PROG L3-MI
DS

2023-10-24

Consignes. *La durée de ce devoir surveillé est d'une heure trente. Aucun document n'est autorisé.* ***Dans tous les exercices, il est possible d'admettre le résultat d'une question pour répondre aux questions suivantes.*** *Toute réponse doit être justifiée.*

Exercice 1 (environ 5 points) : `swap`

Q.1 :

1. Écrivez une fonction `swap` de signature :

   ```
   void swap(void *a, void *b, size_t size)
   ```

   qui échange le contenu de `size` octets de la zone mémoire pointée par `a` avec `size` octets de la zone mémoire pointée par `b`. Vous pouvez si nécessaire supposer que les deux zones mémoires à échanger ne se recouvrent pas.
2. Expliquez l'intérêt de l'utilisation de types `void *` pour les deux premiers arguments de cette fonction.
3. Donnez un exemple d'utilisation de cette fonction pour échanger la valeur de deux variables de type `int`, dans le code ci-dessous (il est seulement demandé d'écrire le code devant se trouver à la place du commentaire `// ...`).

   ```
   int a = 3;
   int b = 4;

   // ...

   assert((a == 4) && (b == 3));
   ```

Exercice 2 (environ 5 points) : `calloc` (3)

La fonction `void *calloc(size_t count, size_t size)` de la bibliothèque C standard a les spécification suivantes : elle alloue sur le tas une zone mémoire contiguë permettant de stocker `count` éléments ayant chacun une taille de `size` octets, et renvoie un pointeur vers cette zone mémoire. De plus, cette zone mémoire est initialisée à zéro (c'est à dire que chaque octet de cette zone mémoire doit valoir `(uint8_t)0`).

Q.1 :

1. Écrivez une implémentation de `calloc` qui utilise `malloc` pour effectuer l'allocation mémoire. Cette implémentation doit effectuer l'initialisation mémoire *octet par octet*.

2. Écrivez une seconde implémentation de `calloc` (qui utilise toujours `malloc` pour l'allocation mémoire) et qui cette fois effectue l'initialisation mémoire en utilisant (autant que possible) des écritures sur 8 octets.
3. Dans le cas d'une architecture 64 bits, quel est l'intérêt de la seconde implémentation par rapport à la première ?
4. Si vous souhaitez dans un programme allouer une zone mémoire sur le tas et l'initialiser à zéro, quels peuvent être les intérêts d'utiliser la fonction `calloc` fournie par votre système plutôt que d'effectuer vous même l'allocation et l'initialisation ?

Exercice 3 (environ 10 points) : produit médian ; fonction de hachage

Dans tout cet exercice :

— On note mod (resp. quo) l'opérateur retournant le reste positif (resp. le quotient associé au reste positif) de la division entière.

— On rappelle l'existence en C des opérateurs suivants :

  — `<<` : informellement, `a << b` renvoie l'entier obtenu en décalant les bits de `a` de `b` positions « vers la gauche » (vers les bits de poids fort), en introduisant `b` zéros dans les bits de droite (de poids faible).

  — `>>` : informellement, `a >> b` renvoie l'entier obtenu en décalant les bits de `a` de `b` positions « vers la droite » (vers les bits de poids faible), en introduisant `b` zéros dans les bits de gauche (de poids fort) (en faisant ici l'hypothèse d'une opérande `a` de type non signé).

— L'efficacité *est* un critère de notation

Q.1 :

1. Écrivez une fonction `uint16_t pm(uint32_t x, uint32_t y)` qui prend en entrée deux entiers $x, y \in [\![0, 2^{32} - 1]\!]$ et retourne l'entier $z \in [\![0, 2^{16} - 1]\!]$ valant $(xy \bmod 2^{40}) \operatorname{quo} 2^{24}$ (ou de façon équivalente $(xy \operatorname{quo} 2^{24}) \bmod 2^{16}$).

   Indice : il peut être utile d'introduire dans votre fonction une variable intermédiaire de type `uint64_t`.

Q.2 : Dans cette question, on demande d'utiliser l'instruction `pext` via son *intrinsèque* de signature `uint64_t _pext_u64 (uint64_t a, uint64_t mask)`. Cette instruction renvoie l'entier obtenu en écrivant de façon contiguë et stable (c-à-d préservant l'ordre) les bits de `a` pour lesquels les bits correspondant de `mask` valent 1, et en complétant si besoin les bits de poids fort par des zéros. Ceci est décrit de façon plus formelle dans la Figure 1.

1. Écrivez une fonction `uint16_t pmpe(uint32_t x, uint32_t y)` de mêmes spécifications que la fonction `pm` de la question précédente, et qui utilise l'instruction `pext`.
2. Sachant que l'instruction `pext` ne fait pas partie du jeu d'instruction x86 de base, mais de l'extension BMI2, expliquez *de la façon la plus précise possible* comment faire pour compiler votre fonction `pmpe`.

Q.3 : La fonction `pm` (ou son implémentation alternative `pmpe`) implémente une famille de fonctions de hachage prenant une entrée de 32 bits et renvoyant une *empreinte* de 16 bits. On souhaite étendre cette fonctionnalité à des domaines plus grands, pour être capable de calculer l'empreinte d'une entrée de taille arbitraire. À cette fin, on définit la fonction :

```
uint16_t hpm(uint32_t k, size_t size, uint32_t m[size])
```

```
tmp := a
dst := 0
m := 0
k := 0
DO WHILE m < 64
        IF mask[m] == 1
                dst[k] := tmp[m]
                k := k + 1
        FI
        m := m + 1
OD
```

FIGURE 1 – L'instruction pext pour une entrée de 64 bits. Source : https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html

dont la valeur de retour h est exprimée par les relations suivantes :

— $h_0 := \texttt{pm(k, m[0])}$
— Pour $i \in [\![1, \texttt{size} - 1]\!]$, $h_i := \texttt{pm}(h_{i-1}, \texttt{m[i]})$
— $h := h_{\texttt{size}-1}$

1. Écrivez la fonction hpm.
2. Que se passe-t il si l'une des valeurs intermédiaires h_i ou l'un des *blocs* m[i] est égal à zéro ?
3. Expliquez pourquoi il est raisonnable de dire qu'hpm n'est pas une « bonne » famille de fonctions de hachage.