Programmation
TP n° 1 : Mémoire

I) Introduction

Exercice 1 : Différentes zones mémoire

Lorsqu'un programme en C est exécuté, le système d'exploitation lui attribue un espace mémoire virtuel divisé en plusieurs zones, chacune ayant un rôle précis dans le fonctionnement du programme. Le schéma ci-dessous illustre l'organisation typique de cette mémoire.

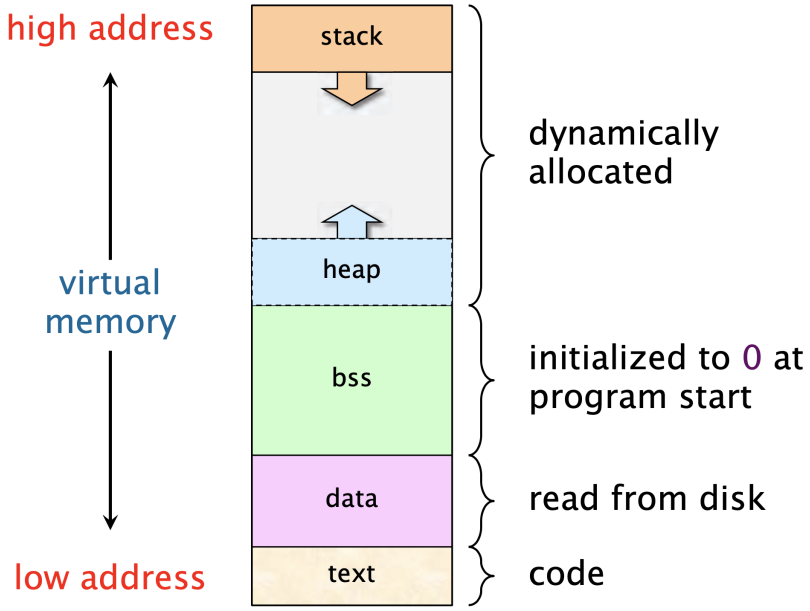

FIGURE 1 – Organisation typique de la mémoire d'un programme en C.

- **Segment `text`** : contient le code exécutable du programme ainsi que celui des bibliothèques utilisées. Cette zone est en lecture et exécution uniquement (`r-x`) afin d'empêcher toute modification du code en mémoire.
- **Segment `data`** : regroupe les variables globales ou statiques *initialisées* (par exemple `int x = 5;`). Elle est chargée depuis le disque lors du démarrage du programme et accessible en lecture/écriture (`rw-`).
- **Segment `bss`** : contient les variables globales ou statiques *non initialisées* (ou initialisées à zéro). Cette zone est automatiquement remplie de zéros par le système au lancement du programme. Elle ne contient donc pas de données réelles dans le fichier exécutable : seules des métadonnées indiquent la taille à réserver en mémoire. Le chargeur du programme alloue ensuite cette zone et la remplit de zéros avant l'exécution. En mémoire, elle possède les mêmes droits que le segment `data` (`rw-`).
- **Tas (`heap`)** : zone d'allocations dynamiques gérée par les fonctions `malloc`, `calloc`, `realloc` et `free`. Sa taille varie pendant l'exécution et elle croît généralement vers les adresses les plus hautes. Les objets y demeurent tant qu'ils ne sont pas libérés explicitement.
- **Pile (`stack`)** : utilisée pour stocker les variables locales, les paramètres de fonction et les adresses de retour. Elle croît vers les adresses les plus basses. Les données y sont libérées automatiquement à la fin des appels de fonction.
- **Zones mappées dynamiquement (`mmap`)** : régions créées par le système pour les bibliothèques partagées, les fichiers mappés en mémoire ou certaines allocations volumineuses. Leurs permissions varient selon l'usage (`r-`, `rw-`, `r-x`, etc.).

L'espace mémoire d'un programme est donc structuré de manière logique : les segments `text`, `data` et `bss` sont statiques, tandis que la pile et le tas sont dynamiques.

1. Allouez de la mémoire dans les différents segments du programme (sauf *mmap*). Affichez l'adresse de chaque variable afin d'observer la disposition des zones mémoire dans l'espace virtuel du processus.
2. Mettez le programme en pause (fonction `pause()`) et relevez son PID avec `getpid()`. Visualisez ensuite la mémoire du processus avec `pmap <PID>` sous Linux ou `vmmap <PID>` sous macOS. Quelles zones observez-vous et quels sont les droits associés à chacune ?

```c
#include <stdio.h>
#include <stdint.h>
#include <string.h>

static const char marker[] = "Hello, .rodata";

int main(void) {
    unsigned char *p1 = (unsigned char*)main;
    unsigned char *p2 = p1;

    char *c = "Hello, tu m'as vu ?";
    printf("Tout commence ici... : %p\n", (void*)p1);

    while (1) {
        if (memcmp(p2, marker, sizeof(marker) - 1) == 0) break;
        putchar((*p2 >= 32 && *p2 <= 126) ? *p2 : '.');
        p2++;
    }

    printf("\n\n Et se termine ici : %p\n", (void*)p2);
    printf("Total : %ld octets\n", p2 - p1);
    return 0;
}
```

3. Essayez de comprendre ce que fait ce programme avant de l'exécuter. Vérifiez ensuite votre hypothèse en compilant et en exécutant le code.

Exercice 2 : Mémoire contiguë

```c
#include <stdio.h>
#include <stdlib.h>

int a[10];
int b[10] = {1};

void test(int n) {
    int c[10];
    static int d[10];
    int *e = malloc(10 * sizeof(int));
    int f[n];

    for (int i = 0; i < 10; i++) {
        c[i] = i;
        d[i] = i;
        e[i] = i;
        f[i] = i;
    }

```

```c
    printf("a : %p\n", (void*)a);
    printf("b : %p\n", (void*)b);
    printf("c : %p\n", (void*)c);
    printf("d : %p\n", (void*)d);
    printf("e : %p\n", (void*)e);
    printf("f : %p\n", (void*)f);

    free(e);
}

int main(void) {
    test(10);
    return 0;
}
```

1. Associez chaque variable à la zone mémoire sur laquelle sera stocké le tableau.
2. Nous allons maintenant nous intéresser aux tableaux dont la taille n'est pas connue à la compilation. Écrivez un programme qui déclare un tableau à taille variable, dont la dimension est demandée à l'utilisateur au moment de l'exécution. Testez différentes entrées et observez le comportement du programme. Quels problèmes pouvez-vous rencontrer et à quoi sont-ils dus ?
3. Écrivez un programme en C qui dump la mémoire du processus et écrit le contenu dans un fichier de sortie. Démarrez la lecture à partir d'une adresse située dans le segment `.text` (par exemple l'adresse d'une fonction) et incrémentez l'adresse octet par octet, en écrivant au fur et à mesure dans le fichier, jusqu'à ce que la lecture échoue (erreur / segfault). Assurez-vous que le programme termine proprement et enregistre le dump partiel avant l'erreur. Inspirez-vous du code fourni dans l'exercice 1.

II) Pagination

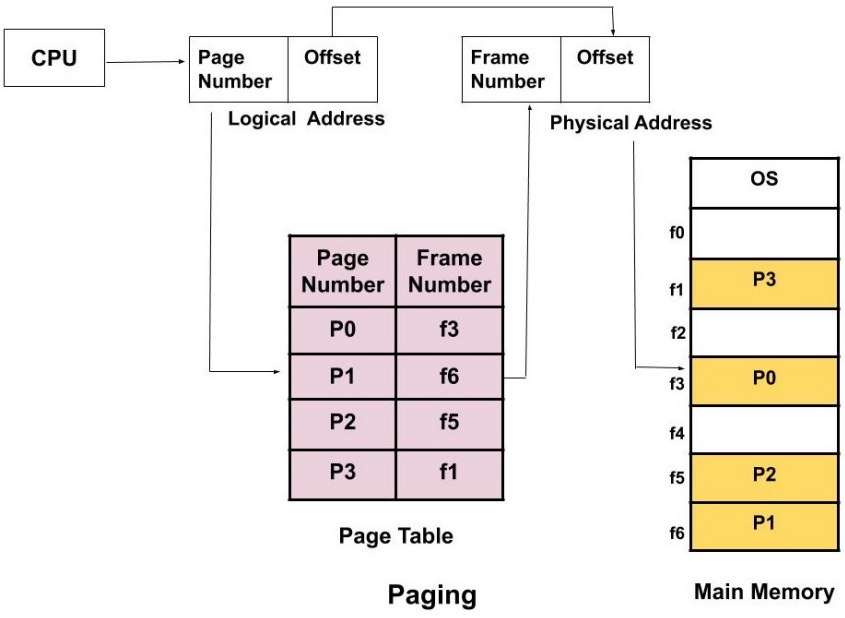

FIGURE 2 – Mécanisme de pagination en mémoire virtuelle.

La **pagination** est un mécanisme fondamental de la mémoire virtuelle qui permet à chaque processus de disposer d'un espace d'adressage *continu et isolé*, même si la mémoire physique est fragmentée ou partagée. Elle repose sur le découpage de la mémoire en blocs de taille fixe appelés *pages*. Chaque page virtuelle du processus correspond à un *cadre* (ou *frame*) dans la mémoire physique. La taille d'une page

est typiquement de 4 Kio ou 8 Kio, et les adresses virtuelles sont divisées en deux parties : le numéro de page virtuelle et le décalage (offset) à l'intérieur de la page. Lorsqu'un processus accède à une adresse virtuelle, la **MMU** (Memory Management Unit) utilise une *table de pages* pour traduire le numéro de page virtuelle (VPN) en un numéro de cadre physique (PFN). L'adresse physique est ensuite calculée en combinant ce cadre et l'offset.

Exercice 3 : Il est temps de tourner la page

1. Écrivez un programme en C qui affiche la taille d'une page mémoire sur votre système. Pour cela, utilisez la fonction `getpagesize()` ou l'appel `sysconf(_SC_PAGESIZE)`. Comparez la valeur obtenue avec vos camarades. Quelle est l'unité utilisée et pourquoi cette taille est-elle fixée par le matériel ?
2. Écrivez un programme qui alloue un grand tableau via `malloc()` (par exemple 1 Go). Affichez ensuite la mémoire réellement utilisée par le processus avec `cat /proc/self/status | grep VmRSS` (ou `vmmap <pid>` sous macOS). Puis initialisez le tableau avec une boucle d'écriture et observez la différence.

Une **faute de page** (*page fault*) se produit lorsqu'un programme tente d'accéder à une page mémoire virtuelle qui n'est pas actuellement présente en mémoire physique (RAM) ou dont les permissions ne permettent pas l'accès demandé. Lorsque cela arrive, le processeur interrompt le programme et déclenche une exception interne. Le noyau analyse alors la cause et détermine la marche à suivre : si la page est légitime mais simplement absente, elle est chargée en mémoire ; sinon, le processus est interrompu avec une erreur de segmentation.
On distingue deux types de fautes de page. Une **faute mineure** (*minor page fault*) survient lorsque la page demandée est déjà présente en mémoire mais n'est pas encore associée au processus (par exemple partagée ou juste non mappée dans sa table de pages). Dans ce cas, le noyau met simplement à jour la table de pages sans lire sur le disque. Une **faute majeure** (*major page fault*) se produit lorsque la page n'existe pas en mémoire et doit être chargée depuis le disque ou un fichier de swap. Ce chargement implique un accès au stockage secondaire, donc beaucoup plus lent (de l'ordre du millier de fois plus qu'un accès mémoire).
Les fautes de page sont un mécanisme normal et nécessaire du *demand paging*, mais un trop grand nombre de fautes majeures ralentit fortement un programme et peut provoquer un *thrashing* si le système passe son temps à échanger des pages plutôt qu'à exécuter le code.

3. La fonction `getrusage()` est définie dans `<sys/resource.h>`. Cette fonction remplit une structure de type `struct rusage` contenant plusieurs informations sur l'utilisation des ressources par le processus courant.

```c
#include <sys/resource.h>

struct rusage usage;
getrusage(RUSAGE_SELF, &usage);
```

La structure `rusage` contient, entre autres, deux champs utiles ici : `ru_minflt` (fautes de page mineures) et `ru_majflt` (fautes de page majeures). Ces valeurs sont des compteurs cumulés depuis le début de l'exécution du programme.

```c
printf("Fautes␣mineures␣:␣%ld\n", usage.ru_minflt);
printf("Fautes␣majeures␣:␣%ld\n", usage.ru_majflt);
```

On peut ensuite mesurer les fautes avant et après l'accès à la mémoire pour observer la différence :

```c
struct rusage before, after;

getrusage(RUSAGE_SELF, &before);
// ... accès mémoire ici ...
getrusage(RUSAGE_SELF, &after);

printf("Fautes␣mineures␣:␣+%ld\n", after.ru_minflt - before.ru_minflt);
printf("Fautes␣majeures␣:␣+%ld\n", after.ru_majflt - before.ru_majflt);
```

Ces compteurs permettent de visualiser les fautes de page générées par le *demand paging* : chaque première écriture dans une nouvelle page provoque une faute mineure, tandis qu'une faute majeure correspond à une page devant être lue depuis le disque (swap).

Reprenez le programme précédent et utilisez la fonction `getrusage()` pour afficher le nombre de fautes de page mineures et majeures. Exécutez le programme avant et après avoir accédé à toutes les pages du tableau. Qu'observez-vous ?

4. Bonus : Utilisez `mmap()` pour mapper un fichier de grande taille (par exemple 100 Mo). Mesurez le temps d'exécution de l'appel à `mmap()` puis parcourez progressivement le fichier en lecture. Observez le nombre de fautes de page. Pourquoi `mmap()` est-il beaucoup plus rapide que la lecture du fichier avec `read()` ?

III) Buffer Overflow

Un *buffer overflow* (dépassement de tampon) se produit lorsqu'un programme écrit plus de données dans une zone mémoire prévue pour une taille fixe (un tampon) que ce que cette zone peut contenir. Les octets excédentaires débordent alors hors du tampon et peuvent écraser des données adjacentes en mémoire. Selon la zone écrasée, les conséquences peuvent aller d'un comportement incorrect simple jusqu'à une corruption de l'exécution du programme.

Exercice 4 : Rien de bien méchant

Dans cet exercice, nous allons chercher à comprendre le fonctionnement d'un *buffer overflow* en proposant un code vulnérable très basique. La compilation du programme devra se faire avec une option particulière afin que le compilateur n'applique pas les mécanismes de protection de la pile :

```
gcc -g -O0 -fno-stack-protector -o mon_programme mon_programme.c
```

1. Écrivez un programme qui initialise un tableau de caractères (alloué sur la pile) de taille fixe (par exemple 4 octets) et qui demande à l'utilisateur de remplir cette chaîne via l'entrée standard. Que se passe-t-il si la chaîne fournie est plus longue que la taille allouée ? Où seront écrits les octets supplémentaires ?
2. Pour vérifier si une entrée non contrôlée peut effectivement écraser d'autres zones mémoire, essayez de modifier une variable initialisée au préalable. Pour vous assurer que le tampon et la variable sont contigus en mémoire, déclarez-les au sein d'une même structure, par exemple :

```c
struct S {
  char buf[4];
  int flag;
};
```

   Implémentez et testez le code vulnérable.
3. Quels pourraient être les risques pour une application en production si une telle vulnérabilité était présente dans son code ?

Exercice 5 : Pour les plus courageux...

En bonus, voici un excellent TP sur les buffer overflows, publié par un chercheur en sécurité de l'université de Xidian (Chine) : https ://github.com/firmianay/Life-long-Learner/blob/master/SEED-labs/buffer-overflow-vulnerability-lab.md