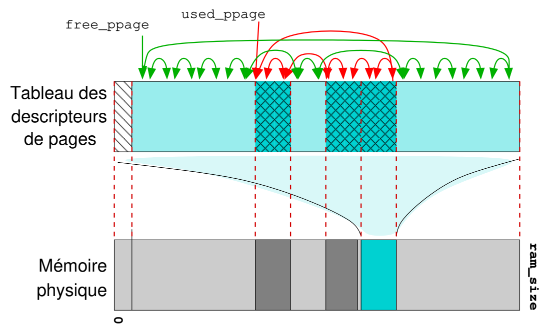


FIG. 4 – Tableau et listes des descripteurs de pages physiques

Le *compteur de références* dont il a été question auparavant est un simple entier mis à jour par les fonctions d'allocation et de libération des pages de mémoire physique. Une valeur de 0 indique que la page n'est utilisée (ou *référencée*) par personne et donc que le descripteur de page se trouve dans la liste `free_pages`. Une valeur strictement positive indique le nombre de fois où cette page est utilisée (ou *référencée*) et donc le descripteur de page se trouve dans la liste `used_pages`. Le mécanisme de comptage de références est un mécanisme classique pour libérer automatiquement la mémoire inutilisée. Ainsi, un même objet (ici : une *page*) peut être utilisé (*référencé*) par plusieurs personnes et l'objet est automatiquement libéré lorsque le compteur passe à 0, c'est-à-dire quand tout le monde a déclaré ne plus l'utiliser.

2.2.3 Initalisation

La fonction `sos_physmem_setup()` est chargée de l'initialisation du système de gestion de la mémoire physique. Elle prend en argument la taille de la mémoire physique disponible, `ram_size`, fournie par Grub exclusivement (le secteur de boot décrit dans l'article 1 ne permet pas de la connaître). Elle retourne les adresses physiques de début et de fin de la "*zone noyau*" utilisée par le code et les données du noyau. Cette zone renferme à la fois l'exécutable du noyau chargé par Grub et le tableau des descripteurs des pages physiques. Les adresses retournées seront utilisées dans les articles suivants.

La fonction commence par calculer l'adresse de début de la zone noyau en utilisant le symbole `__b_kernel`. Celui-ci est défini dans `support/sos.lds`, le script de linkage de SOS.

La fonction poursuit en calculant l'adresse de fin de la zone noyau. Pour cela, il s'agit d'abord de calculer où va pouvoir se situer le tableau des descripteurs de pages physiques. Nous choisissons de le placer après l'exécutable du noyau chargé par Grub (voir la figure 5). L'adresse de ce tableau est contenue dans la macro `PAGE_DESCR_ARRAY_ADDR`. Elle correspond en fait à la première page libre après la fin de l'exécutable du noyau, elle-même définie par le symbole `__e_kernel` dans `support/sos.lds`. À cette adresse, on ajoute la taille du tableau, c'est-à-dire le nombre de pages de mémoire physique multiplié par la taille d'un descripteur, afin d'obtenir l'adresse de fin de la zone noyau :

```
#define PAGE_DESCR_ARRAY_ADDR \
  SOS_PAGE_ALIGN_SUP((sos_paddr_t) (& __e_kernel))
[...]
  *kernel_core_base = SOS_PAGE_ALIGN_INF((sos_paddr_t)(& __b_kernel));
  *kernel_core_top
    = PAGE_DESCR_ARRAY_ADDR
      + SOS_PAGE_ALIGN_SUP( (ram_size >> SOS_PAGE_SHIFT)
                            * sizeof(struct physical_page_descr));
```

La fonction écrit ensuite dans `physmem_base` et `physmem_top` les adresses de début et de fin de la mémoire physique gérée par le gestionnaire de mémoire physique `physmem`. `physmem_base` n'est pas initialisé à 0 car on ne souhaite pas gérer la première page physique. Ainsi, lorsqu'une fonction de `physmem` renverra l'adresse 0, on pourra considérer qu'il s'agit d'un statut d'erreur. Enfin, `physmem_top` correspond à la taille de mémoire physique détectée (paramètre `ram_size` passé à la fonction).

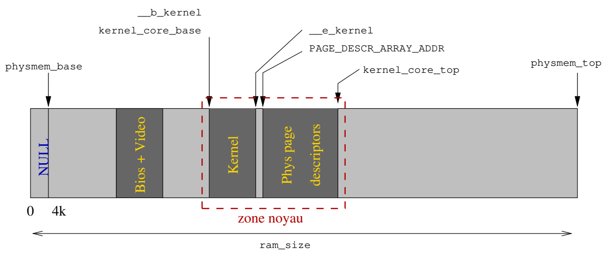


FIG. 5 – Carte de la mémoire physique

Une fois ces différentes variables configurées, l'initialisation des descripteurs proprement dite commence. Pour chaque descripteur, l'adresse physique de la page correspondante est calculée. Ensuite, en fonction de l'adresse de la page, on affecte un comportement différent. Les pages peuvent soit être réservées (c'est le cas uniquement pour la première page), soit être libres, soit être utilisées par la projection en mémoire du BIOS et des mémoires vidéos, soit être occupées par le noyau ou le tableau des descripteurs lui-même.

Le comportement associé à une page libre consiste à initialiser le compteur de références du descripteur à 0 et à l'ajouter dans la liste des pages libres. Le comportement associé à une page occupée (par le noyau ou par une projection en mémoire d'éléments matériels) consiste à initialiser le compteur de références du descripteur à 1 et à l'ajouter dans la liste des pages occupées. Pour les pages réservées, rien n'est effectué.

La figure 5 indique le plan mémoire obtenu à l'issue de l'initialisation, *ie* avant les premières allocations.

2.2.4 Allocation

L'allocation de mémoire physique s'effectue page par page en appelant la fonction `sos_physmem_ref_physpage_new()`. On peut la voir comme une sorte de `malloc(4096);` d'un programme utilisateur sous Unix. Cette fonction prend un booléen `can_block` en paramètre, pour l'instant inutilisé. Il sera utile lorsque le système de *swap* sera mis en place.

`sos_physmem_ref_physpage_new()` s'occupe très simplement de récupérer la première page dans la liste des pages libres, d'incrémenter son compteur de références, puis d'ajouter le descripteur dans la liste des pages occupées, avant de retourner l'adresse physique de la page allouée :

```
sos_paddr_t
sos_physmem_ref_physpage_new(sos_bool_t can_block)
{
  struct physical_page_descr *ppage_descr;

  /* Retrieve a page in the free list */
  ppage_descr = list_pop_head(free_ppage);

  /* Mark the page as used (set ref count to 1) */
  ppage_descr->ref_cnt = 1;

  /* Put the page in the used list */
  list_add_tail(used_ppage, ppage_descr);

  return ppage_descr->paddr;
}
```

2.2.5 Déréférencement d'une page

La fonction `sos_physmem_unref_physpage()` permet de diminuer le nombre de références à une page physique, c'est-à-dire de décrémenter son compteur de références. Et lorsque celui-ci atteint 0, la page est libérée.

Pour le moment, nous pouvons considérer que la fonction `sos_physmem_unref_physpage()` est une sorte d'appel à `free()` par une application Unix car nous n'utilisons pour l'instant (dans la petite démo) que `sos_physmem_ref_physpage_new()` qui positionne les compteurs de références à 1. Dans ces conditions en effet, la page sera libérée systématiquement à l'appel de cette fonction puisque la condition du `if` ci-dessous sera toujours vérifiée :

```
sos_ret_t
sos_physmem_unref_physpage(sos_paddr_t ppage_paddr)
{
  /* By default the return value indicates
     that the page is still used */
  sos_ret_t retval = FALSE;

  struct physical_page_descr *ppage_descr
    = get_page_descr_at_paddr(ppage_paddr);

  /* Unreference the page, and, when no mapping is
     active anymore, put the page in the free list */
  ppage_descr->ref_cnt--;
  if (ppage_descr->ref_cnt <= 0)
    {
      /* Transfer the page, considered USED,
         to the free list */
      list_delete(used_ppage, ppage_descr);
      list_add_head(free_ppage, ppage_descr);

      /* Indicate that the page is now unreferenced */
      retval = TRUE;
    }

  return retval;
}
```

La fonction renvoie `TRUE` lorsque la page a été effectivement libérée, `FALSE` lorsqu'elle est toujours utilisée, et un code d'erreur (négatif) sinon.

2.2.6 Référencement d'une page

La fonction `sos_physmem_ref_physpage()` permet d'augmenter le nombre de références à une page physique, c'est-à-dire d'incrémenter le compteur de références. Nous ne l'utilisons pas pour le moment, mais son code fait l'inverse de la précédente : si la page indiquée par le paramètre `ppage_paddr` n'est pas encore allouée (son compteur de références vaut 0), alors cette opération est réalisée : on la retire de la liste `free_pages` pour l'insérer dans la liste `used_pages`.

2.2.7 Précautions d'usage

Les fonctions précédentes permettent d'allouer et de libérer des pages : les précautions à prendre avec ces fonctions sont donc analogues à celles qu'on doit prendre avec les célèbres `malloc()`/`free()`.

Toutefois, elles ont une autre caractéristique. Elles œuvrent en effet à une gestion *logique* des ressources (les pages physiques) ; les ressources physiques, elles, restent toujours disponibles (la mémoire ne va pas s'évaporer). Ceci veut dire que si l'on manipule des données à des adresses physiques en dehors des pages marquées "utilisées", le noyau ne plantera pas immédiatement. Il n'en demeure pas moins qu'il y a un bogue latent, puisqu'on devrait en principe accéder uniquement aux pages marquées "utilisées". Tôt ou tard, il y aura écrasement de données et donc plantage ou comportement incorrect.

Nous verrons dans l'article suivant que la pagination permet, entre autres, de détecter au plus tôt une partie de ces bogues.

2.3 Et dans Linux, comment ça marche ?

Les connaisseurs de la gestion de la mémoire physique sous Linux seront étonnés de voir que le code de SOS est si simpliste. Linux établit en effet une hiérarchie de gestionnaires de la mémoire physique à deux niveaux [1].

Au premier niveau de la hiérarchie se trouve le nécessaire pour supporter les systèmes NUMA (*Non Uniform Memory Access*), regroupant en particulier les systèmes distribués de type *clusters* [2], voire les systèmes à image unique [3]. Dans ces systèmes, chaque *nœud* (processeur ou multiprocesseur) a intérêt à allouer de la mémoire physique en priorité sur le nœud local. Donc Linux maintient une liste de *nœuds* schématiquement associés à des plages d'adresses et classés dans l'ordre de leur proximité du nœud courant.

Le deuxième niveau de la hiérarchie correspond à la gestion de la mémoire physique sur chaque nœud. Pour des raisons liées à des limitations dans la gestion de l'espace virtuel du noyau Linux (nous en reparlons dans l'article suivant), cette mémoire physique est découpée en plusieurs *zones*. Une zone est un intervalle d'adresses physiques contiguës associé à une priorité : on allouera d'abord dans la zone de plus grande priorité. Par exemple, sur le nœud courant, il y a 3 zones. Dans l'ordre des priorités décroissantes : de 16 Mo à 896 Mo pour les adresses physiques courantes, de 0 à 16 Mo pour le DMA (principalement sur bus ISA), puis de 896 Mo à 4 Go ou 64 Go pour la zone "highmem". La gestion de la mémoire dans chaque zone repose sur un allocateur simple de type "*buddy system*" [1, Chapitre 7] dont nous reparlerons dans l'article 5.

Dans SOS, d'une part la gestion du NUMA n'est pas envisagée, donc la hiérarchie de premier niveau (les *nœuds*) n'apparaît pas. Et d'autre part nous ne découpons pas la gestion de la mémoire en gestionnaires de zones. En effet, notre gestion de la mémoire virtuelle, que nous verrons dans l'article suivant, permet de modifier librement et dynamiquement les pages physiques occupées par le noyau. Ceci nous permettra par exemple de réquisitionner au vol des pages physiques dans la zone 0-16Mo (DMA sur bus ISA), même si elles sont utilisées par le noyau.

3 Testons !

Conformément à l'habitude, la petite démo est aussi peu excitante que les précédentes (voir la figure 6). Elle se résume à la fonction `sos/main.c:test_physmem()`, qui est constituée de deux boucles.

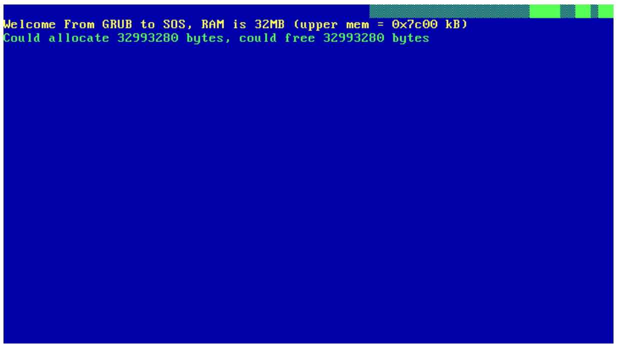


FIG. 6 – Aperçu de la petite démo

La première boucle alloue une page physique à chaque itération, jusqu'à ce que plus aucune page physique ne soit disponible. Le cœur de la boucle consiste ensuite à écrire sur chaque octet de la page allouée : d'une part 2 fois 511 entiers et d'autre part 2 pointeurs de chaînage `prev/next`. Ces éléments sont rassemblés dans une structure `struct my_ppage` qui possède exactement la taille d'une page. Les entiers précédents sont tous affectés à la valeur "adresse de la page". Et les pointeurs de chaînage servent à ajouter chaque page allouée dans la liste des pages allouées par le test (`ppage_list`).

La seconde boucle libère une page de la liste des pages allouées `ppage_list` à chaque itération, jusqu'à ce que cette liste soit vide. Le cœur de la boucle consiste ensuite à vérifier que les 2 fois 511 entiers de la page valent exactement l'adresse de la page. S'il n'y avait plus égalité, alors cela signifierait qu'il y aurait eu écrasement de données, et la démo l'indiquerait.

Enfin, une fois les boucles terminées, on indique la taille de la mémoire qui a été allouée puis libérée.

Important : Pour mettre en place le tableau des descripteurs de pages physiques, le système a besoin de connaître la taille de la mémoire physique (voir la section 2.2.3). Des deux moyens de chargement de SOS (Grub, ou secteur de boot), seul Grub permet de connaître cette taille. Par conséquent, dorénavant, pour tester SOS, seul Grub conviendra comme chargeur d'OS. Si vous utilisez le secteur de boot fourni dans l'article 1, tout ce que fera SOS sera de vous afficher le message "I'm not loaded with Grub !".

Conclusion

Avec la gestion de la mémoire physique, la première brique de la gestion de la mémoire du noyau de SOS vient d'être posée. Le prochain article abordera le fonctionnement de la pagination et sa mise en place de SOS. Profitez de ce démarrage en douceur, rassemblez toutes les forces qui risquent de vous être nécessaires pour affronter la tempête de neurones qui vous attend au prochain article !

Dernières nouvelles : via la liste de diffusion de SOS, Xavier Grave nous a fait parvenir une contribution intéressante et amusante. Il s'agit d'une version de SOS entièrement réécrite en Ada, qui compile et s'exécute ! Pour plus de détails, n'oubliez pas d'aller faire un tour sur le site de SOS `http://sos.enix.org`. D'ailleurs, allez-y régulièrement car nous y mettons les *erratas* (page "Bugs") des articles et les patches associés.

Thomas Petazzoni et David Decotigny
`Thomas.Petazzoni@enix.org` et `d2@enix.org`

Grand 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 Georg Friedrich

Références

[1] Mel Gorman. Understanding the Linux virtual memory manager. `http://www.skynet.ie/~mel/projects/vm/guide/html/understand/`, 2004.

[2] Open Mosix. `http://openmosix.sf.net`.

[3] OpenSSI. `http://openssi.org`.