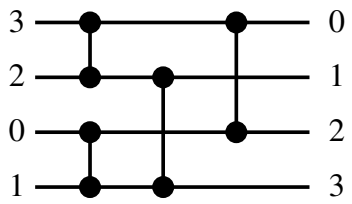
Examen d'Architecture et Système

17 janvier 2024

Durée : 2 heures.

1 Réseaux de tri

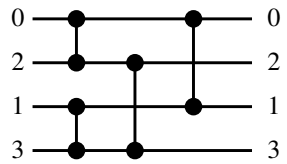
Un *réseau de tri* est un circuit logique d'une forme particuliére. Pour n entiers en entrée, il consiste de n fils horizontaux reliés par des *comparateurs* (représentés par des lignes verticales). Un comparateur prend deux entiers et émet le plus petit sur le fil "haut" et le plus grand sur le fil "bas". Ci-dessous un réseau de tri pour $n = 4$, avec les entrées à gauche et les sorties à droite.



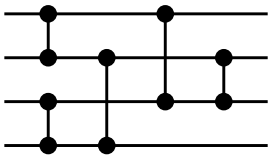
Un reséau de tri est *correct* s'il trie ses n entiers en ordre ascendant, pour toute permutation en entrée. Il est bien connu qu'un réseau est correct si et seulement s'il est correct pour toute séquence qui ne consiste que des 0 et 1.

(a) Le réseau de tri ci-dessus n'est pas correct. Trouver une liste d'entiers où le réseau ne fournit pas la réponse correcte. Dessiner un réseau correct pour $n = 4$.

Solution: $0, 2, 1, 3$ n'est pas trié correctement:



On voit facilement que le plus petit et le plus grand élément seront toujours à leurs places. Il suffit donc d'ajouter un comparateur entre les deux autres à la fin.



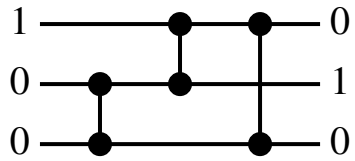
Pour les questions suivantes, prouver ou réfuter l'affirmation. Justifier ses réponses ; il ne suffit pas de répondre oui ou non.

(b) Dans tout réseau correct il y a au moins un comparateur entre toute paire de fils voisins.

Solution: Vrai. On considère un réseau pour n entiers où il n'y a pas de comparateur entre les lignes i et $i+1$. Pour la séquence $0^{i-1}101^{n-1-i}$, il n'y a donc aucun échange, du coup le réseau ne trie pas correctement.

(c) Tout réseau qui contient au moins un comparateur entre toute paire de fils est correct.

Solution: Faux, voir le réseau ci-dessous:

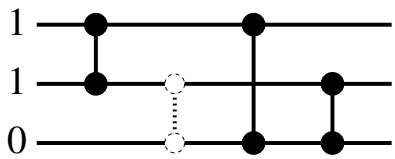


(d) Un réseau correct reste correct si on ajoute n'importe quel comparateur à la fin.

Solution: Vrai. Tous les entiers sont déjà triés avant d'arriver à la fin, du coup un comparateur additionnel ne change rien.

(e) Un réseau correct reste correct si on ajoute un comparateur n'importe où dans le réseau.

Solution: Faux. Le réseau ci-dessous est correct sans le comparateur en blanc, mais faux s'il est inclus (par exemple, pour 1,1,0).



2 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 ℓ 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 significatif 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 ?

Solution: (3+2 points) Les exposants autorisés pour $k = 4$ sont entre -6 et 7. 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. (Exprimer le résultat avec $z.s$, $z.m$, $z.e$.)

(Il n'est pas requis de traiter les cas de $x = 0$, $y = 0$ ou des cas spéciaux comme des débordements, infinité, "not a number".)

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 attendu.

3 Sémaphores

On considère le programme suivant qui consiste d'un thread principal et trois threads A, B, C. Le thread principal (dans `main`) lance les trois threads et attend leur fin. Certains endroits dans le programme sont indiqués par des commentaires (voir ci-dessous).

```
void* A(void* data) {
    // A1
    printf("u\n");
    // A2
    printf("v\n");
    // A3
}

void* B(void* data) {
    // B1
    printf("w\n");
    // B2
    printf("x\n");
    // B3
}

void* C(void* data) {
    // C1
    printf("y\n");
    // C2
    printf("z\n");
    // C3
}

int main() {
   pthread_t ta,tb,tc;
   // MAIN
   pthread_create(&ta,NULL,A,NULL);
   pthread_create(&tb,NULL,B,NULL);
   pthread_create(&tc,NULL,C,NULL);
   pthread_join(ta,NULL);
   pthread_join(tb,NULL);
   pthread_join(tc,NULL);
}
```

On considère les résultats possibles du programme, c'est à dire les différentes séquences des lettres `u,v,w,x,y,z`.

(a) Combien de séquences possibles y a-t-il dans le programme ci-dessus ? (inutile de les énumerer tous)

Solution: Deux façons d'arriver à la bonne réponse:

- Parmi les 6! permutations de `u,v,w,x,y,z`, il faut exclure la moitié où `v` intervient avant `u`, et encore la moitié où `x` intervient avant `w` etc, donc on arrive à $6!/2^3 = 90$.
- On utilise la formule pour "choisir k fois parmi n positions avec répétition": $\binom{n+k-1}{k}$. Commençons avec le mot `uv`; on y insère d'abord les lettres `w,x` sur trois positions (devant `u`, derrière `v` ou entre les deux) ce qui donne $\binom{3+2-1}{2} = 6$ choix. Puis on insère `y,z` dans le mot résultant (avec donc cinq positions possibles) ce qui donne $\binom{5+2+1}{2} = 15$ choix, et $6 \cdot 15 = 90$.

Dans le suivant, on souhaite limiter les séquences à un emsemble précis. Vous pouvez utiliser autant de sémaphores que vous voulez, avec un syntaxe simplifié : `init(s,n)`: créer

sémaphore `s` avec n créneaux ouverts initialement ; `wait(s)`: attendre un créneau ouvert dans `s`; `post(s)`: libérer un créneau dans `s`.

Spécifiez les opérations à ajouter pour limiter les résultats aux ensembles suivants, où le point (.) dénote la concaténation, $[uwx]$ l'ensemble des séquences des lettres u, w, x dans n'importe quel ordre, et $L_1 \# L_2$ n'importe quel entrelacement entre les mots dans L_1 et L_2. Dans les trois cas, donnez les opérations à ajouter dans les endroits `MAIN`, `A1`, etc.

(b) $\{uvwxyz\}$

(c) $[uwy].[vxz]$

(d) $(\{uv\}\#\{wx\}).\{yz\}$

Solution: Dans le suivant, `r,s,t` sont des sémaphores déclarées comme variables globales.

(b) Ayant terminé, `A` libère `B` et celui-ci libère `C`.

```
MAIN: init(s,0); init(t,0);
A3: post(s);
B1: wait(s);
B3: post(t);
C1: wait(t);
```

(c) Solution symétrique : On prend trois sémaphores, chacun associé avec un thread. Chaque thread, ayant fait sa première action, ouvre deux créneaux dans son sémaphore et attend les deux autres sémaphores.

```
MAIN: init(r,0); init(s,0); init(t,0);
A2: post(r); post(r); wait(s); wait(t);
B2: post(s); post(s); wait(r); wait(u);
C2: post(t); post(t); wait(r); wait(s);
```

Solution asymétrique : Un thread attend tous les autres, puis les libère:

```
MAIN: init(s,0); init(t,0);
A2: wait(s); wait(s); post(t); post(t);
B2: post(s); wait(t);
C2: post(s); wait(t);
```

(d) `C` doit attendre `A` et `B` qui eux peuvent s'entrelacer n'importe comment.

```
MAIN: init(s,0);
A3: post(s);
B3: post(s);
C1: wait(s); wait(s);
```

4 Entrée/sortie

On considère le programme dans Figure 1 où un pipe est crée entre père et fils. Le père est censé envoyer un lettre `a` par seconde au fils qui les imprime. On rappelle que `p[0]` est en mode lecture, `p[1]` en mode écriture et que `1` représente la sortie standard.

```
1: int main () {
2:   int p[2];
3:   pipe(p);
4:   close(p[0]);

5:   if (fork()) {
6:     while (1) {
7:       sleep(1);
8:       write(p[1],"a",1);
9:     }
10:  } else {
11:    char c = 'a';
12:    while (1) {
13:      read(p[0],&c,1);
14:      write(1,&c,1);
15:    }
16:  }
17: }
```

Figure 1: Programme avec pipe.

Lorsqu'on lance le programme, on voit que l'écran se remplit sans pause avec des `a` et que Ctrl+C n'arrête pas l'affichage incessant des `a`.

(a) Étant donné ce constat, le père est-il mort ou vivant ? Et le fils ?

Solution: Les observations permettent les conclusions suivantes :

- Seul le fils écrit sur la sortie standard (à priori, l'écran), du coup le fils doit être vivant.
- Ctrl+C fait que le shell envoie SIGINT au groupe des processus "du premier plan". Sauf utilisation de `setpgid`, un fils fait partie du même groupe que son père. Du coup, si Ctrl+C n'arrive pas à tuer le fils, c'est parce que le père n'est plus en premier plan car il est mort, voir (c).

(b) Pourquoi l'écran se remplit-il à une telle vitesse ?

Solution: La vitesse d'affichage nous indique que l'appel de `read` en ligne 13 n'est pas bloquant. En général, ça peut arriver en raison de "fin de ficher" ou d'une erreur. Il n'y a pas "fin de fichier" car le fils lui-même a `p[1]` ouvert. La raison est donc forcement une erreur. Et ceci pour une raison bête : le descripteur `p[0]` a été fermé dans la ligne 4, donc avant le `fork`. On essaye donc à lire dans un fichier fermé.

(c) Question bonus : Si un processus est mort, expliquez pourquoi.

Solution: Le père est mort car il reçoit un signal (non rattrapé) SIGPIPE qui intervient lorsqu'il écrit dans un pipe dont tous les débouchés ont été fermé (ce qui est le cas en raison de la ligne 4).

(d) Qu'est-ce qu'on peut changer pour que le programme se comporte comme prévu ?

Solution: Il suffit d'enlever la ligne 4 (ou l'échanger avec la ligne 5).

Tester la valeur renvoyée par `read` n'arrive pas à établir le comportement prévu, mais en général, c'est le bon réflexe.