7. Gestion de la mémoire

Pr Alain Tchana - (alain.tchana@ens-lyon.fr)

ENS Lyon - France

2019-2020

`https://gitlab.com/lenapster/ensl-asr2.git`

Le fil de l'histoire

Séance passée

- Rôle du gestionnaire de mémoire
- Virtualisation de la mémoire
- Chargement d'un binaire et espace d'adressage
- Segmentation

Ce jour

- Rappel
- Gestionnaire user space du heap
- Programmation C
  - Pointeur, `malloc()`, `realloc()`, `free()`, `brk()`, `sbrk()`, `mmap()`, `unmap()`

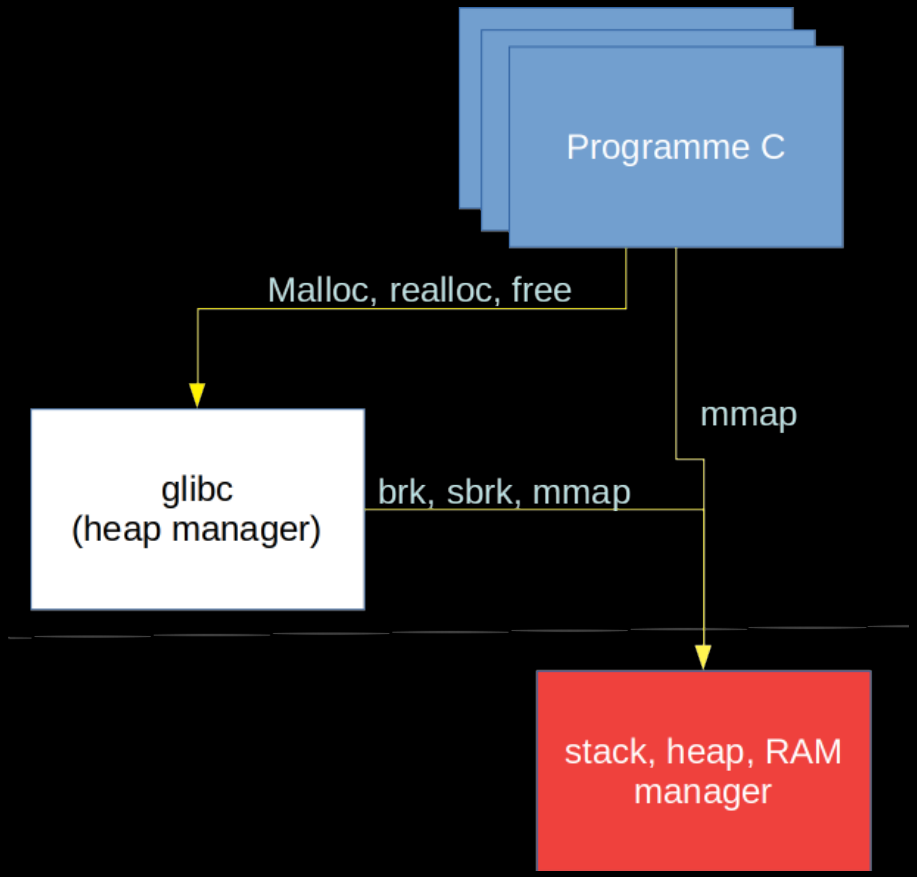
Lien avec la dernière séance

- Le processus est crée
- Ses zones `vm_areas` ont été créées
  - en fonction des sections décrites dans le binaire
- L'allocation de la RAM se fera uniquement lorsque le processeur essaie d'accèder à une zone
- L'allocation effective de la mémoire virtuelle se fera
  - pour la stack: lors des appels de fonctions
  - heap: lorsque le processus fera un `brk()`, `sbrk()`, ou `mmap()`

Rappel

- La gestion de la mémoire se situe à deux niveaux
  - mémoire virtuelle: en user space (exemple glibc pour le heap) et OS (une partie du heap et totalement la stack)
  - mémoire physique (RAM): dans l'OS et le hardware

Rappel


Programme C
Malloc, realloc, free
mmap
glibc
(heap manager)
brk, sbrk, mmap
stack, heap, RAM
manager

Gestionnaire de la mémoire

- Dans les deux niveaux, les questions qui se posent sont
  - comment organiser les bouts de mémoire libres?
  - comment choisir les bouts de mémoire à allouer?
  - comment gérer la pression sur la mémoire?

Gestionnaire du heap

- Nous l'appellerons Heap Manager (HM)
- Invocation provoquée par les appels `malloc()`, `realloc()`, `free()` dans de votre programme
- Le gestionnaire niveau user space
  - maintient un ensemble de blocks
  - en cas de manque de blocks, demande supplémentaire de la mémoire virtuelle à l'OS, en utilisant le syscall `brk()` ou `sbrk()`. Dans ce cas, l'OS déplace tout simplement le pointeur `break`. Le syscall `mmap()` est utilisé lorsque le gestionnaire nécessite un gros bloc. Dans ce cas, l'OS crée une nouvelle `vm_area`

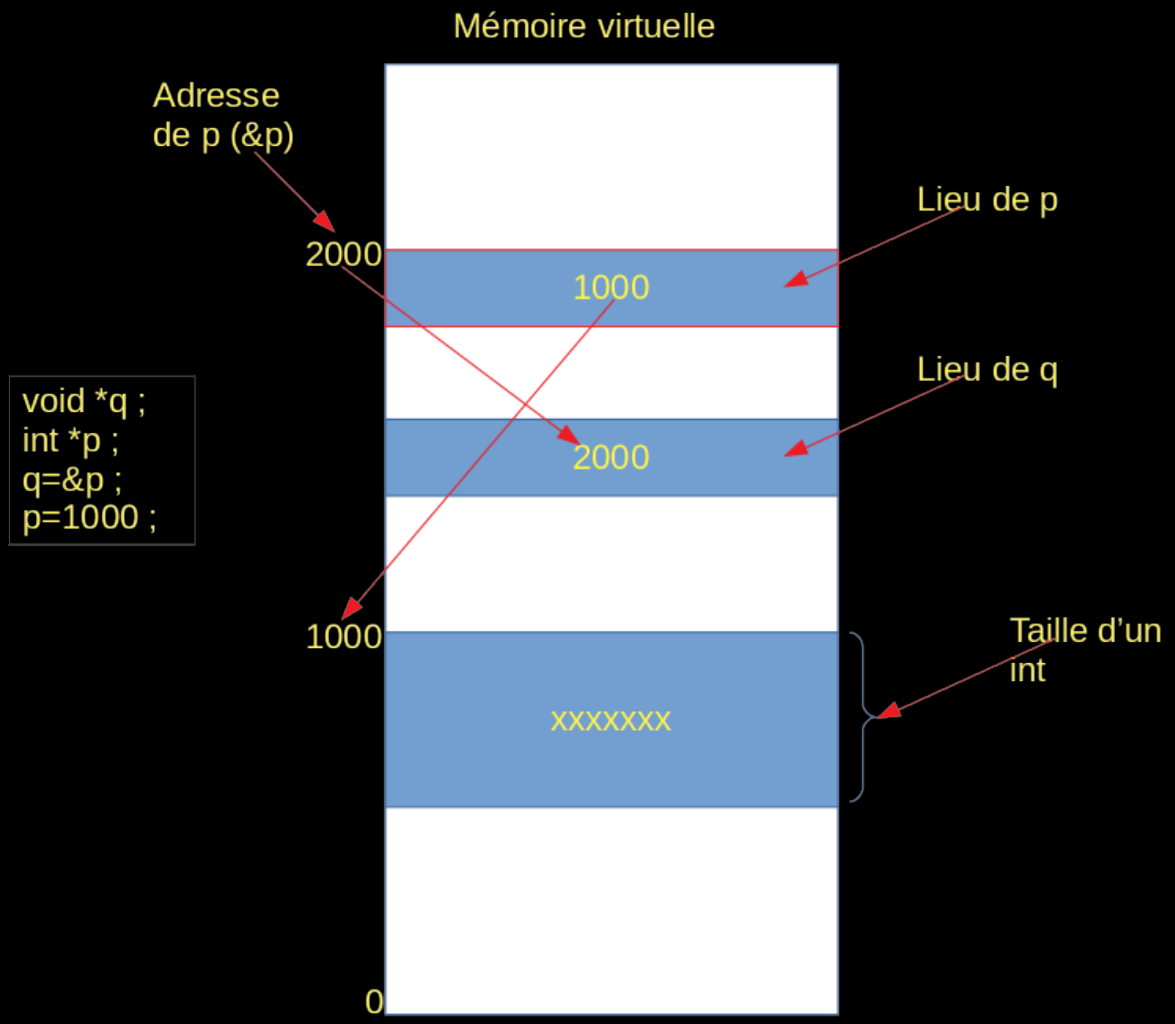
HM

- Avant de parler des politiques de gestion du heap
  - Focus sur les pointeurs, `malloc()`, `realloc()`, `free()` et `mmap()`

Pointeurs

- Une variable pointeur déclarée "`type *p`" c'est tout bête
  - "p" une variable qui contient un très grand nombre. Ce nombre sera interprété comme l'adresse virtuelle d'un lieu mémoire.
  - sachez que "*type*" n'a aucun effet sur ce qu'est "p". Ce dernier reste un grand nombre.
  - "*type*" permet tout simplement de préciser le nombre d'octets qu'on ira chercher à partir du lieu pointé lorsqu'on fera reférence à ce lieu via "p".
  - la lecture ou l'affectation du lieu pointé par "p" se fait avec `*p`
- Soit "`int *p;`"
  - alors "`*p`" représente le contenu du lieu pointé
  - ici, ce contenu sera sur `sizeof(int)` octets (4 par exemple)

Pointeurs


Mémoire virtuelle
Adresse
de p (&p)
2000
1000
Lieu de p
Lieu de q
2000
void *q ;
int *p ;
q=&p ;
p=1000 ;
1000
xxxxxxx
Taille d'un
int
0

Pointeurs

- "`&var`": est l'adresse (virtuelle) du lieu mémoire où se trouve "`var`"
  - "`var`" peut être n'importe quoi, y compris un pointeur "p" déclaré "`int *p`". Dans ce cas, "&p" sera l'adresse du lieu mémoire où se trouve la variable "p" (n'oubliez pas que "p" n'est qu'un grand nombre stocké quelque part en mémoire).
- Personnellement je n'aurai pas utilisé "*" (par exemple `int *p`) pour à la fois la déclaration et l'accès au contenu. J'aurai utilisé la syntaxe "`int &p`" pour rester cohérent, avec
  - "*p" qui est le contenu du lieu pointé
  - "p" qui est une adresse ("&" indique toujours l'adresse)
  - bref...

Pointeurs

- Soit "`type *p;`"
  - "`p=p+20`": va mettre dans "`p`", l'ancienne valeur de "`p`" plus `sizeof(type)` fois 20
  - On voit ici une autre utilité de "`type`"

Pointeurs

- exercices

malloc()

- Lorsqu'on déclare un pointeur sans l'initialiser, il est `NULL` (le pointeur contient la valeur 0, donc pointe sur l'adresse 0)
  - si une instruction essaie d'accéder au lieu pointé, il y aura un segfault car l'OS dira que 0 ne se trouve dans aucune `vm_area` du processus.
  - par contre vous pouvez lire la valeur du pointeur (qui sera `NULL` ou 0)
- L'initialisation d'un pointeur se fait ainsi
  - soit en affectant au pointeur une adresse dont vous savez appartenir à une `vm_area`. La meilleure façon de le faire est de partir d'une variable existante (exemple "`p=&var`").
  - soit via un `malloc()`

void *malloc(size_t size)

- "type *p=(type *)malloc(nombre-octets)"
  - "(type *)" est un cast
  - "*nombre-octets*" signifit tout simplement qu'à partir de l'adresse pointée par "p", "*nombre-octets*" sont réservés, donc accessible à partir de votre programme.
  - généralement, on remplace "*nombre-octets*" par `sizeof(type)` fois le nombre de zones de taille "type" que l'on souhaite réserver.

`void free(void *ptr);`

- Permet de libérer une zone de mémoire précédemment allouée par `malloc()`
  - `ptr` est une valeur qui avait été retournée par un `malloc()`

Implantation de `malloc()` et `free()`

- Ces deux appels sont traités par le HM
  - HM utilise les syscalls `brk()`, `sbrk()`, et `mmap()` pour réserver une large zone mémoire dans le heap auprès de l'OS
  - il utilise par la suite cette zone pour servir les `malloc()`, `realloc()` et `free()`

Implantation de `void *realloc(void *ptr, size_t size);`

- Demande un ajustement de la taille de la zone précédemment réservée par `malloc()`
- HM peut allouer un nouveau bloc pour satisfaire cette requête
  - copie des données de l'ancien bloc vers le nouveau bloc
  - l'ancien bloc est libéré

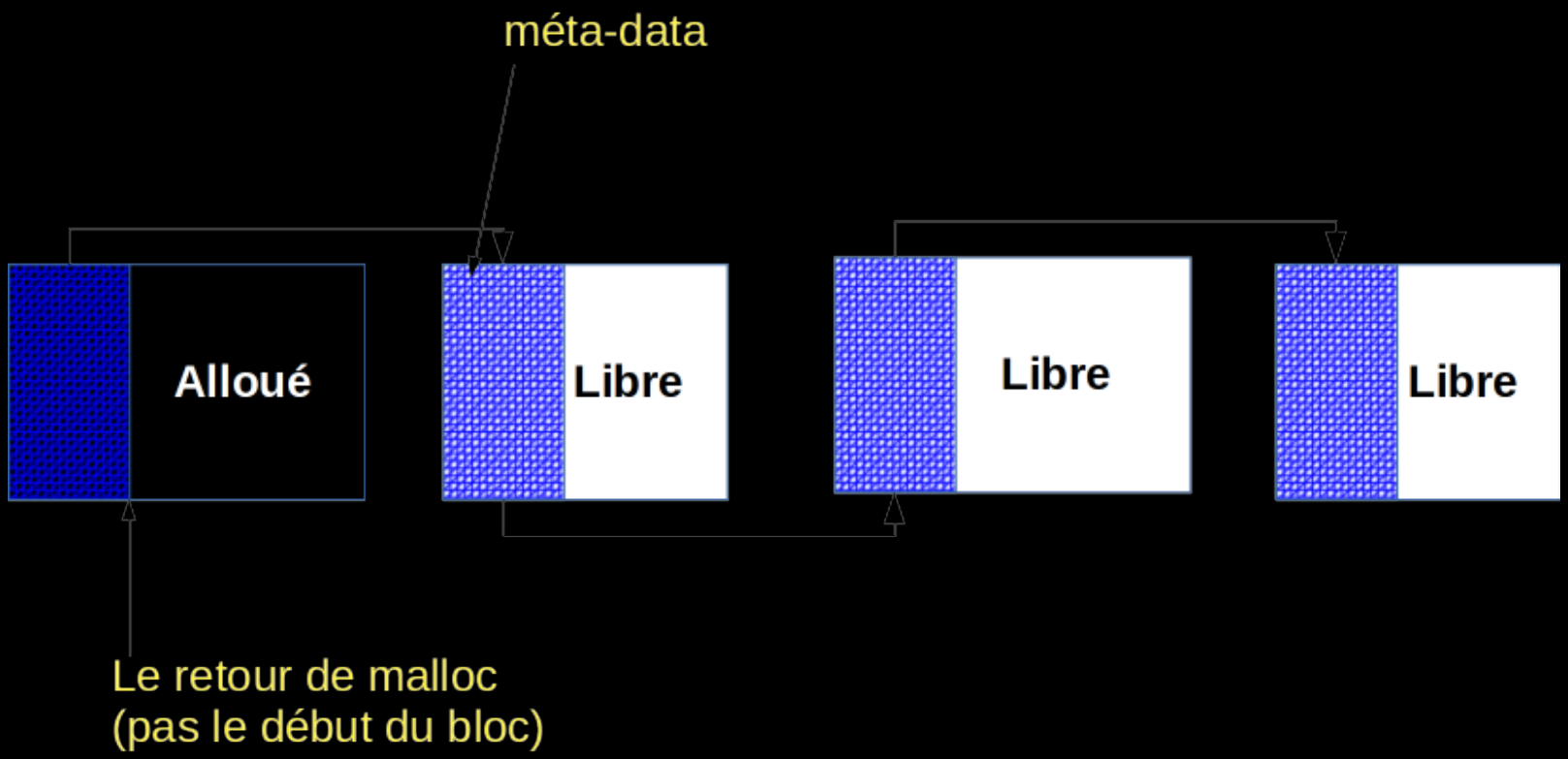
brk() et sbrk()

- syscalls à destination du HM qui se trouve dans l'OS
- `int brk(void *addr);`
  - déplace le pointeur "`break`" (dans `task_struct`) du processus à l'adresse pointée par `addr`
- `void *sbrk(intptr_t increment);`
  - incremente le "`break`" du programme de "`increment`" octets

Gestion de la mémoire virtuelle par HM

- Gère les blocs contigus de mémoire virtuelle
  - chaque bloc est soit libre soit alloué
  - question: quelle structure de données utilisées pour gérer ces blocs, comment répondre à un malloc, un free, sans gaspiller des blocs, en répondant rapidement?
- Approche généralement utilisée: liste chaînée de blocs
  - chaque bloc voit ses premiers octets utilisés pour stocker des méta-data sur le bloc: libre ou pas, taille de la partie data (utilisée par le `malloc()`), pointeur vers le block suivant
  - l'adresse retournée à `malloc()` est le début de la partie data (voir slide suivant)

HM


méta-data
Alloué
Libre
Libre
Libre
Le retour de malloc
(pas le début du bloc)

HM

- Quelques problèmes de sécurité liés à la structure précédente
  - *buffer overflow*: le programme utilise un buffer (pointeur), mais écrit ou lit au dela (en avant)
  - *buffer underflow*: sens inverse du précédent
  - Dans les deux cas, possibilité de lire les méta-data

HM

- Quelques algorithmes d'allocation
  - *first fit*: le premier bloc qui satisfait la requête, recherche toujours à partir du premier bloc
  - *next fit*: comme le précédent, mais recherche à l'endroit où l'on avait laissé le curseur lors du dernier appel
  - *best fit*: recherche le bloc égale ou le plus petit bloc supérieur à la demande
- A prendre en compte
  - jointure des blocs adjacents: quand le faire? (lors de l'allocation, du free?)

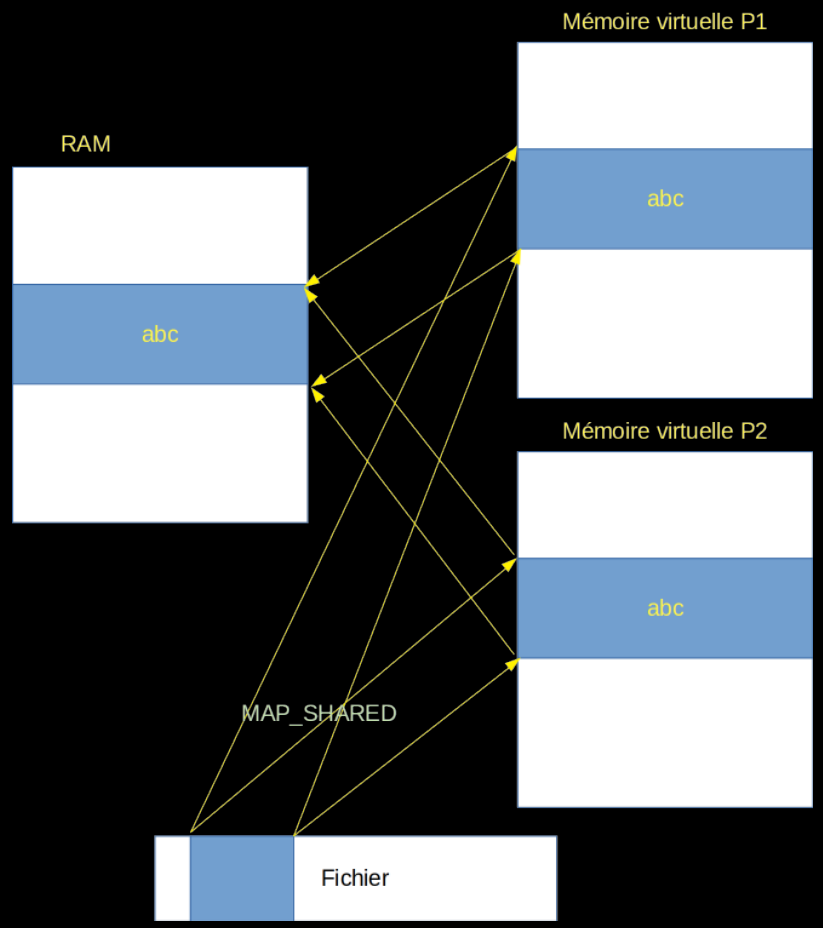
mmap()

- Permet de projeter une portion d'un fichier ou d'une zone mémoire vers l'espace d'adressage du processus
  - ainsi l'accès à la portion projetée se fait via la zone de la mémoire virtuelle, en utilisant les opérations traditionnelles d'accès mémoire (manipulation de pointeurs)
- `void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);`, VOIR LE MAN
  - `addr`: est l'adresse mémoire à partir duquel la portion sera projetée. Autrement dit, pour lire le contenu de la portion, il suffira de faire `*addr` ou `addr[xx]`. Si `addr` est `NULL` alors le choix est laissé à l'OS pour trouver l'endroit où projeter. `mmap` retourne dans tous les cas le lieu de projetion dans la mémoire virtuelle.
  - `length` est la taille de la portion
  - Dans le cas de la projection d'une portion de mémoire virtuelle (pas fichier), on parle de mapping anonyme.

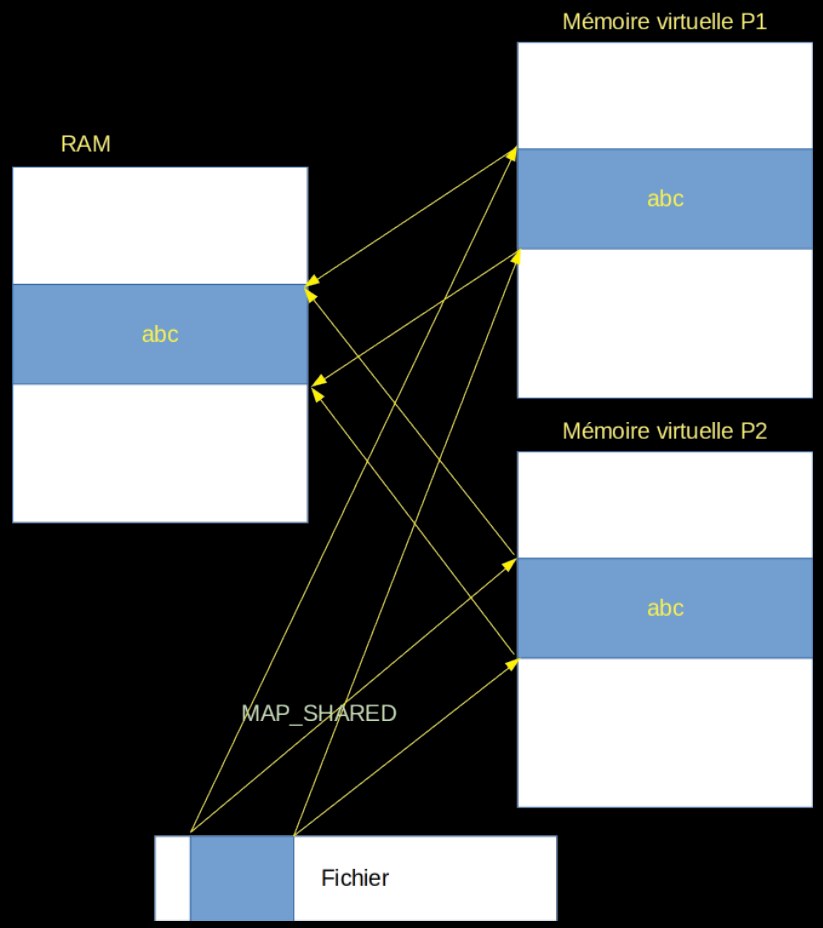
mmap(), suite

- `fd` et `offset` sont utilisés lorsque la projection concerne un fichier sur disque. Dans ce cas, `fd` est le descripteur du fichier, `offset` est le lieu dans le fichier qui marquera le début de la projection. `offset` doit être un multiple de la taille d'une page mémoire (`sysconf(_SC_PAGE_SIZE)`)
- `prot` définit la protection à appliquée à la zone de projection
  - `PROT_EXEC, PROT_READ, PROT_WRITE, PROT_NONE`
- `flags` définit comment la propagation des modifications se fait, car plusieurs processus peuvent partager la zone de projection
  - `MAP_SHARED, MAP_PRIVATE`
  - on peut combiner les précédents avec `MAP_ANONYMOUS, MAP_POPULATE, MAP_UNINITIALIZED`, en utilisant OR
- `unmap(void *addr, size_t length);`
  - pour supprimer une projection précédente

mmap(), MAP_SHARED



mmap(), MAP_PRIVATE



mmap(), sans fichier (anonymous)

- `addr = mmap(NULL, length, PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);`
  - NULL pour laisser l'OS choisir où faire la projection
  - -1 et 0 car pas une projection d'un fichier
  - `MAP_ANONYMOUS`

mmap(), fichier

- On ouvre un fichier (on verra plus tard): `fd = open("/dev/zero", O_RDWR);`
- On projette: `addr = mmap(NULL, length, PROT_READ | PROT_WRITE, MAP_PRIVATE, fd, 0);`