UNIVERSITÉ LIBRE DE BRUXELLES, UNIVERSITÉ D'EUROPE

Faculté des Sciences — Département d'Informatique

ULB

INFO-F-201 – Systèmes d'exploitation

Gestion de la mémoire

Joël Goossens (N8.107) Olivier Markowitch (N8.106)
Arnaud Leponce (N8.213) Yannick Molinghen (N8.210)
Alexis Reynouard (N8.215)

1 Segments

segments

Un programme compilé contient plusieurs ***segments*** :

- `text` qui contient le code du programme
- `data` qui contient les données initialisées
- `bss` qui contient les données non initialisées

On peut consulter la taille des segments d'un programme avec la commande `size`, comme présenté dans le Listing 1.

```
$ size programme
text    data     bss     dec     hex filename
2866     656      16    3538     dd2 programme
```

Listing 1 – Utilisation de `size`

Exercice 1. *A quoi correspondent les colonnes* `dec` *et* `hex` *de la commande* `size` *? Comment le vérifier ?*

Réponse : Ces colonnes indiquent la taille totale de tous les segments (leur somme) en décimal et en hexadécimal. On peut le vérifier en consultant le manuel avec `man size`.

Exercice 2. *Écrivez un programme en C dont la taille du segment* `data` *fait exactement 3000 bytes.*

Réponse : Il suffit d'écrire un programme où on déclare (et initialise) une variable globale de type `char[]` de grande taille (± 2400). La taille est inférieure à 3000 car il existe des variables globales par défaut (stdout, ...) qui sont ajoutées en plus du programme que l'on écrit.

Exercice 3. *Le code du Listing 2 initialise une variable* `code_super_secret`*. Modifiez le programme de sorte à afficher le contenu de cette variable sans jamais l'utiliser (après son initialisation).*

```c
int main() {
    char code_super_secret[] = "le code secret";
}
```

Listing 2 – Exercice de hacker

Exercice 4. *A la lumière de cette section, expliquez ce que signifie l'erreur "segmentation fault" et comment elle survient.*

2 Les assainisseurs

Un assainisseur (**sanitizer**) est un outil qui va permettre de détecter des erreurs durant l'exécution. Dans ce qui suit, on intéressera aux célèbres assainisseurs de Google [1], souvent intégrés par défaut dans les compilateurs de nos jours :

sanitizer

— AdresseSanitizer (ASan) : détection d'accès en mémoire invalides et fuites de mémoire (option `-fsanitize=address`) ;
— ThreadSanitizer (TSan) : détection des accès concurrents et interblocages (option `-fsanitize=thread`) ;
— MemorySanitizer (MSan) : détection d'accès en mémoire non initialisés (option `-fsanitize=memory`) ;
— UndefinedSanitizer (UBSan) : détecter des comportement indéterminés (option `-fsanitize=undefined`).

Pour utiliser un assainisseur, il suffit d'ajouter son option à la liste des options de compilation (celles-ci sont valides pour gcc, g++ et Clang). Il est important de noter que tout le code doit être recompilé lorsque l'option est ajoutée ou retirée. Afin d'avoir un affichage plus commode des erreurs détectées, il est conseillé d'utiliser aussi l'option `-g` (ajout des symboles de débogage) et, pour les performances, d'ajouter `-O2` (optimisation de niveau 2).

Notez que les assainisseurs font leur analyse dynamiquement, c'est-à-dire au cours de l'exécution. Ils détectent à l'exécution des erreurs que le compilateur n'aurait pas pu anticiper. Par conséquent, ils ne détectent que les erreurs qui se sont produites lors de l'exécution de votre programme (tout comme les débogueurs ou les outils comme valgrind). Une erreur qui n'est pas rencontrée durant l'exécution (par exemple dans un `if` dont la condition était fausse) ne sera pas détectée par ces méthodes.

Exercice 5. *Compilez les codes 3 (`overflow.c`), 4 (`use_after_free.c`) et 5 (`memleak.c`) avec les options `-fsanitize=address -g`. Lancez le programme puis analysez le message en sortie et essayez d'y retrouver où l'erreur s'est produite.*

```c
#include <stdlib.h>
int main(void) {
    char *p = malloc(16);
    p[24] = 1;
    free(p);
    return 0;
}
```

Listing 3 – Exemple d'un dépassement de tampon.

```c
#include <stdlib.h>
void foo(char* p) {
    int i = 0;
    *p = i + 'x';
```

1. https://github.com/google/sanitizers

```
}

int main(void) {
    char *p = malloc(16);
    p[0] = 1;
    free(p);
    foo(p);
    return 0;
}
```

Listing 4 – Exemple d'utilisation après un free().

```
#include <stdlib.h>
int main(void) {
    char* p = malloc(20);
    return 0;
}
```

Listing 5 – Exemple fuite de mémoire.

Exercice 6. *Compilez le code 6 (`return_local_pointer.c`) avec les options `-fsanitize=address -g`. Lancez le programme en précisant `ASAN_OPTIONS=detect_stack_use_after_return=` avant (ex. : `ASAN_OPTIONS=detect_stack_use_after_return=1 ./prog`) puis analysez le message en sortie et essayez d'y retrouver où l'erreur s'est produite.*

```
int *f() {
    int i = 42;
    int *p = &i;
    return p;
}
int g(int *p) {
    return *p;
}
int main() {
    return g(f());
}
```

Listing 6 – Exemple d'utilisation d'une variable locale après sa destruction.

Exercice 7. *Compilez le code 7 (`datarace.c`) avec les options `-fsanitize=thread -g`. Lancez le programme puis analysez le message en sortie et essayez d'y retrouver où l'erreur s'est produite.*

```
#include <pthread.h>
#include <stdbool.h>

static volatile bool flip1 = false;
static volatile bool flip2 = false;

void* th_flip(void* p) {
    (void)p;
    while (!flip1) {}
    flip2 = true;
    return NULL;
}

int main() {
    pthread_t th;
```

```
    pthread_create(&th, NULL, th_flip, NULL);
    flip1 = true;
    while (!flip2) {}
    pthread_join(th, NULL);
    return 0;
}
```

Listing 7 – Exemple d'accès concurrents.

Le langage Rust

Profitons-en pour faire un petit détour par le langage Rust qui a la propriété d'être sécurisé en terme de mémoire (memory safe) et en terme de concurrence (thread safe) par défaut, et ce à la compilation, à la différence des assainisseurs qui font les vérifications à l'exécution.

garbage collector

ownership

À la différence de l'approche basée sur un **garbage collector** comme Python, le comptage de références comme les pointeurs partagés en C++ ou l'allocation et la libération manuelle de mémoire comme les langages C et C++, Rust utilise le principe de propriété (**ownership**, comme les `unique_ptr` en C++) pour détecter la durée de vie des variables.

Le concept d'ownership respecte trois règles [2] :

1. Chaque variable a un propriétaire
2. Il ne peut jamais y avoir qu'un propriétaire à la fois → pas de double `free` possible
3. Quand le propriétaire sort de sa portée (**scope**), la valeur est libérée → pas de fuites de mémoire

scope

```
{ // Début de scope
    let s = "salut"; // Début de la durée de vie de 's'
    emprunte(&s); // La fonction "emprunte" 's' avec '&'
} // Fin de la durée de vie de 's', 's' est libérée
```

Listing 8 – Emprunt de variable

```
let s = "salut"; // Début de la durée de vie de 's'
une_fonction(s); // 'une_fonction' devient propriétaire
//'s' est libérée à la fin de 'une_fonction'
```

Listing 9 – Changement de propriétaire

De la même manière, Rust est capable de détecter certains problèmes liés aux threads à la compilation [3] à l'aide de son système d'ownership et d'un système fortement typé.

Dans le Listing 10, Rust fait en sorte de transférer explicitement la propriété au thread à l'aide de l'instruction `move`. C'est donc le thread qui est maintenant responsable de la durée de vie de `x`, et toute utilisation de la variable en dehors du thread est prohibée.

```
let data = String::from("un test");
std::thread::spawn(move || {
    println!("data={data}");
});
// println!("{data}"); // Ceci est une erreur: la variable a été donnée au thread.
```

Listing 10 – Utilisation de `move` dans le cadre d'une `closure`

2. Source : https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html
3. Source : https://doc.rust-lang.org/book/ch16-00-concurrency.html

Pour partager une variable entre des threads, Rust impose que la variable soit thread-safe en vérifiant qu'elle implémente bien certains traits, par exemple en l'englobant dans une `Mutex`. Ainsi, il n'est donc plus possible d'avoir des accès concurrents.

Sans entrer dans les détails, un exemple de partage de variable est présenté dans le Listing 11.

```
let counter = Arc::new(Mutex::new(0));
let cloned_counter = Arc::clone(&counter);
std::thread::spawn(move || {
    let mut num = cloned_counter.lock().unwrap();
    *num += 1;
}).join();
```

Listing 11 – Partage de variable entre plusieurs threads

3 Gestion de la mémoire physique

Au cours du temps, l'OS doit garder une trace des zones de la RAM libres et des zones occupées. Il existe plusieurs manières de faire telles que les tables de bits (**bitmap**), les listes chaînées ou encore les tables de listes chaînées.

bitmap

3.1 Table de bits (bitmap)

Dans cette représentation, la mémoire est divisée en blocs de taille identique (par exemple 4Kio) et chaque entrée dans la table des bits indique si la zone mémoire correspondante est occupée ou non.

Exercice 8. *Considérons un système dont la mémoire physique fait 32Kio et une table de bits qui a 64 entrées. Quelle est la taille d'un bloc ?*

Réponse : Pour 32Kio, La taille d'un bloc est $\frac{32\text{Kio}}{64} = 512$ octets.

3.2 Liste chaînée

Dans cette représentation, la liste chaînée contient des noeuds indiquant si le début et la fin de la portion de mémoire ainsi que si celle-ci est libre ou occupée.

Exercice 9. *Décrivez les blocs de la liste chaînée correspondant à la mémoire représentée dans le tableau 1 en donnant pour chaque bloc*

- *Si la zone mémoire correspondante est libre ou occupée*
- *L'adresse de début de la zone mémoire*
- *La taille de la zone de mémoire*

Occupé			Libre	
0 5	7	12	32	35

Table 1 – Représentation de la mémoire