Organisation mémoire

Pierre Karpman

Lycée Champollion MP2I
https://membres-ljk.imag.fr/Pierre.Karpman/CPGE/2025/

Roadmap

Poly, Section 7

Table des matières

Espace mémoire virtuel

Principe

- Les adresses mémoire manipulables par les programmes (par ex. *via* des pointeurs) sont « virtuelles »
- Elles ne correspondent pas directement à une adresse *physique* (par ex. un emplacement dans une barrette RAM)
- Le système d'exploitation est responsable de la traduction virtuel $\rightarrow$ physique mais délègue une partie du travail au CPU
  - Et vérifie la légalité des accès (*de son point de vue*)

Intérêt

- Isolation des espaces mémoire des différents processus (sauf si demandé expressément, *via* des mécanismes dédiés)
- Essentiel pour la sécurité, la sûreté

Cartographie

Dessin, démo

Faute d'accès : adresse inconnue

Ce n'est pas parce qu'on peut nommer une adresse qu'il s'y trouve quelque chose

Dessin, démo

```c
#include <stdlib.h>
int main(void) {
    int *p = (int *)12;
    *p = 1;

    return EXIT_SUCCESS;
}
```

Faute d'accès : permission refusée

Ce n'est pas parce qu'une adresse existe qu'on peut y habiter

Dessin, démo

```c
#include <stdlib.h>
const int a = 12;
int main(void) {
    *((int *)&a) = 11;

    return EXIT_SUCCESS;
}
```

Corruption mémoire en principe

En pratique, le système peut donner accès à des adresses physiques qui ne correspondent à aucun objet ; les manipuler est illégal

Dessin, démo

```c
#include <stdlib.h>
int main(void) {
    int *p = malloc(4*sizeof(int));
    p[4] = 12;

    return EXIT_SUCCESS;
}
```

Corruption mémoire en action

Le système d'exploitation ne peut rien pour vous si vous faites n'importe quoi (mais le langage dira que c'est UB)

Dessin, démo

```c
#include <stdlib.h>
#include <stdio.h>
int main(void) {
    int *p = malloc(4*sizeof(int));
    int *q = malloc(4*sizeof(int));
    q[0] = 12;
    p[8] = 11;
    printf("%d\n", q[0]);

    return EXIT_SUCCESS;
}
```

Table des matières

Préambule : contexte d'exécution d'un appel de fonction

Lors d'un appel de fonction, il faut de la place pour stocker

- Les arguments
- Les variables locales
- Qui nous a appelé (et d'où) ?
  - Pour savoir quoi faire après que la fonction retourne
  - Quelle est la suite du flot de contrôle ?

Contexte / *stack frame*

Toutes (ou pas…) ces informations (et d'autres) sont stockées dans un *contexte* d'appel/d'exécution ou *stack frame*

Comment stocker les contextes ?

Préambule : durée de vie des contextes

Contraintes usuelles

- Dans la plupart des langages, appeler une fonction depuis une autre *suspend* l'exécution de la seconde, mais *ne la termine pas*
- Tant qu'une fonction n'a pas terminé, il faut garder son contexte en mémoire
  - Pour pouvoir reprendre son exécution
- Quand une fonction termine, on peut généralement oublier son contexte
  - plus d'exécution à reprendre
  - les objets locaux à la fonction ont généralement cessé d'exister (en C : puisque de durée de vie automatique)

Résumé

À tout moment, il est nécessaire & suffisant de garder en mémoire les contextes des *appels* de fonction qui n'ont pas encore terminé

En C ça peut être plus compliqué mais c'est hors programme 

Satisfaire les contraintes : avec une *pile*

Principe d'une pile

- On peut créer une pile (*stack*) vide
- On peut empiler (ajouter) un élément au sommet (en fin) d'une pile
- On peut dépiler (extraire) l'élément au sommet (en fin) d'une pile
- Plus de détails dans quelque temps…

Pile d'exécution

On crée une pile pour stocker les contextes d'appel

- Le contexte de l'appel en cours d'exécution est en sommet de pile
- Quand on appelle une fonction, on empile son contexte
- Quand un appel termine, on dépile son contexte

La pile contient exactement les contextes des appels non terminés

Implémentation pour un environnement d'exécution typique

En *général*

- L'environnement alloue une petite zone de taille fixe au début de l'exécution du programme
  - Taille la plus courante : **8 Mo**
  - Occupe des adresses virtuelles « hautes »
  - (La pile croît *vers le bas*)
- Un *pointeur de pile* indique son sommet
  - Les adresses réellement utilisées sont entre le début et le pointeur de pile

Lors d'un appel de fonction

- On empile les informations nécessaires à l'appel
- On transfert le contrôle à la fonction appelée
  - Qui réserve la place pour tout son contexte, en modifiant simplement le pointeur de pile

Lors d'un retour d'appel

- On dépile tout notre contexte
  - En rétablissant simplement le pointeur de pile à sa valeur d'avant l'appel
- On rend le contrôle à la fonction appelante

Dessin, démo

Débordement de pile

La pile est petite

- Pas la place pour stocker (trop) de variables locales (trop grandes)
- Pas la place pour stocker trop de contextes

Pas des limites du langage, mais de l'environnement d'exécution

Causes typiques de débordement

- Utilisation de tableaux trop grands
- Trop d'appels de fonction
  - Typiquement récursifs

Démo.

Bilan sur la pile

- Logiquement adaptée au stockage d'objets de durée de stockage automatique
  - Solution efficace pour cela
- En *pratique*, limitée aux objets de « petite » taille

Quid du reste ?

Table des matières

Zone du tas

Pour (presque) tout le reste il y a *le tas*

Tas :≈ zone d'adresses virtuelles disponibles à l'allocation

- L'allocation ou libération se fait par des *appels système*
  - Par ex. `brk`, `mmap`, `munmap`… sous UNIX
- Allocation : le système réserve (ou pas…) de la mémoire physique, et l'associe à des adresses virtuelles
  - Pas de limite *a priori* sur la taille
- Libération : on invalide la zone d'adresses virtuelles (ce qui permet de réutiliser la mémoire physique)
  - Jamais automatique (tant que le programme s'exécute)

En C

- Le tas est adapté pour stocker les objets de durée de stockage allouée
  - `malloc`, `free` se chargent de faire les appels systèmes
- Et (concrètement) tout objet de grande taille
  - Même s'il est moralement de durée de stockage automatique

Pagination à la demande

- En pratique, le système d'exploitation peut faire du *surbooking*
  - Concrètement, ce n'est pas parce qu'un appel à `malloc` réussit que l'on a *vraiment* toute la mémoire demandée

Démo.

Table des matières

Format d'exécutable

Quid des objets de durée de stockage *statique* ?

- Si initialisés : on doit connaître leur valeur *dès le tout début de l'exécution*
  - Stockés directement dans l'exécutable (avec le code)
- (Si non initialisés, il suffit d'anticiper la place nécessaire pour plus tard)

Détails déterminés par le *format d'exécutable* (par ex. *ELF*)

Démo ?

Table des matières

Bilan

- Exécution d'un programme : liée à un environnement d'exécution
- Environnement : peut ajouter des contraintes concrètes
  - Par ex. : pas de variables locales trop grandes en C (pas une limite du langage)
- Certains bugs sont détectés par l'environnement
  - Par ex. : accès à une adresse mémoire virtuelle invalide (illégal *à la fois* en C et du point de vue du système)
- Pas toujours facile de distinguer ce qui fait partie du langage / de l'environnement
  - Durée de stockage des objets : notion du langage
  - Pile, tas : notion de l'environnement d'exécution (usuel)
- Mais peu de variabilité dans les environnements en programmation « classique »
  - Par ex. mêmes notions de pile, tas en OCaml