Systèmes d'exploitation — Exercices
Mémoire

Seweryn Dynerowicz (`Seweryn.DYNEROWICZ@umons.ac.be`)

1 Organisation de la mémoire

Un processeur x86 transforme chaque adresse *logique* (*i.e.* numéro de segment + *offset*) manipulée par un processus en une adresse *physique* (*i.e.* envoyée sur le bus mémoire) par le biais de deux unités spécialisées [1] travaillant l'une à la suite de l'autre. L'unité de segmentation réalise la translation de l'adresse *logique* en une adresse *linéaire* en utilisant les données de la table des segments, tandis que l'unité de pagination effectue la traduction de l'adresse *linéaire* en une adresse *physique* en utilisant les données de la table des pages.

Sous Linux, le choix de conception est de se passer de la segmentation ; la *protection* et la *projection* reposent dès lors exclusivement sur la pagination. La compréhension de l'agencement de la mémoire se focalisera dès lors sur l'organisation de l'espace *virtuel* du processus (*i.e.* ensemble des adresses *linéaires*).

Pour une architecture 64 *bits*, il serait possible théoriquement de couvrir un espace de 16 EiB (*i.e. exbibytes*) soit l'équivalent de 18.446.744.073.709.551.616 octets individuellement adressables. En pratique [2], le processeur se limite à utiliser des adresses *linéaires* de 48 *bits* [3], ce qui représente 256 TiB (*i.e. tebibytes*) de mémoire virtuelle. Dans ce cas de figure, un processus occupe les 128 TiB inférieurs, tandis que le noyau est réparti sur les 128 TiB supérieurs. Dans la figure ci-dessous, la répartition de la mémoire virtuelle attribuée au processus est présentée. Les adresses listées à droite de la figure mettent en évidence le numéro de la première et la dernière page de 4 KiB appartenant à chaque région.

Au sommet se trouve l'espace réservé à la *stack* qui est utilisée au fil de l'exécution du processus et qui grandit vers le bas. Dans le cas présent, cette région a une taille initiale de 132 KiB.

Vers le milieu se trouve la région utilisée pour rendre accessible les zones de mémoire partagées avec d'autre processus, les *mappings* de fichiers en mémoire ainsi que les librairies partagées.

L'espace réservé à la *heap* est utilisé pour accueillir tous les objets alloués dynamiquement (*cfr.* Section 3). Cet espace est organisé de manière à savoir, à tout moment, quelles portions sont libres et quelles portions sont occupées. La limite supérieure de l'espace utilisé par la *heap* à un moment donné est appelé le `program break`. Lors d'une demande d'allocation, s'il existe une zone libre en dessous de cette limite, elle peut être utilisée. Dans le cas contraire, il est nécessaire de demander à l'OS de faire monter la limite du `program break` pour créer un espace suffisant.

Directement en dessous de la *heap* se trouve le code machine (*i.e.* `text`). Les variables associées à ce programme peuvent également résider dans cette région ou bien occuper de la place à la base de la `stack`.

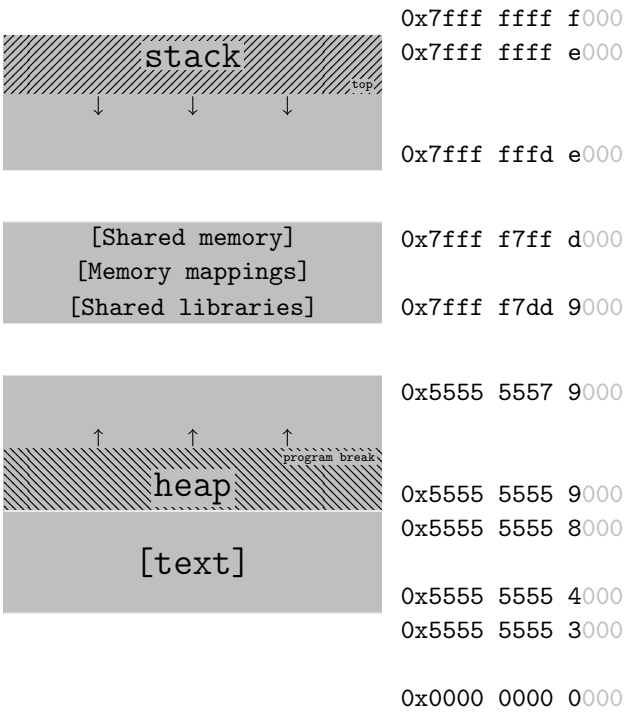

FIGURE 1 – Organisation des premiers 128 TiB de mémoire virtuelle. Les adresses listées à droite mettent en évidence le numéro de la première et de la dernière page de chaque région.

Les régions "vides" sur la figure représente les portions qui n'ont pas d'utilité pour le processus. Le système d'exploitation peut utiliser des *guard pages* pour surveiller les accès mémoire par le processus. Par exemple, cette technique permet de détecter la survenance d'une *stack overflow* ou d'un *stack underflow*.

1. *cfr.* Intel 64 and IA-32 Architectures — System Programming Guide, Chapitre 3, Section 3.1.
2. *cfr.* Intel 64 and IA-32 Architectures — System Programming Guide, Chapitre 4, Section 4.5.
3. Ou des adresses *linéaires* de 57 *bits* (*cfr.* `https://www.kernel.org/doc/Documentation/x86/x86_64/5level-paging.txt`).

2 Pointeurs

Afin de pouvoir manipuler explicitement des adresses en vue de réaliser des accès mémoire, la notion de *pointeur* est utilisée. Le *listing* ci-dessous reprend les concepts essentiels de l'arithmétique des pointeurs [4] que nous décrivons plus bas.

```
1  #include <stdio.h>  // printf(..)
2  #include <stdlib.h> // EXIT_SUCCESS, malloc(..), calloc(..), NULL
3
4  int main(int argc, char* argv[]) {
5    unsigned int  counter = 42;
6    unsigned int* pointer = NULL;
7
8    printf("Value of counter : %u\n", counter);
9    printf("Value of pointer : %p\n", pointer);
10   if(pointer != NULL)
11     printf("Value at pointer : %u\n", *pointer);
12
13   pointer = &counter;
14   (*pointer)++;
15
16   printf("Value of counter : %u\n", counter);
17   printf("Value of pointer : %p\n", pointer);
18   if(pointer != NULL)
19     printf("Value at pointer : %u\n", *pointer);
20
21   return EXIT_SUCCESS;
22 }
```

FIGURE 2 – Manipulations de pointeurs

Pour tous les types qui peuvent être utilisés en C, il est possible de définir un pointeur vers une variable de ce type en le préfixant d'une astérisque. Par exemple, la variable `counter` (ligne 5) est de type entier non-signé tandis que la variable `pointer` (ligne 6) est un pointeur vers un entier non-signé [5]. Pour afficher l'adresse stockée dans un pointeur, le *token* d'affichage pour la fonction `printf(..)` est `%p` (*cfr.* ligne 9). Afin de pouvoir jongler entre pointeur et pointé, deux opérateurs unaires [6] sont disponibles.

L'opérateur de *référencement* (*i.e.* `&`) permet d'obtenir l'adresse d'une variable quelconque pour ensuite travailler dessus, par exemple après l'avoir stockée dans un pointeur. La ligne 12 illustre comment l'adresse de la variable `counter` est récupérée pour être stockée dans le pointeur `pointer`.

L'opérateur de *déréférencement* (*i.e.* `*`) permet d'obtenir la valeur vers laquelle une certaine adresse pointe [7]. La ligne 13 illustre comment il est possible de manipuler directement le contenu stocké à l'adresse où réside la variable `counter` en appliquant cet opérateur sur le pointeur `pointer`.

Dans le cas de figure où le pointeur vers une structure (*e.g.* `struct vector* v`), les accès aux membres de la structure peuvent être décrit de manière plus concise à l'aide d'un opérateur de flèche. Il est préférable d'écrire `v->x` au lieu de `(*v).x` afin d'accroître la lisibilité.

Pour terminer, il existe une relation d'équivalence entre les notions de tableaux et de pointeurs en C ; les deux sont interchangeables. À titre d'illustration, le *listing* ci-dessous contient une version de la fonction `strlen(..)` reposant uniquement sur des pointeurs.

```
1  unsigned int strlen(char* string) {
2    for( char* current = string; *current != '\0'; current++ ) {}
3    return current - string;
4  }
```

FIGURE 3 – Fonction `strlen(..)` avec pointeurs.

4. Ensemble des règles de calculs qui sont utilisées pour les opérations sur des pointeurs
5. La valeur `NULL` est utilisée pour dénoter une adresse invalide
6. Opérateur qui ne prend qu'une seule opérande
7. **Attention :** le *déréférencement* d'un pointeur `NULL` provoque une erreur de segmentation (signal `SIGSEGV`)

3 Allocation et libération dynamique

Un processus peut gérer la mémoire dont il dispose à un moment donné en utilisant les mécanismes d'allocation et de libération. Pour ce qui est de l'allocation, les deux fonctions disponibles diffèrent au niveau de leurs paramètres et dans une moindre mesure par leur comportement.

La fonction d'allocation `malloc(..)` prend en paramètre le nombre d'octets qui sont demandés et renvoie comme valeur de retour l'adresse du début de l'espace alloué, ou une valeur `NULL` dans le cas où l'allocation n'a pas pu être accomplie.

La fonction d'allocation `calloc(..)` prend en paramètre un nombre d'éléments ainsi que la taille en octets d'un de ces éléments et renvoie comme valeur de retour l'adresse du début d'un espace alloué suffisant pour accueillir ces éléments (valeur `NULL` en cas d'échec). Par contraste avec `malloc(..)` qui fourni l'espace mémoire "en l'état", la fonction `calloc(..)` initialise l'espace mémoire avec des valeurs `0`.

À chaque opération d'allocation doit correspondre une opération de libération, en passant à la fonction `free(..)` le pointeur obtenu, pour signaler au système d'exploitation que la mémoire en question n'est plus utilisée. Bien que cette correspondance soit facile à garantir au sein d'une même fonction (allocations au début, libérations avant le retour), il est habituel qu'une fonction réalise une allocation pour ensuite donner la responsabilité à l'appelant de libérer l'espace obtenu. On parle de fuite mémoire (*memory leak*) lorsque un programme contient une allocation dynamique à laquelle aucune libération ne correspond. Bien que la mémoire soit libérée à la terminaison d'un processus, certains processus ne sont pas censés se terminer (*e.g.* serveur Web) et sont de ce fait beaucoup plus sensibles à ce type de problèmes [8].

```
1  #include <stdio.h>  // printf(..)
2  #include <stdlib.h> // EXIT_SUCCESS, malloc(..), calloc(..)
3
4  #define SIZE 1024
5
6  int main(int argc, char* argv[]) {
7    unsigned int* count = malloc(4);
8    unsigned int* array = malloc( SIZE * sizeof(unsigned int) );
9
10   *count = 0;
11   unsigned int* final = &array[SIZE-1];
12   for(unsigned int* current = array; current <= final ; current++) {
13     if (*current == 0)
14       (*count)++;
15   }
16   printf("Number of zeroes in array : %u\n", *count);
17
18   free(count);
19   free(array);
20
21   return EXIT_SUCCESS;
22 }
```

FIGURE 4 – Utilisation basique de `malloc` et `free`

Le programme ci-dessus alloue dynamiquement de l'espace pour une variable de type entier non-signé dont l'adresse est stockée dans le pointeur `count` (ligne 7) ainsi que pour un tableau contenant `SIZE` éléments de type entier non-signé (ligne 8). Le but du programme est de compter le nombre de ces entiers dont la valeur est égale à zéro dans l'espace mémoire qui a été alloué.

Pour ce faire, le compte est initialisé à 0 (ligne 10) et l'adresse du dernier élément est calculée et stockée dans le pointeur `final` (ligne 11). La boucle principale (ligne 12) parcourt chaque élément du tableau par le biais du pointeur `current`. La condition consiste à vérifier que ce pointeur contient une adresse inférieure ou égale à celle du dernier élément. En vertu des règles de l'arithmétique des pointeurs, l'opération d'incrémentation, `current++`, avance l'adresse stockée dans ce pointeur de 4 octets [9]. À chaque fois que la valeur vers laquelle pointe `current` est égale à zéro, `count` est incrémenté d'une unité.

8. Il existe des outils capables de détecter les fuites mémoires (*cfr.* Valgrind ; `https://valgrind.org/`)
9. Pour rappel, un entier non-signé est stocké sur 4 octets.

4 Énoncés

Exercice 1

Quelle est la nature de l'adresse renvoyée par les appels de la famille `malloc` ; linéaire ou physique ? Implémentez un programme pour tester votre réponse. *Astuce :* Utilisez la fonction `getchar` pour bloquer un processus qui sera dès lors en attente d'un caractère saisi au clavier. **Attention :** pour éviter toute confusion [10], vous devez exécuter votre programme avec la commande `setarch x86_64 -R ./a.out`

Exercice 2

Implémentez une fonction qui remplace chaque caractère de tabulation (*i.e.* `'\t'`) par une suite de 4 caractères d'espace (*i.e.* `' '`) dans une chaîne de caractères (*null-terminated*). Utilisez une allocation dynamique pour produire l'emplacement où résidera la chaîne modifiée.

```
char* expand_tabs(char* string);
```

Exercice 3

Implémentez la gestion des opérations d'additions vectorielles et de multiplications scalaires pour des vecteurs à n dimensions. Ces opérations doivent produire un nouveau vecteur sans modifier ceux passés en paramètres. Utilisez les déclarations suivantes. En cas d'incompatibilité, la fonction `add` renverra un pointeur `NULL`.

```
struct vector* new(unsigned int n);
struct vector* add(struct vector* v, struct vector* w);
struct vector* smul(double s, struct vector* v);
```

Exercice 4

Implémentez la gestion des opérations de multiplications matricielles pour des matrices de dimensions $m \times n$. Ces opérations doivent produire une nouvelle matrice sans modifier celles passées en paramètres. Utilisez les déclarations suivantes. En cas d'incompatibilité, la fonction `mul` renverra un pointeur `NULL`.

```
struct matrix* new(unsigned int m, unsigned int n);
struct matrix* mul(struct matrix* m1, struct matrix* m2);
```

Exercice 5

Implémentez la gestion de listes chaînées en mode LIFO. Respectez les déclarations suivantes.

```
struct lifo* mklifo();
void push(struct lifo* lst, signed int elt);
signed int pop(struct lifo* lst);
```

Exercice 6

Implémentez la gestion de listes chaînées en mode FIFO. Respectez les déclarations suivantes.

```
struct fifo* mkfifo();
void enqueue(struct fifo* lst, signed int elt);
signed int dequeue(struct fifo* lst);
```

Travail de groupe

Implémentez la gestion de listes chaînées circulaires en mode FIFO. Les fonctions `rotateToEven` et `rotateToOdd` appliquent une "rotation" sur une liste circulaire en avançant au moins d'une position tous les éléments jusqu'à tomber respectivement sur un élément pair ou impair. Respectez les déclarations suivantes.

```
struct circular* mkcircular();
void insert(struct circular* cycle, signed int elt);
signed int extract(struct circular* cycle);
struct node* rotateToEven(struct circular* cycle);
struct node* rotateToOdd(struct circular* cycle);
```

10. *cfr.* ASLR : `https://en.wikipedia.org/wiki/Address_space_layout_randomization`