Programmation système avancée
TP n° 5 : Allocateurs de mémoire

Exercice 1 :

1. Tapez « `free -h` ». Expliquez la sortie. Quelle est la différence entre la mémoire libre et la mémoire disponible ?
2. Tapez « `/sbin/swapon -s` ». Expliquez la sortie.

Exercice 2 :

1. Écrivez un programme qui prend un entier n en paramètre de ligne de commande puis fait un appel à `mmap` pour allouer n mégaoctets de mémoire (utilisez `strtol` pour analyser le paramètre). Testez votre programme avec 1 MB, 100 MB, 1 GB, 10 GB, 100 GB. Que constatez-vous ? Le comportement est-il le même sous Linux et Mac OS X ?
2. Modifiez votre programme pour qu'il touche toute la mémoire allouée (il suffit d'écrire un octet dans chaque page, utilisez la fonction `getpagesize`). Exécutez votre programme sous `time`. Où le programme a-t-il passé le plus de temps ? Exécutez-le sous `/bin/time`. Combien y a-t-il eu de défauts de mémoire ?

Exercice 3 :

Téléchargez le code fourni. Le fichier `binary-trees.c` contient un *benchmark* standard d'allocation mémoire. Ce *benchmark* est incomplet : pour le compiler, il faut écrire un fichier additionnel qui implémente l'interface décrite dans `alloc.h`.

1. Créez un fichier `alloc1.c` qui implémente `alloc` à l'aide de `calloc` et `dealloc` à l'aide de `free`. Testez le programme, et notez les performances (il faudra compiler avec l'option `-O2`). Exécutez votre programme sous `valgrind`, et assurez-vous qu'il n'y a pas d'avertissements. (Désormais, tous vos programmes devront être testés sous `valgrind`.)
2. Créez un fichier `alloc2.c` qui implémente `alloc` en effectuant un *mapping* anonyme privé à l'aide de `mmap` et `dealloc` à l'aide de `munmap`. Que constatez-vous ?
3. Créez maintenant un fichier `alloc3.c` qui maintient une variable globale `arena` de type `struct node *` valant initialement `NULL` et un variable entière `allocated` valant initialement 0. Lorsque le programme appelle `alloc`, la fonction vérifie si `arena` vaut `NULL` et, si c'est le cas, alloue une zone de `allocsize` = 16 000 000 `struct node` à l'aide de `mmap`. Elle vérifie ensuite que `allocated` est strictement inférieur à `allocsize`, et, si c'est le cas, renvoie `arena` + `allocated` puis incrémente `allocated`. La fonction `dealloc` ne fait rien du tout.
4. Le programme précédent est rapide, mais il ne libère jamais la mémoire. Pour éviter ce problème, nous allons stocker une *bitmap* des `struct node` alloués, ou un bit vaut 1 si la case correspondante de `arena` est libre. La *bitmap* sera représentée par un tableau `uint64_t bitmap[n]`, où `n` = `allocsize/sizeof(uint64_t)`. La fonction `alloc` recherchera un *bit* à 1 puis le mettra à 0, et retournera l'entrée du tableau `arena` correspondante (en s'assurant

qu'elle a été initialisée à 0). Quant à la fonction `dealloc`, elle calculera l'indice correspondant à son paramètre et mettra le *bit* correspondant à 1.

Pour initialiser un mot de la *bitmap*, vous pourrez utiliser la notation `~(0ULL)`. Pour chercher un *bit* à 1, vous pourrez chercher un `uint64_t` non-nul puis trouver le premier *bit* à l'aide de la fonction `ffsll`.

5. La version précédente est assez lente. Plusieurs optimisations sont possibles :

- la plus simple et cependant efficace consiste à se souvenir de `courant`, un indice tel que tel que `bitmap[courant]` est non nul, et de refaire l'allocation suivante sans nouvelle recherche lorsque c'est possible ;
- dans le cas où `bitmap[courant]` vaut 0, on peut optimiser la recherche en recherchant une case vide depuis la position précédente (soit vers la gauche puis vers la droite, ce qui améliore la localité, soit vers le droite en bouclant au bout — expérimentez !) ;
- enfin on peut garder un pointeur `dernier` tel que pour tout `i` supérieur à `dernier`, `bitmap[i]` vaut `~(0ULL)` et, au delà d'un certain taux de remplissage de `bitmap` entre `0` et `dernier`, choisir `bitmap[dernier+1]` au lieu de lancer une nouvelle recherche qui risquerait d'être longue.

6. Votre code ne libère jamais la mémoire allouée. Utilisez la fonction `madvise` avec le paramètre `MADV_DONTNEED` pour informer le noyau que la mémoire virtuelle allouée n'est plus utile. Pourquoi n'aurait-on pas pu utiliser `munmap` ?