

Listes chaînées (I)

Adrien Reboisson

13th December 2005

1 Présentation

1.1 Introduction

L’objectif de ce document est de réaliser en C la simulation du gestionnaire de mémoire ultrasimple du système d’exploitation **Zindo** (qui n’existe évidemment pas mais qui reprends les caractéristiques de systèmes réels !).

En gros, on se donne un espace mémoire de n octets que l’on souhaite rendre disponible à diverses applications. Notre but est juste de voir comment maintenir une “cartographie” de la mémoire (en mémorisant les espaces disponibles et les espaces déjà occupés) afin que l’on puisse simuler la réservation et la libération de certains blocs de mémoire. Il ne faudra d’ailleurs pas hésiter à utiliser des dessins pour visualiser les situations abordées dans la simulation.

En réalité, il va de soi que les n octets ne sont jamais alloués. On s’intéresse uniquement à la structure de données utilisée pour gérer la mémoire de cette architecture. Le choix ici est d’utiliser une liste chaînée, où chaque maillon de la liste représente un espace réservé par une application. Par la suite, un maillon sera appelé un *descripteur*, car il décrit l’adresse de début du bloc de mémoire ainsi que sa taille. Evidemment, ce qui est appelé ici “adresse de début de bloc” est un entier qui n’a de sens que dans l’espace d’adressage que l’on simule et qui n’est absolument pas relié d’une quelconque manière à l’espace mémoire “réelle” du programme en C qui constitue le simulateur. Par la suite, j’utiliserais le terme “EAS” pour préciser d’une adresse qu’elle est en “Espace d’Adressage Simulé” et non en mémoire réelle de l’ordinateur.

Le but est de réaliser 3 fonctions :

- **zin_malloc**, qui alloue virtuellement un nombre donné d’octets pour une application donnée. **zin_malloc** recherche le prochain “trou” dans la mémoire en EAS pouvant accueillir le nombre d’octets demandé et renvoie 0 (ou plutôt une constante **ZIN_NULL** que l’on définira à zéro) si il n’y a pas assez de place pour le faire. Dans le cas contraire, **zin_malloc** mémorise la demande en ajoutant un maillon dans la liste chaînée et en renvoyant l’adresse EAS du début du bloc alloué.

- **zin_free**, qui libère un bloc de mémoire alloué par l'application. Si l'adresse du bloc donné en EAS est invalide ou n'a pas été réservée par l'application, **zin_free** doit arrêter la simulation en affichant "*Segmentation fault*". Par ailleurs, la tentative de libération de blocs à l'adresse EAS **ZIN_NULL** doit être permise (rien n'est effectué, mais le programme n'est pas stoppé.)
- **zin_checkaddr**, qui vérifie qu'une application a bien le droit de lire ou d'écrire une zone mémoire en EAS. Le cas échéant, elle doit également arrêter la simulation en affichant "*Segmentation fault*" ou "*Access violation*" selon les cas.

1.2 Détails d'implémentation

1.2.1 Protection mémoire

Comme tout système moderne, **Zindo** interdit qu'une application X puisse lire ou écrire les données d'une application Y. Par conséquent, on dit que chaque application est identifiée par un nombre entier unique, et que chaque bloc de mémoire allouée en EAS mémorise cet identifiant. Ainsi, **zin_free** et **zin_checkmemory** doivent vérifier que l'application qui demande la modification du bloc l'a bien créée, car il serait incohérent d'autoriser un quelconque accès à un processus qui ne les "connait" pas.

1.2.2 Types introduits

Trois nouveaux types (correspondant au type C "**unsigned int**") sont introduits pour désigner une adresse en EAS, une taille de bloc et l'entier unique représentant l'application (respectivement **zin_addr_t**, **zin_size_t** et **zin_pid_t**) grâce aux lignes suivantes :

```
typedef unsigned int zin_addr_t;
typedef unsigned int zin_size_t;
typedef unsigned int zin_pid_t;
```

1.2.3 Structure descriptrice

Un descripteur est donc représenté par la structure suivante :

```
typedef struct _mem_descriptor_t
{
    zin_addr_t addr;
    zin_size_t size;
    zin_pid_t pid;
    struct _mem_descriptor_t *next;
} mem_descriptor_t;
```

Autrement dit, un descripteur contient 4 informations :

1. L'adresse de début du bloc en EAS,
2. La taille du bloc réservé,
3. Le numéro de l'application qui a réservé le bloc,
4. Un pointeur vers le bloc suivant, s'il existe.

En mémoire, la liste chaînée décrit les blocs alloués dans l'EAS triés par ordre croissant d'adresse de début. Par exemple, si trois blocs b_1, b_2, b_3 de 10 octets chacun ont été alloués aux adresses 100, 250 et 1067, les descripteurs seront chaînés dans l'ordre "naturel", c'est à dire b_1, b_2, b_3 .

1.2.4 Variable d'accès

La liste chaînée doit être stockée dans une variable globale au programme, déclarée comme ceci :

```
mem_descriptor_t* memory = NULL;
```

Ainsi, au début du programme, **memory** est initialisé à **NULL** : la liste ne contient aucun élément. Aucun descripteur n'ayant été alloué, l'EAS est considéré comme entièrement libre.

1.2.5 Définition de l'espace d'adressage

On définit les constantes suivantes pour décrire l'espace d'adressage :

```
#define ZIN_NULL 0
#define ZIN_MEMSPACE 10000
#define ZIN_RESERVED 64
```

- La première constante, **ZIN_NULL**, définit une adresse qui sera toujours détecté comme invalide par le système, tout comme **NULL** l'est en C.
- La seconde constante, **ZIN_MEMSPACE**, définit la taille de l'espace d'adressage virtuel de **Zindoz**. Par défaut, ici, la plage des adresses manipulables va donc de 0 à 9999,
- La troisième constante, **ZIN_RESERVED**, définit un espace mémoire inaccessible à tous les processus destiné à déboguer les accès aux pointeurs initialisés à **NULL** (ou plutôt **ZIN_NULL**). En effet, si un pointeur p initialisé à **ZIN_NULL** est accédé, on sait que le processus a tenté de lire ou d'écrire une adresse invalide : le processeur sur lequel tourne **Zindoz** déclenche une *exception matérielle* qui est interceptée sous la forme du message "*Access violation*". Ce message s'affichant, on peut supposer sans erreur que l'erreur vient de la manipulation d'un pointeur initialisé à **ZIN_NULL**, contrairement à "*Segmentation Fault*" qui désigne simplement l'accès à une zone mémoire interdite. Le problème est que, si

dans une boucle on accède à $p + 1$ (p toujours non initialisé), l'adresse accédée ne sera plus `ZIN_NULL` mais `ZIN_NULL + 1`... Dans un monde idéal, le système “comprendrait” que l'on tente de manipuler un pointeur initialisé à 0 et afficherait également “*Access violation*”. Dans le monde réel, on définit un espace entre 1 à `ZIN_RESERVED-1` qui représente une plage d'adresses interdites gérant l'accès à des adresses non initialisées avec un décalage.

1.2.6 Routines préprogrammées

Il n'y a pas besoin d'autres bibliothèques que la bibliothèque standard (`<stdio.h>`) et de `<malloc.h>`. Les fonctions suivantes pourront être utilisées pour allouer et désallouer un descripteur :

```
#define ERROR_ALLOCDSCRFAILED 1
mem_descriptor_t* alloc_descriptor(zin_addr_t addr,
    zin_size_t size, zin_pid_t pid, mem_descriptor_t *next)
{
    mem_descriptor_t* p = (mem_descriptor_t*) malloc(sizeof(mem_descriptor_t));
    if (p) /** Allocation réussie ?... **/
    {
        /** On remplit les propriétés du descripteur : **/
        p->addr = addr;
        p->size = size;
        p->pid = pid;
        p->next = next;
        return p;
    } else exit(ERROR_ALLOCDSCRFAILED); /** Erreur : on arrête tout. **/
}

void free_descriptor(mem_descriptor_t* p)
{
    free(p); /** On libère le bloc en mémoire **/
}
```

2 Implémentation

2.1 Implémentation de `zin__malloc`

2.1.1 Prototypage

Le prototype de `zin__malloc` doit être le suivant :

```
zin_addr_t zin_malloc(zin_size_t size, zin_pid_t pid)
```

`zin__malloc()` simule l'allocation de *size* octets en EAS pour l'application *pid*. Si l'allocation réussit, la valeur renvoyée est l'adresse de départ du bloc en EAS. Sinon, c'est `ZIN_NULL` qui est retourné.

2.1.2 Stratégie

Si l'EAS est vide et que la taille du bloc entre dans l'espace mémoire disponible, on peut allouer le bloc, et dans ce cas là, l'EAS est décrit uniquement par ce bloc créé. Sinon, on recherche dans la liste chaînée le premier "trou" assez grand pour contenir "size" en comparant deux à deux les descripteurs (si p et q sont deux pointeurs vers des `mem_descriptor_t` désignant deux blocs mémoire qui se suivent, alors l'espace disponible entre les deux blocs se calcule par $q->addr - (p->addr + p->size)$). Si on trouve un trou, on crée et on initialise un descripteur avec la fonction intégrée `alloc_descriptor()` que l'on chaîne *entre* les deux blocs.

2.2 Implémentation de `zin_free`

2.2.1 Prototypage

Le prototype de `zin_free` doit être le suivant :

```
void zin_free(zin_addr_t addr, zin_pid_t pid)
```

`zin_free()` simule la libération d'un bloc d'adresse de début en EAS $addr$ alloué par l'application pid . Si $addr$ est égal à `ZIN_NULL`, `zin_free` ne fait rien. Le cas contraire, si un bloc correspondant réservé par pid est trouvé, le bloc est libéré. Sinon, si $addr$ est dans l'espace réservé (`ZIN_RESERVED`), le programme affiche "*Access violation*" et quitte, sinon le programme affiche "*Segmentation fault*" et quitte.

2.2.2 Stratégie

Il s'agit ici d'une recherche et d'une suppression dans une liste chaînée. Pour chaque descripteur, on teste si son adresse de début et son identificateur d'application sont les mêmes que ceux passés en argument de `zin_free`. Le cas échéant, on libère le bloc avec `free_descriptor()` en reconstruisant le lien entre les deux descripteurs adjacents, sinon, selon la valeur de $addr$, on affiche un des deux messages cités ci-dessus.

2.3 Implémentation de `zin_checkaddr`

2.3.1 Prototypage

Le prototype de `zin_checkaddr` doit être le suivant :

```
void zin_checkaddr(zin_addr_t addr, zin_pid_t pid)
```

`zin_checkaddr` teste si une adresse donnée (qui ne correspond pas forcément à une adresse de début de bloc !) est dans un bloc réservé par pid . Sinon, si $addr$ est dans l'espace réservé (`ZIN_RESERVED`), le programme affiche "*Access violation*" et quitte, sinon le programme affiche "*Segmentation fault*" et quitte.

2.3.2 Stratégie

Un simple parcours dans la liste suffit à vérifier que *addr* est compris pour chaque descripteur chaîné *p* entre **p->addr** et **p->addr + p->size**. Si rien n'est trouvé, selon la valeur de *addr*, on affiche un des deux messages cités ci-dessus.

2.4 Exo bonus

Coder :

```
void zin_cleanup(zin_pid_t pid)
```

...qui libère tous les blocs de mémoire alloués par l'application *pid*. On suppose que cette fonction sera appelée par **Zindo** une fois qu'un programme se sera achevé afin de supprimer tous les espaces mémoire que le mauvais programmeur aura oublié de libérer :-)