Travaux pratiques de programmation

On s'attend à ce que les candidats traitent de manière au moins partielle chacune des deux parties : programmation en C et OCaml d'une part, audit de code d'autre part.

1 Programmation en C et OCaml

Dans cette partie de l'épreuve, il est demandé au candidat de concevoir, écrire et tester un programme répondant aux questions ci-dessous, dans le langage C pour la section 1.2 et dans le langage OCaml pour la section 1.3. L'annexe à la fin de ce sujet donne des fonctions de bibliothèque pouvant être utiles.

On rappelle qu'il sera nécessaire, lors du rendu, de présenter le travail réalisé, les forces et faiblesses des approches adoptées, ainsi que d'expliciter et de commenter les jeux de tests utilisés.

1.1 Contexte : hiérarchie mémoire

La mémoire d'un ordinateur est un élément essentiel permettant de stocker autant les données que le code des différentes applications. La conception d'un composant mémoire repose sur un compromis entre taille de stockage, coût et vitesse d'accès aux données. À coût constant, pour obtenir un composant mémoire plus grand, il sera nécessaire de réduire sa vitesse. Les ordinateurs récents sont composés d'un grand nombre de composants mémoire, certains ayant une très faible capacité de stockage mais étant très rapides, d'autres avec une capacité de stockage plus grande mais plus lente.

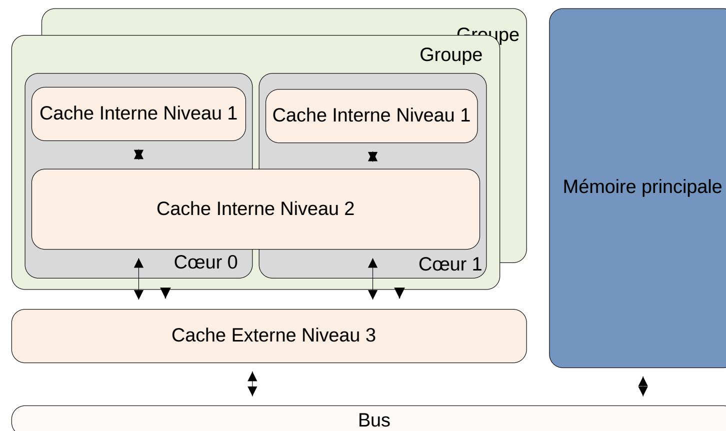

Figure 1 – Exemple de hiérarchie mémoire d'un processeur. Les cœurs sont souvent organisés en groupes partageant une mémoire appelée Cache L2 (cache interne niveau 2).

La figure 1 montre un exemple représentatif de cette structuration de la mémoire d'un processeur. Dans un processeur, l'unité de base de calcul est le cœur. Un processeur est souvent composé de plusieurs de ces cœurs. Pour des raisons de câblage électrique interne, les processeurs sont souvent structurés en groupes de cœurs partageant des éléments communs. La figure 1 montre par exemple un processeur composé de quatre cœurs répartis en deux groupes. Chacun

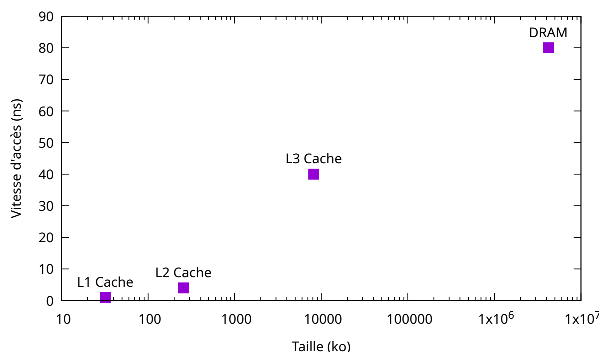

FIGURE 2 – Taille et vitesse d'accès des différents niveaux de mémoire d'un processeur typique de 2016. La taille est représentée en échelle logarithmique.

des cœurs possède une petite mémoire spécifique appelée mémoire Cache Interne de Niveau 1 (souvent abrégé en Cache L1). Chaque groupe de cœurs possède une mémoire légèrement plus lente que celle de niveau L1 mais plus grande, appelée Cache L2. Enfin, l'ensemble des cœurs du processeur partagent un cache encore plus grand mais encore plus lent appelé Cache L3. Ces caches de niveau L1, L2 et L3 sont à l'intérieur du composant électronique appelé processeur. La mémoire principale (souvent appelée DRAM) est, quant à elle, plus grande et composée de barrettes mémoire extérieures au processeur.

La figure 2 donne un exemple des tailles et vitesse d'accès de ces différents types de mémoire. Au vu de la différence de performance des différents types de mémoire, il est intéressant pour un programme de n'utiliser que les caches de niveau les plus bas. Il devient intéressant pour un programme de découper un travail sur une grande masse de données en plusieurs tâches, chacune travaillant sur une quantité de données rentrant dans un cache de bas niveau.

Dans la suite, nous allons tout d'abord évaluer la taille du cache L1, puis nous allons utiliser l'algorithme du crible d'Ératosthène segmenté pour illustrer l'importance des caches mémoire.

1.2 Détection de la taille des caches

Cette partie doit être réalisée en C.

Attention : le code C doit être compilé sans optimisation c'est-à-dire avec `gcc -O0`.

Dans cette partie, nous allons détecter la taille approximative du cache L1. Pour cela, nous allons accéder au contenu d'un tableau en mémoire (contenant des **`int`**) suivant un motif d'accès par saut. Les accès se feront par décalages successifs de 1024 entiers.

1.2.1 Temps d'une série d'accès en mémoire

Question 1. Écrire une fonction réalisant une série d'opérations en mémoire, de prototype suivant :

```
double get_time(unsigned int size, unsigned int steps)
```

Cette fonction commence par allouer un tableau d'entiers, de taille `size - 1`. L'argument `steps` est le nombre d'accès au tableau (en **millions d'accès**), chaque accès consistant en une lecture et une écriture. Chacun de ces accès sera décalé par rapport au précédent de 1024 éléments

(modulo la taille du tableau) par rapport au précédent. La valeur renvoyée est la durée totale des accès (donc de `steps*1000000` accès), exprimée en secondes (voir l'annexe pour la mesure).

1.2.2 Récupération d'une référence de temps

La précision d'une mesure dépend directement de la valeur de `steps`. Une trop faible valeur permet d'obtenir un résultat rapidement, mais peu précis car sujet à l'activité d'autres applications ou du système d'exploitation. Une valeur élevée permet de réduire l'impact de ces perturbations, mais au prix d'un temps plus long. On cherche ici à obtenir le paramètre `steps` pour que l'exécution de la fonction `get_time(1024, steps)` dure un certain nombre de secondes.

Question 2. Écrire une fonction

```
unsigned int find_steps(double duration)
```

qui détermine la plus petite valeur `steps` pour obtenir, avec la fonction `get_time`, une exécution ayant besoin d'au moins `duration` secondes lorsque `size` est égal à 1024.

1.2.3 Taille approximative du cache

Question 3. Écrire un programme qui détermine la taille approximative du cache L1. On explorera les tailles de cache en partant de 1024 entiers, puis en augmentant par tranches de 1024 entiers, jusqu'à arriver à une augmentation du temps de 20% par rapport au cas de base (*i.e.*, 1024 entiers).

Le programme affichera, pour chacun de ces tests, la taille du tableau (arrondie au ko le plus proche), ainsi que le temps associé. Cet affichage ira jusqu'à un tableau de la taille double de la taille déterminée.

Question 4. Afficher le graphe du temps en fonction de la taille du tableau.

1.3 Crible d'Ératosthène

Cette partie doit être réalisée en OCaml.

Les fichiers OCaml doivent impérativement être compilés avec `ocamlopt`*. On veillera à utiliser la commande*

```
ocamlopt source.ml -o /tmp/exec && mv /tmp/exec .
```

pour accélérer le temps de compilation.

Dans cette partie, on se propose d'étudier deux implémentations différentes du crible d'Ératosthène, afin de mettre en évidence l'intérêt de la localité des données. On commence par un crible d'Ératosthène traditionnel, avant de considérer un crible d'Ératosthène segmenté.

Tableaux de booléens. Les tableaux de booléens d'OCaml étant un peu gourmands en mémoire, on va plutôt se servir de chaînes de caractères mutables (du type `bytes`) pour réaliser des tableaux de booléens, avec la convention que le caractère `'0'` représente faux et le caractère `'1'` représente vrai. Pour une chaîne `b` de type `bytes`, on accède à sa longueur avec `Bytes.length b` et à son caractère d'indice i avec `Bytes.get b` i. On modifie le caractère d'indice i avec `Bytes.set b` i c, où c est de type `char`. Enfin, on construit une chaîne mutable de longueur n et dont tous les caractères sont égaux à c avec `Bytes.make` n c.

Question 5. Écrire une fonction `clear (b: bytes) (i: int) (d: int) : int` qui met à `'0'` tous les éléments de la chaîne `b` à partir de l'indice `i` et en procédant de `d` en `d`. La valeur renvoyée est le premier indice (de la forme `i`$+n$`d`) qui dépasse la capacité de `b`. On suppose `i` ≥ 0 et `d` > 0.

Crible d'Ératosthène. On rappelle le principe du crible d'Ératosthène. Pour déterminer les nombres premiers $p \leq N$:

1. On se donne un tableau de booléens B de taille $N+1$, initialisé avec faux dans les cases 0 et 1 et vrai dans toutes les autres cases.
2. Pour p allant de 2 à N,
   si $B[p]$ est vrai, alors p est premier et on met à faux tous les $B[p^2 + ip]$ avec $i \geq 0$.

Ici, le tableau B sera implémenté par une chaîne mutable de type `bytes`, comme expliqué plus haut.

Question 6. Écrire une fonction `sieve (n: int) : int * bytes` qui réalise un crible d'Ératosthène pour déterminer les nombres premiers $p \leq$ `n` et renvoie leur nombre ainsi qu'une chaîne `b` de taille `n`$+1$ indiquant, pour tout entier $0 \leq i \leq$ `n`, s'il est premier. On se servira de la fonction `clear` de la question 5. On pourra tester en vérifiant qu'il y a 25 nombres premiers inférieurs ou égaux à 100 et 168 nombres premiers inférieurs ou égaux à 1000.

Question 7. Déduire de la fonction `sieve` une fonction `primes (n: int) : int array` qui renvoie le tableau des nombres premiers $p \leq$ `n`, dans l'ordre croissant. Ainsi, `primes 10` doit renvoyer le tableau `[| 2; 3; 5; 7 |]`.

Crible d'Ératosthène segmenté. L'idée du crible segmenté consiste à découper le tableau de booléens en segments d'une certaine taille S — un paramètre de l'algorithme — et à effectuer le crible segment par segment. Pour déterminer tous les nombres premiers inférieurs ou égaux à N, le paramètre S doit vérifier la condition $N \leq S^2$.

1. On calcule le tableau `primes` des nombres premiers $p \leq \lfloor\sqrt{N}\rfloor$.
2. On alloue un tableau B de booléens de taille S.
3. Pour $k = 0, \ldots, \lfloor N/S \rfloor$,
   (a) On pose `lo` $\stackrel{\text{def}}{=} kS$ et `hi` $\stackrel{\text{def}}{=} \min((k+1)S - 1, N)$ pour délimiter le segment k.
   (b) On remplit B avec la valeur vrai.
   (c) Pour chaque nombre premier p de `primes`,
   mettre $B[n - \texttt{lo}]$ à faux si n est un multiple de p.

Pour implémenter efficacement cet algorithme, on conserve, pour chaque nombre premier p, le prochain multiple à considérer, ou, plus précisément, sa différence avec la borne gauche `lo` du prochain segment. On se donne pour cela un tableau d'entiers `ofs` de la même taille que le tableau `primes`. Le premier multiple de p à considérer est p^2, comme dans le crible traditionnel. Cela signifie que l'étape 3c ci-dessus ne considère que les nombres p avec $p^2 \leq$ `hi`. On conserve donc dans une variable `next` le nombre de nombres premiers qui doivent être considérés.

La figure 3 illustre le crible segmenté avec $N = 90$ et des segments de taille $S = 20$. Il y a donc 5 segments au total. On a $\lfloor\sqrt{90}\rfloor = 9$ et on commence donc par calculer le tableau `primes` des nombres premiers ≤ 9, soit $2, 3, 5, 7$. Le premier segment ($k = 0$) examine les nombres allant de `lo` $= 0$ à `hi` $= 19$. Comme $3^2 \leq$ `hi` et $5^2 >$ `hi`, on ne considère que les `next` $= 2$ premiers nombres premiers, dont les décalages initiaux sont $2^2 = 4$ et $3^2 = 9$. À l'issue de ce premier crible, on a déterminé 8 nombres premiers, le décalage de 2 vaut 0 (on a terminé sur 20) et le décalage de 3 vaut 1 (on a terminé sur 21). On considère alors le deuxième segment ($k = 1$), qui examine les nombres allant de `lo` $= 20$ à `hi` $= 39$. On considère maintenant `next` $= 3$ nombres premiers, car $5^2 \leq$ `hi` et $7^2 >$ `hi`. Le décalage initial pour 5 est $5^2 -$ `lo` $= 25 - 20 = 5$. Cette fois on a déterminé 4 nombres premiers, le décalage de 2 vaut 0 (on a terminé sur 40), le décalage de 3 vaut 2 (on a terminé sur 42) et le décalage de 5 vaut 0 (on a terminé sur 40). Et ainsi de suite pour les trois autres segments. On note que le dernier segment est incomplet (on ne va pas plus loin que `hi` $= N = 90$). On prendra le temps de bien lire et comprendre la figure 3. Il est fortement suggéré de faire tourner intégralement l'algorithme, en barrant soi-même les cases sur la figure 3.

Question 8. Écrire une fonction

```
find_next (primes: int array) (ofs: int array)
          (lo: int) (hi: int) (next: int) : int
```

qui met à jour le tableau des décalages `ofs`. Les paramètres `lo` et `hi` définissent les bornes du prochain segment. Le paramètre `next` est la valeur actuelle de `next` et la valeur renvoyée est la nouvelle valeur de `next`.

Si on reprend l'exemple de la figure 3, le premier appel à cette fonction est `find_next primes ofs 0 19 0`, qui renvoie 2, et le deuxième appel est `find_next primes ofs 20 39 2`, qui renvoie 3.

Question 9. Écrire une fonction `segmented_sieve (size: int) (n: int) : int` qui implémente le crible segmenté. Les deux paramètres sont S et N, respectivement. On pourra supposer que la condition $N \leq S^2$ est satisfaite. La valeur renvoyée est le nombre total de nombres premiers $p \leq N$.

Pour un entier n, on pourra calculer $\lfloor\sqrt{n}\rfloor$ avec la fonction `isqrt` suivante :

```
let isqrt (n: int) : int =
  truncate (sqrt (float n))
```

Question 10. Déterminer empiriquement une bonne valeur pour le paramètre S. On pourra fixer $N = 10^8$ ou $N = 10^9$ pour cette expérience.

Question 11. Comparer les performances du crible traditionnel (fonction `sieve` de la question 6) et du crible segmenté (fonction `segmented_sieve` de la question 9), au regard du temps d'exécution et de l'espace mémoire utilisé. On pourra mesurer la mémoire utilisée en affichant la valeur donnée par `Gc.allocated_bytes ()` à la fin du programme.

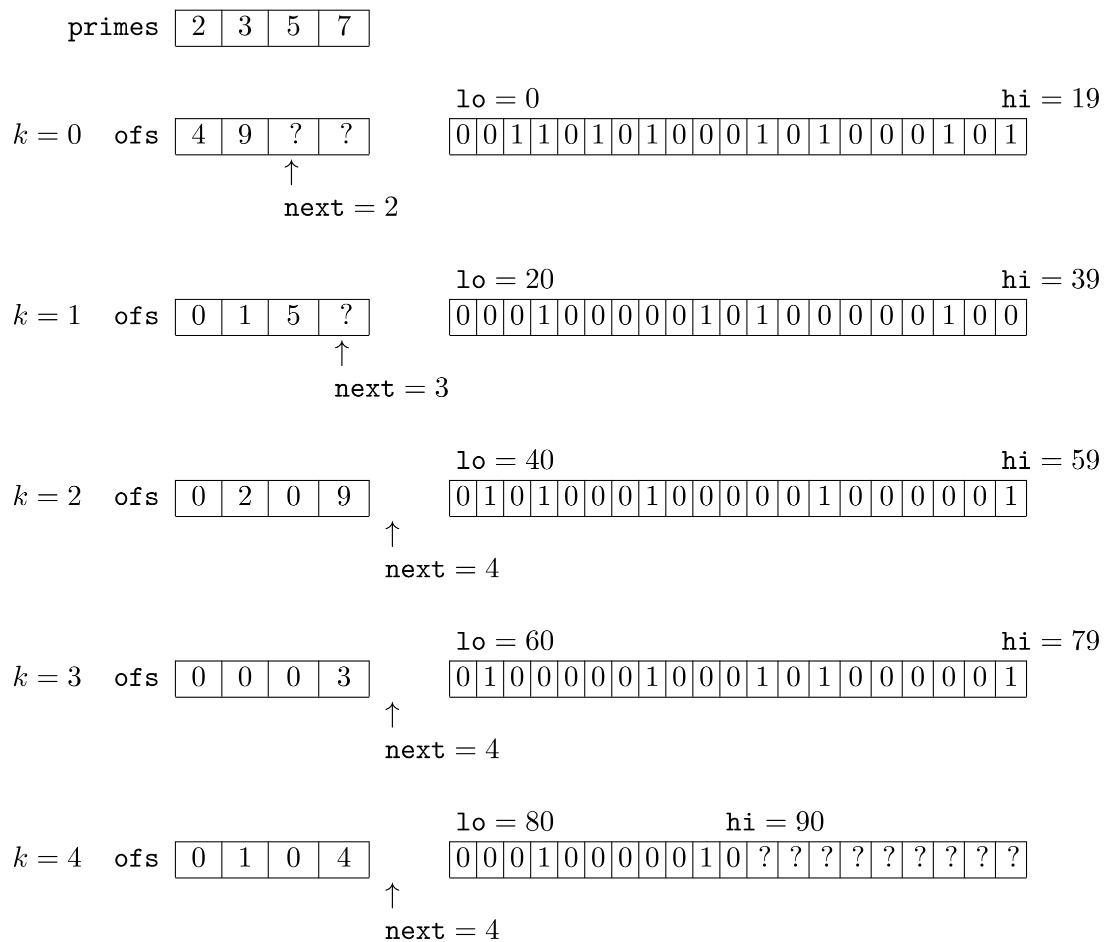

Figure 3 – Crible d'Ératosthène segmenté avec $N = 90$ et $S = 20$.

2 Audit de code C et OCaml

Dans cette partie de l'épreuve, il est demandé au candidat d'étudier un fichier source qui comporte des erreurs ou des maladresses, de qualité de code ou de fonctionnement, et il est demandé d'auditer ce fichier, c'est-à-dire :

- de comprendre et d'être capable d'expliquer le fonctionnement du code à l'oral ;
- de proposer des corrections en réécrivant certaines parties afin de corriger les erreurs ou maladresses éventuelles et de rendre le code plus clair, notamment dans une optique pédagogique ;
- de proposer des améliorations de la complexité en temps ou en mémoire, ou de la sûreté du code.

Le temps indicatif de préparation de cette partie est d'une heure. Le code ci-dessous est disponible dans votre compte, à la racine, sous les noms `~/audit1.ml` et `~/audit2.c` respectivement.

2.1 Code en OCaml

```
1  let min3 x y z =
2    if x < y then if y < z then x else if x < z then x else z
3             else if x < z then y else if y < z then y else z
4
5  let rec edit_distance a b =
6    if String.length a = 0 then String.length b
7    else if String.length b = 0 then String.length a
8    else min3 (edit_distance (String.sub a 0 (String.length a-1)) b + 1)
9           (edit_distance a (String.sub b 0 (String.length b-1)) + 1)
10          (edit_distance (String.sub a 0 (String.length a-1))
11             (String.sub b 0 (String.length b-1))
12           + if a.[String.length a-1] = b.[String.length b-1] then 0 else 1)
```

2.2 Code en C

```
1  #include <string.h>
2
3  int aux1(int x, int y, int z) {
4    if (y < x) x = y;
5    if (z < x) x = z;
6    return x;
7  }
8
9  int aux2(char *s, char *t) {
10   int d;
11   if (*s == 0) {
12     d = strlen(t);
13   } else if (*t == 0) {
14     d = strlen(s);
15   } else {
16     d = aux1(aux2(s+1, t  ) + 1,
17              aux2(s  , t+1) + 1,
18              aux2(s+1, t+1) + *s == *t ? 0 : 1);
```

```
19     }
20     return d;
21  }
```

On rappelle que l'expression e_1 `?` e_2 `:` e_3 évalue e_1, puis renvoie la valeur de e_2 si e_1 est vraie, et la valeur de e_3 sinon.

3 Annexe : bibliothèque C pour mesurer le temps d'exécution

```
#include <time.h>

double clock(void);
  // renvoie la durée d'utilisation du processeur ;
  // pour obtenir des secondes, diviser la valeur renvoyée
  // par CLOCK_PER_SEC
```

Pour mesurer une durée en seconde il est donc nécessaire d'écrire :

```
double start;
double finish;
start = clock();
/*
  Code à mesurer
*/
finish = clock();
double duration = (finish-start)/CLOCKS_PER_SEC;
```

* *
*