Examen d'Architecture et Système

13 janvier 2017

Durée : 2 heures. Vous avez droit à deux feuilles de papier avec vos notes personnelles.

1 Virgule flottante

On s'intéresse de nouveau à la fractale Mandelbrot qui associe à chaque pixel dans une image une coordonnée (x, y) pour y peintre une couleur qui est une fonction de (x, y). Supposons qu'on s'intéresse à zoomer sur un point (x_0, y_0) spécifique. Pour $\epsilon := 0.25$, on regarde donc une série d'images de 1024 x 1024 pixels où l'image numéro i représente l'interval $[x_0 - \epsilon^i, x_0 + \epsilon^i] \times [y_0 - \epsilon^i, y_0 + \epsilon^i]$. Après un certain nombre d'itérations, l'image dévient floue : la précision de la représentation virgule flottante ne suffit plus pour distinguer 1024 valeurs différentes dans la largeur ou la hauteur ; du coup certains pixels seront associés aux mêmes coordonnées. Supposons d'abord que le calcul se fait avec le type `float`. Pour simplifier, on s'intéresse uniquement à la largeur, on fixe donc $x_0 = 0.375$ pour un y_0 quelconque.

Les formats de virgule flottante consistent de trois composants, le *signe*, l'*exposant* et la *mantisse*. Dans ce qui suit le signe sera toujours positive (0). Pour le type `float`, l'exposant possède 8 bits et la mantisse 23. Du coup la représentation binaire de l'exposant sera un entier entre 0 et 255, et on soustrait 127 pour obtenir l'exposant à utiliser. (Les cas spéciaux comme 0, ∞, NaN seront sans conséquence dans ce qui suit.) Dans la mantisse, le bit le plus haut représente 0.5, et il faudra toujours rajouter 1. On considère la représentation binaire suivant d'un `float` (les points indiquent simplement la limite entre signe, exposant, mantisse) :

```
0.011 1110 1.100 0000 0000 0000 0000 0000
```

Avec les explications ci-dessus, ceci est la représentation de $2^{125-127} \cdot (1 + 0.5) = 0.375$.

(a) Donnez la représentation binaire de ϵ et de ϵ^2. (Vous pouvez raccourcir les zéros qui traînent à la fin.)

Solution: (2p) On a $\epsilon = 0.25 = 2^{-2} = 2^{125-127} \cdot (1 + 0)$. Du coup, l'exposant sera représenté par la valeur 127 et la mantisse sera 0. Pour $\epsilon^2 = 2^{-4}$, l'exposant est minoré par 2. Les bonnes réponses sont donc :

```
0.011 1110 1.000 0000 0000 0000 0000 0000
0.011 1101 1.000 0000 0000 0000 0000 0000
```

(b) Donnez la représentation binaire de $x_0 + \epsilon^2$ et de $x_0 + \epsilon^3$. (Attention à choisir le bon exposant dans chaque cas.)

Solution: (2p) On peut soit appliquer l'algorithme pour l'addition virgule flottante ou d'abord calculer le resultat et puis le convertir vers la représentation binaire.

Dans le premier cas, on observe que pour x_0 et ϵ^2, les exposants sont de 125 et 123, respectivement, avec des mantisses tout zéro. On garde l'exposant le plus grand et on décale à droite la mantisse du terme le plus petit. En prenant compte du bit de 1 implicit cela donne les mantisses :

```
100 0000 0000 0000 0000 0000
010 0000 0000 0000 0000 0000
```

et du coup le résultat pour $x_0 + \epsilon^2$ est de :

```
0.011 1110 1.110 0000 0000 0000 0000 0000
```

Pour $x_0 + \epsilon^3$ on obtient, par analogie :

```
0.011 1110 1.100 1000 0000 0000 0000 0000
```

Ou bien, le calcul mat calcul mathématique

$$x_0 + \epsilon^2 = 2^{-2} \cdot (1 + {}^1\!/_2) + 2^{-4} \cdot 1 = 2^{-2} \cdot (1 + {}^1\!/_2 + {}^1\!/_4)$$

$$x_0 + \epsilon^3 = 2^{-2} \cdot (1 + {}^1\!/_2) + 2^{-6} \cdot 1 = 2^{-2} \cdot (1 + {}^1\!/_2 + {}^1\!/_{16})$$

ce qui donne la même solution.

(c) Quelle est la représentation de $x_0 - \epsilon^3$, sachant que $(x_0 + \epsilon^i) + (x_0 - \epsilon^i) = 2x_0$?

Solution: (1p) Solution par calcul direct :

$x_0 - \epsilon^3 = 2^{-2} \cdot (1 + {}^1\!/_2) - 2^{-6} = 2^{-6} \cdot (16 + 8 - 1) = 2^{-6} \cdot (16 + 4 + 2 + 1) = 2^{-2}(1 + {}^1\!/_4 + {}^1\!/_8 + {}^1\!/_{16})$

```
0.011 1110 1.011 1000 0000 0000 0000 0000
```

Un autre façon d'arriver au résultat est d'utiliser l'égalité donnée dans l'énoncé : les exposants de x_0 et de $x_0 + \epsilon^3$ sont égaux, et les quatre premier bits de leurs mantisses sont $(1000)_2 = 8$ et $(1001)_2 = 9$, respectivement. Étant donné que $9 + 7 = 2 \cdot 8$, les quatre premiers bits de $x_0 - \epsilon^3$ doivent être $7 = (0111)_2$.

(d) Combien de valeurs différentes peut-on représenter dans l'intervalle $[x_0 - \epsilon^3, x_0 + \epsilon^3]$? (Une réponse de la forme 2^n suffit.)

Solution: (2p) Les exposants de $x_0 - \epsilon^3$ et $x_0 + \epsilon^3$ étant égaux, toutes les valeurs dans l'intervalle doivent le partager aussi. Du coup, la solution revient à prendre la différence entière entre les deux mantisses, à savoir $9 \cdot 2^{19} - 7 \cdot 2^{19} = 2^{20}$ (éventuellement rajouter 1 selon l'interprétation de la taille d'un intervalle, mais ça ne change rien pour la suite).

(e) Quelle sera la valeur maximale de i telle que la précision suffit pour représenter $1024 = 2^{10}$ valeurs différentes dans l'image numéro i ? Et si le calcul se faisait avec le type `double`, où l'exposant possède 11 bits et la mantisse 52 ?

Solution: (3p) Les valeurs $x_0 \pm \epsilon^i$ resteront sur le même exposant pour tout $i \geq 3$, du coup l'intervalle sera simplement réduit par un facteur de $\epsilon = 2^{-2}$ lors de chaque itération. Après cinq itérations supplémentaires, la taille de 2^{20} se réduit à 2^{10}, la bonne réponse est donc $i = 8$.

Le type double fournit 29 bits de mantisse de plus et permet donc 14 itérations supplémentaires, du coup la réponse est $i = 22$.

2 Entrée/sortie

(a) On considère le programmes programmes P1 et P2 ci-dessous. Quel est le résultat affiché sur l'écran si stdout est *line-buffered* comme d'habitude ? Rappel: 1 représente la sortie standard pour `write`.

```
int main ()
{
    printf("Bonjour");
    fork();
    printf("\n");
}
```

```
int main ()
{
    write(1,"Bonjour",7);
    fork();
    write(1,"\n",1);
}
```

Figure 1: Programmes P1 et P2.

Solution: (2p) Pour P1, si stdout est line-buffered, alors printf sans nouvelle ligne récopie le texte `Bonjour` vers un buffer. Puis, fork produit un deuxième processus avec le même état de mémoire. Du coup, lorsque chaque processus effectue son deuxième printf, le résultat affiché sur l'écran sera le contenu du buffer suivi par la nouvelle ligne, et le résultat est

```
Bonjour
Bonjour
```

Par contre, les write dans P2 débouchent directement sur la sortie standard. On voit donc apparaître `Bonjour` une fois puis deux nouvelles lignes.

(b) Considérons le programme P3 ci-dessous qui a été compilé vers un fichier exécutable `p3`. Rappels : 1 est la sortie standard, 2 est la sortie erreur. `dup2(a,b)` donne connecte `b` à la même sortie que `a`.

(i) Quelle est la sortie de la commande `./p3` ?

Solution: (3p) Le programme mélange les appels ANSI (avec streams) et POSIX (avec descripteurs). Le stream stdout est branché sur le descripteur 1, et le stream stderr sur le descripteur 2. La bufferisation détermine comment les printf se traduisent vers des appels write.

Puisque tout printf est suivi par une nouvelle ligne, les textes seront directement envoyés vers stdout. Il en est de même pour tout fprintf et stderr.[1]En plus, au départ, les deux sorties sont branchés sur le terminal. `trois` disparaît car le descripteur 1 a été fermé avant. Par contre, avant `cinq` l'appel dup2 fait rebrancher 1 au terminal qui est le débouché de 2. L'appel suivant de close(2) n'affecte que stderr. Du coup, toutes les lignes sauf `trois` et `huit` arrivent sur l'écran.

(ii) Quelle est la sortie de la commande `./p3 2> /dev/null` ?

Solution: (1p) Les affichages sur stderr disparaissent car ils vont vers /dev/null. Et `cinq` et `sept` disparaissent car après dup2, la sortie standard y est branchée aussi. `trois` disparaît toujours pour la même raison que dans la première partie. Du coup il n'en reste qu'`un`.

(iii) Question bonus : Si on lance `./p3 > /dev/null`, la sortie est `deux quatre six un trois cinq sept` (en omettant les retours à la ligne). Comment expliquer un tel résultat ?

Solution: Cette question donnait droit à 3 points maximum, mais n'était pas incluse dans les 100 %. En effet, une réponse n'était pas exigée car elle répose sur certains effets discuté assez brièvement dans le cours. Néanmoins, il est possible de conclure certains faits à partir du résultat observé :

Le fait de voir les nombres impairs arrivent sur l'écran ensemble indique qu'il s'agit d'un seul write qui reprend le résultat de plusieurs printf. On en conclut que ces printf ont été bufferisés. Sous cette hypothèse, leur affichage serait le résultat du vidange du buffer à la fin du programme, et à ce moment le descripteur 1 est branché sur l'écran.

Pourquoi donc cette bufferisation ? Il s'avère que lorsque la sortie standard n'est pas branché au terminal, le stream stdout devient plainement bufferisé au lieu de line-buffered (avec, dans ce cas, un buffer assez grand pour retenir toutes les printf).

[1]Techniquement, il s'avère que stderr est non-bufferisé par défaut. Mais ça revient à la même chose : tout fprintf sur stderr donne lieu à un write immédiatement.

```
printf("un\n");                 printf("cinq\n");
fprintf(stderr,"deux\n");       fprintf(stderr,"six\n");
close(1);                       close(2);
printf("trois\n");              printf("sept\n");
fprintf(stderr,"quatre\n");     fprintf(stderr,"huit\n");
dup2(2,1);
```

Figure 2: Contenu de `main` pour le programme P3.

3 Circuits

Soient $x_1, \ldots, x_n \in \{0, 1\}$ des signaux en entrée. On s'intéresse à calculer la somme modulo 3 des entrées. Plus précisement, on veut un circuit avec trois sorties u_0, u_1, u_2 tels que $u_i = 1$ ssi $\sum_{j=1}^{n} x_j = i \pmod 3$.

(a) Construisez un tel circuit pour $n = 2$. Vous avez droit à n'importe quelle porte logique avec au plus deux entrées (donc OU, ET, NON-OU, XOR, EQV, ...). Toute notation raisonnable est admise.

Solution: (2p) u_0, u_1, u_2 sont les résultats des portes NON-OU, XOR, ET, respectivement, sur x_1 et x_2.

Construisons maintenant une solution par dichotomie: on coupe l'ensemble d'entrées en deux (p.ex. $x_1, \ldots, x_n$ d'une part et $x_{n+1}, \ldots, x_{2n}$ d'autre part), et on suppose l'existence de deux circuits qui produisent les bonnes triplets de sorties pour ces deux ensembles ; on les nomme y_0, y_1, y_2 respectivement z_0, z_1, z_2.

(b) Construisez un circuit qui prend en entrée les six signaux $y_0, y_1, y_2, z_0, z_1, z_2$ et qui produit trois sorties u_0, u_1, u_2 tels que $u_i = 1$ ssi $\sum_{j=1}^{2n} x_j = i \pmod 3$.

Note 1: y_0, y_1, y_2 sont mutuellement exclusifs (pareil pour les z_i). Du coup il y a effectivement 9 combinaisons d'entrées, dont trois paires symétriques.

Note 2: Un bonus sera accordé aux solutions n'utilisant que des OU et des EQV.

Solution: 3 points pour n'importe quelle solution correcte, et un bonus de 2 points pour trouver la solution suivante :

Soit $v_i := y_i \vee z_i$ pour $i = 0, 1, 2$. Pour avoir $u_0 = 1$ il faut l'une de trois combinaisons (i) $y_0 \wedge y_0$, (ii) $y_1 \wedge z_2$, (iii) $y_2 \wedge z_1$. Or (i) est le seul cas qui donne $v_1 = v_2 = 0$ et (ii) et (iii) sont les seuls qui donnent $v_1 = v_2 = 1$.

Par analogie on arrive à $u_0 = v_1 \equiv v_2$, $u_1 = v_0 \equiv v2$ et $u_2 = v_0 \equiv v_1$. Un circuit réalisant cette fonction utilise donc trois portes OU et trois portes EQV.

(c) Quelle est la taille et la profondeur d'un tel circuit pour un n donnée ? (Un réponse genre $\mathcal{O}(f(n))$ suffit.)

Solution: (2p) Les circuits de (a) et (b) ne dépendent pas de n, ils sont de taille et profondeur constante. Pour agréger n signaux, on a besoin de $n-1$ tels circuits. Du coup la taille est de $\mathcal{O}(n)$. Par ailleurs, par dichotomie, on peut arranger les circuits de (b) pour faire un circuit de profondeur logarithmique, c'est à dire $\mathcal{O}(\log n)$.

4 Programmation concurrente

On possède n threads, chacun avec un identifiant i. L'action principal de chaque thread est d'afficher son identifiant sur l'écran une fois. On veut que les identifiants apparaissent dans l'ordre croissant.

(a) Écrivez un programme (en pseudo-code) qui réalise cette spécification avec $n = 3$ à l'aide des sémaphores (opérations `init`, `wait`, `post`).

Solution: (2p) Question cadeau pour ceux qui ont suivi les TD : On utilise deux sémaphores a, b, initialement avec zéro créneaux. Le premier thread fait son travail puis libère un créneau sur a. Le deuxième thread attend ce créneaux, fait son travail, puis libère un créneau sur b. Le troisième thread attend le créneau de b.

Une solution généralisée pour n threads et $n-1$ sémaphores. Dans main on initialise les sémaphores et lance les threads:

```
pthread_t th[N];
sem_t s[N-1];

int main ()
{
  for (i = 0; i < N-1; i++) sem_init(s+i,0,0);
  for (i = 0; i < N; i++)
    pthread_create(th+i,NULL,thread,(void*)i);
}
```

Dans un thread i on attend le thread précédent $(i-1)$ et signale au thread prochain en libérant le sémaphore i.

```
void* thread (void *arg)
{
  long int id = (long int) arg;
  if (id > 0) sem_wait(s+id-1);
  printf("%ld\n",id);
  if (id < N-1) sem_post(s+id);
}
```

(b) Soit $n = 2^k$ pour un k quelconque. Décrivez comment réaliser la tâche avec seulement $O(k)$ sémaphores.

Solution: Peut-être cette question était trop difficile, ou qu'il manquait du temps, car vous étiez peu à aborder cette question, et aucune solution n'était entièrement correcte. J'ai compté 2 points pour les 100 % ce qui était le maximum atteint.

L'idée de départ est bien sûr que chaque thread attend des threads en correspondance avec l'écriture binaire de son identifiant. On va supposer que les identifiants des threads sont de 0 à $n-1$, ceux des sémaphores de 0 à $k-1$, et qu'un thread attend les sémaphores qui correspond aux uns dans son écriture binaire, en commençant par le bit le plus significatif. Du coup le thread numéro $40 = (101000)_2$ attend le sémaphore numéro 5, puis numéro 3.

Suivant ce principe, si les sémaphores sont tous initialisés à 0, alors la moitié des threads est bloqué au sémaphore d'index maximum, la moitié des autres au prochain et de suite, et seulement le thread numéro 0 est libre de procéder.

Supposons qu'un thread vient d'afficher son identifiant et que son identifiant est de $id = (x \cdots x01 \cdots 1)_2$ (pour des x quelconque), avec m uns à la fin. Le prochain thread à libérer est $id + 1 = (x \cdots x10 \cdots 0)_2$ qui est bloqué sur le sémaphore numéro m. Or, il y a 2^m threads on total qui sont bloqués sur ce même sémaphore. Il convient de les libérer tous ; néanmoins seul le thread $id + 1$ sera complètement libre de continuer.

Le code d'un thread devient ainsi comme suit, en supposant que *multipost*(s, n) libère n fois le sémaphore s.

```
sem_t s[K-1]; // initialisé à 0
void* thread (void *arg)
{
  long int i, id = (long int) arg;
  for (i = K-1; i >= 0; i--)
    if (id & (1<<i)) sem_wait(s+i);
  printf("%ld\n",id);
  for (i = 0; id & (1<<i); i++);
  if (i < K) multipost(s+i,1<<i);
}
```

(c) Pouvez-vous réaliser la tâche si l'affichage est cyclique ?

Solution: Cette question ne comptait finalement pas pour les 100 %.

Disons qu'un *cycle* consiste en laissant tous les threads faire leur travail. Du coup on veut que les cycles se répètent, et le défi est de signaler à tous les threads qu'ils peuvent recommencer.

Introduisons deux sémaphores q_1 et r_1, initialisés à 0. Supposons d'abord que chaque thread, ayant affiché sont identifiant, libère un créneau sur q_1 pour signaler ce fait à

`main` et attend ensuite sur r_1. De son côté `main` attend n fois sur q_1 et libère ensuite n créneaux sur r_1.

Cette solution paraît naturelle mais présente un inconvénient : le thread 0 pourrait prendre un créneau de r_1, faire son travail et toute de suite grignoter encore un créneau de r_1 au détriment d'un autre thread. La solution est d'enfermer les threads dans un état intermédiaire : on rajoute des sémaphores q_2, r_2 et répète la procédure décrit ci-dessus avec ces deux. Ceci assure que chaque thread aura pris son créneau de r_1 avant que le premier thread recommence.

Le code pour les parties intéressantes :

```
sem_t q1,q2,r1,r2;

void* thread (void *arg)
{
  while (1)
  {
    ... // le même comportement que pour 4b.
    sem_post(&q1); sem_wait(&r1);
    sem_post(&q2); sem_wait(&r2);
  }
}

int main ()
{
  // initialiser tous les sémaphores à 0
  // lancer les threads
  while (1)
  {
    for (i = 0; i < N; i++) sem_wait(&q1);
    multipost(&r1,N);
    for (i = 0; i < N; i++) sem_wait(&q2);
    multipost(&r2,N);
  }
}
```