Programmation Système
TP n° 6 : projection de mémoire (2)
1-2 mars 2016

Exercice 1 : Mémoire partagée et sémaphores POSIX

On se donne la structure de données suivante :

```
struct rendezvous {
    int value;
    sem_t vide;
    sem_t plein;
};
```

Écrivez un programme qui crée à l'aide de `sem_open` une zone de mémoire partagée de taille `sizeof(struct rendezvous)` et initialise `vide` à 1 et `plein` à 0. Il entre ensuite dans une boucle infinie qui décrémente ensuite le sémaphore `plein`, affiche la valeur de `value`, puis incrémente le sémaphore `vide`.
Écrivez un deuxième programme qui ouvre la zone de mémoire partagée, décrémente le sémaphore `vide`, stocke un entier tiré au hasard dans `value`, puis incrémente le sémaphore `plein`. Testez la communication entre les deux programmes.

Exercice 2 : Un allocateur de mémoire spécialisé

La fonction `malloc` permet d'allouer une quantité arbitraire de mémoire. Lorsqu'on alloue beaucoup de structures de la même taille, il est parfois (mais rarement) intéressant d'implémenter un allocateur de mémoire spécialisé.
Dans le fichier

`http://www.pps.univ-paris-diderot.fr/~jch/enseignement/systeme/dict.c`

vous trouverez un correcteur orthographique élémentaire qui manipule un arbre binaire constitué de `struct node` allouées à l'aide de `malloc`. Le but de cet exercice est d'utiliser un allocateur spécifique pour les `struct node`.
On se donne les définitions suivantes :

```
#define ARENA_SIZE (16 * 4096 - 120)
#define NODES_PER_ARENA (ARENA_SIZE / sizeof(struct node))

struct arena {
    struct arena *next;
    char bitmap[NODES_PER_ARENA];
    struct node nodes[NODES_PER_ARENA];
};

struct arena *arenas = NULL;
```

Votre code manipulera une liste chaînée de `struct arena`. Chaque arène contient un tableau de `struct node` dont certaines sont libres et d'autres sont utilisées par le programme utilisateur, ce qu'indique le tableau `bitmap`.

1. Écrivez une fonction

```
struct arena *new_arena(void);
```

qui alloue une `struct arena` à l'aide de `mmap`, initialise le champ `next` à NULL et le champ `bitmap` à un tableau de 1, insère la nouvelle arène en tête de la liste `arenas`, et la retourne. Si l'allocation échoue, votre fonction retournera NULL.

2. Écrivez une fonction

```
struct node *allocate_node_1(struct arena *arena);
```

qui cherche un entier `i` tel que `arena->bitmap[i]` vaut 1, met `arena->bitmap[i]` à 0 puis retourne l'entrée de `arena->nodes` correspondante. Si aucun tel `i` existe, elle retourne `NULL`.

3. Écrivez une fonction

```
struct node *allocate_node(void);
```

qui parcourt la liste des arènes, et essaie d'effectuer un `allocate_node_1` sur chacune d'elles. Si aucune arène ne contient d'entrée libre, votre fonction appellera `new_arena` et fera l'allocation dans la nouvelle arène. Attention à la gestion des erreurs.

Écrivez une fonction `void free_node(struct node *node)` qui ne fait rien. Modifiez les fonctions `new_node` et `destroy_tree` du programme `dict.c` pour qu'elles utilisent vos fonctions. Testez votre code à l'aide de `valgrind`.

4. Modifiez la fonction `free_node` pour qu'elle trouve la bonne entrée dans la liste des arènes et mette l'entrée correspondante de `bitmap` à 1. Testez de nouveau votre programme.

5. Comparez le temps d'exécution du programme d'origine avec votre programme modifié. Conclusion ?

6. S'il vous reste du temps, modifiez la fonction `free_node` pour qu'elle rende au système la mémoire occuppée par une arène qui devient vide.

7. S'il vous reste encore du temps, modifiez la `struct arena` pour qu'elle utilise un vecteur de bits au lieu d'un vecteur d'octets. Vous pouvez vous servir de la fonction `ffs`.