QCM2: contrôle continu IF-3-SYS d'avril 2025

la mémoire virtuelle repose sur un mécanisme de traductions d'adresses: lors de chaque accès mémoire, l'adresse virtuelle émise par le CPU est traduite par la MMU en une adresse physique avant d'être transmise à la mémoire physique .2

Pour chaque affirmation ci-dessous, indiquez si elle est correcte

- [ ] La MMU choisit le placement en mémoire physique des données des processus.
- [ ] La MMU choisit les adresses virtuelles où sont placées les régions allouées via `mmap()`.
- [x] Aucune des autres réponses n'est correcte.
- [ ] La MMU choisit quelle page de mémoire physique évincer (*swap out*) lors d'un défaut de page.
- [ ] La MMU choisit quelles données sont mémorisées dans la mémoire cache du processeur.

Dans cette question, on s'intéresse à comparer la mémoire virtuelle paginée (vue en cours et en TP) avec une autre approche qu'on appellera "partitionnement dynamique". Dans cette seconde approche, chaque processus est stocké d'un seul bloc (contigu) en mémoire physique. Le noyau s'occupe de faire l'allocation et le recyclage de ces blocs lors des créations et fins de processus. Pour chaque affirmation ci-dessous, indiquez si elle est correcte:

- [x] La mémoire virtuelle paginée va typiquement causer moins de *fragmentation externe* que le partitionnement dynamique.
- [ ] La mémoire virtuelle paginée va typiquement consommer moins d'espace sur le processeur que le partitionnement dynamique.
- [x] La mémoire virtuelle paginée va typiquement consommer moins d'espace sur le disque que le partitionnement dynamique.
- [ ] La mémoire virtuelle paginée va typiquement causer moins de *fragmentation interne* que le partitionnement dynamique.

Combien de bits sont nécessaires pour exprimer une adresse physique sur un système avec un PAS de 512 octets ?

Réponse : 9

On s'intéresse à un système avec des adresses virtuelles et physiques encodées sur 16 bits, et une traduction d'adresses utilisant des pages de 4kio. Le diagramme ci-dessous illustre la table de pagination du processus en cours (le signe ∅ indique un PTE invalide, les nombres sont notés en décimal)

VPN	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
PPN	8	6	11	∅	1	10	5	∅	3	∅	7	4	12	∅	∅	2

Classez ces adresses virtuelles par ordre croissant des adresses physiques correspondantes. (Les adresses invalides sont à classer à la fin)

0xFFFF 0x1234 0xACAB 0x0000

On suppose dans cette question un allocateur dynamique dont la freelist est composée initialement de trois blocs libres de taille 300, 100 et 200 (chaînés dans cet ordre). Pour simplifier, on suppose qu'un bloc libre de taille N peut servir à allouer une zone de taille N (autrement dit, on néglige l'espace occupé par les méta-données) et que l'allocateur peut toujours "découper" les blocs sans perte de place.

L'application demande successivement à allouer des zones de tailles 200, puis 150, puis 150.

Pour chaque affirmation ci-dessous, indiquez si elle est correcte:

- [ ] Si l'allocateur applique la stratégie *first-fit*, il arrivera à satisfaire les trois requêtes sans devoir faire appel au noyau.
- [ ] Si l'allocateur applique la stratégie *worst-fit*, il arrivera à satisfaire les trois requêtes sans devoir faire appel au noyau.
- [x] Si l'allocateur applique la stratégie *best-fit*, il arrivera à satisfaire les trois requêtes sans devoir faire appel au noyau.

On se place dans le contexte de notre TP4 (allocation dynamique) et on suppose que les parties 2 (désallocation) et 3 (découpage) ont été implémentées mais pas encore les parties 4 et 5 (boundary tags et fusion). En partant de l'état initial du tas (i.e. après un `mem_init()`) quelle est la plus grande taille mémoire que l'application peut demander (à `mem_alloc()`) deux fois de suite avec succès ?

Réponse : 192

Le programme ci-dessous alloue plusieurs tableaux. Dans quelle section de l'espace d'adressage sera placé le tableau tabB ?

```
int tabA[]={1,2,3,4};
int main(void)
{
   static int tabB[100];
   for(int i=0; i<100; i++)
   {
      tabB[i] = tabA[i % 4];
   }
  return 0;
}
```

- (•) `.data`
- ( ) `.text`
- ( ) `.stack`
- ( ) `.heap`

On s'intéresse dans cette question au programme suivant:

```
int main()
{
    int *p = (int *)malloc(sizeof(int));
    *p = 10;
    p = NULL;
    free(p);
    return *p;
}
```

Pour chaque affirmation ci-dessous, indiquez si elle est correcte:

- (•) Ce programme comporte des bugs de pointeurs, donc si on l'exécute il risque de se planter (*crash*).
- ( ) Aucune des autres réponses n'est correcte.
- ( ) Ce programme fait des opérations inutiles, mais si on l'exécute il renverra toujours la valeur 10.
- ( ) Ce programme comporte une fuite de mémoire, donc si on l'exécute de nombreuses fois, on risque d'épuiser la RAM du système.
- ( ) Ce programme ne sera pas accepté par le compilateur car c'est une erreur d'appeler `free()` sur un pointeur nul.

Quelle est la valeur de la variable `x` après l'exécution de la ligne ci-dessous ?

```
int x = 25 & ~20 ;
```

Réponse : 9