# Examen d'Architecture et Système

18 janvier 2019

Durée : 2 heures.

*Remarques sur la notation : Les Questions 1 à 3 comptaient quasiment à égalité, la Question 4, abordée par peu de monde, comptait en bonus (ainsi que la 3f).*

## 1 Virgule flottante

La représentation par virgule flottante comporte trois composants:

- le signe $s$, un seul bit (1 indique une valeur négative) ;
- l'exposant $e$ avec $k$ bits ;
- la mantisse $m$ avec $\ell$ bits.

La valeur absolue représentée ainsi est de $2^e \cdot m$. Comme dans le standard IEEE-754, on va supposer que la représentation binaire de l'exposant, interpretée comme un entier sans signe, est majorée de $2^{k-1} - 1$, p.ex. si $k = 4$ alors 1001 représente $e = 2$. Les exposants autorisés sont entre $-2^{k-1} + 2$ et $2^{k-1} - 1$. On représente 0 en mettant tous les bits de l'exposant à 0, et $\pm\infty$ en mettant tous les bits de l'exposant à 1. D'ailleurs, la mantisse sera toujours dans le domaine $[1, 2)$, et le bit le plus signaficatif possède le poids $1/2$. Du coup, si $\ell = 3$ et les bits de la mantisse sont 101, alors la mantisse vaut $1 + 1/2 + 1/8$.

(a) Supposons $k = 4$ et $\ell = 5$. Quelles sont les représentations les plus proches de 12, 3.1415 et $-34.5$ ?

**Solution:** La majoration de l'exposant est 7 pour $k = 4$.

- $12 = 8 + 4 = 2^3 \cdot (1 + \frac{1}{2})$, donc 0.1010.10000 avec $((1010)_2 = 10 = 3 + 7)$.
- $3.1415 \approx 2 + 1 + \frac{1}{8} = 3.125$, donc 0.1000.10010
- $-34.5 = -(100010.1)_2$ possède 7 bits significatifs, du coup le dernier bit n'est plus représentable, les solutions possibles sont donc soit 1.1100.00010 soit 1.1100.00011.

(b) Encore pour $k = 4$ et $\ell = 5$, quel est le plus grand entier qu'on peut représenter sans perte de précision ? Quel est le plus petit entier positif qui n'a pas de représentation exacte ? (pour la première question, une formule suffit)

**Solution:** (3+2 points) les exposants autorisés pour $k = 4$ sont entre -6 et 7. Il y avait une erreur dans l'énoncé par rapport à ces limites, corrigée ci-dessus. Logiquement, ceux qui utilisaient la fausse limite supérieur n'ont pas été pénalisés. Avec l'exposant maximal et la mantisse maximale on obtient $2^7 \cdot (1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \frac{1}{32}) = 2^8 - 4 = 252$.

Parmi les plus petits entiers positifs avec 7 bits significatifs, $64 = 2^6$ possède une représentation exacte (1.1101.00000), mais pour $65 = (1000001)_2$, la mantisse ne peut plus représenter le dernier bit. Tout entier entre 1 et 63 possède au plus six bits significatifs dont les cinq derniers rentrent dans la mantisse.

(c) Expliquer la multiplication de deux virgules flottantes. Donnez un algorithme en pseudocode qui calcule $z = x \cdot y$ si les composants de $x$ et $y$ sont des entiers sans signe dénommés $x.s$, $x.m$, $x.e$ etc.

**Solution:** Pour avoir 100% des points, il fallait expliquer que:

- $z.s$ et l'ou exclusif (somme modulo 2) de $x.s$ et $y.s$ ;
- $z.e$ est la somme de $x.e$ et $y.e$, en corrigeant pour la majoration.
- $z.m$ résulte du produit de $x.m$ et $y.m$ (tenir compte que la 1 n'est pas représentée explicitement).

Ce qui donne (encore pour $k = 4$ et $\ell = 5$) :

```
1:  z.s = x.s ^ y.s
2:  z.e = x.e + y.e - 7
3.  z.m = (32+x.e) * (32+y.e)
4:  if (z.m & 2048) { z.e++; z.m = z.m >> 1 }
5:  z.m = (z >> 5) & 31;
```

Bonus pour traiter les cas spéciaux (zéro, débordements). Le code sur la page web traite aussi l'arrondi, mais ceci n'était pas du tout attendu.

# 2 Threads et sémaphores

On considère les programmes P1 et P2 dans la Figure 1 avec une fonction `main` qui crée des threads. Par rapport au vrais appels système, la syntaxe est simplifiée, notamment on passe un ou deux paramètres entiers directement à un thread.

```
1:  int x;

2:  void f (int y) {
3:    ...
4:  }

5:  void g (int y) {
6:    ...
7:  }

8:  int main(int argc,char **argv) {
9:    pthread_t t1,t2;
10:   int x1 = atoi(argv[1]);
11:   int x2 = atoi(argv[2]);
12:   assert(x1 != x2);
13:   pthread_create(&t1,f,x1);
14:   pthread_create(&t2,g,x2);
15:   pthread_join(t1);
16:   pthread_join(t2);
17: }
```

```
1:  int x;
2:  #define N 16

3:  void f (int id, int score) {
4:    printf("id=%d\n",id);
5:  }

6:  int main() {
7:    pthread_t t[N};
8 :   int i,p1[N],p2[N];

9:    // on fait de p1 et p2 deux
          permutations aléatoires
          de 0 à N-1

10:   for (i=0; i<N; i++)
11:     pthread_create(&t[i],f,p1[i],p2[i]);
12:   for (i=0; i<N; i++)
13:     pthread_join(t[i]);
14: }
```

Figure 1: Programmes P1 (à gauche) and P2 (à droite).

Dans P1, `main` accepte deux entiers sur la ligne de commande et les passe aux threads. Il existe une variable globale `x`. L'objectif pour `f` et `g` est de comparer leurs paramètres. Si `x1` est plus grand que `x2`, alors `f` est censé écrire `f gagne` (analogue pour `g`).

(a) Complétez P1, en respectant les contraintes suivantes :

(C1) Les threads peuvent se coordonner en utilisant une seule variable globale `x` ainsi qu'un nombre illimité de sémaphores.

(C2) L'écriture concurrente sur `x` est interdite, mais la lecture concurrente sur `x` est permise.

(C3) On utilisera `init(s,`$n$`)`, `wait(s)` et `post(s)` pour les opérations sur les sémaphores, mais seul `main` peut appeller `init`, et ça une seule fois par sémaphore.

**Solution:** Dans la solution suivante, `f` écrit d'abort, puis `g`. On utilise deux sémaphores que `main` initialisera à 0.

```
void f (int y) {
  x = y;
  post(first);
  wait(second);
  other = x;
  if (other < y)
    printf("f gagne")
}
```

```
void g (int y) {
  wait(first);
  other = x;
  x = y;
  post(second);
  if (other < y)
    printf("g gagne")
}
```

On tourne maintenant à P2 où `main` crée deux permutations (le code pour générer celles-ci est omis). Ensuite, on génère `N=16` threads.

(b) Dans l'état actuel, les affichages `id=0`, `id=1` etc. apparaissent dans un ordre aléatoire. Modifier P2 pour les faire apparaître dans l'ordre croissant. Vous pouvez rajouter des sémaphores, toujours en respectant les contraintes C1–C3. Note: `score` est sans importance ici.

**Solution:** On rajoute des sémaphores `m[0]`, initialisé à 1, et `m[1]` à `m[N-1]`, tous à 0. Le thread avec id $i$ attend `m[i]` et libère `m[i+1]`.

```
void f (int id, int score) {
  wait(m[id]);
  printf("id=%d\n",id);
  if (id+1 < N) post(m[id+1]);
}
```

On veut maintenant simuler un tournoi de foot par élimination. Un match entre deux threads consiste à comparer leurs scores ; celui avec un score plus haut avance au prochain tour, le perdant se termine. Les matches se font un par un, le premier match est entre les threads avec identifiants 0 et 1, le deuxième entre 2 et 3, etc, ensuite en joue les matches du deuxième tour etc. Le gagnant de la finale émet un message à cet effet.

La Figure 2 illustre le tournoi, avec l'ordre des matches entre parentheses.

(c) Étant donné un identifiant entre 0 et 15, écrire une fonction qui énumère les matches que le thread correspondant doit disputer s'il gagne toujours. P.ex. pour `id=11`, l'affichage serait 5,10,13,14.

**Solution:** Il était parfaitement acceptable de spécialiser la solution pour $N = 16$ et répondre avec `id/2, 8+id/4, 12+id/8, 14`. Voici une solution un peu plus générale :

```
int nr = id, base = 0, plus = N/2;
while (base < N-1) {
  printf("%d\n",base+nr/2);
  base += plus; plus /= 2; nr /= 2;
}
```

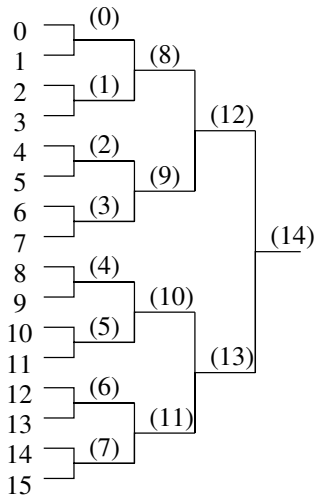


Figure 2: Arbre d'un tournoi

(d) Compléter P2 pour simuler le tournoi, en respectant C1–C3. Vous pouvez rajouter du code à `f` et des sémaphores globales et leurs initialisations dans `main`.

**Solution:** Il ne reste qu'à combiner les solutions de (a) à (c), avec quelques modifications subtiles :

- La solution de (b) sert pour ordonner les matches, mais on laisse deux threads entrer à chaque fois (donc `s[0]` sera initialisé à 2).
- Le protocôle de (a) étant asymmétrique, il faut que les joueurs sachent qui est le premier et qui est le second joueur d'un match. On prend simplement le dernier bit de `nr` dans (c). Ci-dessus, `first` et `second` deviennent `s[0]` et `s[1]` (zéro initialement).
- Synchronisation supplémentaire entre gagnant et perdant pour bien séparer les matchs.

```
void f (int id, int score) {
  int nr = id, base = 0, plus = N/2;
  while (1) {
    int match = base+nr/2, premier = nr & 1, other;

    wait(m[match]);  // attendre mon prochain match

    if (premier) x = score;  // obtention des scores
    post(s[premier]); wait(s[!premier]);
    if (!premier) { other = x; x = score; }
    post(s[premier]); wait(s[!premier]);
    if (premier) other = x;
```

```
      if (score < other) { post(s[premier]); break; }  // perdu

      wait(s[!premier]);
      if (match == N-2) { // vainqueur du tournoi ; break }

      post(m[match+1]);  // libérer le match suivant
      post(m[match+1]);
      base += plus; nr = nr/2; plus = plus/2;
    }
  }
```

# 3 Circuits logiques

Pour rappel, un $k$-multiplexeur est un circuit qui prend en entree $2^k$ signaux $x_0, \ldots x_{2^k-1}$ et un entier $s$ (codé sur $k$ bits). La sortie du multiplexeur est un signal $y = x_s$. (Le multiplexeur sélectionne donc parmi $2^k$ sources d'entrée, selon $s$.)

Un *$k$-codeur* est un circuit qui prend $2^k$ signaux en entrée $(x_0 \cdots x_{2^k-1})$ et fournit un vecteur de $k$ sorties $y_{k-1} \cdots y_0$ représentant un entier $y$. Dans un codeur, on suppose qu'exactement une des entrées a la valeur 1, disons $x_i$. Dans ce cas, la valeur binaire des sorties doit être $i$. Le comportement pour d'autres cas n'est pas spécifié.

(a) Construire un 1-codeur (trivial) et un 2-codeur.

**Solution:** Pour le 1-codeur, $y_0 = x_1$. Pour le 2-codeur, $y_0 = x_1 \vee x_3$ et $y_1 = x_2 \vee x_3$.

(b) Comment géneraliser la construction pour un $k$ quelconque ? Quelle est la taille et la profondeur de votre construction par rapport à $k$ ?

**Solution:** En général, $y_i$ est la disjonction de tous les $x_j$ tel que le $i$-ème bit dans la représentation binaire de $j$ est 1 (donc la moitié des signaux dans tous les cas). Un "ou" entre $2^{k-1}$ signaux peut se construire en profondeur $\mathcal{O}(k)$. La taille est alors $\mathcal{O}(k \cdot 2^{k-1}) = 2^{\mathcal{O}(k)}$.

Un *codeur de priorité* (CP) est comme un codeur, mais il gère le cas où plusieurs entrées ont la valeur 1. Dans ce cas, $y$ prend la valeur du plus grand indice $i$ tel que $x_i = 1$. Une autre sortie $z$ indique si au moins un des $x_i$ était 1. Si $z = 0$, la valeur de $y$ n'est pas spécifiée.

(c) Selon vous, à quoi peut servir un tel circuit dans un ordinateur ?

**Solution:** Au sein du processeur, chaque périphérique peut déclencher une interruption, et celles-ci peuvent intervenir au même moment. Un CP permet de choisir l'interruption la plus importante.

(d) Construisez un 2-CP (i.e., avec 4 signaux en entrée).

**Solution:**

- $z = x_0 \vee x_1 \vee x_2 \vee x_3$
- $y_0 = x_3 \vee (x_1 \wedge \neg x_2)$
- $y_1 = x_3 \vee x_2$

(e) Construire un $(k+1)$-CP à partir de deux $k$-CP.

**Solution:** (J'utilisera des superscripts pour distinguer entre plusieurs vecteurs.) Supposons qu'un $k$-CP sur $x_0, \ldots, x_{2^k-1}$ donne $z_0$ et un $k$-vecteur $y^0$, et qu'un deuxième $k$-CP sur $x_0, \ldots, x_{2^k-1}$ donne $z_1$ et $y^1$.

Nous devons produire $z = z_0 \vee z_1$ et un $(k+1)$-vector $y$. Pour le bit le plus significatif, $y_k = z_1$. Pour le reste, on choisit entre $y^0$ et $y^1$ à l'aide d'un multiplexeur qui sélectionne avec $z_1$.

(f) On vous donne un stock illimité de $k$-CP et $k$-multiplexeurs. Décrivez comment en construire un $2k$-CP.

**Solution:** Similaire à (e), cette fois on a $2^{2k}$ signaux en entrée, et on utilisera $2^k$ $k$-CP pour produire $z_0, \ldots, z_{2^k-1}$ et des $k$-vecteurs $y^0, \ldots, y^{2^k-1}$. Nous devons maintenant calculer $z$ et un $2k$-vecteur $y$, appellons sa partie 'haute' $y'$ et sa partie 'basse' $y''$.

Pour produire $z$ et $y'$, on utilisera encore un $k$-CP sur les $z_i$. Ensuite, $y''$ est sélectionné parmi les $y^i$ à l'aide des multiplexeurs avec $y'$ comme sélecteur.

# 4 Questions diverses

(a) Les appels système permettent la création des processus et des threads. Décrire au moins un scénario où préférerait la création d'un nouveau processus pour accomplir une tâche, et un scénario où la création d'un thread est préférable. (Justifiez votre réponse dans les deux cas, mais un paragraph court suffit.)

**Solution:** Les threads sont utiles quand ils partagent des données, p.ex. pour un calcul tel que le Mandelbrot traité en TP. Les processus sont utiles quand les tâches sont complètement séparés, p.ex. un shell qui lance un processus pour exécuter une commande. Ici, un isolement est même souhaité pour protéger le shell (p.ex. contre les débordements de mémoire, des signaux qui terminent le processus fils etc).

(b) POSIX propose des sémaphores et des spinlock. Comme dans (a), décrivez des situations où une solution ou l'autre est préférable.

**Solution:** Si la ressource (sémaphore/spinlock) est bloquée, un sémaphore met le thread en veille tandis qu'un sémaphore boucle. Sur un seul processeur (cœur) un spinlock gâchit simplement du temps. Sur un système avec plusieurs cœurs, les spinlock peuvent être plus efficaces si l'attente est très courte (voir aussi les exemples "exclusion mutuelle" du 12 décembre).

(c) Le dossier `/home/toto/` contient un dossier `foo/` et un fichier de texte `bar.txt`. La commande `stat /home/toto` indique trois liens durs sur ce dossier. D'où viennent ces liens ?

**Solution:**

- `toto` dans `/home`
- `.` dans `/home/toto`
- `..` dans `/home/toto/foo`